



Departamento de Ingeniería Informática

Tecnología de datos masivos

Informe final

ALUMNA:
FRANCESCA TUNINETTI

Máster en Ciberseguridad e Inteligencia de Datos
2024/2025

Indice

Tecnologías de Datos Masivos	3
Spark	4
Creación del Spark Máster y Spark Slave	5
Clúster Spark.....	7
Configuración común para Spark Máster y Worker	8
Configuración específica del Máster	13
Configuración específica del worker	18
Test del clúster	18
Kafka y Spark Structured Streaming	22
Componentes principales de Kafka.....	22
Instalación y configuración de Kafka	23
Test de Kafka con Spark Structured Streaming	24
Redis y Cassandra	26
Instalación y configuración de Redis	27
Test de la integración entre REDIS, Kafka y Spark	28
Instalación y configuración de Cassandra	29
Test de la integración entre Cassandra, Kafka y Spark	31
MongoDB.....	32
Instalación y configuración de Mongo	33
Test de la integración entre MongoDB, Kafka y Spark.....	34
Conclusiones:	34

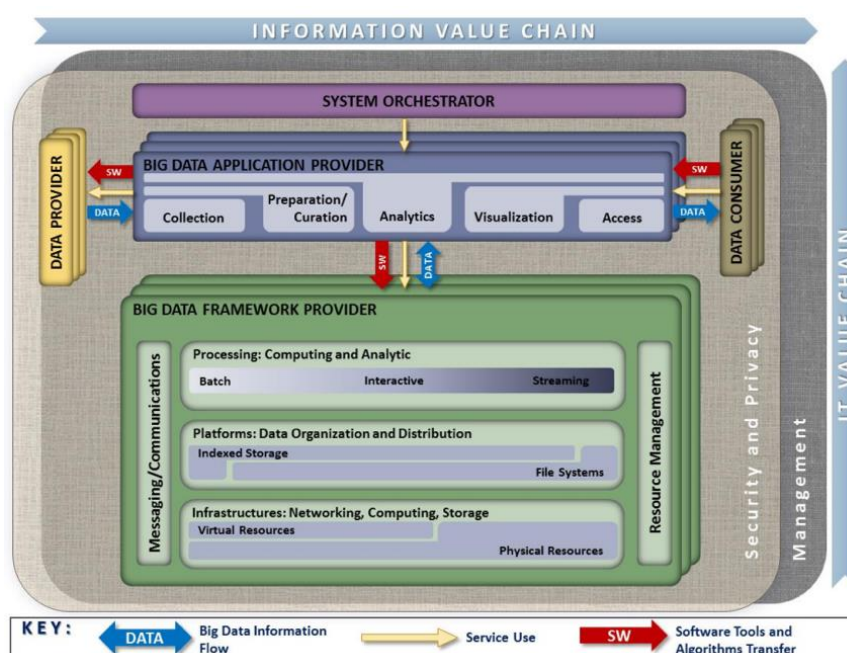
Tecnologías de Datos Masivos

Las TDM nacen para responder a la necesidad de trabajar con cantidades de datos muy grandes y sobre todo datos no estructurados y difícilmente manejables con paradigmas basados sobre arquitecturas y bases de datos tradicionales.

Con estas nuevas tecnologías ha habido un cambio en la escalabilidad de la arquitectura necesaria para gestionar eficientemente los conjuntos de datos. Por esta razón ha surgido la necesidad de crear nuevos modelos de referencia como la **NBDRA** (Arquitectura de Referencia para Big Data), desarrollado por el NIST (Instituto Nacional de Estándares y Tecnología) para proporcionar una estructura estandarizada y modular que facilite la comprensión, diseño e implementación de soluciones de Big Data.

NBDRA define los principios fundamentales y los componentes esenciales para trabajar con Big Data, independientemente de la industria o el contexto tecnológico, promoviendo la interoperabilidad, la escalabilidad y la adaptabilidad.

Estas componentes se pueden resumir de esta manera:



Donde se puede claramente distinguir entre:

- *Data provider*: el origen de los datos, que pueden ser estructurados (como bases de datos relacionales), semi-estructurados (como JSON o XML) o no estructurados (imágenes, videos, texto libre).
- *Data consumer*: sistemas (o usuarios) que consuman los datos procesados para tomar decisiones o generar valor.
- *System Orchestrator*: se encarga de coordinar las interacciones entre los distintos componentes de la arquitectura.
- *Big data Application provider*: utiliza los datos para generar valor sacando información útil para los consumidores .
- *Big data Framework provider*: gestiona a nivel técnico los datos para que el Application provider pueda funcionar.

En las siguientes secciones veremos algunas tecnologías que encajan en este marco, detallando el trabajo hecho en clase para sus instalación y configuración.

Spark

Apache Spark es un framework opensource para el procesamiento de datos distribuidos y es muy utilizado en el ámbito de Big Data ya que permite procesar datos con una escalabilidad horizontal casi sin algún limite, gracias a los múltiples nodos de un clúster que se pueden crear.

En el paradigma NBDRA, se clasifica como parte del *Big Data Framework* de momento que su principal objetivo es proporcionar una plataforma técnica para procesar y gestionar grandes volúmenes de datos distribuidos, actuando como la infraestructura subyacente que permite ejecutar cálculos y transformaciones en datos masivos.

Nace en el lenguaje Scala, un lenguaje de programación que se ejecuta en la Java Virtual Machine (JVM), pero suporta nativamente también:

- Java
- Python
- R

Para entender los principios básicos de esta herramienta muy potente, hemos realizado algunas prácticas en las que hemos configurado un pequeño clúster y ejecutado funciones de MapReduce, integrando también otras tecnologías para datos masivos como bases de datos NoSQL.

En los siguientes capítulos se describirá el trabajo hecho, detallando cada parte para una correcta comprensión de las operaciones.

Creación del Spark Máster y Spark Worker

El primer step ha sido levantar las máquinas necesarias para nuestro clúster de tamaño reducido: compuesto por un máster y un Worker. La diferencia entre estas dos es muy sencilla, el máster es el cerebro de todas las operaciones efectuadas con Spark, la maquina Worker es la que pone a disposición sus recursos y ejecuta la computación.

Las dos se han creado en el entorno del IaaS de la ULL, asequible desde el enlace <http://iaas.ull.es>.

Este entorno es un claro ejemplo de Infrastructure as a Service, que permite administrar recursos físicos como servidores y redes para la creación de máquinas virtuales según la necesidad, a través de una interfaz centralizada.

Hay algunas reglas de configuración en las máquinas creadas por alumnos, como por ejemplo el nombre asignado. Cada una tiene que ser nombrada con el acrónimo de la asignatura de referencia, en este caso TDM, seguidas por la label “SPARK_MASTER” por el máster y “SPARK_WORKER” por el Worker y por

último el nombre de usuario univoco por cada alumno, como se puede observar desde la captura de pantalla que sigue:

Crear máquina virtual

1 Configuración básica

2 Redes

3 Almacenamiento

4 Revisar

Nombre: TDM-SPARK_MASTER-alm123456789

Descripción: Spark Master (driver)

Clúster: Cluster-Rojo (KVMRojo)

Fuente de aprovisionamiento: Plantilla

Plantilla: ubuntu-2204 (base version)

Sistema operativo: Ubuntu Trusty Tahr LTS+

Memoria (MiB): 8192

Total de CPUs virtuales: 8

Optimizado para: Servidor

☐ Iniciar máquina virtual en la creación

☐ Habilitar Cloud-init/Sysprep

[Opciones avanzadas de la topología de la CPU](#)

Siguiente Cancelar

Desde la imagen son también visibles las configuraciones hardware y software aplicadas:

- OS: Ubuntu 2204 base version
- Memoria RAM: 8192 MB
- CPUs Virtuales: 8

No hizo falta cambiar otras configuraciones en la parte de redes y almacenamiento de momento que las máquinas que se crean ya vienen con un adaptador de red y un almacenamiento de 25GB.

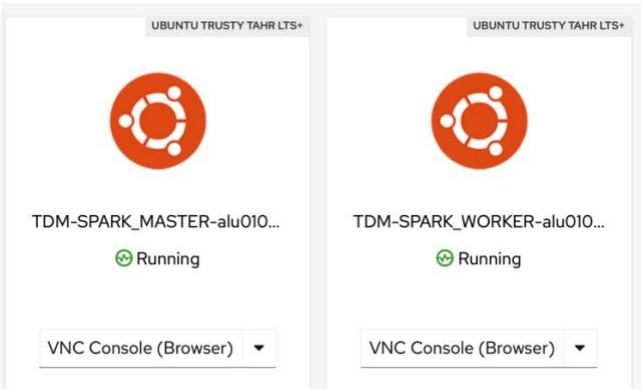
Los IP públicos asignados son los siguientes:

- Spark Máster: 10.6.129.87
- Spark Slave: 10.6.130.200

Como se puede notar, las máquinas no residen en la misma subred sino una en la 10.6.129.87/24 y la otra en la 10.6.130.200/24. Sin embargo, esto no es un

problema porque las conectaremos configurando el clúster de Spark que permitirá que las dos puedan comunicarse sin algún problema.

