Sistemas de procesamiento distribuido en *big data*

PID_00286724

Francesc Julbe López

Tiempo mínimo de dedicación recomendado: 3 horas

Universitat Oberta de Catalunya

Francesc Julbe López

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: Albert Solé Ribalta

Primera edición: septiembre 2021

Av. Tibidabo, 39-43, 08035 Barcelona
Autoría: Francesc Julbe López
Producción: FUOC

Índice

Introducción

En otros módulos nos hemos centrado en preparar y almacenar la información para iniciar la etapa de procesado, en este describiremos dos sistemas muy extendidos que os permitirán procesar esta información de manera distribuida dentro del ecosistema del *big data*: MapReduce y Spark. Además, también introduciremos cómo se pueden aplicar las ideas del cálculo distribuido sobre arquitecturas basadas en GPU.

1. Paradigma MapReduce

MapReduce es un modelo de programación introducido por Google y divulgado en un artículo de Jeffrey Dean y Sanjay Ghemawat publicado en el 6.º Simposio sobre Diseño e Implementación de Sistemas Operativos (OSDI) celebrado en San Francisco en 2004.[1] Nació para dar soporte a la computación paralela sobre grandes volúmenes de datos, con dos características principales:

(1)https://research.google/pubs/pub62/

- uso de clúster de ordenadores y
- hardware no especializado.

El nombre de este sistema está inspirado en las denominaciones de sus dos métodos o funciones de programación principales: *map* y *reduce*, que veréis a continuación. MapReduce ha sido adoptado mundialmente, gracias a que existe una implementación *open source* denominada Hadoop, desarrollada por Yahoo!, que permite usar este paradigma mediante el lenguaje de programación Java. Lo primero que debéis tener en cuenta cuando hablamos de este modelo de cálculo es que algunos potenciales análisis se podrán realizar de forma eficiente. Concretamente, solo son aptos aquellos que pueden calcularse como combinaciones de las operaciones de *map* y de *reduce*.

En general este sistema funciona muy bien para calcular agregaciones, filtrados, procesos de manipulación de datos, estadísticas, etc., operaciones todas ellas fáciles de paralelizar, no requieren de un procesamiento iterativo y es necesario compartir los datos entre todos los nodos del clúster. En la arquitectura MapReduce todos los nodos del clúster se consideran *workers* (trabajadores), excepto uno que toma el rol de *master* (maestro). El maestro se encarga de recopilar trabajadores en reposo (es decir, sin tarea asignada) para asignarles tareas específicas de *map* o de *reduce*. Un *worker* solo puede tener tres estados: reposo, trabajando, completo (*idle*, *in-progress* y *completed*, en inglés). El rol de maestro se asigna de manera aleatoria en cada ejecución. Cada vez que un nodo acaba una tarea, tanto un *map* como un *reduce*, guarda el resultado en el sistema de archivos para garantizar la consistencia y poder continuar la ejecución en caso de producirse un fallo en el hardware. Como hemos indicado, el modelo se basa en dos operaciones: *map* y *reduce*.

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

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

Todo el intercambio de datos entre tareas utiliza estructuras llamadas parejas <clave, valor> (<*key*, *value*> *pairs*, en inglés) o tuplas.

En el ejemplo siguiente se va a mostrar, desde un punto de vista funcional, en qué consiste una tarea MapReduce.

Ejemplo conceptual de proceso MapReduce

Imaginad que tenéis tres ficheros con los datos siguientes:

```
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
```

y deseáis obtener cuántas personas hay en cada ciudad.

Para responder esa pregunta definiréis una tarea *map* que leerá las filas de cada fichero y «etiquetará» cada una en función de la ciudad que aparece, que es el tercer campo de cada una de las líneas que siguen la estructura: nombre, edad, ciudad.

