Curso de Especialización de
Inteligencia Artificial y Big Data (IABD)

Sistemas de Big Data

UD02. Almacenamiento de Datos.
Resumen.

JUAN ANTONIO GARCIA MUELAS

Un fichero es la unidad atómica de almacenamiento empleada por un sistema de ficheros para almacenar datos.

Un **sistema de ficheros** (como por ejemplo FAT32, NTFS o EXT4) es la **infraestructura** necesaria **para** poder **almacenar y organizar datos** en unidades de almacenamiento. Mantiene una estructura de árbol que permite organizar la información en carpetas.

RAID es un **sistema de almacenamiento de datos** que es capaz de distribuirlos **por múltiples unidades**, según estrategias o niveles. No es un sistema de ficheros, sino una capa que trabaja por debajo de ellos.

RAID puede trabajar:

- ✓ Por **hardware**, mediante **controladoras RAID** específicas.
- ✓ Por **software**, haciendo que el procesador ejecute un **software** que realiza el trabajo **equivalente a**l que haría la **controladora** hardware (lo cual es **más lento pero más barato**).

Los niveles más usados son:

RAID 0: Conjunto dividido. Podemos ver **dos discos de 1TB como uno de 2TB sin replicación**.

- ✓ Permite ver unidades virtuales más grandes que las unidades físicas.
- ✓ Sin redundancia.
- ✓ Aproximadamente el doble de velocidad en lectura y escritura.

RAID 1: Conjunto en espejo. Podemos ver **dos discos de 1TB como uno de 1TB con replicación**.

- ✓ Permite tener los datos replicados de forma automática.
- ✓ Mayor seguridad desaprovechando capacidad.
- ✓ Hasta el doble de velocidad de lectura. Sin ganancia para escritura.
- ✓ **Tolerante al fallo** de uno de los discos.

RAID 5: Conjunto dividido con paridad distribuida. Podemos ver **tres discos de 1TB como uno de 2TB con replicación**.

- ✓ Es necesario un mínimo de 3 discos.
- ✓ Permite ver unidades virtuales más grandes que las unidades físicas añadiendo además replicación.
- ✓ Permite mayores rendimientos de lectura al poder leer de varias unidades a la vez.
- ✓ **Tolerante al fallo** de uno de los discos.

Un **sistema de ficheros distribuido** es aquel que es capaz de **almacenar ficheros** distribuyendo el contenido de estos **en las distintas máquinas que conforman un clúster**. Son **agnósticos respecto de los datos** a almacenar y **proporcionan redundancia y replicación** por bloques, que evitan el uso de RAID en los nodos **y almacenar ficheros más grandes** que el espacio de almacenamiento disponible en cualquiera de sus nodos.

No son ideales **para trabajar con ficheros muy pequeños**.

Uno de los **sistemas de ficheros distribuidos más conocidos y empleados** en entornos Big Data es **HDFS** (de *Apache Hadoop*), el cual **tiene como precursor a GFS** (de *Google*).

Dentro del mundo cloud, **el más conocido es *Amazon S3***, aunque existen otras alternativas como ***Google Cloud Storage*** y ***Azure Blob Storage***.

Cada vez es más común el **almacenamiento distribuido en memoria** para Big Data, por el **precio** de la memoria **RAM** cada vez **más bajo y más disponibilidad** en cada máquina, **eliminando** el problema de **latencia en entrada-salida** de las unidades de almacenamiento **y** del **tiempo de transferencia** entre el almacenamiento y la memoria.

El **mayor problema** de los almacenamientos en memoria es que **no proporcionan durabilidad** de por sí (los datos se pierden si se apaga la máquina). Por esa razón, estos almacenamientos en muchas ocasiones **se utilizan sólo** para datos que vamos a utilizar para algún tipo de **analítica** y vienen **copiados desde otro almacenamiento que sí es persistente**.

Se consigue la redundancia en los sistemas de ficheros distribuidos gracias a que el contenido de cada fichero es almacenado en más de un nodo del clúster.

