

Transcoding de Video en MR

Meetup Santiago, Chile

Junio 26, 2018



Presentador



Tomas Sokorai
Product Specialist
Chile team

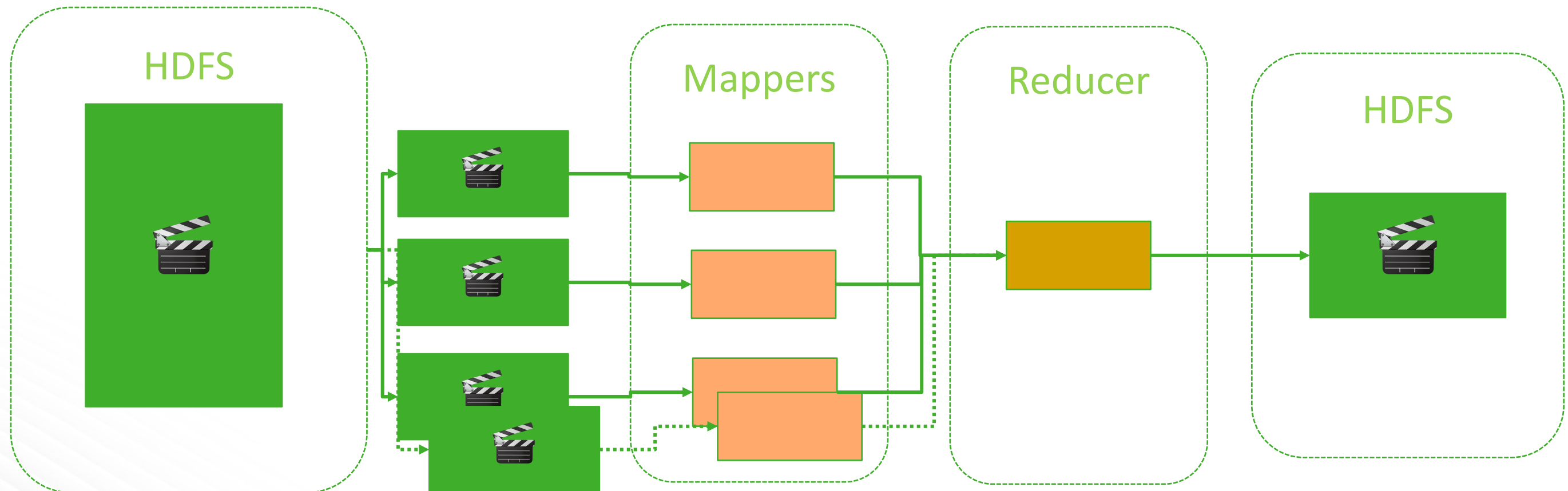
Agenda

- ◆ Trabajo pesado de manera distribuida
- ◆ Por qué (y por qué no...) MapReduce?
- ◆ Que usamos?
- ◆ Como se distribuye la data? Los splits y la localidad de datos
- ◆ Llevando nuestro código a los nodos – Localización de recursos de YARN
- ◆ Los problemas de la memoria no-heap en contenedores de YARN

Realizando tareas pesadas, distribuidas

Un ejemplo del mundo real de proceso pesado

- ◆ Como procesar, y especialmente convertir formatos de vídeo



Por que vídeo como ejemplo didáctico?

- ◆ No utiliza los tipos de datos incluidos en Hadoop, ni tampoco los formatos de archivo de entrada/salida, permitiendo demostrar la completa implementación de como manejar esa data.
- ◆ Requiere librerías externas , lo que permite mostrar detalles de como pasar estos recursos tanto a nuestro "lanzador" del job y los mappers/reducers.
- ◆ Las librerías externas en este caso son nativas, invocadas via JNI, por lo que permiten mostrar algunos problemas de memoria nativa versus el control de memoria que ejerce YARN a los contenedores.
- ◆ El proceso de vídeo requiere mucha CPU, en especial para formatos de alta eficiencia o resoluciones altas, lo que se presta muy bien para distribuirlo en un cluster.

Por qué Map Reduce?

Por qué MapReduce?

