Big data frameworks

Frameworks para el procesamiento distribuido de datos masivos

Francesc Julbe

PID_00237567

Tiempo mínimo previsto de lectura y comprensión: **4 horas**

UOC
Universitat Oberta de Catalunya

Índice

Introducción

La generación de grandes volúmenes de datos es una problemática que se ha acentuado en los últimos años. La gran mayoría de dispositivos electrónicos, servidores de aplicaciones y prácticamente cualquier proceso que forme parte de una cadena de tratamiento de datos generan datos intermedios y finales que, sin un mecanismo de adecuado, quedan almacenados en algún repositorio final sin que ningún usuario los utilice jamás. Asimismo, las fuentes de generación de datos son diversas y sus formatos muy heterogéneos, añadiendo complejidad a su procesado y a la extracción final de conocimiento útil de los mismos. En función de los requerimientos concretos de cada escenario que requiera procesado de datos pueden utilizarse diversas soluciones y metodologías para dicho procesado. Estas soluciones pueden englobarse principalmente en los siguientes grupos:

- Procesado en *batch* o *offline*
- Procesado en *stream* o (casi) en tiempo-real
- Procesado orientado a eventos

En este módulo cubriremos en qué consisten estos modelos de procesado y qué soluciones existen para cada uno de ellos. Sin embargo el ecosistema *big data* está en un momento muy dinámico y volátil, y aparecen nuevas soluciones que aportan interesantes funcionalidades pero que todavía están en fase de desarrollo, por lo que es difícil saber cuáles de estas soluciones se estabilizarán a medio plazo, cuáles se integrarán en un *framework* de más alto nivel (distribuciones completas *big data*, como Cloudera o HortonWorks) y cuáles no encontrarán su camino. Sin embargo, dos grandes pilares vertebrarán el contenido del módulo, como son el *framework* Hadoop y su ecosistema de servicios y aplicaciones; y Apache Spark, actualmente el *framework* con mayor aceptación para el procesado distribuido de datos. Finalmente, se introducirán algunas soluciones existentes en el mercado de los principales fabricantes del sector de la computación.

Objetivos

En los materiales didácticos de este módulo encontraremos las herramientas indispensables para alcanzar los siguientes objetivos:

1. Comprender cuáles son los diferentes modelos de procesado distribuido de datos utilizados en *big data* y en qué escenario es útil cada uno de ellos.

2. Conocer los principales *frameworks* existentes para procesado distribuido como son Hadoop (y su ecosistema) y Apache Spark.

3. Tener una primera introducción a las diversas herramientas y servicios existentes para el procesado distribuido de datos.

4. Conocer cuáles son las distribuciones y soluciones existentes en el mercado que hacen uso de estas herramientas.

1. Modelos de procesamiento distribuido en *big data*

Entendemos por **procesado distribuido de datos** la capacidad de compartir recursos (recursos de computación, almacenamiento, memoria) entre una red de ordenadores para la realización de una serie de operaciones y/o tareas.

Los equipos que componen la red de procesado de datos distribuidos pueden estar situados en diferentes ubicaciones, pero conectadas entre sí. Al disponer de una red de ordenadores destinados a un propósito o tarea que realizar, el rendimiento es fácilmente escalable mediante la adición de otro ordenador a una red.

El procesamiento de datos distribuidos trabaja en este principio y sostiene que un trabajo se hace más rápido si varias máquinas están trabajando en paralelo y de forma síncrona. Por ejemplo, problemas estadísticos complejos pueden dividirse en módulos y asignarse a diferentes máquinas donde se procesan simultáneamente. Esto reduce significativamente el tiempo de procesamiento y mejora el rendimiento. No obstante, la tarea que hay que realizar de forma paralela debe poder ejecutarse de forma paralela. Más adelante veremos ejemplos de este concepto.

Esta arquitectura es muy flexible ya que pueden añadirse o quitarse computadoras a la red (comúnmente llamadas **nodos**) sin que tenga un impacto significativo en su funcionamiento. Dichos nodos pueden estar cerca o lejos geográficamente, solo es necesario que estén conectados entre sí (mediante una IP). El procesado distribuido de datos puede contribuir a la reducción de costes, ya que la carga de computación se distribuye entre los diferentes equipos de la red, en contraste a la adquisición de grandes equipos *mainframe* o *datawarehouse* mucho más caros y monolíticos. Además, si la red está adecuadamente configurada, el procesamiento de datos distribuido es fiable, ya que la carga de computación puede balancearse entre nodos. Además los datos estarán almacenados y replicados entre nodos, lo que permite tolerancia a fallos de algunos nodos (también configurable) sin comprometer el resultado de una tarea concreta. Existen diferentes modos para realizar procesado distribuido de datos. En este apartado describiremos en qué consisten dichos modos y cuáles son sus usos.

Equipos *mainframe*

Los equipos *mainframe* son grandes equipos de cómputo, rápidos y con un coste elevado, utilizados principalmente por grandes empresas u organizaciones.

Datawarehouse

Datawarehouse es un sistema que permite el almacenamiento de información de forma homogénea y fiable, orientado a la consulta de datos y a su tratamiento de forma jerarquizada.

1.1. Procesamiento en *batch*

El **procesado de datos en modo *batch*** es un modo de procesar grandes volúmenes de datos en el que una serie de procesos (lo que puede llamarse transacción) se ejecutan sobre un conjunto de datos. Durante la ejecución de un proceso en *batch* el usuario rara vez tiene que interactuar con el proceso. La ejecución va a requerir unos *inputs* y típicamente producirá un *output* una vez haya finalizado el proceso. Su duración puede ser muy variable, desde minutos hasta horas.

El modo de procesado *batch* es el más ampliamente utilizado ya que son pocos los procesos empresariales de criticidad suficiente que requieran procesado de datos en tiempo real. No obstante, dichos procesos suelen ser críticos como se describirá en apartados posteriores. Puede decirse que el procesamiento de datos en modo *batch* es el caso general de procesado más que un caso especial.

Un proceso en modo *batch* se caracteriza por lo siguiente:

- Tiene acceso a todos los datos o a una gran cantidad de ellos.
- Suele realizar operaciones grandes y complejas.
- Es un proceso más enfocado al *throughput* que al propio tiempo de ejecución (o latencia).
- Su latencia suele medirse en minutos u horas. Típicamente en centros de supercomputación los tiempos de cálculo asignados a una tarea concreta (llamados *Jobs*) no suelen exceder las 48 o 72 horas.

Throughput

Throughput es el volumen de datos procesados por unidad de tiempo.

Suelen identificarse tres factores clave en un problema de *big data*: **volumen**, **velocidad** y **variedad** (las 3 V del *big data*). Si en el problema de cálculo al que nos enfrentamos el volumen de datos es el factor clave, el modo de procesado adecuado es el modo *batch*.

Pueden encontrarse muchos ejemplos de procesado de datos en batch, a continuación vamos a exponer algunos de ellos.

Proceso ETL (*Extract*, *Transform*, *Load*) sobre un archivo astronómico

Se nos plantea la necesidad de realizar la conversión de datos de un archivo astronómico con 2.000.000.000 estrellas y otros objetos celestes, lo que supone un archivo del orden de 3 TB de datos. La necesidad es convertirlo de su formato actual, ficheros de texto (ASCII) con la información organizada en columnas, a un formato más adecuado para su procesado por entornos de procesado distribuido, como podría ser Parquet.

El proceso va a leer todos los datos disponibles en el archivo *(Extraction)*, va a procesar cada una de las estrellas, extrayendo aquellos campos que nos interesen (posición, brillo, tipo de estrella u objeto -planetas, galaxias, etc.-) *(Transform)* y va a convertirla en un ítem para ser almacenado en el archivo en formato Parquet *(Load)*.

Un proceso de estas características se lanzará una sola vez en el clúster, entrará en la cola de procesado y no terminará hasta que toda la conversión se haya realizado. La interactividad con el usuario es inexistente y a partir de un *input* específico (archivo astronómico en ficheros con formato ASCII), generará un *output* también concreto (archivo en otro formato de datos) tras un tiempo de procesado más o menos largo. No hay una fuente dinámica de generación de estrellas, con lo que el contenido del archivo es "estático" y se procesa de una vez.

Una operación de estas características es relativamente sencilla de realizar utilizando el *framework* Hadoop y Spark, herramientas de procesado *batch* que se introducirán más adelante.

Análisis de *logs* de servidores web

Un *log* de Apache es un fichero en el cual un servidor web va anotando las incidencias que se producen durante el acceso a sus contenidos. Las siguientes líneas serian un ejemplo de uno de ellos:

```
64.242.88.10 [07/Mar/2014:16:05:49 -0800] "GET /main/script?
topic=Main.Config HTTP/1.1" 401 12846

64.242.88.10 [07/Mar/2014:16:06:51 -0800] "GET /main/run?
rev1=1.3&rev2=1.2 HTTP/1.1" 200 4523

64.242.88.10 [07/Mar/2014:16:10:02 -0800] "GET /main/hsdiv HTTP/1.1"
200 6291

64.242.88.10 [07/Mar/2014:16:11:58 -0800] "GET /main//WikiSyntax
HTTP/1.1" 200 7352

64.242.88.10 [07/Mar/2014:16:20:55 -0800] "GET /main//Dkn HTTP/1.1"
200 5253

64.242.88.10 [07/Mar/2014:16:23:12 -0800] "GET /main/FileSys-
tem?temp=oopsmore HTTP/1.1" 200 11382

64.242.88.10 [07/Mar/2014:16:23:12 -0800] "GET /main/FileSys-
tem?temp=oopsmore HTTP/1.1" 200 11382

10.10.10.10 [07/Mar/2014: 16:23:12 -0800] "GET / webmin HTTP/1.1" 404
726

10.10.10.10 [07/Mar/2014: 16:23:13 -0800] "GET /admin HTTP/1.1" 404
726

10.10.10.10 [07/Mar/2014: 16:23:14 -0800] "GET /login HTTP/1.1" 404
726

64.242.88.10 [07/Mar/2014:16:24:16 -0800] "GET /main/Mathew HTTP/1.1"
200 4924

64.242.88.10 [07/Mar/2014:16:29:16 -0800] "GET /main/Checks?
topic=Main.Config HTTP/1.1" 401 12851
```

Así, nos da información interesante como la IP que se intenta conectar, la fecha y hora a la que lo hace, la operación que trata de realizar (*GET* en el ejemplo), la página visitada, código de funcionamiento de la operación específica (200 si la página se ha visitado

correctamente, 404 si la página no existía, 403 si no tenía permisos de acceso, etc.) y el volumen de datos descargado de dicha página.

Como podemos adivinar, a pesar de ser información muy simple, puede resultar muy útil para la empresa que gestiona dicho servidor, ya que de ella se pueden extraer información interesante y simple como la página más visitada y la hora a la que hay más visitas También se puede hacer el seguimiento de IP sospechosas y asociadas a prácticas de *malware* y *spam*. Específicamente, dependiendo del tipo de llamada realizada *GET* y del recurso o página solicitada (por ejemplo páginas de administración *admin*), se podrían inferir si se producen ataques a nuestro servidor y agruparlos por IP, de modo que podrían bloquearse dichas IP. Como se puede observar en el *log* de ejemplo, hay unas llamadas desde la IP 10.10.10.10 que merecerían este tipo de análisis ya que aunque no ha realizado nada ilegal, parece estar tratando de acceder a recursos propios de administración sin éxito (código 404).

El problema de procesado que se plantea en este caso es el elevado tamaño que estos *logs* pueden adquirir. Si nuestro servidor es visitado con asiduidad, algo que todas las empresas persiguen, el tamaño de este *log* fácilmente adquirirá el orden del GB en pocos días. Además, los *logs* suelen almacenarse un tiempo antes de ser definitivamente borrados. Por tanto, realizar una operación de detección de ataques a nuestro sistema va a requerir la programación de una tarea *batch* para que explore todos los archivos de *log* de que se dispone e identifique patrones en busca de dichos ataques. Esta operación se realizaría sin interactividad por parte del usuario mientras se ejecuta y sin una modificación de los archivos de *log* durante su tiempo de ejecución, conociendo el resultado cuando la tarea haya finalizado, por lo que constituye una tarea en modo *batch*.

1.2. Procesamiento en tiempo real

Aunque muchos de los escenarios de procesado de datos utilizan el modo *batch*, hay ocasiones en que aparece la necesidad de procesar datos en *streaming* o (casi) en tiempo real (*real-time*). No es útil identificar a un posible comprador cuando este ha abandonado el sitio de e-Commerce, o bien, detectar una intrusión después de que el *hacker* ya ha desaparecido. El procesado en *streaming* también es útil en la ILP, la detección de *spam*, la estimación de tráfico, etc.

ILP

Information Leak Prevention o prevención de fuga de información es el mecanismo y políticas que evitan que información de carácter confidencial de una organización pueda ser accesible a personas ajenas a esta.

Las deficiencias e inconvenientes de procesamiento de datos en *batch* fueron ampliamente identificados por la comunidad de *big data* y se hizo evidente que el procesamiento en casi tiempo real o *streaming* es una necesidad en muchas aplicaciones prácticas. En los últimos años algunas soluciones han aparecido para cubrir esta necesidad, como Apache Storm de Twitter, Spark Streaming, Apache Flink o S4 de Yahoo, algunas de las cuales hablaremos más adelante.

Este modo de procesamiento permite tomar decisión y acciones inmediatas en aquellas situaciones en que es cuestión de segundos o minutos la toma de una decisión. Por tanto, de las 3 V del *big data* ya mencionadas, el modo de procesado de datos en *streaming* se focaliza en la V de velocidad, frente a la V de volumen del modo *batch*.

Es importante tener en cuenta que aunque se mencione el concepto "tiempo real", se produce un abuso del lenguaje, ya que en el mundo de la computación, tiempo real se asocia a procesos y eventos causa-efecto que se mueven en el orden de los mili-segundos o micro-segundos, mientras que en los escenarios de *big data*, la respuesta se moverá en el orden de los segundos o minutos como mucho, de ahí que se utilice el término "casi" cuando hablamos de tiempo real en el procesado de datos en *streaming*.

Podríamos caracterizar el procesado en *streaming* de la siguiente forma:

- Realiza un cálculo o proceso sobre una porción de los datos o una ventana pequeña de datos recientes.
- El cálculo es relativamente simple, en caso contrario perdería velocidad.
- Necesita completar cada cálculo en "casi" tiempo real, probablemente segundos a lo sumo.
- Los cálculos son generalmente independientes y no forman parte de una cadena de procesado compleja.
- Es asíncrono, es decir, la fuente de datos no interactúa con el proceso y no hay señales de sincronización entre ellos.

