ARQUITECTURA DE SISTEMES BIG DATA PEL RECORD LINKAGE DE XARXES SOCIALS

TREBALL FINAL DE GRAU – Informe de progrés I

2018-2019

ALEJANDRO GARCIA CARBALLO
1423957

Índex:

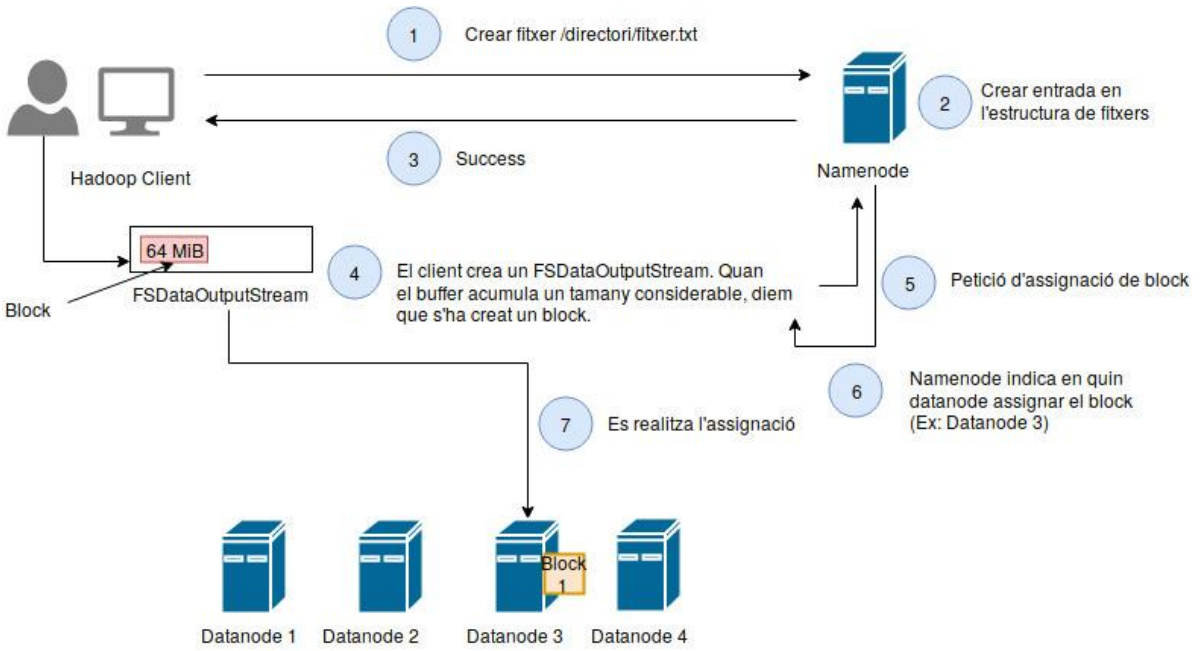
1. Arquitectures de les tecnologies 3
1.1. HDFS (Hadoop Distributed File System) 3
1.2. YARN (Yet Another Resource Manager) 5
1.3. Spark 6
2. Arquitectura del sistema implementat 7
3. Desenvolupament de les imatges utilitzades 8
3.1.Avantatges i inconvenients 8
3.2. Imatges utilitzades 10
4. Xarxa Docker i execució dels containers 14
4.1. Ports Hadoop i Spark 15
5. Configuració Hadoop i Spark 16
6. Comprovació del correcte funcionament 18
7. Seguiment de la planificació 20
8. Bibliografia 21

1 Arquitectures de les tecnologies

Abans de començar a especificar una arquitectura i d'explicar com fer la implementació i configuració d'un node Hadoop, és important entendre cadascun dels mòduls que el composen. En aquest apartat s'explicarà quina és la funció de cadascun dels mòduls i com funcionen, i més endavant explicaré com configuraré cadascun d'aquests mòduls.

1.1 HDFS (Hadoop Distributed File System)

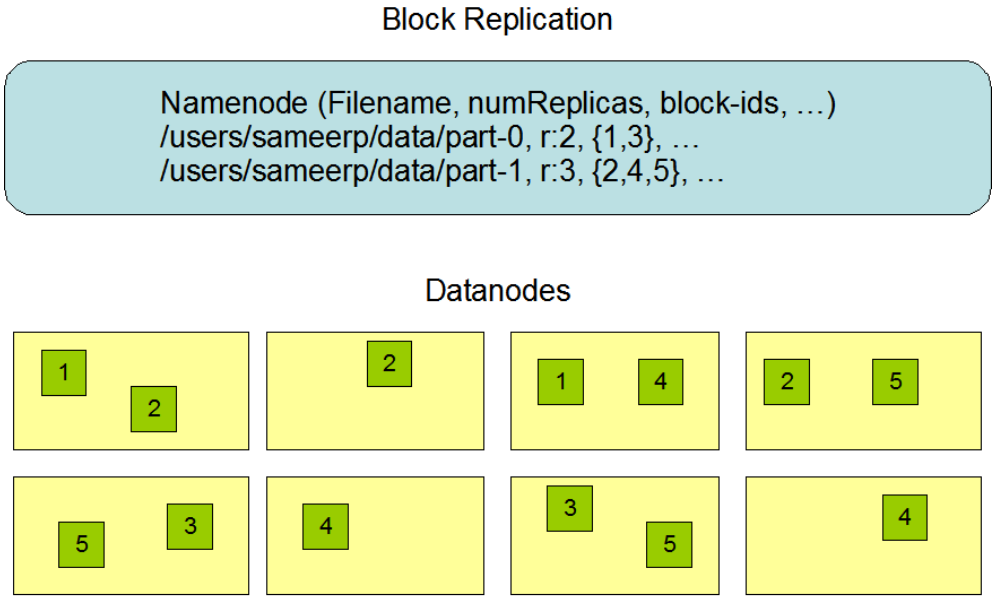
El HDFS [1] [2] té una estructura de master/slave. El master del HDFS s'anomena namenode i s'encarrega de guardar l'estructura del sistema de fitxers i gestionar l'accés als fitxers demanats pel client. Acostuma a ser el node al que se li assigna més RAM, ja que ha de mantenir en memòria tota l'estructura de fitxers entre altres coses. Els nodes treballadors s'anomenen datanodes, normalment un per node del clúster, que gestiona l'emmagatzematge adjuntat als nodes on s'executen. Els usuaris poden emmagatzemar fitxers en el HDFS com si es tractés de qualsevol altre sistema de fitxers conegut com per exemple els de Linux, però a diferència d'aquest, el HDFS és un sistema distribuït. Per aconseguir això, el fitxer que es vol emmagatzemar en el HDFS es divideix en un o més blocs, que acostumen a ser de 64 MiB, i aquests blocs es guarden en el conjunt de datanodes de manera distribuïda. El namenode és l'encarregat de guardar en quin datanode es troben els blocs de dades, per tant, quan un client demana algun fitxer, el namenode buscarà quins son els datanodes que contenen els blocs necessaris, i els datanodes retornaran les dades demanades. Els datanodes també s'encarreguen de la creació, eliminació i replicació dels blocs a través de les instruccions indicades pel namenode.



Il·lustració 1 Funcionament arquitectura HDFS

En la il·lustració 1 podem veure tot el procés que es realitza per fer l'assignació d'un bloc de dades. El client fa una petició per crear un fitxer dins del sistema de fitxers de Hadoop al namenode. El namenode que és l'encarregat de gestionar i guardar l'estructura del sistema de fitxers, crea una entrada en el sistema de fitxers i si l'operació es duu a terme correctament, retorna un success al client. A partir d'aquí, el client crea un buffer que va acumulant dades fins arribar al tamany d'un bloc. El client pregunta al namenode quin és el datanode al que ha de assignar aquest bloc, el namenode indica a quin datanode assignar-ho i finalment es realitza l'assignació.