Un almacenamiento en memoria es apropiado cuando:

- ✓ Los datos llegan rápidamente.
- ✓ Se requiere **analítica** continua y/o en **tiempo real** o en **streaming**.
- ✓ Es necesario realizar **consultas interactivas** o poder visualizar datos en tiempo real.
- ✓ El mismo conjunto de datos se utiliza a la vez para varias tareas.
- ✓ Es necesario poder **acceder de forma iterativa** al mismo conjunto de datos sin necesidad de volver a cargarlo de desde disco cada vez.
  - Análisis de datos exploratorio o iterativo o Algoritmos basados en grafos.
- ✓ Hay que desarrollar **soluciones** Big Data de **baja latencia** con que soporten transacciones *ACID*.

Un almacenamiento en memoria no es adecuado cuando:

- ✓ El procesamiento es por **lotes**.
- ✓ Necesitamos trabajar con **grandes cantidades de datos**.
- ✓ La persistencia de datos según llegan desde sus fuentes es una necesidad.
- ✓ El **presupuesto es limitado** y prevemos que vamos a ampliar memoria con nuevos nodos.

HDFS es uno de los sistemas de ficheros distribuidos para Big Data más conocidos y usados al formar parte de Hadoop (plataforma de Big Data opensource). Es una versión del sistema de ficheros distribuido de Google (GFS).

Hadoop está diseñado para ser **capaz de almacenar y gestionar una gran cantidad de datos** incluyendo ficheros de gran tamaño, distribuyéndolos por un clúster cuyos **nodos** son **commodity hardware**.

En un clúster HDFS encontraremos un nodo trabajando como Namenode y varios trabajando como Datanode.

Dado que permite guardar datos en nodos de precio reducido, se suele pensar en el **componente HDFS** como un "disco duro enorme que sale barato".
Gracias ello y a que es ***schema-on-read***, una estrategia muy común es **guardar en HDFS**, según llegan, todos los **datos que consideramos** que **pueden tener** algún **valor**, a la espera de ver **más adelante** cómo **serán interpretados** para obtener dicho valor.

Para alcanzar sus objetivos, **HDFS particiona** los ficheros **en bloques** de gran tamaño (**128 MB por defecto**, aunque ese valor es configurable), minimizando el número de búsquedas para trabajar a la velocidad de transferencia.

Esta característica implica que **HDFS no está optimizado para latencia sino para velocidad** de transferencia.

En HDFS el Namenode puede detectar cuándo un Datanode deja de estar accesible.

Otra característica interesante de HDFS es la llamada "**rack awareness**", según la cual se **guarda** constancia de cuál es la **topología de conexiones** (vía switch) **de los nodos** del clúster, **minimizando los saltos** entre switch.

ACCESO MEDIANTE LÍNEA DE COMANDO.

Listado de ficheros:
```
$ hadoop fs -ls <ruta>
```

Creación de **directorios**:
```
$ hadoop fs -mkdir <ruta>
```

Copia de ficheros del sistema de ficheros **local a HDFS**:
```
$ hadoop fs –copyFromLocal <origen_local> <destino>
```

Copia de ficheros **desde HDFS al** sistema de ficheros local:
```
$ hadoop fs –copyToLocal <origen> <destino_local>
```

Impresión del **contenido** de un fichero:
```
$ hadopp fs -cat <ruta_fichero> //fichero completo
```

```
$ hadopp fs -tail <ruta_fichero> //último kB
```

Copiar un fichero de una ruta a otra **dentro de HDFS**:
```
$ hadoop fs –cp <origen> <destino>
```

Mover un fichero de una ruta a otra **dentro de HDFS**:
```
$ hadoop fs –mv <origen> <destino>
```

Eliminar un fichero de HDFS:
```
$ hadoop fs –rm [-r] <ruta>
```

Acceder a la ruta para ver más comandos:
```
$ hadoop fs –help
```

ACCESO MEDIANTE PYTHON.

Existen varias librerías o paquetes en diversos lenguajes de programación.

