Arquitectures de programari per a *big data*

PID_00286207

Remo Suppi Boldrito

Temps mínim de dedicació recomanat: 4 hores

Universitat Oberta de Catalunya

Remo Suppi Boldrito

Enginyer de Telecomunicacions. Doctor en Informàtica per la UAB. Professor del Departament d'Arquitectura de Computadors i Sistemes Operatius en la Universitat Autònoma de Barcelona.

L'encàrrec i la creació d'aquest recurs d'aprenentatge UOC han estat coordinats pel professor: Josep Jorba Esteve

Primera edició: febrer 2022

Av. Tibidabo, 39-43, 08035 Barcelona
Autoria: Remo Suppi Boldrito
Producció: FUOC

Índex

Introducció

***Big data* (BD)** és una de les tendències tecnològiques que impregna la nostra societat i que d'una manera significativa permet que les decisions basades en l'experiència donin lloc a les decisions basades en dades.

Per complementar aquest punt l'alumne haurà de llegir la part 1 del llibre *Big Data for Dummies*[1] per analitzar i contextualitzar què és el *big data*, quins són els fonaments de *big data* i les característiques, quins tipus de *big data* existeixen i on es troba el punt de transició entre els models de còmputs abans de *big data* i aquest nou concepte de còmput distribuït.

(1)Podeu trobar aquest llibre al següent enllaç: https://bit.ly/3vqCD9m

Les preguntes que l'alumne haurà de plantejar-se seran:

- Com obtenir coneixement de quantitats massives de dades?
- Què és i com funciona *data analytics*?
- En quina infraestructura es realitza aquest processament?
- Com es guarda –si és que es guarda– aquesta quantitat ingent de dades?
- Quines eines es necessiten per fer aquest processament?
- Es poden emmagatzemar i recuperar quantitats massives de dades en BD relacionals i de manera eficient?
- Quina és l'arquitectura de maquinari i programari per processar *big data*?
- Com es pot gestionar un gran volum de dades sense causar problemes en el centre de dades actual?

Objectius

Els objectius principals d'aquest mòdul són els següents:

1. Conèixer les diferents infraestructures de programari que s'apliquen al processament de *big data*.

2. Analitzar el concepte de MapReduce i la seva utilització en el processament de grans volums d'informació.

3. Realitzar proves de conceptes (exemples) amb les eines més comunes en entorns *big data* com Hadoop i Spark.

4. Analitzar distribucions específiques basades en Docker com a prova de concepte dels diferents components.

L'estudiant haurà de centrar la seva atenció en els següents conceptes fonamentals d'aquest mòdul:

1. Processament de *big data*

2. Algorisme de MapReduce

3. Hadoop + HDFS

4. Spark

5. Distribucions basades en Docker com a proves funcionals del processament de *big data*

1. Conceptes sobre el processament de *big data*

Com ja s'ha plantejat anteriorment, *big data* s'ha transformat en una expressió d'ús habitual, ja que, tant en centres d'R+D+i com en les empreses, indústries i institucions, cada vegada hi ha més quantitat de dades i calen noves tècniques i eines per processar-les i obtenir-ne informació. La revolució de les TIC aquest últim segle no només ha causat un diluvi de dades (*data deluge*), sinó també de la varietat, és a dir, estructurades, semiestructurades i no estructurades, i la velocitat (milions de dispositius i aplicacions que generen dades enormes cada segon).

Aquest enorme creixement de les dades ha arribat al límit de processament/emmagatzematge de les infraestructures de gestió de la informació existents, que ha obligat les empreses a invertir més en maquinari i en actualitzacions de bases de dades. Aquesta tendència, que aparentment soluciona el problema inicial, no és una solució real, ja que el conjunt de dades continua creixent i, per tant, s'entra en un cicle de més maquinari + emmagatzematge, i així contínuament. A més, la infraestructura tradicional no és eficient a causa dels alts costos, les limitacions d'escalabilitat (quan es tracta de petabytes) i la incompatibilitat dels sistemes de bases de dades relacionals amb dades no estructurades.

Per abordar l'enorme quantitat de dades de la web, el 2004 Google va presentar un model de programació anomenat **MapReduce** (MR) per fer, de manera paral·lela, tasques de cerca sobre els seus clústers de servidors. Per aconseguir aquest objectiu van publicar les seves idees sobre un sistema d'arxius distribuïts per tenir les dades a prop d'on s'haguessin de processar (2003) i després l'algorisme de processament MapReduce (el desembre de 2004).

Basats en aquestes idees, els investigadors Doug Cutting (desenvolupador de Lucene, un creador d'índexs de cerca, i Nutch, una aranya –*spider* o *crawler*– o motor de cerca de continguts per Internet) i Mike Cafarella (professor de la Universitat de Michigan i també implicat en el desenvolupament de Nutch) van crear un **sistema d'arxius i un entorn de processament** i van fer les implementacions per executar Nutch (el motor de cerca esmentat prèviament) sobre aquesta infraestructura.

El 2006, quan Doug Cutting estava treballant amb Yahoo, van fer millores en el programari, que havien provat amb Nutch, i van crear un entorn anomenat **Hadoop** com un projecte de codi obert a la Apache Software Foundation (versió 0.1.0). Des de llavors, Hadoop s'ha convertit en l'estàndard *de facto* per

emmagatzemar, processar i analitzar centenars de terabytes o petabytes de dades, i com un projecte 100 % de codi obert i disponible per a la comunitat que el vulgui utilitzar fins i tot amb finalitats comercials.

Hadoop permet el processament distribuït d'una quantitat de dades enorme en un conjunt de servidors de baix cost que emmagatzemen i processen les dades i que poden escalar sense límits (aquest enfocament es denomina escalat horitzontal).

Exemple: Hadoop a Yahoo!

D'acord amb les dades de 2015, els clústers de Hadoop a Yahoo! incloïen més de quaranta mil servidors i emmagatzemaven quaranta petabytes de dades d'aplicacions.

Després del llançament de l'entorn Hadoop, *big data analythics* es va transformar en el gran repte tecnològic que permetia a les empreses o institucions fer el salt estratègic de la retrospectiva a la prospectiva extraient valor dels grans conjunts de dades. Com en qualsevol idea nova, existeix una comunitat escèptica que manifesta el seu rebuig al «*big data* i al que comporta», ja que sosté que «durant dècades, les empreses han pres decisions comercials basades en dades transaccionals emmagatzemades en bases de dades relacionals (DBRMS) i ara no és diferent», o que són «tècniques i eines (DBRMS) d'eficiència demostrada i que no és necessari canviar».

En aquestes afirmacions es passa per alt que existeix un enorme potencial de dades no tradicionals i no estructurades, com els registres web, xarxes socials, correu electrònic, dades de sensors, imatges, àudio i vídeo, que poden contenir informació útil o valuosa. Aquestes opinions tampoc tenen en compte el que passa al món real, on la **presa de decisions basada en dades** ha crescut exponencialment en els últims anys amb una reducció significativa de la presa de decisions basades en les evidències o l'instint, la intuïció o l'experiència.

1.1. Problemes a resoldre en *big data*

Com ja s'ha definit, en una arquitectura per al processament de *big data*, com en qualsevol entorn orientat a dades, existeixen quatre nivells que interactuen: **recol·lecció de dades**, **emmagatzematge**, **processament** i **visualització**, més un de més global d'**administració**.

Aquesta arquitectura és l'habitual que es pot trobar en qualsevol entorn orientat a dades (per exemple, *datamining*, *business intelligence* o *deep learning*), però quan s'associa a aquests entorns el concepte de *big data*, cadascun d'aquests nivells ha evolucionat o s'han generat noves eines, algorismes o entorns per adequar-se al *big data*.

En la **recol·lecció** de les dades orientades a *big data* han sorgit una gran quantitat d'eines orientades a aquest efecte (*data ingestion*), que permeten el **processament seqüencial habitual de dades emmagatzemades** (*batch* o lots) per

obtenir el següent tram o seqüència (*chunk*) de dades des de l'últim llegit; en tot cas, en la seva majoria, i a causa del volum que suposen, les eines han evolucionat per recol·lectar eficientment **fluxos de dades** (*streams*) en temps real.

En l'**emmagatzematge** també han sorgit grans canvis per adaptar-se a les característiques de les dades; això s'ha reflectit en una gran quantitat de desenvolupaments de motors de bases de dades i sistemes d'arxius distribuïts per adequar-se a la realitat de processament i, així, disposar d'escalabilitat, fiabilitat i proximitat de les dades per al seu tractament.

Exemple: motors de bases de dades

NoSQL, documents, columnes, clau/valor, grafs.

Exemple: sistemes d'arxius distribuïts

Apache HDFS, BeeGFS, DiscoDDFS, Google GFS, BaiduFS, GlusterFS, QuantcastFS, CephFS, GridGain-in-memoryFS, LustreFS.

Probablement, una de les principals dificultats de *big data* és en el **processament**, ja que els algorismes i les tècniques s'han hagut de transformar per mantenir l'escalabilitat i les prestacions sobre un conjunt de dades i una plataforma de còmput distribuïda. Per això s'han desenvolupat nous paradigmes de processament de la informació que han caracteritzat tot l'entorn (i en cadascuna de les capes per adequar-les a les seves necessitats). Aquests nous mètodes i algorismes es coneixen com a **MapReduce** (MR) o **Massive Paralell Processing** (MPP).

MapReduce és un paradigma de programació que rep el seu nom per les dues funcions que el formen (*map* i *reduce*) i de l'entorn (*open source*) Apache Hadoop en el qual s'implementa.

Vegeu també

Parlem breument sobre MapReduce en el mòdul «Introducció a les infraestructures per a *big data*» d'aquesta assignatura.

Aquest paradigma, si bé no és la solució de tots els problemes, treballa amb grans conjunts de dades (petabytes) i s'executa sobre **sistemes d'arxius distribuïts** (HDFS) i vinculats a eines com HBase, Hive, Impala o Cassandra (bases de dades).

Aquest paradigma és apte per processar aquelles dades que poden treballar sobre tuples del tipus [clau, valor] i on la funció **map()** processa en paral·lel les tuples d'un domini i genera una llista de parells en un domini diferent, que s'agruparan sota la mateixa clau amb un esquema de processament *master-worker* en forma d'arbre.

Posteriorment, la funció **reduce()** s'aplica de manera paral·lela per a cada grup i genera una col·lecció de valors per a cada domini.

Per veure un altre enfocament a una implantació de MR s'analitzarà com seria el codi Python d'aquestes funcions per comptar paraules. Per fer-ho, es desenvoluparà la funció **mapper.py**, que llegirà les dades de l'entrada estàndard (és a dir, d'on li vingui l'entrada per defecte i que s'identifica amb STDIN), els dividirà en paraules i generarà una llista de línies que mapegen paraules en sortida estàndard (on estigui assignada i que s'identifica per STDOUT). Aquest

Vegeu també

Sobre el codi Python podeu consultar les següents referències:

Referència 1.
Referència 2.

codi no calcularà una suma (intermèdia) de les ocurrències d'una paraula i en el seu lloc es generaran tuples equivalents de sortida, encara que una paraula específica pugui aparèixer diverses vegades en l'entrada.