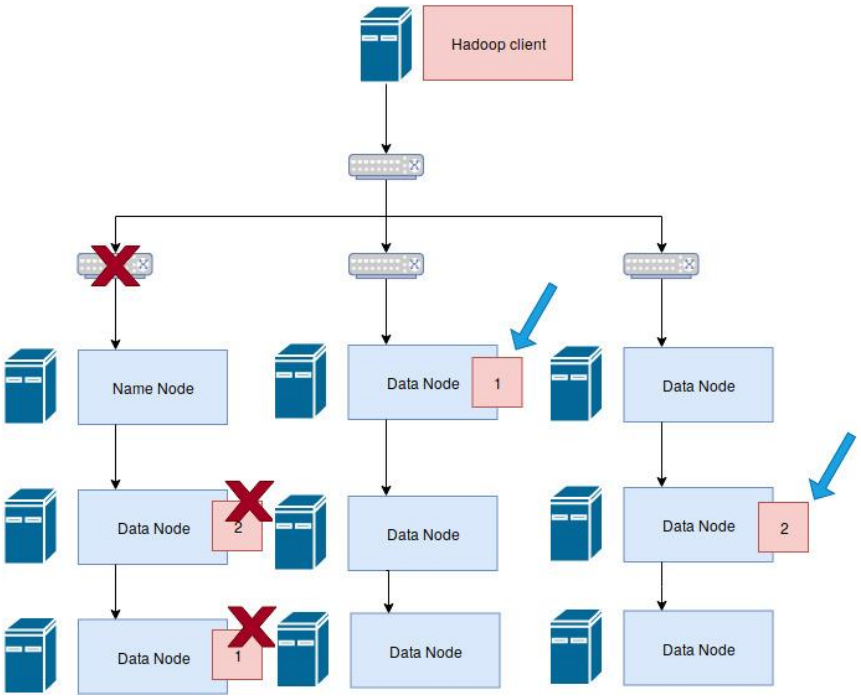
El HDFS és un sistema que garanteix alta disponibilitat i tolerància a fallades . Per aconseguir això Hadoop aporta una senzilla però efectiva solució:



Il·lustració 2 Replicació de blocs de dades [1]

Quan s'emmagatzema un bloc en un Datanode, es realitza una rèplica en un altra datanode. El nombre de copies que es realitzen es poden concretar a través del Replication Factor, que es pot configurar per tal d'indicar el nombre de còpies. Quan un bloc no està disponible en un datanode, les dades s'hauran de llegir des de un altra datanode que contingui una copia. S'ha d'escollir un nombre adequat de replication factor segons les nostres necessitats, ja que fer un procés de replicació de blocs es costós. L'espai que podem ocupar amb les rèpliques pot augmentar molt i arribar a ser un problema. Normalment s'utilitza un factor de replicació de 3, si el nombre és més gran a 3, la rèplica 4 i les següents a aquesta s'acostumen a assignar de manera aleatòria mantenint el nombre de rèpliques per sota del límit superior de la següent manera: $\frac{repliques - 1}{racks + 2}$

Si estem treballant en una topologia rack, Hadoop permet aplicar el que s'anomena "rack awareness".



Il·lustració 3 Funcionament rack awareness

Aplicant rack awareness el que fem es assegurar que si un switch deixa de funcionar, i per tant, tots els datanodes que estan funcionant amb aquest switch també deixen de funcionar, sempre tinguem una copia del bloc de dades en un altre rack. En aquest cas, les dades es llegiran de les rèpliques que contenen els blocs en altres datanodes.

Una pregunta que pot sorgir és: que fem si el namenode falla? Al ser una arquitectura master/slave, si el namenode falla implicaria que no es podrien dur a terme cap tipus d'operació. Per solucionar això, el HDFS fa un backup de la informació del namespace (tota l'estructura de fitxers) i es gestiona un fitxer de logs (editLogs) per registrar totes les operacions que es realitzen. Es selecciona un node per a ser un Namenode en standby i en el cas de que el namenode deixi de funcionar, aquest namenode secundari llegirà tant les dades del backup com els logs per poder fer el paper de namenode.

Amb aquesta explicació queda demostrat que el HDFS és un sistema que garanteix alta disponibilitat i tolerància a fallades, i el funcionament bàsic d'aquest. Tot això servirà a l'hora de configurar el HDFS.

1.2 YARN (Yet Another Resource Manager)

Yarn és un gestor de recursos que va sorgir a partir de la versió 2 de Hadoop. Inicialment, els components principals de Hadoop eren el HDFS i Mapreduce. Amb l'aparició de YARN, a la part més alta de l'arquitectura de Hadoop es pot utilitzar un mòdul de processament de dades distribuïdes diferent a Mapreduce, i a més, poder gestionar-ho.

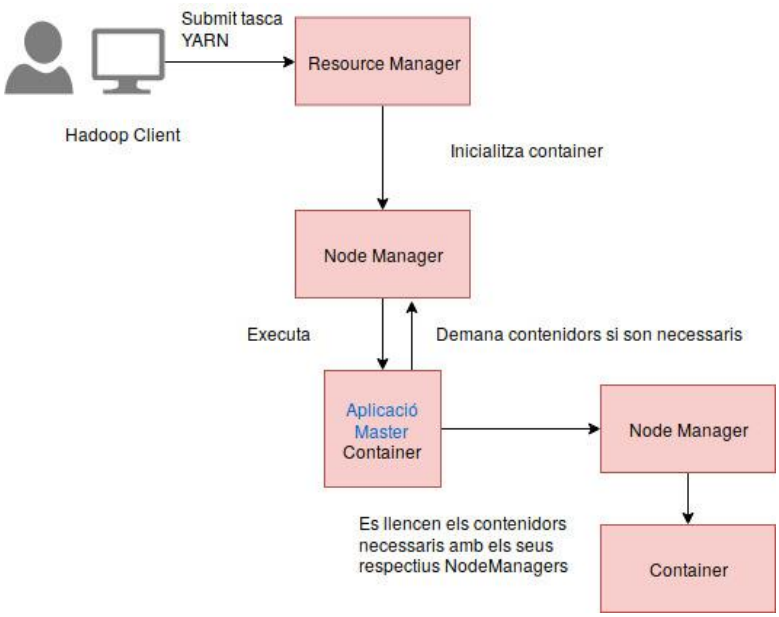
La idea fonamental de YARN és dividir les funcionalitats de la gestió de recursos i la planificació / supervisió de tasques en dimonis separats. La idea és tenir un

ResourceManager (RM) global i un ApplicationMaster (AM) per aplicació. Una aplicació és un treball únic o un DAG de treballs.

El framework de computació de dades està format pel ResourceManager i el NodeManager. El ResourceManager és l'encarregat de gestionar els recursos entre totes les aplicacions del sistema. El NodeManager és a totes les màquines i s'encarrega de monitoritzar la utilització de recursos en cadascuna d'aquestes, i a la vegada informar al ResourceManager d'això.

El ResourceManager està format per dos components principals:

- **Scheduler:** És l'encarregat d'assignar els recursos a les diferents aplicacions tenint en compte les limitacions de capacitat, les cues de treball, etc.
- **ApplicationManager:** S'encarrega d'acceptar peticions de treball i de negociar amb els contenidors on s'executen aquests treballs per tal d'executar una aplicació específica.