A continuación vamos a describir algunos.

Detección de un terremoto procesando datos en tiempo real de Twitter

El objetivo del estudio fue detectar si se estaba produciendo un terremoto en algún lugar del planeta utilizando datos y mensajes de Twitter. El uso de tecnología *streaming* en este caso está plenamente justificado ya que un terremoto no tiene una duración de más de unos segundos.

A grandes rasgos, inicialmente se crea un stream de datos proveniente de Twitter, una llamada del estilo:

```
TwitterUtils.createStream(...)
.filter(_.getText.contains('earthquake') ||
_.getText.contains('shaking'))
```

Este filtro descartaría aquellos tuits que no contengan la palabra "earthquake" ni "shaking". No obstante, no todos los tuits que contengan esas palabras implican que se esté produciendo un terremoto en el lugar en el que la persona lo escribe, de modo que deberíamos proceder a clasificar los mensajes entre aquellos que escribiría alguien que está sufriendo un terremoto y aquellos que no. En este ejemplo, los tuits del tipo "Earthquake!" o "Now it is shaking" los asumiríamos como verdaderos o identificaciones positiva de un terremoto en ese momento en la ubicación de la persona que escribe el tuit, mientras que "Attending an Earthquake Conference" o "Yesterday there was an earthquake" serian identificaciones negativas de un terremoto.

Existen muchos clasificadores de información. En el artículo se utilizó un clasificador llamado Support Vector Machines (SVM) y su función principal es poder etiquetar una serie de datos como "verdadero" o "falso" dependiendo de un modelo previamente entrenado. Sin entrar en detalles, ya que no es el objetivo de esta asignatura, entrenar un modelo significa identificar qué tipo de respuestas esperamos como verdaderas y cuáles como

Lectura recomendada

El experimento de detección de un terremoto procesando datos en tiempo real de Twitter se describe en http://www.ymatsuo.com/papers/www2010.pdf Aunque el artículo es relativamente antiguo, constituye un buen ejemplo del uso del procesado de datos en *streaming*.

falsas. Tras haber entrenado a nuestro clasificador con una serie de *inputs* de estas características (true->'Earthqueake', true -> 'Now is shaking', false-> 'Attending an Earthquake Conference', false -> 'Yesterday there was an earthquake'), nuestro sistema de procesado en *streaming* estaría en disposición de escuchar el stream de datos proveniente de Twitter y detectar positivos casi en tiempo real y tomar las acciones que fueran necesarias.

Monitorización de una red de sensores en un entorno de IoT *(Internet of Things)*

Otro escenario menos específico que el anterior de procesado en *streaming* sería un sistema que monitorea a tiempo real una nube de sensores que informan de parámetros relevantes de un dispositivo (coches, *smartphones*, sensores en aviones, etc.) y permite, a partir de realizar análisis de datos en tiempo real, tomar ciertas decisiones, como por ejemplo informar de mejores rutas de tráfico debido a congestiones o accidentes.

El Dreamliner es un avión muy avanzado de Boeing que dispone de una compleja red de sensores que monitorizan gran cantidad de parámetros del funcionamiento de avión. Los sensores están conectados de forma inalámbrica a un procesador central de datos. Como ejemplo, tomemos el sistema de Active Gust Alleviation, lo que podría traducirse como sistema de "atenuación de ráfagas de viento". Este sistema usa sensores para medir la turbulencia en el morro de avión y ajusta instantáneamente los *flaps* de las alas para contrarrestarlo.

Análisis de *logs* de servidores web

Tal y como hemos visto en el subapartado anterior, el modo de procesado en *batch* nos puede ayudar a encontrar IP fraudulentas y modelizar un comportamiento sospechoso (páginas que intenta visitar, operaciones que trata de ejecutar, etc.). Una vez hayamos definido un modelo de comportamiento sospechoso, podemos poner en marcha un sistema de procesado en *streaming* que esté analizando los *logs* que se generan en tiempo real, tratando de identificar comportamientos no deseados de forma inmediata y lanzar alertas, lo que mejoraría la respuesta a dichos ataques.

1.3. Procesado de datos en base a eventos complejos

El procesamiento en base a eventos complejos (o *Complex Event Processing*, CEP) puede entenderse como un subconjunto del procesado de datos en *streaming* y consiste en la capacidad de predecir eventos de alto nivel que pueden derivarse de un conjunto específico de factores de bajo nivel. CEP identifica y analiza las relaciones de causa y efecto entre los eventos en casi tiempo real, permitiendo la toma de decisiones y medidas eficaces en respuesta a situaciones específicas.

CEP es un paradigma en evolución concebido originalmente en la década de 1990 por el Dr. David Luckham en la Universidad de Stanford y en su origen estaba muy focalizado en el mercado de valores. CEP se utiliza en la gestión de la política de seguridad de riesgos, la gestión de relaciones con clientes (en los llamados CRM), servidores de aplicaciones y especialmente en situaciones que implican numerosos factores que interactúan de forma variable, tales como la inversión y los entornos de préstamo para las instituciones financieras, o en la gestión de amenazas para redes de comunicaciones. La idea que subyace en una aplicación CEP es el análisis de diferentes eventos para encontrar patrones entre ellos. Aunque los términos pueden confundirse, podría decirse que un sistema CEP se construye generalmente encima de un sistema de procesado

en *streaming*. Las consultas de los sistemas *streaming* son de más bajo nivel y pueden operar en ventanas de tiempo corto, mientras que las consultas de los sistemas CEP pueden ser más complejas y tomar periodos de tiempo más largo.

Monitorización de temperatura y sistema de alarma en un *datacenter*

Tomemos el caso de un *datacenter* con sus *racks* de servidores. Para cada servidor se monitoriza la temperatura y el consumo de CPU en intervalos de tiempo definidos. En cada uno de estos intervalos, se genera un evento formado por una medida de temperatura y consumo de CPU. Un sistema podría monitorizar dichos eventos y, en el caso de que en varias medidas de temperatura se superase un límite establecido, se generaría una alerta, lo que obligaría a algún sistema de protección a tomar medidas como limitar la carga de CPU en dicho servidor.

Los patrones que se han de detectar sobre los eventos podrían ser mucho más complejos, como también establecer correlaciones entre las medidas de temperatura de los servidores cercanos al servidor que se está sobrecalentando.

Por tanto, CEP puede entenderse como un sistema en *streaming* en el cual se le añade una capa de abstracción en la que los datos entrantes se modelan como datos con "significado" para el usuario, los eventos.

Existen algunas herramientas para desarrollar aplicaciones CEP como Esper, aunque herramientas de procesado en *streaming*, de las que hablaremos más adelante, podrían implementar un sistema CEP.

1.4. Resumen

A modo de resumen, podemos decir que:

- El procesado de datos en *batch* es muy eficiente en el tratamiento de grandes volúmenes de datos. Los datos se introducen en el sistema y se procesan produciendo unos *outputs*. Aquí el tiempo invertido en el procesado no es un problema prioritario. Los trabajos están configurados para ejecutarse sin intervención manual y dependiendo del tamaño de los datos que se procesan y la potencia de cálculo del sistema, la salida se puede retrasar de manera significativa.

- El procesado de datos en *stream* se focaliza en la velocidad de lectura y el análisis de los datos. Estos deben ser procesados dentro del período de tiempo pequeño o casi en tiempo real. El procesado en *streaming* permite la toma de decisiones rápidas basadas en tendencias que se desarrollan en ventanas de tiempo pequeñas.

- El procesamiento de eventos complejos se ocupa de eventos discretos y puede entenderse como un caso concreto dentro del procesado de datos en *streaming*.

2. Procesado distribuido en modo *batch*

En la llamada era del *big data*, el volumen de datos disponible para analizar con el objetivo de realizar cualquier tipo de tarea y/o consulta ha superado con creces la capacidad de procesamiento de una sola máquina. Cuando tratamos con grandes volúmenes de datos, se nos plantean dos interrogantes:

- ¿Cómo almacenamos y accedemos a esos datos?
- ¿Cómo analizamos esos datos?

Para dar respuesta a dichos interrogantes, Hadoop aparece en el mercado ofreciendo una forma de almacenar y procesar estos datos en un entorno heterogéneo y altamente distribuido.

2.1. Hadoop framework

En 2003, Google se enfrentó al desafío de indexar toda la World Wide Web, lo que obviamente planteó muchas dificultades. Por un lado, la Web crecía a un ritmo muy elevado y por otro lado, ya habían alcanzado el límite de escalabilidad de sus soluciones relacionales (*Relational Database Management System* RDBM). Así, Google buscó y encontró un nuevo enfoque para poder procesar y analizar grandes cantidades de datos. Dicho enfoque se basaba en dos pilares:

- **GFS** (Google File System): es un sistema de ficheros distribuido entre todos los nodos de un clúster. Este sistema aportaba varias funcionalidades:
  - Los datos se analizan localmente, allí donde están almacenados.
  - Los datos están replicados a través de los nodos del clúster, lo que da tolerancia a fallos y además abstrae la complejidad de la infraestructura para el desarrollador con un punto de entrada único al sistema de ficheros aunque estos estén distribuidos.
- **MapReduce**: es un paradigma de computación distribuida en la cual una tarea se descompone en pequeñas subtareas, independientes entre sí que se ejecutan en los nodos del clúster, para ser agregadas en un proceso posterior una vez hayan finalizado todas las subtareas.

Esta arquitectura estaba enfocada a sacar partido de las arquitecturas más predominantes en la época: RAM limitada y grandes capacidades de disco. En la nueva arquitectura, el programador solo debía ocuparse de la lógica del procesado, mientras que el propio *framework* realiza las funciones de gestión de la

infraestructura que soporta dicha lógica. Apache Hadoop fue la primera iteración *open source* del nuevo enfoque introducido por Google, centrándose en tres grandes prioridades:

- **Fiable y tolerante a fallos**: existe replicación de datos entre nodos y la gestión de réplicas es transparente al desarrollador.

- **Económico**: el *framework* no requiere de grandes inversiones en costosos equipos, ya que puede ejecutarse en hardware más estándar, llamado *commodity hardware*.

- **Escalable**: el clúster permite añadir nuevos nodos de forma sencilla pudiendo llegar a miles de nodos.

Commodity hardware

Los *commodity hardware* son dispositivos estándar como ordenadores de sobremesa o estaciones de trabajo.

El *framework* Hadoop está formado por un conjunto de servicios destinados a ofrecer funcionalidades de procesado distribuido sobre un clúster formado por nodos, que comparten sus recursos de computación (CPU), memoria y almacenaje.

Los proyectos sobre los que trabaja Hadoop y forman su esqueleto son:

- **Hadoop Common**: conjunto de librarías sobre las que se basa el procesado distribuido, utilidades y otras herramientas.

- **Hadoop Distributed File System (HDFS)**: servicio de almacenaje de datos compartido entre los nodos del clúster.

- **Hadoop YARN**: componente que se ocupa de la gestión de recursos en el clúster

- **Hadoop MapReduce**: modelo de procesado de datos en entorno distribuido.

El *framework* Hadoop ofrece servicios clave sobre los que operan otros servicios y herramientas *big data*. Algunas de estas herramientas están en desarrollo, otras están en producción y otras ya han quedado obsoletas. En los siguientes subapartados vamos a presentar las herramientas más importantes, que forman el llamado ecosistema Hadoop.

2.1.1. Paradigma MapReduce

El procesado propuesto dentro del *framework* de Hadoop es el llamado MapReduce, propuesto por Google.

Abstracción

Como se puede deducir por su nombre, el procesado se realiza en dos tareas, el **Map** y el **Reduce**:

- **Map**: esta tarea es la encargada de "etiquetar" o "clasificar" los datos que se leen desde disco, típicamente de HDFS, en función del procesado que estemos realizando.

- **Reduce**: esta tarea es la responsable de agregar los datos etiquetados por la tarea Map. Puede dividirse en dos etapas, la `shuffle` y el propio `reduce` o agregado.

Todo el intercambio de datos entre tareas utiliza estructuras llamadas parejas <clave,valor> (o <*key*,*value*> en inglés) o tuplas. En el siguiente ejemplo vamos a mostrar, desde un punto de vista funcional, en qué consiste una tarea MapReduce.

Ejemplo de tarea MapReduce

Imaginemos que tenemos tres ficheros con los siguientes datos:

Fichero 1:

- Carlos, 31 años, Barcelona
- María, 33 años, Madrid
- Carmen, 26 años, Coruña

Fichero 2:

- Juan, 12 años, Barcelona
- Carmen, 35 años, Madrid
- José, 42 años, Barcelona

Fichero 3:

- María, 78 años, Sevilla
- Juan, 50 años, Barcelona
- Sergio, 33 años, Madrid

Dados estos ficheros, podríamos preguntarnos cuántas personas hay en cada ciudad. Para responder esa pregunta definiremos una tarea `mapper` que leerá las filas de cada fichero y las "etiquetará" cada una en función de la ciudad que aparece, que es el tercer campo de cada línea: Nombre, edad, Ciudad. Para cada línea, nos devolverá una tupla de la forma: <ciudad, cantidad>. Al final de la ejecución de la tarea `mapper` en cada fichero tendremos:

- Fichero 1: (Barcelona, 1), (Madrid, 1), (Coruña, 1)
- Fichero 2: (Barcelona, 1), (Madrid, 1), (Barcelona, 1)
- Fichero 3: (Sevilla, 1), (Barcelona, 1), (Madrid, 1)

Como vemos, nuestras tuplas están formadas por una clave (*key*), que es el nombre de la ciudad, y un valor (*value*), que es el número de veces que aparece en la línea, que siempre es 1. La tarea `reducer` se ocupará de agrupar los resultados según el valor de la clave. Así, recorrerá todas las tuplas agregando los resultados por misma clave y devolviéndonos:

- (Barcelona, 4)
- (Sevilla,1)
- (Madrid, 3)
- (Coruña, 1)

Obviamente, en este ejemplo la operación no requería un entorno distribuido. Sin embargo, en un entorno con millones de registros de personas una operación de estas características sería muy efectiva.

Conectando con lo descrito en subapartados anteriores, una operación MapReduce es un procesado en modo *batch*, ya que recorre de una vez todos los datos disponibles y no devuelve resultados hasta que ha finalizado.

Es importante tener en cuenta que las funciones de agregación (`reducer`) deben ser conmutativas y asociativas.

