Apache Hadoop

Competencias

- Conocer los principales componentes del ecosistema Hadoop.
- Conocer cómo funciona la escritura y lectura de archivos en el Hadoop Distributed File System.
- Conocer los términos de Maestro y Esclavo en el ecosistema Hadoop.
- Conocer cómo se implementa el manejo de recursos mediante YARN.
- Comprender el uso y cómo se relacionan HDFS, YARN y Hadoop Streaming.
- Implementar MapReduce mediante el jar de Hadoop Streaming en AWS EMR.

Motivación

Ya tenemos conocimiento sobre cómo opera el paradigma MapReduce: **Divide y vencerás**. Buscamos dividir nuestra tarea en dos operaciones: una que permita mapear y asignar a múltiples trabajadores, y otra que permita recolectar el procesamiento de los trabajadores. Hasta el momento la implementación ha estado limitada a `Python` y `bash-shell` de Linux.

En esta lectura aprenderemos sobre Hadoop, una suite de software orientada a hacer el proceso de distribuir archivos e implementar tareas más afables. A lo largo de esta lectura aprenderemos cuáles son las principales capas que componen Hadoop, así como su funcionamiento e interacción con otros componentes. Para generar una representación distribuida de los datos utilizaremos el *Hadoop Distributed File System*, conoceremos cómo se asignan tareas mediante YARN, y finalmente implementaremos *Hadoop Streaming MapReduce* para implementar nuestros scripts en Python en un flujo de trabajo desde AWS EMR. Adicionalmente, aprenderemos a identificar los componentes de Hadoop ya instalados en nuestra instancia AWS EMR.

¿Qué es Hadoop?

Hadoop (Apache Hadoop) es un **modelo de programación** para el procesamiento de datos. Se puede entender como un compendio de softwares orientados a la implementación de una red de computadores para resolver cálculos de cantidades masivas de datos. A grandes rasgos, el ecosistema Hadoop lo podemos segmentar en tres capas:

- **Capa de Almacenaje**: Mediante *Hadoop Distrtibuted File System* (de aquí en adelante, HDFS), implementamos un sistema de administración de archivos de manera distribuida que almacena información. Tiene sus orígenes en el sistema de almacenamiento distribuido de Google (Google File System).

- **Capa de Administración de Recursos**: Hadoop YARN (Yet Another Resource Manager) se incorporó en el 2012 como una de las capas base del ecosistema. Es la plataforma encargada de la administración de recursos computacionales en los clusters existentes y asignarlos en las aplicaciones agendadas por el/los usuario/s.

- **Capa de Aplicaciones**: A diferencia de la implementación primitiva Hadoop 1.x, donde sólo se disponía de MapReduce, en la última versión de Hadoop se implementan múltiples módulos encargados de flexibilizar el rango de acción de nuestras tareas. Estas abarcan desde la implementación de modelos de Machine Learning, la integración con bases de datos relacionales, consultas relacionales y no relacionales, y la creación de lenguajes de scripting transpilados a código Map Reduce.

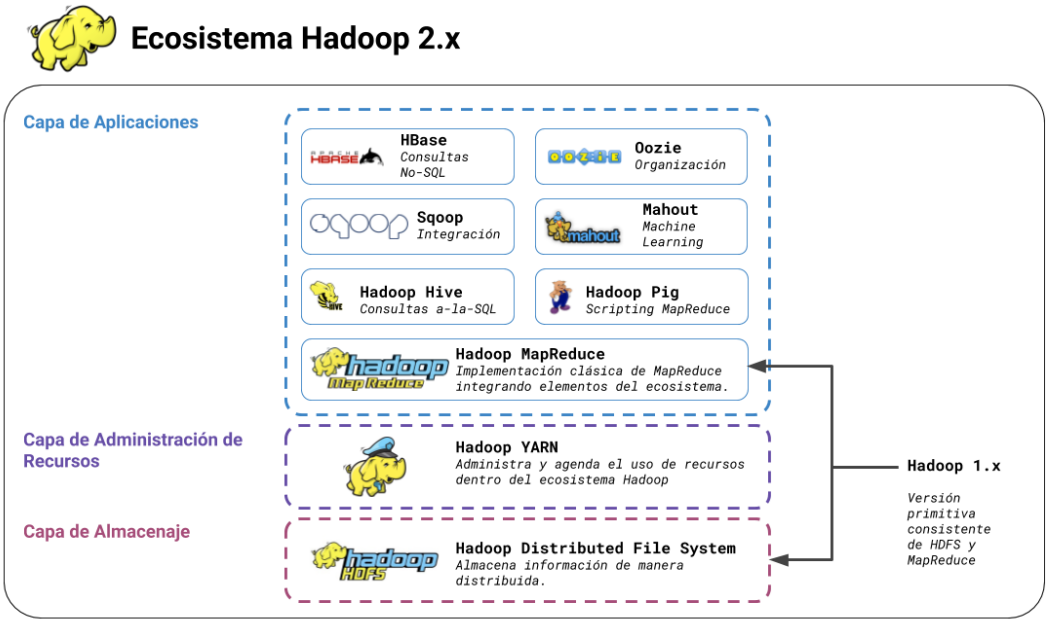

Imagen 1. hadoop.

Arquitectura de Hadoop

Por lo general, nosotros trabajaremos entre la capa de almacenaje y la capa de aplicaciones, relegando el funcionamiento de la capa de administración de recursos a un segundo plano. Dentro del ecosistema de Hadoop, podemos asociar nuestras tareas al almacenamiento de información (mediante `HDFS`) o la ejecución de instrucciones (mediante `MapReduce`, `Hive`, `Spark`, entre otros). Por lo general hablaremos de nodos **Maestros**, aquellos diseñados para el despacho de tarea a nodos **Esclavos**, que son los encargados de ejecutar la tarea encomendada. Mientras que la acción de un nodo maestro puede variar dependiendo si corresponde una tarea de almacenaje o de ejecución, todos los nodos esclavos podrán ejecutar tareas relacionadas al almacenaje o ejecución.

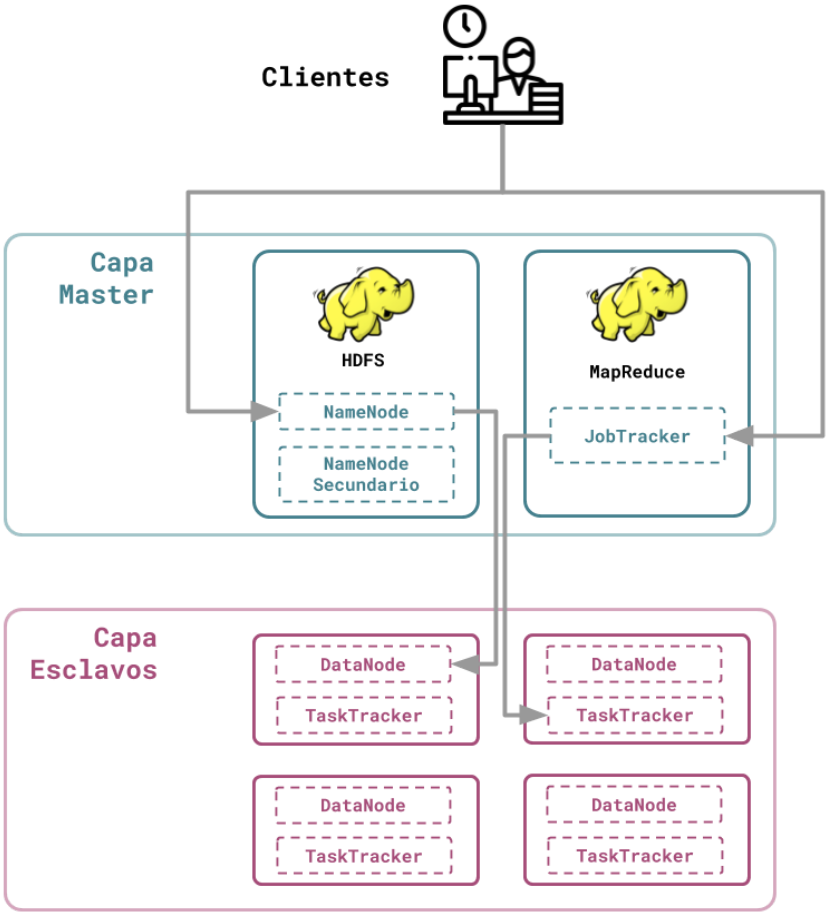

Imagen 2. Arquitectura.

Existen dos tipos de nodos que controlan el proceso de ejecución de tareas:

- **JobTrackers**: El JobTracker es el que recibe la instrucción de trabajo por parte del cliente. En conocimiento de la tarea encomendada, éste determina los nodos esclavos a implementar y rescatar información.

- **TaskTrackers**: Un TaskTracker se comunicará constantemente con el JobTracker respecto a la tarea encomendada. Si este deja de reportar sobre su estado actual en la tarea dado un periodo determinado de tiempo, El JobTracker asumirá que está muerto y encomendará la tarea a otro nodo en el cluster.