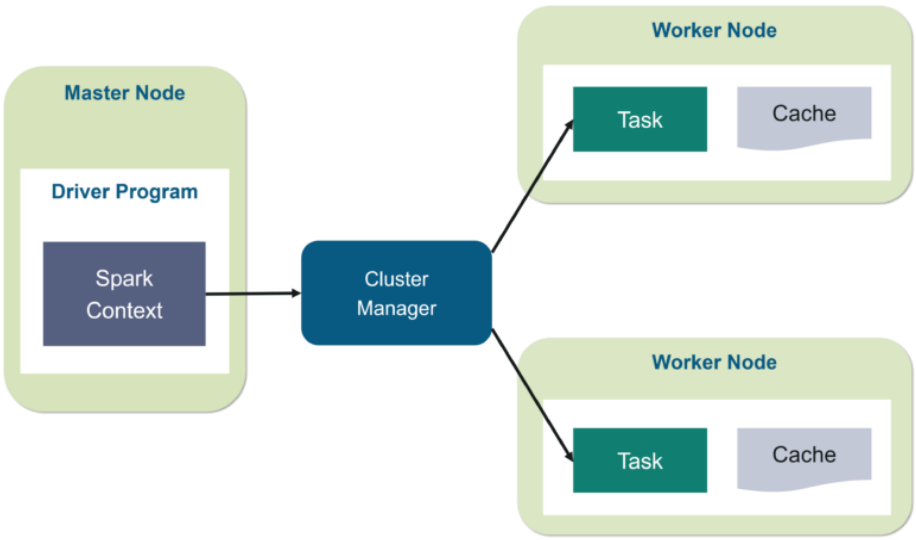
Il·lustració 4 Funcionament YARN

1.3 Spark

Spark [3] [8] és un framework de codi obert de computació de dades distribuïdes en clústers. La característica principal de Spark és la computació en clústers en memòria, amb la qual s'aconsegueix la velocitat de processament d'una aplicació. Les característiques principals de Spark són les següents:

- **Velocitat:** Spark executa 100 vegades més ràpid que Hadoop Mapreduce en el processament de dades molt grans.

- **Treballa en memòria:** Una capa de programació simple proporciona la capacitat de poder treballar en memòria i a la vegada també permet persistència de dades en disc.
- **Desplegament:** Pot ser desplegat a través de diversos gestionadors de clusters com ara Hadoop YARN, Mesos o el propi de Spark.
- **Temps real:** Ofereix computació en temps real i baixa latència gràcies a es treballa en memòria.
- **Diversitat de llenguatges disponible:** Gràcies al conjunt de APIs de les que disposa es pot treballar en llenguatges d'alt nivell com ara Java, Scala, Python o R.



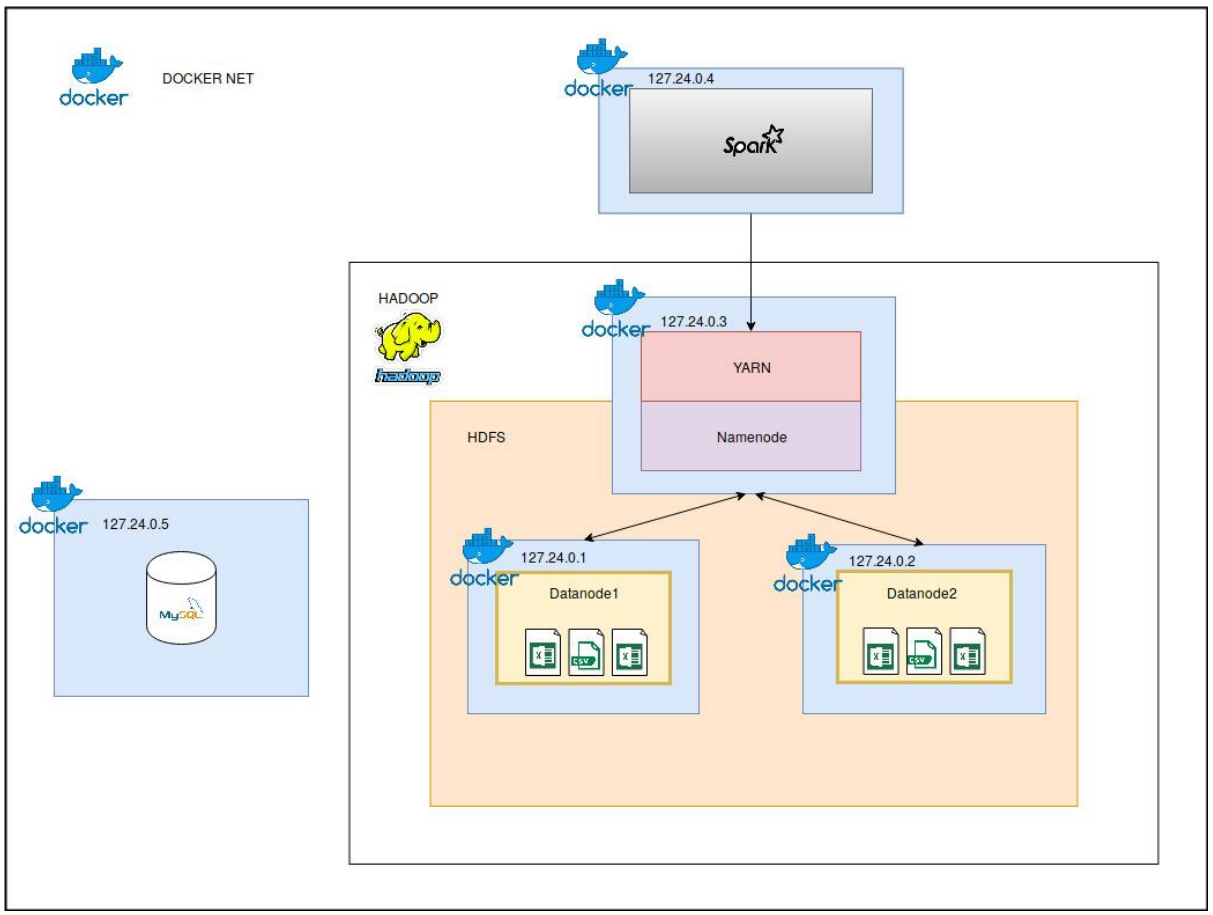
Il·lustració 5 Arquitectura Spark [8]

El node mestre és el que conté el driver program, l'encarrega de conduir tota l'aplicació. El codi escrit (o bé les ordres de la shell si utilitzem l'intèrpret) és el que es comporta com a driver program. Dins d'aquest driver program el primer que s'ha de fer sempre es crear el Spark Context, el qual és l'encarregat de establir connexió amb entorn d'execució de Spark i preparar els serveis interns d'aquests. Aquest Spark Context treballa amb un clúster manager per tal de gestionar diversos treballs. Un treball es divideix en múltiples tasques que son distribuïdes a través dels worker nodes. Els workers nodes són els nodes esclaus que s'encarreguen bàsicament d'executar les tasques.

2 Arquitectura del sistema implementat

En la il·lustració 6 tenim l'arquitectura general del projecte. Com es pot observar, hi ha 5 containers Docker: un container farà la funció de namenode i yarn manager, després tindrem dos datanodes, on s'emmagatzemen les dades csv i excels, un container que contindrà Spark i finalment un dedicat a MySQL. Els elements de Hadoop necessiten estar en una mateixa xarxa de Docker, per tal de poder comunicar-se, per això he creat una xarxa amb Docker a la que pertanyen tots els elements de l'arquitectura, de manera que no s'han de configurar els fitxers de /etc/hosts com s'hauria de fer en una instal·lació habitual sense Docker.

He decidit utilitzar dos datanodes ja que actualment la quantitat d'informació a emmagatzemar no és gran. Realment podría ser emmagatzemada en un únic datanode, però he decidir afegir un més per a poder fer un ús del replication factor. D'aquesta manera, tractaré de que sempre hi hagi una rèplica en els dos datanodes i si algun dels dos té algun inconvenient, es podran llegir les dades des de l'altre datanode i aconseguir així un sistema amb més disponibilitat.



Il·lustració 6 Arquitectura general del sistema implementat

3 Desenvolupament de les imatges utilitzades

Per realitzar aquesta arquitectura he utilitzat Docker. Per implementar l'arquitectura hi havien dues opcions principals: utilitzar imatges Docker ja creades (de Docker Hub, per exemple [1] [4]) o bé crear les teves pròpies imatges i tot el necessari per a que funcionin. En un inici, la meva idea era utilitzar imatges ja creades i després fer el deployment d'aquestes, però durant la realització d'aquesta part he trobat molts problemes a l'hora d'utilitzar imatges ja creades que m'han fet perdre molt de temps. Per aquesta raó finalment m'he decidit a crear les meves pròpies imatges per a quasi tots els mòduls.