La funció de **comptar** es deixarà per al següent pas (**reduce**), que farà el recompte de la suma final. Per descomptat, és una manera d'implementar i es pot canviar el comportament com es desitgi. Per fer les proves funcionals s'ha de canviar l'atribut d'execució amb `chmod +x mapper.py`.

```
#!/usr/bin/env python3
"""mapper.py"""
import sys
# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # generate tuples
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step.
        print ('%s\t%s' % (word, 1))
```

Si s'executa amb la frase plantejada inicialment:

```
echo "La ciència és una equació diferencial. La religió és una condició de frontera" |./mapper.py
```

El resultat serà:

La	1
ciència	1
és	1
una	1
equació	1
diferencial.	1
La	1
religió	1
és	1
una	1
condició	1
de	1
frontera	1

El següent codi que s'ha d'escriure serà `reducer.py`, que llegirà els resultats de mapper.py des de l'entrada estàndard (STDIN) i sumarà les ocurrències per generar el compte final a la sortida estàndard (STDOUT). Cal recordar també que s'ha d'executar `chmod +x reducer.py` per activar els permisos d'execució.

```
#!/usr/bin/env python3
"""reducer.py"""
from operator import itemgetter
import sys
current_word = None
current_count = 0
word = None
# input comes from STDIN
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # parse the input we got from mapper.py
    word, count = line.split('\t', 1)
    # convert count (currently a string) to int
    try:
        count = int(count)
    except ValueError:
        # count was not a number, so silently
        # ignore/discard this line
        continue
    # this IF-switch only works because Hadoop sorts map
    # output by key (here: word) before it is passed to the
    # reducer
    if current_word == word:
        current_count += count
    else:
        if current_word:
            # write result to STDOUT
            print ('%s\t%s' % (current_word, current_count))
        current_count = count
        current_word = word
# do not forget to output the last word if needed!
if current_word == word:
    print ('%s\t%s' % (current_word, current_count))
```

Ara es pot provar la funcionalitat executant:

```
echo "La ciència és una equació diferencial. La religió és una condició de frontera" |
./mapper.py | sort -k1,1 |./reducer.py
```

Com es pot veure, se simula el que després es veurà que fa l'entorn d'execució (Hadoop), que en el nostre cas és ordenar les tuples per paraules (*key*). La sortida serà:

ciència	1
condició	1
de	1
diferencial.	1
equació	1
és	2
frontera	1
La	2
religió	1
una	2

Si es prova amb un text extens, per exemple *Alice's Adventures in Wonderland* (3.731 línies) del projecte Gutenberg (OpenBook), on s'ha ordenat la sortida per veure les paraules més repetides (només es mostren les primeres línies):

```
cat Alice.txt |./mapper.py | sort -k1,1 |./reducer.py | sort -k2 -rn
```

Figura 1. *Alice's Adventures in Wonderland*

El **paradigma MPP** ofereix una solució tradicional adaptada a *big data* basada en dividir els grans conjunts de dades en seccions (*slices*), que seran més fàcils de gestionar; cadascun d'ells serà assignat a un element de còmput per al seu processament. El dispositiu de còmput els processarà i, quan acabi, el sistema combinarà els resultats parcials per donar un resultat final (equivalent a una seqüència *fork-join* en un model *master-workers*). Atès que els elements de còmput (processadors) treballaran sobre el seu segment de dades assignat de la BD i es comunicaran mitjançant missatges quan generin els resultats, no hi ha interacció entre ells; això rep el nom de «feblement acoblats» (altres autors ho denominen *shared nothing*).

Entre les característiques principals d'aquests sistemes es poden enumerar les seves possibilitats de sintonització i escalat il·limitat (en principi al nombre de nodes disponible), amb el consegüent increment del seu rendiment (pràcticament en forma lineal) i sense colls d'ampolla.

Existeix gran quantitat de bibliografia sobre la comparació MR i MPP; no obstant això, moltes vegades la solució no prové d'un o de l'altre, i atès que es complementen bé quant a les prestacions i els tipus de dades que obtenen millors rendiments, és habitual que s'utilitzin arquitectures mixtes on generalment s'aplica primer MR sobre dades no estructurades i, posteriorment, MPP per processar i visualitzar les dades estructurades generades pel primer.

Lectura recomanada

A. Grishchenko. «Hadoop vs MPP». *Distributed Systems Architecture*. https://0x0fff.com/hadoop-vs-mpp/

2. L'entorn Hadoop

El projecte Apache™ Hadoop® és una plataforma *open source* per al còmput distribuït, de confiança i escalable i utilitzat per una gran quantitat d'empreses i institucions, que implementa l'algorisme de MapReduce.

> És un entorn per al processament distribuït de grans conjunts de dades per mitjà de clústers de còmput; utilitza models de programació senzills i té la possibilitat d'escalar des de servidors individuals a milers de processadors.

Bàsicament, Hadoop (V2.x o V3.x) està format per quatre mòduls:

1) HadoopYARN: marc per a la programació de tasques i gestió de recursos del clúster.

2) Hadoop MapReduce: sistema basat en YARN per al processament paral·lel de grans conjunts de dades.

3) Hadoop Distributed FileSystem (HDFS): sistema d'arxius distribuït que proporciona accés d'alt rendiment a les dades de l'aplicació.

4) Hadoop Common: les utilitats comunes que suporten els altres mòduls de Hadoop.

No obstant això, Hadoop es pot relacionar, integrar o ampliar de manera fàcil i ràpida amb els següents projectes (llistem alguns dels més habituals o referenciats):

- **Ambari**: eina web per a l'aprovisionament, la gestió i el monitoratge de clústers Hadoop amb HDFS, MapReduce, Hive, HCatalog, HBase, ZooKeeper, Oozie, Pig i Sqoop.
- **Avro**: serialització de dades.
- **Cassandra**: base de dades escalable sense punts de fallada únics.
- **Chukwa**: recopilació de dades en sistemes distribuïts.
- **Drill**: *SQL query engine* de baixa latència que suporta aplicacions distribuïdes per a l'anàlisi interactiva de *datasets*.

- **Flume**: sistema distribuït per recollir, agregar i moure grans quantitats de dades des de diferents fonts i també de *datastores* centralitzats.

- **HBase**: base de dades orientada a columnes distribuïda i escalable.

- **Hive**: magatzem de dades que proporciona resum de dades i consultes *ad hoc*.

- **Mahout**: entorn de *machine learning* i *data mining*.

- **Pig**: llenguatge de flux de dades d'alt nivell.

- **Spark**: motor de càlcul en memòria i especialitzat a processar dades de *streams* (fluxos de dades).

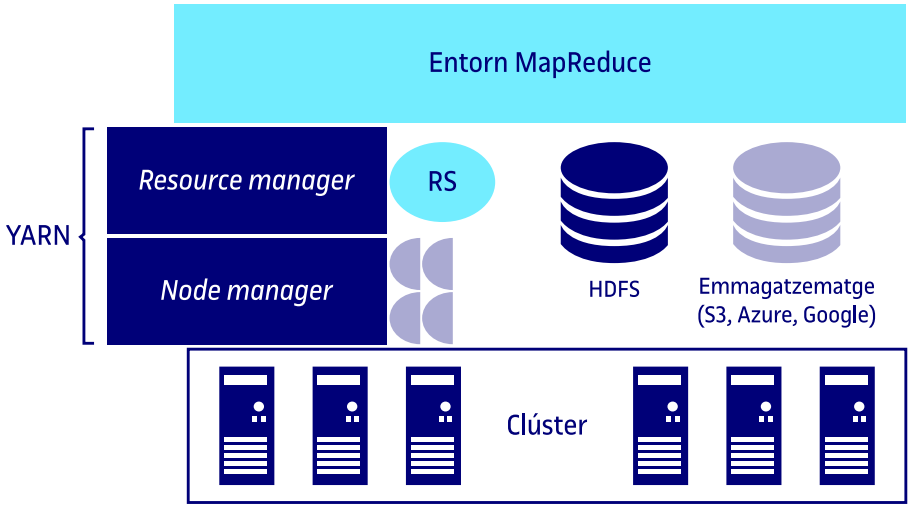
- **Sqoop**: eina per transferir dades entre Hadoop i bases de dades estructurades, com per exemple bases de dades relacionals.

- **Tez**: entorn de programació de flux de dades generalitzat per a grafs dirigits.

- **ZooKeeper**: servei de coordinació d'alt rendiment per a aplicacions distribuïdes.

La relació entre els mòduls de l'arquitectura Hadoop amb MapReduce (per a més detalls consulteu la documentació), queda representada a la figura següent:

Figura 2. Arquitectura Hadoop amb MapReduce



On:

1) Entorn MapReduce: capa de programari que permet l'execució d'aplicacions sota aquest paradigma.

2) YARN (*Yet Another Resource Negotiator*): és la part responsable de gestionar els recursos (CPU, memòria, etc.) que utilitzaran les aplicacions; es compon de dos grans mòduls:

a) *Resource manager*, RM (un per clúster), que actua com a supervisor i coneix on estan situats els *workers* i de quants recursos disposa. Inclou una sèrie de serveis, però el més important és el ***resource scheduler*** (RS), que decideix com i a qui assignar-los.

b) *Node manager*, NM (>1 per clúster), és el *worker* de la infraestructura i manté informat el *resource manager* sobre el seu estat; gestiona la memòria i *vcores* que poden ser assignats sota la gestió de RS, que decidirà com s'utilitza aquesta capacitat per mitjà de fraccions de la capacitat del NM anomenades *containers*.

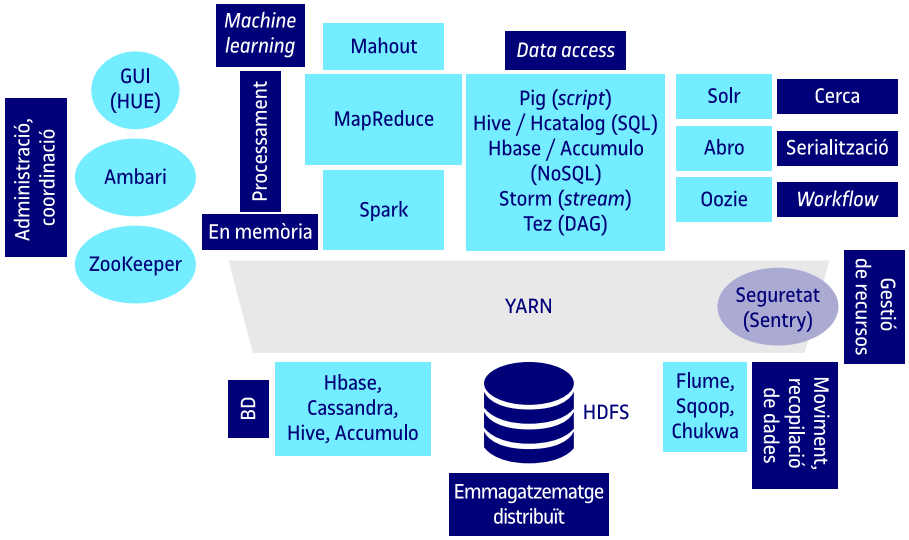
3) *HDFS Federation*: és la part de l'entorn responsable de proveir emmagatzematge permanent, fiable i distribuït; és utilitzat per emmagatzemar les entrades i sortides de l'aplicació (no les intermèdies). Es pot complementar amb altres opcions d'emmagatzematge *cloud* (per exemple, S3, Google StorageObjects, etc.).