Propiedad asociativa

Una operación asociativa cumple la propiedad (*ab*) *c* es igual a c (*ba*). Una operación no asociativa es, por ejemplo, la resta, donde (5 – 3) – 2 = 0 no es lo mismo que 5 – (3 – 2) = 4.

MapReduce fue un paradigma muy novedoso para poder explorar grandes archivos en entornos distribuidos, sin embargo tiene algunas limitaciones importantes:

- Utiliza un modelo forzado para cierto tipo de aplicaciones, que obliga a crear etapas adicionales MapReduce para ajustar la aplicación al modelo. Puede llegar a emitir valores intermedios extraños o inútiles para el resultado final, e incluso creando funciones `Map` y/o `Reduce` de tipo identidad, es decir, que no son necesarias para la operación que hay que realizar pero que deben crearse para adecuar el cálculo al modelo.

- MapReduce es un proceso *file-based*, lo que significa que el intercambio de datos se realiza utilizando ficheros. Esto produce un elevado flujo de datos en lectura y escritura de disco, afectando a su velocidad y rendimiento general si un procesado concreto está formado por una cadena de procesos MapReduce.

Implementación MapReduce en Hadoop

Desde un punto de vista de proceso en entorno Hadoop, vamos a describir cuáles son los componentes de una tarea MapReduce:

1) Nuestro clúster está formado por varios nodos, controlados por un nodo maestro. Cada nodo almacena ficheros localmente y son accesibles mediante el sistema de ficheros HDFS (típicamente es HDFS, aunque no es un requisito necesario). Los ficheros se distribuyen de forma homogénea en todos nuestros nodos. La ejecución de un programa MapReduce implica la ejecución de las tareas de `map()` en muchos o todos los nodos del clúster. Cada una de estas tareas es equivalente, es decir, no hay tareas `map()` específicas o distintas a las otras. El objetivo es que cualquiera de dichas tareas pueda procesar cualquier fichero que exista en el clúster.

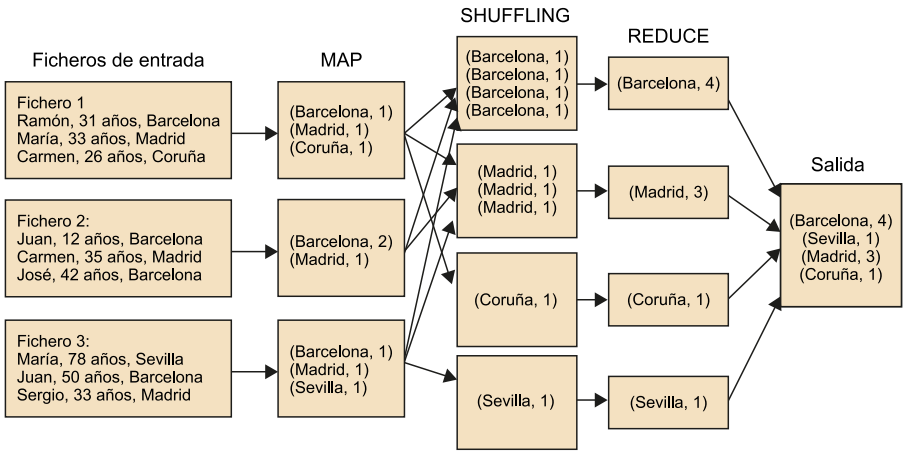
2) Cuando la fase de Map ha finalizado, los resultados intermedios (tuplas <clave, valor>) deben intercambiarse entre las máquinas para enviar todos los valores con la misma clave a un solo `reduce()`. Las tareas `reduce()` también se ejecutan en los mismos nodos que las `map()`, siendo este el único intercambio de información entre tareas (ni las tareas `map()` ni las `reduce()`

intercambian información entre ellas, ni el usuario puede interferir en este proceso de intercambio de información). Este es un componente importante de la fiabilidad de una tarea MapReduce, ya que si un nodo falla, reinicia sus tareas sin que el estado del procesado en otros nodos dependa de él.

Sin embargo, en el proceso de intercambio de información entre Map y Reducer, podemos introducir dos nuevos conceptos: *Partition* (o Partición) y *Shuffle*. Cuando una tarea `map()` ha finalizado en algunos nodos, otros nodos todavía pueden estar realizando tareas `map()`. Sin embargo, el intercambio de datos intermedios ya se inicia y estos son mandados a la tarea `reduce()` correspondiente. El proceso de mover datos intermedios desde los *mapper* a los *reducers* se llama *shuffle*. Se crean subconjuntos de datos, agrupados por "clave", lo que da lugar a las particiones, que son las entradas a las tareas `reduce()`. Todos los valores para una misma clave son agregados o reducidos juntos, indistintamente de cuál era su tarea *mapper*. Así, todos los *mapper* deben ponerse de acuerdo en donde mandar las diferentes piezas de datos intermedios.

Cuando el proceso Reduce ya se ha completado agregando todos los datos, estos son guardados de nuevo en disco. La figura 1 nos muestra de forma gráfica el ejemplo descrito.

Figura 1. Diagrama de flujo con las etapas Map y Reduce del ejemplo descrito



Lectura recomendada

Para más detalles sobre MapReduce puede consultarse el siguiente módulo de Yahoo: https://developer.yahoo.com/hadoop/tutorial/module4.html

Limitaciones de Hadoop

Hadoop es la implementación del paradigma MapReduce que solucionaba la problemática del cálculo distribuido utilizando las arquitecturas predominantes hace una década. Sin embargo, actualmente presenta importantes limitaciones:

- Es complicado aplicar el paradigma MapReduce en muchos casos, ya que una tarea debe descomponerse en subtareas *map* y *reduce*.

- Es lento. Una tarea puede requerir la ejecución de varias etapas MapReduce y, en ese caso, el intercambio de datos entre etapas se llevará a cabo utilizando ficheros, haciendo uso intensivo de lectura y escritura en disco.

- Hadoop es un *framework* esencialmente basado en Java que requiere la descomposición de un procesado en tareas *map* y *reduce*. Sin embargo esta aproximación al procesado de datos resultaba muy dificultosa para el científico de datos sin conocimientos de Java. De modo que diversas herramientas aparecieron para flexibilizar y abstraer al científico del paradigma MapReduce y su programación en Java. Entre estas herramientas podemos citar:
  - **Flume**, para almacenar datos en streaming hacia HDFS.
  - **Sqoop**, para el intercambio de datos entre bases de datos relacionales y HDFS.
  - **Hive** o **Impala**, para realizar consultas tipo SQL sobre datos almacenados en HDFS.
  - **Pig**, para definir cadenas de procesado sobre datos distribuidos.
  - **Giraph**, para análisis de grafos.
  - **Mahout**, para desarrollar tareas de *machine learning*.
  - **HBase**, como sistema de almacenamiento NoSQL.
  - **Oozie**, como gestor de flujos de trabajo.

Así, el científico de datos debe conocer un amplio conjunto de herramientas, cada una con propiedades distintas, lenguajes distintos y muy poco (o ninguna) compatibilidad entre ellas.

2.1.2. Ecosistema Hadoop

El *framework* Hadoop provee unos servicios o herramientas básicas (HDFS, YARN, implementación MapReduce y Hadoop Common) sobre los cuales operan los servicios del ecosistema destinados a realizar una tarea especializada y concreta. Al conjunto de herramientas que operan sobre Hadoop se las suele identificar como herramientas del **ecosistema Hadoop**. Las herramientas que forman parte de dicho ecosistema se pueden clasificar en función de su área funcional. Así, existen herramientas para la gestión de sistemas de ficheros distribuidos, modelos de programación distribuida, bases de datos NoSQL, librerías para aprendizaje automático o *machine learning*, etc.

En esta subapartado vamos a hablar de algunos de los proyectos más relevantes dentro del ecosistema Hadoop y cuáles son sus principales características. Aunque la lista es muy amplia, aquí nos centraremos en aquellos que han tenido más recorrido hasta ahora.

Herramientas para gestión de almacenaje

En muchas situaciones es necesario mover los datos disponibles entre los distintos gestores de almacenamiento de un mismo proyecto.

Sqoop (SQL To Hadoop) es una herramienta cuyo propósito es el intercambio de datos entre bases de datos relacionales (RBDMS) y HDFS. Hace uso del modelo MapReduce para realizar la copia de datos desde una base de datos relacional, pero usando solo tareas Map donde el número de *mappers* para realizar la conversión es configurable. Sqoop analiza el número de filas que hay que importar y divide la tarea entre los diferentes *mappers*, generando código Java para cada tabla que se importa y que puede ser conservado para posteriores usos.

Del mismo modo que los datos pueden importarse al HDFS, estos pueden exportarse a una base de datos desde un repositorio HDFS. La herramienta dispone de una consola para realizar las tareas de importación y exportación del tipo:

```
%> sqoop import -connect jdbc:mysql:...
```

Ofrece compatibilidad con casi cualquier tipo de base de datos mediante una conexión JDBC genérica. No obstante, también dispone del llamado "modo-directo", en el cual el rendimiento es mucho mejor gracias al uso de utilidades propias de cada servidor de base de datos. No obstante, esta modalidad de momento está limitada a MySQL y Postgres y no todas las funcionalidades Sqoop están disponibles en modo directo.

Aunque Sqoop puede ejecutarse en paralelo, cuanto mayor sea el paralelismo mayor será la carga sobre la base de datos, así que dicho paralelismo debe dimensionarse adecuadamente.

Sqoop2

Hasta ahora Sqoop era una herramienta cliente, es decir, el propio cliente se encargaba de conectarse a la base de datos (con usuario y contraseña) y al HDFS a la vez para realizar la transferencia. Así, apareció una mejora con Sqoop2, en la cual se define una nueva arquitectura cliente-servidor. El cliente, esta vez, solo necesita acceso al servidor, de modo que no tiene acceso directo a la base de datos y permite al administrador del sistema configurar Sqoop, limitando los recursos a utilizar, estableciendo una política de seguridad, etc.

Herramientas de exploración y consulta de datos distribuidos

Debido a la complejidad de explorar y consultar datos en entornos distribuidos como HDFS mediante procesos MapReduce, han aparecido herramientas que proporcionan una capa de abstracción que facilita la tarea de acceder a esos datos sin necesidad de programar *mappers* y *reducers* en Java. A continuación presentamos las tres más populares:

1) Apache Hive

Apache Hive, desarrollada por Facebook, es una herramienta que permite realizar consultas utilizando un lenguaje similar a SQL sobre datos distribuidos en HDFS en vez de usar programación de bajo nivel de etapas MapReduce.

SQL-92

SQL-92 es la tercera revisión del estándar SQL (*Structured Query Language*), el lenguaje de programación específico para la gestión de bases de datos relacionales.

El uso de Hive requiere conocimiento de SQL ya que utiliza un lenguaje llamado HiveQL, que es un subconjunto del estándar SQL-92. Las consultas escritas utilizando Hive se traducen en tareas Map/Reduce que se ejecutan en el clúster. Entre sus características principales cabe remarcar que es una herramienta altamente escalable, permitiendo realizar consultas sobre petabytes de datos, ofreciendo una escalabilidad difícilmente igualable. Sin embargo es una herramienta lenta. Al utilizar procesado de datos Map/Reduce adolece de sus defectos, lo que convierte a Hive en una buena herramienta para realizar consultas en modo *batch* sobre (muy) grandes volúmenes de datos. Hive se ejecuta como un proceso servidor sobre el cual se realizan las consultas.

A grandes rasgos, una consulta Hive podría tener el siguiente aspecto:

```
SELECT city, SUM(cost) as TOTAL
FROM customer
JOIN orders
ON (customers.cust_id=orders.cust_id)
WHERE state LIKE 'CA'
GROUP BY city
ORDER BY total DESC;
```

Hive proporciona una *shell* que permite ejecutar estas consultas llamada *beeline*, que se conecta al servidor Hive:

```
%> beeline -u jdbc:hive2://hostName:10000 -n username -p password
```

El uso de *beeline* no difiere mucho de cualquier herramienta que permita realizar exploraciones en bases de datos relacionales. Sin embargo, en Hive no existen tablas como tales, sino que en realidad son ubicaciones en el HDFS como:

```
/user/hive/warehouse/<database_name>
```

Las tablas pertenecientes a una base de datos se almacenarán como subdirectorios del directorio raíz de la base de datos y los ficheros almacenados pueden tener varios formatos soportados por Hive.

La ubicación de las tablas Hive y sus tipos de datos se almacenan, esta vez sí, en una base de datos relacional llamada *Metastore*. Esta base de datos no contiene datos de usuario final, sino que es solo una base de datos que Hive necesita

para caracterizar el contenido de una ubicación en HDFS como una tabla. Así, el usuario no tendrá visibilidad sobre esa tabla cuando realice consultas sobre los datos.

Hive puede usar Apache Tez, una nueva herramienta de ejecución de tareas en Hadoop MapReduce. Con Tez se aprovecha la ejecución siguiendo el modelo Directed Acyclic Graph (DAG). DAG permite modelar el dataflow de una consulta de una manera más intuitiva, evitando el uso de múltiples etapas de procesado MapReduce y la escritura de datos intermedios en memoria, con lo cual incrementa sustancialmente su velocidad. Puede decirse que Hive es el estándar de facto de SQL-en-Hadoop.

Dataflow

Dataflow es el conjunto de tareas encadenadas que realizan unas operaciones sobre unos datos. Los datos de salida de una tarea son la entrada a la siguiente tarea.

2) Impala

Impala, de forma parecida a Hive, permite realizar consultas SQL sobre datos distribuidos utilizando también un subconjunto del SQL-92. La sintaxis para realizar consultas de Impala es casi idéntica a Hive y se llama Impala SQL. Sin embargo, Impala es mucho más rápido que Hive y su uso está más enfocado a realizar consultas interactivas, con latencias de apenas unas décimas de segundo incluso menos.

Impala también dispone de una Shell a través de la cual se pueden realizar consultas:

```
sh>impala-shell
impala> SELECT cust_id, fname, lname FROM customers WHERE zipcode='20525';
```

Impala utiliza un motor SQL especializado y no traduce la consulta a tareas MapReduce, lo que incrementa el rendimiento de forma muy sustancial respecto a Hive.

Sin embargo, Impala no ofrece la escalabilidad que Hive posee. Aunque teóricamente Impala tiene la misma escalabilidad que Hive, al alcanzar un número de nodos elevado su rendimiento decrece de forma importante.

Idénticamente a Hive, la información relativa a las tablas que se han de consultar se almacena en el llamado *metastore*, el mismo que Hive. Si se modifican propiedades de las tablas utilizando Hive, se modificará el *metastore* que describe dicha tabla, e Impala va a detectar dichos cambios (no automáticamente, pero este es solo un aspecto práctico del uso de Impala).