3.1 Avantatges i inconvenients

A continuació presentaré quins son els avantatges i inconvenients que he trobat a l'hora d'utilitzar imatges creades i crear una imatge pròpia:

Utilitzar imatges ja creades	
Avantatges	**Inconvenients**
Reducció de temps d'implementació.	Problemes i errors: dificultat a l'hora de trobar la causa d'aquests.
Maduresa del projecte.	Components no són a mida. Pot contenir mòduls que no necesites.
Diversitat d'imatges.	Pot acabar prolongant el temps previst a causa dels possibles errors que sorgeixen.

Com a avantatges tenim que si tot funciona correctament, t'estalvies una gran quantitat de temps. A més, molts del projectes que hi ha darrera d'algunes imatges porten molt de temps en desenvolupament i per tant, son projectes més madurs que les imatges que he realitzat. Hi ha una gran quantitat i diversitat d'imatges que faciliten trobar imatges adequades a les teves necessitats. Però també hi ha diversos inconvenients. Si algunes de les imatges conté algun error o per alguna causa no tot funciona correctament, al no tenir un coneixement de com està muntada aquesta, quan s'han de trobar solucions als problemes es complica molt. També esta el fet de que la majoria d'imatges no compleixen amb els requisits estrictament, alguns utilitzen mòduls extra que no necessito i d'altres tenen versions bastant posterior a les actuals de les diferents tecnologies. Aquest problema també sorgeix a causa de que ni Hadoop ni Spark tenen unes imatges oficials, les imatges existents no son oficials i per tant el manteniment tampoc és el mateix. En canvi, imatges com ara la de MySQL [5], son oficials i a l'hora d'utilitzar-les no m'han donat cap mena de problemes.

Crear imatges pròpies	
Avantatges	**Inconvenients**
Mòduls a mida. Tens només els mòduls que tu decideixes.	Has de dedicar temps a la creació de la imatge.
Poder utilitzar la versió més actualitzada de les tecnologies.	La maduresa d'aquestes imatges potser no es el mateix que algunes de les creades.
Coneixement més profund del funcionament del sistema.	Al ser per a ús propi, no es rep feedback per a corregir possibles errors.

Com a avantatges tenim que podem escollir quins mòduls volem utilitzar i no tenir res que no utilitzem. La versió de les tecnologies a utilitzar pot ser la més actual. A més, al crear tu mateix tota l'estructura de les imatges, entens el funcionament de cadascuna de les parts i suposa una major facilitat a l'hora de trobar els errors i arreglar-los. Com a inconvenients tenim que s'ha de dedicar part del temps a desenvolupar aquestes imatges i que el temps que porta en desenvolupament aquesta imatge segurament es menor que el de les imatges

ja creades. També tenim l'inconvenient de que en aquest cas, si utilitzem la imatge per a ús propi, no es rep feedback de la comunitat per tal de corregir possibles errors.

3.2 Imatges utilitzades

Les diferents imatges que utilitzaré son les creades en els següents Dockerfiles, els quals he creat basant-me en alguns ja creats [4] [6] i modificant i afegint tot allò que veia necessari:

- **hadoop-base-tfg:** Aquesta imatge és la base de la resta i la que conté tots els elements bàsics necessaris per a poder tenir un correcte funcionament en tota la resta d'imatges. Aquesta imatge la he basat en ubuntu:16.04, ja que és un sistema al qual estic bastant acostumat i al ser Linux servia per a poder realitzar la feina. Utilitzaré l'última versió de Hadoop i Spark. Està creada amb variables d'entorn per a poder actualitzar fàcilment el seu contingut. Les variables creades són les següents:

```
FROM ubuntu:16.04

# set environment vars
ENV HADOOP_HOME /opt/hadoop
ENV HADOOP_VERSION 3.2.0
ENV JAVA_HOME /usr/lib/jvm/java-8-openjdk-amd64
ENV HDFS_NAMENODE_USER root
ENV HDFS_DATANODE_USER root
ENV HDFS_SECONDARYNAMENODE_USER root
ENV YARN_RESOURCEMANAGER_USER root
ENV YARN_NODEMANAGER_USER root
```

Il·lustració 7 Variables entorn hadoop base

Per a poder fer funcionar Hadoop és necessari tenir SSH ja que els diferents nodes es comuniquen a través de SSH, tenir el JDK de Java, per a la versió 3.2.0 de Hadoop podem utilitzar el JDK 8 i un editor per si és necessari modificar algun document dins el contenidor.

```
# install packages
RUN \
  apt-get update && apt-get install -y \
  ssh \
  rsync \
  vim \
  openjdk-8-jdk
```

Il·lustració 8 Instal·lació de SSH i JDK

Seguidament, descarrego l'última versió de Hadoop, es descomprimeix el fitxer i creo les variables d'entorn necessàries per al fitxer de hadoop-env.sh, on s'indica tant el directori de java 8 jdk com els usuaris de Hadoop i el directori dels executables de Hadoop en el PATH del sistema:

```
# download and extract hadoop, set JAVA_HOME in hadoop-env.sh, update path
RUN \
  wget http://apache.mirrors.tds.net/hadoop/common/hadoop-$HADOOP_VERSION/hadoop-$HADOOP_VERSION.tar.gz && \
  tar -xzf hadoop-$HADOOP_VERSION.tar.gz && \
  mv hadoop-$HADOOP_VERSION $HADOOP_HOME && \
  echo "export JAVA_HOME=$JAVA_HOME" >> $HADOOP_HOME/etc/hadoop/hadoop-env.sh && \
  echo "export HDFS_NAMENODE_USER=$HDFS_NAMENODE_USER" >> $HADOOP_HOME/etc/hadoop/hadoop-env.sh && \
  echo "export HDFS_DATANODE_USER=$HDFS_DATANODE_USER" >> $HADOOP_HOME/etc/hadoop/hadoop-env.sh && \
  echo "export HDFS_SECONDARYNAMENODE_USER=$HDFS_SECONDARYNAMENODE_USER" >> $HADOOP_HOME/etc/hadoop/hadoop-env.sh && \
  echo "export YARN_RESOURCEMANAGER_USER=$YARN_RESOURCEMANAGER_USER" >> $HADOOP_HOME/etc/hadoop/hadoop-env.sh && \
  echo "export YARN_NODEMANAGER_USER=$YARN_NODEMANAGER_USER" >> $HADOOP_HOME/etc/hadoop/hadoop-env.sh && \
  echo "PATH=$PATH:$HADOOP_HOME/bin" >> ~/.bashrc
```

Il·lustració 9 Instal·lació Hadoop

Com he comentat anteriorment, Hadoop es comunica amb la resta de nodes a través de SSH. Per tal d'evitar que es demani la password entre els nodes que pertanyen al cluster de Hadoop, es creen claus SSH que es guardaran en el fitxer "authorized_keys". També he configurat el fitxer de ssh_config per tal de permetre el ssh de hosts coneguts sense necessitat de password:

```
# create ssh keys
RUN \
  ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa && \
  cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys && \
  chmod 0600 ~/.ssh/authorized_keys
```

Il·lustració 10 Creació SSH keys

Aquesta configuració la passaré al sistema de la imatge a través de la comanda ADD:

```
# copy ssh config
ADD configs/ssh_config /root/.ssh/config
```

Il·lustració 11 Afegint configuració SSH

La manera com he gestionat el tema de configuracions en tots els casos es similar, creo els fitxers de configuració en local i els copio a través de ADD.

Com vaig explicar al començament, Hadoop té una arquitectura master/slave (en aquesta última versió de Hadoop se'ls anomena workers en comptes de slaves). Per tant, en algun fitxer s'ha d'indicar quins són els nodes treballadors. Aquest fitxer s'anomena "workers", i conté les IP (o hostnames) dels nodes que son treballadors.

```
COPY configs/* /tmp/

RUN mv /tmp/workers $HADOOP_HOME/etc/hadoop/workers
```

Il·lustració 12 Afegint configuració dels workers de Hadoop

En aquest cas, les IP associades tenen un alies dins de la xarxa creada per Docker i per tant he posat el hostname en comptes de la IP.