- ◆ Si bien MR es una tecnología "antigua", es muy sencillo implementar trabajos que sean muy eficientes para trabajos que requieren gran procesamiento.
- ◆ El transcoding de video normalmente se realiza para almacenamiento o para estandarizar los formatos ya almacenados, por lo que no requiere respuesta interactiva.
- ◆ Por qué no? MR no sirve para procesos interactivos de baja latencia.
- ◆ Aplicaciones reales:
 1. Sistemas de call centers/ estaciones de consulta para almacenar por largo tiempo de grabaciones de vídeo de atención a clientes.
 2. Sitios de video de usuarios web, estilo YouTube, etc.

Que usamos?

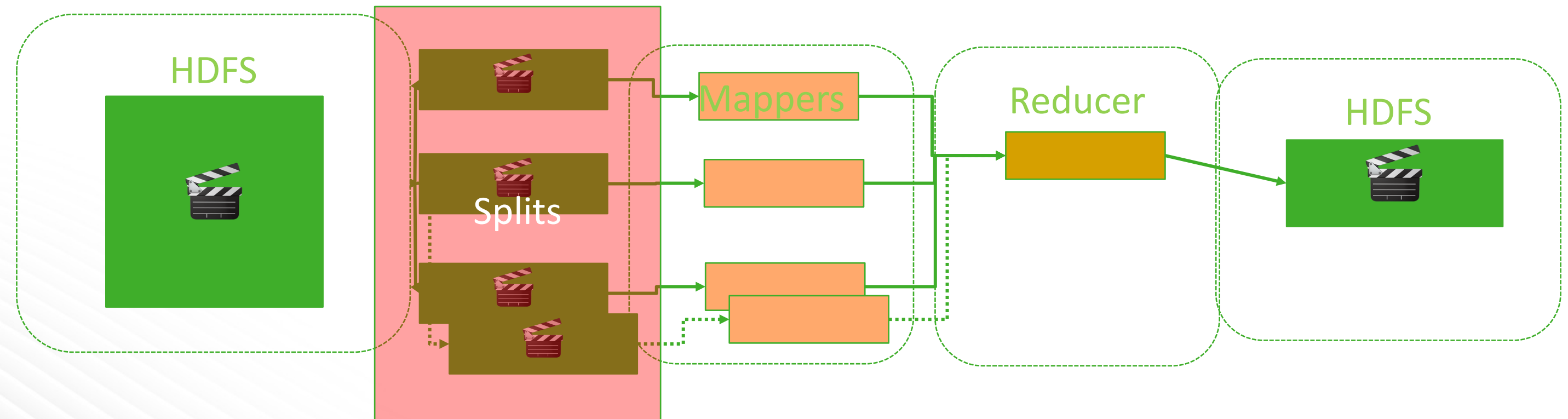
Que usamos?

- Para el transcoding obviamente la idea es reutilizar un proyecto que implemente todo lo que necesitamos con respecto a bajo nivel de video: Los codecs de audio y video (MPEG,H264,MP3,AAC,etc.), y los muxers/demuxers de contenedores (AVI, MKV, MPEG, etc.)
- Nos basaremos en una de las librerías Open Source más populares, como lo es FFMPEG.
- Obviamente no podemos invocar directamente desde Java el código nativo de las librerías de FFMPEG, por lo que utilizaremos el "envoltorio" Java de FFMPEG, Humble Video <https://github.com/artclarke/humble-video>

Como llevar la data a los procesos

Trozando la data, los SPLITS de entrada

- La clave para hacer proceso distribuido eficiente y que éste sea escalable es tratar de obtener bloques de data autocontenidos y locales (que no requieran acceso via red a otro nodo).
- En este caso para los mappers en MR, estos bloques lógicos de data se llaman "splits"



Como sabe MR el formato de mis datos?

- ◆ A menos que se trate de simples formatos como líneas de texto, MR no sabe de formatos específicos de tu data.
- ◆ Para MR los datos pueden venir de cualquier parte, no solo desde HDFS (DB, generación sintética, sensor, etc.)
- ◆ Ejemplo de trabajos no-HDFS: Sqoop MR jobs.
- ◆ Se debe definir nuestra propia clase para manejar la generación de splits y leer esta data registro por registro: heredando de InputFormat y RecordReader.
- ◆ En nuestro ejemplo la data viene desde archivos de HDFS, por lo que reutilizaremos la clase ya definida para esto en MR, FileInputFormat, la que extendemos para obtener nuestro propio RecordReader.

