GESTIÓN DE DATOS Y DE INFORMACIÓN

TEMA 3

Base de datos Distribuidas: MapReduce, Spark y HDFS

1º Introducción al BigData 3
1.2 Características del Big Data 3
2º MapReduce 4
2.1 Fases de una tarea MapReduce 4
2.2 Optimizaciones de MapReduce 5
3º Apache Spark 5
3.1 Resilient Distribute Dataset (RDD) 6
3.2 Uso de Apache Spark desde Python 6
3.2.1 Funciones de creación y guardado de RDDs 7
3.2.2 Funciones que aplican transformaciones a los RDDs 7
3.2.3 Funciones que aplican acciones a los RDDs 8
3.3 Características avanzadas de Apache Spark 9
4º Apache Hadoop 9
4.1 MapReduce en Hadoop 9
4.2 Hadoop Distributed Filesystem (HDFS) 10
4.2.1 Lectura de ficheros en un HDFS 10
4.2.2 Escritura de ficheros en un HDFS 11

1º Introducción al BigData

Def. El Big Data es la confluencia de distintas tendencias tecnológicas basadas tanto en el procesamiento como en el almacenamiento de la información, las cuales empiezan a surgir a partir de los años 2000. Para que algo se pueda considerar Big Data debe cumplir dos factores:

- La cantidad de información a tratar no se puede almacenar ni procesar con un ordenador o servidor convencional.
- La cantidad de información debe crear a una gran velocidad.

-- La principal utilidad es Big Data es ayudarnos a entender mejor como funciona la realidad con el objetivo de poder ser más eficaces al interaccionar u operar con al misma. Podemos clasificar la información del Big Data en tres tipos de datos diferentes:

- **Datos estructurados:** Tienen una estructura fija y bien definida.
  - Suelen provenir de las bases de datos relacionales tradicionales, cuyo esquema forzaba a que todos los datos tuvieran una determinada estructura fija que no variara entre los ejemplares.
- **Datos Semiestructurados:** No tienen una estructura o formato fijo pero constan de etiquetas o marcadores con los que se puede extraer información con facilidad.
  - Suelen provenir de fuentes como páginas web, ficheros de log, documentos XML, etc
- **Datos no estructurados:** No tienen un tipo predefinido, ni estructuras ni etiquetas. En algunos casos se pueden usar metadatos para obtener información de forma sencilla, sin embargo, lo normal es que sea difícil de extraer o que se necesite el uso de inteligencia artificial.
  - Algunos ejemplos son ficheros multimedia, mensajes de e-mail o mensajería instantánea.

1.2 Características del Big Data

-- El Big Data consta de tres características principales, denominadas las tres Vs, las cuales son:

- **Volumen:** Es la característica principal del Big Data y se refiere a que se almacenan volúmenes crecientes de información que son muy grandes, por lo que es necesario tener tecnologías específicas para almacenar y procesar tal cantidad de datos.
- **Velocidad:** Referente tanto a que los datos se generan con gran velocidad como a que la gran cantidad de información acumulada fuerza a las empresas a procesarla muy rápido.
- **Variedad:** Referente a que las fuentes de donde proceden los datos pueden ser muy diversas.

-- A parte de estas tres, también se pueden considerar dos características adicionales:

- **Veracidad:** Tener una gran cantidad de información proveniente de fuentes muy diversas puede no provocar la perdida de veracidad en las mismas, por lo que deberíamos establecer unas pautas que aseguren la fiabilidad de los datos obtenidos.
- **Valor:** Los datos deben poder obtenerse de manera rentable y eficiente.

2º MapReduce

Def. MapReduce es un esquema de cómputo diseñado para procesar grandes cantidades de datos en clúster de manera distribuida, utilizando para ello, un sistema de ficheros distribuido.