Per a la configuració dels fitxers de configuració principals de Hadoop (més endavant explicaré amb més detall) he creat els .xml en local i els he copiat al path de Hadoop on es guarden totes les configuracions:

```
# copy hadoop configs
ADD configs/*xml $HADOOP_HOME/etc/hadoop/
```

Il·lustració 13 Afegint totes les configuracions principals

Finalment el que faig és afegir el script start-hadoop.sh, que és l'encarregat de iniciar tots els serveis necessaris per a l'execució de Hadoop i de realitzar el format del namenode, que bàsicament el que fa és crear la metadata dels datanodes, que es guarda i es gestiona des de el namenode:

```
#!/bin/bash

# start ssh server
/etc/init.d/ssh start

# format namenode
$HADOOP_HOME/bin/hdfs namenode -format

# start hadoop
$HADOOP_HOME/sbin/start-dfs.sh
$HADOOP_HOME/sbin/start-yarn.sh
$HADOOP_HOME/sbin/mr-jobhistory-daemon.sh start historyserver

# keep container running
tail -f /dev/null
```

Il·lustració 14 Script start-hadoop.sh

- **namenode-tfg:** Aquesta imatge té tot el necessari per a ser el namenode. El hostname del resource manager de YARN també està en aquest mateix node.

```
FROM hadoop-base-tfg:latest

RUN mkdir -p /opt/hadoop/data/nameNode
EXPOSE 8020 9870 8088
VOLUME /opt/hadoop/data
CMD service ssh start
CMD bash start-hadoop.sh
```

Il·lustració 15 Dockerfile namenode-tfg

Creo el directori que utilitzarà el DFS per al namenode i faig un EXPOSE (fer accessibles aquests ports des de fora del container que es crea al executar la imatge) dels ports als quals haig d'accedir. Amb VOLUME faig que aquest directori sigui persistent, per tal de que quan l'execució del container s'aturi, no es perdin les dades. Finalment inicio el service de ssh i executo el script start-hadoop.sh comentat anteriorment.

- **datanode-tfg:** El datanode bàsicament haurà de fer les dades persistents i fer que es mantingui en execució per a que pugui ser utilitzar per el namenode:

```
FROM hadoop-base-tfg:latest

RUN mkdir -p /opt/hadoop/data/dataNode
EXPOSE 9864
VOLUME /opt/hadoop/data
CMD [ "sh", "-c", "service ssh start; bash"]
```

Il·lustració 16 Dockerfile datanode-tfg

- **spark-tfg:** Aquesta imatge té spark descarregat i conté tota la configuració necessària per a poder comunicar-se amb YARN.

```
FROM hadoop-base-tfg:latest

# set environment vars
ENV HADOOP_HOME /opt/hadoop
ENV SPARK_HOME /opt/spark
ENV SPARK_VERSION 2.4.0
ENV PATH "$PATH:$SPARK_HOME/bin"
ENV HADOOP_CONF_DIR $HADOOP_HOME/etc/hadoop
ENV LD_LIBRARY_PATH $HADOOP_HOME/lib/native:$LD_LIBRARY_PATH
```

Il·lustració 17 Variables entorn de Spark

La versió descarregada de Spark es la més actual fins al moment. A continuació [], descarrego Spark, realitzo l'extracció i creo les variables necessàries de Spark per al hadoop-env-sh. Es renombra el fitxer de spark-defaults.conf.template perquè posteriorment el substitueixo per un configurat:

```
#download and extract Spark
RUN \
  wget https://archive.apache.org/dist/spark/spark-$SPARK_VERSION/spark-$SPARK_VERSION-bin-hadoop2.7.tgz && \
  tar -xvf spark-$SPARK_VERSION-bin-hadoop2.7.tgz && \
  mv spark-$SPARK_VERSION-bin-hadoop2.7 $SPARK_HOME && \
  echo "export SPARK_HOME=$SPARK_HOME" >> $HADOOP_HOME/etc/hadoop/hadoop-env.sh && \
  echo "export HADOOP_CONF_DIR=$HADOOP_CONF_DIR" >> $HADOOP_HOME/etc/hadoop/hadoop-env.sh && \
  echo "export LD_LIBRARY_PATH=$LD_LIBRARY_PATH" >> $HADOOP_HOME/etc/hadoop/hadoop-env.sh && \
  mv $SPARK_HOME/conf/spark-defaults.conf.template $SPARK_HOME/conf/spark-defaults.conf
```

Il·lustració 18 Descarregant i instal·lant Spark

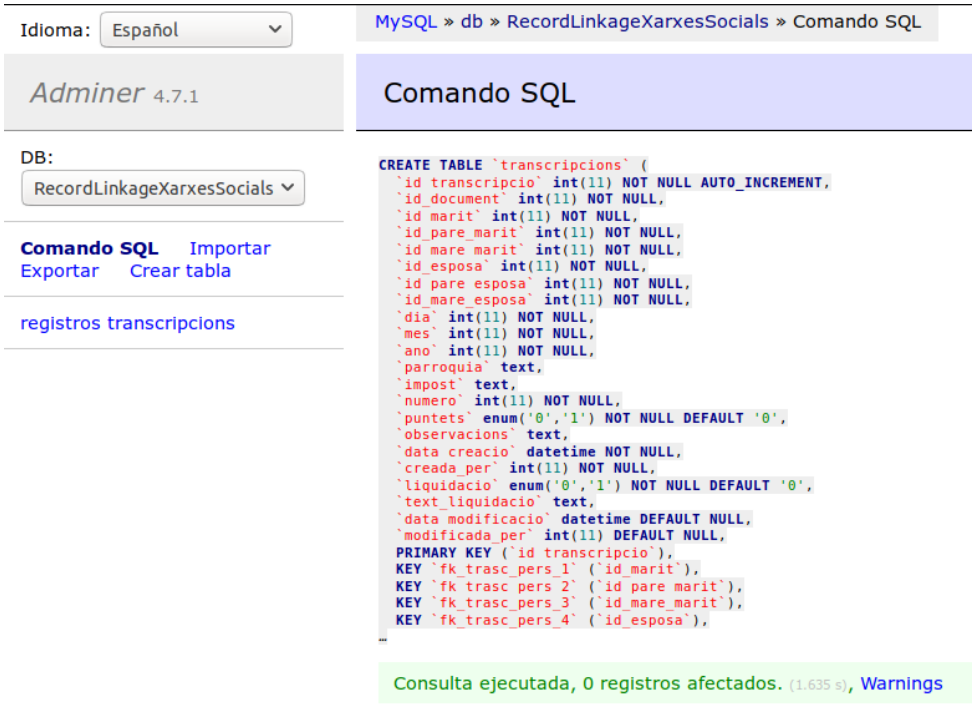
Faig la substitució per el fitxer configurat, s'exposen els ports necessaris i es manté el container en funcionament per a que no es tanqui:

```
#copy spark config
ADD configs/spark-defaults.conf $SPARK_HOME/conf/

# expose various ports
EXPOSE 4040 8088
```

Il·lustració 19 Afegint configuració de spark

- **mysql:** Per al mòdul de MySQL, he utilitzat la imatge oficial [5].
  Per a importar les dades en aquest container, he executat un script proporcionat per el tutor per a crear primer les taules.



Il·lustració 20 Creant taules

Seguidament he importat les dades en format csv a MySQL. Les dades estan separades per ";", ho indiquem a l'hora de fer el import i esborrem el nom de les columnes per a que no hi hagi errors a l'hora de realitzar el import.

