

Tema 2 – Introducción a Hadoop y HDFS – TCDM

T. Fernández, F. García, D. Sevilla

Máster en Tecnologías de Análisis de Datos Masivos: Big Data
Universidad de Murcia

2023

- 1 Introducción a Hadoop
- 2 Instalación
- 3 Introducción a HDFS
- 4 YARN y MapReduce
- 5 Ejemplo de programa MapReduce
- 6 Filesystems en Hadoop
- 7 Interfaz en línea de comandos
- 8 Interfaz Java
- 9 Interfaz Python
- 10 Herramientas para la gestión del HDFS
- 11 Otras interfaces a HDFS
- 12 Otros aspectos

1. Introducción a Hadoop



Implementación open - source de MapReduce

- Procesamiento de enormes cantidades de datos en grandes clusters de hardware barato (*commodity clusters*)
 - Escala: petabytes de datos en miles de nodos

Tres partes

- Almacenamiento distribuido: HDFS
- Planificación de tareas y negociación de recursos: YARN
- Procesamiento distribuido: MapReduce

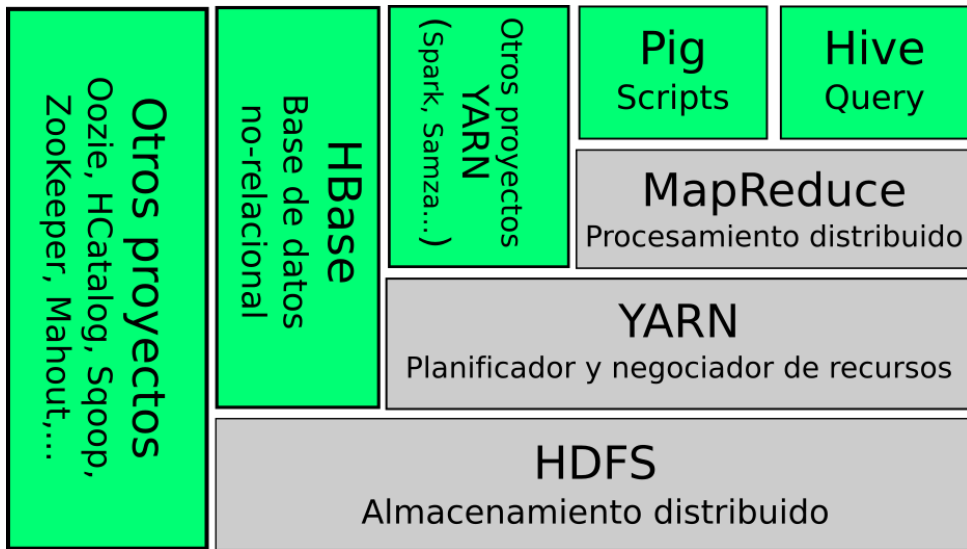
Características de Hadoop

Tres partes

- Almacenamiento distribuido: HDFS
- Planificación de tareas y negociación de recursos: YARN
- Procesamiento distribuido: MapReduce

Ventajas

- Bajo coste: clusters baratos o cloud
- Facilidad de uso
- Tolerancia a fallos



2. Instalación

Instalación relativamente simple: aplicación Java

- Paquete fuente: hadoop.apache.org/releases.html
- Sistemas preconfigurados proporcionados por empresas como Cloudera/Hortonworks (www.cloudera.com/products/hdp.html)

Instalación relativamente simple: aplicación Java

- Paquete fuente: hadoop.apache.org/releases.html
- Sistemas preconfigurados proporcionados por empresas como Cloudera/Hortonworks (www.cloudera.com/products/hdp.html)

Modos de funcionamiento:

- Standalone: todo en un nodo, para pruebas
- Pseudodistribuido: funciona como una instalación completa, pero en un solo nodo
- Totalmente distribuido

Principales ficheros de configuración:

- `core-site.xml`: parámetros de configuración general
- `hdfs-site.xml`: configuración del HDFS
- `yarn-site.xml`: configuración de YARN
- `mapred-site.xml`: configuración del MapReduce

Fichero `core-site.xml`:

- `fs.defaultFS`: nombre del sistema de ficheros a usar (HDFS u otro), por defecto `file:///`
- `hadoop.tmp.dir`: directorio base para otros directorios temporales, valor por defecto `/tmp/hadoop-${user.name}`
- `hadoop.security.authentication`: indica el tipo de autenticación, puede ser `simple` (sin autenticación) o `kerberos`, por defecto `simple`
- `hadoop.security.authorization`: indica si está activada la autorización a nivel de servicio, por defecto `false`

3. Introducción a HDFS

HDFS: *Hadoop Distributed File System*

Hadoop puede acceder a diferentes tipos de filesystems (local, HDFS, KFS, S3, ...)

- Se recomienda HDFS: *Hadoop Distributed File System*

HDFS: *Hadoop Distributed File System*

Hadoop puede acceder a diferentes tipos de filesystems (local, HDFS, KFS, S3, ...)

- Se recomienda HDFS: *Hadoop Distributed File System*

HDFS: Ventajas

- Diseñado para almacenar ficheros muy grandes en *commodity hardware*
- Elevado ancho de banda
- Fiabilidad mediante replicación

HDFS: *Hadoop Distributed File System*

Hadoop puede acceder a diferentes tipos de filesystems (local, HDFS, KFS, S3, ...)

- Se recomienda HDFS: *Hadoop Distributed File System*

HDFS: Ventajas

- Diseñado para almacenar ficheros muy grandes en *commodity hardware*
- Elevado ancho de banda
- Fiabilidad mediante replicación

HDFS: Inconvenientes

- Elevada latencia
- Poco eficiente con muchos ficheros pequeños
- Modificaciones siempre al final de los ficheros
- No permite múltiples escritores (modelo *single-writer, multiple-readers*)

Namenode

Mantiene la información (metadatos) de los ficheros y bloques que residen en el HDFS

Datanodes

Mantienen los bloques de datos

- No tienen idea sobre los ficheros

Bloques

Por defecto 128 MB, tamaño configurable por fichero

- bloques pequeños aumentan el paralelismo (un bloque por Map)
- bloques más grandes reducen la carga del NameNode

Replicados a través del cluster

- Por defecto, 3 réplicas (configurable por fichero)

Backup/Checkpoint node

Mantiene backups y checkpoints del NameNode

- debería ejecutarse en un sistema con características similares al NameNode

Múltiples propiedades configurables (fichero `hdfs-site.xml`)