La siguiente tabla muestra una comparativa entre Hive e Impala frente a las principales características de una base de datos relacional tradicional.

Comparativa entre Hive, Impala y bases de datos relacionales

Funcionalidad	Base de datos relacional	Hive	Impala
Lenguaje	SQL	SQL (*subset*)	SQL (*subset*)
Actualización de campos individuales	Sí	No	No
Borrado de campos individuales	Sí	No	No
Transacciones	Sí	No	No
Indexación	Extensiva	Limitada	No
Latencia	Muy baja	Alta	Baja
Tamaño soportado	terabytes	petabytes	petabytes

3)Apache Pig

Apache Pig es otra herramienta para explorar datos en entornos distribuidos. Tiene algunas similitudes a Hive, ya que permite a los desarrolladores crear rutinas de ejecución de consultas para el análisis de grandes conjuntos de datos distribuidos en un clúster Hadoop sin tener que programar a bajo nivel en MapReduce. Sin embargo, Pig no utiliza consultas tipo SQL. Pig permite a los usuarios escribir transformaciones complejas usando un lenguaje de script sencillo, llamado Pig Latin.

El lenguaje Pig Latin, más orientado a definir *workflows*, es traducido a secuencias MapReduce para que pueda ser ejecutado para trabajar sobre los datos en el entorno distribuido (HDFS). Pig es una herramienta útil para transformar y agregar datos (tareas ETL). Una consulta en Pig tendría este aspecto:

Workflow

Conjunto de tareas, sus interacciones, eventos e intercambio de información que, en su conjunto, realizan una operación completa.

```
people = LOAD '/user/training/customers' AS (cust_id, name);
orders = LOAD '/user/training/orders' AS (ord_id, cust_id, cost);
groups = GROUP orders BY cust_id;
totals = FOREACH groups GENERATE group, SUM(orders.cost) AS t;
result = JOIN totals BY group, people BY cust_id;
DUMP result;
```

Este script se traduce en una tarea MapReduce para ser ejecutada en el clúster y está formada, probablemente, por una secuencia de tareas MapReduce.

Spork

Actualmente hay un proyecto de integración de Pig con Spark, de modo que las tareas se ejecutarían en un clúster Spark en vez de MapReduce. Esta integración recibe el nombre de Spork.

Herramientas de almacenamiento en HDFS

Mencionaremos algunas de las herramientas de almacenamiento de datos en entornos HDFS, aunque hemos comentado otras anteriormente:

1) HBase. HBase es una base de datos NoSQL de código abierto, distribuido y escalable para el almacenamiento de *big data*. Está escrita en Java e implementa el concepto de Bigtable desarrollado por Google. Así como Bigtable aprovecha el almacenamiento de datos distribuidos proporcionado por el sistema de archivos de Google, Apache HBase Bigtable proporciona capacidades similares sobre Hadoop y HDFS.

2) Cassandra DB. Apache Cassandra es una base de datos distribuida para gestionar grandes cantidades de datos estructurados a través de muchos servidores *commodity*, al tiempo que proporciona servicio de alta disponibilidad y ningún punto único de fallo. Cassandra ofrece capacidades que las bases de datos relacionales y otras bases de datos NoSQL simplemente no pueden igualar tales como: disponibilidad continua, escalabilidad, simplicidad operativa y distribución de grandes cantidades de datos.

Enlaces de interés

Otras herramientas de almacenamiento NoSQL que hay que considerar son MongoDB y Couchbase.

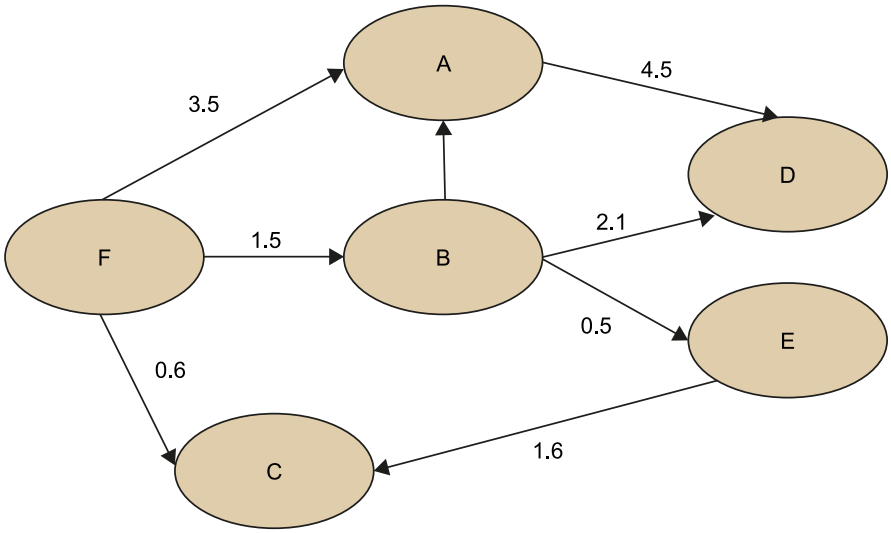
Herramientas de análisis de grafos

Muchos algoritmos de análisis de datos en *big data* están desarrollados para procesar datos de forma independiente unos de otros, realizando un agregado posterior de los resultados. Sin embargo, algunos problemas se centran en las relaciones existentes entre elementos individuales de datos, como podrían ser las redes sociales, enlaces entre páginas web o mapas de carreteras. Estas relaciones pueden representarse mediante los llamados grafos. El procesado de dichos grafos requiere de algoritmos de procesado específicos. La figura 2 muestra un ejemplo gráfico.

Grafos

Los grafos son una representación gráfica de un conjunto de objetos llamados nodos o vértices unidos entre ellos mediante enlaces, también llamadas aristas.

Figura 2. Ejemplo de grafo en que 6 puntos se interconectan entre ellos mediante unas aristas.



Típicamente, un problema de análisis de grafos se puede descomponer en los siguientes pasos:

- Creación del grafo, donde se describen las relaciones entre los diferentes componentes del sistema que se analiza. Por ejemplo, enlaces de una página web.

- Representación: tabla con las dependencias representadas (vértices y ejes)

- Análisis: este tipo de problemas habitualmente requieren de procesos iterativos, difíciles de paralelizar. Este es el enfoque tomado por librerías como Pregel o GraphLab.

- Análisis posterior: por ejemplo, añadir recomendaciones en función del resultado del análisis.

A continuación describiremos un ejemplo típico de trabajo con grafos.

PageRank

PageRank es un algoritmo utilizado por Google para clasificar páginas web en base a las referencias cruzadas entre ellas. La idea que subyace es que las páginas web más importantes tienen más referencias de otras páginas web.

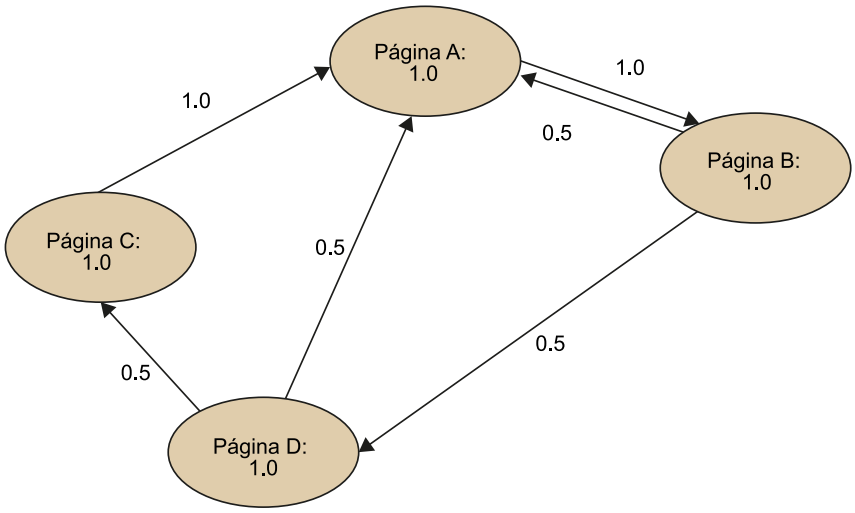
Damping

Damping es la probabilidad de que un usuario vaya a dejar de navegar por las páginas referenciadas entre ellas. Se estima que su valor es 0,85.

Ejemplo de trabajo con PageRank

En el siguiente ejemplo, consideraremos cuatro páginas que se referencian entre sí (página A, B, C y D). Cada una contribuye a las otras con su peso (valor dentro del círculo o vértice de la figura 3) dividido por el número de referencias a las otras (si una página tiene un peso de 1,0 y dos referencias a otras páginas, contribuye con 0,5 a cada una de ellas).

Figura 3. Representación gráfica del ejemplo PageRank descrito.



Las contribuciones se ajustan por el factor de *damping*, un ajuste estadístico que tiene un valor aproximadamente de 0,85.

Así, en cada iteración i el peso de cada página se actualiza como su peso P_i (i de iteración) menos el factor de *damping* (d), más la suma de los pesos del resto de páginas que la referencian, aplicándole el factor d.

Para calcular el peso de la página A en la iteración $i + 1$, utilizamos la siguiente ecuación:

$$P_{i+1}(A) = (P_i(A) - d) + d(P_i(B) + P_i(C) + P_i(D))$$

donde $d = 0{,}85$ y P_i es el peso de la página en la iteración i.

En nuestro ejemplo, se aplicaría como sigue:

$$P_{i+1}(A) = (1 - 0{,}85) + 0{,}85(1{,}0 + 0{,}5 + 0{,}5) = 1{,}85$$

Al aplicarse a todo el grafo, tras la primera iteración obtenemos:

$$P_1(A) = 1{,}85P_1(B) = 1{,}0P_1(C) = 0{,}58P_1(D) = 0{,}58$$

Tras la segunda:

$$P_2(A) = 1{,}31P_2(B) = 1{,}7P_2(C) = 0{,}39P_2(D) = 0{,}57$$

...y tras la iteración 10:

$$P_{10}(A) = 1{,}43P_{10}(B) = 1{,}38P_{10}(C) = 0{,}46P_{10}(D) = 0{,}73$$

La solución va a converger cuando el incremento en cada iteración sea muy pequeño. El número de iteraciones va a depender de la precisión que deseamos obtener; cuantas más iteraciones, mayor tiempo de computación tomará.

Veamos ahora **Apache Giraph**, una herramienta para procesado de grafos utilizada por Facebook para análisis de usuarios y sus conexiones. La herramienta se fundamenta en Pregel, la arquitectura de procesado de grafos de Google. Giraph traduce un grafo a un conjunto de tareas MapReduce.

Una tarea en Giraph se inicia a partir de los datos de entrada. Típicamente, la entrada de datos sería un fichero en el que cada línea contiene información de las relaciones entre vértices:

- **Identificador del vértice (*id*)**: puede ser una etiqueta o un objeto más complejo.
- **Valor del vértice**: ofrece información adicional sobre el vértice y puede almacenar tanto valores como objetos más complejos.
- **Eje**: tupla con el *id* del vértice de destino y un peso para dicho vértice.

Un ejemplo sería el siguiente:

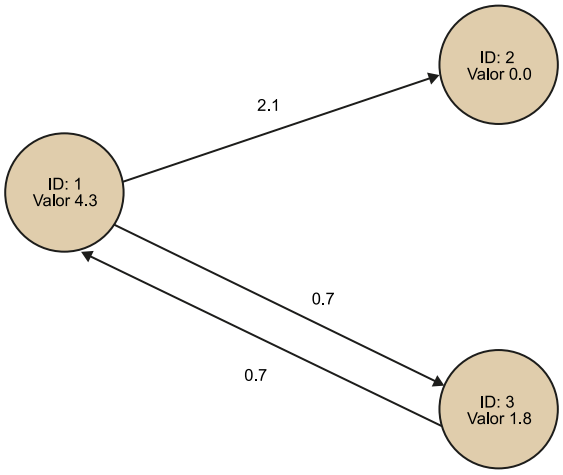
(1, 4.3, ((2, 2.1), (3, 0,7)))

(2, 0,0, ())

(3, 1,8, ((1, 0,7))

Que se representa gráficamente en la figura 4.

Figura 4. Representación gráfica de los vértices del ejemplo



Sobre un grafo como este se podría ejecutar un algoritmo *PageRank* o un *Shortest Path*. En Giraph, cada nodo *worker* o esclavo realiza una operación, almacenando datos en el sistema HDFS. Dicho nodo mantiene el grafo en memoria (todo o parte de él) durante la ejecución. Si es muy grande, este se divide y distribuye entre varios nodos.

Shortest Path

Shortest Path es un algoritmo para encontrar la distancia mínima entre dos vértices en un grafo, definida como aquella en la que la suma de los pesos de los vértices para unir dichos vértices es mínima.

Un algoritmo en Giraph es un proceso iterativo de "super-pasos" que consisten en:

Super-paso

Un super-paso es cada una de las iteraciones de Giraph en las que los valores de los vértices y las aristas se actualizan.

- **Procesado local:** cada nodo realiza el procesado local utilizando los datos de que dispone sin intercomunicación con otros nodos.

- **Comunicación:** los diferentes procesos se intercambian información sobre su estado y valores.

- **Sincronización:** todos esperan a que el proceso de comunicación haya finalizado. El agregador es el componente que aglutina los valores de cada vértice en el super-paso S. Al combinarlos, este valor se distribuye a todos los vértices para la siguiente iteración o super-paso $S + 1$.

Los nodos se intercambian mensajes en los que se incluye el identificador del vértice *I*, el valor del vértice *V*, el valor del eje (*edge E*) y *M*, como tipo de mensaje. Este protocolo es llamado *Bulk Synchronous Processing* (BSP). Cada vértice puede mandar un número ilimitado de mensajes en un super-paso a otro vértice actualizando el estado del grafo para la siguiente iteración. Al utilizar este protocolo, Giraph evita la creación de un solo proceso MapReduce para cada iteración y solo utiliza la etapa Map para realizar cálculos sin utilizar Reducers.

El grafo se carga una sola vez en memoria y los cálculos se realizan con poco acceso a disco. El modelo MapReduce completo para análisis de grafo generaría un gran *overhead* debido a la escritura de datos en disco para cada super-paso y la recarga del grafo actualizado para la nueva iteración.

Overhead

Overhead es el exceso en el uso de cualquier recurso de computación –ya sea memoria, tráfico de datos, tiempo de cálculo, etc.– que condiciona y puede penalizar su rendimiento.