HDFS: Hadoop Distributed File System

Cuando un conjunto de datos supera sustancialmente su alojamiento físico, es necesario realizar una partición de ésta a lo largo de múltiples máquinas. Estas máquinas se encuentran organizadas en sistemas distribuidos. Uno de los problemas asociados a la implementación de estos es la mantención de la red de conexión entre las máquinas. No solo es costoso, también es proclive a fallas. Uno de los casos paradigmáticos es evitar que el sistema pierda datos cuando uno de los nodos que componen el sistema se caiga.

Partamos por estudiar y definir la capa de almacenamiento en el ecosistema. HDFS es un sistema de administración para almacenar archivos de grandes dimensiones, orientado a patrones de procesamiento streaming, que está implementado en un cluster de equipos. Esto significa que las aplicaciones e instrucciones indicadas se ejecutan directamente en el proceso MapReduce. Posteriormente visitaremos cómo podemos integrar nuestros scripts primitivos de MapReduce en el ecosistema.

HDFS se diseñó alrededor de la idea que el patrón más eficiente de procesamiento de datos es **write once, read many**. Por lo general, el trabajo con datos tiene el siguiente flujo: Partimos de un conjunto de datos generados o copiados e implementamos análisis **en estos**. Cada análisis puede involucrar una parte substancial de los datos asociados a un fenómeno, por lo que se prioriza tener un tiempo de lectura más rápida por sobre la capacidad de modificar los registros.

Dado esta orientación a **write once, read many**, HDFS acerca la lógica de procesamiento a los datos (acoplamos nuestro proceso lógico de extracción a los datos) y no al revés, situación donde los datos pueden perder integridad y características dado el procesamiento y asimilación al proceso lógico.

Componentes de HDFS: Bloques

Para entender cómo HDFS logra dividir un archivo en múltiples partes y cómo mantiene un registro la identificación de cada parte en el sistema, implementa la división en **bloques** y la identificación de cada uno de estos en **NameNodes** y **DataNodes**.

En HDFS, los archivos se dividen en bloques que se almacenan como unidades independientes dentro del cluster. Por tanto, las distintas partes de un mismo archivo se encuentran físicamente distribuidas en las múltiples máquinas del cluster.

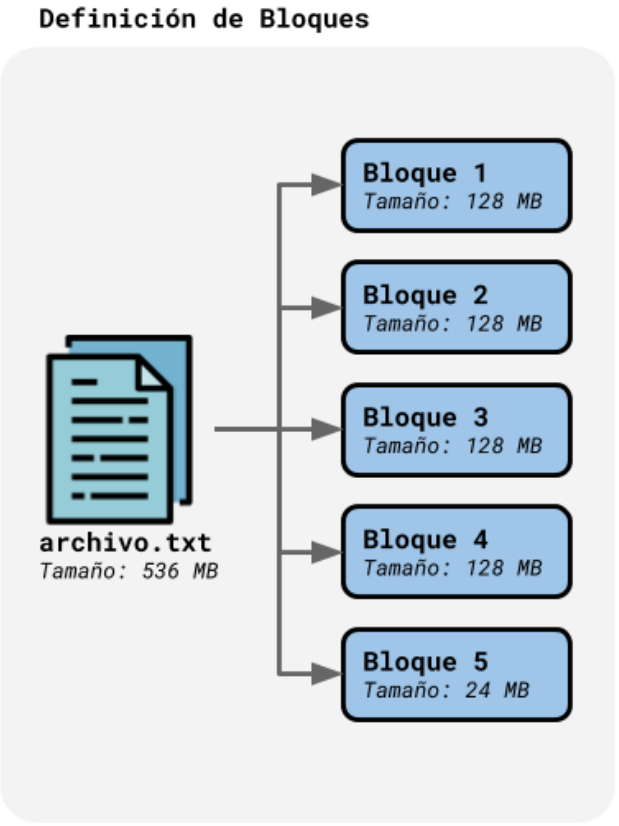

Imagen 3. Bloques.

Tomemos el siguiente ejemplo: Supongamos que tenemos un archivo que deseamos distribuir en HDFS. Este tiene un peso total de 536 MB. HDFS tomará el archivo y buscará representarlo en bloques de menor tamaño, por defecto uno de 128 MB. Todos los bloques que representan el archivo son del mismo tamaño a excepción del último que puede ser menor o igual en tamaño. La aplicación de Hadoop será la encargada de asignar una pertenencia a cada bloque.

Por defecto el tamaño de los bloques es de 128 MB. Esta decisión se toma en función a la cantidad de metadatos (Datos que hacen referencia a la ubicación del bloque, pero no contiene información relevante para el usuario). Supongamos el siguiente ejercicio: Tenemos este archivo distribuido, el cual podríamos particionar en tamaños de 4 MB, dando un total de 134 bloques de igual tamaño. El problema asociado es que la cantidad de metadatos que hacen referencia a cada bloque aumentará de manera substancial. Así, aumentamos de manera innecesaria el tiempo de lectura e identificación de los componentes.

Ventajas de particionar a nivel de bloque

1. Podemos asignar archivos del orden de Tera y Petabytes difíciles de contener en un computador singular, a lo largo de múltiples clusters aprovechando sus discos duros.

2. Simplifica el subsistema de almacenamiento al fijar el tamaño de cada bloque (a diferencia de hacer responsivo el tamaño condicional al peso del archivo) y preservar la metadata en un bloque separado (evitando la duplicación innecesaria de información).

3. Para asegurar el correcto funcionamiento frente a bloques dañados, fallas en discos y máquinas; cada bloque se replica en una pequeña cantidad a lo largo de máquinas que pueden estar separadas físicamente. Si un bloque deja de estar disponible, se puede leer una copia desde otra ubicación. Un bloque que ya no está disponible dado la corrupción o la falla de la máquina, se puede replicar desde sus ubicaciones alternativas a otras máquinas en vivo para que el factor de replicación vuelva al nivel normal.

Componentes de HDFS: Namenodes y Datanodes

Hasta el momento tenemos nuestro archivo particionado en bloques. Un problema de esto es que todavía no tenemos orden asociado a cada bloque. Para solucionar esto, en HDFS existe la figura de **nodos** que operan en un patrón Maestro-Esclavo:

- **NameNode**: Corresponde al maestro. Está encargado de la gestión de los espacios de nombre (Namespace) en el sistema de archivos. Mantiene el árbol del sistema de archivos y los metadatos de todos los archivos y directorios en el árbol. La información se aloja en el disco mediante dos archivos: **imagen del espacio de nombres** y el **registro de edición**.

- **DataNode**: Corresponde al esclavo. Está encargado de almacenar y recuperar bloques de información cuando se les indique (por instrucción del cliente o el NameNode. Informan periódicamente a NameNode sobre los bloques alojados. Un DataNode puede comunicarse con otros Data Nodes para replicar bloques por redundancia. Cada bloque de un archivo se replica por lo menos tres veces en distintos DataNode, de manera tal de evitar problemas de máquina o de disco.

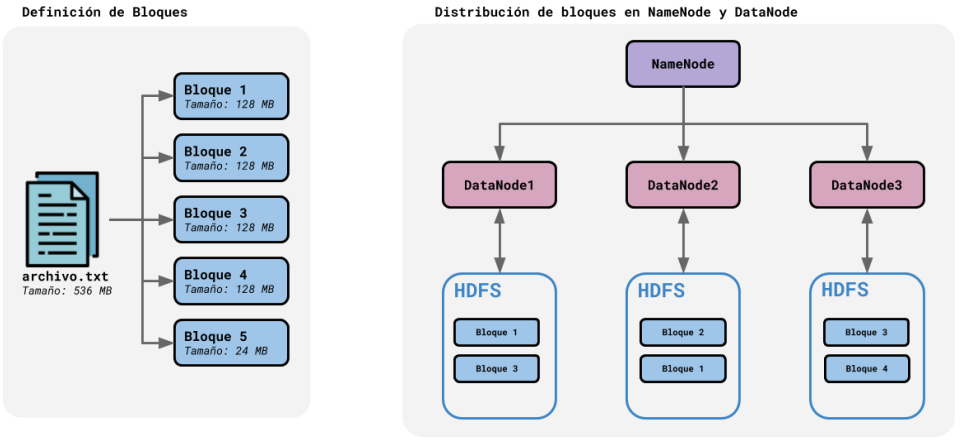

Imagen 4. Distribución de bloques.

El sistema de archivos no puede funcionar sin un NameNode. Si este es destruido, todos los archivos alojados en los Data Nodes no tendrían uso ni forma de poder reorganizarlos. Para evitar este problema con el NameNode, Hadoop proporciona dos mecanismos:

- **Copia de seguridad**: Hadoop realiza copias del estado persistente de los metadatos.