- `dfs.namenode.name.dir`: lista (separada por comas) de directorios donde el NameNode guarda sus metadatos (una copia en cada directorio), por defecto `file://${hadoop.tmp.dir}/dfs/name`
- `dfs.datanode.data.dir`: lista (separada por comas) de directorios donde los datanodes guardan los bloques de datos (cada bloque en sólo uno de los directorios), por defecto `file://${hadoop.tmp.dir}/dfs/data`
- `dfs.namenode.backup.address`: dirección y puerto de Backup node (por defecto, `0.0.0.0:50100`)

HDFS: propiedades configurables (II)

- `dfs.blocksize`: tamaño de bloque para nuevos ficheros, por defecto 128MB
- `dfs.replication`: nº de réplicas por bloque, por defecto 3
- `dfs.replication.max`: máximo nº de réplicas permitido por bloque, por defecto 512
- `dfs.namenode.replication.min`: mínimo nº de réplicas permitido por bloque, por defecto 1

Interfaz con HDFS

Varias interfaces:

- 1 Interfaz en línea de comandos: comando `hdfs dfs`
- 2 Interfaz web
- 3 Interfaz Java
- 4 Interfaz Python

Interfaz con HDFS

Varias interfaces:

- 1 Interfaz en línea de comandos: comando `hdfs dfs`
- 2 Interfaz web
- 3 Interfaz Java
- 4 Interfaz Python

Interfaz en línea de comandos:

- Permite cargar, descargar y acceder a los ficheros HDFS desde línea de comandos
- Ayuda: `hdfs dfs -help`

Más información: hadoop.apache.org/docs/stable3/hadoop-project-dist/hadoop-hdfs/HDFSCommands.html,
hadoop.apache.org/docs/stable3/hadoop-project-dist/hadoop-common/FileSystemShell.html

4. YARN y MapReduce

YARN: *Yet Another Resource Negotiator*

Se encarga de la gestión de recursos y job - scheduling/monitorización usando tres demonios:

- *Resource manager* (RM): planificador general
- *Node managers* (NM): monitorización, uno por nodo
- *Application masters* (AM): gestión de aplicaciones, uno por aplicación

YARN: *Yet Another Resource Negotiator*

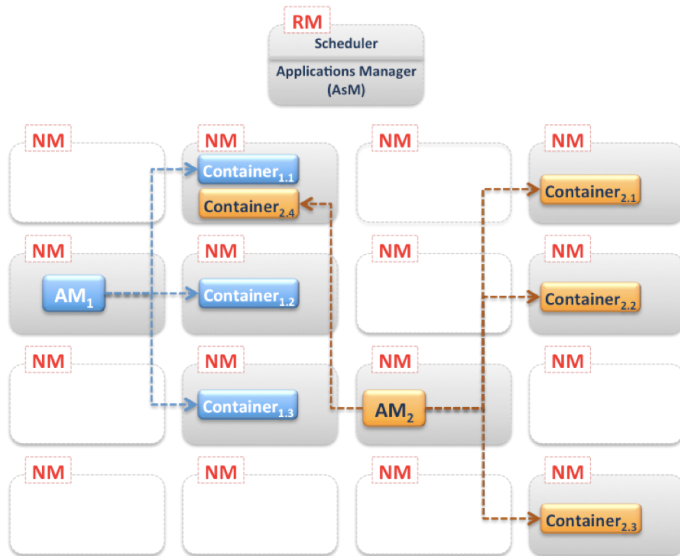
Se encarga de la gestión de recursos y job - scheduling/monitorización usando tres demonios:

- *Resource manager* (RM): planificador general
- *Node managers* (NM): monitorización, uno por nodo
- *Application masters* (AM): gestión de aplicaciones, uno por aplicación

Permite que diferentes tipos de aplicaciones (no sólo MapReduce) se ejecuten en el *cluster*

- Las aplicaciones se despliegan en contenedores (YARN JVMs)
- En Hadoop v3 se pueden usar contenedores Docker

Arquitectura YARN



Resource manager

- Arbitra los recursos entre las aplicaciones en el sistema
- Demonio global, obtiene datos del estado del cluster de los node managers
- Dos componentes:
 - Scheduler: planifica aplicaciones en base a sus requerimientos de recursos
 - Applications Manager: acepta trabajos, negocia contenedores y gestiona fallos de los Application Masters

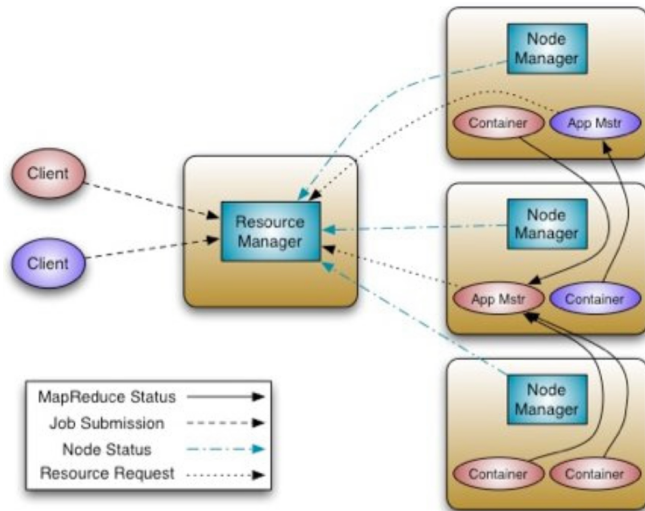
Node managers

- Uno por nodo
- Monitorizan los recursos del cluster

Application masters

- Uno por aplicación, se encarga de gestionar el ciclo de vida de la aplicación
- Solicita recursos (*contenedores*) al Resource manager y ejecuta la aplicación en esos contenedores
 - en una aplicación MapReduce en un contenedor se ejecutan tareas Map o Reduce
 - el AM se ejecuta en su propio contenedor
- Trabaja con los Node managers para ejecutar y monitorizar las tareas

Elementos de control YARN



Fuente: A. Murthy, V. Vavilapalli, "Apache Hadoop YARN", Addison-Wesley, marzo 2014.

YARN: propiedades configurables (I)

Múltiples propiedades configurables (fichero `yarn-site.xml`)

- `yarn.resourcemanager.hostname`: el *host* ejecutando el ResourceManager
- `yarn.scheduler.maximum-allocation-vcores`,
`yarn.scheduler.minimum-allocation-vcores`: nº máximo y mínimo de cores virtuales (threads) que pueden ser concedidos a un contenedor
- `yarn.scheduler.maximum-allocation-mb`,
`yarn.scheduler.minimum-allocation-mb`: memoria máxima y mínima que puede ser concedida a un contenedor (la memoria solicitada se redondea a un múltiplo del mínimo)

YARN: propiedades configurables (II)

- `yarn.nodemanager.aux-services`: lista de servicios auxiliares que deben implementar los NodeManagers (uno de ellos, el barajado MapReduce)
- `yarn.nodemanager.resource.memory-mb`: cantidad de memoria que puede reservarse para contenedores YARN en un nodo (si -1 se determina automáticamente, si la detección está habilitada)

Permite lanzar y gestionar trabajos en YARN:

- `yarn jar`: ejecuta un fichero jar
- `yarn application`: información sobre las aplicaciones ejecutándose en YARN
- `yarn container`: información sobre los contenedores
- `yarn node`: información sobre los nodos
- `yarn top`: información sobre el uso del cluster
- `yarn rmadmin`: comandos para la administración del cluster

Más información: hadoop.apache.org/docs/stable3/hadoop-yarn/hadoop-yarn-site/YarnCommands.html

Hadoop incorpora una implementación de MapReduce

- Programable en Java
- Uso de otros lenguajes mediante sockets (C++) o Streaming (Python, Ruby, etc.)