2.1.3. Distribuciones Hadoop

Existen tres grandes distribuciones de Hadoop:

- **Cloudera** es la más antigua y fue creada por técnicos de Facebook, Google, Oracle y Yahoo en 2008. Es la que tiene una mayor comunidad de usuarios y, aunque su núcleo es Hadoop, ofrece otras utilidades y herramientas propietarias de Cloudera para la gestión de un clúster Cloudera (Cloudera Manager, por ejemplo).

- **Hortonworks** es el único proveedor comercial que ofrece una distribución de código abierto de Apache Hadoop sin software propietario adicional.

La distribución HDP2.0 de Hortonworks se puede descargar directamente desde su página web sin coste.

- **MapR**, a diferencia de Hortonworks y Cloudera, va un paso más allá y ofrece algunas soluciones propietarias como un sistema de ficheros distribuido propio, el MapRFS, además de otras herramientas orientadas a la manejabilidad y facilidad de uso. Podríamos decir que es una solución más preparada para entornos de producción.

2.1.4. Resumen

Es importante tener en cuenta que el ecosistema Hadoop está formado por muchas herramientas que proveen servicios y utilidades para trabajar sobre el *framework* Hadoop. Muchas de ellas están en fase de desarrollo y otras, probablemente, no van a tener más recorrido.

Enlace de interés

Puede consultarse la lista completa de proyectos en el siguiente enlace: https://hadoopecosystemtable.github.io/

Tras la irrupción de Spark, del que hablaremos a continuación, muchas de estas herramientas van a ir perdiendo utilidad y, probablemente, van a ser mantenidas solo por el uso generalizado que se ha hecho de ellas en algunas empresas e instituciones. Sin embargo, poco a poco deberían ir perdiendo peso en favor de Spark y su ecosistema.

2.2. Spark framework

A partir del paradigma MapReduce y del *framework* Hadoop, muchas herramientas han aparecido para cubrir necesidades diversas, como hemos comentado en los subapartados anteriores. Sin embargo, de entre todas las plataformas y herramientas *big data* aparecidas tras la irrupción de Hadoop, la más destacable es Apache Spark.

Spark se presenta con el objetivo de solucionar tres problemas clave:

- **Usabilidad.** Escribir aplicaciones en Spark es realmente sencillo y la configuración de un clúster Spark es relativamente fácil, mejorando significativamente el desarrollo de aplicaciones *big data* más allá de MapReduce.

- **Rendimiento.** Spark resuelve uno de los grandes problemas de MapReduce, su rendimiento. Sin duda, a pesar de lo novedoso del paradigma MapReduce, su rendimiento es bajo debido a que trabaja esencialmente en base a ficheros, además de estar sujeto a un modelo de difícil implementación. Sin embargo, Spark hace uso de la memoria RAM del sistema, sa-

cando provecho de las arquitecturas modernas, en las cuales la memoria RAM ya no es un recurso tan costoso como lo fue anteriormente.

- **Simplicidad.** Spark se presenta como un motor de procesado distribuido, más allá de una herramienta específica. Lo que permite ofrecer múltiples herramientas sobre él. Hasta ahora, algunas funcionalidades de consulta o procesado requieren el uso de diferentes herramientas. Sin embargo, Spark ofrece muchas funcionalidades agrupadas en un conjunto de herramientas, como SparkSQL, *streaming*, Machine Learning o GraphX. Antes de Spark, una operación que pudiera requerir extraer datos de una fuente de datos diversa, aplicar algún algoritmo de Machine Learning y finalmente realizar consultas sobre esos datos podría requerir tres motores distintos: Sqoop o Flume, Mahout para las tareas de Machine Learning, y Hive o Impala como herramientas de consultas sobre entornos distribuidos. Spark, sin embargo, ofrece esta funcionalidad a través de las distintas herramientas del *framework* con total interactividad entre ellas.

En este subapartado vamos a estudiar, de forma general, cuáles son los componentes de Spark y cómo funcionan desde el punto de vista de su arquitectura.

2.2.1. Características Spark

Spark extiende el paradigma MapReduce ofreciendo más modelos de computación, facilitando la realización de tareas más complejas e incrementando la velocidad de procesado hasta dos órdenes de magnitud, gracias a un uso de memoria mucho más intenso que Hadoop.

Pipeline

Pipeline es la transformación de un flujo de datos en un proceso formado por varias tareas ejecutadas de forma secuencial, siendo la entrada de cada una la salida de la anterior.

Por otro lado, Spark hace uso del modelo de computación DAG (Directed Acyclic Graph), un modelo de ejecución de tareas en el cual estas son vértices de un grafo y el orden de ejecución se especifica por la direccionalidad de los ejes del grafo. *Acyclic* significa que no hay bucles en el grafo. Además, en un modelo de computación DAG, los procesos se ejecutan en paralelo. Este modelo ofrece una mayor flexibilidad para implementar computaciones en modo *pipeline*, mucho más avanzado que el modelo MapReduce.

A modo de ejemplo, tómese la tarea de *machine learning* de análisis de componentes principales (*Principal Component Analysis* o PCA) sobre un paquete de datos formado por 2 millones de *arrays* de 480 valores cada uno. Utilizando un clúster de 16 nodos, Apache Mahout, herramienta basada en MapReduce de la que hablaremos más adelante, tomó 7200 segundos. Spark realizó la misma operación en 150 segundos.

A continuación analizaremos algunos de los aspectos más relevantes de Spark y que lo convierten en una plataforma tan atractiva actualmente.

Facilidad de programación

Sin duda, Spark ha simplificado enormemente el modelo de programación sobre entornos distribuidos. Un elemento clave de esta simplificación es la estructura RDD (*Resilient Distributed Dataset*). RDD es una colección de objetos

distribuida a través del clúster de forma inmutable. Puede crearse RDD cargando datos externos (desde una ubicación local o un entorno distribuido como HDFS, para el usuario es transparente) o mediante la transformación de un RDD ya existente. Sobre RDD se pueden realizar transformaciones o acciones:

- Una transformación sobre un RDD es una operación que da como resultado otro RDD. Spark ofrece múltiples tipos de transformaciones sobre RDD como *map*, *reduce*, *filter* o *join* entre otras. Las transformaciones sobre un RDD se ejecutan de forma *lazy*, es decir, solo son ejecutadas cuando se invoca una acción.

- Una acción sobre un RDD es una operación que da como resultado un valor final.

Como ejemplo, consideremos el siguiente código en Scala:

```
val datosFichero = sc.textFile("/home/datos.txt")
val numeroLineas = datosFichero.count()
```

En este ejemplo, los datos no se leen realmente del fichero hasta que se realiza la llamada `count()`. La variable `datosFichero` es un RDD, mientras que `numeroLineas` será un tipo numérico entero. Sin embargo, ¿qué ocurre si volvemos a llamar la función `count()` sobre `datosFichero`? Pues que el fichero volverá a leerse de nuevo, lo que no sería eficiente. En este caso, si prevemos que un RDD va a ser utilizado de forma repetida en una aplicación nos será de utilidad utilizar la funcionalidad de Spark `cache()` o `persist()`. Al utilizar esta función, cuando se ejecute la acción, los datos van a ser procesados (en este caso se leerá el fichero) y almacenados en la memoria.

```
val datosFichero = sc.textFile("/home/datos.txt")
datosFichero.cache()
val numeroLineas = datosFichero.count()
val numeroLineas2 = datosFichero.count()
```

En el ejemplo utilizado, `numeroLineas2` ya no va a cargar de nuevo el fichero sino que cargará los datos almacenados en memoria. El comportamiento del programa dependerá de la memoria disponible: si los datos del RDD no caben en la memoria, volverá a leer los datos de disco. Como puede deducirse, cuanto mayor sea la memoria disponible para la ejecución de una aplicación Spark, mejor será su rendimiento. Pero también unas buenas prácticas en el desarrollo de aplicaciones Spark tendrán un impacto muy significativo en el rendimiento de una aplicación. Spark soporta diferentes lenguajes de programación como Scala, Python, Java y recientemente R, mediante el paquete SparkR.

Componentes de un programa en Spark

A continuación vamos a describir cuáles son los componentes de una aplicación Spark. Estos componentes tienen relación directa en cómo se lanza la ejecución de una aplicación Spark y qué gestor de recursos utiliza, ya sea el clúster de Spark, YARN o MESOS. Dichos gestores ya se han visto anteriormente. La aplicación *driver* o controladora es la aplicación que inicia el programa, crea el llamado *SparkContext*, declara las transformaciones y acciones sobre los RDD serializados y los manda al *Master Node* de nuestro clúster. El *driver* pide recursos al Master para realizar una serie de tareas sobre los RDD. Cuando el proceso *driver* necesita recursos para ejecutar trabajos o tareas, se los pide al Master y este asigna los recursos y utiliza los *workers* para crear ejecutores (o *executors* en inglés). Los ejecutores están activos mientras la aplicación se esté ejecutando. El ejecutor, a su vez, puede usar múltiples *threads* para la ejecución de una tarea asignada.

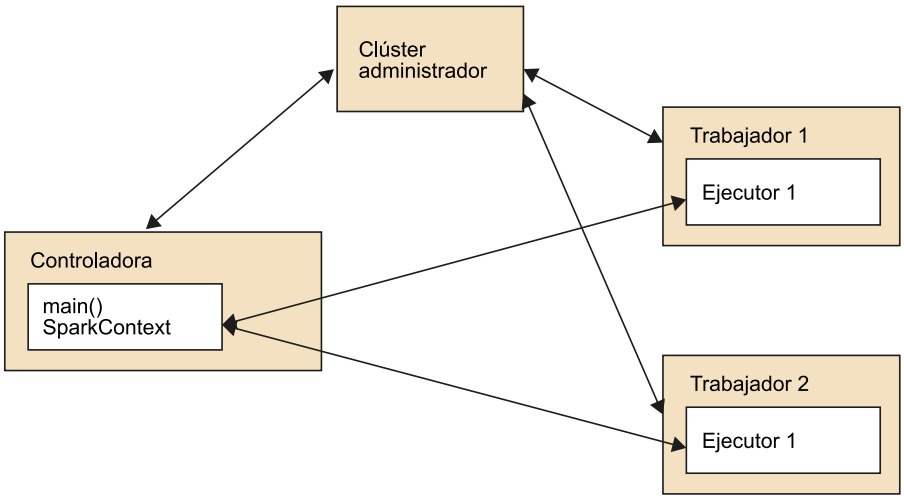
Thread

Thread, también llamado hilo, es un proceso que se ejecuta en paralelo o de forma asíncrona a la de otros procesos que forman una aplicación.

Es importante tener en cuenta que Spark es compatible con diversos gestores de recursos, por tanto, dependiendo del gestor de recursos utilizado (Cluster Spark, YARN o MESOS), pueden configurarse los recursos que se han de utilizar para cada uno de los componentes del programa: memoria para el *driver*, número de *workers*, número de ejecutores para cada *worker*, número de *threads* y memoria para cada por ejecutor, etc.

Como muestra la figura 5, existe comunicación entre ejecutores de una misma aplicación, pero cada aplicación Spark corre sin interacción con cualquier otra aplicación Spark que se esté ejecutando, ya que tareas de aplicaciones distintas se ejecutan en JVM diferentes.

Figura 5. Componentes de un programa Spark ejecutado en un *clúster*



Desde un punto de vista práctico, la aplicación esta típicamente contenida en un fichero *jar*. Este *jar* debe ser copiado a los diferentes *workers* para poder ejecutar la tarea a realizar. Además, las tareas que han de realizar los ejecutores deberán ser serializables. Los pasos para ejecutar una tarea sobre un RDD son:

1) La tarea que se debe realizar se serializa en el nodo *driver*, donde serializar significa traducir en bytes un paquete de datos contenido en memoria para poder ser transferidos, en este caso, a través de la red, a los diferentes nodos.

2) Cuando la tarea es recibida en el nodo, se de-serializa y se ejecuta.

Una aplicación Spark admite diversos modos de ejecución:

- **Modo local**: la aplicación se ejecuta en el ordenador que la está invocando, tanto el *driver* como los ejecutores. Si el equipo tiene más de una CPU o núcleos, Spark aprovechará la arquitectura para paralelizar las tareas.

- **Modo cliente**: en este modo, el driver se ejecuta fuera del clúster, pero las tareas paralelizables se ejecutan en el clúster (en YARN, hablaríamos del modo "yarn-client"). Un ejemplo de este modo sería la consola de Spark, que se ejecuta en el equipo que la invoca (un portátil u ordenador de sobremesa) mientras que las operaciones sobre los RDD se ejecutan en el clúster.

- **Modo clúster**: El driver y las tareas que hay que paralelizar se ejecutan dentro del clúster. Como ejemplo, en este modo la consola no es ejecutable, ya que la tarea queda en cola de ejecución dentro del clúster sin interactividad con el usuario.

Paralelismo en Spark

Un RDD, aunque se ha presentado como una entidad monolítica, no es un solo bloque ocupando un espacio de memoria en un nodo, sino que es una entidad que se construye a partir de bloques de memoria distribuidos en cada nodo, dando lugar a las llamadas particiones. Este particionamiento lo realiza Spark y es transparente para el usuario, aunque puede tener control sobre él. Si los datos se leen del sistema HDFS, cada partición va a corresponderse con un bloque de datos del sistema HDFS. Sin embargo, para las particiones creadas dinámicamente, el usuario puede controlar el número de particiones. El concepto partición tiene un peso importante para comprender cómo paraleliza Spark.

Cuando Spark lee un fichero, si este es mayor que el tamaño de bloque definido por el sistema HDFS (típicamente 64 MB o 128 MB), se crea una partición para cada bloque. Sin embargo, el usuario puede definir el número de particiones a crear cuando está leyendo el fichero:

```
scala-shell> sc.textFile("filename", 4)
```

Esta llamada creará un RDD que apuntará a los datos del fichero *filename* formado por 4 particiones distribuidas entre los nodos. Spark va a rendir mejor con ficheros grandes (mayores que el tamaño de bloque del sistema HDFS),

ya que esto permitirá mayor paralelización cuando se procesen los datos del fichero cargado. El número mínimo de particiones, en cualquier caso, es de dos, incluso para ficheros muy pequeños.

Cuando se leen varios ficheros de una ubicación, cada fichero se va a asignar como mínimo a una partición. La casuística de lectura de ficheros en Spark y su asignación a particiones y RDD es algo variada, sin embargo, es importante tener en cuenta que cada partición es la unidad de datos que van a ser procesados por un ejecutor distribuido en el clúster.

