

Apunts CE_5075 7.1

lloc: [Institut d'Ensenyaments a Distància de les Illes
Balears](#)
Curs: Big data aplicat
Llibre: Apunts CE_5075 7.1

Imprès per: Carlos Sanchez Recio
Data: dimarts, 25 de març 2025, 07:22

Taula de continguts

1. Introducció

2. Necessitat de Spark i característiques principals

3. Arquitectura

4. PySpark

5. RDD

5.1. Com crear un RDD

5.2. Transformacions

5.3. Accions

5.4. DoubleRDD

5.5. Persistència dels RDD

6. Creació d'un clúster Spark

6.1. Instal·lació de Docker

6.2. Creació de les imatges

6.3. Execució dels contenidors

6.4. Creació d'un quadern Jupyter

7. Spark en Google Colab

1. Introducció

En aquest lliurament veurem **Apache Spark**, un framework per a l'anàlisi de dades i l'aprenentatge automàtic.

Spark forma part de l'ecosistema Hadoop, ja que pot executar-se en un clúster Hadoop, sobre HDFS i YARN. Proporciona un motor d'execució distribuït més eficient que MapReduce, la qual cosa permet treballar en temps real. D'altra banda, a diferència de les altres eines que hem vist fins ara, Spark també pot configurar-se de manera independent, sense HDFS ni YARN.

A més del motor d'execució, Spark proporciona unes API per a diversos llenguatges, Python entre ells. I incorpora també un conjunt de llibreries per a, entre d'altres, fer anàlisi de dades i aplicar algorismes d'aprenentatge automàtic.

Tot això fa que Spark s'hagi convertit en la plataforma més utilitzada avui en dia en l'àmbit de les dades massives.

Per entendre millor el context del que suposa Spark, val la pena recuperar un requadre que vàrem introduir en el lliurament 4 i que ens parla l'evolució dels sistemes distribuïts per al tractament de dades massives, des de Hadoop MapReduce fins a Spark, passant per Pig, Hive i Impala:



En aquest curs estam veient l'evolució dels sistemes distribuïts de big data.

La primera aproximació va ser programar els treballs **MapReduce** en **Java**. Posteriorment, apareix **Pig**, que amb el seu llenguatge Pig Latin, ofereix un major nivell d'abstracció i facilita molt l'escriptura dels treballs MapReduce. Després hem vist **Hive**, que una vegada que hem definit un magatzem de dades, ens permet interactuar amb les dades emprant **SQL**, independentment de com estiguin emmagatzemades. **Impala** és la següent passa, on també interactuam mitjançant SQL, però d'una manera molt més eficient, sense utilitzar treballs MapReduce. La darrera passa en aquesta evolució és **Spark**, també molt eficient, que pot treballar tant sobre HDFS com amb altres formats d'emmagatzematge, destacant en entorns *cloud*. Spark proporciona, a més, llibreries molt potents per a l'anàlisi de dades i l'aprenentatge automàtic amb dades massives.

IMPORTANT

En aquest lliurament introduïrem el perquè Spark va aparèixer i quina és la seva arquitectura. Veurem com treballar amb els RDD, l'estructura de dades bàsica de Spark. Finalment aprendrem a crear un clúster Spark amb diversos nodes i veurem com també podem treballar sobre els servidors de Google.

Aquest lliurament té una continuació en el darrer lliurament dels dos mòduls de big data. En el mòdul de Sistemes de Big Data estudiarem en profunditat les API per a fer anàlisi de dades (**Spark SQL**) i aprenentatge automàtic (**MLlib**). D'altra banda, en Big data aplicat treballarem amb **Databricks**, una plataforma per a l'anàlisi de dades massives, basada en Spark, completament en el núvol. En concret, ho farem sobre Azure.

2. Necessitat de Spark i característiques principals

Al llarg d'aquest mòdul hem anat veient diverses eines de l'ecosistema Hadoop. Totes elles (amb l'excepció d'Impala) estan basades en MapReduce, el model de programació distribuïda que fa servir Hadoop. MapReduce permet executar de manera distribuïda i tolerant a fallades volums enormes de dades. Però al llarg del curs hem pogut observar el seu gran problema: quan hem executat petites tasques, MapReduce introdueix una latència que fa que l'execució sigui molt lenta. MapReduce està dissenyat per fer feina en mode batch, on aquesta sobrecàrrega no és un problema. Però ho fa inviable per treballar amb dades en temps real.

Ja hem comentat en altres ocasions que normalment en un entorn Hadoop, les eines d'anàlisi no fan feina directament amb les dades operacionals de l'organització. Normalment, hi ha un procés d'ingestió (ETL) que carrega un subconjunt de les dades operacionals en un magatzem de dades. Les grans empreses fan un procés de bolcat cada 24 hores, però mai podem tenir les dades actualitzades al 100%. I això, en molts d'entorns és una limitació molt important.

Així doncs, hi ha una necessitat de poder treballar en un entorn distribuït, però amb una major eficiència i donant suport a aplicacions transaccionals (OLTP), que necessiten un accés a les dades en temps real. I és per cobrir aquesta necessitat que apareix Apache Spark, un altre motor d'execució distribuït que substitueix MapReduce.



Apache Spark™ is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters.

Font: <https://spark.apache.org/>

DEFINICIÓ

Apache Spark™ és un motor multi-llenguatge per a l'execució de programes d'enginyeria de dades, ciència de dades i aprenentatge automàtic, sobre màquines de node únic o sobre clústers.

Una de les principals característiques de Spark és la de computació en memòria: les dades amb les quals treballa Spark s'emmagatzemen en una caché en la memòria dels nodes. Això evita haver d'estar accedint contínuament als discos, la qual cosa fa que el processament sigui molt més eficient, entre 10 i 100 vegades més ràpid. Com a contrapartida, es necessita una gran quantitat de RAM en els nodes.

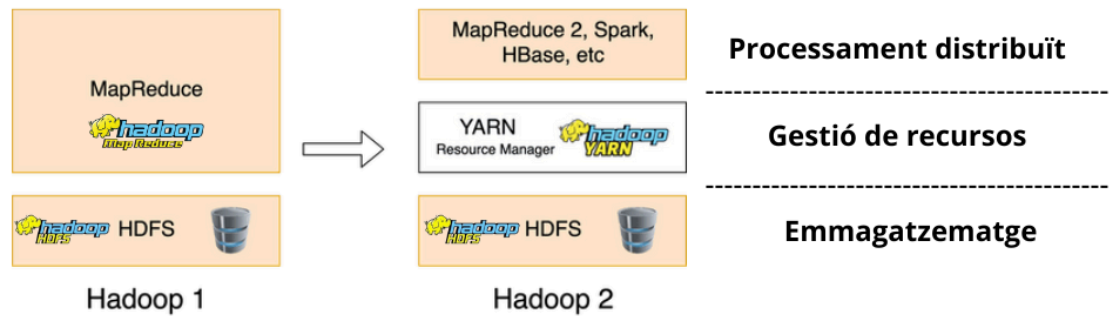
A diferència d'altres eines que hem vist anteriorment, Spark no té una interfície gràfica per interactuar amb ell, sinó que ho feim directament mitjançant diversos llenguatges de programació. Spark proporciona una API de programació per a Scala (Spark està escrit en aquest llenguatge), Java, Python i R. També té cert suport per a SQL. A més, un dels seus principals punts forts és que integra un gran conjunt de llibreries. Entre elles, tenim Spark SQL, per treballar amb dades estructurades i molt utilitzada per a l'anàlisi de dades, i MLlib, per a l'aprenentatge automàtic. Aquestes dues llibreries les estudiarem a fons en el mòdul de Sistemes de Big Data, en el lliurament 8.

Tot i que ja hem vist que la pròpia web de Spark el defineix com a un motor d'execució, sovint quan xerram de Spark ens referim no només al motor sinó a tota la plataforma o *framework* que inclou totes les API que proporciona en diversos llenguatges.

Spark va sorgir a la Universitat de Califòrnia Berkeley en 2009. En 2010 va passar a ser un projecte de codi obert amb llicència BSD (Berkeley Software Distribution) i en 2013 va ser donat a l'Apache Software Foundation, que va llicenciar-lo amb llicències Apache. Actualment, Apache Spark s'ha convertit en l'eina més emprada per a la computació distribuïda. Segons la seva web, milers de companyies fan servir Spark, incloent el 80% de les que estan a la llista del Fortune 500 (les 500 empreses més grans dels Estats Units).

Spark és un dels projectes més populars i actius de l'Apache Software Foundation i, sens dubte, el que més d'entre els de l'ecosistema Hadoop. Per fer-nos una idea, podem veure la classificació d'estrelles dels repositoris de l'Apache Software Foundation en GitHub: <https://github.com/apache?sort=stargazers>. Spark és actualment el 4t classificat, amb més de 40.000 estrelles, només per darrera d'ECharts i Superset, dos projectes sobre visualització de dades, i Dubbo, un framework de serveis web i RPC.

Ja hem avançat que Spark forma part de l'ecosistema Hadoop que hem estudiat al llarg de tot el curs. Recordem que YARN és el component, que es va introduir amb Hadoop 2.0, que permet a Hadoop suportar no només MapReduce sinó també altres motors d'execució. Recuperem una imatge del lliurament 2 per veure com encaixa Spark dins l'ecosistema Hadoop:



Imatge: Aparició de YARN en Hadoop v2

Per tant, Spark pot treballar en clústers Hadoop, sobre YARN, amb les dades emmagatzemades en el sistema de fitxers HDFS.