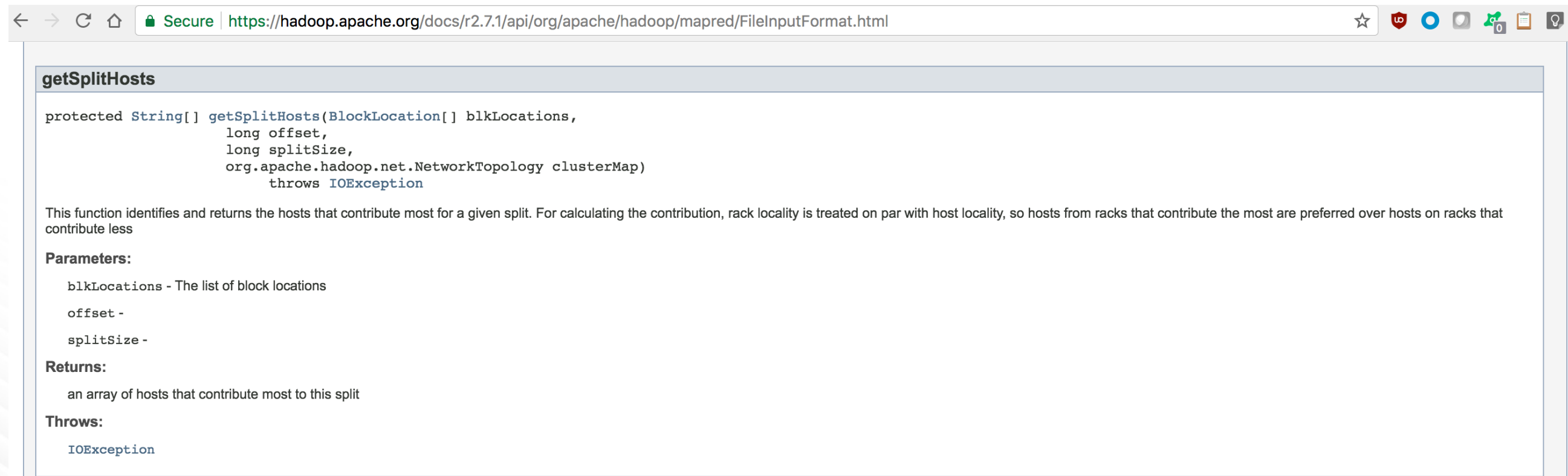
Como segmentamos nuestro vídeo?

- ◆ Dado que vamos a leer desde HDFS, vamos a reutilizar la forma estándar que tiene FileInputFormat para asignar splits: Cada split es igual a un bloque de HDFS.
- ◆ Esto es con propósito de simplificar: Una manera correcta sería utilizar el índice de keyframes del archivo y asegurarnos que los splits empiecen y terminen en un keyframe.
- ◆ Esta implementación permite que se pierda uno o más cuadros al inicio del bloque debido a que puede empezar en medio de un cuadro no en un keyframe



Donde procesamos cada split?

- Dado que leeremos desde HDFS, usaremos la forma estándar de selección de host, que se puede ver en la documentación del API de MR para FileInputFormat (la que extendemos)
- Importancia de la localidad.



The screenshot shows a web browser window with the URL <https://hadoop.apache.org/docs/r2.7.1/api/org/apache/hadoop/mapred/FileInputFormat.html>. The page displays the `getSplitHosts` method signature and its description. The method signature is: `protected String[] getSplitHosts(BlockLocation[] blkLocations, long offset, long splitSize, org.apache.hadoop.net.NetworkTopology clusterMap) throws IOException`. The description states: "This function identifies and returns the hosts that contribute most for a given split. For calculating the contribution, rack locality is treated on par with host locality, so hosts from racks that contribute the most are preferred over hosts on racks that contribute less". The parameters are listed as: `blkLocations` - The list of block locations, `offset` - , and `splitSize` - . The return value is: an array of hosts that contribute most to this split. The throws section lists: `IOException`.

```
getSplitHosts
```

```
protected String[] getSplitHosts(BlockLocation[] blkLocations,  
                                long offset,  
                                long splitSize,  
                                org.apache.hadoop.net.NetworkTopology clusterMap)  
    throws IOException
```