- **NameNode Secundario**: Su función es combinar periódicamente la imagen del espacio de nombres con el registro de edición. Este generalmente se ejecuta en una máquina física separada. Si bien se puede utilizar para recuperar los metadatos de los bloques, por lo general presenta un rezago respecto al NameNode primario, y en casos de pérdida total del NameNode primario, la pérdida de información es casi segura.

Ya identificando el funcionamiento de los NameNode y DataNode, entendamos a groso modo cómo opera la lectura y escritura de los datos. Partamos por analizar cómo funciona la escritura de un archivo desde el local a HDFS.

Fase de Escritura de Datos

Caveat: *Una exposición técnica al proceso de escritura se puede encontrar en White, t (2014), pp.72-74: Data Flow. Anatomy of a File Write. Ch 3: The Hadoop Distributed Filesystem.*

Por la fase de escritura de datos, hacemos referencia al proceso de cargar los datos en nuestro entorno local hacia HDFS. En la siguiente imágen se puede identificar el proceso:

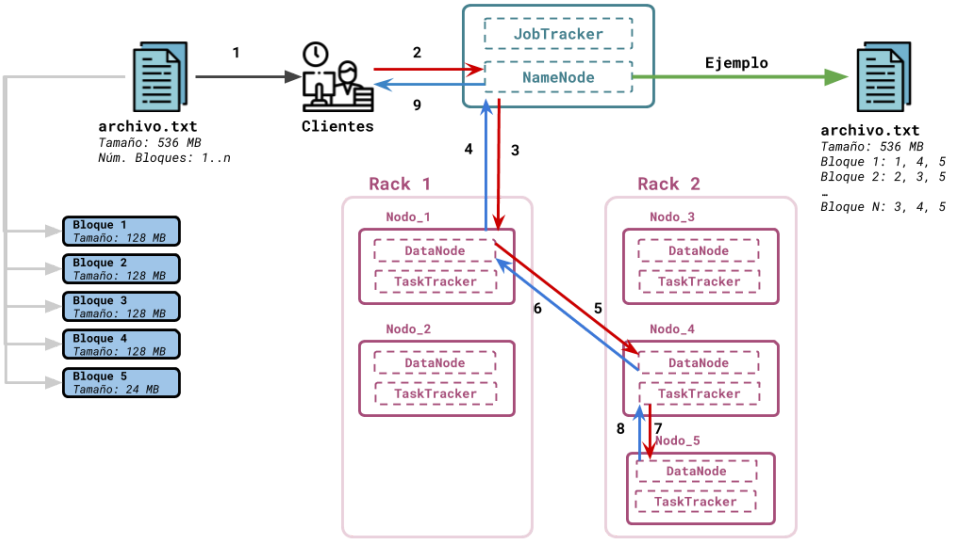

Imagen 5. Fase Escritura.

- **1:** El cliente realiza una llamada RPC (Remote Procedure Call: Llamada remota a un proceso) al NameNode donde solicita una lista de todos los Data Nodes disponibles. Antes de seleccionar el DataNode de la primera réplica, el NameNode genera ID's únicas a nivel de bloque.

- **2:** Antes de iniciar la copia, el NameNode genera un Pipeline de tres Data Nodes por los cuales viajarán las réplicas de los datos. Este Pipeline se instruye en el proceso de copia, designando en qué nodo y rack se implementará el proceso.

- **3 y 4:** En la primera copia, primero se realiza la **Escritura** y posteriormente se hace la **Notificación al NameNode**. En base a la notificación de almacenaje exitoso, se procede al siguiente DataNode.

- **5 y 6:** La segunda copia se implementa de similar manera siguiendo los pasos de **Escritura** y **Notificación al NameNode**.

- **7 y 8:** La tercera copia se implementa de similar manera siguiendo los pasos de Escritura y Notificación al NameNode.

- **9:** Se cierra la transacción de datos y se notifica sobre la copia.

Fase de Lectura de Datos

Caveat: Una exposición técnica al proceso de lectura se puede encontrar en White (69-71): Data Flow. Anatomy of a File Read. Ch 3: The Hadoop Distributed Filesystem.

Por la fase de lectura de datos, hacemos referencia al proceso de leer los datos alojados en nuestro entorno Hadoop en HDFS. En la siguiente imágen se puede identificar el proceso:

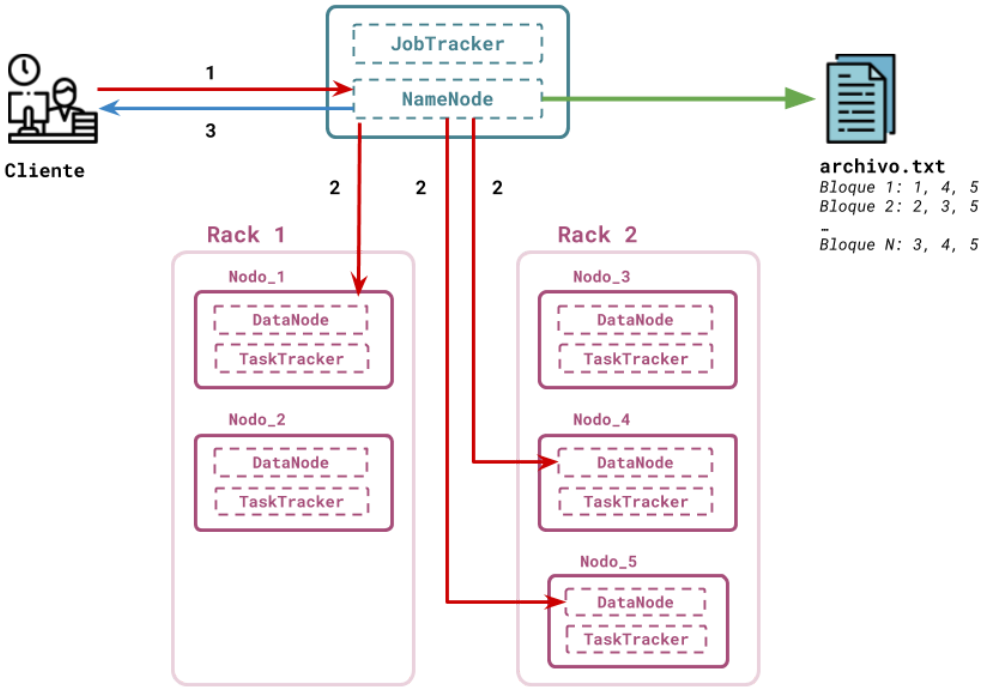

Imagen 6. Lectura de Datos.

- **1:** El cliente realiza una llamada RPC al NameNode para determinar la ubicación de los bloques constituyentes del archivo en cuestión. Para cada bloque identificado, el NameNode retorna los ID's de cada DateNode que presentan la copia de ese bloque. Antes de leer los bloques, la lista de ID's se ordena en función a la proximidad del cliente al nodo.

- **2:** El cliente se conecta al primer bloque más cercano. Los datos contenidos se envían al cliente. Una vez que se alcance el final del bloque, la conexión temporal se cierra. El proceso se repite con el siguiente bloque en la lista.

- **3:** Se devuelven los datos al usuario y posteriormente se cierra la conexión.

YARN: Yet Another Resource Management

YARN es el administrador de recursos y calendarización de trabajos en el ecosistema Hadoop. Este se implementó en Hadoop 2 para mejorar la implementación de MapReduce, pero se puede utilizar para apoyar el trabajo de otros paradigmas de computación distribuida. Si bien presenta una API mediante la cual el usuario puede administrar los clusters, por lo general es un trabajo que relegamos a aplicaciones como MapReduce, Hive, Spark, entre otros. Así, mantendremos nuestro conocimiento sobre YARN en lo mínimo.

El servicio de YARN se implementa mediante tres daemons (procesos latentes):

- **Administrador de recursos** (`Resource Manager`): Encargado de administrar el uso de recursos globales en el cluster.

- **Administrador de nodos** (`NodeManager`): Encargado del monitoreo y ejecución de contenedores. Los contenedores ejecutan un proceso específico a una aplicación **con un conjunto limitado de recursos** tales como Memoria RAM y CPU.

- **Ejecutor de aplicaciones** (`ApplicationMaster`): Gestiona la calendarización de aplicaciones y la ejecución de sus tareas.

El proceso de implementación de una aplicación en YARN se visualiza en la siguiente imagen:

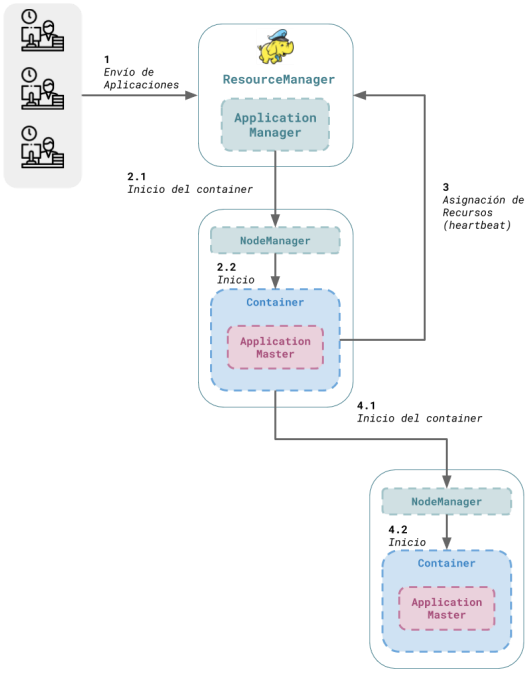

Imagen 7. Container.