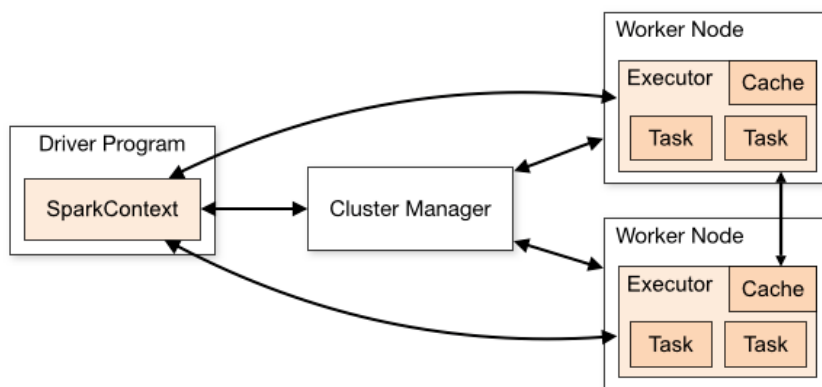
No obstant això, Spark també permet ser instal·lat i executat en màquines individuals o clústers sense YARN ni HDFS, constituint per si mateix un grup d'eines autosuficient. En aquest cas, Spark (sol xerrar-se de *Spark standalone*) treballa directament i sense intermediaris sobre les fonts de dades existents.

I també pot integrar-se en altres plataformes com ara Apache Mesos, un altre projecte Apache per gestionar clústers d'ordinadors, Kubernetes, una plataforma per a automatitzar la implementació, escalat i administració d'aplicacions en contenidors, o Amazon Elastic Compute Cloud (EC2), la plataforma de computació en el núvol d'Amazon.

3. Arquitectura

El nucli de Spark és el seu motor d'execució, que s'encarrega de la distribució i supervisió de les aplicacions. Cada tasca es distribueix entre diversos nodes de treball del clúster denominats **workers**. Al node que gestiona i coordina totes les tasques se'l denomina **mestre** o **master** o **manager**. Totes les tasques realitzades pels nodes **workers** s'agreguen al final per a produir una única sortida.

La següent imatge mostra com s'executa una aplicació en un clúster Spark:



Imatge: Arquitectura d'un clúster d'Apache Spark. Font: spark.apache.org

Per a executar una aplicació en Spark, el programa principal (anomenat *driver program*) comença creant un objecte *SparkContext*, el punt d'entrada de l'API a les funcionalitats de Spark.

A continuació, l'objecte *SparkContext* es connecta a un procés denominat *Cluster Manager*, el qual és l'encarregat d'assignar els recursos necessaris a les aplicacions. Existeixen diversos tipus de *Cluster Manager*, depenent de la configuració amb què s'executa Spark (en un clúster Spark standalone, en un clúster YARN, en un clúster Mesos o sobre Kubernetes). Així doncs, el *Cluster Manager* assigna processos *Executor* en aquells nodes *worker* del clúster que siguin necessaris.

Un *Executor* és un procés, que s'executa sobre un node esclau o *worker*, que executa càlculs i emmagatzema dades per a l'aplicació. Un *Executor* conté un espai de memòria caché per tal d'emmagatzemar les dades, així com un conjunt de slots per executar tasques.

L'aplicació es fragmentada en múltiples tasques. Una tasca (*task*) és la unitat de treball que s'envia a un *Executor*. Així, finalment, l'objecte *SparkContext* enviarà les tasques (i les dades) als *Executor* corresponents, que les executaran i li tornaran els resultats.



Podeu trobar més detalls a la documentació oficial d'Apache

Spark: <https://spark.apache.org/docs/latest/cluster-overview.html>

4. PySpark

Ja hem mencionat que Apache Spark proporciona APIs en diversos llenguatges. En concret, PySpark és l'API de Python per a Apache Spark.

El punt d'entrada a totes les funcionalitats de PySpark és la classe *SparkContext*, que representa la connexió a un clúster Spark i pot ser utilitzat, entre d'altres coses, per a crear RDDs, l'estructura de dades bàsica de Spark (ho veurem en detall en l'apartat següent). Ja hem vist que l'objecte *SparkContext* és el que es connectarà a un *Cluster Manager* i enviarà les dades (RDDs) i tasques als *Executor* dels nodes *worker* del clúster. Tot això, però, ho fa de manera transparent per al programador. És important tenir en compte que només podem tenir un únic objecte *SparkContext* actiu a la vegada.

Quan s'installa PySpark, també s'hi inclou un *shell* que permet interactuar amb el clúster, executant sentència a sentència el nostre codi. Hi accedim executant l'ordre **pyspark**. En el *shell* es proporciona directament una variable **sc** per al *SparkContext*.

```
[cloudera@quickstart ~]$ pyspark
Python 2.6.6 (r266:84292, Jul 23 2015, 15:22:56)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-11)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel).
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/lib/zookeeper/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/lib/flume-ng/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/lib/parquet/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/lib/avro/avro-tools-1.7.6-cdh5.13.0.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
23/04/26 05:10:16 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
23/04/26 05:10:16 WARN util.Utils: Your hostname, quickstart.cloudera resolves to a loopback address: 127.0.0.1; using 10.0.2.15 instead (on interface eth0)
23/04/26 05:10:16 WARN util.Utils: Set SPARK_LOCAL_IP if you need to bind to another address
Welcome to

  ____      _
 / ___|  __| | | |
 \___ \  | | | | | |
  ___) | | | | | | |
 |____|_|_|_|_|_|_|_|

version 1.6.0

Using Python version 2.6.6 (r266:84292, Jul 23 2015 15:22:56)
SparkContext available as sc, HiveContext available as sqlContext.
>>> sc
<pyspark.context.SparkContext object at 0x2cdcc50>
>>> sc.version
u'1.6.0'
>>> █
```

Imatge: Shell de PySpark

En l'apartat següent veurem com crear i interactuar amb els RDD. I ho farem emprant el *shell* de la màquina virtual Cloudera Quickstart que hem fet servir durant el curs. En els apartats 6 i 7 ja veurem la manera més habitual de treballar amb PySpark: mitjançant quaderns de Jupyter o de Colab.

5. RDD

El processament distribuït normalment es fa en múltiples fases. Així doncs, entre una fase i la següent, han de compartir dades. Si totes aquestes dades s'emmagatzemen en disc, això suposa una important sobrecàrrega d'operacions d'entrada i sortida, que fan que tot el procés sigui més lent.

Si s'aconsegueix que totes aquestes dades que comparteixen les diverses fases es guardin en memòria, tendrem un sistema molt més eficient, entre 10 i 100 vegades més ràpid. És amb aquest objectiu que Spark proporciona els **RDD, Resilient Distributed Datasets** (conjunts de dades distribuïts resilients).

Un RDD és l'estructura de dades bàsica de Spark i està format per una col·lecció d'elements distribuïda entre els nodes d'un clúster i tolerant a fallades. Un RDD presenta les següents característiques clau:

- **Avaluació peresosa (*lazy evaluation*):** totes les transformacions que anem fent sobre un RDD no es computen fins que no és imprescindible. Mentrestant, només es van guardant dins un DAG (graf acíclic dirigit). Això estalvia molt de temps de procés i espai de memòria.
- **Computació en memòria:** totes les dades del procés es mantenen en RAM, no en discos, fent el procés molt més ràpid.
- **Tolerància a fallades:** les dades són replicades entre diversos *executors* de diferents nodes *worker* del clúster. Així doncs, si un node cau, el sistema pot seguir operant.
- **Particionament:** quan feim feina amb grans volums de dades, totes les dades poden no cabre en un únic node. Spark particiona automàticament els RDD i els distribueix entre els diferents nodes.
- **Immutabilitat:** les dades d'un RDD són immutables, la qual cosa evita molts de problemes a l'hora de compartir dades entre processos.
- **Persistència:** tot i que per defecte, tal i com hem dit, les dades es gestionen en memòria, es pot configurar perquè ho facin també en disc o només en disc.



En tot aquest apartat 5 treballarem amb el shell de PySpark de la màquina virtual Cloudera Quickstart. Aquesta inclou una instal·lació de Spark, tot i que bastant antiga. En qualsevol cas, ens serà suficient per veure com treballar amb els RDD.

En l'apartat 6 veurem com configurar un clúster Spark, utilitzant les darreres versions disponibles, i utilitzarem quaderns de Jupyter per crear les nostres aplicacions.

Per últim, en l'apartat 7 veurem com podem instal·lar Spark en els servidors de Google i treballar amb quaderns de Colab.

5.1. Com crear un RDD

Tenim diverses maneres de crear un RDD. La més senzilla és parallelitzant una col·lecció. En l'exemple següent ho farem a partir d'una tupla amb els dies de la setmana:

```
dies = ("dilluns", "dimarts", "dimecres", "dijous", "divendres", "dissabte", "diumenge")
diesRDD = sc.parallelize(dies)
```

Recordem que `sc` és l'objecte *SparkContext*, que representa una connexió a un clúster Spark i que és el punt d'entrada principal a les funcionalitats de Spark. En concret, aquest objecte és el que es fa servir per crear RDD.

Si escrivim:

```
diesRDD
```

veurem que ens diu que és un RDD de tipus col·lecció paral·lela:

```
ParallelCollectionRDD[2] at parallelize at PythonRDD.scala:423
```

Amb el mètode *collect*, podem recuperar les dades que conté un RDD:

```
diesRDD.collect()
```

Retorna:

```
['dilluns', 'dimarts', 'dimecres', 'dijous', 'divendres', 'dissabte', 'diumenge']
```

Una segona manera és a partir d'un fitxer de text. Suposem que tenim un fitxer al path d'HDFS */tmp/persones.txt*, amb el contingut que hem emprat altres vegades durant aquest curs:

```
1, Joan, 50
2, Aina, 30
3, Pep, 40
```

Amb la següent sentència podem crear un RDD a partir del fitxer:

```
personesRDD = sc.textFile("/tmp/persones.txt")
```

Si ara escrim:

```
personesRDD
```

podem veure que és un RDD de tipus MapPartitions:

```
/tmp/persones.txt MapPartitionsRDD[1] at textFile at NativeMethodAccessorImpl.java:-2
```

Si ara veim les dades que conté:

```
personesRDD.collect()
[u'1, Joan, 50', u'2, Aina, 30', u'3, Pep, 40']
```

Podem veure que conté 3 posicions, cada una amb el valor d'una fila de l'arxiu.



IMPORTANT

És important tenir en compte que quan generem un RDD a partir d'un fitxer de text, no estem processant el format de cada una de les files. És a dir, no tenim columnes, només files. Si ho necessitam, ho haurem de programar nosaltres.

Això és una de les restriccions dels RDD i que, tal i com veurem en el mòdul de Sistemes de Big Data, se soluciona fent servir els tipus Dataset i DataFrame del mòdul Spark SQL.