This function identifies and returns the hosts that contribute most for a given split. For calculating the contribution, rack locality is treated on par with host locality, so hosts from racks that contribute the most are preferred over hosts on racks that contribute less

Parameters:

- `blkLocations` - The list of block locations
- `offset` -
- `splitSize` -

Returns:

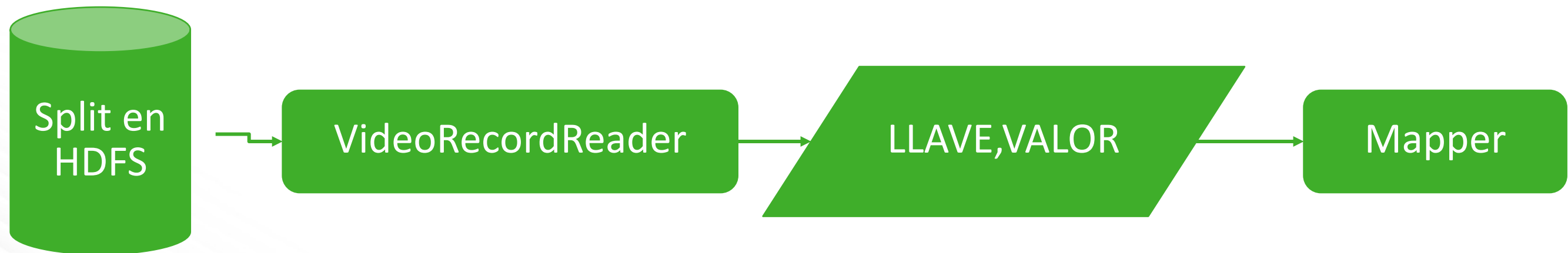
- an array of hosts that contribute most to this split

Throws:

- `IOException`

Leyendo registro por registro: Nuestro propio RecordReader

- ◆ Aquí empezamos a diverger fuertemente de la implementación existente de las clases de MR, debido a que los registros deben representar partes de archivo real de video.
- ◆ Utilizaremos la unidad de trabajo de las librerías de muxing: El MediaPacket.
- ◆ RecordReader está a cargo de generar los pares <LLAVE,VALOR> para el proceso de mapping.



Claves y valores para vídeo

- Ahora que sabemos que la entrada para los mappers son tuplas <llave,valor>, que elegiremos para reпреstar estos?
- Para la llave, que define cada frame dentro de un archivo? (Un archivo puede contener muchos streams de vídeo y audio)?
- Para simplificar, usaremos la posición del archivo de cada paquete que leemos y también a que stream pertenece, con lo que terminamos con una llave compuesta: <posArch,nrStream>
- Para el valor, usaremos la unidad mencionada anteriormente: MediaPacket.