	Tabla	Motor?	Colación?	Longitud de datos?	Longitud de índice?	Espacio libre?	Incremento automático?	Registros?	Comentario?
☐	**persones**	InnoDB	utf8_general_ci	16 384	16 384	0	3 502 798	0	
☐	**transcripcions**	InnoDB	utf8_general_ci	16 384	114 688	0	600 531	0	
	2 en total		utf8_bin	32 768	131 072	0			

Il·lustració 21 Taules creades amb les dades importades

I a continuació afegeixo les constraints, que en un inici no hem deixava realitzar-les ja que hi havia referències circulars.

```
ALTER TABLE transcripcions
ADD CONSTRAINT `fk trasc pers 1` FOREIGN KEY (`id marit`) REFERENCES `persones` (`id persona`) ON DELETE CASCADE ON UPDATE CASCADE,
ADD CONSTRAINT `fk_trasc_pers_2` FOREIGN KEY (`id_pare_marit`) REFERENCES `persones` (`id_persona`) ON DELETE CASCADE ON UPDATE CASCADE,
ADD CONSTRAINT `fk trasc pers 3` FOREIGN KEY (`id mare marit`) REFERENCES `persones` (`id persona`) ON DELETE CASCADE ON UPDATE CASCADE,
ADD CONSTRAINT `fk_trasc_pers_4` FOREIGN KEY (`id_esposa`) REFERENCES `persones` (`id_persona`) ON DELETE CASCADE ON UPDATE CASCADE,
ADD CONSTRAINT `fk trasc pers 5` FOREIGN KEY (`id pare esposa`) REFERENCES `persones` (`id persona`) ON DELETE CASCADE ON UPDATE CASCADE,
ADD CONSTRAINT `fk_trasc_pers_6` FOREIGN KEY (`id_mare_esposa`) REFERENCES `persones` (`id_persona`) ON DELETE CASCADE ON UPDATE CASCADE
```

Consulta ejecutada, 985 registros afectados. (3.253 s) Modificar

Il·lustració 22 Afegint les constraints

4 Xarxa Docker i execució dels containers

Per a gestionar l'execució dels containers primerament he desenvolupat un script en el qual he creat una xarxa de Docker anomenada "hadoop". En aquest script, es pot passar per paràmetre el nombre de datanodes que vols que s'inicialitzin, però actualment no s'haurien d'inicialitzar més de 2, ja que la configuració dels nodes "workers" està pensada per a dos datanodes. Els datanodes són executats en mode "detached", ja que aquests no han de realitzar cap execució sinó que han d'estar executant-se per a poder comunicar-se amb el namenode.

```
#Iniciando datanodes
if [ "$1" == "" ] ;
then
    echo "**** Iniciando 1 datanode ****"
    #docker run -dti --net hadoop --net-alias node1  -h node1 --name node1 --link namenode --link spark datanode-tfg
    docker run -dti --net hadoop --net-alias node1 -h node1 --name node1 datanode-tfg
    #-p 9864:9864
else
    counter=1
    for datanode in $(seq 1 $1);
    do
        echo "**** Iniciando datanode$counter ****"
        #commandSTR="docker run -dti --net hadoop --net-alias node$counter  -h node$counter --name node$counter --link namen
        commandSTR="docker run -dti --net hadoop --net-alias node$counter -h node$counter --name node$counter datanode-tfg"
        eval $commandSTR
        counter=$((counter+1))
    done
fi
```

Il·lustració 23 Script iniciar datanodes

Quan s'executen les imatges, es concreta que tots els containers estiguin en la mateixa xarxa "hadoop", i afegeixo un hostname (node1, node2, namenode…) per a poder fer una configuració més general i no haver d'assignar IPs fixes als contenidors. Finalment, per a cadascun dels ports accessibles del namenode, realitzo un mapeig dels ports, per a que puguin ser accessibles des de fora.

```
# Iniciamos namenode y spark
echo "**** Iniciando Namenode ****"
docker run -d --net hadoop --net-alias namenode --name namenode -h namenode -p 8020:8020 -p 9870:9870 -p 8088:8088 namenode-tfg

echo "**** Iniciando Spark ****"
docker run -dti --net hadoop --net-alias spark --name spark -h spark -p 4044:4044 spark-tfg
```

Il·lustració 24 Script iniciar namenode i spark

A part també he creat un script per fer el build de tots els Dockerfiles que hi ha en el directori. Tinc pensat fer el deployment en un docker-compose per poder gestionar els serveis de manera més còmoda que amb aquests scripts.

4.1 Ports Hadoop i Spark

Els ports per defecte han canviat de la versió 2 de Hadoop respecte de la versió 3 [9]. Com el Hadoop que estic utilitzant és la versió 3, els ports principals per defecte són els següents:

Nom	Valor	Descripció
9870	dfs.namenode.http-address	Interficie web on es poden consultar les dades principals del HDFS
8088	yarn.resourcemanager.webapp.address	Interficie web on es poden consultar les dades relacionades amb el resource manager de YARN
4044	spark.ui.port	Interficie per veure les aplicacions que s'estan executant en Spark

5 Configuració Hadoop i Spark

Per a que l'arquitectura compleixi certs requisits, Hadoop i Spark poden ser configurats a través de diferents fitxers. Tots els fitxers de configuració de Hadoop estan en format xml, on l'estructura per afegir una propietat és la següent:

```
<configuration>
        <property>
                <name> "nom de la propietat" </name>
                <value> "valor que volem assignar a aquesta propietat" </value>
        </property>
</configuration>
```

En la informació dels nodes dels clústers on treballaré s'especifiquen les següents característiques per a cadascun dels nodes:

Característiques
2 x 8 processor cores x64
64 GB de memòria
1 TB de disc dur
1 GB network connection

Els fitxers que he configurat són els següents:

- **core-site:** En aquest fitxer es configuren els paràmetres principals del HDFS.

Name	Value
fs.defaultFS	hdfs://namenode:9000

En la propietat fs.defaultFS s'indica quin és el sistema de fitxers per defecte. Gràcies a això es poden utilitzar les comandes de hdfs per a gestionar el contingut (pujar fitxers, llegir fitxers…) sense haver d'especificar tota la URL.

- **hdfs-site:** Conté tot el referent als diferents mòduls del HDFS.

Name	Value
dfs.replication	2
dfs.namenode.name.dir	/opt/hadoop/data/nameNode
dfs.datanode.data.dir	/opt/hadoop/data/dataNode

El dfs.replication fa referència al replication factor. Com he explicat anteriorment, al utilitzar dos datanodes, he decidir que el replication factor sigui de 2, per tal de realitzar una replica en els dos datanodes.

En el dfs.namenode.name.dir i dfs.namenode.data.dir el que s'indica és quin és el path on s'emmagatzemen les dades tant en el namenode com el datanode respectivament.

- **yarn-site:** Configuració de tot el referent a YARN.

Name	Value
yarn.resourcemanager.hostname	namenode
yarn.nodemanager.resource.memory-mb	55296
yarn.scheduler.minimum-allocation-mb	2048
yarn.scheduler.maximum-allocation-mb	21504

El paràmetre yarn.nodemanager.resource.memory-mb indica el total de memòria que pot ser assignada per als containers. He assignat un màxim de 54 GB disponible per als contenidors, ja que s'ha de tenir en compte que també s'ha de reservar memòria per al SO, i per als altres processos com ara el Namenode, Datanodes, etc. Per tant, dels 64 GB disponibles, he deixat 10 GB lliures i 54 GB utilitzables per als containers.

- **spark-defaults.conf:** Configuració referent a Spark [2] [5].

Name	Value
spark.master	yarn
spark.dynamicAllocation.initialExecutors	5
spark.executor.memory	21g
spark.executor.cores	5

Per a escollir aquesta configuració he seguit un enfocament balancejat entre "Tiny executors" i "Fat" (un executor per core) [10]. Quan s'executa una aplicació Spark utilitzant un gestor de clústers com Yarn, hi haurà diversos daemons que s'executaran en segon pla com NameNode, NameNode secundari i DataNode. Per tant, mentre s'especifiquen els executors, hem d'assegurar-nos que deixem de banda bastants nuclis (~ 1 nucli per node) perquè aquests daemons funcionin sense problemes.