Per últim, també és molt habitual obtenir un RDD a partir d'una transformació sobre un RDD ja existent. Per exemple, suposem que tenim un RDD a sobre el qual aplicam una transformació mitjançant el mètode map:

```
a = sc.parallelize((1,2,3,4))
b = a.map(lambda x : (x, 1))
b
```

Retorna un RDD de tipus PythonRDD:

```
PythonRDD[4] at RDD at PythonRDD.scala:43
```

Si cridam al mètode collect, podem veure que per a cada valor ha creat una tupla a la qual ha afegit el valor 1:

```
b.collect()
[(1, 1), (2, 1), (3, 1), (4, 1)]
```



Hi ha dos tipus d'operacions sobre RDD: transformacions i accions.

Una **transformació** és una operació sobre un RDD que sempre genera un nou RDD.

En canvi, una **acció** és una operació sobre un RDD que **no** genera un nou RDD.

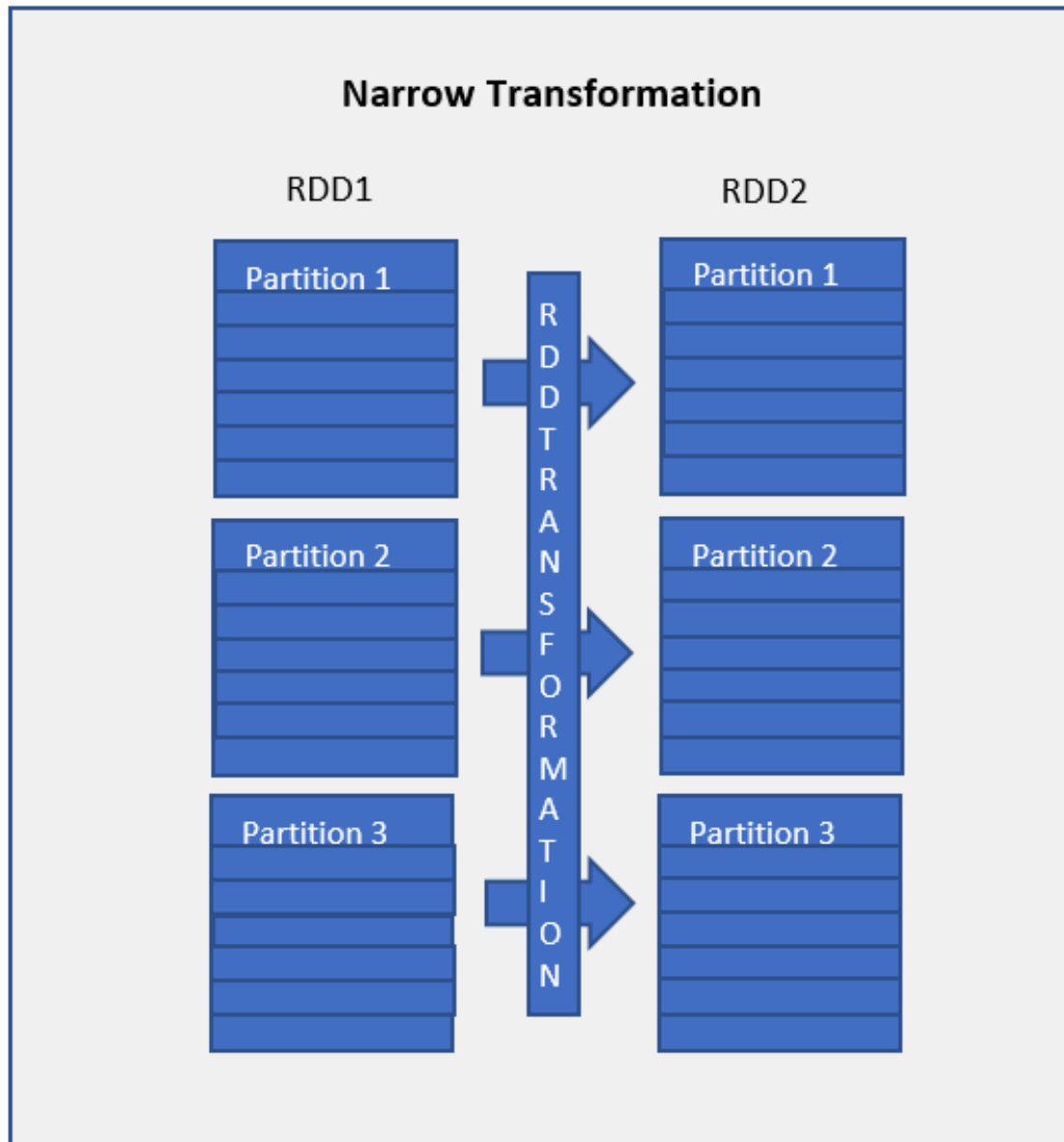
IMPORTANT

En els dos subapartats següents veurem les principals transformacions i accions sobre els RDD.

5.2. Transformacions

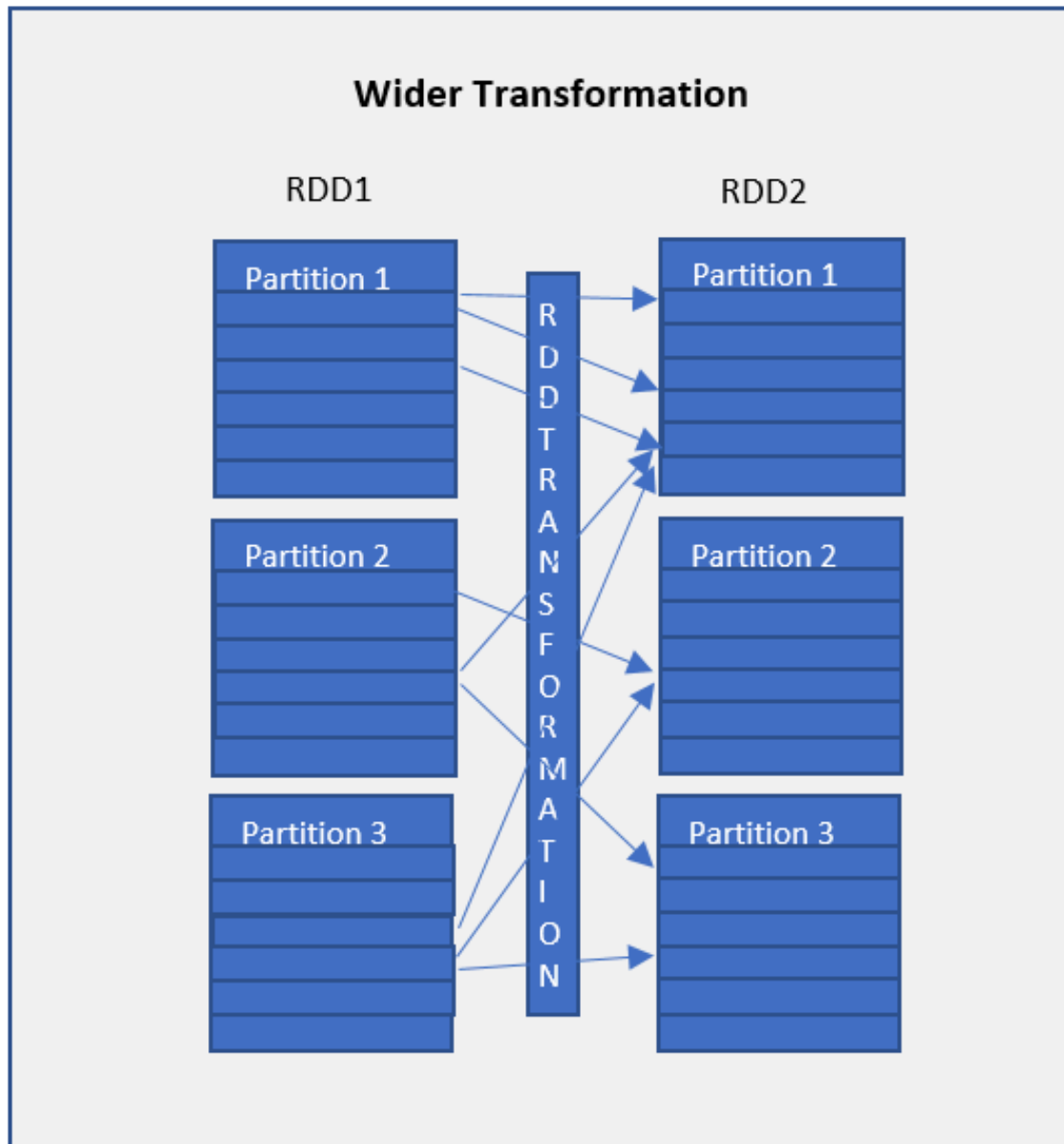
Tal i com hem dit, una transformació és una operació que retorna un nou RDD. Tenim dos tipus de transformacions, les anomenades estretes (*narrow*) i les amples (*wide*).

Si tenim un RDD amb diverses particions, les **transformacions estretes** no necessiten mesclar (*shuffle*) dades entre particions. És a dir, podem aplicar la transformació per separat a cada una de les particions i el resultat d'aplicar una transformació sobre una partició és una única partició. Les transformacions estretes més habituals són *map* i *filter*.



Imatge: Transformació estreta. Font: sparkbyexamples.com

En canvi, en les **transformacions amples**, sí que es necessària una fase de mescla, la qual cosa dona lloc a una altra distribució de les dades entre les particions. Les transformacions amples més habituals són *groupByKey* i *reduceByKey*.