1. **Envío de aplicaciones**: Un cliente contacta al administrador de recursos y solicita ejecutar un proceso de aplicación maestra.

2. **Inicio del container**: El administrador de recursos busca un administrador de nodos que pueda ejecutar la aplicación maestra en un container. En base a esta etapa, se genera un proceso de asignación de recursos en función a las demandas de la aplicación maestra. Existen dos alternativas para solucionar la asignación:

3. **Asignación de recursos**: Puede generar una solicitud de mayor asignación de recursos al container. Esto se conoce como Heartbeat.

4. **Inicio del container**: O puede generar una solicitud de más containers para generar un proceso distribuido.

Al igual que con HDFS, lo que determina si está ejecutando solo uno o dos Resource Managers por cluster es la tolerancia a interrupciones de YARN. Con solo un Resource Manager -si éste se cae, su cluster ya no puede aceptar nuevos jobs y proporcionar algunas de las capacidades de YARN para administrar los jobs que se están ejecutando (aunque los trabajos existentes que se ejecutan estarán bien). Con dos Resource Manager ejecutándose en una configuración Activo/En espera, el que está En espera puede asumir el control en los casos en que el Activo falla o necesita desactivarse para su mantenimiento.

Práctico 1: Copiando archivos en HDFS

Para practicar la administración de archivos en el ecosistema Hadoop, realizaremos nuestra primera importación de datos a HDFS. Partamos por generar nuestra instancia de trabajo con AWS EMR. Recuerden que a grandes rasgos la creación de la instancia se compone de los siguientes pasos:

1. Ingresar a la Consola AWS.
2. Seleccionar AWS EMR desde "Find Services".
3. Seleccionamos "Create cluster" (o podemos clonar la configuración de alguno)
4. Nos aseguramos de tener nuestro keypair con el cual configuraremos el acceso.
5. Accedemos utilizando ssh y manejamos archivos entre nuestro computador y la instancia con scp.

La principal virtud de implementar una instancia de trabajo con AWS EMR es el hecho que la configuración del ambiente de trabajo en Hadoop es tedioso. De hecho, la instalación de la suite no es compleja, pero hay que configurar varios puntos respecto a servidores y nombres de nodos. Para efectos prácticos del curso (y de sus trabajos), por lo general preferimos trabajar con instancias ya configuradas como las provistas con AWS EMR.

Verificando la configuración de nuestra instancia AWS EMR.

El primer punto a tener en consideración es verificar que los componentes efectivamente existen en nuestro computador. Para lograr esto, podemos escribir en la consola which, seguido por el nombre del programa. En este caso, vamos a evaluar si es que hadoop, hdfs y yarn se encuentran instalados en nuestra instancia de trabajo. De manera adicional, identifiquemos la versión de Python por defecto que utiliza nuestra instancia y si es que existe nuestra versión preferida de Python 3.6.

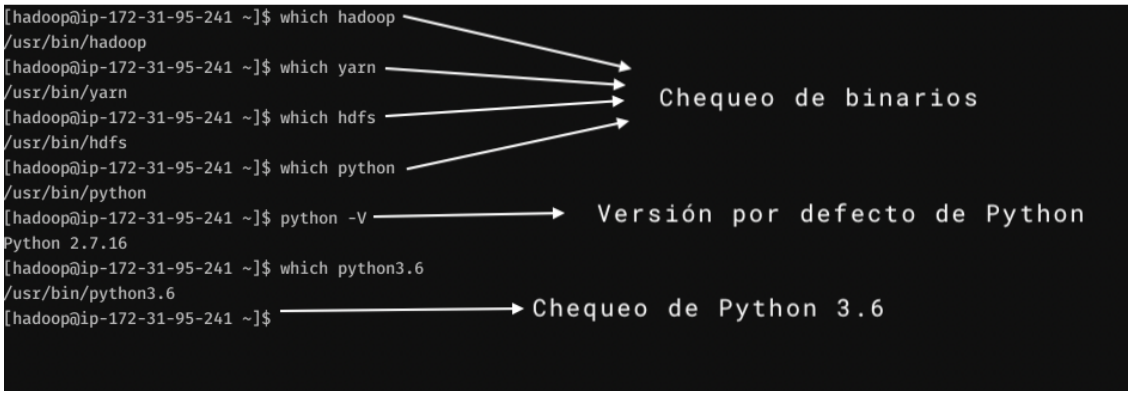

Imagen 8. Chequeo.

Si todo sale bien, la consola nos debería retornar algo similar a `/usr/bin/hdfs`. La extensión `/usr/bin` es el directorio donde se alojarán todos aquellos archivos ejecutables listos para usar. La existencia de los archivos hadoop, hdfs y yarn nos permitirá ejecutar acciones con estos. Es una buena práctica asegurar dónde van a existir los archivos, dado que podemos acceder a la ubicación exacta de alguna versión.

Carga de datos en HDFS desde un local

Partamos por un ejercicio simple: Dentro de nuestra instancia de trabajo, generamos una carpeta llamada `train_data` **en el root `/` de nuestro sistema HDFS** con el comando hdfs `dfs -mkdir /train_data`. Como se aprecia, las instrucciones dentro de HDFS son idénticas a las implementadas en un sistema Linux, con la salvedad que deben ser precedidas de `hdfs dfs` y el comando debe ser precedido por un signo menos `-`. Así, si quisiéramos tener una lista de todas las carpetas que se encuentran en el root `/`, la instrucción será `hdfs dfs -ls /`

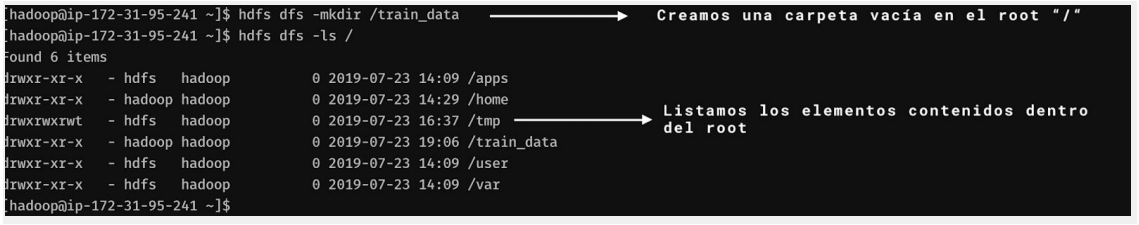

Imagen 9. hdfs-ls.

Ya tenemos nuestra carpeta dentro de HDFS, y nuestro siguiente punto es incorporar los archivos `train_delivery_data` y `test_delivery_data` que se encuentran en el bucket `s3://bigdata-desafio/challenges/u2act1/`. Para ello haremos uso de la instrucción `aws s3 cp s3://bigdata-desafio/challenges/u2act1/ train_data_local --recursive`, donde guardaremos todo el contenido del bucket en una nueva carpeta llamada `train_data_local.`

Posterior a la descarga de los archivos `csv`, podemos realizar la copia. Para este caso, vamos a implementar la instrucción `hdfs dfs -copyFromLocal train_data_local/ /train_data`, mediante la cual vamos a copiar el contenido de la carpeta a nuestra carpeta vacía creada en HDFS. Un aspecto a considerar de la instrucción `-copyFromLocal` es el hecho que funciona como una copia **restringida sólo a archivos presentes en el sistema local**. Para aquellos archivos que se encuentren en otras ubicaciones, podremos implementar variantes como `-put` o el comando `s3-dist-cp` que visitaremos más adelante. Si listamos los contenidos en la ruta de HDFS, observaremos que se encuentran en la ruta `/train_data/train_data_local/`.

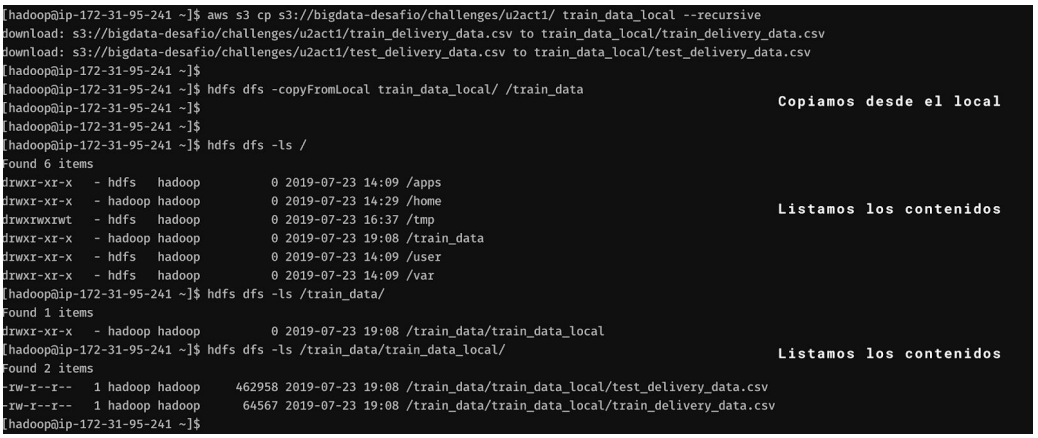

Imagen 10. contenidos.