El client HDFS té problemes amb un gran nombre de fils concurrents. Es va observar que HDFS aconseguia un rendiment complet d'escriptura amb ~ 5 tasques per executor. Per tant, és bo mantenir el nombre de nuclis per executor per sota d'aquest número.

Basant-me en aquestes recomanacions, assignaré 5 cores per executor per a tenir un bon rendiment del HDFS. Deixaré un core lliure per als daemons de Hadoop i Yarn. Per tant, el número de cores disponible pesarà a ser 16 - 1 = 15. Per tant, el nombre total de cores en el clúster serà de:

$$total\ cores\ =\ \#cores\ disponibles\ \cdot\ total\ nodes\ en\ cluster\ =\ 15\ *\ 2\ =\ 30$$

Com a número de executors disponibles tenim:

$$\text{Num. executors disponibles} = \frac{total\ cores}{num.cores\ per\ executor} = \frac{30}{5} = 6$$

Deixant un core per al ApplicationManager, tenim --num-executor = 5.

Si tenim que num.executors disponibles és igual a 6, cadascun dels nodes tindrà:

$$Executors\ per\ node\ =\ \frac{num.exec.disponibles}{total\ nodes\ cluster} = \frac{6}{2} = 3$$

Amb això tenim que la memòria per cada executor serà de:

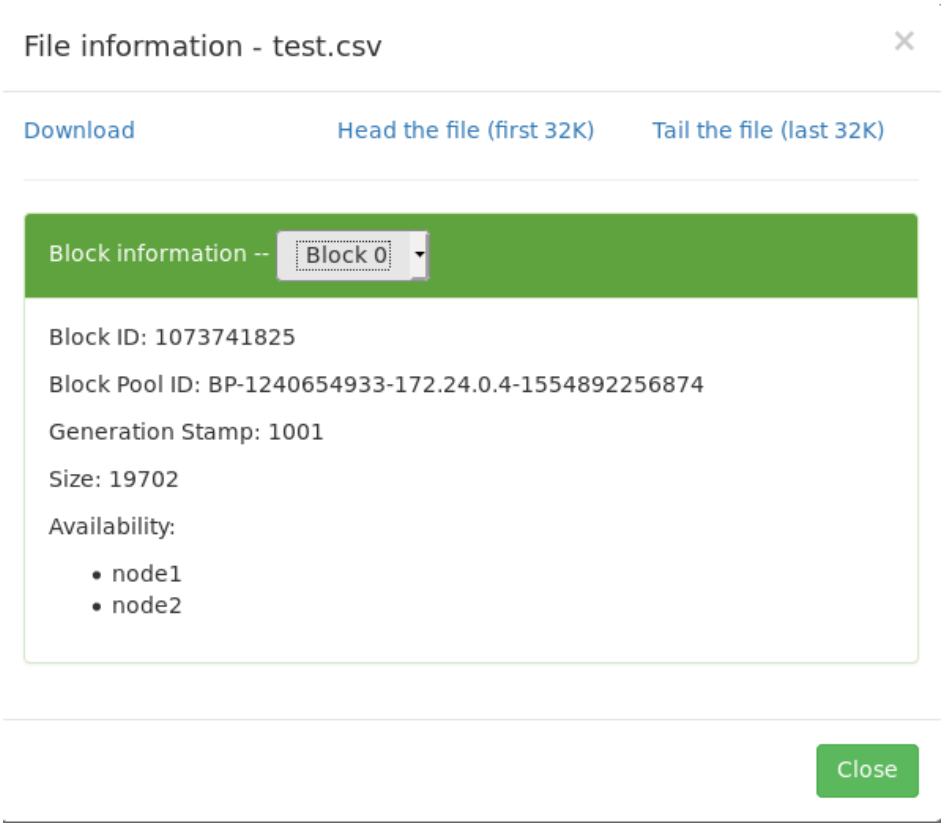
$$Memòria\ per\ executor\ =\ \frac{total\ memòria}{executor\ per\ node} = \frac{64\ GB}{3}\ =\ 21\ GB$$

6 Comprovació del correcte funcionament

Per a comprovar si tota la configuració funcionava he afegit algun arxiu al HDFS i he comprovat que s'estigués realitzant una replica seguint la configuració de replication factor = 2.
Per pujar el contingut he utilitzat la següent comanda:

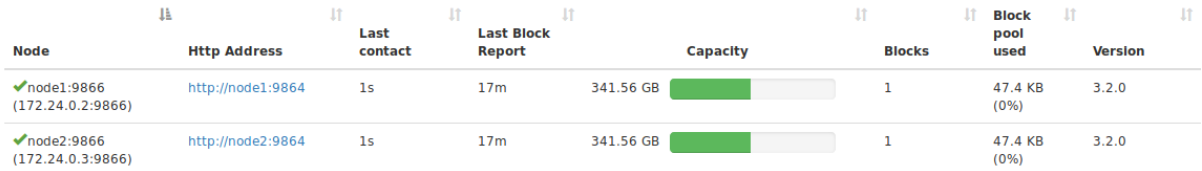
```
root@namenode:/# hadoop fs -put test.csv /test
```

Després, he anat a la interfície gràfica del HDFS i he comprovat que el fitxer s'hagués distribuït:



Il·lustració 25 Distribució fitxer en HDFS

Com podem comprovar, la disponibilitat del fitxer està tant en el node1 com en el node2. Si mirem informació de cadascun dels datanodes podem veure com cadascun dels datanodes té un bloc d'informació:



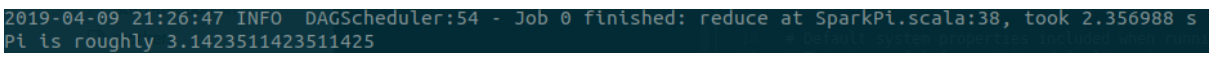
Node	Http Address	Last contact	Last Block Report	Capacity	Blocks	Block pool used	Version
node1:9866 (172.24.0.2:9866)	http://node1:9864	1s	17m	341.56 GB	1	47.4 KB (0%)	3.2.0
node2:9866 (172.24.0.3:9866)	http://node2:9864	1s	17m	341.56 GB	1	47.4 KB (0%)	3.2.0

Il·lustració 26 Disponibilitat fitxer en els datanodes

Referent a Spark, he realitzat una execució d'una de les aplicacions que venen d'exemple per defecte en Spark, per comprovar que hi hagués una comunicació correcte entre spark i yarn, i que la configuració permet realitzar l'execució correctament. Per realitzar l'execució he utilitzat la següent comanda:

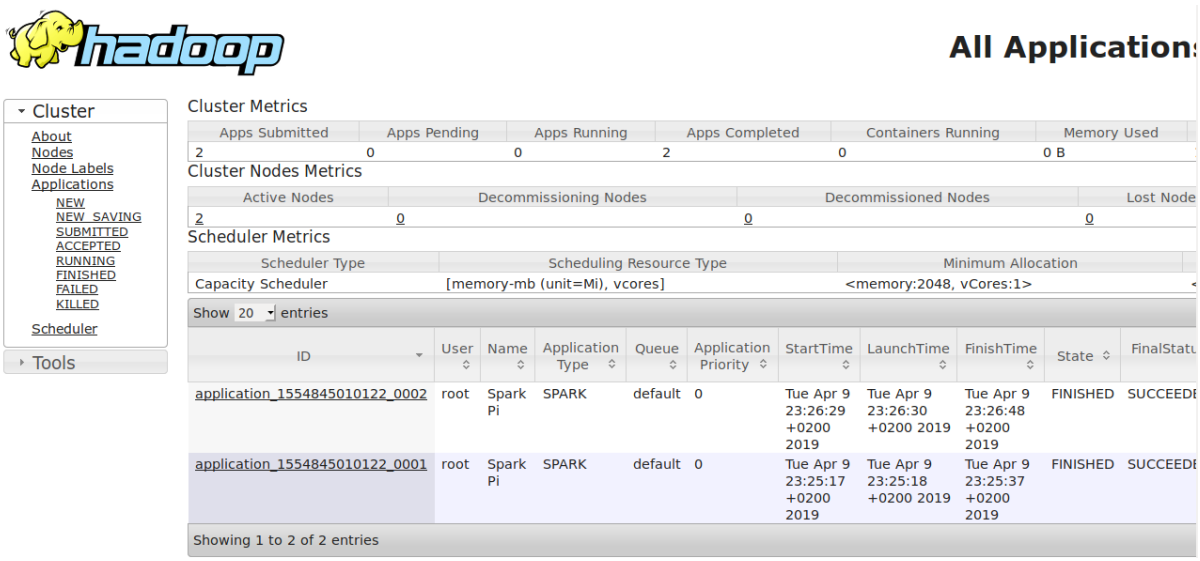
*./spark-submit --deploy-mode client *
*--class org.apache.spark.examples.SparkPi *
$SPARK_HOME/examples/jars/spark-examples_2.11-2.4.0.jar 10