Procesando la data

Mapper

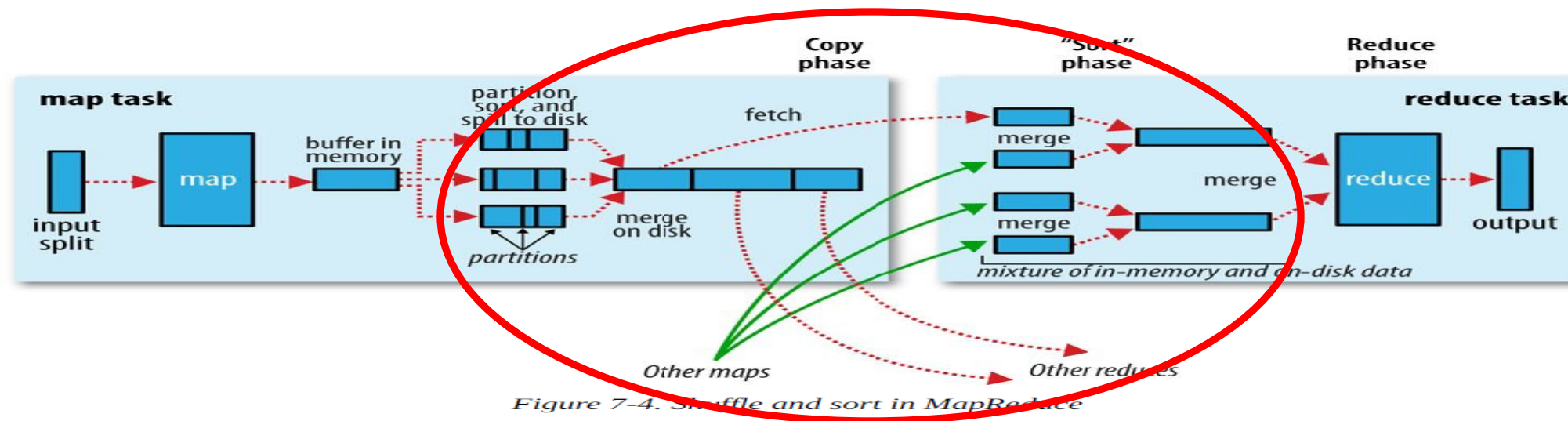
- ◆ Ya cubrimos como obtener la data desde el archivo HDFS a las tuplas <llave,valor> del mapper, así que ahora veremos detalles de como procesamos esta data en el mapper mismo.
- ◆ En nuestro ejemplo el mapper descomprime/decodifica la data original.
- ◆ Los cuadros descomprimidos son escalados a 640x480, tamaño arbitrario que es suficientemente pequeño para hacer más rápida la demostración y demuestra como se puede procesar las imágenes.
- ◆ Finalmente el vídeo reescalado es recodificado con el codec especificado en la línea de comandos
- ◆ Para propósitos de simplificación los streams de audio son ignorados por el momento.
- ◆ The input to the Reducer stage is also a <key,value> pair, but since after re-encode the file position is irrelevant, the key is now based on the DTS (Decode Time Stamp) of the frame and the stream.

Mapper, que sale?

- ◆ Debemos generar como salida desde el mapper la unidad de trabajo de nuestro(s) reducers.
- ◆ La entrada de la etapa reducir es también una tupla <llave,valor>, pero como hemos alterado totalmente el tamaño de los paquetes que escribiremos, la posición de archivo es irrelevante en nuestra salida, por lo que elegimos usar como llave algo que tenga más sentido, como el DTS (Decode Time Stamp) del cuadro en el stream.

Entre el Mapper y Reducer

- ◆ Dado que la salida del mapper es producida de manera paralela (la gracia de el proceso distribuido), como ensamblamos nuestro archivo final con la data en el orden correcto?
- ◆ Esta es la etapa de shuffle y sort que se encarga de ello.
- ◆ La única personalización que debemos hacer es proveer una clase que compare nuestras llaves y nos permita general el orden correcto: Nuestra propia clase SortComparator: CompositeKeyComparator



Reducer

- ◆ Para mantener la implementación simple, solo crearemos un reducer para generar el archivo de vídeo final.
- ◆ El número de reducers puede controlarse via `mapreduce.job.reduces`, en este caso 1.
- ◆ En reducer es super simple y solo pasa las tuplas <llave,valor> a nuestra clase `OutputFormat`, la que realiza el trabajo inverso que realizamos en `InputFormat`: Realiza el muxing de nuestros paquetes ya codificados dentro de un contenedor de streams Matroska (MKV).

Juntándolo todo

- ◆ Como juntamos todas estas etapas y clases en un trabajo que podamos lanzar a nuestro cluster?
- ◆ Existe una clase llamada `org.apache.hadoop.mapreduce.Job` que nos provee el control para nuestro trabajo y con la que registraremos nuestras clases y los parámetros básicos de nuestro trabajo.
- ◆ Usaremos "hadoop" para lanzar nuestro JAR que contiene las clases y provee un punto de entrada `Main()` típico de aplicación Java de línea de comandos.

Demo práctico

Obteniendo las librerías requeridas

- ◆ Como comentamos en las slides anteriores, requerimos un set no-hadoop (y no- java) que nos provean de un set de codecs y muxers rápidos.
- ◆ Pero no solo nuestros mappers y reducers requieren de ellas, nuestro lanzador de trabajo tambien las require para obtener la información básica de streams y codecs y pasarla a los mappers y reducers.
- ◆ Que pasa si corremos la "tal como está"?