Esta es una demostración evidente de como la computación distribuida se realiza, conectando diferentes hardware colocados en diferentes lugares permitiéndoles de compartir recursos que se utilizan para procesar grandes cantidades de datos.

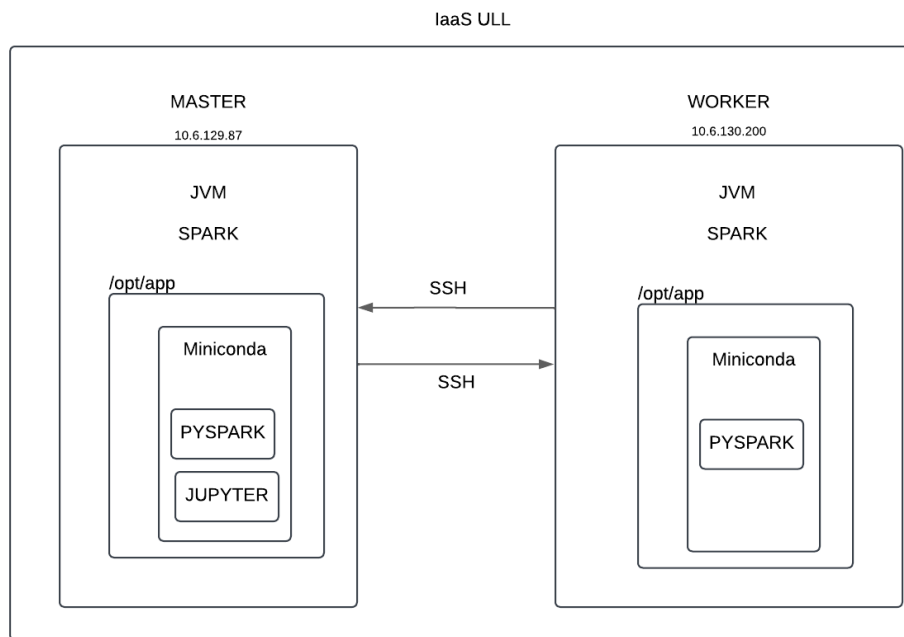


Recursos hardware de las máquinas creadas en laaaS:

Utilization			
CPU	Memory	Networking	Disk
99.125% <small>Disponibile of 100%</small>	5.42 <small>Disponibile of 8 GiB</small>	100% <small>Disponibile of 100%</small>	4.37 <small>Unallocated of 25 GiB Provisioned</small>

Clúster Spark

Una vez creadas las VM, el objetivo ahora es crear, configurar y arrancar nuestro clúster de Spark garantizando que todo el entorno de trabajo sea preparado para computar.



Configuración común para Spark Máster y Worker

Como primera cosa, hemos arrancado las dos máquinas para podernos conectar a través de protocolo SSH en el puerto por defecto (puerto 22).

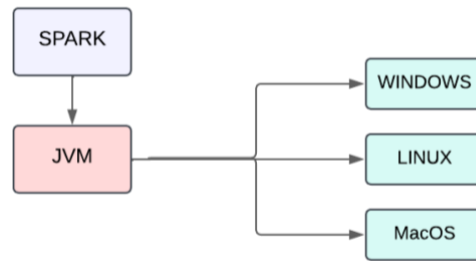
```
$ ssh usuario@10.6.129.87
```

```
$ ssh usuario@10.6.130.200
```

Una vez conectados, limpiamos, actualizamos el sistema e instalamos algunas herramientas útiles para descargar y manejar archivos.

Instalación y configuración de Java

Lo que hace que Spark sea cross-platform es la Java Virtual Machine (JVM). El compilador de Scala convierte el programa en bytecode, que viene ejecutado por la JVM que lo puede ejecutar en diferentes OS, como es visible en el siguiente esquema:



Por esta razón, necesitamos instalar Java (OpenJDK 11) en nuestro Máster y configurar variables de entorno que nos permitan ejecutar los comandos en el terminal sin especificar cada vez la ruta del ejecutable.

Para crearlas utilizamos el editor nano, y creamos:

- *JAVA_HOME*: define la ruta donde está instalado Java que Spark utilizará para saber dónde encontrar el entorno de Java.
- *PATH*: es una variable de entorno del sistema operativo que contiene una lista de directorios donde el sistema busca programas ejecutables.

```
GNU nano 6.2 /etc/profile.d/java.sh
export JAVA_HOME=/usr/lib/jvm/default-java
export PATH=$JAVA_HOME/bin:$PATH
```

Instalación y configuración de Scala

Siendo Scala el lenguaje con el que Apache Spark, claramente lo vamos a necesitar para nuestro entorno.

Descargamos la versión 2 de Scala y lo instalamos.

Instalación y configuración de Miniconda

A este punto hemos instalado Java, Scala y configurado las variables de entorno de Java. Para poder utilizar la interfaz de Spark para Python (PySpark), que es una de las más utilizadas en el ámbito de machine learning y big data, necesitamos un entorno que gestione Python de manera sencilla y eficiente.

Por esta razón, instalaremos Miniconda: una distribución ligera de Anaconda, que permite manejar entornos virtuales y librerías de Python, como la de PySpark.

Durante esta práctica y las siguientes, manejaremos diferentes tecnologías. Para una mejor organización, hemos decidido crear un directorio común en el que instalaremos todos los programas utilizados para el laboratorio en la ruta `/opt/app`.

El primer programa instalado en su interior ha sido Miniconda: durante la ejecución del script, nos pidió especificar el directorio de instalación y como mencionado, ponemos `/opt/app/miniconda3`.

Llegados a este punto, hemos terminado la instalación y nos falta actualizar la configuración del shell para que incluya los comandos de Miniconda y nos permita utilizarlo enseguida. Para hacerlo, ejecutamos:

```
$ /opt/app/miniconda3/bin/conda init
```

Reiniciamos el shell para poder cargar la nueva configuración y podemos notar como, justo al lado del inicio de la línea del terminal aparece un nuevo prompt con el prefijo (base). Esto significa que Miniconda se ha activado correctamente en cuanto “base” es el entorno virtual predeterminado creado durante su instalación y contiene:

- El lenguaje Python en su versión por defecto
- Herramientas esenciales para el correcto funcionamiento del entorno virtual como conda (el gestor de paquetes y entornos) y otras librerías básicas.

Sin especificar nada, este entorno se activaría de forma automática cada vez que el usuario arranca el sistema. Para dejar libertad en el entorno a usar, desactivamos esta opción.

Ahora para entrar en el entorno base de Miniconda, habrá que ejecutarse el comando manualmente:

```
$ conda activate
```

Y para salir:

```
$ conda deactivate
```

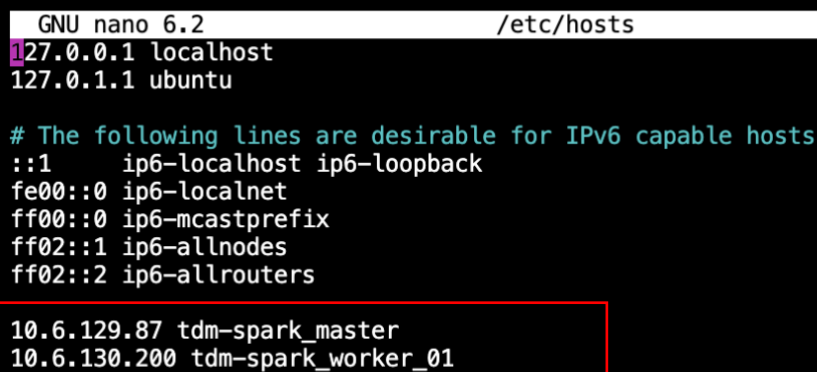
Repetimos los mismos steps para cargar los comandos de Miniconda también en root. Esta operación nos asegura que en caso de operaciones que requieran privilegios de root, no haya problema de compatibilidad.

Una vez terminado, limpiamos el directorio de los archivos de instalación y seguimos actualizando conda, el gestor de paquetes contenido en Miniconda.

Instalación y configuración de Spark

Para añadir la configuración del clúster que queremos crear, editamos el fichero `/etc/hosts` para añadir todas las direcciones IP.