Para cada línea, la tarea *map* os devolverá una tupla de la forma: <ciudad, cantidad>. Así, al final de la ejecución de la tarea *map*, en cada fichero tendréis:

```
• 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 veis, vuestras tuplas están formadas por una clave (*key*), que es el nombre de la ciudad, y un valor (*value*), que representa el número de veces que aparece en la línea, que siempre es 1 (cada valor se cuenta a sí mismo como 1).

La tarea *reduce* se ocupará de agrupar los resultados según el valor de la clave. Así, recorrerá todas las tuplas agregando los resultados por una misma clave y os devolverá el siguiente resultado:

```
(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 módulos anteriores, una operación MapReduce es un procesado en modo de lotes (*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 ya que se van a realizar, como veréis, en paralelo por cada uno de los ficheros de datos y no es posible controlar el orden de procesado.

Propiedad conmutativa

Una operación es conmutativa si cumple la propiedad: (a + b) = (b + a).

Propiedad asociativa

Una operación + es asociativa si cumple la propiedad (a + b) + c = a + (b + c). Por ejemplo, la suma es una operación asociativa, mientras que la resta no lo es.

2. Implementación MapReduce en Hadoop

Generalmente, como hemos comentado, un clúster está formado por varios nodos, controlados por un nodo maestro. Cada nodo almacena datos localmente y son accesibles mediante un sistema de ficheros (típicamente es HDFS, aunque no es un requisito necesario). Los ficheros se distribuyen de forma balanceada entre todos los nodos. Así, 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.

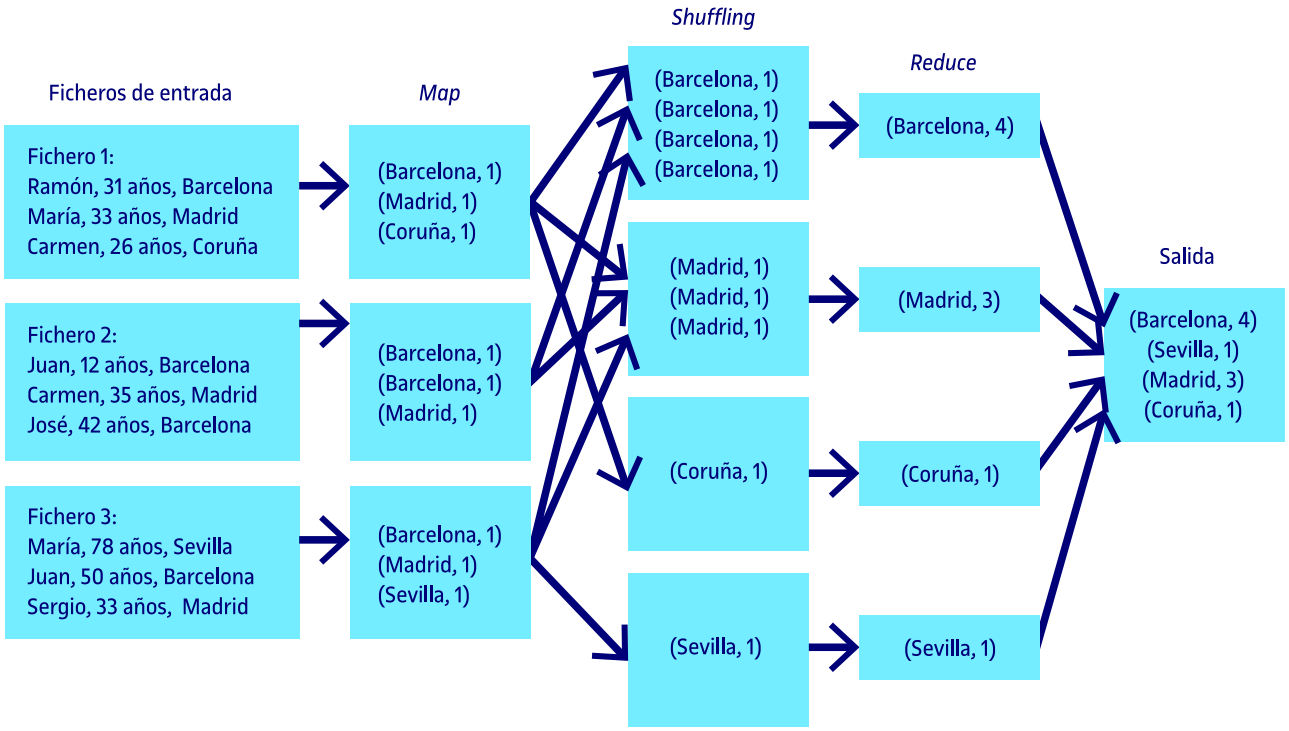
Cuando la fase de *map* ha finalizado, los resultados intermedios (tuplas <clave, valor>) deben intercambiarse entre las máquinas para enviar todos los valores con la misma clave a un solo *reduce*. Las tareas *reduce* también se ejecutan en los mismos nodos que las *map*, y este el único intercambio 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 en la fiabilidad de una tarea MapReduce, ya que, si un nodo falla, el sistema reinicia sus tareas sin que el estado del procesamiento en otros nodos dependa de él.