-- Este esquema está basado en al estrategia de “divide y vencerás”, de modo que se divide el set de datos completo sobre los que se tiene que realizar el análisis en fragmentos más pequeños. Cada uno de estos fragmentos será procesado por un nodo del clúster y posteriormente se agregarán los resultados obtenidos de cada uno de los nodos participantes.

-- MapReduce esta formado por dos funciones principales, las cuales son:

1. **Función Map:** Se aplica en primer lugar de manera distribuida sobre el conjunto de datos a analizar. Esta función toma una pareja del tipo (K_1, V_1) y devuelve parejas del tipo (K_2, V_2).
2. **Función Reduce:** Se aplica en segundo lugar sobre los datos devueltos por la función Map. Esta función toma una pareja del tipo (K_2, [V_2]) y devuelve parejas del tipo (K_3, V_3).

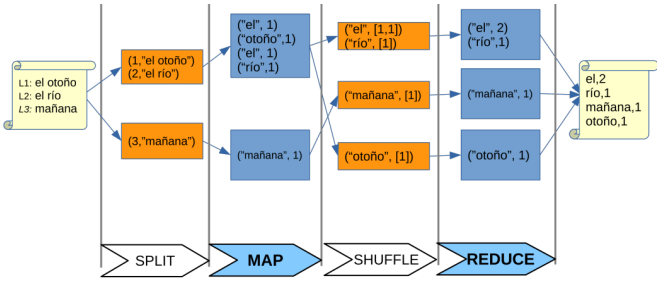
-- Tanto al función Map como la función Reduce pueden devolver 0, 1 o varias parejas. El tipo de las parejas tomadas como entrada por la función Reduce será determinado por la función Map.

```
# key : None
# line: str
def mapper(self, key, line):
  for word in line.split():
    yield (word, 1) # Emite la pareja
```

```
# key    : str
# values: generador de int
def reducer(self, key, values):
  yield (key, sum(values)) # Emite pareja
```

2.1 Fases de una tarea MapReduce

1. **Fase Split:** Divide el fichero en varios chunks con un tamaño aproximado entre 64 y 128 MB, cada uno de los cuales es asignado a un *mapper* del clúster. Lo ideal es maximizar la localidad de los datos, haciendo que el *mapper* se ejecute en el nodo que ya tiene almacenado el chunk.
2. **Fase Map:** Se divide el chunk en varias parejas (clave, valor) y para cada una de ellas el *mapper* aplica la función Map proporcionada por el usuario. Las parejas generadas por la función Map de almacenan de forma local en el equipo del clúster, de modo que al terminar el *mapper* el nodo tendrá almacenados una serie de parejas intermedias.
3. **Fase Suffle anda Sort:** Fase encargada de repartir las parejas intermedias generadas por la función Map entre los distintos *reducers*. Este reparto se puede hacer atendiendo a varios criterios, pero dos parejas con la misma clave siempre deberán ser asignadas al mismo *reducer*.
    - Una vez se han repartido las parejas, estas se ordenan por el valor de su clave y se agrupan, dando como resultado parejas (clave, [valor]), donde el valor es la secuencia de los distintos valores asociados a dicha clave. Estas parejas serán la entrada que recibirá la función Reduce.
4. **Fase Reduce:** Se aplica la función Reduce proporcionada por el usuario a las parejas de entrada del tipo (clave, [valor]). El *reducer* vuelca la salida de la función Reduce en un fichero dentro del sistema distribuido, de modo que estos ficheros se pueden combinar posteriormente o ser utilizados como entrada de nuevas tareas MapReduce.



2.2 Optimizaciones de MapReduce

-- Transferir una gran cantidad de parejas entre los nodos que forman el clúster congestiona la red y afecta al rendimiento de la tarea MapReduce. Existen varias formas de optimizar el tráfico generado entre la fase Map y la Reduce, uno d ellos cuales es utilizar la fase intermedia Combiner.

-- Combiner es una fase adicional que se puede incluir en la ejecución de una tarea Mapreduce y que se encuentra entre las fases Map y Shuffle. La función de esta es agregar las parejas generadas por el *mapper* para reducir la cantidad de datos a transmitir entre los nodos.