Imatge: Transformació ampla. Font: sparkbyexamples.com

A continuació veurem alguns exemples de les transformacions més habituals dels dos tipus.

map (transformació estreta)

Retorna un nou RDD que es crea passant cada element del RDD font a través d'una funció (normalment una funció lambda o anònima). És l'equivalent al *map* de l'algorisme MapReduce.

```
dies = ("dilluns","dimarts","dimecres","dijous","divendres","dissabte","diumenge")
diesRDD = sc.parallelize(dies)
dieslongitudRDD = diesRDD.map(lambda dia: (dia, len (dia)))
for dia in dieslongitudRDD.collect():
    print(dia)
```

Escriu la longitud (nombre de caràcters) de cada un dels dies:

```
('dilluns', 7)
('dimarts', 7)
('dimecres', 8)
('dijous', 6)
('divendres', 9)
('dissabte', 8)
('diumenge', 8)
```

Tot i que, com hem dit, és molt habitual emprar funcions lambda, també podem definir i emprar una funció pròpia. Per exemple, el següent codi, defineix una funció *f* que es fa servir en la transformació *map* per separar les paraules d'un text llegit des d'un fitxer que conté la primera frase del Quijote:

```
def f(s):
    return s.split(" ")

quijoteRDD = sc.textFile("/tmp/quijote.txt")
paraulesRDD = quijoteRDD.map (f)
for a in paraulesRDD.collect():
    print(a)
```

El resultat és el següent:

```
[u'En', u'un', u'lugar', u'de', u'la', u'Mancha,', u'de', u'cuyo',
u'nombre', u'no', u'quiero', u'acordarme,', u'no', u'ha', u'mucho',
u'tiempo', u'que', u'vivi\u0301a', u'un', u'hidalgo', u'de', u'los',
u'de', u'lanza', u'en', u'astillero,', u'adarga', u'antigua,',
u'roci\u0301n', u'flaco', u'y', u'galgo', u'corredor.']
```



La *u* davant quan s'imprimeix un string és una cosa de Python2 (que ja no passa amb Python3). Aquí apareixen perquè s'ha utilitzat la màquina virtual de Cloudera Quickstart, que fa servir la versió 2.6.

***flatMap* (transformació estreta)**

És molt semblant a *map*, però després de fer aquest *map*, aplanar (*flattens*) el resultat, és a dir, retorna una seqüència en lloc d'un únic element.

```
dieslongitudRDD2 = diesRDD.flatMap(lambda dia: (dia, len (dia)))
for d in dieslongitudRDD2.collect():
    print(d)
```

Escriu la mateixa informació que *map*, però en la següent seqüència:

```
dilluns
7
dimarts
7
dimecres
8
dijous
6
divendres
9
dissabte
8
diumenge
8
```

***mapValues* (transformació estreta)**

La transformació *mapValues* s'aplica sobre el que s'anomena un **RDD de parelles** (*pair RDD*), un RDD que conté parelles clau-valor, on el valor és una llista d'elements. Per exemple, tenim una llista amb tipus d'animals i els animals que hi pertanyen.

```
animals = [("mamifer", ["moix", "ca", "cavall"]),\
            ("ocell", ["voltor", "flamenc"]),\
            ("peix", ["tonyina", "rajada"])]
animalsRDD = sc.parallelize(animals)
```

Amb la transformació *mapValues*, aplicam una funció a cada un dels valors d'un RDD de parelles.

En l'exemple següent, definim una funció *f* que retorna la longitud de la llista i l'utilitzam en la transformació *mapValues*:

```
def f(x): return len(x)

animalsRDD2 = animalsRDD.mapValues(f)
for c in animalsRDD2.collect():
    print(c)
```

El resultat és:

```
('mamifer', 3)
('ocell', 2)
('peix', 2)
```

filter (transformació estreta)

Retorna un nou RDD amb els elements que compleixen una certa condició, especificada mitjançant una funció.

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
numerosRDD = sc.parallelize(numeros)
parellsRDD = numerosRDD.filter(lambda n: n % 2 == 0)
for n in parellsRDD.collect():
    print(n)
```

parellsRDD conté només els números parells (els que compleixen que $n\%2$ és 0), així que escriu:

```
2
4
6
8
10
```

reduceByKey (transformació ampla)

Retorna un RDD de parelles clau-valor on els valors de cada clau s'han agregat utilitzant una funció. És l'equivalent al *reduce* de l'algorisme MapReduce. És a dir, fa una fusió (*merge*) de les claus repetides i els aplica una funció de reducció (*reduce*).

Vegem un exemple, on contem les ocurrences de paraules en una llista amb les mascotes d'un grup de 10 persones, on primer aplicam una transformació *map*, on assignam el valor 1 a cada mascota, i després una transformació *reduceByKey* on sumam les ocurrences de cada clau (cada tipus de mascota):

```
mascotes = ["moix", "ca", "ca", "ocell", "moix", "ca", "ca", "moix", "peix", "ca"]
mascotesRDD = sc.parallelize(mascotes)
mascotesRDD2 = mascotesRDD.map(lambda dia: (dia, 1)).reduceByKey(lambda x, y: x + y)
for m in mascotesRDD2.collect():
    print(m)
```

Veim que escriu per a cada tipus de mascota, el número d'ocurrences:

```
('ca', 5)
('ocell', 1)
('peix', 1)
('moix', 3)
```

groupBy (transformació ampla)

Agrupa les dades del RDD font i genera un conjunt de parelles clau-valors, on la clau és la sortida d'una funció i el valor representa tots els elements tals que la funció retorna aquesta clau.

En el següent exemple, anam a utilitzar la longitud per agrupar els valors (dies), de manera que per a cada longitud, tendrem tots els dies amb aquesta longitud:

```
longitudsRDD = diesRDD.groupBy(lambda dia: len(dia))
for x in longitudsRDD.collect():
    print(x[0], list(x[1]))
```

Vegem que, per exemple, per al 8, té una llista amb els tres dies que tenen longitud 8 (dimecres, dissabte i diumenge):

```
(8, ['dimecres', 'dissabte', 'diumenge'])
(6, ['dijous'])
(9, ['divendres'])
(7, ['dilluns', 'dimarts'])
```

groupByKey (transformació ampla)

Si tenim una llista de tuples clau-valor, agrupa els valors a partir de les claus de la llista. Això retorna un nou *pair RDD*, és a dir, un RDD de parelles clau-valor, on el valor de cada clau és una llista.

Vegem el següent exemple a partir d'una llista de parelles clau-valor:

```
parelles = [("a",1), ("b",1), ("a",3), ("c",1), ("b",2)]
parellesRDD = sc.parallelize(parelles)
grupsRDD = parellesRDD.groupByKey()
for g in grupsRDD.collect():
    print(g)
```

Això escriu:

```
('a', <pyspark.resultiterable.ResultIterable object at 0x1bddf10>)
('c', <pyspark.resultiterable.ResultIterable object at 0x1bdde90>)
('b', <pyspark.resultiterable.ResultIterable object at 0x1bf56d0>)
```

Hauríem d'emprar la transformació *mapValues()* que hem vist abans:

```
grupsRDD = parellesRDD.groupByKey().mapValues(list)
for g in grupsRDD.collect():
    print(g)
```

Això ja sí que escriu el que volem:

```
('a', [1, 3])
('c', [1])
('b', [1, 2])
```

5.3. Accions

Una acció és una operació que du a terme una computació sobre un RDD i que retorna un valor. Una acció pot retornar qualsevol cosa que no sigui un RDD (si no, seria una transformació). A diferència de les transformacions, les accions no tenen una avaluació peresosa i s'executen en el moment. A més, fa que s'executin també les transformacions prèvies (que s'han anat guardant en un DAG).

Algunes de les accions més emprades són *first*, *reduce*, *takeOrdered* i *count*. Vegem alguns exemples amb cada una d'elles.

first

Retorna el primer element del RDD.

Vegem un exemple:

```
dies = ("dilluns","dimarts","dimecres","dijous","divendres","dissabte","diumenge")
diesRDD = sc.parallelize(dies)
diesRDD.first()
print(dia)
```

Retorna *'dilluns'*.

reduce

Té una funció com a argument, que utilitza per reduir els elements del RDD d'entrada fent servir aquesta funció.

Vegem un exemple, on volem sumar els valors d'un RDD:

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
numerosRDD = sc.parallelize(numeros)
numerosRDD.reduce(lambda x,y: x+y)
```

Retorna 55, que és la suma de tots els valors.

En lloc d'una funció lambda, podem emprar una funció del llenguatge o definida per nosaltres. Per exemple:

```
numerosRDD.reduce(min)
```

```
numerosRDD.reduce(max)
```

Retornen respectivament el mínim (1) i el màxim (10) del RDD.

takeOrdered

Retorna els primers *n* elements ordenats. Per defecte es fa servir l'ordre natural, però se'n poden emprar d'altres.

Per exemple:

```
diesRDD.takeOrdered(3)
```

Retorna els 3 primers elements del RDD per ordre alfabètic:

```
['dijous', 'dilluns', 'dimarts']
```

count

Retorna el número d'elements del RDD.

Per exemple:

```
diesRDD.count()
```


Retorna 7.

5.4. DoubleRDD

Els anomenats DoubleRDDs, o RDD de números *double*, són aquells que tots els seus valors són números reals de precisió doble (*double*).

Spark proporciona una sèrie de funcions estadístiques molt útils per treballar amb aquests DoubleRDDs. Podeu trobar una llista completa d'aquestes funcions en la documentació de l'API de Java (de la darrera versió de Spark):

<https://spark.apache.org/docs/latest/api/java/org/apache/spark/rdd/DoubleRDDFunctions.html>

Vegem alguns exemples:

```
>>> numeros = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]
>>> numerosRDD = sc.parallelize(numeros)
>>> numerosRDD.mean()
5.5
>>> numerosRDD.variance()
8.25
>>> numerosRDD.sum()
55
>>> numerosRDD.stdev()
2.8722813232690143
>>> numerosRDD.stats()
(count: 10, mean: 5.5, stdev: 2.87228132327, max: 10, min: 1)
```



També es poden emprar aquestes funcions si el RDD conté números enters, se'n farà una conversió automàtica.

5.5. Persistència dels RDD

Com ja sabem, les transformacions Spark són peresoses (*lazy*). Això vol dir que si cream un RDD a partir d'un fitxer, per exemple amb:

```
rdd = sc.textFile("/tmp/dades.txt")
```

En realitat *rdd* no conté dades, només una estructura que diu que les seves dades s'han de llegir des del fitxer */tmp/persones.txt*. Si després aplicam una sèrie de transformacions passa el mateix: les dades segueixen estant en el fitxer.