4) Clúster: és el conjunt de nodes de la infraestructura. És important destacar que la infraestructura YARN i l'HDFS estan totalment desacoblats i a sobre d'ells es podrien executar altres entorns. En el cicle de vida de l'aplicació, gestionat per YARN, intervenen tres mòduls: *job submitter* (client), *resource manager* (supervisor o màster) i *node manager* (*worker*), per la qual cosa el *workflow* d'una aplicació seria:

a) el client envia l'aplicació a l'RM,
b) aquest sobre la base de la informació de RS,
c) assigna el contenidor,
d) que a més contacta amb el *node manager*,
e) que al seu torn llança el contenidor
f) i aquest executa l'aplicació.

A la figura següent es pot veure un entorn Hadoop amb els seus actors més importants i el seu rol (un dels possibles) dins de la plataforma.

Figura 3. Entorn Hadoop



2.1. Casos d'ús amb Hadoop i Linux

Com a primera prova (després es veuran contenidors i altres distribucions i mètodes preinstal·lats) per a la instal·lació de Hadoop es recomana seguir les indicacions de la pàgina dels desenvolupadors i instal·lar el que es diu *single cluster* (és a dir, tots els components funcionen sobre un sistema que pot ser una màquina virtual). Per fer-ho, primer s'ha de verificar i instal·lar programari necessari (ssh, java...), descarregar la distribució de Hadoop i configurar el que es denomina *pseudo-distributed operation,* ja que tots els components estaran en el mateix node. Amb aquesta configuració ja es podran provar tots els exemples que hi ha a continuació, però si es desitja configurar un clúster complet es poden seguir les indicacions de Hadoop Cluster Setup, que és la recomanada per a entorns de processament en producció.

Per a aquesta prova de concepte (analitzar la funcionalitat de Hadoop) s'utilitzarà el codi MR per comptar paraules, dades del temps i dades (*dades obertes*) de receptes mèdiques utilitzant la instal·lació indicada prèviament preparada en una MV. El llenguatge habitual de treball en Hadoop és Java, i es provaran alguns exemples en aquest llenguatge, però es dedicarà temps a treballar amb Python per processar les dades amb un llenguatge que és utilitzat en moltes disciplines.

El «truc» per processar codi Python, com s'ha comentat anteriorment, és utilitzar la Hadoop Streaming API, que permetrà passar dades entre *mapper.py* i *reducer.py* per mitjà de STDIN (entrada estàndard) i STDOUT (sortida estàndard). Per fer-ho, només s'haurà d'utilitzar la llibreria *sys.stdin* de Python per llegir les dades d'entrada i generar les dades de sortida amb la llibreria *sys.stdout* i Hadoop Streaming s'encarregarà de tota la resta.

2.1.1. Exemple de WordCount (Java)

Per a aquest exemple s'utilitzarà el codi Java que es pot trobar a MapReduce Tutorial, que utilitza les interfícies Java de Hadoop + HDFS + Yarn *framework* per comptar paraules. Per simplificar l'execució, aquest exemple es desenvoluparà sobre una única MV, però amb tot l'entorn complet de Hadoop i el seu ecosistema, el qual es pot estendre ràpidament a un clúster (fins i tot d'MV) per fer un processament amb més prestacions, que per a aquesta prova de concepte no és necessari.

Sobre la MV es trobarà un usuari **hadoop** i al *home* d'aquest usuari un directori ***wordcount*** amb un arxiu de codi (WordCount.java) i dos arxius de dades (alice.txt i war-and-peace.txt). Al directori $HOME/hadoop (/home/hadoop/hadoop) hi ha instal·lat i configurat tot el programari de Hadoop i a $HOME/.bashrc totes les variables d'entorn per executar Java i Hadoop.

1) Per començar s'han d'inicialitzar els processos que suporten tot aquest còmput (anomenats *daemons*) executant en un terminal:

```
start-dfs.sh
```

```
start-yarn.sh
```

Es pot observar obrint un navegador l'estat del sistema de processament posant com URL: http://127.0.0.1:50070.

Els treballs (*jobs*) enviats a processar es poden consultar a l'URL: http://127.0.0.1:8088/clúster.

2) Per poder executar el codi Java s'ha de compilar (transformar el codi del programa en codi executable pel processador) amb:

```
hadoop com.sun.tools.javac.Main WordCount.java
```

Això generarà un conjunt d'arxius que s'han de compactar en un únic arxiu anomenat, en aquest exemple, *wc.jar*:

```
jar cf wc.jar WordCount*.class
```

3) A continuació s'ha de pujar l'arxiu de dades a processar (alice.txt) al sistema d'arxius distribuït que «acostarà» les dades a cada node de dades de l'entorn d'execució:

```
hdfs dfs -put alice.txt input/alice.txt
```

4) Finalment es podrà executar amb Hadoop:

```
hadoop jar wc.jar WordCount input/alice.txt output
```

5) I es podrà analitzar la sortida amb:

```
hdfs dfs -cat /output/*
```

Després es podrà repetir el mateix procés amb un altre llibre de més gran (war-and-peace.txt) i només caldrà pujar-lo a l'HDFS i tornar a executar canviant només l'arxiu d'entrada.

6) Ordres útils:

- Com inicialitzar els directoris de l'HDFS necessaris per pujar les dades a processar per Hadoop: `hadoop namenode -format`
- Com iniciar els *Hadoop daemons*: `start-dfs.sh start-yarn.sh`
- Com crear un nou directori a l'HDFS:
  `hdfs dfs -mkdir /hduser/input`
- Com obtenir la llista d'ordres per treballar amb l'HDFS: `hdfs dfs`
- Com copiar un arxiu al meu directori local a l'HDFS:
  `hdfs dfs -put /home/hadoop/file.txt input/file.txt`
- Com llistar els arxius a l'HDFS: `hdfs dfs -ls`
- Com executar un programa:
  `hadoop jar file.jar MainClass input/input.txt output`
- Com visualitzar les dades de sortida generats per un programa a l'HDFS:
  `hdfs dfs -cat output/*`
- Pàgina d'administració de Hadoop: `http://localhost:50070/`
- Pàgina d'administració de Yarn: `http://127.0.0.1:8088/clúster`
- Com acabar una sessió i apagar tot l'entorn Hadoop:
  `/home/hadoop/Hadoop/sbin/stop-all.sh`

2.1.2. Exemple MapReduce Weather

Moltes aplicacions basades en dades generen grans *datasets* que poden ser descarregats i en aquest cas s'utilitzaran les del National Centers for Environmental Information (NCEI) en un exemple del qual es parla al llibre *Hadoop: the Definitive Guide* el codi d'exemple del qual es troba a https://github.com/tomw-

hite/hadoop-book (en aquest cas s'utilitzarà el codi Java de l'apartat 2). Aquest *dataset* està organitzat per anys i diferents arxius de cada estació meteorològica on cada registre té una sèrie d'informació indicada al document «Local Climatological Data (LCD) Dataset Documentation».

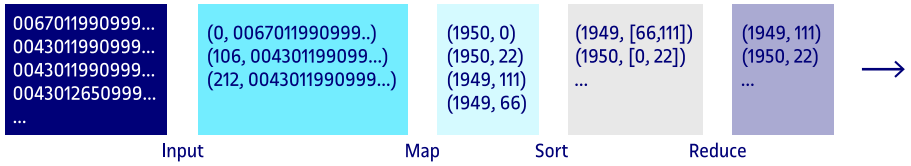
Per exemple, les dades del 2020 ocupen 4,1 GB (www.ncei.noaa.gov/data/local-climatological-data/archive/). L'inventari de les estacions, la seva ubicació i el mapa es troben a www.ncei.noaa.gov/products/land-based-station/station-histories.

Lectura recomanada

T. White (2015). *Hadoop: The Definitive Guide: Storage and Analysis at Internet Scale* (4a edició, cap. II). Sebastopol: O'Reilly Media.

El problema a resoldre és trobar la temperatura màxima de cada any en el *dataset* d'una estació, per la qual cosa haurem de centrar l'estudi en la data i la temperatura de cada registre en un esquema similar al següent:

Figura 4. Esquema de registre



De les línies que tenen un format com:

(0, 0067011990999991950051507004...9999999N9+00001+99999999999...)

(106, 0043011990999991950051512004... 9999999N9+ 00221+ 99999999999...)

La funció *map* només es queda amb l'any i la temperatura i genera com a resultat parells (1950, 0)(1950, 2)(1950, -11)(1949, 111)(1949,78)... i abans d'enviar a la funció *reduce* s'agrupen i ordenen com (1949, [111,78]) (1950, [0,22,-11].

Amb això la funció *reduce* iterarà sobre la llista de cada any buscant la temperatura més alta i generarà com a resultat (1949, 111)(1950, 22) (recordeu que les temperatures estan en ºF i per això hi ha valors com 111F = 43 °C).

1) Per compilar:

```
export CL=`hadoop classpath`
javac -classpath $CL -d maxTemperature_classes MaxTemperature.java MaxTemperatureMapper.java
MaxTemperatureReducer.java
```

2) Crear l'arxiu únic (*jar*):

```
jar -cvf maxTemperature.jar -C maxTemperature_classes.
```

3) Posar l'arxiu de dades (data.txt) en l'HDFS:

```
hdfs dfs -put data.txt /input/data.txt
```

Es poden baixar les dades d'interès des dels URL indicats anteriorment o des de Github (on es troben els exemples de *Hadoop: the Definitive Guide*).

4) Per executar:

```
hadoop jar maxTemperature.jar MaxTemperature input/data.txt output
```

5) Consultar els resultats:

```
hdfs dfs -cat output/*
```

6) Utilitzant els URL anteriors i el procediment indicat, es pot filtrar i processar, per exemple, la temperatura màxima de l'estació 010100 que conté 715175 registres, i que resulta un exercici interessant per processar i analitzar, donada la magnitud de les dades.

2.1.3. Exemple de receptes mèdiques (Python)

Per a aquest cas d'ús s'analitzaran dades de l'arxiu de receptes de Gencat mitjançant un entorn Hadoop (i que les eines habituals com ara fulls de càlcul no poden processar per la seva grandària).

Per complir amb aquest objectiu es disposarà de l'arxiu de dades baixat de la plataforma abans esmentada amb totes les receptes venudes a Catalunya pel CatSalut, i que s'han baixat de les dades originals en format CSV, que es dirà *rece.csv*. També es desenvoluparà la funció *mapper-rec.py* i *reduce-rec.py* i un programa (*shell-script*) anomenat *to-execute* amb la sintaxi per cridar a Hadoop amb codi Python.