Este fichero permite asociar los nombres de los hosts a sus IP, de modo de resolverlos sin necesidad de una query DNS:



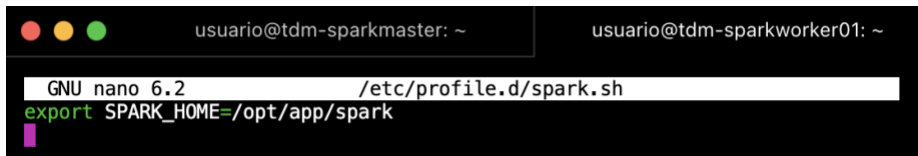
```
GNU nano 6.2 /etc/hosts
127.0.0.1 localhost
127.0.1.1 ubuntu

# The following lines are desirable for IPv6 capable hosts
::1 ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters

10.6.129.87 tdm-spark_master
10.6.130.200 tdm-spark_worker_01
```

Para que Spark funcione correctamente es importante crear los directorios locales necesarios y configurar los siguientes ficheros:

- crear una variable de entorno SPARK_HOME para Spark en el fichero `/etc/profile.d/spark.sh` y cargarla.

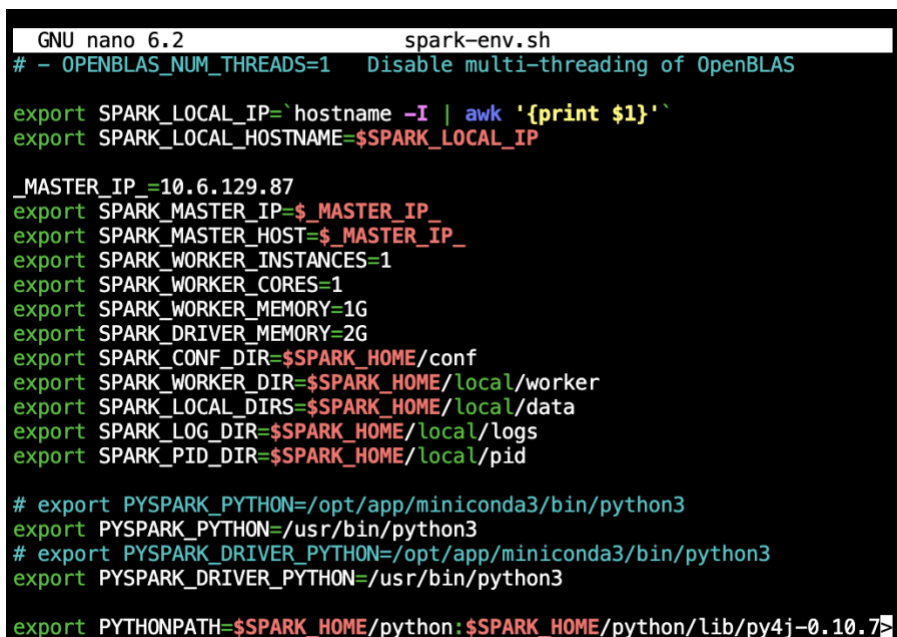


```

GNU nano 6.2 /etc/profile.d/spark.sh
export SPARK_HOME=/opt/app/spark

```

- copiamos la plantilla de configuración `spark-env.sh` para personalizarla según nuestras necesidades, en el directorio `/opt/app/spark`.



```

GNU nano 6.2 spark-env.sh
# - OPENBLAS_NUM_THREADS=1 Disable multi-threading of OpenBLAS

export SPARK_LOCAL_IP=`hostname -I | awk '{print $1}'`
export SPARK_LOCAL_HOSTNAME=$SPARK_LOCAL_IP

_MASTER_IP=10.6.129.87
export SPARK_MASTER_IP=$_MASTER_IP
export SPARK_MASTER_HOST=$_MASTER_IP
export SPARK_WORKER_INSTANCES=1
export SPARK_WORKER_CORES=1
export SPARK_WORKER_MEMORY=1G
export SPARK_DRIVER_MEMORY=2G
export SPARK_CONF_DIR=$SPARK_HOME/conf
export SPARK_WORKER_DIR=$SPARK_HOME/local/worker
export SPARK_LOCAL_DIRS=$SPARK_HOME/local/data
export SPARK_LOG_DIR=$SPARK_HOME/local/logs
export SPARK_PID_DIR=$SPARK_HOME/local/pid

# export PYSPARK_PYTHON=/opt/app/miniconda3/bin/python3
export PYSPARK_PYTHON=/usr/bin/python3
# export PYSPARK_DRIVER_PYTHON=/opt/app/miniconda3/bin/python3
export PYSPARK_DRIVER_PYTHON=/usr/bin/python3

export PYTHONPATH=$SPARK_HOME/python:$SPARK_HOME/python/lib/py4j-0.10.7

```

Entre las variables configuradas hay:

- La dirección IP del Máster para que el worker pueda conectarse.
- Parámetros relacionados a la asignación de recursos y gestión de procesos para la computación en el worker.
- El intérprete Python para PySpark. Se podría elegir entre lo de Miniconda y un intérprete de sistema, lo que elegimos en nuestra configuración.

Configuración específica del Máster

Instalación y configuración de Jupyter

Para la ejecución de código de manera interactiva, instalamos en el Spark Máster Jupyter lab a través de Conda.

Gracias a esta aplicación, será posible escribir código en una interfaz gráfica web y ver los resultados de la computación directamente en el notebook.

Si pensamos en Jupyter como una aplicación diseñada para usuarios, podemos considerarla como parte del *Big Data Application Provider* de momento que no realiza el procesamiento de datos directamente sino se apoya en frameworks como Spark para llevar a cabo tareas computacionales intensivas con el objetivo de producir información de valor.

Después de crear un directorio dedicado para los notebooks, generamos y modificamos el fichero de configuración (desde la cuenta usuario) modificando los siguientes parámetros:

- *ip*: aquí hemos puesto la dirección IP del Spark Máster
- *notebook_dir*: la dirección del directorio en que pondremos los notebooks
- *port*: cambia el puerto por defecto a 8090.
- *open_browser*: evita que Jupyter abra el navegador automáticamente.

Con el entorno base activado, arrancamos Jupyter lab con el comando

```
(base) $ jupyter lab
```

```
(base) usuario@tdm-sparkmaster:~$ jupyter lab
[I 2025-01-08 00:31:33.324 ServerApp] jupyter_server_fileid | extension was successfully linked.
[I 2025-01-08 00:31:33.330 ServerApp] jupyter_server_ydoc | extension was successfully linked.
[W 2025-01-08 00:31:33.334 LabApp] 'ip' has moved from NotebookApp to ServerApp. This config will be passed to ServerApp. Be sure to update your config before our
xt release.
[W 2025-01-08 00:31:33.334 LabApp] 'notebook_dir' has moved from NotebookApp to ServerApp. This config will be passed to ServerApp. Be sure to update your config b
ore our next release.
[W 2025-01-08 00:31:33.334 LabApp] 'port' has moved from NotebookApp to ServerApp. This config will be passed to ServerApp. Be sure to update your config before ou
next release.
[W 2025-01-08 00:31:33.334 LabApp] 'port' has moved from NotebookApp to ServerApp. This config will be passed to ServerApp. Be sure to update your config before ou
next release.
[W 2025-01-08 00:31:33.341 ServerApp] notebook_dir is deprecated, use root_dir
[I 2025-01-08 00:31:33.341 ServerApp] jupyterlab | extension was successfully linked.
[I 2025-01-08 00:31:33.347 ServerApp] nbclassic | extension was successfully linked.
[I 2025-01-08 00:31:33.374 ServerApp] notebook_shim | extension was successfully linked.
[I 2025-01-08 00:31:33.384 ServerApp] notebook_shim | extension was successfully loaded.
[I 2025-01-08 00:31:33.385 ServerApp] Configured File ID manager: ArbitraryFileIDManager
[I 2025-01-08 00:31:33.385 FileIDExtension] ArbitraryFileIDManager: Configured root dir: /home/usuario/notebooks
[I 2025-01-08 00:31:33.385 FileIDExtension] ArbitraryFileIDManager: Configured database path: /home/usuario/.local/share/jupyter/file_id_manager.db
[I 2025-01-08 00:31:33.385 FileIDExtension] ArbitraryFileIDManager: Successfully connected to database file.
[I 2025-01-08 00:31:33.385 FileIDExtension] ArbitraryFileIDManager: Creating File ID tables and indices with journal_mode = DELETE
[I 2025-01-08 00:31:33.385 ServerApp] jupyter_server_fileid | extension was successfully loaded.
[I 2025-01-08 00:31:33.385 ServerApp] jupyter_server_ydoc | extension was successfully loaded.
[I 2025-01-08 00:31:33.385 LabApp] JupyterLab extension loaded from /opt/app/miniconda3/lib/python3.8/site-packages/jupyterlab
[I 2025-01-08 00:31:33.385 LabApp] JupyterLab application directory is /opt/app/miniconda3/share/jupyter/lab
[I 2025-01-08 00:31:33.386 ServerApp] jupyterlab | extension was successfully loaded.

JupyterLab

Read the migration plan to Notebook 7 to learn about the new features and the actions to take if you are using extensions.
https://jupyter-notebook.readthedocs.io/en/latest/migrate_to_notebook7.html

Please note that updating to Notebook 7 might break some of your extensions.

[I 2025-01-08 00:31:33.387 ServerApp] nbclassic | extension was successfully loaded.
[I 2025-01-08 00:31:33.387 ServerApp] Serving notebooks from local directory: /home/usuario/notebooks
[I 2025-01-08 00:31:33.388 ServerApp] Jupyter Server 1.24.0 is running at:
[I 2025-01-08 00:31:33.388 ServerApp] http://10.6.129.87:8090/lab?token=e8e0104699ff0a0c1b4aa6beef718d0839a3d22676ea1dc6
[I 2025-01-08 00:31:33.388 ServerApp] or http://127.0.0.1:8090/lab?token=e8e0104699ff0a0c1b4aa6beef718d0839a3d22676ea1dc6
[I 2025-01-08 00:31:33.388 ServerApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[W 2025-01-08 00:31:33.387 ServerApp] No web browser found: could not locate runnable browser.

To access the server, open this file in a browser:
file:///home/usuario/.local/share/jupyter/runtime/jpserver-234652-open.html
Or copy and paste one of these URLs:
http://10.6.129.87:8090/lab?token=e8e0104699ff0a0c1b4aa6beef718d0839a3d22676ea1dc6
or http://127.0.0.1:8090/lab?token=e8e0104699ff0a0c1b4aa6beef718d0839a3d22676ea1dc6
```