Los nodos del clúster pueden tener particiones de un RDD sobre el que se va a ejecutar una tarea. En ese caso, si en ese mismo nodo hay un ejecutor asignado, este va a tratar los datos de su mismo nodo, lo que se conoce como *data locality*. De este modo se evita el tráfico innecesario en la red que supondría tener ejecutores procesando datos que residen en nodos distintos, incrementando en tráfico en la red. Esto es posible porque el programa *driver* conoce donde residen los datos (el **NameNode** de HDFS se lo indica) y también conoce la ubicación de cada ejecutor, ya que es el propio *driver* quien los registra antes de la ejecución de un programa. Sin embargo, si no es posible que un ejecutor procese los datos que localmente están en ese nodo, estos van a tener que transferirse por la red.

Finalmente, cuando hay una llamada al método `collect`, por ejemplo, los datos viajan hasta el programa *driver*. Aunque Spark, en tiempo de ejecución, va a tratar de que cada ejecutor procese los datos que contiene localmente ('localidad de datos'), esto no puede definirse antes de empezar la ejecución, como si se puede en Hadoop.

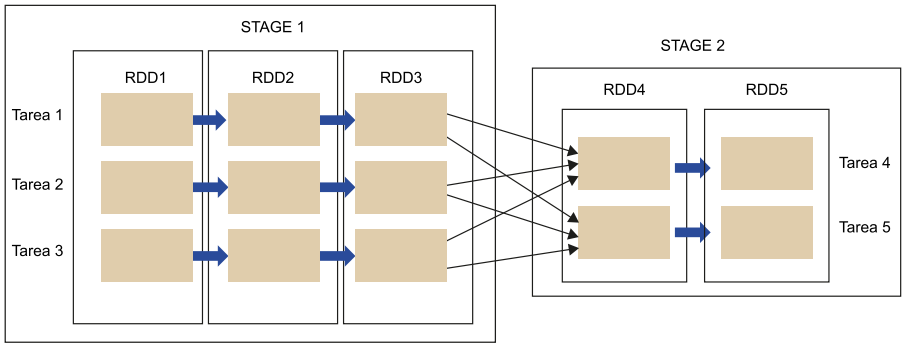
Para entender el paralelismo en Spark, vamos a vincular el concepto partición con su tratamiento en un proceso Spark: cuando un fichero se lee del sistema HDFS, se crean particiones tal y como se ha indicado y un conjunto de estas particiones forman el RDD. Al ejecutar operaciones de transformación sobre el RDD se preservan las particiones creadas, de modo que no hay movimiento de datos por la red. Sin embargo, al realizar una tarea Reducer (por ejemplo *groupByKey*), se reconfiguran las particiones, y los datos provenientes de diferentes particiones se agrupan en un conjunto menor de particiones. De nuevo, si nuevas transformaciones se aplicaran sobre las particiones estas volverían a preservase.

Al conjunto de operaciones sobre las que se realizan transformaciones preservando las particiones se las llama *stage*. Dicho de otro modo, tareas que se ejecutan en el nodo sin intercambio de datos con otros nodos, beneficiándose de la localidad de datos. El final de un *stage* se presenta cuando se reconfiguran las particiones (operaciones *reduce*), que inician el llamado *shuffling* o intercambio de datos entre nodos.

Para beneficiarse de la localidad de datos y evitarse la creación de datos intermedios, un pipeline de operaciones sobre un RDD se ejecutará para cada elemento del RDD, en vez de ejecutarse toda una transformación sobre el RDD antes de ejecutarse la siguiente de forma secuencial.

Para ilustrar esta idea obsérvese un ejemplo en la figura 6.

Figura 6. Procesado en paralelo de RDD

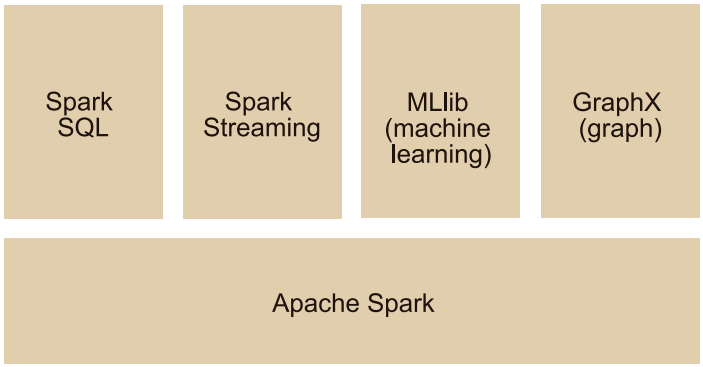


Un RDD1 inicial está dividido en tres particiones (bloques de color en la figura 6). Se desea realizar una transformación de este RDD1 a otro RDD2 y seguidamente a otro RDD3. Tal como Spark gestiona los RDD y sus particiones, no se procesan todos los elementos del RDD1 en la transformación hasta el RDD2, sino que se procesa cada elemento del RDD1 hasta el RDD3, y así para cada partición (en el ejemplo, al tener tres particiones habrá tres tareas). En el paso de RDD3 a RDD4 se produce el *shuffling*, producido por una operación *reduce*, configurando dos nuevas particiones, sobre las cuales se puedes realizar transformaciones, etc., preservando el mismo modelo de procesado paralelo y configuración de particiones.

2.2.2. Spark API: MlLib, DataFrames, GraphX, Stream

Como se ha indicado anteriormente, Spark soluciona una de las problemáticas del ecosistema Hadoop, que es la gran cantidad de herramientas necesarias para realizar distintas funciones. Con Spark, se ofrece un *framework* único con diversas funcionalidades. Sobre el motor de procesado Spark se proveen cuatro grandes API: Spark SQL, Spark Streaming, MLlib y GraphX, mostrados en la figura 7.

Figura 7. Spark API



En este subapartado hablaremos de cuáles son esas funcionalidades y sus principales características.

Machine Learning

Los algoritmos de Machine Learning (MLlib) suelen requerir cálculos computacionales intensos, en procesos muy iterativos. Muchos sistemas de análisis de datos tradicionales no escalan bien sobre grandes conjuntos de datos. En consecuencia, nuevas solucionas han aparecido sobre el *framework* Hadoop como por ejemplo Mahout, que utiliza el paradigma MapReduce, o la API MLlib de Spark. MLlib es parte de un proyecto mayor llamado MLbase, de AMPLabs, cuyo objetivo es implementar soluciones de Machine Learning altamente escalables.

Dentro de dicho proyecto, MLlib es la librería de Machine Learning o aprendizaje automático de Spark. Es accesible mediante una API que se incluye en las distribuciones de Spark, proporcionando una de las funcionalidades más potentes del *framework*. Actualmente, MLib tiene una gran cantidad de contribuciones y amplía su oferta de forma significativa en cada nueva versión. Sin entrar en detalle de las funcionalidades que la librería ofrece, estas se agrupan como sigue:

- **Clasificación supervisada y regresión**: sistemas en los que a partir de un conjunto de ejemplos clasificados, también llamado "conjunto de entrenamiento", intentamos asignar una clasificación a un segundo conjunto de datos sin clasificar. Spark incluye diferentes algoritmos como SVM (Máquinas de vectores de soporte), regresión logística, regresiones lineales, árboles de decisión, etc.

- **Clasificación no supervisada o *clustering***: sistemas en los que no disponemos de una batería de ejemplos previamente clasificados, sino que únicamente a partir de las propiedades de los datos disponibles intentamos dar una agrupación según su similitud. Spark implementa algunos de los algoritmos más conocidos como por ejemplo k-means y Gaussian Mixture Models (o GMM).

Enlace de interés

Se recomienda visitar la página de MLlib de Spark, donde se describen el funcionamiento de los algoritmos implementados con ejemplos en los diferentes lenguajes de programación soportados por Spark.

- **Reducción de dimensionalidad**: debido al elevado número de variables de un problema, podemos usar las técnicas de reducción de dimensionalidad para descartar aquellas que tienen poco peso en la caracterización del problema que tratamos de evaluar. Spark implementa el análisis por componentes principales (o PCA) y la descomposición en valores singulares (o SVD).

- Se incluyen también métodos de evaluación de modelos, algoritmos de Filtrado Colaborativo (ALS o mínimos cuadrados alternantes), utilizado en sistemas recomendadores.

SparkSQL

La API SparkSQL es la continuación del proyecto Shark y su objetivo es proveer herramientas de procesado de datos de forma estructurados en entornos distribuidos. En un principio, Apache Shark fue un *framework* para integrar Hive en Spark. Sin embargo, como ya se ha indicado, Hive es una herramienta orientada a procesos *batch* y la industria buscaba aplicaciones con latencias menores que Hive no podía ofrecer. Además, Shark heredó una gran cantidad de aplicaciones y código Hive difícil de mantener y optimizar. Así, SparkSQL nace con la idea de tomar las fortalezas de Hive, evitando sus debilidades. En el núcleo de SQLSpark se encuentra el *Catalyst Optimizer* un *framework* optimizador extensible que hace uso de las funcionalidades del lenguaje Scala para optimizar la ejecución de consultas.

El acceso a la API de SparkSQL la ofrece el *SQLContext* de Spark, que se deriva del ya introducido *SparkContext*. Hay dos implementaciones del Spark SQL Context: el SQLContext propiamente dicho y el HiveContext, para esas aplicaciones que necesitan ofrecer compatibilidad con Hive.

DataFrames

SparkSQL introduce la API DataFrame, una librería que permite trabajar con los datos distribuidos como si fueran tablas. El DataFrame es la principal abstracción en SparkSQL y se construye en base a un RDD de Spark, pero a diferencia de este, el DataFrame contiene objetos del tipo *Row*, mientras que el RDD contiene "elementos" genéricos. Los DataFrames son muy flexibles y pueden construirse a partir de ficheros con datos estructurados, como ficheros Parquet y JSON, o pueden ser el resultado de la transformación de un RDD ya existente o la agregación de otros DataFrames.

SparkSQL incluye tres tipos de fuentes de datos: JSON, Parquet y JDBC. Aunque también pueden usarse librerías de terceros para cargar datos en formato Avro, HBase, CSV y MySQL entre otros.

Un DataFrame comparte muchas características con los RDD: son inmutables, una consulta sobre un DataFrame da como resultado otro DataFrame (también en modo *lazy*) y una acción desencadena la ejecución de la consulta en si (idénticamente como una acción en un RDD desencadena las transformaciones definidas sobre el RDD previamente).

Un DataFrame se caracterizada por su esquema, que contiene la información relativa a sus datos (metadatos), como por ejemplo nombres de los campos, el número de columnas, etc.

La API permite realizar consultas sobre los datos distribuidos mediante funciones como: `select()`, `where()`, `sort()`, `join()`.

Así, por ejemplo:

```
> peopleDF.select(peopleDF("name"), peopleDF("age") + 10)
> peopleDF.sort(peopleDF.age.desc())
```

Aunque las consultas también pueden ejecutarse de forma SQL directa como:

```
> peopleDF.registerTempTable("people")
> sqlContext("SELECT * FROM people WHERE name LIKE "A%" ")
```

Los resultados pueden guardarse en ficheros Parquet, tablas Hive u otros formatos específicos con la ayuda de librerías externas. El DataFrame es un nivel de abstracción superior a un RDD; así, el RDD del que deriva puede obtenerse como:

```
> rddPadre = dataFrame.rdd
```

e inversamente, un RDD puede convertirse a un DataFrame mediante la llamada:

```
> sqlContext.createDataFrame(rdd)
```

¿Cuál es la mejor herramienta para realizar consultas SQL sobre datos distribuidos?

He aquí algunas de sus características comparativas:

- **Hive.** Orientado a *batch*, es una herramienta estable y altamente escalable para cantidades de datos enormes, pero no ofrece capacidades de consulta interactiva.

- **Impala.** Es la más rápida, muy orientada al Business Intelligence y a consultas SQL específicamente.

- **Spark SQL.** Como parte de la plataforma Spark, es altamente integrable con el resto de herramientas Spark, lo que la dota de un enorme potencial para definir cadenas de procesado complejas, incluyendo consultas SQL, Machine Learning, etc.

GraphX

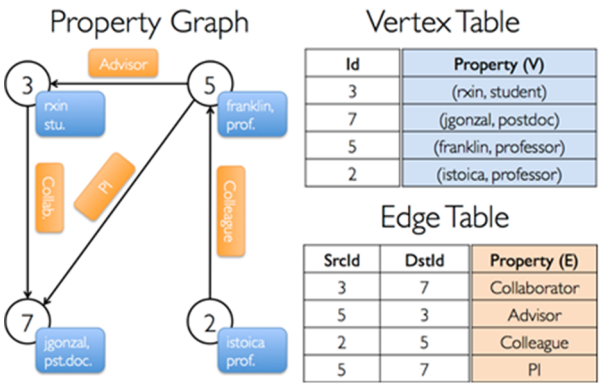
GraphX es la API de Spark para el procesado y computación paralela de grafos. Tal y como hemos visto con las otras API de Spark, en GraphX la capa de abstracción RDD es la llamada *Resilient Distributed Property Graph* y expone una serie de operadores y algoritmos específicos para simplificar el trabajo con grafos. GraphX implementa una versión optimizada de la API Pregel.

Así como Giraph es un *framework* especializado en el trabajo con grafos que implementa el método Pregel y el modelo BSP (Bulk Synchronous Parallel) en toda su extensión, GraphX es la API de Spark para extender sus funcionalidades para trabajar con grafos, que a su vez son representables como tablas, unificando el procesado de datos y el de grafos en paralelo.

En ocasiones el usuario va a preferir representar los datos como tablas en vez de vértices y ejes, de modo que GraphX permite trabajar con RDD que pueden visualizarse a la vez como componentes de un grafo (vértices: VertexRDD; y ejes: EdgeRDD) y sus características o como colecciones (RDD propiamente dichos).

Por tanto, un grafo en GraphX es solo una vista de los mismos datos y cada vista tiene sus operadores específicos. Como muestra la figura 8, el mismo grafo (vértices y ejes) pueden tratarse como una colección de datos, ya sea de vértices o de aristas.

Figura 8. Representaciones de un grafo en GraphX y sus posibles vistas: vértices o aristas



3. Procesado en *streaming*

Hasta ahora hemos hablado de herramientas utilizadas para trabajar en entornos de procesado *batch*. A continuación presentaremos las herramientas más habituales para entornos de procesado distribuido en *streaming*.