Podemos revisar el contenido dentro de los csv en HDFS con los comandos `head`, `tail` y `cat`. Para este ejemplo vamos a ver los últimos registros en el archivo `train_delivery_data.csv` con el comando `hdfs dfs -tail /train_data/train_data_local/train_delivery_data.csv`. Un aspecto a considerar es el hecho que debemos especificar la ruta absoluta de nuestro archivo alojado en HDFS.

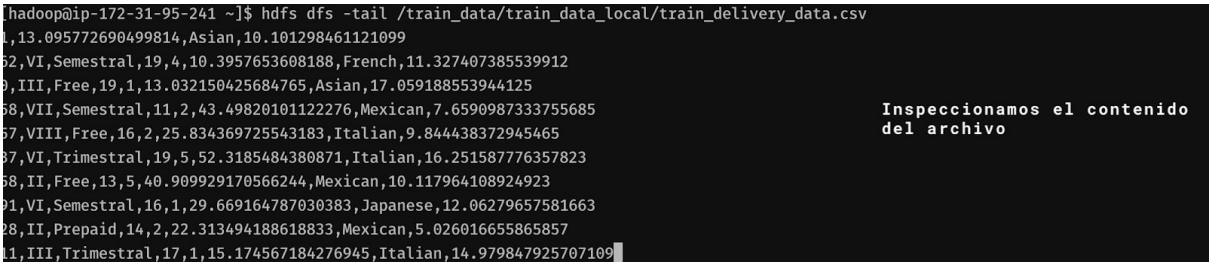

Imagen 11. Inspeccionar.

Ahora creamos un archivo llamado `lorem.txt` con la frase *"Lorem Ipsum dolor sit amet*" mediante el comando echo `"Lorem ipsum dolor sit amet" > lorem.txt"`. Este archivo lo vamos a copiar a nuestro root `/` en HDFS con el comando `hdfs dfs -put lorem.txt /`. Si todo sale bien, podremos visualizar su aparición con `hdfs dfs -ls /`. Por último, vamos a eliminarlo con `hdfs dfs -rm /lorem.txt`. Debemos tener en consideración cuál es la ruta absoluta del archivo. Si deseamos eliminar una carpeta de manera recursiva, debemos incluir `-r` en la expresión `hdfs dfs -rm -r ruta/de/carpeta .`

```
[hadoop@ip-172-31-95-241 ~]$ echo "Lorem ipsum dolor sit amet" > lorem.txt
[hadoop@ip-172-31-95-241 ~]$ hdfs dfs -put lorem.txt /
[hadoop@ip-172-31-95-241 ~]$ hdfs dfs -ls /
Found 7 items
drwxr-xr-x   - hdfs   hadoop          0 2019-07-23 14:09 /apps
drwxr-xr-x   - hadoop hadoop          0 2019-07-23 14:29 /home
-rw-r--r--   1 hadoop hadoop         27 2019-07-23 19:12 /lorem.txt
drwxrwxrwt   - hdfs   hadoop          0 2019-07-23 16:37 /tmp
drwxr-xr-x   - hadoop hadoop          0 2019-07-23 19:08 /train_data
drwxr-xr-x   - hdfs   hadoop          0 2019-07-23 14:09 /user
drwxr-xr-x   - hdfs   hadoop          0 2019-07-23 14:09 /var
[hadoop@ip-172-31-95-241 ~]$ hdfs dfs -rm /lorem.txt
Deleted /lorem.txt
[hadoop@ip-172-31-95-241 ~]$
```

Imagen 12. lorem.

Por último, podemos ver el uso de HDFS a nivel de disco y a nivel de nodo maestro. La primera instrucción es `hdfs dfs -du -h`, la cual indica que vamos a utilizar el uso de disco con `-du`, y vamos a convertir el output a humano `-h`. La segunda instrucción es `hdfs dfs -df -h`, la cual indica el tamaño del cluster, así como su uso y el espacio disponible.

```
[hadoop@ip-172-31-95-241 ~]$ hdfs dfs -du
[hadoop@ip-172-31-95-241 ~]$
[hadoop@ip-172-31-95-241 ~]$
[hadoop@ip-172-31-95-241 ~]$
[hadoop@ip-172-31-95-241 ~]$ hdfs dfs -df -h
Filesystem                                Size   Used  Available  Use%
hdfs://ip-172-31-95-241.ec2.internal:8020  1.6 T  3.9 G      1.6 T    0%
[hadoop@ip-172-31-95-241 ~]$
```

Imagen 13. hdfs.

Práctico 2: Implementado Hadoop Streaming en nuestra instancia AWS EMR

Un ejemplo tradicional de un problema que no es posible procesar en paralelo es calcular o contar el número de evaluaciones de un producto, realizadas por los usuarios de un sistema. Para cada evaluación, veremos qué nota colocó el usuario al producto y sumaremos las evaluaciones con la misma nota, independiente del producto referido. Podemos reducir el tiempo de ejecución en una tarea como esta si tenemos más máquinas y cada una está encargada de revisar un conjunto limitado de evaluaciones. Consumiremos una base de datos que se encuentra en la siguiente dirección del bucket del curso: `s3://bigdata-desafio/lecture-replication-data/movies/fullregister.data`. Los primeros registros tiene la siguiente forma:

user id	item id	rating	timestamp
196	242	3	881250949
186	302	3	891717742
22	377	1	878887116
244	51	2	880606923
166	346	1	886397596

Tabla 1. Aws.

Donde `user_id` identifica al usuario, `item_id` la película, `rating` es la puntuación asignada entre 1 y 5 y `timestamp` es un indicador sobre la fecha de emisión del voto. Este ejemplo es una implementación inspirada en los datos de **MovieLens**, parte del proyecto GroupLens de la Universidad de Minnesota.

Nuestro objetivo será implementar un contador de ocurrencias de rating de cada item, mediante Hadoop Streaming. Para lograr esto, sistematizaremos el procedimiento en una serie de pasos:

- Cargar los archivos en HDFS.
- Identificar el jar de Streaming.
- Desarrollar los scripts Mapper y Reducer.
- Implementar el streamer.
- Rescatar los resultados.

Paso 1: Carga de datos en HDFS desde un bucket s3 con s3-dist-cp

Hasta el momento, la carga de archivos se implementa con un patrón de acceso simple desde el local. Resulta que podemos implementar un patrón distribuído de copia con el comando `hdfs distcp /ruta/local /ruta/destino`. Para este ejemplo, vamos a copiar desde s3 a la distribución hdfs de la instancia con `s3-dist-cp`, que es un ejecutable similar en funcionamiento a `distcp`, pero optimizado para AWS S3.

`distcp` es implementado como un trabajo MapReduce donde la copia se paraleliza en el paso de map a lo largo del cluster. Este trabajo MapReduce no presenta pasos de reduce. Cada archivo se copia por un map, y `distcp` intenta asignar una cantidad similar de datos a cada bucket de datos. Para implementar `s3-dist-cp` debemos identificar una fuente que se enrutará en `--src=` y un destino en `--dest=`.

```
s3-dist-cp \
--src=s3://bigdata-desafio/lecture-replication-data/movies/ \
--dest=hdfs:///home/hadoop/moviedata/
```

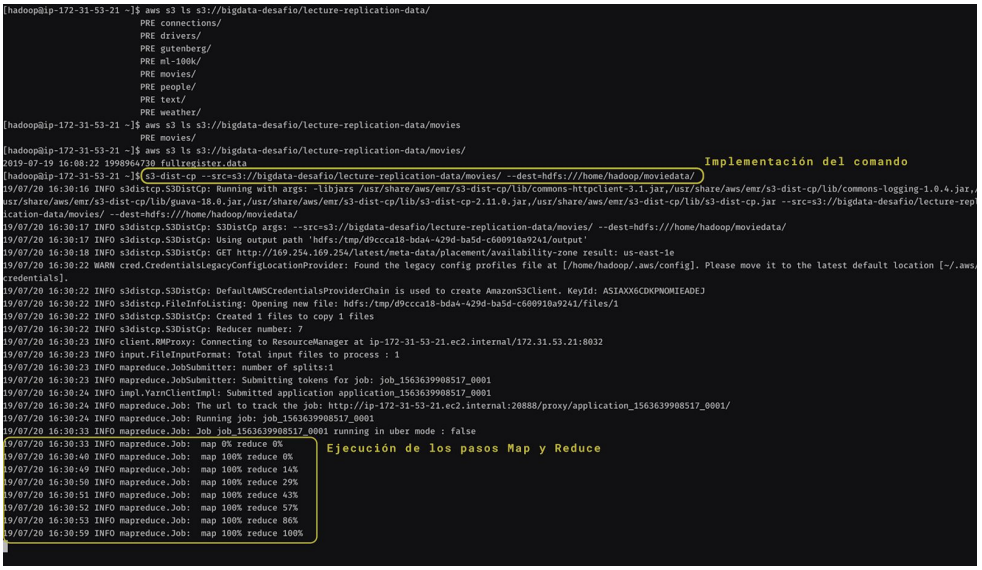

Imagen 14. implement.