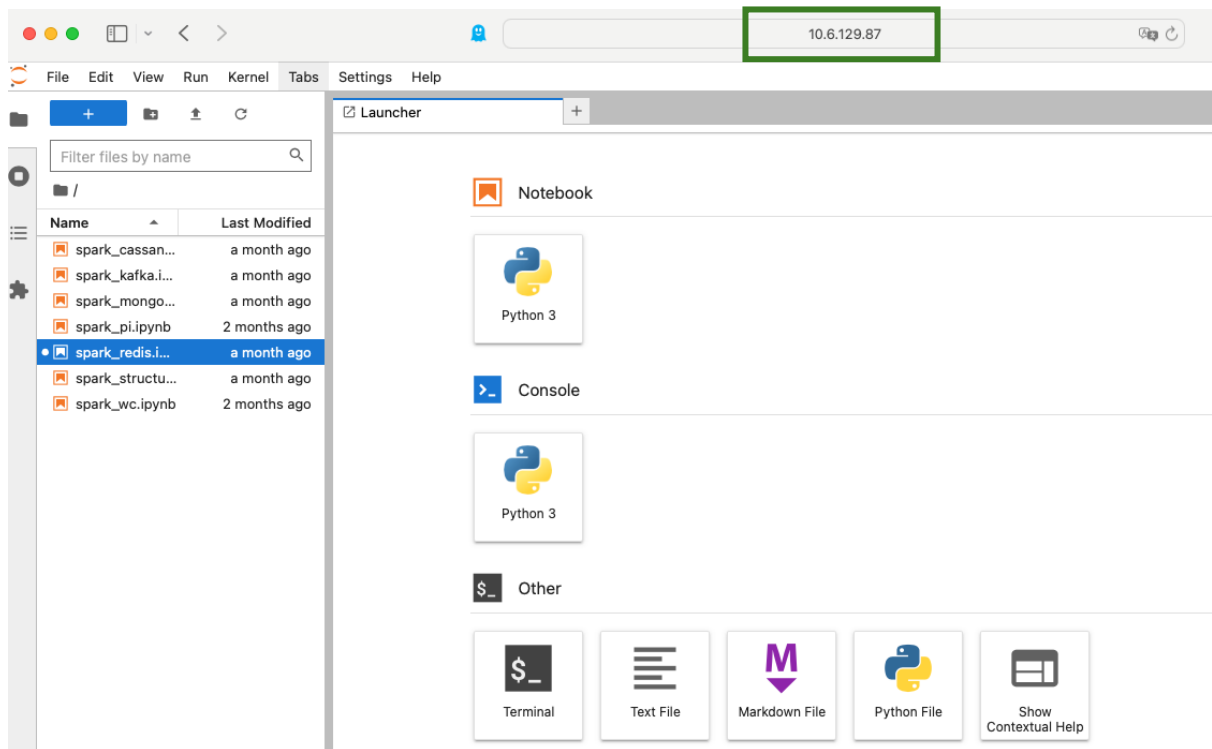
Como es visible en la captura, Jupyter nos devuelve un enlace desde el que es posible abrir la interfaz gráfica en el navegador.

En este caso el enlace es:

<http://10.6.129.87:8090/lab?token=e8e0104699ff0a0c1b4aa6beef718d0839a3d22676ea1dc6>

Observándolo, podemos notar que es compuesto por:

- la dirección IP del nodo donde se está ejecutando Jupyter Lab, que coincide con el nodo Spark Master en este caso
- el puerto que le hemos configurado (8090)
- un token generado por la aplicación para garantizar la seguridad. Permite acceder a la interfaz gráfica sin necesidad de autenticarte y además permite compartir el acceso a los notebooks a través de este enlace.



Configuración de Spark específica del Máster

Creamos una copia de la plantilla de configuración `spark-defaults.conf` para personalizar las siguientes cosas:

- Cambiar el puerto de la interfaz web de Spark de 4040 (valor predeterminado) a 8082.
- Indicar la IP del nodo Máster
- Configurar los recursos del Máster y del Worker
- Asignar puertos específicos para diferentes servicios internos de Spark para evitar problemas con el firewall.

Una vez terminado, hemos podido finalmente arrancar el Spark Máster con el comando:

```
$ sbin/start-master.sh
```

Para acceder a la interfaz web del Máster, hemos introducido la URL compuesta por la IP del máster y el puerto elegido (8080):

Spark Master at spark://10.6.129.87:7077

URL: spark://10.6.129.87:7077
 Alive Workers: 1
 Cores in use: 1 Total, 0 Used
 Memory in use: 1024.0 MiB Total, 0.0 B Used
 Resources in use:
 Applications: 0 Running, 9 Completed
 Drivers: 0 Running, 0 Completed
 Status: ALIVE

~ Workers (1)

Worker Id	Address	State	Cores	Memory	Resources
worker-20241205180918-10.6.130.200-40653	10.6.130.200:40653	ALIVE	1 (0 Used)	1024.0 MiB (0.0 B Used)	

~ Running Applications (0)

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

~ Completed Applications (9)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20241205191602-0008	SparkMongoDB	1	1024.0 MiB		2024/12/05 19:16:02	usuario	FINISHED	5,8 min
app-20241205190921-0007	SparkCassandra	1	1024.0 MiB		2024/12/05 19:09:21	usuario	FINISHED	2,2 min
app-20241205185723-0006	SparkCassandra	1	1024.0 MiB		2024/12/05 18:57:23	usuario	FINISHED	12 min
app-20241205184703-0005	SparkCassandra	1	1024.0 MiB		2024/12/05 18:47:03	usuario	FINISHED	8,8 min
app-20241205181921-0004	SparkRedis	1	1024.0 MiB		2024/12/05 18:19:21	usuario	FINISHED	14 min
app-20241128185708-0003	StructuredNetworkWordCount	1	1024.0 MiB		2024/11/28 18:57:08	usuario	FINISHED	7,8 min
app-20241128184709-0002	KafkaWordCount	1	1024.0 MiB		2024/11/28 18:47:09	usuario	FINISHED	9,6 min

Esta interfaz está diseñada para supervisar el estado y las aplicaciones del clúster Spark. Está dividida en varias secciones que proporcionan información sobre los nodos del clúster, las aplicaciones en ejecución y las completadas.

1. En la parte superior se encuentran:
 - a. los detalles del nodo Máster: su IP y el puerto donde está escuchando (7077).
 - b. Alive Workers: indica cuántos nodos Worker están conectados al clúster. En este caso debería ser 0 en cuanto todavía no hemos establecido la conexión, pero esta captura ha sido guardada una vez terminado todo el laboratorio.
 - c. Cores y memory en uso: se refiere a la memoria total disponible en los Workers del clúster y cuánta se está utilizando.
 - d. Número de aplicaciones actualmente en ejecución y las completadas.

2. En la sección *Workers* se puede observar el listado de los nodos Worker conectados al clúster con la información sobre el estado relacionada.
3. Sección *Running Applications* lista las aplicaciones que se están ejecutando actualmente en el clúster (en este caso 0).
4. Sección *Completed Applications*: contiene todas las aplicaciones completadas por el clúster y la información relacionada como ID, nombre, recursos utilizados y más.

Configuración de la comunicación Máster – Workers

Después de parar el proceso de Spark en el Máster, empezamos el proceso de generación y distribución de las claves públicas/privadas para que las VM puedan comunicar con protocolo SSH entre ellas sin necesidad de contraseña.