-- La función Combiner toma parejas de entrada del tipo (K_2, [V_2]), de modo que será necesario realizar la combinación de las parejas generadas por cada *mapper* antes de ejecutar el Combiner. Las parejas devueltas por la función Combine deberán ser del mismo tipo que las generadas por la función Map, es decir (K_2, V_2).

```
# key    : str
# values: generador de int
def combiner(self, key, values):
  yield (key, sum(values))
```

-- El sistema no tiene la obligación de realizar la fase Combiner, se trata de una fase de optimización, por lo que se ejecutará si al sistema le viene bien ejecutarla. Tampoco es necesario que se ejecute en todas las parejas de todos los nodos *mapper*, sino que puede ejecutarse en subconjuntos de parejas.

-- La tarea MapReduce debe producir la misma salida independientemente de si se ejecuta o no la fase Combiner y sobre cuantas parejas se ejecuta. Para cumplir esto, la fase Combiner deberá ser conmutativa y asociativa.

3º Apache Spark

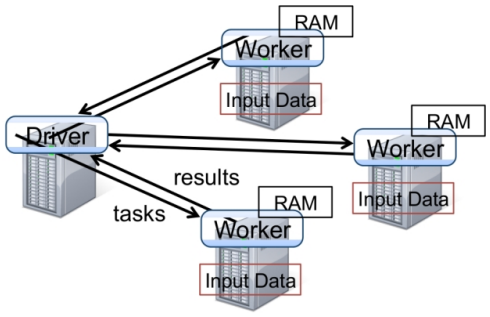
Def. Apache Spark es uno de los proyectos más activos actualmente de Apache, el cual conforma un sistema de computo distribuido alternativo al uso de MapReduce. Al igual que este último, Apache Spark cuenta con dos características principales:

- **Escalabilidad:** Aumentar la capacidad de cómputo es tan sencillo como añadir nuevos equipos al clúster donde se está ejecutando Apache Spark.
- **Resistencia a fallos:** Los equipos que forman el clúster pueden dejar de funcionar, pero el cómputo recuperará los datos perdido y seguirá funcionando.

-- Apache Spark es mucho más flexible que MapReduce gracias a que proporciona una gran cantidad de operaciones para aplicar sobre el conjunto de datos. Estas operaciones se pueden combinar sin la necesidad de volver los resultados intermedios en el disco.

-- Por otra parte, Apache Spark almacena los datos utilizados en memoria principal, lo que hace que la velocidad de computo es mucho mayor. El lenguaje oficial que utiliza Apache Spark es Scala, sin embargo, también tiene soporte para los lenguajes Java, Python y R. Además del núcleo,

-- El programa *driver* de Apache Spark se conecta a un clúster de *workers*, y su función es definir uno o varios RDDs e invocar acciones a realizar por los *workers* con dichos RDDs. Los *workers* pueden almacenar fragmentos de los RDDs en memoria entre operaciones.



3.1 Resilient Distribute Dataset (RDD)

Def. Un RDD es una colecciones de elementos que se reparte entre la memoria de los diferentes equipos que conforman el clúster. Se trata del concepto fundamentar de Apache Spark sobre el cual se expresan todos los cómputos.

-- Los elementos que conforma el RDD deben ser todos del mismo tipo, o bien ser parejas con los mismos componentes. Los RDD cumplen dos características:

- **Resiliencia:** Si un equipo deja de funcionar, el fragmento de RDD que contenía dejará de ser accesible, pero este no se perderá, ya que el resto de equipo podrá recalcularlo.
- **Inmutabilidad:** Una vez se ha creado un RDD, este no se puede modificar.

-- Las distintas particiones de un RDD son calculadas de forma automática por Spark, sin embargo, el programador puede indicar cuantas particiones quiere crear o como se debe realizar el particionado. Los RDDs pueden ser creados desde ficheros o ser el resultado de transformar otros RDDs.