Com a resultat hem obtingut correctament en número Pi, i ha trigat 2,35 segons en realitzar l'execució:



Il·lustració 27 Execució aplicació spark nombre Pi

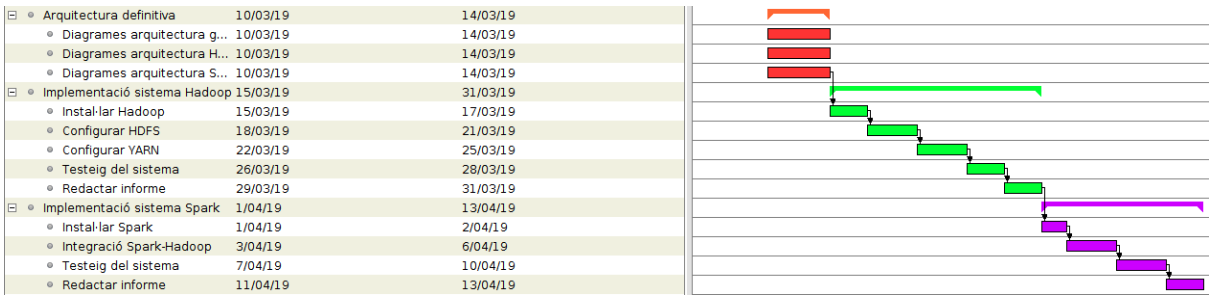
I per comprovar que s'està executant sobre YARN he mirat en la interfície del resource manager de YARN, on podem veure que s'han realitzat dues execucions correctament:



Il·lustració 28 Resource manager YARN

7 Seguiment de la planificació

La planificació prevista per dur a terme durant el primer informe de progrés és la següent:



Il·lustració 29 Planificació inicial

Durant aquest mes, la meva idea principal era aconseguir algunes imatges ja creades de Docker Hub per a realitzar després el deployment, però totes les imatges que vaig anar provant o bé donaven algun tipus d'error que era difícil de trobar, ja que no coneixia a fons com havien muntat aquella imatge, o bé no compleix amb els requisits que volia per a aquesta arquitectura. Aquest fet m'ha fet perdre molt de temps i finalment vaig decidir muntar uns Dockerfiles propis basant-me en d'altres ja existents i anar modificant o afegint allò que volia, i d'aquesta manera entendre més a fons com funcionava tot. A causa d'aquest enrederiment, no he pogut realitzar un testing complet, com ara realitzar testings de performance [13][14], provant amb diferents tamanys de dades. Per això, he re-planificat el Gantt inicial, traient temps de la implementació de bases de dades en forma de graf, per a poder realitzar un testeig més profund del sistema, en la següent entrega tindré les següents tasques:

Il·lustració 30 Re-planificació

He afegit una fase de testeig a la que he assignat uns 6 dies de duració. La resta de tasques si que s'han arribat a cumplir, he implementat un sistema Hadoop Spark (i també el mòdul de MySQL), realitzant totes les configuracions necessàries.

8 Bibliografia

[1] I. Ermilov, «Scalable Spark/HDFS Workbench using Docker» 23 Març 2016. [En línia]. Disponible: https://www.big-data-europe.eu/scalable-sparkhdfs-workbench-using-docker/. [Últim accés: 20 Març 2019].

[2] Apache, «Spark Configuration» [En línia]. Disponible: https://spark.apache.org/docs/latest/configuration.html. [Últim accés: 21 Març 2019].

[3] Ntim, «Docker-hadoop» 3 Febrer 2018. [En línia]. Disponible: https://github.com/ntim/docker-hadoop. Últim accés: 21 Març 2019].

[4] Arvind, «Creating hadoop docker image» 27 Juny 2018. [En línia]. Disponible: http://bigdatums.net/2017/11/04/creating-hadoop-docker-image/. [Últim accés: 22 Març 2019].

[5] F. H. Linode, «Install, Configure, and Run Spark on Top of a Hadoop YARN Cluster» 1 Juny 2018. [En línia]. Disponible: https://www.linode.com/docs/databases/hadoop/install-configure-run-spark-on-top-of-hadoop-yarn-cluster/. [Últim accés: 22 Març 2019].

[6] F. H. Linode, «How to Install and Set Up a 3-Node Hadoop Cluster» 1 Juny 2018. [En línia]. Disponible: https://www.linode.com/docs/databases/hadoop/how-to-install-and-set-up-hadoop-cluster/. [Últim accés: 22 Març 2019].

[7] Apache, «HDFS Architecture» 13 Novembre 2018. [En línia]. Disponible: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html. [Últim accés: 23 Març 2019].

[8] N. Vaidya, «Apache Spark Architecture – Spark Cluster Architecture Explained» 27 Febrer 2019. [En línia]. Disponible: https://www.edureka.co/blog/spark-architecture/. [Últim accés: 25 Març 2019].

[9] S. Lippens, «Hadoop 3 ports default» 16 Novembre 2018. [En línia]. Disponible: https://www.stefaanlippens.net/hadoop-3-default-ports.html. [Últim accés: 29 Març 2019].

[10] Spoddutur, «Distribution of Executors, Cores and Memory for a Spark Application running in Yarn» [En línia]. Disponible: https://spoddutur.github.io/spark-notes/distribution_of_executors_cores_and_memory_for_spark_application.html. [Últim

accés: 30 Març 2019].

[11] MySQL, «MySQL Docker official images» [En línia]. Disponible: https://hub.docker.com/_/mysql. [Últim accés: 1 Abril 2019].

[12] L. Journal, «Hadoop Tutorial» 31 Gener 2017. [En línia]. Disponible: https://www.youtube.com/watch?v=KZwb-QTmxks&list=PLkz1SCf5iB4dw3jbRo0SYCk2urRESUA3v. [Últim accés: 2 Abril 2019].

[13] Xueyuan, Brian and Yuansong, "Experimental evaluation of memory configurations of Hadoop in Docker environments," *2016 27th Irish Signals and Systems Conference (ISSC)*, Londonderry, 2016, pp. 1-6.
doi: 10.1109/ISSC.2016.7528448
keywords: {Big Data;cloud computing;parallel processing;storage management;memory configurations;Docker environments;big data analytics;QuickStart option;Apache Hadoop;clouds;Containers;Yarn;Random access memory;Memory management;Resource management;Software;Hadoop;configuration;Docker;memory},
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7528448&isnumber=7528348

[14] R. Zhang, M. Li and D. Hildebrand, "Finding the Big Data Sweet Spot: Towards Automatically Recommending Configurations for Hadoop Clusters on Docker Containers," *2015 IEEE International Conference on Cloud Engineering*, Tempe, AZ, 2015, pp. 365-368.
doi: 10.1109/IC2E.2015.101
keywords: {Big Data;cloud computing;Big Data sweet spot;configuration automatic recommendation;cloud-based analytics environment complexity;k-nearest neighbor algorithm;Hadoop;container-driven clouds;Containers;Yarn;Big data;Engines;Performance gain;Resource management;Linux},
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7092945&isnumber=7092808