1. Generamos un par de claves SSH en el nodo Master con el comando *ssh-keygen*, que crea una clave privada que permanecerá en el Master y una clave pública que se compartirá con los nodos Worker y que se agrega en su archivo *authorized_keys*, configurando también los permisos para garantizar su seguridad.
2. Compartimos la clave pública generada con el Worker, distribuyéndola a su archivo *authorized_keys*. Esta operación se realiza con el comando *ssh-copy-id*, que automatiza la transferencia y configuración de las claves en caso tengamos múltiples nodos Worker. A partir de este momento, el Worker reconocerá el Máster como una entidad confiable.
3. Probamos la conexión establecida a través del protocolo SSH, comprobando que no nos pida ninguna contraseña.

4. Generamos una copia de la plantilla slaves (`$SPARK_HOME/conf/slaves`) y la editamos para incluir el host name del worker (`tdm-spark_worker_01`) comentando el localhost que aparece al principio.

Este fichero sirve para que el Máster sepa cuales nodos forman parte de su clúster.

Este step concluye la creación y configuración del clúster, arrancados de nuevo el proceso Spark en el máster, es posible ver desde la interfaz web que aparece el worker en la sección relacionada.

Configuración específica del worker

En la práctica desarrollada en clase, hemos creado solo un worker, pero como ya mencionado, esta estructura puede escalarse horizontalmente según necesidad sin algún problema.

Editamos también aquí el fichero `/etc/hosts` para añadir la configuración del clúster y en el directorio `/opt/app` creado, descargamos, descomprimos e instalamos Spark.

Para evitar problemas de permisos, se ha cambiado el owner del directorio a root y se ha creado un enlace simbólico para simplificar la ruta de acceso a Spark sin especificar el nombre del fichero completo.

Test del clúster

Test en consola

Como primer test del clúster, hemos intentado generar el número π con el método MonteCarlo usando Spark desde consola, con el comando:

```
$ ./bin/run-example SparkPi 10000
```

```

25/01/08 11:53:23 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have all completed, from pool
25/01/08 11:53:23 INFO DAGScheduler: ResultStage 0 (reduce at SparkPi.scala:38) finished in 91,463 s
25/01/08 11:53:23 INFO DAGScheduler: Job 0 is finished. Cancelling potential speculative or zombie tasks for this job
25/01/08 11:53:23 INFO TaskSchedulerImpl: Killing all running tasks in stage 0: Stage finished
25/01/08 11:53:23 INFO DAGScheduler: Job 0 finished: reduce at SparkPi.scala:38, took 91,678014 s
Pi is roughly 3.1416396991416398
25/01/08 11:53:23 INFO SparkUI: Stopped Spark web UI at http://10.6.129.87:8082
25/01/08 11:53:23 INFO StandaloneSchedulerBackend: Shutting down all executors
25/01/08 11:53:23 INFO CoarseGrainedSchedulerBackend$DriverEndpoint: Asking each executor to shut down
25/01/08 11:53:23 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
25/01/08 11:53:23 INFO MemoryStore: MemoryStore cleared
25/01/08 11:53:23 INFO BlockManager: BlockManager stopped
25/01/08 11:53:23 INFO BlockManagerMaster: BlockManagerMaster stopped
25/01/08 11:53:23 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator stopped!
25/01/08 11:53:23 INFO SparkContext: Successfully stopped SparkContext
25/01/08 11:53:23 INFO ShutdownHookManager: Shutdown hook called
25/01/08 11:53:23 INFO ShutdownHookManager: Deleting directory /opt/app/spark-3.0.3-bin-hadoop3.2/local/data/spark-4a2652d8-f63d-4918-bca3-ee8e7969da3e
25/01/08 11:53:23 INFO ShutdownHookManager: Deleting directory /tmp/spark-45e20c37-445e-4520-907c-0465e71966d9

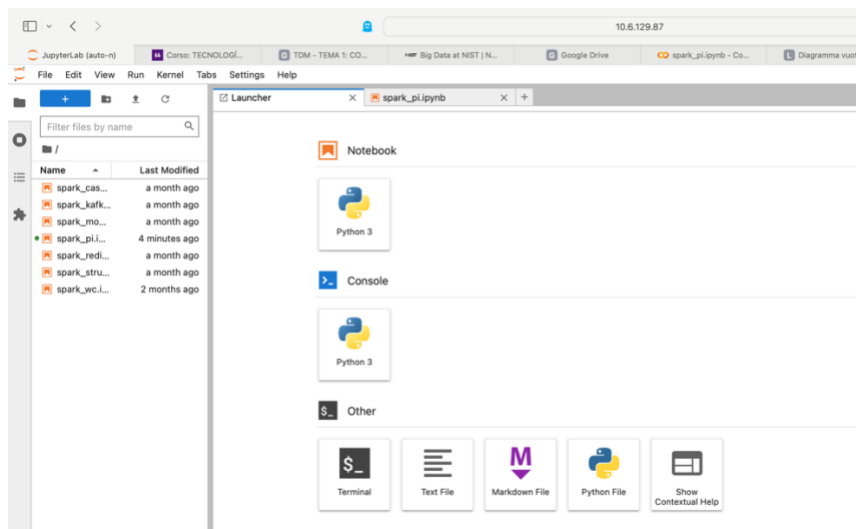
```

Test en Jupyter lab

Para probar los notebooks en Jupyter Lab, activamos el entorno virtual de Miniconda y lo arrancamos con el comando:

```
(base) $ jupyter lab
```

Para después ir a la interfaz web disponible al enlace proporcionado por Jupyter.



A través de esta página es posible cargar los notebooks directamente, de manera sencilla a través de la user interface.

Ejecución del notebook spark_py

Este notebook ejecuta la misma función para estimar el valor de π

- Comprobamos que la maquina sea la correcta

Hostname: tdm-sparkmaster
IP Address: 10.6.129.87

- Las rutas de los ficheros necesarios para el correcto funcionamiento de Spark:

```
/usr/lib/jvm/default-java  
/opt/app/spark  
/opt/app/miniconda3/bin/python  
/opt/app/spark/python/lib  
/opt/app/miniconda3/bin:/opt/app/miniconda3/condabin:/usr/lib/jvm  
snap/bin:/opt/app/cassandra/bin:/opt/app/mongodb/bin
```

- El contexto de Spark se ha encontrado:

SparkContext

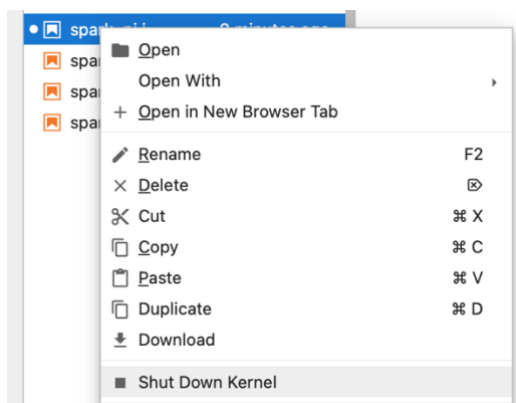
Spark UI

Version	v3.0.3
Master	spark://10.6.129.87:7077
AppName	Calculate Pi

- Se ha ejecutado la función para el cálculo del π

```
# calculate the value of Pi i.e. 3.14...  
def inside(p):  
    x, y = random.random(), random.random()  
    return x*x + y*y < 1  
  
num_samples = 100000000  
count = sc.parallelize(range(0, num_samples)).filter(inside).count()  
pi = 4 * count / num_samples  
  
# print the value of Pi  
print(pi)  
  
3.14153148
```

- Shutdown del kernel para librar los recursos



Ejecución del notebook spark_wc (Word counter)

Este notebook carga un fichero de text, en este caso *bootstrap.log*

```
-rw-r--r-- 1 root root 64549 feb 17 2023 /var/log/bootstrap.log
```

Y cuenta las palabras del fichero cargado, con una función basada en MapReduce:

```
[6]: # read text file
#f = sc.textFile("/var/log/dnf.log")
#f = sc.textFile("/var/log/boot.log")
f = sc.textFile("/var/log/bootstrap.log")

[7]: # define map and reduce transformations
wc = f.flatMap(lambda line: line.split(' ')) \
      .map(lambda word: (word, 1)) \
      .reduceByKey(lambda a,b: a+b)

[8]: # trigger action to execute transformations and collect results
wc.collect()

[9]: [{"build/chroot/var/lib/apt/lists/partial/ftpmaster.internal_ubuntu_dists_jammy_InRelease",
      1},
      ('gpgv:', 6),
      ('21', 2),
      ('', 360),
      ('using', 7),
      ('Good', 2),
      ('Automatic', 2),
      ('Signing', 2),
      ('(2018)', 2),
      ('URL:http://ftpmaster.internal/ubuntu/dists/jammy/main/binary-amd64/by-hash/SHA256/37cb57f1554cbfa71c5a29ee9ffee18a9a8c1782t
e0874b7ff4ce8f9c11',
      1),
      ('build/chroot/var/lib/apt/lists/partial/ftpmaster.internal_ubuntu_dists_jammy_restricted_binary-amd64_Packages.xz"',
      1),
      ('[155528/155528]', 1),
      ('URL:http://ftpmaster.internal/ubuntu/pool/main/b/base-files/base-files_12ubuntu4_amd64.deb',
      1),
      ('[62940/62940]', 1),
      ('build/chroot/var/cache/apt/archives/partial/base-files_12ubuntu4_amd64.deb"',
      1),
      ('build/chroot/var/cache/apt/archives/partial/base-passwd_3.5.52build1_amd64.deb"',
      1),
      ('[768660/768660]', 1),
      ('build/chroot/var/cache/apt/archives/partial/bash_5.1-6ubuntu1_amd64.deb"',
      1),
      ('[17:19:34', 37),
      ('[92010/92010]', 1),
      ('URL:http://ftpmaster.internal/ubuntu/pool/main/d/debconf/debconf_1.5.79ubuntu1_all.deb',
      1),
      1},
```

Esta función divide todas las líneas en un array de palabras, cada una asociada con un número 1 en el conteo inicial.