-- Sobre los RDD de construyen interfaces de cómputo de alto nivel como pueden ser los DataFrames o los DataSets, los cuales permiten realizar operaciones en paralelo que afectan a todos los fragmentos que componen el RDD.Sobre los RDDs se pueden realizar dos tipos de operaciones:

- **Transformaciones:** Realizan algún calculo sobre los datos y generan un nuevo RDD. Las transformaciones no se ejecutan de forma instantánea, sino que son planificadas en el tiempo, de modo que retrasarlas puede permitir optimizar su ejecución.
- **Acciones:** Retornan un valor al *driver* o vuelcan los datos en el almacenamiento. Las acciones son realizadas una vez se han llevado a cabo todas las transformaciones.

3.2 Uso de Apache Spark desde Python

-- El lanzamiento de Apache Spark desde Python puede realizarse utilizando dos ordenes distintas:

- **pyspark:** Lanza el intérprete interactivo de Spark.
- **Spark-submit:** Permite lanzar un fichero Python como programa Spark completo.

-- La mayoría de operaciones utilizadas en Spark aceptan funciones como parámetros. Si la función que queremos pasar como parámetro ya se encuentra definida, podremos hacerlo unicamente utilizando su nombre, aunque normalmente utilizaremos funciones anónimas.

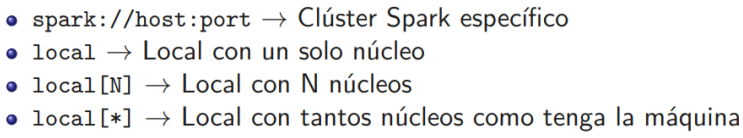
-- El punto principal de entrada de Spark es la clase *SparkSession*, la cual incluye los métodos *SparkContext*, utilizados para la creación de EDDs.

-- Si utilizamos pySpark, este cargará automáticamente la clase *SparkSession* dentro de la variable *spark*, y el contenido de *SpartContext* en la variable *sc*. En el caso de utilizar spark-submit deberemos realizar estos pasos en el inicio del programa.