Paso 2: Identificación del jar streaming

Hadoop provee de una API que permite ejecutar tareas MapReduce en múltiples lenguajes, tales como Python o R. La API Hadoop Streaming implementa flujos estándar de Unix (*Standard Input, Standard Output, Standard Error*) como la interfaz entre los componentes de Hadoop y los scripts creados por el usuario. La API se encontrará en la librería del usuario.

En nuestra instancia de trabajo AWS EMR, debemos identificar dónde se encuentra físicamente y cómo se llama. Para lograr esto vamos a implementar la sintaxis `find /usr/lib -name '*streaming*' -print`, donde buscaremos dentro de la ubicación `/usr/lib` todos aquellos elementos que contengan 'streaming' en el nombre. Para este ejemplo vamos a utilizar `/usr/lib/hadoop-mapreduce/hadoop-streaming-2.8.5-amzn-4.jar`

```
[hadoop@ip-172-31-53-21 ~]$ hdfs dfs -ls
[hadoop@ip-172-31-53-21 ~]$ hdfs dfs -ls /home/hadoop/
Found 1 items
drwxr-xr-x   - hadoop hadoop          0 2019-07-20 16:30 /home/hadoop/moviedata
[hadoop@ip-172-31-53-21 ~]$ hdfs dfs -ls /home/hadoop/moviedata
Found 1 items
-rw-r--r--   1 hadoop hadoop 1998964730 2019-07-20 16:31 /home/hadoop/moviedata/fullregister.data
[hadoop@ip-172-31-53-21 ~]$ find /usr/lib -name "*streaming*" -print
/usr/lib/hadoop-mapreduce/hadoop-streaming-2.8.5-amzn-4.jar
/usr/lib/hadoop-mapreduce/hadoop-streaming.jar
/usr/lib/hadoop/hadoop-streaming-2.8.5-amzn-4.jar
/usr/lib/hadoop/hadoop-streaming.jar
/usr/lib/hive-hcatalog/share/hcatalog/hive-hcatalog-streaming-2.3.5-amzn-0.jar
/usr/lib/hue/desktop/core/src/desktop/static/desktop/docs/impala/topics/impala_disable_streaming_preaggregations.json
/usr/lib/hue/build/static/desktop/docs/impala/topics/impala_disable_streaming_preaggregations.json
/usr/lib/hue/build/static/desktop/docs/impala/topics/impala_disable_streaming_preaggregations.2f195ab59647.json
/usr/lib/hue/build/static/jobsub/templates/actions/streaming.df9715856fd3.html
/usr/lib/hue/build/static/jobsub/templates/actions/streaming.html
/usr/lib/hue/apps/oozie/src/oozie/importlib/xslt2/workflows/0.5/nodes/streaming.xslt
/usr/lib/hue/apps/oozie/src/oozie/importlib/xslt/workflows/0.2/nodes/streaming.xslt
/usr/lib/hue/apps/oozie/src/oozie/importlib/xslt/workflows/0.1/nodes/streaming.xslt
/usr/lib/hue/apps/oozie/src/oozie/importlib/xslt/workflows/0.5/nodes/streaming.xslt
/usr/lib/hue/apps/oozie/src/oozie/importlib/xslt/workflows/0.3/nodes/streaming.xslt
/usr/lib/hue/apps/oozie/src/oozie/importlib/xslt/workflows/0.2.5/nodes/streaming.xslt
/usr/lib/hue/apps/oozie/src/oozie/importlib/xslt/workflows/0.4/nodes/streaming.xslt
/usr/lib/hue/apps/oozie/src/oozie/templates/editor2/gen/workflow-streaming.xml.mako
/usr/lib/hue/apps/oozie/src/oozie/templates/editor/gen/workflow-streaming.xml.mako
/usr/lib/hue/apps/jobsub/src/jobsub/old_migrations/0002_auto__add_ooziestreamingaction__add_oozieaction__add_oozieworkflow__ad.py
/usr/lib/hue/apps/jobsub/src/jobsub/static/jobsub/templates/actions/streaming.html
/usr/lib/oozie/lib/oozie-sharelib-streaming-5.1.0.jar
/usr/lib/oozie/embedded-oozie-server/webapp/WEB-INF/lib/oozie-sharelib-streaming-5.1.0.jar
/usr/lib/python2.7/dist-packages/awscli/customizations/streamingoutputarg.py
/usr/lib/python2.7/dist-packages/awscli/customizations/streamingoutputarg.pyo
/usr/lib/python2.7/dist-packages/awscli/customizations/streamingoutputarg.pyc
```

Imagen 15. Identify.

Paso 3: Desarrollo de los scripts Mapper y Reducer

A continuación implementaremos nuestros scripts de Map y Reduce respectivamente. Un aspecto a considerar es el hecho que están desarrollados con Python 3.6, por lo que debemos tomar un par de precauciones para su correcta implementación en Hadoop Streaming:

- **Agregar un "shebang" al inicio de los scripts**: Un shebang es una órden que se implementa al inicio de cada script, la cual hace referencia a un ejecutable específico. Dado que nuestros scripts están en Python 3.6, debemos direccionar el shebang a este ejecutable. Para poder identificar la ruta de Python 3.6, podemos hacer `which python3.6`. Para este caso, nuestro shebang será `#!/usr/bin/python3.6` y será agregado en la primera línea de nuestros mapper y reducer.

- `mapper.py`

```
#!/usr/bin/python3.6

import re, sys

feed_document = sys.stdin

for line_in_document in feed_document:
  (usr_id, movie_id, rating, timestamp) = line_in_document.split('\t')
   print(f"{rating}\t1")
```

- `reducer.py`

```
#!/usr/bin/python3.6

import sys

feed_mapper_output = sys.stdin
previous_counter = None
total_count = 0

for line_ocurrence in feed_mapper_output:
   rating, ocurrence = line_ocurrence.split('\t')

   if rating != previous_counter:
       if previous_counter is not None:
           print(f"{previous_counter}\t{str(total_count)}")
       previous_counter = rating
       total_count = 0

   total_count += int(ocurrence)

print(f"{previous_counter}\t{str(total_count)}")
```

- **Cambiar los permisos de ejecución**: El segundo elemento a considerar es el tipo de permiso de los scripts. Debemos instruir al sistema que los scripts sean ejecutables mediante la instrucción chmod `+x <script.py>.` De esta manera, indicaremos al sistema que el script será ejecutable, considerando el shebang instruido en el script.

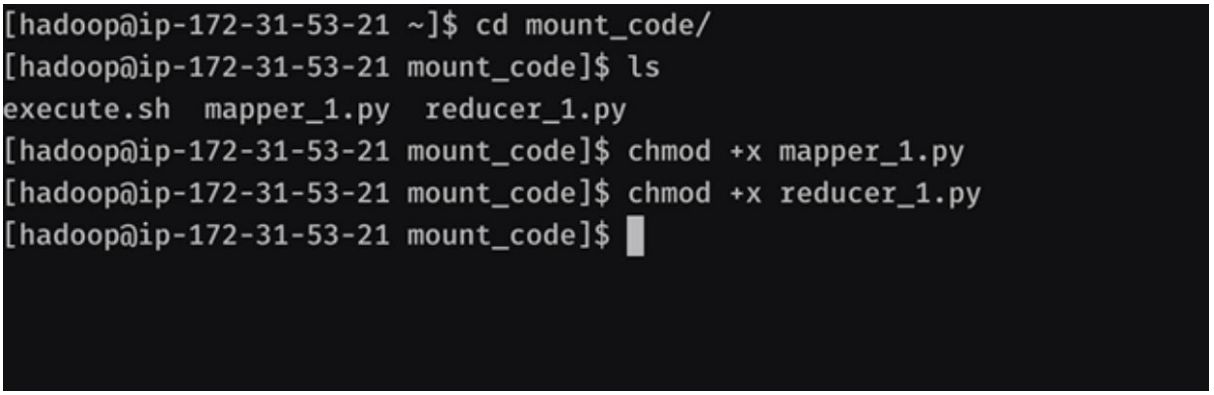

Imagen 16. Permissions.