Quan cridam una acció, per exemple amb *rdd.count()*, en aquest moment sí que es llegeixen les dades, es computen les transformacions i s'obté un valor. Però si es torna a cridar *rdd.count()*, tot aquest procés s'haurà de repetir: s'hauran de tornar a llegir les dades del fitxer i aplicar les transformacions. És per això que després de fer una transformació, especialment quan treballem amb grans volums de dades, és convenient que es facin persistents aquestes dades, per tal de no haver d'estar rellegint les dades d'entrada i recomputant les transformacions.

Tenim dos mètodes per fer-ho.

El primer i més senzill és **cache()**. El que fa és que cacheja les dades en memòria. Per exemple, aquí passam les dades a caché una vegada hem aplicat una transformació *map* i una altra *reduceByKey*:

```
rdd = sc.textFile("/tmp/dades.txt")
```

```
rdd = rdd.map (lambda word: (word, 1)).reduceByKey (lambda x, y: x + y)
```

```
rdd.cache()
```

Quan ja no caben més dades en la caché, se segueix una política LRU (Least Recently Used), és a dir, s'eliminen de la caché les dades que no s'han utilitzat des de fa més temps. Però si tot el RDD no cap dins la caché, s'haurà de rellegir i tornar a computar les transformacions.

La segona manera de fer persistents les dades és mitjançant el mètode **persist()**. Aquest mètode ens dona un major control ja que permet especificar un entre diversos nivells d'emmagatzematge:

- **MEMORY_ONLY**: les dades es guarden en memòria com a objectes deserialitzats de Java en la JVM. Si el RDD no hi cap, algunes particions no es cachejaran i hauran de ser recalculades on-the-fly quan sigui necessari. Aquest és el nivell per defecte
- **MEMORY_AND_DISK**: les dades també es guarden com a objectes deserialitzats de Java en memòria. Però si el RDD no hi cap, les particions que no hi caben, es guarden en disc.
- **MEMORY_ONLY_SER**: les dades s'emmagatzemen en memòria com a objectes serialitzats de Java (un array de bytes per partició). En general, això ocupa menys espai que deserialitzat, però té un cost de CPU.
- **DISK_ONLY**: totes les particions s'emmagatzemen en disc
- **MEMORY_ONLY_2**: el mateix que **MEMORY_ONLY**, però replicant cada partició en dos nodes del clúster
- **MEMORY_AND_DISK_2**: el mateix que **MEMORY_AND_DISK**, però replicant cada partició en dos nodes del clúster
- **OFF_HEAP**: semblant a **MEMORY_ONLY_SER** però les dades s'emmagatzemen en la memòria off-heap, és a dir, en la memòria del sistema operatiu, fora de la JVM. Això requereix habilitar la memòria off-heap.

Per exemple:

```
rdd = sc.textFile("/tmp/dades.txt")  
rdd = rdd.map (lambda word: (word, 1)).reduceByKey (lambda x, y: x + y)  
rdd.persist(MEMORY_ONLY_SER)
```

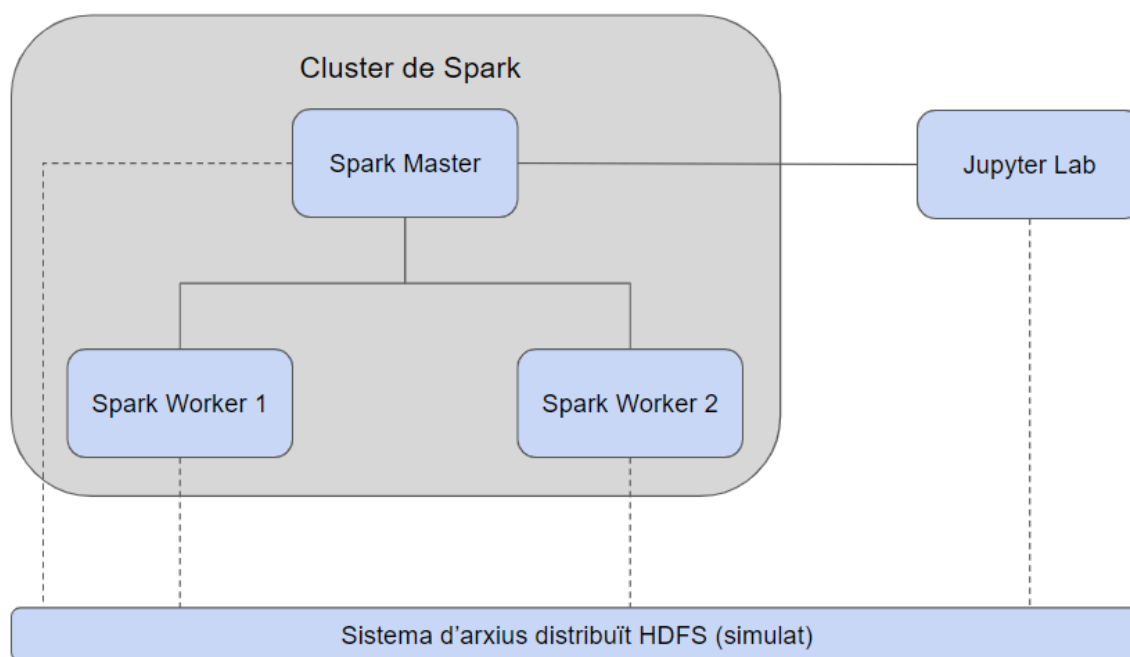
Per últim, també tenim un mètode ***unpersist()*** per a descartar dades persistents.

6. Creació d'un clúster Spark

En aquest apartat anam a crear el nostre propi clúster de Spark amb un node mestre i dos workers. A més, afegirem un altre node on instal·larem JupyterLab i ens permetrà executar quaderns Jupyter amb el nostre codi emprant PySpark.

L'objectiu no és muntar un clúster per a un entorn de producció, sinó veure com funciona un clúster amb algunes petites proves. En el primer lliurament del curs vàrem configurar un clúster de Hadoop sobre 3 màquines virtuals de VirtualBox. Ara, per simplificar la instal·lació i reduir els requeriments de processador i memòria, farem feina amb contenidors Docker. A més, tot i que l'habitual és que el clúster estigui configurat sobre HDFS, també per simplificar la instal·lació, nosaltres ho simularem mitjançant un directori compartit entre els nodes.

Així doncs, aquesta és l'arquitectura que volem implementar:



Imatge: Arquitectura del clúster Spark

Els nodes fan servir els següents ports:

- El node mestre executa el procés Spark Master en el port 7077 (al qual es connectaran els workers) i la interfície web en el port 8080
- Els nodes worker tenen la interfície web del Spark Worker en el port 8081
- El node de JupyterLab té la interfície web de JupyterPort 8888

A més, mapejam aquests ports de manera que localhost (o la IP o nom de la màquina) fa públiques les següents interfícies:

- port 7077: procés Spark Master
- port 8080: interfície web del Spark Master
- port 8081: interfície web del Spark Worker 1
- port 8082: interfície web del Spark Worker 2
- port 8888: interfície web de JupyterLab

Aquest clúster el muntarem sobre una màquina amb la darrera versió estable d'Ubuntu Desktop, actualment la 24.04.2 LTS (tot i que no hi haurà problemes amb qualsevol altra 22 o 24 LTS). Podeu fer-ho sobre una màquina física o virtual. En qualsevol cas, és recomanable que tengui almenys 4 cores i 4 GB de RAM.

6.1. Instal·lació de Docker

Docker ofereix als desenvolupadors una eina econòmica i fiable per a poder crear, distribuir i desplegar aplicacions distribuïdes i complexes en una gran varietat d'entorns, proporcionant una capa d'abstracció gràcies a la qual no és necessari tenir en compte el sistema operatiu que executa per sota d'aquest. En els darrers anys, Docker s'ha convertit en la tecnologia de contenidors més utilitzada, ja que implementa una API que permet executar contenidors lleugers de manera aïllada.



Un **contenedor** és una unitat de programari que inclou tot el necessari perquè una aplicació s'executi, incloent-hi el codi font, les llibreries, i la resta de dependències necessàries per a la seva correcta execució en qualsevol entorn.

D'aquesta manera, una vegada que tenim una aplicació configurada com a un contenidor Docker, aquest contenidor es pot executar en qualsevol entorn (només cal que aquest tenguí Docker instal·lat). Pot ser, per exemple, una màquina amb Ubuntu o un servei en el núvol d' AWS. És per tant una forma d'empaquetar una aplicació que permet una gran escalabilitat d'una manera senzilla.

IMPORTANT

No entrarem en gaire detalls sobre Docker. Podeu trobar molta més informació en la seva web: <https://www.docker.com/>

El primer que farem per instal·lar Docker en el nostre Ubuntu és actualitzar els repositoris:

```
sudo apt update
```

Si és necessari (ens ho indicarà quan fem l'update), també hauré de fer un upgrade:

```
sudo apt upgrade
```

A continuació hem d'instal·lar algunes dependències necessàries:

```
sudo apt install apt-transport-https ca-certificates curl software-properties-common -y
```

Ara hem de passar a usuari root (*sudo su*) per afegir la GPG Key per poder descarregar Docker des del repositori oficial (ens donarà un *warning* que podem ignorar):

```
sudo su
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | apt-key add -
```

Encara com a root, afegim el repositori de Docker a la llista de repositoris:

```
add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu focal stable"
```

Tornem a l'usuari sense privilegis (*exit*) i abans d'instal·lar Docker amb apt, és necessari tornar a actualitzar la llista de repositoris:

```
exit
sudo apt update
```

I ara ja podem instal·lar Docker i Docker Compose:

```
sudo apt install docker-ce docker-compose -y
```