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()
sc = spark.sparkContext
```

-- Spark-submit admite varios parámetros de entrada, siendo el mas importante es parámetro *master*, con el cual podremos indicar la URL del clúster en el que se ejecutará el programa Spark.



3.2.1 Funciones de creación y guardado de RDDs

- **SparkContext.textFile (name):** Crea un RDD con los datos guardados en uno o varios ficheros indicados como parámetro.
  - **Name <string>:** Nombre del fichero del cual se quiere obtener los datos para formal el RDD. Se pueden especificar varios ficheros separados por un espacio.
  - **MinPartitions <int> = None?:** Número mínimo de particiones de queremos que tenga el RDD devuelto por la función.
  - **use_unicode <string> = True?**

- **SparkContext.parallelize (c):** Crea un RDD a partir de una colección de datos existente en el programa indicados como parámetro.
  - **c <[iterable]>:** Iterable donde se encuentran los datos con los que se formará el RDD.
  - **numSlices <[int]> = None?**

- **RDD.saveAsTextFile (path):** Guarda el contenido del RDD que ejecuta la función en un archivo en forma de cadenas de texto.
  - **path <string>:** Dirección del archivo donde se va a guardar el RDD.
  - **CompressionCodecClass <string> = None?:** Nombre completo del codec de compresión que se va a utilizar para generar y guardar el archivo.

- **RDD.saveAsNewAPIHadoopDataset (conf) → None:** Envíe un RDD de Python de pares clave-valor a un sistema de archivos Hadoop, utilizando la API Hadoop OutputFormat. Las pares se convierten para la salida utilizando convertidores que pueden ser especificados por el usuario.
  - **conf <dict[str, str]>:** Configuración del sistema Hadoop al cual se envía la información.
  - **keyConverter <str> = None?:** Clase utilizada como convertidor de claves.
  - **valueConverter: <str> = None?:** Clase utilizada como convertidor de valores.

- **RDD.saveAsPickleFile (path: str, batchSize: int = 10) → None:** Guarda el contenido del RDD que ejecuta la función en un archivo en forma de objetos serializados.
  - **path <string>:** Dirección del archivo donde se va a guardar el RDD.
  - **batchSize <int> = 10?**

- **RDD.saveAsSequenceFile (path, compressionCodecClass) → None:** Envíe un RDD de Python de pares clave-valor a un sistema de archivos Hadoop, el cual utilizará los tipos “writable”para convertir los datos del RDD y almacenarlos como una secuencia.
  - **path <string>:** Dirección del archivo donde se va a guardar el RDD.
  - **compressionCodecClass <string> = None?:** Nombre completo del codec de compresión que se va a utilizar para generar y guardar el archivo.

3.2.2 Funciones que aplican transformaciones a los RDDs

- **RDD.filter (condition):** Crea y devuelve un nuevo RDD formado por todos aquellos elementos contenidos en el RDD que ha ejecutado la función, los cuales han pasado el filtro indicado como parámetro. Este filtro se trata de una función que toma un elemento y devuelve un booleano.
  - **Condition <function>:** Función que realizará el filtrado.

- **RDD.map (f):** Crea y devuelve un nuevo RDD formado por todos aquellos elementos contenidos en el RDD que ha ejecutado la función, a los cuales ha aplicado la función indicada como parámetro.
  - **f <function>:** Función que se aplicará a los elementos del RDD.
  - **PreservesPartitioning: <bool> = False?**

- **RDD.flatMap (f):** Crea y devuelve un nuevo RDD formado por todos aquellos elementos contenidos en el RDD que ha ejecutado la función, a los cuales ha aplicado la función indicada como parámetro y sobre los que posteriormente se ha realizado en aplanamiento.
  - **f <function>:** Función que se aplicará a los elementos del RDD.
  - **PreservesPartitioning: <bool> = False?**

- **RDD.union (other):** Crea y devuelve un nuevo RDD formado por la unión del que ha invocado la función (poniendo los elementos de este en primer lugar) y el RDD indicado como parámetro.
  - **Other <RDD>:** RDD con el que se va a realizar la fusión.

- **RDD.reduceByKey (f) → RDD:** Crea y devuelve un RDD cuyos campos son una combinación de todos los valores asociados a la misma clave y aplicando a los mismos la función indicada como parámetro. El RDD que invoca la función debe contener datos del tipo clave-valor.
  - **f <function>:** Función que se aplicará a cada uno de los elementos del RDD.
  - **numPartitions <int> = None?:** Indica el número de particiones que se realizarán en el RDD devuelto por la función.
  - **PartitionFunc <function> = functionHash:** Función aplicada como particionador.

- **RDD.keys () → RDD:** Retorna un RDD conformado por los elementos clave que contenidos en el RDD que ejecuta la función, sin eliminarse las claves duplicadas. El RDD que invoca la función debe contener datos del tipo clave-valor.

- **RDD.join(other) → RDD:** Combina el RDD que invoca la función con el RDD indicado como parámetro uniendo los valores de las claves comunes y omitiendo el resto. El RDD que invoca la función debe contener datos del tipo clave-valor.
  - **other <RDD>:** RDD con el que se va a realizar la combinación.
  - **numPartitions <int> = None?:** Indica el número de particiones que se realizarán en el RDD devuelto por la función.

3.2.3 Funciones que aplican acciones a los RDDs

- **RDD.collect () → list<>:** Crea y devuelve una colección formada por todos los elementos del RDD que invoca la función.

- **RDD.count () → int:** Retorna el número de elementos que contiene el RDD.

- **RDD.countByValue () → Dict[K, int]:** Retorna un array de elementos clave-valor con el número de veces que aparece cada elemento dentro del RDD.

- **RDD.reduce (f):** Combina los elementos del RDD utilizando la función indicada como parámetro. La función indicada debe ser conmutativa y asociativa.
  - **f <function>:** Función que se aplicará a los elementos del RDD para hacer la reducción.

3.3 Características avanzadas de Apache Spark

-- Apache Spark utiliza un tipo de variables denominadas acumuladores, las cuales forman parte del *SparkContext* y son compartidas entre todos los *workers*. Su función principal es de depuración y aunque los *workers* solo pueden incrementarla, el programa *driver* puede acceder al contenido.