Paso 4: Implementación del streamer

Resolviendo estos puntos, podemos ensamblar todas las partes en nuestro streamer. La instrucción es un tanto verbosa, por lo cual vamos a separarla en partes (los caracteres \ sirven como saltos de línea):

```
hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming-2.8.5-amzn-4.jar \    # Identificamos el streamer
  -file mount_code/mapper.py -mapper mount_code/mapper.py \                 # Identificamos el mapper (con shebang y permisos)
  -file mount_code/reducer.py -reducer mount_code/reducer.py \              # Identificamos el reducer (con shebang y permisos)
  -input hdfs:///home/hadoop/moviedata/fullregister.data \                  # Ruta absoluta hdfs
  -output register-results                                                  # Ruta de resultados en hdfs
```

Imagen 17. Stream.

- **Identificación del streamer**: Con la ruta absoluta del streamer identificada con `find`, la vamos a integrar dentro de la instrucción hadoop jar `/usr/lib/hadoop/hadoop-streaming-2.8.5-amzn-4.jar`. Mediante esta, le decimos a hadoop que utilizaremos un archivo con múltiples funciones y ejecutables que se encuentran empacadas en el jar.

- **Declaración del Mapper**: Debemos declarar dónde se encontrará el mapper mediante la opción `-file` y se instruirá al streamer de implementar la rutina del mapper en esta fase con la opción `-mapper.`

- **Declaración del Reducer**: Debemos declarar dónde se encontrará el reducer mediante la opción `-file` y se instruirá al streamer de implementar la rutina del reducer en esta fase con la opción `-reducer.`

- **Ruta del input**: Declaramos dónde se encuentra el/los archivos a analizar. El input de Hadoop Streaming puede ser algún archivo alojado en local o en algún protocolo como `hdfs`, `s3`, entre otros.

- **Ruta del output**: Declaramos dónde se alojarán los resultados producto del proceso Hadoop Streaming. La carpeta con resultados estará guardada dentro del sistema HDFS.

Una vez ejecutado el Streamer, analizaremos el proceso y el output que nos arroja de manera secuencial. El output inicia por entregar información sobre la conexión realizada mediante el Streamer a los Resource Managers de YARN, así como definir la cantidad de splits realizados en el documento de input. Una vez que se montan los elementos conformantes del trabajo, se inicia el proceso de Map (el cual avanza de manera progresiva), y una vez que se completa el Map, prosigue el paso Reduce que también avanza de manera progresiva.

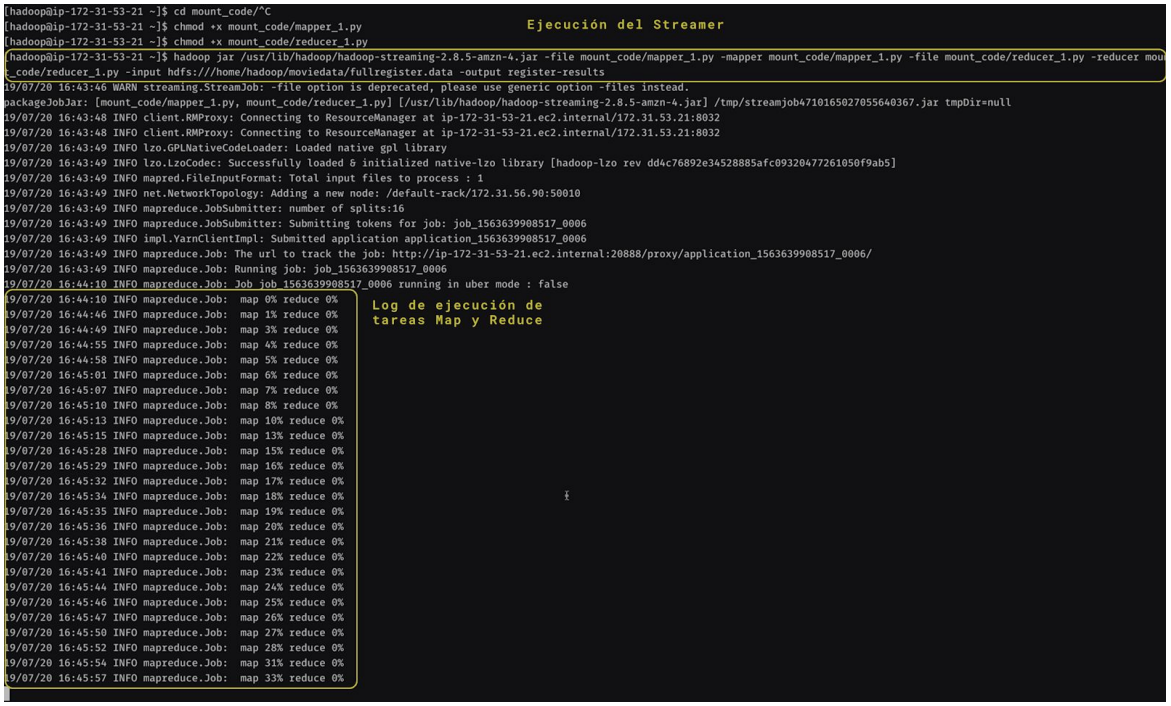

Imagen 18. Ejecución.

En este caso, nuestra tarea fue exitosamente ejecutada, lo cual se notifica con la línea JOB `job_1563639908517_006 completed succesfully`. Seguido de esta notificación, en la sección `File System Counters` el output indica información asociada a la lectura del archivo, tanto a nivel local como de nuestro sistema HDFS. Una segunda sección es `Job Counters`, la cual entrega información sobre el tiempo de ejecución y recursos asignados a la tarea, así como un contador de la cantidad de tareas exitosas y fallidas. La última sección `Map-Reduce Framework` entrega información sobre las tareas de administración en cada una de las fases.

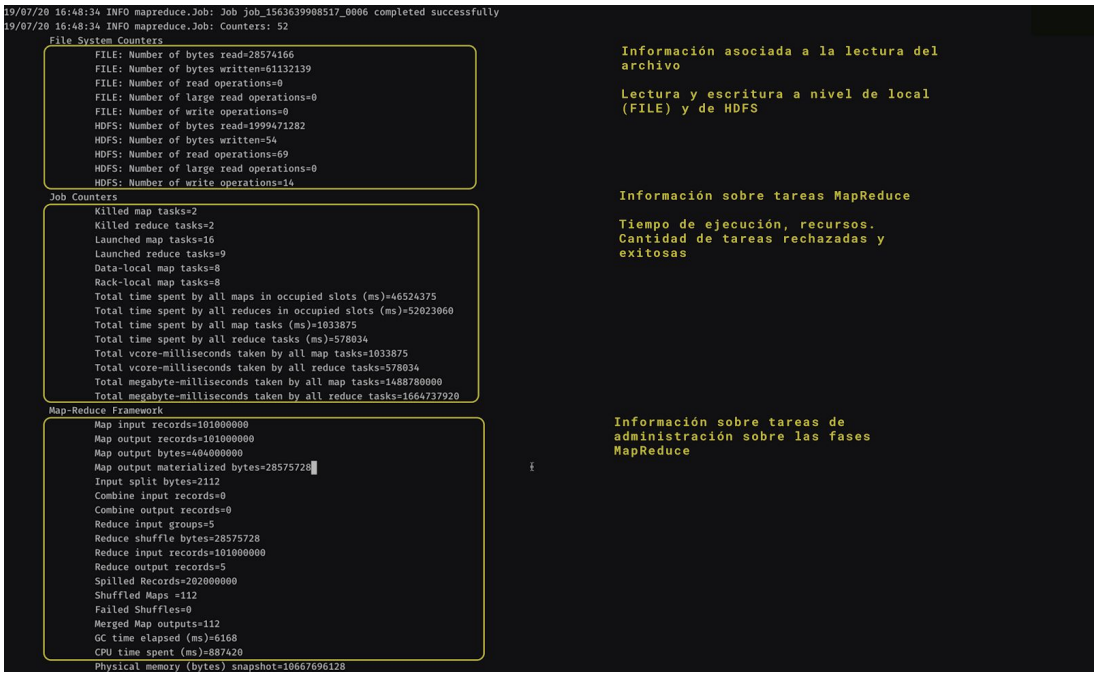

Imagen 19. MapReduce.

La última parte del output indica cuántas fallas existieron en la fase de shuffle y dónde se encuentran los resultados.

Imagen 20. Shuffle.

Paso 5: Rescatando los resultados

Hasta el momento, sólo sabemos que nuestra tarea se ejecutó exitosamente con Hadoop Streaming y los resultados fueron guardados en la carpeta register-results. Revisemos mediante hdfs dfs -ls dónde se encuentra esta carpeta. Dado que en el paso del output del streamer no definimos la ruta absoluta, se asume que es el directorio de trabajo actual, por lo que no es necesario especificar dónde está.