Una vez terminado, se pasa a la fase de reducción, donde las palabras idénticas son agrupadas y sus valores 1 son sumados para calcular el conteo total de cada palabra en el texto.

El modelo de programación MapReduce es el más adecuado para analítica de agregación en cuanto a procesamiento distribuido y escalabilidad, ya que permite dividir los datos en múltiples fragmentos que se procesan en paralelo a través de una fase de mapeo (Map) y luego se combinan en una fase de reducción (Reduce), logrando así una alta eficiencia en el manejo de grandes volúmenes de datos.

Kafka y Spark Structured Streaming

Hemos visto cómo Spark y Jupyter encajan en el marco NBDRA. En esta sección analizaremos Kafka y Spark Structured Streaming para entender ejemplos de cómo se gestionan los flujos de datos en tiempo real.

Kafka es una plataforma distribuida dedicada al transporte de flujos de datos en tiempo real, que organiza los datos en *topics* y los transmite desde los productores (producers) hasta los consumidores (consumers) en forma de mensajes. Aunque Kafka puede manejar flujos de datos de baja latencia (en intervalos de milisegundos o segundos), su propósito principal no es procesar los datos, sino garantizar su transmisión de manera escalable.

Por el contrario, Spark Structured Streaming es una librería específica de Spark diseñada para trabajar con flujos de datos en tiempo real. En lugar de procesar bloques estáticos (como en el procesamiento por lotes), Spark Structured Streaming permite aplicar operaciones analíticas y de transformación sobre datos dinámicos de manera continua o en micro-lotes, integrándose con fuentes como Kafka para realizar el procesamiento.

Componentes principales de Kafka

- **Zookeeper:** su función principal es proporcionar servicios de coordinación y sincronización entre los diferentes brokers que forman parte del clúster. Sin este mecanismo, los brokers no podrían comunicarse entre sí de manera eficiente, lo que afectaría la escalabilidad y la tolerancia a fallos
- **Topics:** los topics son el núcleo lógico de cómo Kafka organiza y almacena los datos. Un topic es básicamente un "canal" al que los producers envían mensajes y del que los consumers los leen.
- **Broker:** el broker es el núcleo operativo de Kafka. Cada broker es una instancia de Kafka que se encarga de almacenar y gestionar los mensajes

dentro de los tópicos. Un broker maneja las particiones de los tópicos y es responsable de garantizar que los datos se almacenen y estén disponibles para los consumers. Además, Kafka utiliza un sistema de replicación, donde las particiones se copian en múltiples brokers para garantizar que no se pierdan datos en caso de fallo

- **Producers:** los producers son los responsables de generar y enviar datos hacia Kafka. Representan las aplicaciones o sistemas que producen los eventos o mensajes que se almacenarán en los tópicos.
- **Consumers:** quienes consuma los datos, pueden ser aplicaciones que los procesan o usuarios que los visualizan.
- **Subscripciones:** las subscripciones son la forma en que los consumers se conectan a los topics para leer los datos.

Instalación y configuración de Kafka

En esta sección, muchas operaciones descritas serán parecidas a las que ya vimos en el capítulo de Spark.

Descargamos la última versión de Kafka a través de wget y la instalamos en nuestro directorio dedicado a las aplicaciones del laboratorio /opt/app.

Después de cambiar los permisos de la carpeta de Kafka, configurar el enlace simbólico de manera parecida a como ya lo hicimos y crear los directorios locales necesarios, editamos los siguientes ficheros:

- `config/zookeeper.properties`: para modificar el directorio por defecto donde el zookeeper guarda sus datos.

```
# the directory where the snapshot is stored.  
#dataDir=/tmp/zookeeper  
dataDir=/opt/app/kafka/local/zookeeper
```

- `config/server.properties`: en este fichero modificamos la configuración del servidor de Kafka definiendo la dirección IP y el puerto en los que el

broker de Kafka escuchará las conexiones entrantes, indicando que no se utilizará encriptación TLS/SSL (PLAINTEXT).

```
# The address the socket server listens on. It will get the value returned from
# java.net.InetAddress.getCanonicalHostName() if not configured.
#   FORMAT:
#   listeners = listener_name://host_name:port
#   EXAMPLE:
listeners=PLAINTEXT://10.6.129.87:9092
```

Además, especificamos el directorio donde Kafka almacenará los logs de las particiones de los tópicos y finalmente indicamos la dirección IP y el puerto del servidor de ZooKeeper al que Kafka debe conectarse (2181).

En ambos casos, el IP será lo del Spark Máster.

```
# A comma separated list of directories under which to store log files
log.dirs=/opt/app/kafka/local/logs
```

```
# Zookeeper connection string (see zookeeper docs for details).
# This is a comma separated host:port pairs, each corresponding to a zk
# server. e.g. "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002".
# You can also append an optional chroot string to the urls to specify the
# root directory for all kafka znodes.
zookeeper.connect=10.6.129.87:2181
```

Test de Kafka con Spark Structured Streaming

Test en consola

Después de arrancar los demonios de Zookeeper y Kafka, hemos podido finalmente testar la instalación creando nuestro primer tópico gracias al comando:

```
$ bin/kafka-topics.sh --create --replication-factor 1 --
partitions 1 --topic our-first-topic --bootstrap-server
10.6.129.87:9092
```

Se nota como el puerto especificado en el comando es lo mismo que hemos configurado en el fichero de configuración en el step anterior.

En la siguiente captura se puede observar el tópico creado:

```
(base) usuario@tdm-sparkmaster:/opt/app/kafka$ bin/kafka-topics.sh --describe --topic our-first-topic --bootstrap-server 10.6.129.87:9092
Topic: our-first-topic TopicId: -uEEJLoDQKuovdcMYislw PartitionCount: 1 ReplicationFactor: 1 Configs: segment.bytes=1073741824
Topic: our-first-topic Partition: 0 Leader: 0 Replicas: 0 Isr: 0
(base) usuario@tdm-sparkmaster:/opt/app/kafka$
```


Y en las siguientes es posible ver en los dos terminales del Spark Máster, un productor que ha enviado un mensaje, y el consumer que lo está visualizando. La transmisión del mensaje se hizo gracias a Kafka.

term 1 (Spark Master)

term 2 (Spark Master)

```
$ bin/kafka-console-
producer.sh --topic our-first-
topic --bootstrap-server
10.6.129.87:9092
```

```
$ bin/kafka-console-consumer.sh --
topic our-first-topic --from-
beginning --bootstrap-server
10.6.129.87:9092
```

Test en Jupyter lab

Arrancamos Jupyter desde el entorno de Miniconda y ejecutamos los notebooks:

- **spark_kafka**

El objetivo de este notebook es enseñar el MapReduce hecho sobre texto enviado en tiempo real desde el terminal abierto en Jupiter.

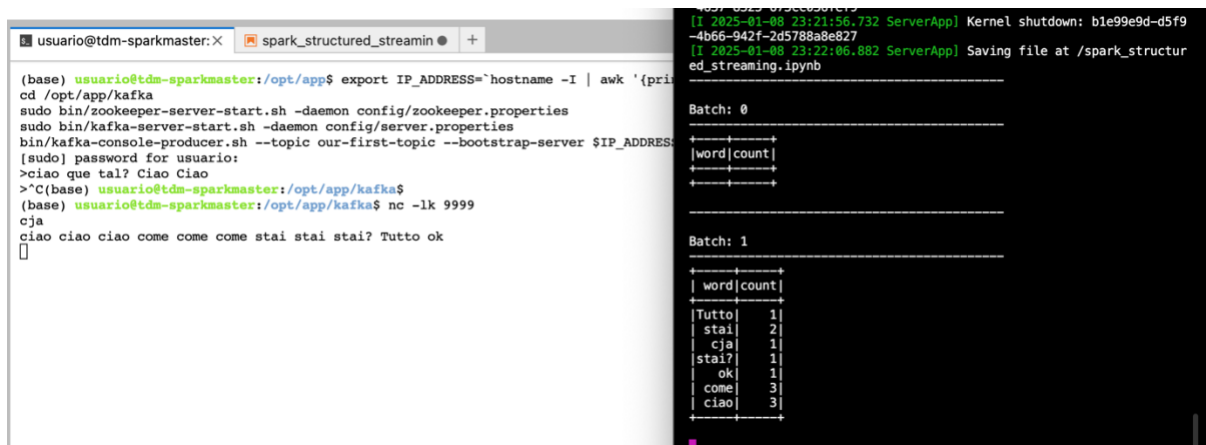
Para la transmisión del mensaje se utiliza Kafka y para el procesamiento Spark Structured Streaming:

word	count
Ciao	2
tal?	1
que	1
ciao	1

- **spark_structured_streaming**

En este notebook, el objetivo es el mismo que en el anterior, pero utilizando únicamente Spark Structured Streaming como transmisor y procesador de mensajes.