```
nerr = sc.accumulator(0)
# Acumulador como variable global
lines = sc.textFile('fichero.txt')
result = lines.map(lambda x: parse(x,nerr))
result.count()  # Lanza el cómputo en el RDD
print('#Errors: ', nerr)
```

```
def parse(line, acc):
  if len(line) < 5:
    acc += 1  # Error detectado
    return []
  else:
    return line.split()
```

-- Los datos utilizados en las transformaciones se codifican y transmiten a cada una de las tareas ejecutadas en los *workers*. Esto puede supone run problema ya que un *worker* puede ejecutar varias tareas, d emodo que recibirá varias veces los mismos datos, produciendo un alto consumo de ancho de banda y aumentando el tiempo de ejecución.

-- Para evitar esto Apache Spark proporciona las variables *broadcast*, las cuales sirven para definir datos de solo lectura que pueden compartirse de manera eficiente entre los distintos workers. Estos datos se envían una solo vez a cada *worker* y son almacenados, de modo que pueden reutilizarse, además, el envío se realiza mediante un protocolo optimizado para la transmisión de grandes datos.

```
stopwords = ['a', 'the', 'an', 'to', ...]
stopwords_bc = sc.broadcast(stopwords)
# La variable de broadcast es global

def valid_word(word):
  return word not in stopwords_bc.value
  # Se accede al contenido de la variable de
  # a través del atributo 'value'


lines = sc.parallelize(['a', 'hello', 'a'])
lines.filter(valid_word).collect()
```

4º Apache Hadoop

Def. Apache Hadoop es un framework de código abierto para la computación distribuida centrado en ofrecer un funcionamiento escalable y confiable. Surge con la premisa de poder aplicar MapReduce a la resolución de problemas mediante el uso de un sistema de ficheros distribuido.

4.1 MapReduce en Hadoop

-- Las tareas de MapReduce realizadas en Hadoop son escritas en lenguaje Java. El *mapper* se define como una clase heredera de la clase *Mapper* y debe contiene el método *map()*, mientras que el *reducer* se define como una clase que hereda de la clase *Reducer* y que contiene el método *reduce()*.

-- Los métodos *map()* y *reduce()* emitiran las parejas a través del objeto *context.write(clave, valor)*. Existen tipos de datos predefinidos para los valores que componen dichas parejas

```
public static class IntSumReducer
  extends Reducer<Text,IntWritable,Text,IntWritable> {

  private IntWritable result = new IntWritable();

  public void reduce(Text key, Iterable<IntWritable> values,
                     Context context)
                     throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
  }
}
```

```
public static class IntSumReducer
  extends Reducer<Text,IntWritable,Text,IntWritable> {

  private IntWritable result = new IntWritable();