Múltiples propiedades configurables (fichero `mapred-site.xml`)

- `yarn.app.mapreduce.am.resource.cpu-vcores`: cores virtuales usados por el AM
- `yarn.app.mapreduce.am.resource.mb`: cantidad de memoria requerida para el AM
- `yarn.app.mapreduce.am.command-opts`: opciones Java para el AM
- `mapreduce.{map,reduce}.cpu.vcores`: cores solicitados al scheduler para cada tarea map/reduce
- `mapreduce.{map,reduce}.memory.mb`: memoria solicitada al scheduler para cada tarea map/reduce
- `mapreduce.{map,reduce}.java.opts`: opciones Java para los contenedores

Permite gestionar trabajos MapReduce:

- `mapred job`: interactúa con trabajos MapReduce
- `mapred archive`: crea archivos .har (más información en la *Hadoop Archives Guide*)
- `mapred streaming`: Permite el uso de otros lenguajes de programación para programar tareas Map y Reduce (se verá después) *Hadoop Streaming*
- `mapred distcp`: copia recursiva entre clusters Hadoop (más información en la *Hadoop DistCp Guide*)

Más información: hadoop.apache.org/docs/stable3/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapredCommands.html

Parámetros de configuración de YARN y MapReduce

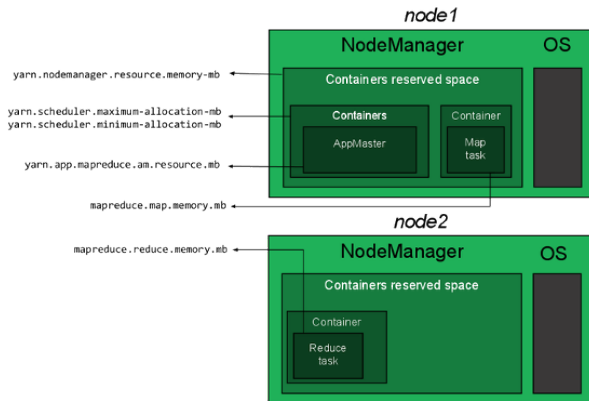
Necesitamos balancear el uso de RAM, cores y discos

- Ajustar los parámetros de Hadoop al hardware disponible

Los parámetros más sensibles son los referidos a la memoria

- `yarn.scheduler.maximum-allocation-mb`,
`yarn.scheduler.minimum-allocation-mb`,
`yarn.nodemanager.resource.memory-mb`
- `yarn.app.mapreduce.am.resource.mb`,
`yarn.app.mapreduce.am.command-opts`, `mapreduce.map.memory.mb`,
`mapreduce.reduce.memory.mb`, `mapreduce.map.java.opts`,
`mapreduce.reduce.java.opts`

Memoria en YARN y MapReduce



Hadoop puede seleccionar el valor `yarn.nodemanager.resource.memory-mb` de forma automática.

Fuente: <https://docs.deistercloud.com/Technology.50/Hadoop/Hadoop cluster.20.xml>

No existe una fórmula mágica para determinar los mejores valores

- Diferentes ajustes para diferentes cargas de trabajo
- Aproximaciones heurísticas como la presentada por Hortonworks
- Parte de:
 - Memoria disponible por nodo
 - Número de cores por nodo
 - Número de discos por nodo

Memoria disponible por nodo

Memoria total menos la reservada para el sistema

- La reservada será un porcentaje de la total
- Una aproximación es la de la tabla

Memoria total por nodo	Memoria para el sistema
< 8GB	1 GB
8GB - 16 GB	2 GB
24 GB	4 GB
48 GB	6 GB
64 GB - 72 GB	8 GB
96 GB	12 GB
128 GB	24 GB
> 128 GB	MemTotal/8

Número de contenedores por nodo

Función de la memoria disponible, nº de cores y nº de discos:

$$\text{Ncontenedores} = \min(2 \times \text{Ncores}, \\ 1.8 \times \text{Ndiscos}, \\ \text{RAMdisponible} / \text{TamañoMínimoContenedor})$$

Memoria por contenedor

La memoria mínima por contenedor va a depender:

- De la memoria total del nodo y la memoria disponible
- Del número de contenedores por nodo

Memoria total por nodo	Tamaño Mínimo por Contenedor
< 4GB	256 MB
4 GB - 8 GB	512 MB
8 GB - 24 GB	1024 MB
> 24 GB	2048 MB

$$\text{RAMporcontenedor} = \max(\text{TamañoMínimoContenedor}, \text{RAMdisponible}/\text{Ncontenedores})$$

Valores de los parámetros

Parámetro	Valor
yarn.nodemanager.resource.memory - mb	$N_{\text{contenedores}} \times \text{RAM por contenedor}$
yarn.scheduler.minimum - allocation - mb	RAM por contenedor
yarn.scheduler.maximum - allocation - mb	$N_{\text{contenedores}} \times \text{RAM por contenedor}$
mapreduce.map.memory.mb	RAM por contenedor
mapreduce.reduce.memory.mb	$2 \times \text{RAM por contenedor}$
mapreduce.map.java.opts	$0.8 \times \text{RAM por contenedor}$
mapreduce.reduce.java.opts	$0.8 \times 2 \times \text{RAM por contenedor}$
yarn.app.mapreduce.am.resource.mb	$2 \times \text{RAM por contenedor}$
yarn.app.mapreduce.am.command - opts	$0.8 \times 2 \times \text{RAM por contenedor}$

Cada nodo del cluster tiene: 12 cores, 48 GB RAM y 12 discos

- $\text{RAMdisponible} = 48 \text{ GB} - 6 \text{ GB} = 42 \text{ GB}$
- $\text{TamañoMínimoContenedor} = 2048 \text{ MB} = 2 \text{ GB}$
- $\text{Ncontenedores} = \min(2 \times 12, 1.8 \times 12, 42/2) = 21$
- $\text{RAMporcontenedor} = \max(2, 42/21) = 2 \text{ GB}$