Aunque es posible hacerlo de esta manera, esta solución es mucho menos potente en comparación con la del primer notebook, ya que Kafka garantiza la persistencia de los datos en los tópicos, incluso si el consumer no estaba conectado en el momento de la transmisión. Por el contrario, con Spark Structured Streaming directamente desde una socket, los datos son volátiles y no se pueden recuperar si el consumidor no estaba activo en ese momento.



The screenshot shows a Jupyter Notebook interface with two tabs: 'usuario@tdm-sparkmaster: X' and 'spark_structured_streamin'. The left pane displays terminal commands for setting up Kafka, including starting Zookeeper and Kafka, creating a topic, and running a console producer. The right pane shows the output of a Spark Structured Streaming job, displaying two batches of word counts. Batch 0 is empty, and Batch 1 contains the following data:

word	count
Tutto	1
stai	2
cja	1
stai?	1
ok	1
come	3
ciao	3

Redis y Cassandra

Redis y Cassandra son dos database NoSQL que proporcionan soluciones a problemas diferentes en base a como han sido diseñados.

Redis es un database key-value, desarrollado en C, que guarda sus datos en la memoria RAM y por esto es muy rápido en sus operaciones. Sus aplicaciones principales se encuentran en el caching o agente de mensajes.

Por otro lado, Cassandra es un database opensource orientado a columnas, desarrollado en Java y adaptado al procesamiento de transacciones en tiempo real.

Instalación y configuración de Redis

Descargamos la última versión de Redis y después de descomprimir, lo instalamos siempre en la ruta `/opt/app`.

Cambiamos el propietario del directorio y hacemos un enlace simbólico como hemos hecho en las instalaciones anteriores y finalmente compilamos la app y ejecutamos los tests con los comandos:

```
$ cd redis; sudo make distclean; sudo make; sudo make test
```

Editamos el fichero `redis.conf` para añadir la IP del Spark Máster:

```
#bind 127.0.0.1 -:::1
bind 127.0.0.1 10.6.129.87
```

Ahora el servidor Redis está preparado para aceptar peticiones en el puerto 6379.

Lo arrancamos sin el modo protegido:

```
$ sudo src/redis-server --protected-mode no
```

y lo ponemos en background para seguir ejecutando el cliente de Redis con el comando:

```
$ src/redis-cli
```

En el terminal de la MV o de Jupyter lab, arrancamos los siguientes demonios:

- Zookeeper

```
$ sudo bin/zookeeper-server-start.sh -daemon
```

- Kafka

```
$ sudo bin/kafka-server-start.sh -daemon config/server.properties
```

- Producer de Kafka

```
$ bin/kafka-console-producer.sh --topic our-first-topic --
bootstrap-server 10.6.129.87:9092
```

```
usuario@tdm-sparkmaster: ~  
(base) usuario@tdm-sparkmaster:~$ cd /opt/app/kafka  
(base) usuario@tdm-sparkmaster:/opt/app/kafka$ sudo bin/zookeeper-server-start.sh -daemon  
[sudo] password for usuario:  
(base) usuario@tdm-sparkmaster:/opt/app/kafka$ sudo bin/kafka-server-start.sh -daemon config/server.properties  
(base) usuario@tdm-sparkmaster:/opt/app/kafka$ bin/kafka-console-producer.sh --topic our-first-topic --bootstrap-server 10.6.129.87:9092  
>Hola, como estás? Buen día, hola  
>^C(base) usuario@tdm-sparkmaster:/opt/app/kafka$  
(base) usuario@tdm-sparkmaster:/opt/app/kafka$ bin/kafka-console-producer.sh --topic our-first-topic --bootstrap-server 10.6.129.87:9092  
>ciao come stai?  
>
```

Test de la integración entre REDIS, Kafka y Spark

Arrancamos el servidor de Jupyter lab si ya no lo hicimos y ejecutamos el notebook `spark_redis`:

SparkSession - in-memory

SparkContext

Spark UI

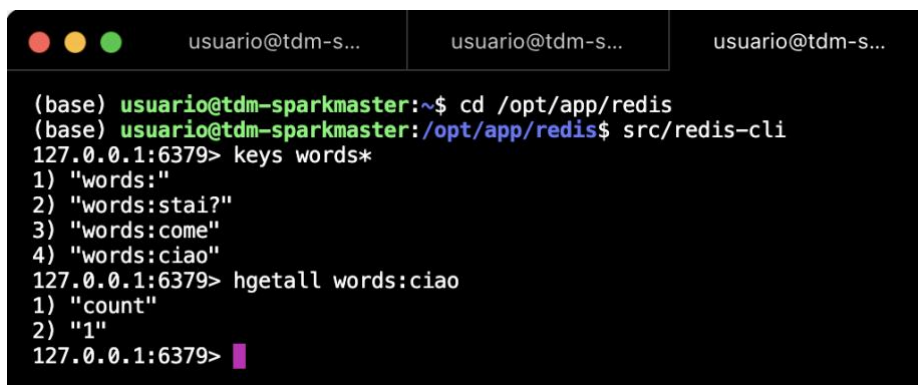
Version v3.0.3
Master spark://10.6.129.87:7077
AppName SparkRedis

```
[12]: redisTable= spark \  
      .read \  
      .format("org.apache.spark.sql.redis") \  
      .option("table", "words") \  
      .option("key.column", "word") \  
      .load()  
      redisTable.printSchema()  
      redisTable.show()  
  
root  
 |-- word: string (nullable = true)  
 |-- count: long (nullable = false)  
  
+----+-----+  
| word|count|  
+----+-----+  
|    |    |  
| ciao|    1|  
| come|    1|  
| stai?|    1|  
+----+-----+
```

Abrimos el cliente Redis en otra terminal para inspeccionar las claves y los valores almacenados en la base de datos.

Con el comando `keys words*` listamos todas las claves que comienzan con el prefijo `words`. Esto nos permite identificar todas las entradas relacionadas con las palabras procesadas por Spark y almacenadas en Redis.

Con el comando `hgetall words:ciao` inspeccionamos todos los campos y valores asociados con la clave `words:ciao`. En este caso, vemos que el único campo almacenado es `"count"` y su valor es `"1"`, lo que indica que Spark Structured Streaming calculó que la palabra `"ciao"` apareció 1 vez y lo almacenó en Redis.



```
(base) usuario@tdm-sparkmaster:~$ cd /opt/app/redis
(base) usuario@tdm-sparkmaster:/opt/app/redis$ src/redis-cli
127.0.0.1:6379> keys words*
1) "words:"
2) "words:stai?"
3) "words:come"
4) "words:ciao"
127.0.0.1:6379> hgetall words:ciao
1) "count"
2) "1"
127.0.0.1:6379> █
```

Cerramos el kernel para concluir esta prueba.

Instalación y configuración de Cassandra

Descargamos Cassandra, descomprimos el paquete y lo instalamos en `/opt/app`.