Nos centraremos en **Snakebyte** en **Python** que implementa el protocolo **Hadoop RPC**. Permite acceder al Namenode **sin llamadas fs o hdfs dfs**.

```
from snakebite.client import Client
client = Client('localhost', 9000)
```
Listado del contenido de un directorio:
```
for x in client.ls(['/']):
```
Crear un directorio:
```
for x in client.mkdir(['/foo/bar', '/input'], create_parent=True):
```
Eliminar ficheros y directorios:
```
for x in client.delete(['/foo', '/input'], recurse=True):
```
Copiar ficheros de HDFS al sistema de ficheros local:
```
for x in client.copyToLocal(['/input/input.txt'], '/tmp'):
```
Lectura de ficheros de HDFS:
```
for x in client.text(['/input/input.txt']):
    print x
```

Las **bases de datos relacionales necesitan de conocer el formato de los registros** para escribirlos. **Para solventar** este problema **está NoSQL**.

Las bases de datos *NoSQL* ("no sólo *SQL*" o "no relacionales"), son un conjunto de bases de datos que por alguna razón **escapan** de la **denominación** de base de datos **relacional**.

Por lo general no usan *SQL* como lenguaje de consulta, aunque esto no es una regla absoluta.

Son BD schema-on-read, donde **los datos se pueden escribir sin atender a ningún esquema preestablecido**, porque donde **si se utiliza** algún tipo de esquema es en todo caso **al leerlos**.

Que una base de datos sea schema-on-write significa que debe establecerse el formato de los datos a almacenar antes de comenzar a escribirlos.

Conceptos generales:

- ✓ **Sharding: Dividir los conjuntos de datos en subconjuntos** (shards) **para distribuirlos** por un clúster. **Facilita tratar** conjuntos de **datos** más grandes que de forma tradicional, la **distribución** de la carga **entre nodos para** conseguir **escalabilidad** horizontal y **proporciona tolerancia parcial a fallos**.
  El mayor **problema** de este funcionamiento es que **debe diseñarse la estrategia de sharding** (el algoritmo o la fórmula que determina qué registro va en qué shard) teniendo en cuenta cuáles van a ser las **necesidades de acceso/consulta** a los datos.
- ✓ **Replicación**: Hacer **copias de los datos en distintos nodos** del clúster, para una **mayor disponibilidad** (se lee desde más nodos) **y** conseguir la **tolerancia a fallos**.
  Existen dos estrategias distintas a seguir para implementar el mecanismo de replicación:
  - **Maestro-esclavo (*master-slave*):**
    - Todas las **escrituras** (inserciones, actualizaciones y borrados) se realizan en el **nodo maestro y después son replicadas** a los nodos esclavo.
    - El **maestro es el punto único de fallo**.
    - Las **lecturas** pueden hacer **en cualquier nodo** (por lo general desde los esclavos para liberar de carga al nodo maestro).
    - Está indicado para trabajos de **lectura intensiva**, ya que la capacidad de lectura se puede escalar en horizontal añadiendo nodos esclavo.
    - **No está indicado para** trabajos de **escritura intensiva** porque todas ellas se realizan en el (único) nodo maestro.
    - No está indicado para casos en los que la consistencia de las lecturas sea una necesidad, ya que se puede realizar una lectura en un nodo esclavo antes de que le llegue un dato ya escrito en el maestro.
  - **Par-a-par (*peer-to-peer*):**
    - No hay un nodo maestro sino que **todos** (*peers*) **están al mismo nivel jerárquico**.
    - Al no haber maestro, **no hay un punto único de fallo**.
    - Se **puede escribir y leer en todos los nodos**.
    - Cada vez que **se escribe en uno la escritura se replica posteriormente** en los demás.
    - **Puede producir inconsistencias de escritura** si se modifica a la vez un mismo dato en distintos nodos.
    - Puede a su vez emplear **dos estrategias para gestionar la concurrencia**:
      - **Pesimista**: Emplea bloqueos para **asegurar que** un registro **no se puede modificar a la vez en dos nodos distintos**.
        Disminuye la disponibilidad durante el tiempo que tarda en levantarse el bloqueo.
      - **Optimista**: **No hay bloqueos**, permitiendo inconsistencias durante el tiempo que las escrituras tardan en propagarse.
        La **base de datos** se mantiene **en todo momento disponible**.
- ✓ **Sharding con replicación**: **Combinar** las dos características anteriores **para obtener** las **ventajas** de ambas (cada shard está a su vez replicado).