Ejemplo: valores de los parámetros

Parámetro	Valor
yarn.nodemanager.resource.memory-mb	43008
yarn.scheduler.minimum-allocation-mb	2048
yarn.scheduler.maximum-allocation-mb	43008
mapreduce.map.memory.mb	2048
mapreduce.reduce.memory.mb	4096
mapreduce.map.java.opts	-Xmx1638m
mapreduce.reduce.java.opts	-Xmx3276m
yarn.app.mapreduce.am.resource.mb	4096
yarn.app.mapreduce.am.command-opts	-Xmx3276m

Propiedades en el `yarn-site.xml`:

- `yarn.nodemanager.{vmem,pmem}-check-enabled`: si *true*, se chequea el uso de la memoria virtual/física
- `yarn.nodemanager.vmem-pmem-ratio`: ratio memoria virtual/física que pueden usar los contenedores

El NodeManager puede chequear el uso de la memoria virtual/física del contenedor, matándolo si:

- Su memoria física excede `"mapreduce.{map,reduce}.memory.mb"`
- Su memoria virtual excede `"yarn.nodemanager.vmem-pmem-ratio"` veces el valor `"mapreduce.{map,reduce}.memory.mb"`

5. Ejemplo de programa MapReduce

Ejemplo MapReduce: WordCount

El programa WordCount es el ejemplo canónico de MapReduce

- Veremos una implementación muy simple

Ejemplo MapReduce: WordCount

El programa WordCount es el ejemplo canónico de MapReduce

- Veremos una implementación muy simple

Definimos tres clases Java

- Una clase para la operación Map (**WordCountMapper**)
- Una clase para la operación Reduce (**WordCountReducer**)
- Una clase de control, para inicializar y lanzar el trabajo MapReduce (**WordCountDriver**)

Mapper

```
public class WordCountMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {
    @Override
    public void map(LongWritable key, Text value, Context ctxt)
        throws IOException, InterruptedException {
        Matcher matcher = pat.matcher(value.toString());
        while (matcher.find()) {
            word.set(matcher.group().toLowerCase());
            ctxt.write(word, one);
        }
    }
    private Text word = new Text();
    private final static IntWritable one = new IntWritable(1);
    private Pattern pat =
        Pattern.compile("\\b[a-zA-Z\\u00C0-\\uFFFF]+\\b");
}
```

Reducer

```
public class WordCountReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {
    @Override
    public void reduce(Text key, Iterable<IntWritable> values,
        Context ctxt) throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable value : values) {
            sum += value.get();
        }
        ctxt.write(key, new IntWritable(sum));
    }
}
```

Driver (I)

```
public class WordCountDriver
    extends Configured implements Tool {
    public int run(String[] arg0) throws Exception {
        if (arg0.length != 2) {
            System.err.printf("Usar: %s [ops] <entrada> <salida>\n",
                getClass().getSimpleName());
            ToolRunner.printGenericCommandUsage(System.err);
            return -1;
        }
        Configuration conf = getConf();
        Job job = Job.getInstance(conf);
        job.setJobName("Word Count");
        job.setJarByClass(getClass());
        FileInputFormat.addInputPath(job, new Path(arg0[0]));
        FileOutputFormat.setOutputPath(job, new Path(arg0[1]));
    }
}
```

Driver (II)

```
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);

job.setNumReduceTasks(1);

job.setMapperClass(WordCountMapper.class);
job.setCombinerClass(WordCountReducer.class);
job.setReducerClass(WordCountReducer.class);

return (job.waitForCompletion(true) ? 0 : -1);
}
public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new WordCountDriver(), args);
    System.exit(exitCode);
}
}
```

Aspectos a tener en cuenta:

- 1 La nueva API (desde 0.20.0) se encuentra en `org.apache.hadoop.mapreduce` (la antigua en `org.apache.hadoop.mapred`)
- 2 Preferiblemente, crear un jar y ejecutarlo con:

```
yarn jar fichero.jar [opciones]
```

- Para gestionar las aplicaciones, utilizad:
 - en general, la opción `application` del comando `yarn` (`yarn application -help` para ver las opciones)
 - para trabajos MapReduce, la opción `job` del comando `mapred` (`mapred job -help` para ver las opciones)
- Más información en
 - hadoop.apache.org/docs/stable3/hadoop-yarn/hadoop-yarn-site/YarnCommands.html
 - hadoop.apache.org/docs/stable3/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapredCommands.html

Hadoop Streaming

- API que permite crear códigos map-reduce en otros lenguajes
- Utiliza streams Unix como interfaz entre Hadoop y el código
- Permite usar cualquier lenguaje que pueda leer de la entrada estándar y escribir en la salida estándar (Python, Ruby, etc.)

Hadoop Streaming

- API que permite crear códigos map-reduce en otros lenguajes
- Utiliza streams Unix como interfaz entre Hadoop y el código
- Permite usar cualquier lenguaje que pueda leer de la entrada estándar y escribir en la salida estándar (Python, Ruby, etc.)

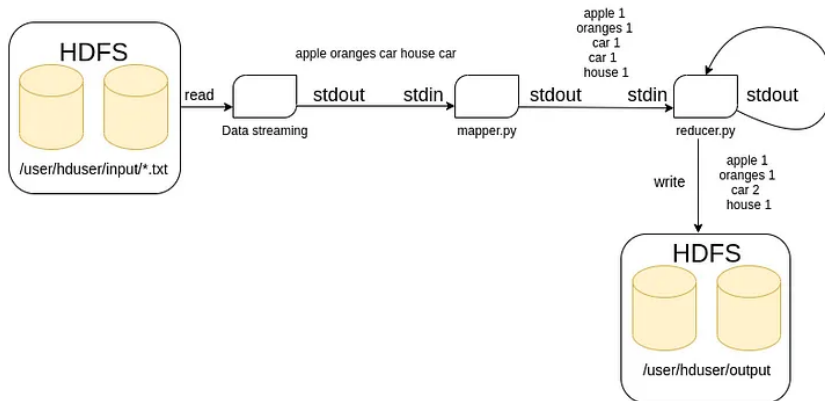
Hadoop Pipes

- Interfaz C++ a Hadoop MapReduce
- Usa sockets como canal de comunicación entre el NodeManager y el proceso C++ que ejecuta el map o el reduce

- Como se ha visto, Hadoop ofrece alternativas para programar en otros lenguajes de programación
- Utilizaremos dos alternativas:
 - 1 Hadoop Streaming directamente
 - 2 Librerías, como MrJob, que también usan el interfaz de Streaming, pero ofrecen más abstracciones y permiten por ejemplo lanzar trabajos en otras arquitecturas (EMR, Spark, etc.)