Un dels principals problemes a resoldre és que, dins del codi Python, haurem de tenir en compte que els camps són, per exemple:

> 2021,02,61,LLEIDA,0-1 any,Dona,A,TRACTE ALIMENTARI I METABOLISME,A01, PREPARATS ESTOMATOLÒGICS,A01A,PREPARATS ESTOMATOLÒGICS, A01AB,Antiinfecciosos i antisèptics per al tractament oral-local,6,6,16.98,9.91

On es poden veure els camps separats per «,», però existeixen registres com per exemple el que està marcat:

> 2021,02,61,LLEIDA,0-1 any,Dona,P,"**PRODUCTES ANTIPARASITARIS, INSECTICIDES I REPEL·LENTS**",P01,ANTIPROTOZOARIS,P01A,AGENTS CONTRA L'AMEBIASI I ALTRES MALALTIES PER PROTOZOARIS,P01AB,Derivats del nitroimidazol,1,1,2.17,1.30

On es pot observar que hi ha el camp «PRODUCTES ANTIPARASITARIS, INSECTICIDES I REPEL·LENTS» que inclou una «,» dins del camp i que dificultarà el processament per part de Python, ja que no serà tan fàcil separar per camps. En aquest cas es podrà fer sense problemes, ja que l'*string* està entre «...».

El codi comentat del *mapper-rec.py* és:

```
#!/usr/bin/env python3

import sys # les entrades vindran de la STDIN (standard input)
for line in sys.stdin:

    line = line.strip()  # treure espais en blanc al començament i al final de la línia

    if line[0] == "a":  # treure la capçalera que comença per any
        continue

    words = line.split(",")        # separar els camps per ‘,’

    wordFinal = []  # array de camps finals (inclosos els separats per , dins
    union = ""   # string temporal per anar ajuntant les parts dels camps
    for word in words:  # per a tots els camps
        if word[0]=='"':  # si el camp comença per "?
            union = word.replace(word[0], "")  # s'agrega a unió i es treu la " i continua
            continue
        if (union != "") and word[len(word)-1] !='"': # si ja tenen les parts i no s'ha arribat
                                                      # al final s'afegeix
            union = union + word
       elif (union != "") and word[len(word)-1] =='"': # si ja es tenen les parts i s'ha arribat al
                                                      # final, es treu i s'afegeix al wordFinal
            word = word.replace('"', "")
            wordFinal.append(union + word)
            union = ""     # s'esborra union per al pròxim
        else:
            wordFinal.append(word)    # en cas contrari s'afegeix
    #print ("line", line)
    #print ("-->", wordFinal)
    #print ("--->", wordFinal[13])
    wordsNoWhite = wordFinal[13].replace(" ", "_")  #canvia els espais en blanc per _
    print ('%s\t%s' % (wordsNoWhite, 1))   # es generen les tuples amb el camp 13.
          # descripció d'ATC4
```

El codi del *reducer-rec.py* és exactament igual a l'esmentat anteriorment:

```
#!/usr/bin/env python3
```

```
from operator import itemgetter
import sys
current_word = None
current_count = 0
word = None

for line in sys.stdin:  # input comes from STDIN

    line = line.strip()  # remove leading and trailing whitespace
        # parse the input we got from mapper.py
    word, count = line.split('\t', 1)
        # convert count (currently a string) to int
    try:
        count = int(count)
    except ValueError:
            # count was not a number, so silently
            # ignore/discard this line
        continue
    # this IF-switch only works because Hadoop sorts map
    # output by key (here: word) before it is passed to the
    # reducer
    #print (word,count)
    if current_word == word:
        current_count += count
    else:
        if current_word:
            # write result to STDOUT
            print ('%s\t%s' % (current_word, current_count))
        current_count = count
        current_word = word
# do not forget to output the last word if needed!
if current_word == word:
    print ('%s\t%s' % (current_word, current_count))
```

Per executar aquest codi primer es pot provar simulant l'entorn Hadoop simplement canviant els permisos al codi i executant:

```
chmod +x mapper-rec.py
chmod +x reducer-rec.py
cat rece.csv |./mapper-rec.py | sort -k1,1 |./reducer-rec.py
```

Que donarà una llista com:

Acid_aminosalicilic_i_agents_similars 14461

Acid_ascorbic_(vitamina_C)_monofarmac 924

Acid_folic_i_derivats 17652

Acid_salicilic_i_derivats 6609

Acids_biliars_i_derivats 13294

...

Per executar en l'entorn Hadoop, s'han de seguir els següents passos:

1) Posar en marxa l'entorn: `start-dfs.sh` i a continuació executar l'ordre `start-yarn.sh`.

2) Pujar l'arxiu rece.csv a l'HDFS al directori *input*:

```
hdfs dfs -put rece.cvs input
```

3) Es pot verificar que està tot bé amb `hdfs dfs -ls input/*`.

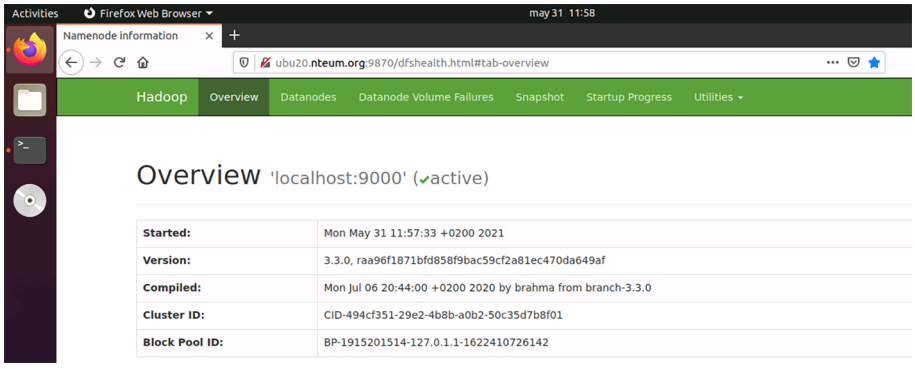
4) Finalment executar:

```
hadoop jar /home/hadoop/etc/hadoop/tools/lib/hadoop-streaming-3.3.0.jar -mapper "mapper-rec.py"
-file./mapper-rec.py -reducer "reducer-rec.py" -file./reducer-rec.py -input "input/rece.csv"
-output output
```

Com que la sintaxi és complexa i es poden cometre errors, es pot fer un *script,* que és una ordre en un arxiu que simplifica la seva execució, anomenat, per exemple, `to-execute`. Se li han d'agregar els permisos d'execució amb l'ordre `chmod +x to-execute` i després es podrà executar com: `./to-execute`.

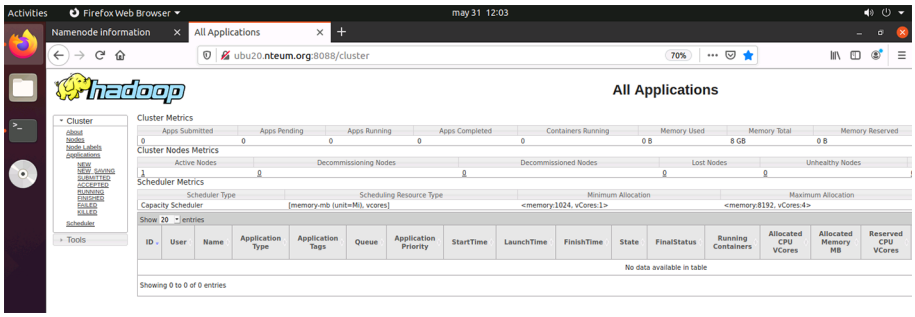
5) L'estat d'HDFS es pot observar obrint un navegador qualsevol i posant com URL: localhost:9870 o el domini de la màquina (en aquest cas ubu20.nteum.org), que mostrarà una sortida com la següent:

Figura 5. Estat d'HDFS



També es pot veure la pàgina de Hadoop en aplicacions, indicant com URL (i en aquest cas ha de ser el *nom.domini* de la màquina): **ubu20.nteum.org:8088**.

Figura 6. Aplicacions de Hadoop



6) Per veure els resultats: `hdfs dfs -cat output/*`.

7) Per eliminar els resultats (és necessari per fer altres proves sobre el mateix directori de sortida, ja que per protecció no deixa sobreescriure els directoris de sortida): `hdfs dfs -rm output/*` i a continuació el directori `hdfs dfs -rmdir output`.

8) Finalment, quan s'hagin acabat les proves s'haurà de parar l'entorn amb: `stop-dfs.sh` i a continuació `stop-yarn.sh`.

3. L'entorn Spark

Spark és un motor de processament optimitzat de dades en memòria que pot executar operacions amb dades de Hadoop (en HDFS) i amb un millor rendiment que MapReduce.

També pot processar dades en altres entorns (com ara clústers) i, en minimitzar les lectures i escriptures de disc, aquest ofereix un rendiment molt alt a col·leccions de dades dinàmiques i complexes i brinda funcionalitat interactiva com a suport de consultes específiques (*ad hoc*) d'una manera escalable. Spark estén el paradigma de processament estàtic de dades emmagatzemades i per seqüències (lots o *batch*), que és el que utilitza MapReduce, cap a aplicacions dinàmiques i en temps real o sobre fluxos de dades. Spark suporta programes en diferents llenguatges de programació, que inclouen Java, Scala, Python i R i permet interactuar amb dades incloses en bases de dades SQL.

Si bé MapReduce ha estat durant anys la metodologia de desenvolupament de programari per al processament distribuït per lots (*batch*) de *big data*, aquest tipus de processament no és la resposta per a totes les situacions computacionals. Amb l'interès de buscar millors prestacions a MapReduce, els investigadors van començar a elaborar noves propostes i dues de les més significatives han estat **Apache Drill** i **Spark**.

Drill és un motor de llenguatge de consulta estructurat (SQL) d'alt rendiment destinat a proporcionar funcions d'exploració de dades, i Spark va ser dissenyat per ser un motor de processament de propòsit general per a tasques interactives, per lots i de flux que eren capaços d'aprofitar els mateixos tipus de recursos de processament distribuïts que fins ara havien impulsat les iniciatives MapReduce.

No han estat els únics intents de millorar les prestacions sobre *big data,* ja que també es poden esmentar els següents:

- **Apache Pig** (un llenguatge MapReduce especialitzat d'alt nivell) desenvolupat per Yahoo! per facilitar l'accés a les dades emmagatzemades en Hadoop.

- **Apache Hive**, que va aportar el poder de SQL a MapReduce com a resultat de la recerca de Facebook per a interaccions de dades més interactives i basades en SQL.

- **Apache Impala**, com a entorn de consultes a l'estil de l'analítica de negocis per a Hadoop.