Només ens queda activar el servei:

```
systemctl start docker
```

Per comprovar l'estat del servei:

```
systemctl status docker
```

```
toni@ubuntu0:~$ systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
   Active: active (running) since Tue 2023-04-25 12:40:33 CEST; 41s ago
 TriggeredBy: ● docker.socket
    Docs: https://docs.docker.com
   Main PID: 5267 (dockerd)
      Tasks: 8
     Memory: 27.9M
        CPU: 493ms
    CGroup: /system.slice/docker.service
            └─5267 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock

abr 25 12:40:32 ubuntu0 dockerd[5267]: time="2023-04-25T12:40:32.381349023+02:00" level=info msg="[core] [Channel #4] Channel Conne>
abr 25 12:40:32 ubuntu0 dockerd[5267]: time="2023-04-25T12:40:32.381821065+02:00" level=info msg="[core] [Channel #4] SubChannel #5]>
abr 25 12:40:32 ubuntu0 dockerd[5267]: time="2023-04-25T12:40:32.381942611+02:00" level=info msg="[core] [Channel #4] Channel Conne>
abr 25 12:40:32 ubuntu0 dockerd[5267]: time="2023-04-25T12:40:32.545391251+02:00" level=info msg="Loading containers: start."
abr 25 12:40:32 ubuntu0 dockerd[5267]: time="2023-04-25T12:40:32.989966007+02:00" level=info msg="Loading containers: done."
abr 25 12:40:33 ubuntu0 dockerd[5267]: time="2023-04-25T12:40:33.059885440+02:00" level=info msg="Docker daemon" commit=c9c331 gra>
abr 25 12:40:33 ubuntu0 dockerd[5267]: time="2023-04-25T12:40:33.060205432+02:00" level=info msg="Daemon has completed initializat>
abr 25 12:40:33 ubuntu0 dockerd[5267]: time="2023-04-25T12:40:33.112401663+02:00" level=info msg="[core] [Server #7] Server create>
abr 25 12:40:33 ubuntu0 systemd[1]: Started Docker Application Container Engine.
abr 25 12:40:33 ubuntu0 dockerd[5267]: time="2023-04-25T12:40:33.141971318+02:00" level=info msg="API listen on /run/docker.sock"
```

Imatge: Comprovació de l'estat de Docker

Per acabar, per comprovar que tot està funcionant correctament, executam la següent ordre que descarrega la imatge de prova *hello-world* i l'executa en un contenidor:

```
sudo docker run hello-world
```

Aquest és el resultat:

```
toni@ubuntu0:~$ sudo docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:4e83453afed1b4fa1a3500525091dbfca6ce1e66903fd4c01ff015dbcb1ba33e
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/
```

Imatge: Resultat d'executar el contenidor hello-world

6.2. Creació de les imatges

Ara que ja tenim Docker configurat en la nostra màquina, ja estam preparats per a crear les imatges i els contenidors que ens permetran definir el nostre clúster de Spark. Una **imatge** en Docker és un arxiu estàtic, compost per múltiples capes, i que s'utilitza com a plantilla base per a crear un contenidor. Els **contenidors** són instàncies de les imatges. De fet, la principal diferència entre els contenidors i les imatges és que els contenidors són imatges que tenen una capa d'escriptura en la qual es van afegint els canvis ocorreguts una vegada s'executa el contenidor.

En el nostre cas, volem crear 3 imatges diferents, una per al node mestre, una altra que servirà per als dos nodes worker, i una tercera per al node amb JupyterLab.

Totes tres parteixen d'una imatge base anomenada *openjdk:8-jre-slim*, que té una distribució de Debian molt lleugera amb l'Open JDK 8. Sobre aquesta imatge base, cada imatge instal·larà les dependències necessàries en cada cas: Python 3 (per poder executar PySpark), Spark, Hadoop i JupyterLab. Farem feina amb les darreres versions estables: 3.5.5 de Spark i 4.3.6 de JupyterLab.

Les imatges es configuren mitjançant un arxiu de text. Podeu descarregar-les des de git-hub:

- **spark-master.Dockerfile:** <https://github.com/tnavarrete-iedib/bigdata-24-25/blob/main/docker/spark-master.Dockerfile>
- **spark-worker.Dockerfile:** <https://github.com/tnavarrete-iedib/bigdata-24-25/blob/main/docker/spark-worker.Dockerfile>
- **jupyterlab.Dockerfile:** <https://github.com/tnavarrete-iedib/bigdata-24-25/blob/main/docker/jupyterlab.Dockerfile>

Guardarem els tres fitxers en un directori *cluster* a l'arrel del *home* del nostre usuari (/home/toni/cluster en el meu cas).

Si mirau el contingut de cada fitxer, trobareu els comentaris que expliquen què fa cada cosa. El més important és fixar-se en el següent:

- En tots tres, partim de la imatge base *openjdk:8-jre-slim*
- En la imatge per al master configuram que el Spark Master s'executa en el port 7077 i la interfície web en el port 8080.
- En la imatge per als workers configuram que el Spark Master al qual s'han de connectar es troba a spark-master:7077 i que la interfície web està en el port 8081
- En la imatge per al node amb JupyterLab configuram que la seva interfície web s'executa al port 8888

Ara podem descarregar el fitxer **build.sh** (<https://github.com/tnavarrete-iedib/bigdata-24-25/blob/main/docker/build.sh>) que conté un script de bash que munta les tres imatges definides anteriorment. Quan les munta, Docker descarrega la imatge base que hem triat, i seguint el guió que es defineix en els fitxers Dockerfile anteriors, instal·la totes les dependències i configuracions especificades.

Executem el fitxer build.sh, des del directori *cluster*:

```
sudo bash build.sh
```

Un cop hagi acabat, podem comprovar que s'han muntat correctament les tres imatges (a més de *hello-world* que ja havíem descarregat en l'apartat anterior):

```
sudo docker images
```

O bé:

```
sudo docker image ls
```

```
toni@ubuntu1:~/cluster$ sudo docker image ls
```

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|--------------|--------|--------------|----------------|--------|
| jupyterlab | latest | 6e77b106283b | 10 minutes ago | 2.18GB |
| spark-worker | latest | 4b99345a5b13 | 2 hours ago | 1.46GB |
| spark-master | latest | be198b74b837 | 4 hours ago | 1.46GB |

Imatge: *Imatges Docker disponibles*

6.3. Execució dels contenidors

Una vegada que ja tenim creades les imatges, podem definir els 4 contenidors, un per a cada node del nostre clúster. Per fer-ho, necessitam descarregar el fitxer **docker-compose.yml** (<https://github.com/tnavarrete-iedib/bigdata-24-25/blob/main/docker/docker-compose.yml>).

En aquest fitxer configuram el contenidor *spark-master* en base a la imatge *spark-master* i exposam els ports 7077 i 8080. A continuació, configuram els contenidors *spark-worker-1* i *spark-worker-2* a partir de la imatge *spark-worker*. Fixau-vos que en el *spark-worker-2* hem canviat el port on exposam la interfície web del 8081 al 8082. És a dir, si accedim a localhost:8082 (o el nom o IP de la màquina on tenim Docker instal·lat), internament dirigirà la petició a *spark-worker-2:8081* (port 8081 del contenidor *spark-worker-2*). Per últim, també configuram el contenidor *jupyterlab* a partir de la imatge *jupyterlab* i exposam el port 8888. Fixau-vos també que hem definit un volum anomenat *hdfs-simulat* que referencia */opt/workspace*.

Ara, des del directori *cluster* ja podem executar els contenidors, mitjançant l'ordre següent:

```
sudo docker-compose up
```

IMPORTANT

Quan aturem la màquina i la tornem a posar en marxa, les imatges ja estan creades, així que l'únic que haurém de fer és tornar a executar l'ordre

```
sudo docker-compose up
```

per posar en marxa els contenidors.

Podem comprovar que s'han desplegat els contenidors amb l'ordre següent:

```
sudo docker container ls
```

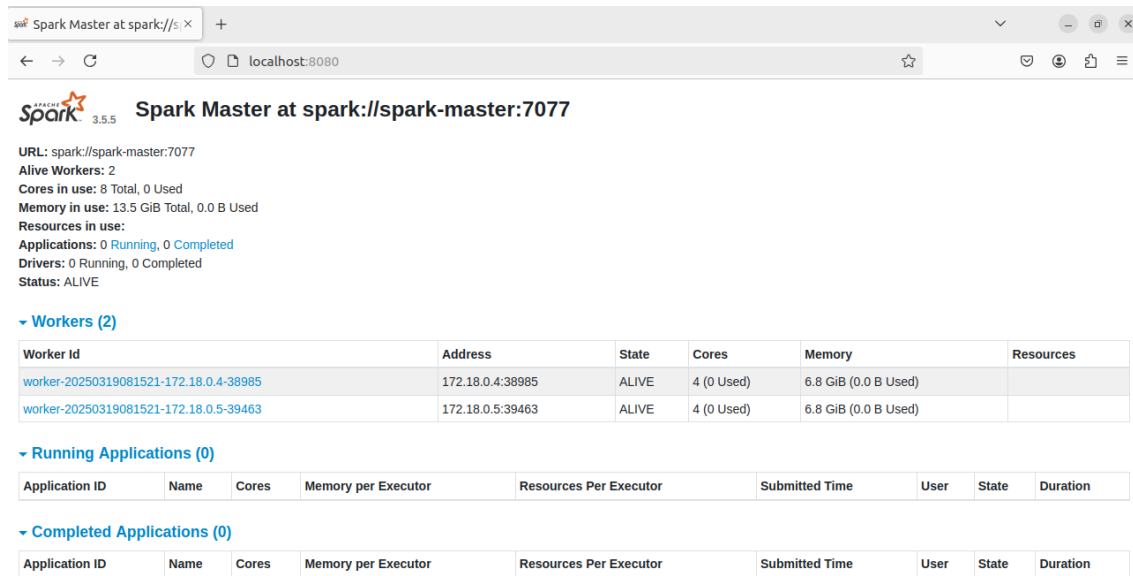
Hi han d'aparèixer els quatre contenidors, un de la imatge *spark-master*, dos de *spark-worker* i un altre *jupyterlab*:

```
toni@ubuntu1:~/cluster$ sudo docker container ls
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS
e0ba49f57f28   spark-worker   "/bin/sh -c 'bin/spa..." 2 minutes ago Up 2 minutes  0.0.0.0:8082->8081/tcp, [::]:8082->8081/tcp
c996928eb891   spark-worker   "/bin/sh -c 'bin/spa..." 2 minutes ago Up 2 minutes  0.0.0.0:8081->8081/tcp, [::]:8081->8081/tcp
10a22c42b389   spark-master   "/bin/sh -c 'bin/spa..." 2 minutes ago Up 2 minutes  0.0.0.0:7077->7077/tcp, [::]:7077->7077/tcp, 0.0.0.0:8080->8080/tcp, [::]:8080->8080/tcp
0d86cbbac5c3   jupyterlab     "/bin/sh -c 'jupyter..." 2 minutes ago Up 2 minutes  0.0.0.0:8888->8888/tcp, [::]:8888->8888/tcp
```

Imatge: Llista de contenidors desplegats

Ara, des del navegador web podem accedir a la interfície web de cada node.