MapReduce en Python (cont.)

MapReduce Example



(fuente: <https://rancavil.medium.com/mapreduce-example-with-python-b435a9858718>)

MapReduce en Python: Hadoop Streaming – Mapper

El programa en Mapper Python correspondiente al programa visto en Java es el siguiente, para WordCount:

```
#!/usr/bin/env python3
import sys

for line in sys.stdin:    # Input is read from STDIN
    line = line.strip()   # remove leading and trailing whitespace
    words = line.split()  # split the line into words

    for word in words:
        # Output is written to STDOUT, key, 1, separated by a tab (\t)
        print('%s\t%s' % (word, 1))
```

● Nótese que:

- Es un programa completo en Python (se puede ejecutar directamente para probar)
- Recibe el fichero de entrada por la entrada estándar
- Si recibe un conjunto de pares clave-valor normalmente están separados por tabulador ('\t') (en este caso se procesa la línea completamente)
- La salida se produce a STDOUT, y de forma estándar separa con tabulador

MapReduce en Python: Hadoop Streaming – Reducer

El programa en Reducer Python correspondiente al programa visto en Java es el siguiente, para WordCount:

```
import sys

current_word = None
current_count = 0
word = None

for line in sys.stdin:    # input comes from STDIN
    line = line.strip()    # remove leading and trailing whitespace

    word, count = line.split('\t', 1)    # split the input from mapper by a tab

    try:
        count = int(count)    # convert count from string to int
    except ValueError:
        continue    # Upon error, discard the line
```

MapReduce en Python: Hadoop Streaming – Reducer (cont.)

```
# comparing the current word with the previous word
# (since they are ordered by key (word))
if current_word == word:
    current_count += count
else:
    if current_word:
        # write result to STDOUT
        print('%s\t%s' % (current_word, current_count))
    current_count = count
    current_word = word

if current_word:    # do not forget to output the last word if needed!
    print('%s\t%s' % (current_word, current_count))
```

- Nótese que:

- Es un programa completo en Python (se puede ejecutar directamente para probar)
- Recibe los datos de entrada por la entrada estándar
- Recibe las salidas de los distintos mapper, en este caso, pares clave-valor separadas por tabulador:

```
aaa\t1  
aaa\t1  
bbb\t1  
ccc\t1
```

- **OJO:** La fase de *Shuffle&Sort* efectivamente ordena las claves
- El programa tiene que ir comparando cada clave con la clave de la línea anterior para ir reconstruyendo la cuenta, ya que recibe todos los pares en orden
- La salida se produce a STDOUT, y de forma estándar separa con tabulador

MapReduce en Python: Hadoop Streaming – Reducer (cont.)

Ejecución:

```
mapred streaming -file wordcount-mapper.py -mapper wordcount-mapper.py \  
-file wordcount-reducer.py \  
-reducer wordcount-reducer.py \  
-input <FICHERO EN HDFS (puede haber varios input)> \  
-output <DIRECTORIO EN HDFS>
```

- Los parámetros **-file** especifican los ficheros a enviar a cada uno de los contenedores de computación (se incluye el fichero mapper y reducer y los ficheros de código o datos que se necesiten)
- Se especifica el mapper y el reducer con sus correspondientes parámetros
- Puede haber varios input
- Se puede usar un único parámetro **-files** con los ficheros separados por comas

MapReduce en Python: MrJob

- MrJob es una biblioteca que permite escribir programas MapReduce en Python
- Permite ejecutar los programas en Hadoop, EMR, etc.
- En el caso de WordCount, el código sería:

```
from mrjob.job import MRJob

class MRWordFrequencyCount(MRJob):

    def mapper(self, _, line):
        line = line.strip()
        for word in line.split():
            yield word, 1

    def reducer(self, key, values):
        yield key, sum(values)

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

MapReduce en Python: MrJob (cont.)

- El programa hereda de la clase `MRJob`
- Incluye una función `mapper` y una función `reducer`, que son generadores
- Devuelven pares clave-valor a través de la construcción `yield`
 - `yield` devuelve un valor a la función que lo pide (normalmente un bucle `for` o una *comprehension*), y queda suspendida hasta que se le vuelva a pedir un nuevo valor
- La infraestructura de MrJob se encarga de leer los datos de entrada y de imprimir los resultados
- En las prácticas 2 y 3 veremos los distintos parámetros que pueden aceptar los trabajos de MrJob

MapReduce en Python: MrJob (cont.)

- Para ejecutar el programa, se puede hacer en local:

```
python3 wordcount.py <fichero>
```

Esto permite comprobar localmente si la salida es correcta

- En el caso de querer ejecutarlo en Hadoop, se tiene que utilizar la opción **-r** **hadoop**:

```
python3 wordcount.py -r hadoop hdfs:///ruta/al/fichero
```

6. Filesystems en Hadoop

Filesystems en Hadoop

Hadoop tiene una noción abstracta de los filesystems

- HDFS es un caso particular de filesystem

Algunos filesystems soportados:

FS	URI	Descripción
Local	<i>file</i>	Disco local
HDFS	<i>hdfs</i>	Sistema HDFS
HFTP	<i>hftp</i>	RO acceso a HDFS sobre HTTP
HSFTP	<i>hsftp</i>	RO acceso a HDFS sobre HTTPS
WebHDFS	<i>webhdfs</i>	RW acceso a HDFS sobre HTTP
S3 (nativo)	<i>s3n</i>	Acceso a S3 nativo
S3 (block)	<i>s3</i>	Acceso a S3 en bloques

Ejemplo:

- `hadoop fs -ls file:///home/pepe`

Para usar con HDFS se recomienda el comando `hdfs dfs`:

- `hdfs dfs -help`

Tres modos principales:

- 1 Usando línea de comandos: comando `hdfs dfs`
 - Permite cargar, descargar y acceder a los ficheros desde línea de comandos
 - Vale para todos los filesystems soportados
- 2 Usando el interfaz web
- 3 Programáticamente: API Java/Python
- 4 Mediante otras interfaces: WebHDFS, HFTP, HDFS NFS Gateway

7. Interfaz en línea de comandos

Interfaz en línea de comandos (I)

Algunos comandos de manejo de ficheros

Comando	Significado
hdfs dfs -ls <path>	Lista ficheros
hdfs dfs -ls -R <path>	Lista recursivamente
hdfs dfs -cp <src> <dst>	Copia ficheros HDFS a HDFS
hdfs dfs -mv <src> <dst>	Mueve ficheros HDFS a HDFS
hdfs dfs -rm <path>	Borra ficheros en HDFS
hdfs dfs -rm -r <path>	Borra recursivamente
hdfs dfs -cat <path>	Muestra fichero en HDFS
hdfs dfs -tail <path>	Muestra el final del fichero
hdfs dfs -stat <path>	Muestra estadísticas del fichero
hdfs dfs -mkdir <path>	Crea directorio en HDFS
hdfs dfs -chmod ...	Cambia permisos de fichero
hdfs dfs -chown ...	Cambia propietario/grupo de fichero
hdfs dfs -du <path>	Espacio en bytes ocupado por ficheros
hdfs dfs -du -s <path>	Espacio ocupado acumulado
hdfs dfs -count <paths>	Cuenta nº dirs/ficheros/bytes

Interfaz en línea de comandos (II)

Movimiento de ficheros del sistema local al HDFS:

Comando	Significado
<code>hdfs dfs -put <local> <dst></code>	Copia de local a HDFS
<code>hdfs dfs -copyFromLocal ...</code>	Igual que -put
<code>hdfs dfs -moveFromLocal ...</code>	Mueve de local a HDFS
<code>hdfs dfs -get <src> <loc></code>	Copia de HDFS a local
<code>hdfs dfs -copyToLocal ...</code>	Copia de HDFS a local
<code>hdfs dfs -getmerge ...</code>	Copia y concatena de HDFS a local
<code>hdfs dfs -text <path></code>	Muestra el fichero en texto

Interfaz en línea de comandos (III)

Otros comandos:

Comando	Significado
<code>hdfs dfs -setrep <path></code>	Cambia el nivel de replicación
<code>hdfs dfs -test -[defsz] <path></code>	Tests sobre el fichero
<code>hdfs dfs -touchz <path></code>	Crea fichero vacío
<code>hdfs dfs -expunge</code>	Vacía la papelera
<code>hdfs dfs -usage [cmd]</code>	Ayuda uso de comandos

Más información: <http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/FileSystemShell.html>

8. Interfaz Java

API que permite interactuar con los filesystems soportados por Hadoop

- Utiliza la clase abstracta `org.apache.hadoop.fs.FileSystem`

Otras clases de interés en `org.apache.hadoop.fs` y `org.apache.hadoop.io`

- `Path`: representa a un fichero en un `FileSystem`
- `FileStatus`: información del fichero
- `FSDataInputStream`: stream de entrada de datos para un fichero, con acceso aleatorio
- `FSDataOutputStream`: stream de salida de datos para un fichero
- `IOUtils`: Funcionalidades para I/O

Ejemplo: lectura de un fichero en HDFS

```
public class FileSystemCat {  
    public static void main(String[] args) throws Exception {  
        String uri = args[0];  
        // Configuración por defecto  
        Configuration conf = new Configuration();  
        // Objeto para acceder al filesystem HDFS  
        FileSystem fs = FileSystem.get(URI.create(uri), conf);  
        // InputStream  
        FSDataInputStream in = null;  
        try {  
            // Abre el FSDataInputStream con el PATH indicado  
            in = fs.open(new Path(uri));  
            // Copia con un buffer de 4096 bytes  
            // No cierra los buffers al terminar (false)  
            IOUtils.copyBytes(in, System.out, 4096, false);  
        } finally {  
            IOUtils.closeStream(in);  
        }  
    }  
}
```

Ejecución del código anterior

Definir correctamente la variable `HADOOP_CLASSPATH` y usar el comando `hdfs` para lanzar el fichero `class`

```
$ export HADOOP_CLASSPATH="."
$ hdfs mipaquete.FileSystemCat fichero_en_HDFS
```

También es posible obtener el fichero `jar` y ejecutarlo con `hadoop jar` (no es una aplicación YARN)

Interfaces implementadas por FSDataInputStream:

- ➊ **Seekable**: permite movernos a una posición en el fichero (método `seek`)
- ➋ **PositionedReadable**: permite copiar a un buffer partes de un fichero

Dos métodos de `FileSystem` para abrir los ficheros para escritura:

- 1 `create`: crea un fichero para escritura (crea los directorios padre, si es preciso)
- 2 `append`: abre un fichero para añadir datos

Ambos métodos devuelven un `FSDatOutputStream`

- `FSDatOutputStream` no permite `seek` (solo escritura al final del fichero)
- El método `hflush()` garantiza coherencia, los datos son visibles para nuevos lectores
- El método `hsync()` garantiza que los datos se mandan a disco (pero pueden estar en la caché del disco)

Otras operaciones con ficheros y directorios

Crear un directorio:

- Método `mkdirs` de `FileSystem`

Información sobre ficheros y directorios:

- Métodos `getFileStatus` y `listStatus` de `FileSystem`
- Clase `FileStatus`

Patrones de nombres de ficheros (*globbing*)

- Método `globStatus` de `FileSystem`
- Interfaz `PathFilter`, para filtrar con expresiones regulares

Borrar ficheros o directorios, de forma recursiva o no:

- Método `delete` de `FileSystem`

9. Interfaz Python

- El interfaz Python depende de la librería usada
- Existen varias, pero la más reciente y utilizada es **pyarrow**
- Esta librería tiene una abstracción que permite tratar de la misma forma varios tipos de sistemas de ficheros (cloud, local, HDFS, etc.)
- El interfaz es parecido al que hemos visto para Java, y veremos algunos ejemplos
- **OJO:** **pyarrow** necesita encontrar todas las librerías de HDFS, por lo que hay que establecer primero:

```
export CLASSPATH='hdfs classpath --glob'
```

Acceder a un sistema de ficheros HDFS

pyarrow tiene el tipo específico `pyarrow.fs.HadoopFileSystem`:

```
from pyarrow import fs
from pyarrow.fs import HadoopFileSystem

hdfs = HadoopFileSystem("hdfs://namenode", port=9000, user='luser')
```

Acceder a un sistema de ficheros HDFS (cont.)

Los métodos son los heredados de la clase abstracta `pyarrow.fs.FileSystem`. Aquí los métodos para tratar con los metadatos, creación, información y borrado de ficheros y directorios:

- `copy_file(self, src, dest)` – Copy a file.
- `create_dir(self, path, *, bool recursive=True)` – Create a directory and subdirectories.
- `delete_dir(self, path)` – Delete a directory and its contents, recursively.
- `delete_dir_contents(self, path, *, ...)` – Delete a directory's contents, recursively.
- `delete_file(self, path)` – Delete a file.
- `equals(self, FileSystem other)`
- `from_uri(uri)` – Instantiate `HadoopFileSystem` object from an URI string.
- `get_file_info(self, paths_or_selector)` – Get info for the given files.
- `move(self, src, dest)` – Move / rename a file or directory.
- `normalize_path(self, path)` – Normalize filesystem path.

Acceder a un sistema de ficheros HDFS (cont.)

- Para listar el contenido de un directorio, hay que hacer uso de la clase `pyarrow.fs.FileSelector`. En ella se especifica un *path* padre y si se quiere entrar recursivamente en subdirectorios
- El listado serán elementos `pyarrow.fs.FileInfo`:

```
from pyarrow.fs import FileSelector

file_selector = FileSelector('patentes-mini')
for file_info in hdfs.get_file_info(file_selector, recursive=False):
    print(file_info)
```

Y la salida:

```
<FileInfo for 'patentes-mini/apat63_99.txt': type=FileType.File, size=1570472>
<FileInfo for 'patentes-mini/cite75_99.txt': type=FileType.File, size=132243>
<FileInfo for 'patentes-mini/country_codes.txt': type=FileType.File, size=3637>
```

Operaciones para *streams* de ficheros. Los *streams* se tratan como la abstracción `file` de Python:

- `open_append_stream(self, path[, ...])` – Open an output stream for appending.
- `open_input_file(self, path)` – Open an input file for random access reading.
- `open_input_stream(self, path[, compression, ...])` – Open an input stream for sequential reading.
- `open_output_stream(self, path[, ...])` – Open an output stream for sequential writing.

Acceder a un sistema de ficheros HDFS (cont.)

- Creación de un fichero y escritura **secuencial**:

```
with hdfs.open_output_stream('fichero') as stream:  
    stream.write(b'data')
```

- Apertura para lectura secuencial:

```
import sys  
import shutil  
  
with hdfs.open_input_stream('fichero') as stream:  
    data: binary = stream.read(4096)
```

- Apertura para escritura al final (*append*):

```
with hdfs.open_append_stream('fichero') as stream:  
    stream.write(b'data')
```

Hay varias formas de hacerlo. La más sencilla, pero tiene el problema de que lee todo el contenido primero en memoria

```
from pyarrow import fs
from pyarrow.fs import HadoopFileSystem
import sys

hdfs = HadoopFileSystem("hdfs://namenode", port=9000, user='luser')
with hdfs.open_input_stream(sys.argv[1]) as stream:
    print(stream.readall())
```

Otra más sofisticada usando las bibliotecas del sistema:

```
from pyarrow import fs
from pyarrow.fs import HadoopFileSystem
import sys
import shutil

hdfs = HadoopFileSystem("hdfs://namenode", port=9000, user='luser')

with hdfs.open_input_stream(sys.argv[1]) as stream:
    shutil.copyfileobj(stream, sys.stdout.buffer)
```


FileSystemCat en Python (cont.)

Y otra versión más "estándar" (se utiliza `os.write` porque permite escribir datos en binario. El *stream* de `pyarrow` lee datos binarios, y daría error al escribir en `sys.stdout`, ya que este último sólo acepta `str`, no `bytes`):

```
from pyarrow import fs
from pyarrow.fs import HadoopFileSystem
import sys
import os

hdfs = HadoopFileSystem("hdfs://namenode", port=9000, user='luser')

with hdfs.open_input_stream(sys.argv[1]) as stream:
    # Read stream in chunks and copy to stdout
    while True:
        data: bytes = stream.read(4096)
        if not data:
            break
        os.write(sys.stdout.fileno(), data)
```

Otras herramientas para mover datos

Es posible mover datos a/desde HDFS usando otras herramientas

- **distcp** Transferir datos en paralelo entre dos filesystems Hadoop
 - Ejemplo

```
hadoop distcp hdfs://nnode1/foo hdfs://nnode2/bar
```
 - Aplicación MapReduce map-only
 - Puede usar otros filesystems (HFTP, WebHDFS, etc.)
 - Interesante para mover cantidades masivas de datos
 - Más opciones: `hadoop distcp`
- Apache Flume servicio para recoger, agregar y mover grandes cantidades de datos de log a HDFS
- Apache Sqoop transferencia masivas de datos entre bases de datos estructuradas y HDFS

10. Herramientas para la gestión del HDFS

Hadoop proporciona un conjunto de herramientas para chequear y optimizar el HDFS

- `hdfs dfsadmin`: optiene información del estado del HDFS
- `hdfs fsck`: chequeo del filesystem
- `hdfs balancer`: herramienta de rebalanceo de bloques entre datanodes

Algunas opciones (usar `hdfs dfsadmin comando`)

Comando	Significado
<code>-help</code>	Ayuda
<code>-report</code>	Muestra estadísticas del filesystem
<code>-setQuota</code>	Fija una cuota en el número de nombres en un directorio (nº de ficheros/directorios)
<code>-clrQuota</code>	Borra la cuota de nombres
<code>-setSpaceQuota</code>	Fija una cuota en el espacio ocupado en un directorio
<code>-clrSpaceQuota</code>	Borra la cuota de espacio
<code>-refreshNodes</code>	Actualiza los nodos que se pueden conectar
<code>-safemode</code> <code>-saveNameSpace</code>	fija o chequea el <i>safe mode</i> en <i>safe mode</i> , salva el filesystem en memoria a un nuevo fichero <code>fsimage</code> y resetea el fichero <code>edits</code>

Chequea la salud de los ficheros en HDFS

- Chequea los bloques:
 - *Over-replicated*: con replicas de más
 - *Under-replicated*: con replicas de menos
 - *Misreplicated*: replicas mal colocadas
 - Corruptos
 - *Missing replicas*: sin réplicas
- Ejemplos:
 - Chequea recursivamente todo el HDFS
`hdfs fsck /`
 - Informa del número de bloques de un fichero y su localización
`hdfs fsck /user/pepe/foo -files -blocks -racks`

11. Otras interfaces a HDFS

Otros modos de acceder a HDFS

- WebHDFS: proporciona una API REST para acceder a HDFS mediante HTTP
 - Necesita acceso a los nodos del cluster (los datos se transmiten directamente desde los nodos)
 - Activada mediante la propiedad `dfs.webhdfs.enabled` del fichero `hdfs-site.xml` del Namenode (por defecto, `true`)
- HttpFS: servidor que proporciona un gateway REST HTTP gateway que soporta las operaciones de HDFS (lectura/escritura)
 - El servidor de HttpFS actúa como gateway: permite acceder a los datos en HDFS detrás de un firewall
- HDFS NFS Gateway: soporta NFSv3 y permite que HDFS sea montado como parte del sistema de ficheros local del cliente
 - Permite usar HDFS como un sistema de ficheros UNIX local
 - Permite copiar ficheros de HDFS al sistema de ficheros local y viceversa soportadas (p.e. no permite random writes ni enlaces duros)
 - Más información: <http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsNfsGateway.html>

12. Otros aspectos

Namenode principal

Estructura de directorios