Después de cambiar el propietario del directorio y hacer el enlace simbólico, editamos `etc/profile.d/cassandra.sh`, el fichero de perfil del sistema, para incluir las variables del sistema:



```
GNU nano 6.2 /etc/profile.d/cassandra.sh
export CASSANDRA_HOME=/opt/app/cassandra
export PATH=$PATH:$CASSANDRA_HOME/bin
```

Cargamos estas variables y creamos los directorios locales cambiándoles su propietario:

```
$ source /etc/profile.d/cassandra.sh
$ cd cassandra; sudo mkdir data logs; sudo chown -R
usuario:usuario data logs
```

Cambiamos la configuración por defecto para permitir clientes y conexiones remotas a través del fichero `conf/cassandra.yaml`, añadiendo la dirección IP del Spark Máster

```
# For security reasons, you should not expose this port to the internet.
#rpc_address: localhost
rpc_address: 10.6.129.87
```

Ponemos Cassandra en background y abrimos dos otras ventanas del terminal para:

- arrancar el servidor de Jupyter lab
- arrancar el cliente CQL ((Cassandra Query Language)

```
(base) usuario@tdm-sparkmaster:/opt/app/cassandra$ bin/cqlsh 10.6.129.87
Connected to Test Cluster at 10.6.129.87:9042
[cqlsh 6.1.0 | Cassandra 4.1.7 | CQL spec 3.4.6 | Native protocol v5]
Use HELP for help.
cqlsh>
```

Arrancando el cliente CQL (*Cassandra Query Language*) es posible:

- Con el comando HELP, visualizar el listado de comandos disponibles en

CQL

```
cqlsh> HELP

Documented shell commands:
=====
CAPTURE CLS COPY DESCRIBE EXPAND LOGIN SERIAL SOURCE UNICODE
CLEAR CONSISTENCY DESC EXIT HELP PAGING SHOW TRACING

CQL help topics:
=====
AGGREGATES                CREATE_KEYSPACE          DROP_TRIGGER            TEXT
ALTER_KEYSPACE            CREATE_MATERIALIZED_VIEW DROP_TYPE               TIME
ALTER_MATERIALIZED_VIEW   CREATE_ROLE              DROP_USER               TIMESTAMP
ALTER_TABLE              CREATE_TABLE             FUNCTIONS              TRUNCATE
ALTER_TYPE               CREATE_TRIGGER           GRANT                  TYPES
ALTER_USER              CREATE_TYPE              INSERT                 UPDATE
APPLY                   CREATE_USER              INSERT_JSON            USE
ASCII                   DATE                     INT                    UUID
BATCH                   DELETE                  JSON
BEGIN                  DROP_AGGREGATE          KEYWORDS
BLOB                   DROP_COLUMNFAMILY      LIST_PERMISSIONS
BOOLEAN                DROP_FUNCTION          LIST_ROLES
COUNTER                DROP_INDEX             LIST_USERS
CREATE_AGGREGATE        DROP_KEYSPACE          PERMISSIONS
CREATE_COLUMNFAMILY     DROP_MATERIALIZED_VIEW REVOKE
CREATE_FUNCTION         DROP_ROLE              SELECT
CREATE_INDEX           DROP_TABLE             SELECT_JSON
```

- Con el comando DESCRIBE CLUSTER; se puede visualizar la información básica sobre el clúster, configurada en el fichero `cassandra.yaml`:

```
cqlsh> DESCRIBE CLUSTER

Cluster: Test Cluster
Partitioner: Murmur3Partitioner
Snitch: DynamicEndpointSnitch
```

- Con el comando DESCRIBE KEYSPACES; se obtiene un listado de todos los keyspaces del clúster:

```
cqlsh> DESCRIBE KEYSPACES
```

```
mykeyspace  system_auth      system_schema  system_views
system      system_distributed  system_traces  system_virtual_schema
```

- Con el comando `SHOW VERSION`; podemos ver la versión de Cassandra instalada.

```
cqlsh> SHOW VERSION
```

```
[cqlsh 6.1.0 | Cassandra 4.1.7 | CQL spec 3.4.6 | Native protocol v5]
```

En Cassandra, una base de datos se define como *KeySpace*. Hemos creado la nuestra con el comando:

```
cqlsh> CREATE KEYSPACE mykeyspace WITH REPLICATION = { 'class' :
'SimpleStrategy', 'replication_factor' : 1 };
```

Y para empezar a utilizarla:

```
cqlsh> USE mykeyspace;
```

```
cqlsh:mykeyspace>
```

Se puede notar que cuando se empieza a utilizar un keyspace, su nombre viene a lado del prompt `cqlsh`.

Creemos la tabla *words* para después almacenar los datos que provienen de Kafka:

```
cqlsh:mykeyspace> CREATE TABLE words (word_id text PRIMARY KEY, count
int);
```

También en este caso tenemos que arrancar los demonios de Zookeeper y Kafka, además de un producer de Kafka en nuestro terminal (o lo de Jupyter lab).

Test de la integración entre Cassandra, Kafka y Spark

En Jupyter Lab ejecutamos el notebook **spark_cassandra**:

```
[4]: SparkSession - in-memory
```

```
SparkContext
```

```
Spark UI
```

```
Version      v3.0.3
```

```
Master       spark://10.6.129.87:7077
```

```
AppName      SparkCassandra
```

```
[21]: df = spark \
      .read \
      .format("org.apache.spark.sql.cassandra") \
      .options(table="words", keyspace="mykeyspace") \
      .load() \
      .select("word_id", "count") \
      .where("count > 0") \
      .show()
```

word_id	count
hello	2
bye	2

En los terminales del producer y consumer podemos ver los datos y sus procesamiento con MapReduce de Spark Structured Streaming

```
(base) usuario@tdm-sparkmaster: /opt/app/kafka$ bin/kafka-console-producer.sh
topic --bootstrap-server 10.6.129.87:9092
>Hello bye hello bye
```

```
Batch: 1
+-----+-----+
|word_id|count|
+-----+-----+
|  hello|    2|
|   bye|    2|
+-----+-----+
```

También en la terminal donde tenemos cqlsh, podemos comprobar que se está escribiendo correctamente lo que lanzamos desde Kafka, mirando al contenido de la tabla words:

```
cqlsh:mykeyspace> SELECT * FROM words;
```

word_id	count
hello	2
bye	2

MongoDB

Es un claro ejemplo de cómo el paradigma de las bases de datos relacionales ha evolucionado.

Sus estructuras de datos son:

- Document: equivale a una fila.
- Field: equivale a una columna.
- Collection: equivale a una tabla.

Esta base de datos resulta muy potente para almacenar datos no estructurados y es ideal para casos en los que la flexibilidad del esquema y el escalado horizontal son prioritarios.

Instalación y configuración de Mongo

Los step de instalación se repiten:

- Descargamos la última versión de MongoDB:
- Descomprimos el paquete en el directorio /opt/app:
- Cambiamos el propietario del directorio y hacemos un enlace simbólico:
Editamos el fichero `/etc/profile.d/mongodb.sh`, lo del perfil del sistema, para incluir las variables del sistema:

```
GNU nano 6.2 /etc/profile.d/mongodb.sh
export MONGODB_HOME=/opt/app/mongodb
export PATH=$PATH:$MONGODB_HOME/bin
```

- Ejecutamos ese fichero para cargar las variables:
`$ source /etc/profile.d/mongodb.sh`
- Creamos los directorios por defecto que usa MongoDB y le cambiamos el propietario

A este punto, antes de probar la integración, arrancamos los siguientes demonios:

- Lo de mongod
`$ mongod --dbpath /var/lib/mongo --logpath /var/log/mongodb/mongod.log --fork --bind_ip 10.6.129.87`
- Zookeeper
- Kafka
- Producer de Kafka

Test de la integración entre MongoDB, Kafka y Spark

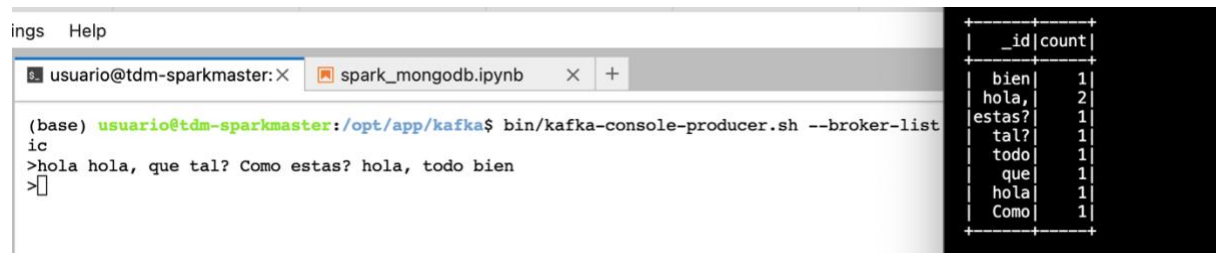
Arrancamos Jupyter lab y ejecutamos el notebook `spark_mongodb`:

SparkSession - in-memory

SparkContext

[Spark UI](#)

Version v3.0.3
Master spark://10.6.129.87:7077
AppName SparkMongoDB



The screenshot shows a JupyterLab interface. On the left, a terminal window displays the command `bin/kafka-console-producer.sh --broker-list` and the output `hola hola, que tal? Como estas? hola, todo bien`. On the right, a data table is displayed with two columns: `_id` and `count`. The table contains the following data:

_id	count
bien	1
hola,	2
estas?	1
tal?	1
todo	1
que	1
hola	1
Como	1

Conclusiones:

Hemos explorado el procesamiento de datos distribuidos con Spark, el manejo de datos en tiempo real con Kafka y Spark Structured Streaming, y hemos integrado estas tecnologías con bases de datos NoSQL. Esto nos ha permitido demostrar cómo el paradigma de datos ha evolucionado, pasando de estructuras relacionales tradicionales a modelos capaces de manejar datos no estructurados y escalar de manera eficiente en entornos distribuidos.