Tenim la interfície web del node mestre a <http://localhost:8080>. Podem observar que tenim dos nodes workers definits (a 172.18.0.4 i 172.18.0.5).



Spark Master at spark://spark-master:7077

URL: spark://spark-master:7077
 Alive Workers: 2
 Cores in use: 8 Total, 0 Used
 Memory in use: 13.5 GiB Total, 0.0 B Used
 Resources in use:
 Applications: 0 Running, 0 Completed
 Drivers: 0 Running, 0 Completed
 Status: ALIVE

▼ Workers (2)

| Worker Id | Address | State | Cores | Memory | Resources |
|--|------------------|-------|------------|----------------------|-----------|
| worker-20250319081521-172.18.0.4-38985 | 172.18.0.4:38985 | ALIVE | 4 (0 Used) | 6.8 GiB (0.0 B Used) | |
| worker-20250319081521-172.18.0.5-39463 | 172.18.0.5:39463 | ALIVE | 4 (0 Used) | 6.8 GiB (0.0 B Used) | |

▼ Running Applications (0)

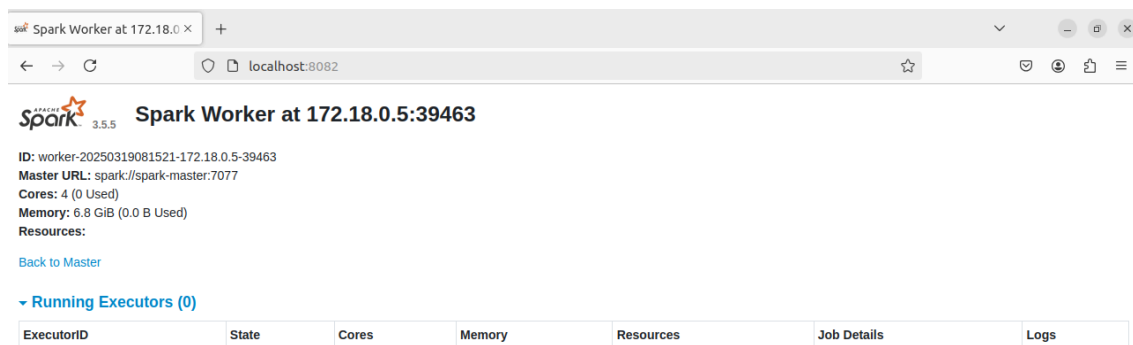
| Application ID | Name | Cores | Memory per Executor | Resources Per Executor | Submitted Time | User | State | Duration |
|----------------|------|-------|---------------------|------------------------|----------------|------|-------|----------|
|----------------|------|-------|---------------------|------------------------|----------------|------|-------|----------|

▼ Completed Applications (0)

| Application ID | Name | Cores | Memory per Executor | Resources Per Executor | Submitted Time | User | State | Duration |
|----------------|------|-------|---------------------|------------------------|----------------|------|-------|----------|
|----------------|------|-------|---------------------|------------------------|----------------|------|-------|----------|

Imatge: Interfície web del node mestre

Per veure la interfície web dels dos nodes worker, podem fer clic sobre qualsevol d'ells a la interfície del mestre, o anar directament a <http://localhost:8081> (o <http://172.18.0.4:8081>) per al worker 1 o <http://localhost:8082> (o <http://172.18.0.5:8081>) per al worker 2.



Spark Worker at 172.18.0.5:39463

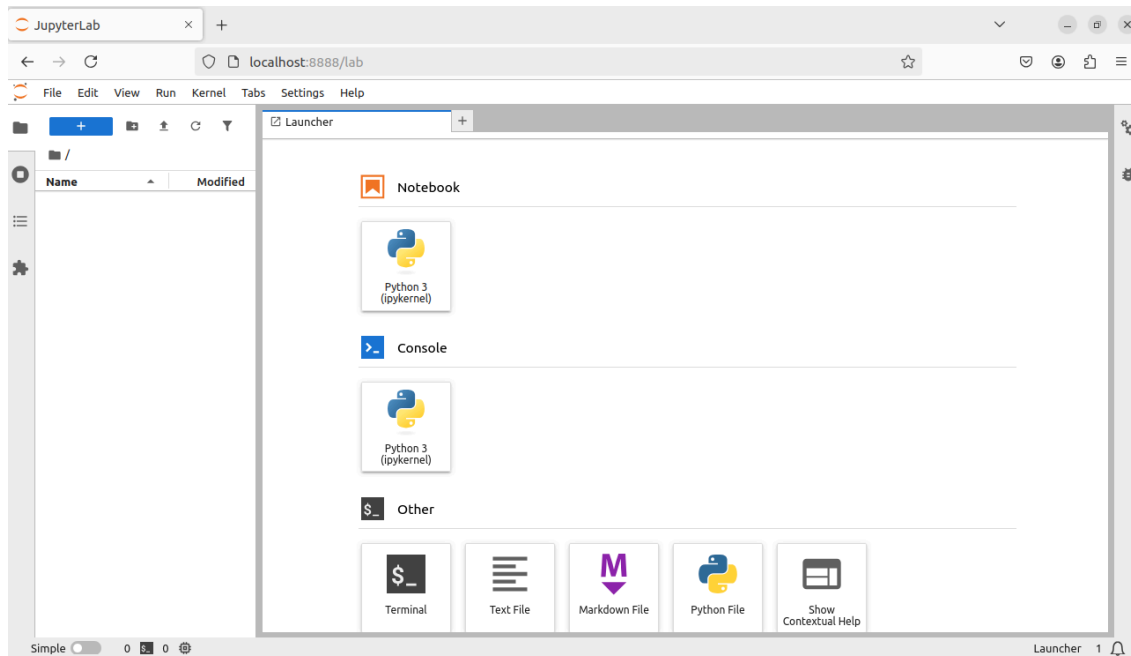
ID: worker-20250319081521-172.18.0.5-39463
 Master URL: spark://spark-master:7077
 Cores: 4 (0 Used)
 Memory: 6.8 GiB (0.0 B Used)
 Resources:
[Back to Master](#)

▼ Running Executors (0)

| ExecutorID | State | Cores | Memory | Resources | Job Details | Logs |
|------------|-------|-------|--------|-----------|-------------|------|
|------------|-------|-------|--------|-----------|-------------|------|

Imatge: Interfície web del node worker 2

Finalment, podem accedir també a la interfície web de JupyterLab des de <http://localhost:8888>:



Imatge: *Interfície web de JupyterLab*

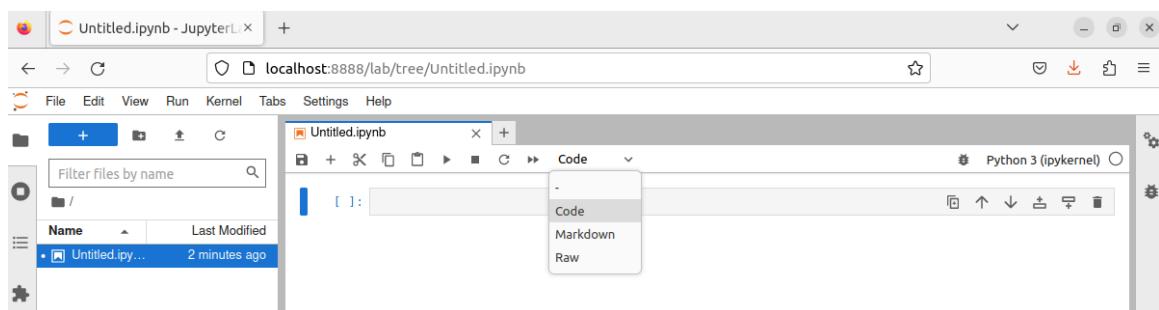
6.4. Creació d'un quadern Jupyter

Des de la interfície web de JupyterLab, feim clic al botó Python 3 (ipykernel) del bloc Notebook per crear un nou quadern Jupyter.



Imatge: Botó per crear un nou quadern Jupyter

Fet això ens apareix la típica interfície d'un quadern de Jupyter, on podem escriure i executar cel·les de codi en Python 3 o de text amb Markdown.



Imatge: Quadern Jupyter en blanc

Podem escriure el nostre codi de la mateixa manera que ho hem fet anteriorment amb el shell de PySpark. La única diferència és que aquí no tenim l'objecte `sc` (*SparkContext*), sinó que l'hem de definir nosaltres. Instanciam la classe *SparkContext* passant-li dos arguments: la ubicació del Spark Master (*spark://spark-master:7077*) i el nom de l'aplicació (li direm *Spark des de JupyterLab*):

```
from pyspark.context import SparkContext
sc = SparkContext('spark://spark-master:7077', 'Spark des de JupyterLab')
sc
```

És probable que ens doni un warning que podem ignorar. Això crea el nostre objecte `sc` com a punt d'entrada a PySpark:

SparkContext

Spark UI

| | |
|----------------|---------------------------|
| Version | v3.5.5 |
| Master | spark://spark-master:7077 |
| AppName | Spark des de JupyterLab |

Imatge: Objecte *SparkContext*

Si tornem a la interfície web del node mestre, podem veure que ara tenim una aplicació anomenada *Spark des de JupyterLab*:

Spark Master at spark://spark-master:7077

URL: spark://spark-master:7077
 Alive Workers: 2
 Cores in use: 8 Total, 8 Used
 Memory in use: 13.5 GiB Total, 2.0 GiB Used
 Resources in use:
 Applications: 1 Running, 1 Completed
 Drivers: 0 Running, 0 Completed
 Status: ALIVE

▼ Workers (2)

| Worker Id | Address | State | Cores | Memory | Resources |
|--|------------------|-------|------------|---------------------------|-----------|
| worker-20250319081521-172.18.0.4-38985 | 172.18.0.4:38985 | ALIVE | 4 (4 Used) | 6.8 GiB (1024.0 MiB Used) | |
| worker-20250319081521-172.18.0.5-39463 | 172.18.0.5:39463 | ALIVE | 4 (4 Used) | 6.8 GiB (1024.0 MiB Used) | |

▼ Running Applications (1)

| Application ID | Name | Cores | Memory per Executor | Resources Per Executor | Submitted Time | User | State | Duration |
|-------------------------|-------------------------|-------|---------------------|------------------------|---------------------|------|---------|----------|
| app-20250319082702-0001 | Spark des de JupyterLab | 8 | 1024.0 MiB | | 2025/03/19 08:27:02 | root | RUNNING | 2.2 min |

Imatge: Interfície web del node mestre executant l'aplicació

Podem fer clic sobre el ID de l'aplicació i en veurem més detalls. En concret, podem veure que s'està executant als dos workers, dedicant 4 cores i 1024 MiB cadascun.