```
${dfs.namenode.name.dir}/  
├── current  
│   ├── VERSION  
│   ├── edits_000000000000000000000001-0000000000000000000019  
│   ├── edits_inprogress_0000000000000000000020  
│   ├── fsimage_0000000000000000000000  
│   ├── fsimage_0000000000000000000000.md5  
│   ├── fsimage_0000000000000000000019  
│   ├── fsimage_0000000000000000000019.md5  
│   └── seen_txid  
└── in_use.lock
```

Ficheros en `dfs.namenode.name.dir`

- `VERSION` información sobre la versión de HDFS
- Ficheros `edits_startID-endID`: logs de transacciones ya finalizadas
- Fichero `edits_inprogress_startID`: logs de transacciones actuales
- Ficheros `fsimage`: información de los metadatos del filesystem
 - Contiene información de directorios y ficheros, incluyendo los bloques (inodos) que los forman
 - La localización de los bloques en los Datanodes se guarda en memoria

Cuando se inicia el Namenode:

- ➊ Carga el último `fsimage` en memoria y aplica las modificaciones indicadas en `edits`
- ➋ Con esta imagen reconstruida, crea un nuevo `fsimage` y un `edits` vacío
- ➌ Espera a que los Datanodes le envíen información de los bloques que tienen
 - Esta información se guarda en memoria

Durante la inicialización, el sistema está en modo seguro (*safe mode*)

- Solo permite acceso de lectura
- El modo seguro termina 30 segundos después de que el 99.9 % de los bloques alcancen un nivel mínimo de replicación
- Propiedades ajustables:

Propiedad	Por defecto
<code>dfs.namenode.replication.min</code>	1
<code>dfs.namenode.safemode.threshold-pct</code>	0.999
<code>dfs.namenode.safemode.extension</code>	30 s

Checkpoint node (aka Namenode secundario)

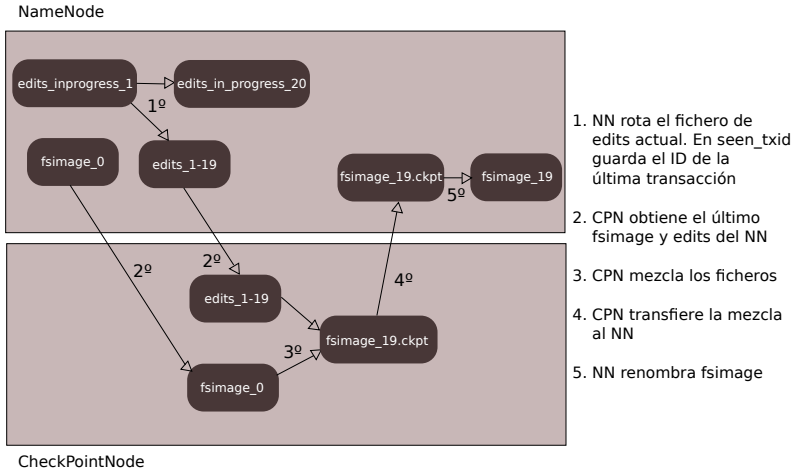
En un sistema ocupado, el fichero del `edits` puede crecer demasiado

- El Checkpoint Node se ocupa de mezclar `edits` y `fsimage` para inicializarlo
 - Proceso costoso en recursos
 - El CPN tiene requisitos de memoria similares a los del NN
- Checkpoint realizado cada hora (`dfs.namenode.checkpoint.period`) o cada 1 M transacciones (`dfs.namenode.checkpoint.txns`)
- Se puede cambiar por un Backup Node
 - Replica completa de la memoria del NN (necesita la misma cantidad de memoria)
 - Realiza los checkpoints

En caso de fallo total del Namenode, se puede recuperar el último checkpoint

- Iniciar el demonio del Namenode usando `hdfs namenode -importCheckpoint`

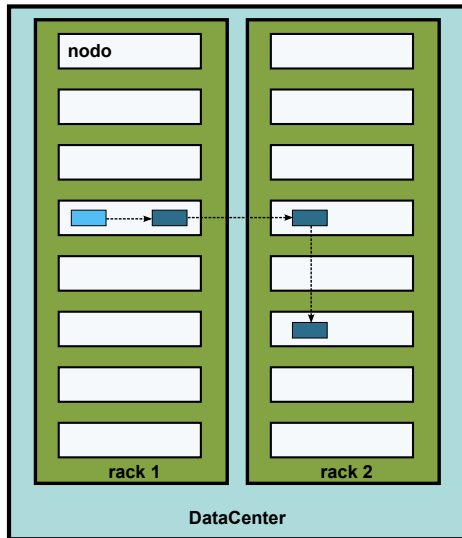
Checkpoint node



Localización de las réplicas

Política por defecto:

- 1ª réplica: en el nodo del cliente o en un nodo al azar
- 2ª réplica: en un *rack* diferente de la primera (elegido al azar)
- 3ª réplica: en el mismo rack que la 2ª, pero en otro nodo
- Otras réplicas: al azar (se intenta evitar colocar demasiadas réplicas en el mismo rack)



El Namenode es un *single point of failure* (SPOF)

- Si falla es imposible acceder a los datos
- Posibilidad de recuperación a partir de los checkpoints
- Conveniente guardar varias réplicas de los datos del namenode (RAID, indicar en `dfs.namenode.name.dir` directorios en diferentes máquinas, etc)

Mejoras en la versión 2.0

- HDFS High - Availability
- HDFS Federation

HDFS High - Availability

Un par de Namenodes en configuración activo - standby

- si falla el Namenode activo, el otro ocupa su lugar

Consideraciones

- Los Namenodes deben usar un almacenamiento compartido de alta disponibilidad
- Los Datanodes deben enviar informes de bloques a los dos Namenodes (el block mapping va en memoria, no en disco)
- Los Clientes deben manejar el fallo del Namenode de forma transparente

Más información: hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithNFS.html,
hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithQJM.html

El Namenode mantiene, en memoria, referencias a cada fichero y bloque en el filesystem

- problemas de escalabilidad

HDF Federation, introducida en la versión 2.0

- Permite usar varios Namenodes
- Cada uno gestiona una porción del espacio de nombres del filesystem
- Los Namenodes no se coordinan entre sí
- Cada Datanodes se registra con todos los Namenodes

Más información: hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/Federation.html

Novedades en HDFS v3

- Uso de *códigos de borrado (erasure coding)* para reducir el overhead de la replicación
 - Reduce el overhead a no más del 50 %
 - Ejemplo: ficheros de 6 bloques:
 - replicación x3: 18 bloques
 - EC: 9 bloques (6 datos + 3 paridad)
 - Implica un mayor coste de procesamiento
- Soporte de múltiples NameNodes en stand-by
- Soporte de balanceo de datos intra-nodo