  public void reduce(Text key, Iterable<IntWritable> values,
                     Context context)
                     throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
  }
}
```

4.2 Hadoop Distributed Filesystem (HDFS)

Def. HDFS es un componente de Haddop para la gestión del sistema de ficheros distribuido entre las diferentes máquinas del clúster.

-- Se trata de un sistema diseñado para soportar ficheros de gran tamaño, el cual proporciona un gran rendimiento para aplicaciones de única escritura pero múltiples lecturas, esto gracias a la gran velocidad para leer un fichero completo.

-- Este sistema esta pensado para ejecutarse en un gran número de máquinas de bajo coste, las cuales se interconectan para formar clústers. Debido a esto, es mucho más fácil que las máquinas sufran errores, por lo que el sistema debe tener una gran tolerancia a fallos mediante replicación de datos.

-- Un clúster HDFS está compuesto por dos tipos diferentes de nodos:

- **Namenode:**Equipos principales que almacenan toda la información referente a los ficheros con los que se va a operar. Estos también contienen metadatos y los indicen de donde se encuentra cada bloque que componen los ficheros.
- **Datanodes:** Equipos secundarios que almacenan los bloques de datos que componen los ficheros.

-- El HDFS se organiza en bloques, los cuales son la unidad mínima de datos que se pueden transferir a un sistemas de archivos. El tamaño de estos bloques es relativamente grande, entre 64MB y 128 MB, con el objetivo de reducir el tiempo de búsqueda empleado por los equipos.

-- Esta organización en nodos permite: Dividir el fichero a operar entre distintos nodos del clúster, simplificar el almacenamiento de los datos y facilitar la replicación para conseguir una alta tolerancia a fallos.

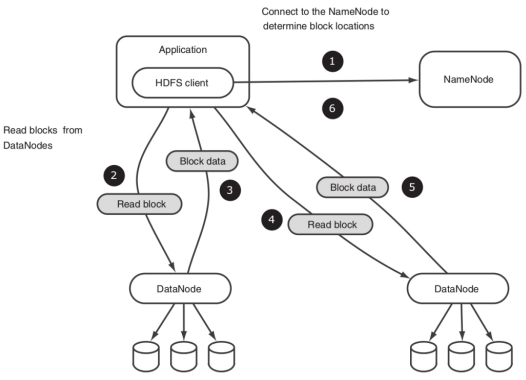
-- La información y correcto funcionamiento de los N*ameNodes* es muy importante, por lo que deberemos asegurar la tolerancia a fallos en los mismos. Esto se consigue de dos maneras:

- Escribiendo periódicamente el estado del sistema de ficheros con el objetivo de poder recuperarlo si se produce algún fallo.
- Teniendo N*ameNodes* secundarios que copien el estado del principal, de modo que estarían disponibles en el caso de que este falle.

4.2.1 Lectura de ficheros en un HDFS

-- El primer paso para realizar la lectura es que la aplicación se conecte con el N*ameNode* y le solicite la dirección de los primeros 10 bloques que conforman el archivo a operar. Los nodos son devueltos por orden de cercanía, es decir, si están en el mismo equipo, si están en otro equipo del mismo rack o si se encuentran en un rack distinto.

-- Una vez se conoce la dirección de los *DataNodes*, la aplicación se conecta con ellos y le solicita los datos deseados. Posteriormente se repetirá el proceso para los siguientes 10 bloques.



-- En el caso de que se produzca algún error de lectura o conexión, se probará con otro *DataNode* y se recordará este para no volver a probarlo más adelante. Si al comprobar el checksum de los bloques obtenidos se encuentra algún error,se informará al *NameNode* y se probará con otro *DataNode*.

4.2.2 Escritura de ficheros en un HDFS

-- El primer paso a realizar es que la aplicación se conecte con el *NameNode* indicándole que se va a llevar a cabo la escritura de un fichero. Este responde a la aplicación indicándole una lista de *DataNodes* para escribir el primer bloque, tendiendo en cuenta la cercanía.

-- En cada uno de los *DataNodes* indicados por se almacenará una copia de los nodos a escribir, teniendo en cuenta que al menos uno de ellos se sitúa fuera del propio rack donde se ejecuta la aplicación.

-- Cuando la aplciación tenga los suficientes datos se enviarán al primer *DataNode*, el cual los almacenará y reenviará al siguiente, y así sucesivamente. Cada uno de los *DataNodes* deberá afirmar la correcta escritura del bloque para confirmar que este se ha escrito con éxito.