Application: Spark des de JupyterLab

ID: app-20250319082702-0001
 Name: Spark des de JupyterLab
 User: root
 Cores: Unlimited (8 granted)
 Executor Limit: Unlimited (2 granted)
 Executor Memory - Default Resource Profile: 1024.0 MiB
 Executor Resources - Default Resource Profile:
 Submit Date: 2025/03/19 08:27:02
 State: RUNNING
[Application Detail UI](#)

▼ Executor Summary (2)

| ExecutorID | Worker | Cores | Memory | Resource Profile Id | Resources | State | Logs |
|------------|--|-------|--------|---------------------|-----------|---------|-------------------------------|
| 1 | worker-20250319081521-172.18.0.4-38985 | 4 | 1024 | 0 | | RUNNING | stdout stderr |
| 0 | worker-20250319081521-172.18.0.5-39463 | 4 | 1024 | 0 | | RUNNING | stdout stderr |

Imatge: Detalls de l'aplicació "Spark des de JupyterLab"

I si entrem en la interfície web de qualsevol dels workers, veurem que també està executant un executor per a la nostra aplicació:

Spark Worker at 172.18.0.5:39463

ID: worker-20250319081521-172.18.0.5-39463
 Master URL: spark://spark-master:7077
 Cores: 4 (4 Used)
 Memory: 6.8 GiB (1024.0 MiB Used)
 Resources:
[Back to Master](#)


▼ Running Executors (1)

| ExecutorID | State | Cores | Memory | Resources | Job Details | Logs |
|------------|---------|-------|------------|-----------|--|-------------------------------|
| 0 | RUNNING | 4 | 1024.0 MiB | | ID: app-20250319082702-0001 Name: Spark des de JupyterLab User: root | stdout stderr |

Imatge: Interfície web del node worker 2 executant l'aplicació

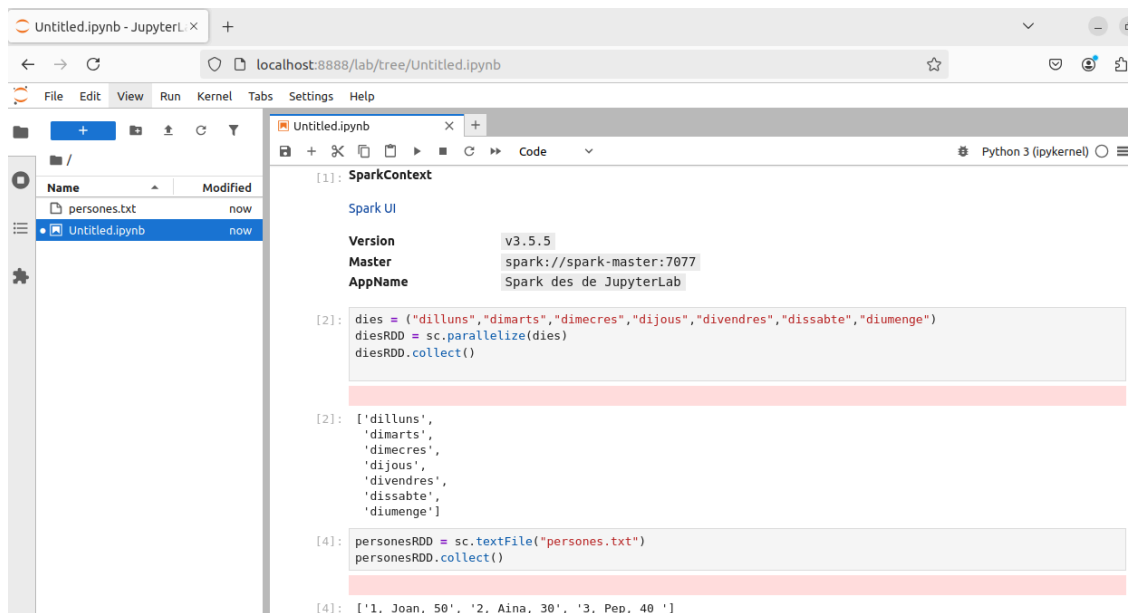
I podem seguir escrivint el nostre codi PySpark. Per exemple, podem crear un RDD a partir d'una tupla:

```
dies = ("dilluns", "dimarts", "dimecres", "dijous", "divendres", "dissabte", "diumenge")
diesRDD = sc.parallelize(dies)
diesRDD.collect()
```


Podem també pujar un fitxer (persones.txt) amb la icona  i generar un RDD a partir d'aquest fitxer de la manera que varem fer a l'apartat 5.1:

```
personesRDD = sc.textFile("persones.txt")
personesRDD.collect()
```

La següent imatge mostra el resultat:



Imatge: Vista del quadern Jupyter



Podeu descarregar aquest quadern Jupyter des de https://github.com/tnavarrete-iedib/bigdata-24-25/blob/main/docker/spark_jupyter.ipynb

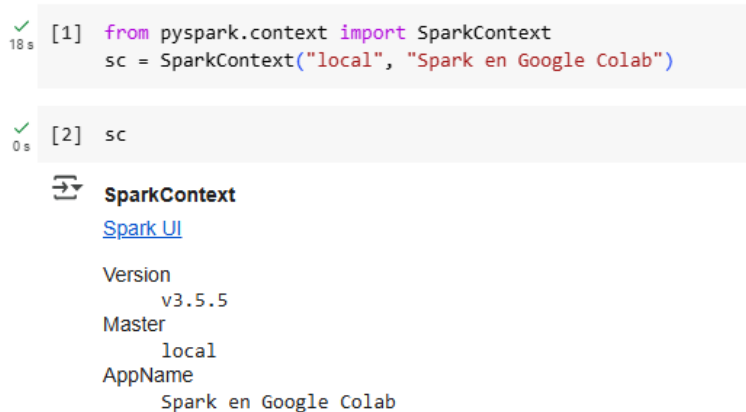
7. Spark en Google Colab

Per fer les nostres proves, serà més senzill fer-ho sobre els servidors de Google mitjançant Google Colab. En aquest apartat anam a veure com crear un quadern que ens permetrà treballar amb PySpark sobre els servidors de Google. Convé aclarir que un quadern Colab no necessàriament s'ha d'executar sobre els servidors de Google, també es podria configurar perquè es connectàs a un clúster Spark existent (amb l'opció *Conectarse a un entorno de ejecución local*).

Actualment, l'entorn de Colab ja té instal·lat Spark i PySpark, amb la qual cosa no ens hem de preocupar de fer-hi cap instal·lació (en edicions anteriors del curs sí que era necessari). Aixíí que ja podem començar a escriure directament el nostre codi. Igual que hem vist amb els quaderns de Jupyter, haurem de començar configurant el nostre objecte *SparkContext* que ens serveix com a punt d'entrada. Li passam com a arguments la ubicació del node mestre i el nom que li donam a l'aplicació:

```
from pyspark.context import SparkContext
sc = SparkContext("local", "Spark en Google Colab")
```

Vegem el resultat:



Imatge: Objecte *SparkContext* en un quadern de Google Colab

Ara farem les mateixes operacions que en el quadern de Jupyter. Començarem creant un RDD a partir d'una tupla:

```
dies = ("dilluns", "dimarts", "dimecres", "dijous", "divendres", "dissabte", "diumenge")
diesRDD = sc.parallelize(dies)
diesRDD.collect()
```

A continuació, volem carregar les dades del fitxer *persones.txt*. Abans, però, muntarem la nostra unitat de Google Drive:

```
from google.colab import drive
drive.mount('/content/drive')
```

D'aquesta manera, en */content/drive/MyDrive* tendrem accés a tot el contingut de la nostra unitat de Google Drive.

I ara carregaré el fitxer *persones.txt* que he copiat en el directori *dades* de la meua unitat de Google Drive:

```
personesRDD = sc.textFile('/content/drive/MyDrive/dades/persones.txt')
personesRDD.collect()
```

Vegem el resultat:

```
✓ [1] from pyspark.context import SparkContext
18 s sc = SparkContext("local", "Spark en Google Colab")
```

```
✓ [2] sc
0 s
```

**SparkContext**[Spark UI](#)

Version

v3.5.5

Master

local

AppName

Spark en Google Colab

```
✓ [3] dies = ("dilluns", "dimarts", "dimecres", "dijous", "divendres", "dissabte", "diumenge")
1 s diesRDD = sc.parallelize(dies)
diesRDD.collect()
```



```
['dilluns',
 'dimarts',
 'dimecres',
 'dijous',
 'divendres',
 'dissabte',
 'diumenge']
```

```
✓ [6] from google.colab import drive
19 s drive.mount('/content/drive')
```



Mounted at /content/drive

```
✓ [7] personesRDD = sc.textFile('/content/drive/MyDrive/dades/persones.txt')
2 s personesRDD.collect()
```



```
['1, Joan, 50', '2, Aina, 30', '3, Pep, 40']
```

Imatge: Vista del quadern Colab

**ALERTA**

Teniu aquest quadern Colab accessible a https://colab.research.google.com/drive/1a4_UFoulGBR9ycQ-bNj5L2jNqh51buNX