Ya que la replicación puede emplear dos distintas estrategias para gestionar la concurrencia, aparecen por lo tanto dos posibilidades al añadir *sharding*.

- ***Sharding* con replicación maestro-esclavo:** Para **cada *shard* habrá un maestro y** determinado **n**úmero de **esclavos**. El nodo que contiene un **maestro** de *shard* es el **punto único de fallo** para ese *shard*. Las **escrituras** que afectan a un *shard* se realizan **en su maestro y** después **se propagan** a los esclavos.
- ***Sharding* con replicación par-a-par:** Para **cada *shard*** hay **varias réplicas al mismo nivel** de jerarquía distribuidas por el clúster. Al no haber maestros **no** hay ningún **punto único** de fallo. Las **lecturas y escrituras** que afectan a un *shard* pueden realizarse **en cualquiera** de sus réplicas (lo cual puede producir los mismos problemas que ya vimos al hablar de la replicación de forma individual).

La gran mayoría de las bases de datos ***NoSQL*** caen dentro de **4 familias** principales:

✓ **Documentales:** Guardan **parejas clave-documento**, donde el documento es texto codificado **típicamente** en formato **JSON y** tiene una **estructura arbitraria**.

Una base de datos documental es aquella en la que para una determinada clave se **almacena un documento completo** (típicamente en formato *JSON*) **sin** tener que **cumplir con** ningún **esquema** determinado de antemano (es decir se trata de bases de datos ***schema-on-read***).

Del mismo modo que **no** se utilizan esquemas en el momento de escritura, tampoco existe el concepto de **tabla**. Sin embargo existe el **concepto de colección**, dentro de cada una de las cuales **podemos almacenar documentos de tipo similar**, supliendo así la función organizativa de las tablas.

Principales ventajas:

- Al ser ***schema-on-read*** no se necesita saber cómo serán los datos desde un principio.
- Al permitir **heterogeneidad** en la estructura de los **documentos proporcionan** gran **flexibilidad**.
- **Funcionan muy bien con *sharding* y replicación**, consiguiendo con ello escalabilidad, tolerancia a fallos y alta disponibilidad.
- **Permiten crear índices** sobre secciones de los documentos que forman parte de cada colección, gracias a lo cual pueden acelerarse las búsquedas dentro de las mismas.
- Al poder guardar diversa información en cada documento sobre un mismo tipo de entidad (información denormalizada) **se evita** en muchos casos la necesidad de hacer **el equivalente a un *JOIN* de *SQL***.
- **Según** la base de datos, su **configuración** y su uso, **pueden** llegar a emplearse para operaciones que necesiten **cumplir con *ACID***. Por ejemplo, *MongoDB* permite transacciones *ACID* cuando se afecta **a un único documento** (no soportando transacciones *ACID* multidocumento).

Principales inconvenientes:

- No son totalmente *ACID*, **limitá**ndose **a** cumplir con ***BASE***.
- Al no ser relacionales y no estar definido un esquema para las colecciones, **se pierde** la capacidad de realizar ***JOIN*** como ocurre con *SQL*.
  - Por lo tanto, si se necesita trabajar con datos que residen en distintas colecciones, debe ser la aplicación la que haga varios accesos y después termine de realizar la operación en función de lo que reciba.
  - Esto implica que si bien estas bases de datos son "*schemaless*", sí que **es importante** hacer **una buena planificación inicial** decidiendo qué guardaremos en

los documentos de cada colección. Ello deberá hacerse teniendo en cuenta cuáles serán las consultas más típicas durante el uso de la base de datos.

Ejemplos de bases de datos documentales: MongoDB. Apache CouchDB.

✓ **Clave-Valor:** Guardan **parejas clave-valor**, donde el valor puede ser secuencia conjunto de bytes.

Principales ventajas:

- Al almacenar valores de forma arbitraria se **necesita saber cómo serán los datos** desde un principio y proporcionan gran flexibilidad.
- **Funcionan muy bien con *sharding* y replicación**, consiguiendo con ello escalabilidad, tolerancia a fallos y alta disponibilidad.