Veámoslo!

Como obtenemos los JARs que nuestro lanzador de job requiere?

- ◆ Esto se logra fácilmente con la variable de entorno HADOOP_CLASSPATH.
- ◆ Que pasa si corremos nuestro lanzador de trabajo con esta clase apuntando a las librerías, pero nada más?

Veámoslo!

Distribuyendo nuestros JARs específicos al cluster.

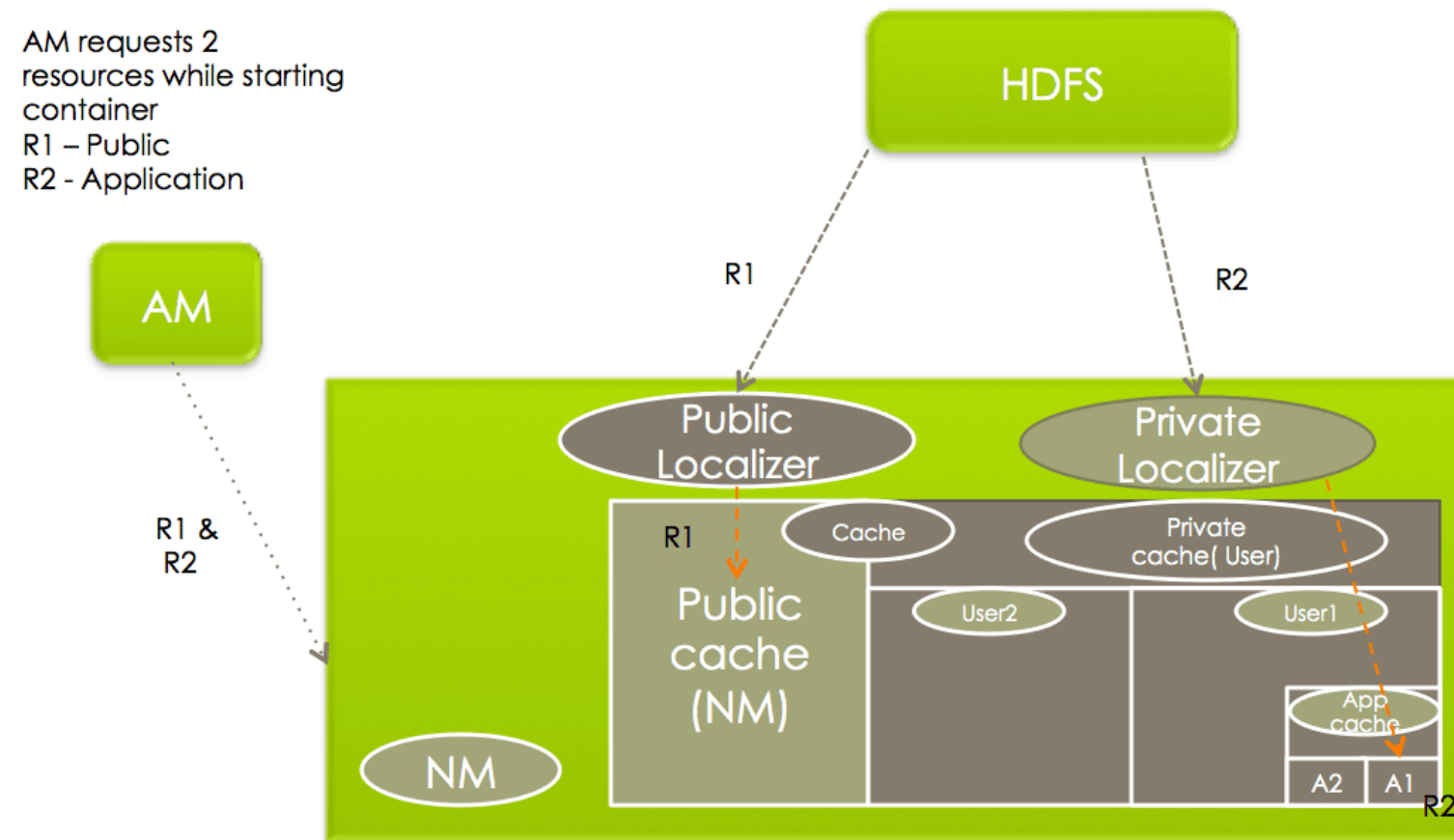
- ◆ En primer lugar debemos hacer llegar a HDFS nuestros JARs. Existen varias opciones para ello:
- ◆ DistributedCache: Podemos agregar programáticamente a el `org.apache.hadoop.mapreduce.Job` los archivos que deseamos distribuir a los nodos.
- ◆ Crear un JAR "gordo" : Agregamos todos los contenidos de los otros JARs al JAR que usamos para distribuir nuestras clases.
- ◆ Usar la opción `-libjars` del comando "hadoop"
- ◆ Dado que para DistributedCache necesitamos implementar más código y nos implica menos flexibilidad, y dado que el JAR "gordo" no es para nada de elegante, usaremos la opción `"-libjars"`.