3.1. Apache Flume

Apache Flume es una herramienta cuya principal funcionalidad es recoger, agregar y mover grandes volúmenes de datos provenientes de diferentes fuentes hacia el repositorio HDFS en casi-tiempo real. Es útil en situaciones en las que los datos se crean de forma regular y/o espontánea, de modo que a medida que nuevos datos aparecen ya sean *logs* o datos de otras fuentes, estos son almacenados en el sistema distribuido automáticamente.

El uso de Apache Flume no se limita a la agregación de datos desde *logs*. Debido a que las fuentes de datos son configurables, Flume permite ser usado para recoger datos desde eventos ligados al tráfico de red, redes sociales, mensajes de correo electrónico a casi cualquier tipo de fuente de generación de datos dinámica.

Podemos dividir la arquitectura de Flume en los siguientes elementos:

- **Fuente externa.** Se trata de la aplicación o mecanismo, como un servidor web o una consola de comandos, desde la cual se generan eventos de datos que van a ser recogidos por la fuente.

- **Agente (o *agent*).** Es un proceso Java que se encarga de recoger eventos desde la fuente externa en un formato reconocible por Flume y pasárselos transaccionalmente al canal. Si una tarea no se ha completado, debido a su carácter transaccional, esta se reintenta por completo y se eliminan resultados parciales.

- **Canal.** Un canal actuará de almacén intermedio entre el agente y el sumidero (descrito a continuación). El agente será el encargado de escribir los datos en el canal y permanecerán en él hasta que el sumidero u otro canal los consuman. El canal puede ser "memoria", de modo que los datos se almacenan en la RAM o pueden ser almacenados en disco, usando ficheros como almacén de datos intermedios o una base de datos mediante conexiones JDBC.

- **Sumidero o *sink*.** Será encargado de recoger los datos desde el canal intermedio dentro de una transacción y de moverlos a un repositorio externo, otra fuente o a un canal intermedio. Flume es capaz de almacenar datos en

diferentes formatos HDFS como por ejemplo texto, SequeneceFiles, JSON o Avro.

3.2. Apache Kafka

Apache Kafka es un proyecto Apache originariamente desarrollado por Linkedin. Es un sistema distribuido de publicación-subscripción de mensajes y está diseñado para procesar en tiempo real la actividad en un *stream* de datos. Desde un punto de vista terminológico, en Kafka existen los:

1) *Topics*: categorización de la información por Kafka. Un *topic* es una categoría en la cual se publican mensajes.

2) *Producers*: procesos que publican mensajes acerca de un *topic*.

3) *Consumers*: procesos consumidores de la información acerca de un *topic* en base a una subscripción a dicha categoría.

4) *Broker*: el clúster Kafka, consistente en uno a más servidores.

¿Cómo funciona? Para cada *topic*, el *broker* Kafka mantiene la información particionada, distribuida y replicada entre los servidores del clúster. Cada partición es una secuencia inmutable y ordenada de mensajes que se actualiza de forma continua y cada mensaje en la partición está identificado con un *id*, que lo identifica de forma unívoca. Todos los mensajes se guardan en el servidor Kafka durante un período de tiempo independientemente de si se han consumido o no. El particionamiento de la información permite paralelismo y escalabilidad.

Cada servidor Kafka cuenta con un proceso líder, que se encarga de gestionar todas las lecturas y escrituras de mensajes. Otros procesos llamados *followers* imitan la actividad del líder permitiendo que en caso de fallo del líder, un *follower* pueda reemplazarle y mantener el estado del sistema actualizado. Además, para cada servidor, las particiones tienen líderes y *followers* distintos, balanceando la carga en el clúster. Los *producers* o productores se encargan de publicar mensajes en cada *topic*, pudiendo escoger en qué partición publicar el mensaje y finalmente, cada mensaje publicado en un *topic* es entregado a un solo consumidor "suscrito" a ese *topic* de información.

El paralelismo en Kafka viene determinado por el particionamiento de la información. Si un *topic* tiene *N* particiones, el grupo de consumidores (procesos que procesan la información) puede asignar un máximo de *Nthreads* de procesado en paralelo. Así, si un *topic* está formado por 10 particiones y tenemos 15 consumidores, solo 10 consumidores van a leer información de dicho *topic*, mientras que 5 permanecerán inactivos.

3.2.1. Integración con Spark

No obstante, existe otro modo de acceder a Kafka mediante Spark, en **modo directo**. En esta modalidad se permite la integración con Spark de modo directo, sin el uso de consumidores. En este caso, el consumidor de información consulta al servidor Kafka acerca de la actualización de la información en cada *topic* y partición. Aquí, Spark Streaming crea tantas particiones RDD como particiones en Kafka y accede a ellas de forma paralela. Como penalización, el modo directo no es tan fiable como el modo tradicional de consumir información de Kafka.

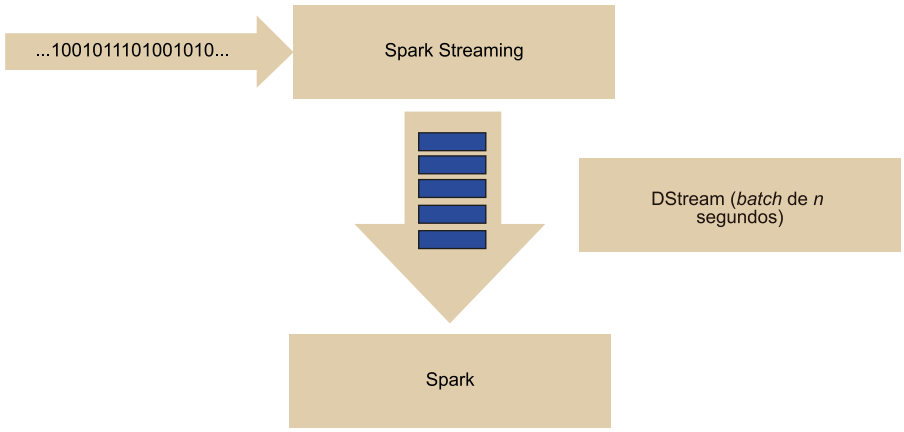
3.3. Spark Streaming

La última gran API de Spark es la Spark Streaming. El objetivo de esta API es el procesado de un flujo continuo de datos que son consumidos por la aplicación. En esta API la capa de abstracción para trabajar con el flujo continuo de datos la ofrece el llamado DStream (*Discretized Stream*) que es una secuencia de RDD representando el flujo continuo de datos como pequeños procesos *batch*. Así, el DStream en Spark Streaming toma el papel del RDD en el núcleo de Spark.

3.3.1. ¿Cómo funciona el *streaming* en Spark?

Spark convierte el flujo de datos en pequeños RDD de *n* segundos, con lo que en realidad está operando con pequeñas operaciones *batch* y procesa de forma separada cada uno de estos RDD, como se muestra en la figura 9.

Figura 9. Paquetización de un stream de datos en RDD



La fuente de datos para la creación de un DStream puede ser un *socket*, fuentes compatibles como Flume, Kafka, Twitter o incluso ficheros en el HDFS. En este caso, se monitoriza el contenido de un directorio HDFS y al detectar cambios en su contenido se realiza el procesado.

Socket

Un *socket* es un canal de comunicaciones entre dos puntos definido con las direcciones IP y los números de puerto de los sistemas a comunicar.

El DStream ofrece muchas de las funcionalidades ya disponibles en las RDD de Spark, como las transformaciones y las acciones, que en el caso de DStreams se las llama **operaciones de salida**. De forma similar a los RDD, una transformación aplicada a un DStream produce un nuevo DStream.

Existen tres modos de procesado de DStreams:

- Procesado de cada RDD de forma independiente.

- Acumulado de datos a medida que los RDD se van procesando.

- Modo *sliding window*. En este caso, un conjunto de RDD se procesan, devolviendo un resultado. En una nueva iteración, se procesa el mismo conjunto de RDD, con la diferencia que se han añadido nuevos RDD generados por el *stream* y se han eliminado los más antiguos, produciendo el efecto de una ventana que se va desplazando en el tiempo. Todos los parámetros de tiempo que dan forma a esta ventana son configurables por el usuario.

Spark Streaming puede ser usado con las herramientas del ecosistema para proveer datos como Apache Flume o Kafka y es compatible con el resto de las API de Spark, lo que ofrece un soporte importante a la API.

3.4. Apache Storm

Apache Storm es un *framework* competidor de Spark Streaming. Storm ofrece una latencia muy baja (menor que la de Spark) con capacidad de análisis distribuido de datos en tiempo real y *machine learning*. Además, es una solución altamente escalable y tolerante a fallos.

Desde un punto de vista de arquitectura, Storm está compuesto por tres componentes:

- ***Master node*, Nimbus**. Similar a un monitor de tareas o al *Master Node* y *Job Tracker* de Hadoop, distribuye código a través del clúster, asigna tareas y monitoriza el estado del clúster, pudiendo reasignar tareas en caso de fallos en el clúster.

- **Supervisores.** Gestionan la ejecución de tareas individuales en los nodos por los *workers* que son procesos que corren en una JVM con múltiples hilos o *threads* y que realizan las tareas de procesado en *streaming*.

- **Zookeeper.** No es un componente propiamente dicho de Storm, pero es necesario ya que es el componente que coordina el clúster desde un punto de vista de sincronía entre el Nimbus y los supervisores.

JVM

La Java Virtual Machine es una aplicación que interpreta y ejecuta programas escritos en el lenguaje de programación Java.

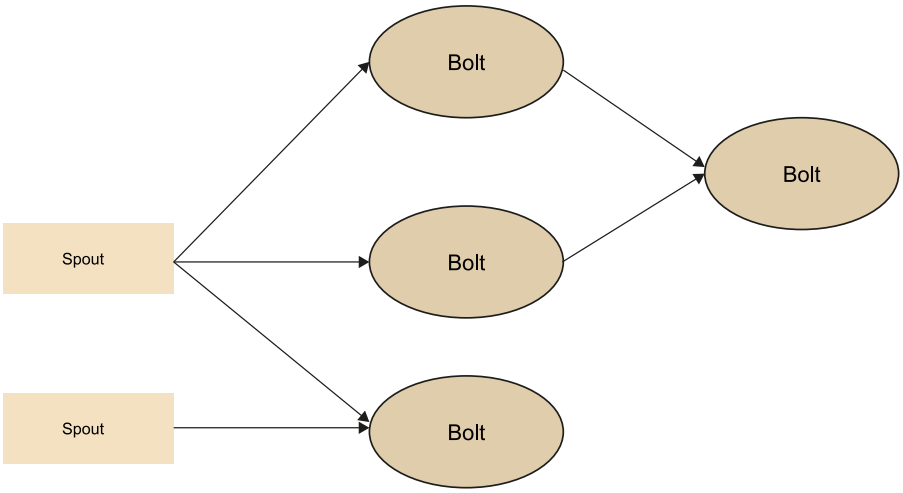
Desde un punto de vista de procesado, Strom procesa los datos en forma de tuplas con pares "clave,valor", ya estudiadas anteriormente. El valor puede ser cualquier tipo de dato, primitivo u objeto serializado.

Storm introduce los *spouts*, que son un *wrapper* que añade una estructura a una fuente de datos en *streaming*, como podría ser una fuente de mensajes Kafka o datos de Twitter, para poder ser procesados por el sistema. El *bolt* es el encargado de consumir las tuplas que el *spout* proporciona, los procesa y los envía a la salida del sistema o a otro *bolt*. Es recomendable que cada *bolt* realice una sola tarea lo que mejora la eficiencia y escalabilidad. Los *spouts* y los *bolts* crean la llamada topología de Storm, similar a un grafo, representada en la figura 10.

Wrapper

Un *wrapper* es un adaptador que permite transformar los mecanismos de acceso o incluso dotar de funcionalidad extra a una función para que sea accesible para otras funciones que no son compatibles con esta.

Figura 10. Diagrama funcional de Storm.



La funcionalidad para trabajar con Storm la provee la API Trident. Esta API permite trabajar con los datos de Storm como si fueran procesos en *micro-batch*, proporcionando una capa abstracta de alto nivel, similar a la API de Spark Streaming. Gracias a esta abstracción, puede trabajarse con las tuplas y realizar filtros, agregaciones, etc.

Storm ofrece buenas funcionalidades en cuanto a fiabilidad del sistema garantizando que cada paquete de datos del *stream* se va a leer como mínimo una vez. Además, a través de la API Trident, se garantiza que un paquete de datos se leerá solo una vez. Funcionalidad que también ofrece Spark Streaming. Sin embargo, Spark no puede garantizar que en caso de fallo de un nodo no se vayan a perder datos que ese nodo tenga almacenados en memoria es ese momento. Pueden almacenarse datos en HDFS de forma temporal para garantizar resistencia a fallos, pero penaliza el rendimiento del sistema. Storm garantiza la lectura de todos los *streams* de entrada. Sin embargo, la fiabilidad del sistema va a depender de la fiabilidad de la fuente continua de datos. Como ejemplo, Kafka es fiable, mientras que Twitter no es fiable.

Storm es compatible con muchos lenguajes de programación como Java, Scala, Python, Ruby o Clojure entre otros. Sin embargo, Spark provee una mayor integración con todo el resto del *framework* Spark. Hasta el momento, Storm tiene una base de usuarios importante y ha dejado de ser un proyecto en incubación. Tanto Hortonworks como MapR lo integran en su distribución, pero Cloudera, no. Por tanto, es difícil saber cuál será el *framework* predominante para procesar datos en *streaming* a medio y largo plazo. A pesar de la comunidad de usuarios de Storm, es una herramienta especializada más dentro del ecosistema *big data* y Hadoop, mientras que Spark Streaming cuenta con el apoyo de todo el framework Spark y está plenamente integrado en él.

4. Otras herramientas *big data*

A continuación vamos a presentar brevemente algunas herramientas que merecen una especial atención en el entorno *big data*, ya sea por su peso histórico (Mahout, principalmente) o por ser herramientas en desarrollo pero que introducen nuevas funcionalidades que potencialmente pueden tener mucha aceptación en el mundo *big data*.

4.1. Mahout

Mahout es un proyecto de Apache cuyo objetivo es ofrecer capacidad de realizar operaciones de Machine Learning en entornos distribuidos sobre Hadoop. Así, las operaciones que se han de realizar se dividen en tareas MapReduce que, en muchas ocasiones, pueden ser una secuencia de ellas. Hasta la aparición de Spark, Mahout era una de las mejores plataformas para este tipo de computación. Sin embargo, debido a la complejidad de las tareas que se realizan y la rigidez del modelo MapReduce, el tiempo de cálculo era muy grande y la necesidad de secuenciar cadenas de operaciones MapReduce generaba una gran cantidad de datos intermedios (cabe recordar que MapReduce es un modelo de computación en el que el intercambio de datos se basa en ficheros). Spark mejoró en órdenes de magnitud los tiempos de cálculo para realizar la misma operación, dejando a Mahout en una situación que ha obligado a replantearse su arquitectura.