Si repetimos el comando `hdfs dfs -ls register-results`, veremos que el output contiene una serie de archivos con los nombres `_SUCCESS` y `part-0000` con un número indicativo a la tarea entregada por los reducers. El archivo `_SUCCESS` simplemente indica que la tarea fue completada con éxito, mientras que cada uno de los archivos con nombre `part` contienen el resultado agregado a nivel del Reducer 0, 1, 2, 3 y 4.

Por lo general estaremos interesados en concatenar todos los resultados de los archivos part en un archivo de texto plano. Esto podemos realizarlo con la instrucción `hdfs dfs -getmerge register-results ~/mount_code/results.txt`, donde debemos especificar el directorio de output, seguido de una dirección en nuestra instancia de trabajo. Este archivo `results.txt` contendrá la tarea finalizada.

```
[hadoop@ip-172-31-95-241 ~]$ hdfs dfs -ls
Found 1 items
drwxr-xr-x   - hadoop hadoop          0 2019-07-23 14:43 register-results
[hadoop@ip-172-31-95-241 ~]$ hdfs dfs -ls register-results
Found 6 items
-rw-r--r--   1 hadoop hadoop          0 2019-07-23 14:43 register-results/_SUCCESS
-rw-r--r--   1 hadoop hadoop         10 2019-07-23 14:42 register-results/part-00000
-rw-r--r--   1 hadoop hadoop         11 2019-07-23 14:42 register-results/part-00001
-rw-r--r--   1 hadoop hadoop         11 2019-07-23 14:43 register-results/part-00002
-rw-r--r--   1 hadoop hadoop         11 2019-07-23 14:43 register-results/part-00003
-rw-r--r--   1 hadoop hadoop         11 2019-07-23 14:42 register-results/part-00004
[hadoop@ip-172-31-95-241 ~]$ hdfs dfs -getmerge register-results ~/mount_code/results.txt
[hadoop@ip-172-31-95-241 ~]$ more ~/mount_code/results.txt
1       6171100
2       11483700
3       27416450
4       34515740
5       21413010
[hadoop@ip-172-31-95-241 ~]$
```

Imagen 21. results.

A grandes rasgos, el flujo de trabajo implementado podemos resumirlo con la siguiente imágen:

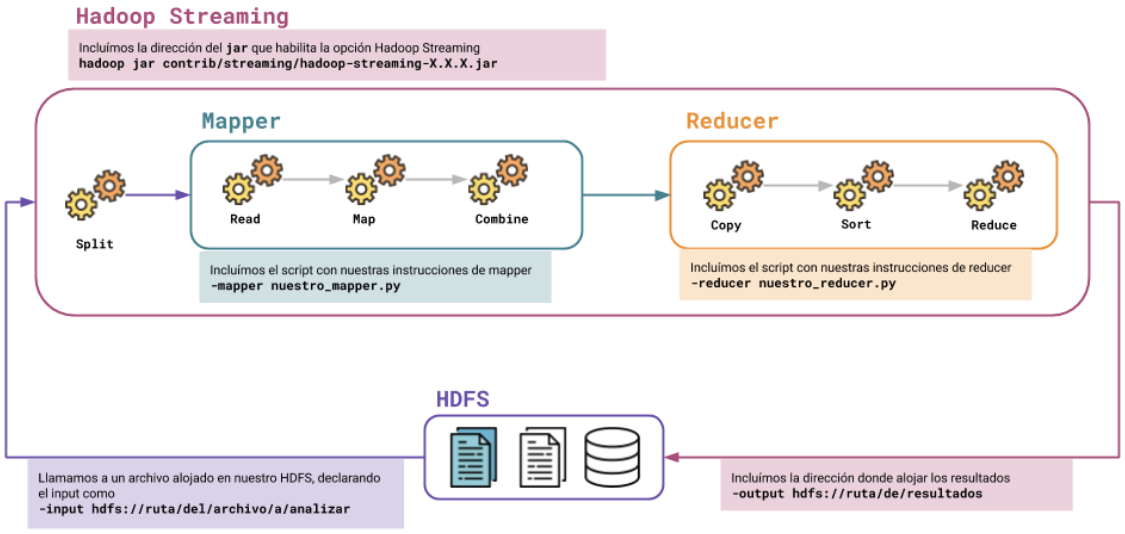

Imagen 22. Streaming.

Implementación de MapReduce con MRJob

MRJob es una librería para implementar Hadoop Streaming creada por Yelp, que permite simplificar el proceso de desarrollo de una aplicación MapReduce. Una de las ventajas es que podemos implementar nuestro código en distintos ambientes como el Local, clusters de Hadoop o correrlo en la nube como los servicios AWS EMR.

Para instalarlo en nuestra instancia de trabajo AWS EMR, escribimos `sudo pip-3.6 install mrjob`. Tomemos el mismo ejemplo de Rating Counter implementado anteriormente, pero refactorizado en código de MRJob. A continuación se presenta el código implementado.

```
!#/usr/bin/python3.6


"""
Paso 1 - Importamos la clase mrjob.job.MRJob
"""
from mrjob.job import MRJob

"""
Paso 2 - Creamos una clase llamada RatingCounter, la cual heredará toda
la funcionalidad
de la clase mrjob.job.MRJob que importamos arriba.
"""

class RatingCounter(MRJob):

"""
Paso 3 - El método mapper define la fase de mapeo.

Toma la llave y valores como argumentos y devuelve los datos de manera
temporal con yield.
A diferencia de return, yield nos permitirá enviar un valor asociado a
la función,
sin la necesidad de efectivamente ejecutarla.

Esto genera una mayor rapidez de código.
"""
   def mapper(self, key, line):
      (user_id, movie_id, rating, timestamp) = line.split('\t')
       yield (rating, 1)
```

```
"""
Paso 4 - El método reducer define la fase de reducir.

Toma la llave y un iterador de valores como argumentos y devuelve los
datos de manera temporal con yield.
El yield devolverá una tupla con forma (output_key, output_value)
"""
   def reducer(self, rating, ocurrences):
       yield (rating, sum(ocurrences))

"""
Paso 5 - Ejecución.

El último paso es generar la ejecución del comando con la línea
RatingCounter.run().
El método .run() vendrá por herencia por parte de la clase MRJob.
"""


if __name__ == '__main__':
   RatingCounter.run()
```

Guardamos nuestro procedimiento en un archivo llamado `mrjob_rating.py`, nos aseguramos de hacerlo ejecutable con `chmod +x mrjob_rating.py` y podemos implementar la línea de comando `./mrjob_rating.py fullregister.data -r hadoop > results` para implementar el mismo flujo de trabajo con los datos en local, pero ejecutando el cluster de Hadoop que se encuentra habilitado en nuestra instancia con la opción -r hadoop. El output puede guardarse en otro archivo, como en este caso un archivo llamado `results`.

Imagen 23. Rating.

La primera fase de nuestra ejecución muestra información sobre la ubicación actual del binario de Hadoop, así como el Jar de Streaming. También se va a generar una carpeta de manera temporal para representar los datos y posteriormente implementar los pasos del script. Posterior a eso, MRJob generará un reporte sobre la implementación de la tarea de Streaming.

Cierre: Algunas especificaciones sobre Hadoop Streaming

Hasta el momento nuestra implementación con Hadoop Streaming se ha mantenido al mínimo, y relegamos todos los detalles técnicos sobre la cantidad de workers en las fases de Map y Reduce al programa en sí. Si bien los trabajos se cumplen en su totalidad, no hemos hablado sobre la optimización de Hadoop Streaming. Hay varios elementos a considerar respecto a cómo mejorar el desempeño de la tarea:

- **Cantidad de Mappers**: Un aspecto a considerar es el hecho que muchos mappers en una aplicación se puede considerar ineficiente dado que estamos más recursos de los necesarios. Una heurística aproximada es considerar un tiempo de ejecución al nivel de mappers de 1 minuto. Un tiempo de ejecución menor probablemente indique un número excesivo de mappers.

- **Cantidad de Reducers**: Algo similar ocurre con la cantidad de reducers. La heurística sugerida es tener tareas de Reducers con un tiempo de ejecución aproximado de 5 minutos.

- **Combiners**: En la lectura no hablamos de combiners, que son pasos intermedios de preprocesamiento entre Map y Reduce. Una buena forma de optimizar el trabajo es ver en la medida de lo posible, qué tareas podemos separar del mapper o del reducer en un propio paso.

- **Compresión Intermedia**: Otro aspecto avanzado es el hecho de comprimir el tamaño del output del mapper para tener una mejora en el desempeño.

- **Modificación de la fase de Shuffle**: La fase Shuffle de MapReduce tiene aproximadamente doce parámetros de ajuste para mejorar el uso de memoria.

Referencias

1. White, T. (2011). Hadoop. 2nd edition: O'Reilly Media.
2. Sammer, E. (2012). Hadoop operations. Sebastopol, CA: O'Reilly.
3. Awesome ! Hadoop HDFS Commands Cheat Sheet. (2018, June 04). Retrieved from https://linoxide.com/linux-how-to/hadoop-commands-cheat-sheet/