Por qué no funciona mi opción -libjars?.

- ◆ Hay un detalle con esta manera aparentemente trivial: La opción –libjars NO es parseada automáticamente a menos que el código que lanza nuestro trabajo lo haga de manera explícita.
- ◆ La manera de que nuestro lanzador del trabajo puede tomar ventaja de esto es usar directamente GenericOptionsParser, o podemos implementar la interfaz `org.apache.hadoop.util.Tool` en nuestra clase de lanzamiento.
- ◆ Para nuestro ejemplo, la implementación de Tool es la elegida.

Como llegan nuestros JARs a los nodos?

- Este proceso es llamado "localización" y depende de que el servicio de YARN que corre en nuestros nodos de cómputo, el NodeManager, que se encarga de bajar estos archivos de HDFS y almacenarlos en un cache en el sistema de archivos local.

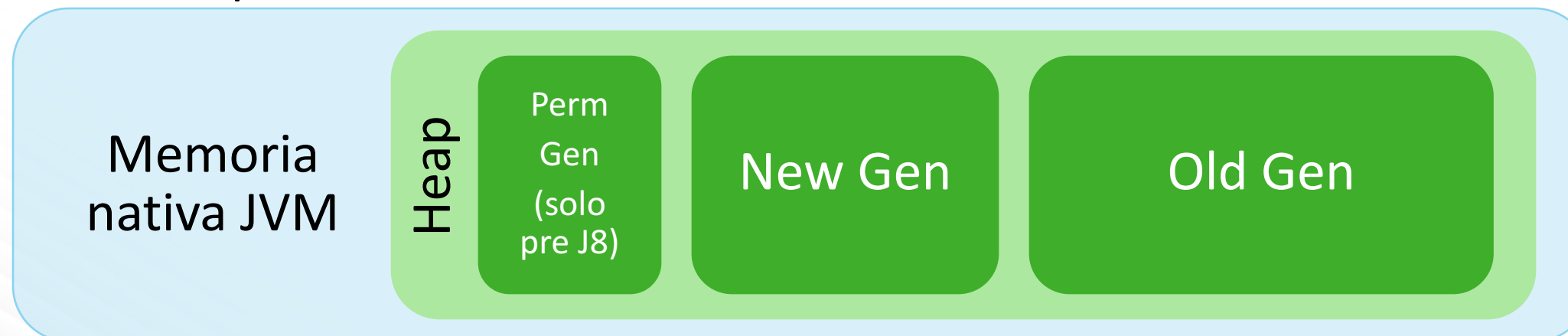


Diferentes lugares para diferentes tiempos de vida

- ◆ *Público*: `<local-dir>/filecache`
- ◆ *Privado*: `<local-dir>/usercache/<user>/filecache`
- ◆ *Aplicación*: `<local-dir>/usercache/<user>/appcache/<app-id>/`
- ◆ Los directorios *Público* y *Privado* son mantenidos entre invocaciones de nuestro trabajo, mantenido como caché LRU (Least Recently Used).
Público es usado para almacenar por ejemplo el archivo `mapreduce.tar.gz`, que contiene la infraestructura básica de MR.
- ◆ *Aplicación*: Los archivos son borrados automáticamente cuando nuestro trabajo termine.
- ◆ Cuidado que si se sube (o baja) un archivo corrupto, se puede mantener corrupto en el cache debido a la expiración LRU.
En estos casos borrar los directorios `filecache` y `usercache`, y asegurarse que los recursos de origen estén en buen estado.