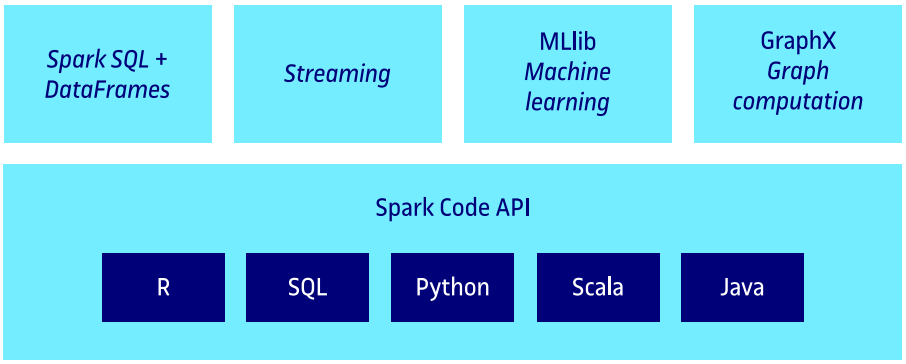
- **Prest**, com un motor de processament distribuït d'alta velocitat per executar consultes SQL en bases de *big data*.

Partint del disseny, Spark estava orientat a explotar el potencial, en particular la velocitat i l'escalabilitat, ofert per processament *in-memory* de dades, ja que l'algorisme de MapReduce tendeix a escriure contínuament dades intermèdies en el disc al llarg d'un cicle de processament. En aquest sentit, Spark pot oferir resultats amb la mateixa configuració de dades fins a cent vegades més ràpid que MapReduce.

A més, les seves premisses de disseny s'han concentrat a oferir **rendiment**, **simplicitat**, **facilitat d'administració i desenvolupament més ràpid d'aplicacions**, per això una gran quantitat d'organitzacions i empreses ho han adoptat ràpidament.

Com es va esmentar anteriorment, Spark utilitza un model simple de programació que pot ser en Python, R, Scala o Java i disposa de *application programming interfaces* (APIs) ben definides per a diferents casos d'ús com: Streaming, SQL, ML, DataFrames i Graph Processing. La figura següent mostra l'ecosistema de Spark:

Figura 7. Ecosistema de Spark



3.1. Casos d'ús comuns per a Spark

Tenint en compte les característiques que presenta, com ara velocitat, simplicitat i productivitat per al desenvolupador, existeixen una gran quantitat de dominis que s'han bolcat a treballar amb Spark. Com per exemple: aplicacions amb flux de dades (*streaming*), intel·ligència artificial, aprenentatge automàtic i profund, intel·ligència empresarial o integració de dades.

Un cas específic dins de l'àmbit de la salut és el processament de dades generades *on-line* de monitors i sensors d'un pacient, que permeten disposar d'un registre continu i escalable de grans quantitats d'informació per detectar variacions o canvis en la informació monitorada per generar alertes, informes o registres simplement de les parts que es consideren essencials per guardar.

Un altre domini en el qual Spark comença a donar els seus fruits és en l'àmbit de la intel·ligència artificial (IA), l'aprenentatge automàtic (ML) i l'aprenentatge profund (DL), ja que les organitzacions comencen a utilitzar aquestes tècniques i algorismes cada vegada més per fer front a la seva estratègia d'anàlisi de dades. En qualsevol cas d'ús associat amb IA, ML i DL es troben capacitats sofisticades de classificació i reconeixement de patrons que serveixen a aplicacions de decisió basades en dades. Però s'ha de tenir en compte que, segons l'opinió d'experts, les solucions d'IA necessiten entre vuit a deu vegades el volum de dades utilitzat per a les solucions actuals de *big data*. Aquestes dades provenen de moltes fonts, com ara dades de sistemes de gestió corporativa (ERP), bases de dades, *datalakes*, sensors, dades públiques, aplicacions mòbils, xarxes socials i dades heretades i que poden ser estructurades o no estructurades i de diversos formats, per la qual cosa el processament en memòria i els fluxos de dades altament eficients de Spark el converteixen en un motor d'anàlisi ideal per a la preparació, transformació i manipulació de dades en aquesta mena de projectes basats en IA.

Spark proporciona el rendiment necessari per permetre l'anàlisi *on-line* (o *on-the-fly*) de sensors i monitors de senyals que, en un escenari de seguiment i vigilància de pacients, per exemple, permet una combinació d'anàlisi per un sistema d'informació per a que els professionals puguin prendre decisions més ràpides i precises.

3.2. Estructura de les dades

Un aspecte important en *big data* és vincular i gestionar dades de diverses fonts. Aquesta tasca és àrdua i requereix molta dedicació i un còmput intensiu; aquest procés es coneix com extreure, transformar i carregar (*extract, transform, and load* – ETL). Spark inclou mecanismes per fer la integració de proveïdors i plataformes de tecnologia d'emmagatzematge de dades i permet realitzar aquest procés de manera ràpida i senzilla pel seu processament en memòria i sense emmagatzemar les dades en el disc dur, la qual cosa aconsegueix eficiències des de x10 fins a x100 en relació amb les aplicacions clàssiques de MapReduce basades en disc.

Per aconseguir aquests resultats, Spark treballa amb diferents estructures de dades que han anat evolucionant amb el temps: *resilient distributed datasets* (RDD), *dataframes* i *datasets*.

Inicialment, el 2011 Spark es va desenvolupar amb RDD, però en veure l'auge de Spark en altres àmbits, els desenvolupadors van implementar *dataframes* el 2013 i *datasets* el 2015.

Els **conjunts amb resiliència de dades distribuïdes (*resilient distributed datasets*, RDD)** són l'estructura de dades fonamental (inicial) de Spark. Permet emmagatzemar les dades distribuïdes en els múltiples nodes del clúster i fer el processament en paral·lel. És tolerant a fallades, ja que si es realitzen múltiples transformacions en l'RDD i després, per qualsevol motiu, hi ha una fallada en algun node, l'RDD és capaç de recuperar-se automàticament.

En més detall són estructures en memòria destinades a contenir la informació que es vol carregar i després, processar per Spark. S'etiqueten «amb **resiliència**» perquè mantenen un registre del seu historial (per exemple, modificacions i eliminacions) perquè puguin reconstruir-se si hi ha fallades durant el seu processament, i a més són **distribuïts** perquè el *dataset* es parteix i es reparteix entre els diferents nodes de l'arquitectura subjacent del clúster per augmentar l'eficiència per mitjà del processament paral·lel.

Una aplicació de Spark pot crear una estructura RDD i després carregar dades que poden provenir de gairebé qualsevol font de dades, com per exemple: Hadoop, Apache Cassandra, Amazon S3, Apache HBase o bases de dades relacionals, entre d'altres. Hi ha tres maneres de crear un RDD:

1) paral·lelitzar una col·lecció de dades existent,

2) llegir un fitxer de dades extern, o

3) crear RDD a partir d'un RDD ja existent, per exemple:

```
# 1) paralel·lització de la recopilació de dades
my_list = [1, 2, 3, 4, 5, 6]
my_list_rdd = sc.parallelize (my_list)

# 2) Referència a un fitxer de dades extern
file_rdd = sc.textFile ("ruta_del_fitxer")
```

Una pregunta que ens podem fer és: quan hem d'utilitzar un RDD? És recomanable utilitzar l'RDD en les següents situacions:

- Quan es volen fer transformacions de baix nivell en el conjunt de dades (més endavant es veuran transformacions sobre els RDD).

- Quan les dades carregades no tenen un esquema i és necessari especificar manualment l'esquema de cada conjunt de dades quan es crea l'RDD.

Els ***dataframes*** es van integrar a Spark a partir de la versió 1.3 per superar les limitacions de l'RDD. Spark Dataframes és la col·lecció distribuïda de dades, però organitzats en les columnes nomenades (admeten capçaleres), i permet als desenvolupadors depurar el codi durant el temps d'execució, la qual cosa no és possible amb els RDD.

Els *dataframes* poden llegir i escriure les dades en diversos formats com CSV, JSON, AVRO, HDFS i taules HIVE, i estan optimitzats per processar grans conjunts de dades per a la majoria de les tasques de preprocessament, per la qual cosa no és necessari que el desenvolupador escrigui funcions complexes. Per crear un *dataframe,* per exemple, a PySpark:

```
spark.createDataFrame (
 [
 (1, 'Pedro'),
 (2, 'Ana'),
.
.
.
.
 (100, 'Alicia')
 ],
 ['id', 'Nom'] # etiqueta de les columnes
)
```

Els ***datasets*** són una extensió dels *dataframes* que inclouen els seus beneficis i els dels RDD en permetre processar de manera eficient dades estructurades i no estructurades. És una estructura que permet un processament ràpid i proporciona una interfície segura, ja que el compilador de Spark validarà els tipus de dades de totes les columnes del conjunt de dades durant la compilació únicament i llançarà un error si hi ha alguna discrepància en els tipus de dades. Qui utilitzi RDD trobarà que el codi és molt similar, però notarà un increment notable en la velocitat de processament. Com que s'ha de verificar el codi durant la compilació, els *datasets* només estan disponibles per a Scala i Java (no per a Python).

Entre les característiques principals de les tres estructures de dades es poden esmentar:

Taula 1. Característiques principals de les estructures de dades

	RDD	***Dataframe***	***Dataset***
Representació	Col·lecció de dades distribuïda sense esquema	Col·lecció de dades distribuïda organitzada en columnes amb nom	Extensió de *dataframes* amb control dels tipus de dades
Optimització	Sense optimització	Optimitzat	Optimitzat

	RDD	***Dataframe***	***Dataset***
Esquema	Esquema manual	Cerca automàtica de l'esquema	Ídem *dataframe* però amb una SQL Engine
Operacions	Lent per fer operacions com per exemple agrupar dades	Realitza agregacions més ràpid que RDD i *datasets*	Millor que RDD però pitjor que *dataframes*

Els *dataframes* permeten un conjunt d'**operacions** per a la manipulació de dades estructurades (a continuació se'n mostren algunes). Abans de començar a treballar amb un *dataset* s'ha de crear una SparkSession i aquests paràmetres es creen automàticament si s'accedeix per *pyspark*:

Vegeu també

A la bibliografia d'aquest mòdul trobareu més informació sobre manipulació de dades estructurades.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('Spark Training').getOrCreate()
sc = spark.sparkContext
```

Per crear un *dataframe* des d'un arxiu CSV:

```
df = spark.read.format('csv').options(delimiter=',', header=True).load('/carpetas/archivo.csv')
```

Per convertir RDD a *dataframe*:

```
rdd = sc.parallelize([(1,2),(3,4),(5,6)])
rdf = rdd.toDF()
```

Per crear-lo utilitzant el mètode spark.createDataFrame:

```
tdf = spark.createDataFrame([('Pedro',24),('Ana',43)],['nombre','edad'])
tdf.printSchema()
root
 |-- name: string (nullable = true)
 |-- age: long (nullable = true)
```

Per mostrar les columnes del *dataframe*:

```
tdf.columns
['nom', 'edat']
```

Per mostrar les dades d'un *dataframe*:

```
df.show(5, truncate=False)
+------------+---------+
|nom         |   edat  |
+------------+---------+
|Pedro       |   24    |
|Ana         |   43    |
```