Actualmente Mahout ya trabaja, a pesar de ser versiones en desarrollo, sobre Spark a través del nuevo entorno Samsara.

Enlace de interés

Puede visitarse la página web http://mahout.apache.org/users/basics/algorithms.html en la que se describen los algoritmos que Mahout implementa y ver cómo aquellos que se implementaron para MapReduce están *deprecated*, centrando ahora su desarrollo hacia Spark.

4.2. BlinkDB

BlinkDB es una herramienta todavía en desarrollo, pero que introduce una interesante novedad. Permite realizar consultas similares a SQL (casi idénticas a las utilizadas en Hive y SparkSQL) sobre entornos distribuidos aceptando márgenes de error. En otras palabras, podemos realizar una consulta en la que aceptamos que los resultados devueltos tienen un margen de error. El beneficio principal es que reduce de forma drástica los tiempos de computación. Así, para situaciones en las que se está realizando exploración de datos distribuidos, el científico de datos puede tolerar cierto margen de error si a cambio reducimos el tiempo de consulta en varios órdenes de magnitud.

Deprecated

Deprecated quiere decir que una utilidad no tienen continuidad en su desarrollo.

Veamos un ejemplo de consulta con errores:

```
SELECT avg(sessionTime)
FROM Table
WHERE city='San Francisco'
```

```
ERROR 0.1 CONFIDENCE 95.0%
```

También permite devolver todos los resultados que cumplan los criterios de consulta, pero limitando el tiempo. Así no es necesario cargar todos los resultados si una muestra ya es representativa para el trabajo que se está realizando. Por tanto, cuanta mayor resolución queramos para nuestra consulta, más tiempo de ejecución va a requerir. Sin embargo, la mejora del rendimiento es exponencial cuanta menor resolución se pida lo que agiliza el proceso de exploración de datos por parte del científico de datos.

BlinkDB es compatible con SparkSQL y Hive y está en desarrollo por el equipo del AMPlab, también responsables del desarrollo de Spark.

4.3. Apache Flink

Apache Flink es otra plataforma para procesar datos en *streaming* de baja latencia, similar a Spark Streaming o Storm. Sus funcionalidades son muy parecidas a Spark Streaming. Es capaz de procesar datos en streaming o fuentes estáticas como ficheros de un sistema HDFS. Tiene sus propias API para trabajar sobre colecciones (pueden hacerse transformaciones, agrupaciones, filtrado, etc.). También tiene librerías de Machine Learning, API para trabajar con grafos y API para trabajar con datos estructurados con funcionalidades de consulta SQL.

Comparándolo con las dos herramientas presentadas anteriormente para *streaming* (Spark y Storm), Flink se sitúa entre ambas aproximaciones:

- Ofrece capacidades de procesar datos en *batch*, algo que no es posible en Apache Storm.
- Es una plataforma que trabaja en *streaming* puro, no como Spark que trabaja en mini-*batches*.

Es todavía pronto para saber cuál va a ser la plataforma de *streaming* predominante.

Ejemplo de uso de Apache Flink

El ejemplo presentado en la subapartado 1.3 de procesado de datos en CEP se desarrolla en detalle en esta URL utilizando Apache Flink: https://flink.apache.org/news/2016/04/06/cep-monitoring.html

Enlace de interés

En el siguiente enlace se ofrece una interesante comparativa de rendimiento entre Flink, Spark y Storm:

https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

4.4. Elasticsearch

Elasticsearch es una herramienta que permite indexar un gran volumen de datos y posteriormente hacer consultas variadas como búsquedas aproximadas, búsquedas de texto completo, texto resaltado, etc. Está basado en Lucene, un motor de búsqueda de texto escrito en Java.

La velocidad de las búsquedas es muy rápida al estar los datos indexados. Su funcionalidad se ofrece a través de una interfaz REST, intercambiando datos con formato JSON, ocultando el núcleo de Lucene. La interfaz REST es accesible con cualquier lenguaje como Java, Python o PHP entre otros.

Elasticsearch puede trabajar con Hadoop y Spark.

Enlace de interés

Puede verse un ejemplo de la sintaxis y consulta en la siguiente URL:

https://www.elastic.co/guide/en/elasticsearch/reference/current/search-uri-request.html

5. Soluciones Vendor

A continuación listaremos cuatro soluciones propuestas por algunos de los principales proveedores de soluciones hardware y software del mercado, como son Amazon, IBM, Microsoft y Hewlett-Packard. El modelo propuesto por todos ellos es parecido: uso de clústeres desplegables bajo demanda en la nube basados en tecnologías Hadoop y Spark.

5.1. Amazon

Amazon es una de las empresas que más fuerte ha apostado por las plataformas de procesado distribuido, integrando un paquete de servicios de procesado en su plataforma Amazon WS (Web Services). Concretamente, se incluye Amazon EMR (Amazon Elastic MapReduce), una plataforma que permite la creación dinámica de clústeres Hadoop, incluyendo todas las herramientas ya presentadas para el procesado *big data* como Hive, Impala, Spark, etc.

La capacidad de cálculo la proporciona el servicio EC2 (Elastic Compute Cloud), que permite la creación de clústeres virtuales llamados **instancias**, totalmente personalizado a las necesidades del usuario (plataformas Linux o Windows, y recursos de memoria y computación). La capacidad de almacenamiento la provee el servicio S3 (Simple Storage Service).

Así, la solución completa de procesado podría estar formada por el sistema de almacenamiento S3, mientras que EMR (formado por instancias EC2) lee y escribe datos en el sistema S3.

Desde un punto de vista tarifario, Amazon cobra por el consumo de los recursos. Cuando una tarea de procesado finaliza, el clúster se apaga.

5.2. IBM Analytics

IBM también ofrece soluciones para procesado de *big data* a través de la plataforma IBM Analytics, una suite de herramientas para tratamiento de datos *big data* como: IBM SPSS Modeler, para desarrollar modelos predictivos; IBM Cognos Business Intelligence, para funcionalidades de Business Intelligence; herramientas de gestión de contenidos (ECM - Enteprise Content Management); almacenamiento de datos en el Cloud, (similar al S3 de Amazon); Data warehousing; (IBM Open Platform) servicios de despliegue de clústeres Hadoop de forma dinámica en la que incluye varias de las aplicaciones ya descritas en este tema, como Spark, Kafka, Flume, Pig, Hive, etc. En el núcleo de su plataforma, IBM ha hecho una apuesta por Apache Spark como motor de procesado de datos distribuidos.

5.3. Google

Google también ofrece servicios de *cloud* para el despliegue de clústeres Hadoop y Spark. El Cloud DataProc, solución integrada en el Google Cloud Platform, permite desplegar clústeres Hadoop y Spark incluyendo las herramientas del ecosistema ya estudiadas (Pig, Hive, etc.). Como en todos los productos y herramientas de Google, su solución está muy enfocada a la facilidad de uso.

Además, ha implementado servicios adicionales para consultas SQL como Google BigQuery, similar en funcionalidades a SparkSQL, o herramientas para la definición de *pipelines* como el Cloud Dataflow.

Su plataforma incluye un paquete de Machine Learning, todavía en fase de desarrollo, que incluye una tecnología propia llamada TensorFlow. TensorFlow propone un modelo de procesado distribuido basado en un grafo en el que los vértices son operaciones y los ejes son tensores (vectores multidimensionales). El grafo define el *dataflow* completo y puede ejecutarse en CPU o en GPU.

GPU

GPU o unidad de procesamiento gráfico (*Graphics Processor Unit*) es un coprocesador dedicado al procesamiento de gráficos.

Desde el punto de vista tarifario, Google factura por uso.

5.4. Microsoft Azure / HDInsight

Dentro del paquete de servicios Microsoft Azure, que es una solución para ofrecer servicios a nivel empresarial en la nube, Microsoft incluye su paquete Microsoft HDInsight. HDInsight es una distribución de Hadoop en la nube que ofrece servicios de procesado integrando la mayoría de las herramientas presentadas en este curso, como operaciones *batch* con MapReduce, SDQL con Hive, NoSQL con HBase, Spark, Storm, etc. En este caso se utiliza la distribución de Hortonworks de Hadoop.

El sistema ofrece capacidad de escalado en la nube, creando clústeres Hadoop bajo demanda. Su uso se tarifica mensualmente dependiendo de la configuración deseada y de los servicios a utilizar en el clúster.

5.5. HPE Vertica

También podemos incluir a Hewlett-Packard Enterprise (HPE), ya que también comercializa soluciones *big data*. Como fabricante de hardware, HPA ofrece soluciones de infraestructura especialmente diseñadas para desplegar clústeres Hadoop, soportando las distribuciones de Hortonworks y Cloudera, con especificaciones de servidores HPE. Desde el punto de vista de software de análisis de datos en entornos distribuidos, comercializa la solución Vertica, basada en un motor de consulta SQL orientado a columna que permite realizar consultas sobre clústeres Hadoop. La solución Vertica OnDemand ofrece capacidad de análisis de *big data* en la nube.

Resumen

Al inicio de este módulo didáctico hemos introducido los diferentes modos de procesamiento de datos en *big data*, haciendo especial hincapié en los modos *batch* y *streaming*, ilustrado con algunos ejemplos.

En el segundo apartado se han presentado cuáles son los *frameworks* más utilizados en la actualidad en el panorama *big data* para realizar procesado de datos en modo *batch*. Prestando especial atención al *framework* Hadoop y sus servicios, así como Apache Spark, la plataforma de referencia en la actualidad para procesado distribuido.

A continuación, en el tercer apartado hemos presentado las herramientas que ofrecen funcionalidades de procesado de datos en *streaming*.

Finalmente, los apartados 4 y 5 hemos presentado otras herramientas disponibles en el mercado, desde proyectos en desarrollo y que están teniendo relevancia en el ecosistema *big data* (apartado 4), hasta las soluciones que ofrecen las grandes empresas tecnológicas, como por ejemplo IBM, Google o Amazon (apartado 5).

Glosario

commodity hardware *m* Dispositivos como ordenadores de sobremesa o estaciones de trabajo.

damping *m* Probabilidad de que un usuario vaya a dejar de navegar por las páginas referenciadas entre ellas.

dataflow *m* Conjunto de tareas encadenadas que realizan unas operaciones sobre unos datos. Los datos de salida de una tarea son la entrada a la siguiente tarea.

datawarehouse *m* Sistema que permite el almacenamiento de información de forma homogénea y fiable, orientado a la consulta de datos y a su tratamiento de forma jerarquizada.

deprecated *m* Que no tienen continuidad en su desarrollo.

equipo *mainframe* *m* Gran equipo de cómputo, rápido y con un coste elevado, utilizado principalmente por grandes empresas u organizaciones.

GPU *f* Véase **unidad de procesamiento gráfico**

Grafo *m* Representación gráfica de un conjunto de objetos llamados nodos o vértices unidos entre ellos mediante enlaces, también llamados aristas.

hilo *m* Proceso que se ejecuta en paralelo o de forma asíncrona a la de otros procesos que forman una aplicación.
en **thread**

Information Leak Prevention *m* Mecanismo y políticas que evitan que información de carácter confidencial de una organización pueda ser accesible a personas ajenas a esta.

Java Virtual Machine *f* Aplicación que interpreta y ejecuta programas escritos en el lenguaje de programación Java.
sigla **JVM**

overhead *m* Exceso en el uso de cualquier recurso de computación ya sea memoria, tráfico de datos, tiempo de cálculo, etc. que condiciona y puede penalizar su rendimiento.

PageRank *m* Algoritmo utilizado por Google para clasificar páginas web en base a las referencias cruzadas entre ellas. La idea que subyace es que las páginas web más importantes tienen más referencias de otras páginas web.

pipeline *m* Transformación de un flujo de datos en un proceso formado por varias tareas ejecutadas de forma secuencial, siendo la entrada de cada una la salida de la anterior.

propiedad asociativa *f* Propiedad matemática tal que $(a + b) + c$ es igual a $c + (b + a)$.

Shortest Path *m* Algoritmo para encontrar la distancia mínima entre dos vértices en un grafo, definida como aquella en la que la suma de los pesos de los vértices para unir dichos vértices es mínima.

socket *m* Canal de comunicaciones entre dos puntos definido con las direcciones IP y los números de puerto de los sistemas que se han de comunicar.

SQL-92 *m* Tercera revisión del estándar SQL (*Structured Query Language*).

super-paso *m* Cada una de las iteraciones de Giraph en las que los valores de los vértices y las aristas se actualizan.

thread *m* Véase **hilo**.

throughput *m* Volumen de datos procesados por unidad de tiempo.

workflow *m* Conjunto de tareas, sus interacciones, eventos e intercambio de información que, en su conjunto, realizan una operación completa.

wrapper *m* Adaptador que permite transformar los mecanismos de acceso o incluso dotar de funcionalidad extra a una función para que sea accesible para otras funciones que no son compatibles con esta.

unidad de procesamiento gráfico *f* Coprocesador dedicado al procesamiento de gráficos.

en Graphics Processor Unit
sigla **GPU**

Bibliografía

Dean, Jeffrey; Ghemawat, Sanjay (2004, diciembre). *MapReduce: Simplified Data Processing on Large Clusters*. OSDI'04: Sixth Symposium on Operating System Design and Implementation. San Francisco (CA).

Ghemawat, Sanjay; Gobioff, Howard; Leung, Shun-Tak (2003, octubre). *The Google File System*. 19th ACM Symposium on Operating Systems Principles, Lake George (NY).

Luckham, David C. (2001). *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co.: Boston (MA). ISBN:0201727897

Malewicz, Grzegorz; Austern, Matthew H.; Bik, Aart J. C; Dehnert, James C.; Horn, Ilan; Leiser, Naty; Czajkowski, Grzegorz (2010). *Pregel: A System for Large-Scale Graph Processing*. Google, Inc.

Redekopp, Mark; Simmhan, Yogesh; Prasanna, Viktor K. (2004, diciembre). *Optimizations and Analysis of BSP Graph Processing Models on Public Clouds*. Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium, May 2013, pp. 203 - 214, ISSN:1530-2075, ISBN: 978-1-4673-6066-1, INSPEC Accession Number: 13683465. University of Southern California. San Francisco (CA)