Principales inconvenientes:

- Al no ser relacionales y no estar definido un esquema para el contenido de los valores, **se pierde** la capacidad de realizar ***JOIN*** como ocurre con *SQL*.
- Al ser **agnósticas acerca de los valores almacenados** no permiten ningún tipo de indexación para hacer **búsquedas** sino que todo acceso se hace **según clave.** Para solventar esto en cierta medida, algunas bases de datos clave-valor permiten seleccionar por clave según expresiones regulares.

Ejemplos de bases de datos clave-valor: Redis. Memcached. MemcacheDB. Berkeley DB. Voldemort.

✓ **Columnares: Almacenan la información por columnas en lugar de por filas** (inspiradas en BigTable de Google).

El acceso también es por columnas.

Principales ventajas:

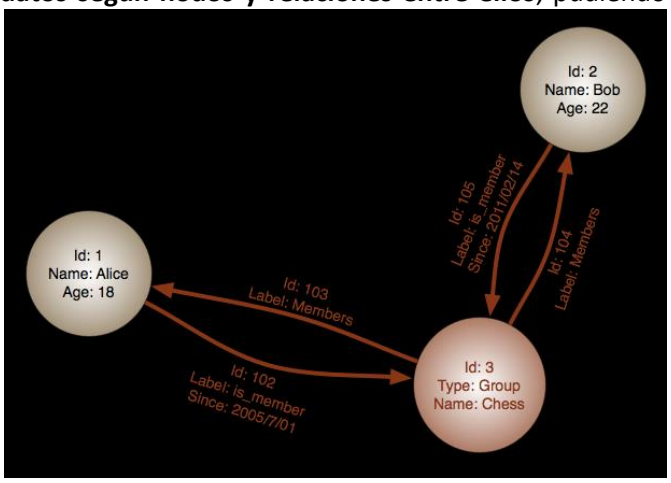
- Debido al modo en el que almacenan la información se **ahorra mucho espacio** en casos en los que hay muchos **campos vacíos**.
- **Pueden escalar para manejar grandes volúmenes** de datos.
- Al poder acceder sólo a las columnas que sea necesario sin tener que visitar registros/filas completas, **pueden realizar lecturas muy rápidas**.
- Son una **opción válida para soportar datos en sistemas tipo OLAP**, los cuales se ven beneficiados por el hecho de poder acceder a la información por columnas.

Principales inconvenientes:

- **No son muy eficientes para accesos a nivel de fila**, lo que a su vez implica que **no son** una buena decisión para **OLTP**.
- No son muy eficientes si se quieren ir añadiendo **datos sobre la marcha** (para modificar una celda puede ser necesario escanear una columna completa).

Ejemplos de bases de datos columnares: Google BigTable. HBase. Apache Cassandra.

- ✓ **Orientadas a grafo: Guardan los datos según nodos y relaciones entre ellos**, pudiendo haber **propiedades** tanto en unos como en otras.



Dado que la información está representada en grafos, estas bases de datos **utilizan lenguajes específicos** para poder recorrerlos en busca de la información que sea necesario acceder o almacenar.

Principales ventajas:

- **Funcionan muy bien para problemas** cuyos datos se organizan en base a **complejas relaciones**, como por ejemplo:
  - **Motores** basados en **reglas.**
  - Sistemas que requieran **analizar rápidamente estructuras en forma de red** (como por ejemplo las redes sociales).
- Pueden soportar transacciones **ACID**.

Principales inconvenientes:

- La propia naturaleza de los grafos **dificulta** (y en muchos casos imposibilita) el poder **particionar**los en varios **nodos de clúster,** por lo que el grafo más grande que se puede tratar debe caber en el almacenamiento o memoria de un único nodo.
- Los mecanismos de recorrido del grafos son **ineficientes para problemas** en los que **no hay relaciones complejas**.

Ejemplos de bases de datos orientadas a grafo: AllegroGraph. Neo4j.

RECOMENDABLE REVISAR TAREA TEMA 2 SOBRE MONGODB.