```
+------------+---------+
```

Per als RDD, Spark permet dos tipus principals d'operacions: **transformacions** i **accions**.

1) Les **transformacions** inclouen filtrat, mapatge o manipulació de les dades, generant un nou RDD, ja que els RDD són immutables, és a dir que l'RDD original roman sense canvis. Aquesta immutabilitat permet que Spark faci un seguiment de les modificacions que es van dur a terme durant la transformació i, per tant, desfer els canvis posteriors que s'han realitzat. Un altre punt interessant del treball amb RDD és que les transformacions són «mandroses» (*lazy*), és a dir, no s'executen fins al moment en què es necessiten, per exemple, mitjançant una acció. Aquest comportament ajuda a millorar el rendiment en retardar els càlculs fins al moment en què es requereixen, en lloc de realitzar càlculs que podrien no ser necessaris segons el flux del programa.

2) Les **accions** són funcions per obtenir informació de les dades, però no les modifiquen, com per exemple Recompte, Recuperació d'un element específic, Agregació (unió d'informació, no inserció). Quan una aplicació sol·licita una acció, Spark avalua l'RDD inicial i després crea instàncies d'RDD en funció del que se li demana que faci, mirant quin lloc del clúster serà el més adequat, des del punt de vista de l'eficiència, per col·locar el nou RDD.

Com s'ha esmentat, les transformacions generen un nou RDD i entre les més utilitzades es poden trobar les següents:

- **map(func)**: retorna un nou RDD passant cada element de la font a través de la funció *func*.

- **filter(func)**: retorna un nou RDD seleccionant aquells elements de la font que la seva *func* retorna *true*.

- **flatMap(func)**: similar a *map*, però cada entrada pot ser mapejada a 0 o més sortides, la qual cosa podria retornar una seqüència en lloc d'un sol ítem.

- **union(otherDataset)**: retorna un nou *dataset* amb la unió de tots dos.

- **distinct([numTasks]))**: retorna un nou RDD que conté aquells elements diferents del RDD original.

- **groupByKey([numTasks])**: si l'RDD està organitzat per claus i valor (K, V) retorna un nou *dataset* agrupat per K (K, Seq[V]).

- **reduceByKey(func, [numTasks])**: genera un nou RDD on els valors per a cada clau estan agregats per la *func*.

- **sortByKey([ascending], [numTasks])**: retorna un nou RDD ordenat.

Entre les accions més importants es troben:

- **redueix(func)**: agrega els elements del *dataset* utilitzant *func*.
- **collect()**: retorna tots els elements del *dataset*.
- **count()**: retorna el nombre d'elements del *dataset*.
- **take(n)**: retorna un *array* amb els primers *n* elements del *dataset*.
- **saveAsTextFile(path)**: escriu els elements del *dataset* com un arxiu de text en el sistema d'arxius local, en HDFS o qualsevol altre sistema d'arxius suportat per Hadoop.
- **countByKey()**: només disponible en RDDs del tipus (K, V) i retorna un mapa de (K, Int) tuples per a cada *key* K.
- **foreach(func)**: executa una funció *func* sobre cada element del *dataset*.

3.3. Instal·lació de Spark, PySpark i Spider

PySpark és una llibreria de Spark escrita en Python per executar una aplicació Python usant les capacitats d'Apache Spark, és a dir, PySpark és una API de Python per a Spark.

Spark bàsicament està escrit en Scala, però donada la ràpida adaptació a entorns de ciència de dades es va desenvolupar l'API PySpark per a Python usant Py4J (llibreria de Java integrada dins de PySpark), i permet que Python interactuï de manera dinàmica amb objectes Java, per la qual cosa és necessari tenir instal·lat Java (es recomana versió 8) per executar PySpark.

És simple realitzar la instal·lació de PySpark, ja que es poden seguir les instruccions de la pàgina dels desenvolupadors i es pot executar en un node o estendre en un clúster.

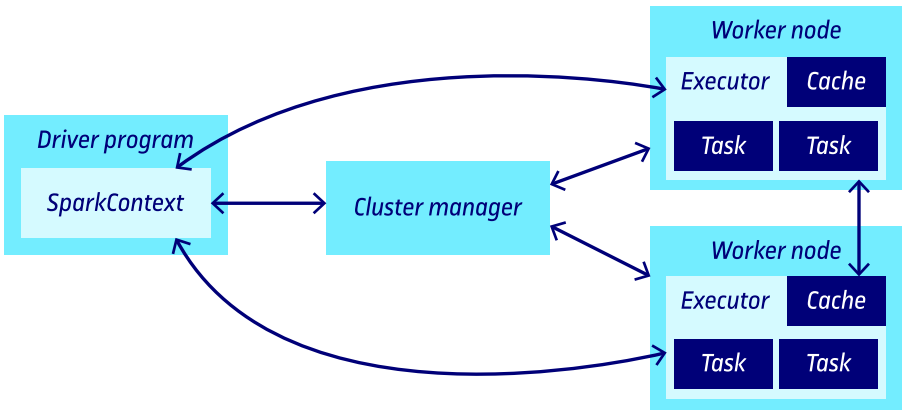
Per facilitar el desenvolupament de programes en aquesta prova de concepte s'ha instal·lat sobre una màquina virtual (MV) en la qual s'ha inclòs, a més de Spark i PySpark, un IDE (*integrated development environment*) anomenat Spyder, que permet tenir un entorn d'edició, desenvolupament i execució sobre PySark (també s'utilitza molt Jupyter notebook).

Enllaç recomanat

Per veure la configuració i integrar Spyder amb PySpark es pot consultar la referència següent: «Setup and run PySpark on Spyder IDE».

PySpark és una opció molt utilitzada en ciència de dades i aprenentatge automàtic, ja que hi ha nombroses biblioteques de ciència de dades escrites en Python que inclouen NumPy o TensorFlow, entre d'altres. Apache Spark funciona en una arquitectura mestre-treballador (*master-worker*), en què el mestre es diu *driver* i els treballadors *workers*. Quan executa una aplicació Spark, el *driver* crea un **context**, que és un punt d'entrada a la seva aplicació, i totes les operacions (transformacions i accions) s'executen en els nodes treballadors, i el *cluster manager* administra els recursos (vegeu la figura següent).

Figura 8. Context de Spark



Entre els *cluster managers* més utilitzats per Spark es poden enumerar els següents:

- **Standalone**: és un *cluster manager* simple inclòs amb Spark per a una configuració fàcil d'un clúster.
- **Apache Mesos**: utilitza com a *cluster manager*Mesos, que permet executar aplicacions Hadoop MapReduce i PySpark (encara que els desenvolupadors consideren que aquest mètode és obsolet).
- **Hadoop YARN**: és el gestor de recursos des de Hadoop 2 i normalment és el *cluster manager* més utilitzat.
- **Kubernetes**: és un entorn *open source* per automatitzar, desplegar, gestionar i escalar aplicacions en *contenidors* (Docker).
- Local: no és realment un *cluster manager* però és útil quan s'executa Spark sobre un MV o un ordinador, i és el que s'utilitzarà en aquestes proves de concepte. A local se li pot indicar un paràmetre [k] per llançar *k workers* (idealment igual al nombre de *cores* de la màquina virtual –que es pot modificar amb el paràmetre VCPU en la seva creació– o física).

Enllaç recomanat

Existeix una gran quantitat de llibreries i paquets que permeten estendre la funcionalitat de Spark i que es troben a https://spark-packages.org/

Un cop d'instal·lat i configurat, Spark inclou, a més de la consola a la qual es pot accedir posant en un terminal **pyspark**, una interfície web que permet analitzar el desenvolupament dels treballs en execució posant com URL en un navegador http://localhost:4040. Aquesta interfície mostrarà, per exemple:

Jobs, Stages, Tasks, Storage, Environment, Executors i SQL per controlar l'estat de l'aplicació en execució com es mostra a les imatges següents generada amb la MV prèviament esmentada.

Figura 9. Interfície de Spark

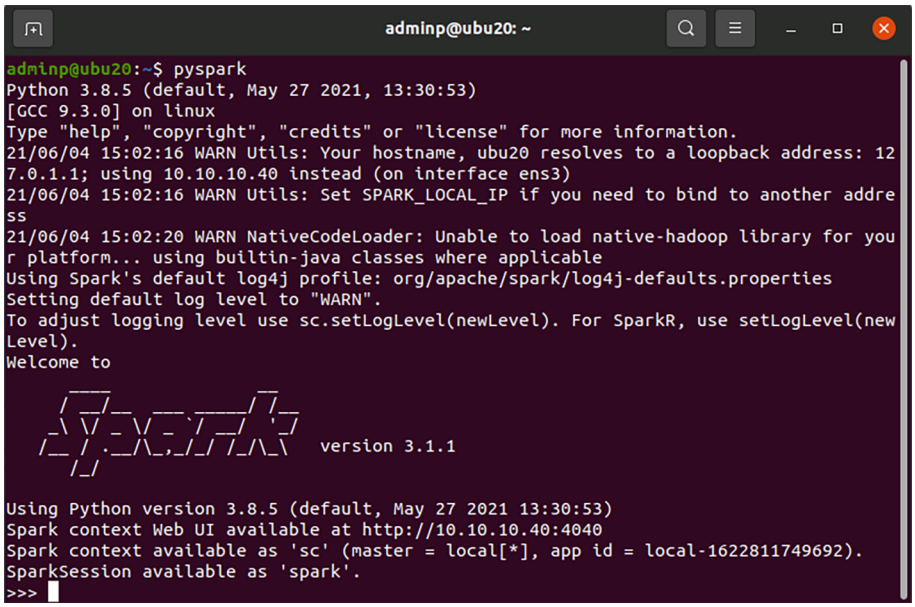
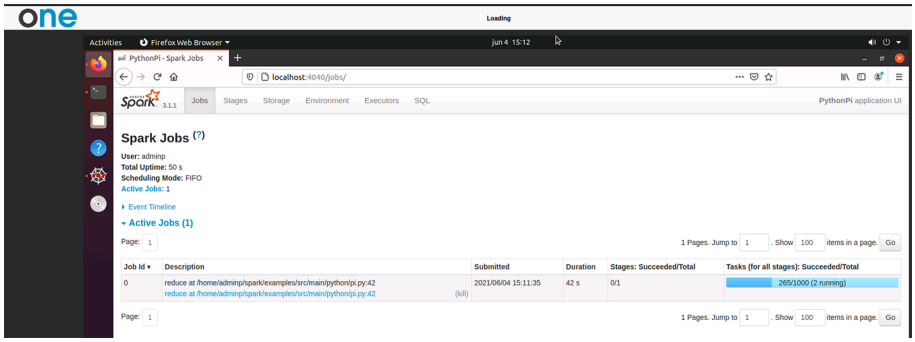


Figura 10. *Spark jobs*



Com es pot interactuar amb Spark de manera fàcil? Simplement executant pyspark i quan es tingui el *prompt* >>> executar, per exemple (vegeu que quan s'inicia pyspark ja genera el context com **sc** i crea una sessió com **spark**, per la qual cosa ja es pot entrar codi Python per processar les dades de manera interactiva):

Figura 11. Interactuar amb Spark

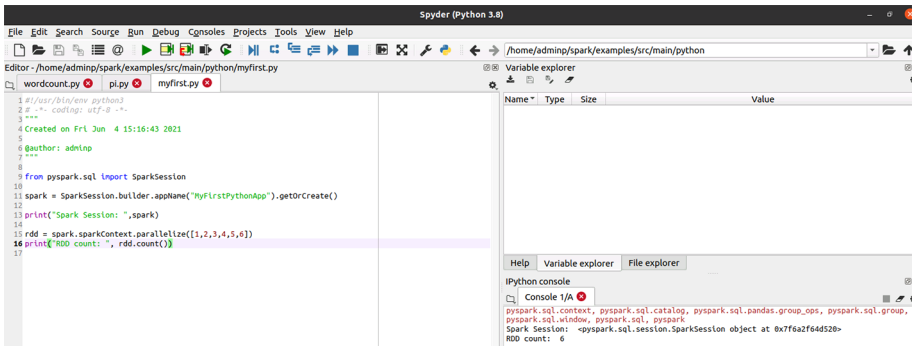
En aquest exemple s'ha creat un vector de dades (*data*), després s'ha creat l'RDD (*disDAta*) i paral·lelitzat les dades, sobre les quals després s'ha executat una acció per comptar les dades.

3.4. Programa en PySpark utilitzant Spyder (IDE)

En la figura següent es pot veure un programa simple (*myfirst.py*), que es troba en el directori /home/adminp/spark/examples/src/main/python/ i a continuació es descriu cadascuna de les línies:

- Línia 9: importació de les llibreries.
- Línia 11: creació d'una sessió.
- Línia 13: impressió de valor de la variable *spark*.
- Línia 15: creació d'un RDD distribuït amb un *array* de dades d'1 a 6.
- Línia 16 impressió d'una acció (*count*) sobre l'RDD per comptar el nombre d'elements de l'RDD.

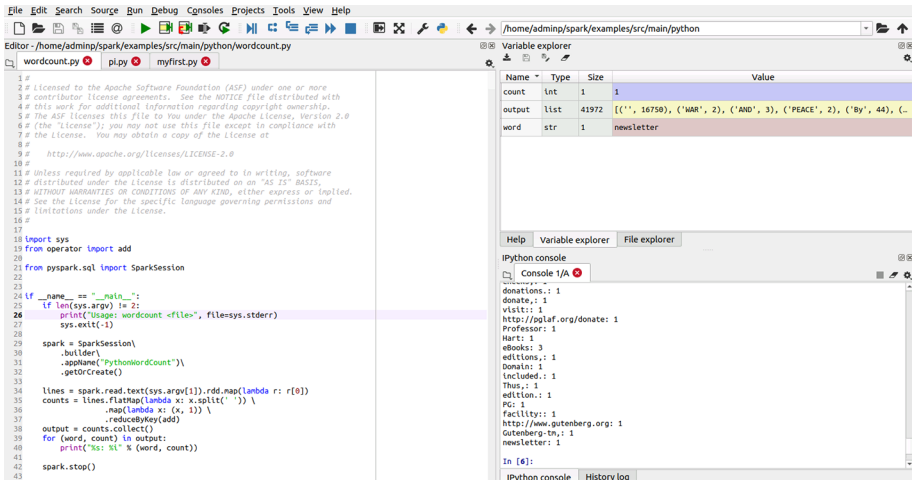
Figura 12. *myfirst.py*



Per executar el programa en Spyder simplement s'obre l'arxiu i després es clica la fletxa d'execució (la fletxa de color verd del menú d'icones). A la finestra dreta inferior es pot veure el resultat de l'execució.

En el mateix directori que hem comentat abans es podrà trobar el programa *wordcount.py,* que és similar als que ja hem tractat anteriorment.

Figura 13. *wordcount.py*



En aquest, el codi de cada línia és:

- Línies 18, 19, 21: importació de les llibreries.
- Línies 25, 26, 27: funció d'error si no es proveeix un arxiu per processar.
- Línia 29: creació de la sessió de Spark.
- Línia 34: lectura de l'arxiu i creació d'un RDD amb cada línia de l'arxiu.
- Línia 35: creació d'un segon RDD amb les paraules separades per espais, generació de la tupla (w, 1) i reducció per la clau (*key*), en aquest cas la paraula.
- Línia 38: emmagatzematge de l'RDD en un *array.*
- Línia 39: impressió de les dades amb el format paraula:comptador.
- Línia 42: final de la sessió Spark.

Per executar s'ha de fer clic a *play* (la fletxa de color verd del menú d'icones), però abans s'haurà d'indicar l'arxiu d'entrada (en aquest cas *war-peace.txt*); això es fa al menú *Run> Configuration per file> marcar Command Line Options* i en el quadre de text incloure el directori on hi ha l'arxiu d'entrada (en aquest cas

el directori txt/war-peace.txt). A la sortida es poden observar a la dreta, a la finestra superior, les variables utilitzades i el seu valor, i a la inferior, el resultat del processament.

4. Distribucions específiques

4.1. Cloudera

És una companyia que, sobre la base d'Apache Hadoop, ha desenvolupat un ecosistema propi basat en suport empresarial i serveis juntament amb programes de formació per a grans clients.

La distribució original *open source* d'Apache Hadoop es deia CDH (Cloudera Distribution Hadoop) i s'enfocava al desenvolupament d'aquesta tecnologia orientada a empreses. Un aspecte interessant d'aquesta companyia i beneficiós per a l'*open source* és que part del temps dels seus experts es dedicava (segons l'empresa fins a un 50 %) a diferents projectes *open source* (Apache Hive, Apache Avro, Apache HBase, etc.), i formava també part del directori de l'Apache Software Foundation.

El 2019, l'empresa es va unir amb una altra gran empresa que disposava d'un entorn similar de gran qualitat anomenat Hortonworks. Van ajuntar les dues distribucions i es va generar una macrodistribució amb el millor de cadascuna d'elles. El 2019-2020, orienta el seu negoci cap al *cloud* (fent entorns consistents en *cloud* i anàlisi de dades) i llança la seva primera Cloudera Data Platform Private Cloud i actualitza les seves plataformes amb nous serveis (CDP Data Engineering, CDP Operational Database i CDP Data Visualization Services).

Actualment (2021), el producte es denomina Cloudera Data Platform (CDP), que és una plataforma estable i escalable centrada en entorns *cloud* que rep el nom de First Enterprise Data Cloud. La CDP permet analitzar dades per mitjà d'entorns *cloud* (propis o *multicloud*, per exemple: Azure, Google, IBM...), però **lamentablement ha tancat les seves llicències** (malgrat que els seus principals components són *open source*) i ara es regeix per un sistema de *paywall* i es requereix una subscripció vàlida per accedir a qualsevol dels seus productes:

- Apache Hadoop (CDH)
- Hortonworks Data Platform (HDP)
- Data Flow (HDF/CDF)
- Cloudera Data Science Workbench (CDSW)

4.2. Apache Bigtop

L'objectiu principal de Bigtop és un projecte (*open source*) basat en la comunitat que estableix les bases per a l'empaquetat, la implementació i les proves d'interoperabilitat de projectes relacionats amb Hadoop.

Això inclou proves en diversos nivells (empaquetat, plataformes, temps d'execució, actualització, etc.), i ho desenvolupa una comunitat amb un enfocament en el sistema com un tot, en lloc de projectes individuals. Aquest projecte desenvolupa i publica codi de prova de compilació, empaquetat i integració que depèn de les versions oficials dels projectes relacionats amb Apache Hadoop (HDFS, MapReduce, HBase, Hive, Pig, ZooKeeper, etc.). El seu interès és que, a mesura que es trobin errors i altres problemes, poder solucionar-los com més aviat millor.

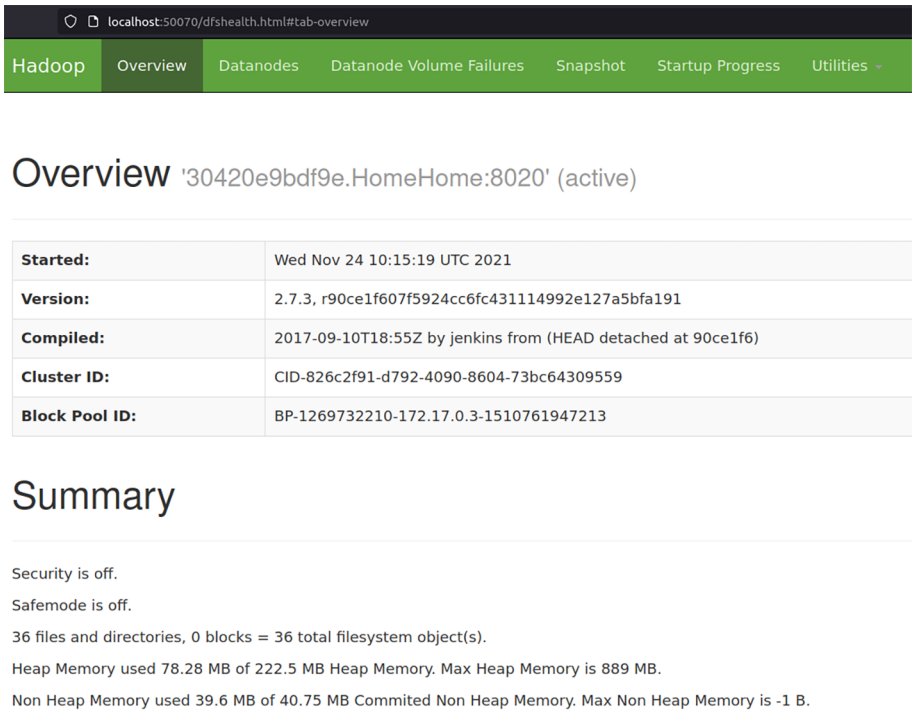
L'última versió de Bigtop (3.0.0) és d'octubre de 2021 i inclou els següents components: alluxio, ambari, bigtop-ambari-mpack, bigtop-groovy, bigtop-jsvc, bigtop-utils, elasticsearch, flink, gpdb, hadoop, hbase, hive, kafka, kibana, livy, logstash, oozie, phoenix, solr, spark, sqoop, tez, ycsb, zeppelin, zookeeper.

Com a prova de concepte es pot executar o construir un pseudo clúster de Hadoop amb Docker utilitzant les imatges que proveeixen els desenvolupadors o construint les imatges de Docker amb les utilitats que es donin. Per exemple, per desplegar Hadoop HDFS només cal fer:

```
docker run -d -p 50070:50070 bigtop/sandbox:1.2.1-ubuntu-16.04-hdfs
```

A localhost:50070 es podrà veure la interfície, com es mostra a la figura següent:

Figura 14. Interfície de Bigtop



Com es pot observar, no és una versió actualitzada, però és totalment funcional. Per acabar l'execució del contenidor cal mirar el CONTAINER_ID amb `docker ps --all` i després executar `docker stop CONTAINER_ID`. Per eliminar el contenidor `docker rm CONTAINER_ID`, per eliminar la imatge s'ha de mirar l'IMAGE_ID amb `docker images` i esborrar-la amb `docker rmi IMAGE_ID`. Com es pot observar a la pàgina del desenvolupador també es pot emmagatzemar el resultat de l'execució del *docker run* en una variable (per exemple, BIGTOP, que serà el CONTAINER_ID i després utilitzar $BIGTOP com a valor de CONTAINER_ID).

Si es vol executar Hadoop HDFS + Hbase:

```
BIGTOP=$(docker run -d -p 50070:50070 -p 16010:16010 bigtop/sandbox:1.2.1-ubuntu-16.04-hdfs_hbase)
docker exec -ti $BIGTOP hbase shell
```

Per executar Hadoop HDFS + Spark Standalone:

```
BIGTOP=$(docker run -d -p 50070:50070 -p 8080:8080 bigtop/sandbox:1.2.1-ubuntu-16.04-hdfs_spark
-standalone)
docker exec -ti $BIGTOP spark-shell
```

Si es vol executar Hadoop HDFS + YARN + Hive + Pig:

```
BIGTOP=$(docker run -d -p 50070:50070 -p 8088:8088 bigtop/sandbox:1.2.1-ubuntu-16.04-hdfs_yarn
_hive_pig)
docker exec -ti $BIGTOP hive
```

```
docker exec -ti $BIGTOP pig
```

Es podrien crear nous contenidors amb les instruccions que hi ha a la pàgina que hem comentat (o al directori bigtop-3.0.0/docker/sandbox/Readme.md) per a l'última versió, però s'ha de fer un treball important, ja que només està configurat per a Ubuntu 16.04, per exemple, i aquest ja és obsolet.

Enllaç recomanat

A https://dlcdn.apache.org/bigtop/bigtop-3.0.0/repos/ hi ha els repositoris per instal·lar-lo a sobre d'Ubuntu20.04 o Debian 10, per exemple.

Un altre aspecte interessant és el directori *provisioner* dins de la distribució que permet utillizar Docker o Vagrant per generar Virtual Hadoop Cluster. En aquesta prova s'ha fet servir Docker i per mitjà de Docker Compose es pot crear un Bigtop Virtual Hadoop Cluster sobre contenidors Docker. Aquest clúster es pot fer servir per:

- provar els testos inclosos a Bigtop (*smoke tests*)
- provar les *Bigtop puppet recipes*
- executar el test d'integració amb l'aplicació de l'usuari

En aquest cas s'ha canviat de nom l'arxiu *config_ubuntu-20.04.yaml* com *config.yaml* i s'ha executat `./docker-hadoop.sh --create 2` per crear un clúster Hadoop amb dos nodes sobre Docker 20.10.7. A l'arxiu Readme.md es mostren diferents maneres d'accedir o executar ordres dins dels contenidors, però utilitzant l'*script* es pot entrar dins del primer contenidor executant per exemple:

```
./docker-hadoop.sh --exec 1 bash
```

I al directori/arxiu *config/hosts* es podran veure les IP assignades a cada contenidor i al navegador de *host* es podrà posar com a URL IP:50070 per veure l'estat del clúster o IP:8088 per veure l'estat de totes les aplicacions, com es mostra a continuació. L'estat dels contenidors posats en marxa es pot veure amb:

```
./docker-hadoop.sh -l
```

Figura 15. Estat dels contenidors

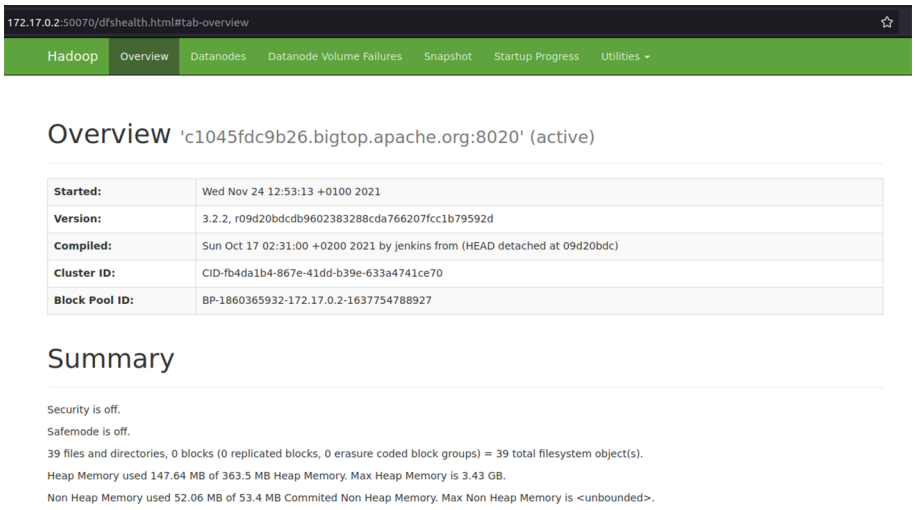
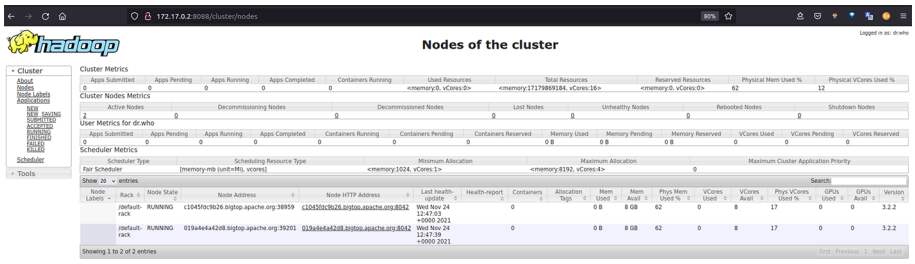


Figura 16. Estat de les aplicacions



Enllaç recomanat

Per a més informació sobre l'última versió 3.0.0 consulteu «Bigtop 3.0.0 Release».

4.3. Big Data Europe

Un altre entorn interessant, com a prova de concepte, és el projecte Big Data Europe, que permet desplegar diferents components de la infraestructura programari per al processament de *big data* (per exemple un clúster en Hadoop o Spark) per mitjà de Docker i amb Swarm com a gestor de recursos. Per exemple, per desplegar un clúster Hadoop s'ha de fer un clonat del repositori big-data-europe/docker-hadoop:

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

A continuació:

```
docker-compose up
```

Per executar l'exemple de comptador de paraules:

```
make wordcount
```

O per desplegar un entorn a Swarm:

```
docker stack deploy -c docker-compose-v3.yml hadoop
```

El `docker-compose` crearà una xarxa que es pot localitzar executant `docker network list` (en el nostre cas *docker-hadoop_default*) i fent `docker network inspect docker-hadoop_default` es podrà veure l'assignació de les IP als diferents nodes de les interfícies de Hadoop en els següents URL:

- Namenode: http://<*IP_address*>:9870/dfshealth.html#tab-overview
- History server: http://<*IP_address*>:8188/applicationhistory
- Datanode: http://<*IP_address*>:9864/
- Nodemanager: http://<*IP_address*>:8042/node
- Resource manager: http://<*IP_address*>:8088/

Quan s'introdueixi l'URL del *resource manager* (http://<IP_address>:8088/) al navegador s'obtindrà el mateix entorn que es mostra a les figures anteriors (p, ex. la figura 16).

Bibliografia

Abbasi, M. A. (2017). *Learning Apache Spark 2*. Birmingham: Packt Publishing. https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781785885136 (última consulta: novembre de 2021).

Apache Spark. «Spark SQL, DataFrames and Datasets Guide». https://spark.apache.org/docs/3.1.1/sql-programming-guide.html (última consulta: novembre de 2021).

Bengfort, B; Kim, J. (2016). *Data Analytics with Hadoop: An Introduction for Data Scientists*. Sebastopol: O'Reilly Media.

Chambers, B.; Zaharia. M. (2018). *Spark: The Definitive Guide. Big Data Processing Made Simple*. Sebastopol: O'Reilly.

Damji, J. S.; Wenig, B. *et al.* (2020). *Learning Spark: Lightning-Fast Data Analytics*. Sebastopol: O'Reilly Media.

DataNoon (2018). «Pyspark DataFrame Operations - Basics | Pyspark DataFrames». https://datanoon.com/blog/pyspark_dataframe_operations/ (última consulta: novembre de 2021).

DeRoos, D. (2014). *Hadoop For Dummies*. Hoboken: Wiley & Sons.

Hurwitz, J.; Nugent, A.; Halper, F.; Kaufman, M. (2013). *Big Data for Dummies*. Hoboken: Wiley & Sons. https://big-data.digital/big-data-for-dummies/ (última consulta: novembre de 2021).

Mendelevitch, O.; Stella, C; Eadline, D. (2017). *Practical Data Science with Hadoop and Spark: Designing and Building Effective Analytics at Scale*. Boston: Addison-Wesley Professional.

Schneider, R. D.; Karmiol, J. (2019). *Spark for Dummies (2nd IBM Limited Edition)*. Hoboken: Wiley & Sons. www.ibm.com/downloads/cas/WEB4XBOR (última consulta: novembre de 2021).

Scott, J. A. (2015). *Getting Started with Apache Spark*. San Jose (EE. UU.): MapR Technologies.

Shook, A.; Miner, D. (2012). *MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems*. Sebastopol: O'Reilly Media.

Singh, C.; Kumar, M. (2019). *Mastering Hadoop 3: Big data processing at scale to unlock unique business insights*. Birmingham: Packt Publishing.

White, T. (2015). *Hadoop: The Definitive Guide: Storage and Analysis at Internet Scale* (4a edició, cap. II). Sebastopol: O'Reilly Media.

Webs amb icones amb llicència d'ús lliure

www.iconarchive.com (última consulta: novembre de 2021).

www.customicondesign.com (última consulta: novembre de 2021).

http://icons8.com (última consulta: novembre de 2021).

Repositoris útils d'OpenData

«Cosmic»: és una base de dades analitzada per experts que abraça la gran varietat de mecanismes de mutació somàtica que causen el càncer humà. En aquest lloc, ved «Cosmic, the Catalogue Of Somatic Mutations In Cancer» (conté dealls sobre milions de mutacions en milers de tipus de càncer. Està en creixement constant, tant en contingut com en abast). Vegeu «Cancer Gene Census» (última consulta: novembre de 2021).

«OpenData, COVID-19»: NCATS està generant un seguit de conjunts de dades mitjançant la selecció d'un panell d'assajos relacionats amb el SARS-CoV-2 davant de tots els medicaments aprovats. Aquests conjunts de dades, així com els protocols d'assaig que s'utilitzen per generar-los, es posaran immediatament a disposició de la comunitat científica en aquest lloc web a mesura que es completin (última consulta: novembre de 2021).

«SARS-CoV-2 proteins – NCBI Datasets BETA»: es pot seleccionar i descarregar un conjunt de dades sobre proteïnes de genomes complets de SARS-CoV-2, incloses seqüències de nucleòtids i proteïnes, anotacions, estructures i un informe de dades detallat (última consulta: novembre de 2021).