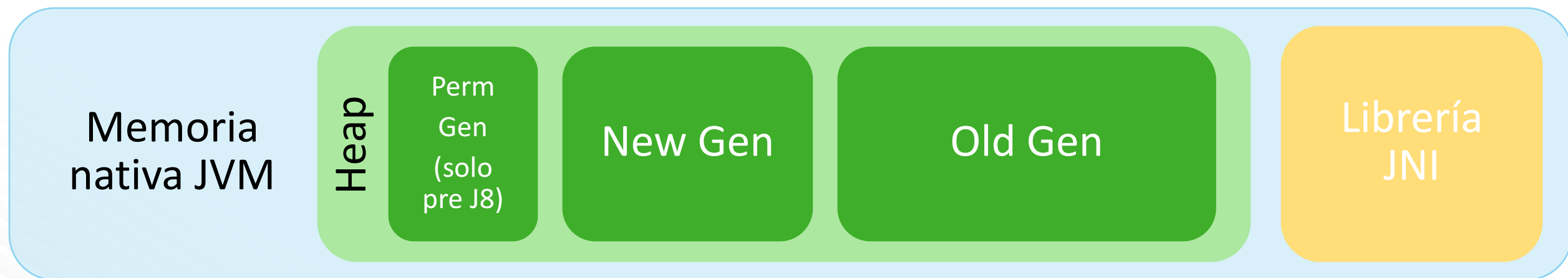
Límites? No necesitamos límites!

- ◆ Mencionamos que las librerías de codecs/muxing eran nativas, que significa esto?
- ◆ Significan que no están escritas en Java o diseñadas para correr directamente en la JVM (no son bytecode Java), son librerías nativas del OS. Por ejemplo .so para Unix/Linux o .dll para el mundo MS.
- ◆ Para poder ser invocadas desde Java, son "envueltas" en llamadas via JNI (Java Native Interface).
- ◆ Esto significa que el uso de memoria interno de estas librerías nativas (variables, buffers, código, etc.) NO se agregan a el área principal de la JVM (Heap) si no que al address space de el proceso mismo de Java



Como las librerías JNI cambian nuestra huella de memoria

- Normalmente la área nativa de la JVM es pequeña comparada con el memoria usada para Heap, ya que solo mantiene pequeños caches y estados internos, no data de las aplicaciones mismas.
- Pero esto cambia cuando introducimos las librerías nativas, y se vuelve un uso no-trivial.
- Debido a esto con los mismos parámetros de Xmx (máxima memoria de heap) el proceso de la JVM será mucho más grande con solo utilizar las librerías nativas!



Por qué nos importa?

- Porque YARN por defecto tiene habilitado una característica que monitorea el uso de memoria de cada contenedor, y mata cualquier contenedor que cruce ese límite: El típico “Container [algo] is running beyond physical memory limits. Current usage: X GB of Y GB physical memory used” mensaje que casi todos los usuarios de YARN han experimentado en algún momento.
- Una posibilidad es incrementar el tamaño mínimo de contenedor o el tamaño de contenedor de mappers/reducers, de manera que la memoria nativa sea pequeña comparada con el tamaño total de el contenedor.
- Pero como queremos sacar máximo uso a nuestro cluster, muchas veces el tamaño de memoria de contenedor va a limitar cuantos contenedores (y por ende vcores de CPU) vamos a poder correr en el mismo momento. `yarn.nodemanager.pmem-check-enabled=false` nos ayuda en este caso.
- Esto también puede suceder cuando el tamaño de contenedor es demasiado pequeño.

Veámoslo corriendo

- Toma un tiempo, dado que nuestra implementación no está para nada optimizada y los recursos computacionales que usamos son bastante limitados.

El código al que nos referimos en esta presentación

- ◆ <https://github.com/tsokorai/TranscodeMapReduce>

Gracias!