Aunque transparente al usuario, en el proceso de intercambio de información entre el *maper* y *reducer* debéis introducir dos nuevos conceptos: partición (*partition*) y *shuffle*.

Cuando una tarea *map* ha finalizado en algunos nodos, otros nodos todavía pueden estar realizando tareas del mismo tipo. 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 la que fuera su tarea *mapper*. Así, todos los *mapper* deben ponerse de acuerdo en dónde 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 muestra de forma gráfica el ejemplo descrito anteriormente.

Figura 1. Diagrama de flujo con las etapas *map* y *reduce* del ejemplo descrito



Fuente: Casas Roma, Nin Guerrero y Julbe López (2019)

2.1. Limitaciones de Hadoop

Hadoop solucionaba la problemática del cálculo distribuido utilizando las arquitecturas predominantes hace una década, sin embargo, presentaba importantes limitaciones:

- Es complicado aplicar el paradigma MapReduce en muchos casos, ya que una tarea debe descomponerse en subtareas *map* y *reduce*. En algunos casos, no es fácil esta descomposición y en otros deriva en algoritmos muy ineficientes.

- 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.

- Es un entorno esencialmente basado en Java que requiere la descomposición en tareas *map* y *reduce*. No obstante, esta aproximación al procesamiento de datos resultaba muy dificultosa para el científico de datos sin conocimientos de Java, de modo que diversas herramientas han aparecido para flexibilizar y abstraer a la comunidad no especializada del paradigma MapReduce y su programación en Java. Entre estas herramientas podemos citar:
  - Apache Hive o Apache Impala[2] para realizar consultas del tipo SQL sobre datos almacenados en HDFS.

[(2)]https://impala.apache.org

[(3)]https://pig.apache.org/

- Apache Pig[3] para definir cadenas de procesado sobre datos distribuidos.

Estas herramientas no son actualmente de uso muy extendido y en gran medida han sido sustituidas por Apache Spark principalmente para tareas ETL y de consulta SQL sobre datos en entornos distribuidos.

A modo de conclusión, cabe destacar que MapReduce fue un paradigma muy novedoso para poder explorar grandes archivos en entornos distribuidos. Es un entorno de trabajo que puede desplegarse, en su implementación Hadoop, en equipos de bajo coste o convencionales (*commodity hardware*) y esto lo hace muy atractivo. Varias herramientas que ofrecen funcionalidades diversas en el ecosistema de datos masivos como Hive o Sqoop se han desarrollado a partir del framework MapReduce. Sin embargo, este entorno de trabajo presenta algunas limitaciones importantes:

- Utiliza un modelo forzado para cierto tipo de aplicaciones y algoritmos. Debido al limitado conjunto de operaciones, la adaptación de algoritmos os puede obligar a crear etapas adicionales *map* o *reduce* para ajustar el algoritmo al modelo. Al ser forzadas, estas etapas suelen emitir resultados temporales un tanto extraños o poco útiles para el resultado final, incluso creando funciones *map* o *reduce* de tipo identidad. Es decir, que no son necesarias para la operación que se va a realizar, pero que se deben crear para adecuar el cálculo al modelo.

- MapReduce es un proceso basado en ficheros, lo que significa que el intercambio de datos se realiza utilizando ficheros. Esto produce un elevado flujo de lectura y escritura de datos a disco, elemento que afecta altamente su velocidad y rendimiento general si un procesamiento concreto está formado por una cadena larga de procesos MapReduce secuenciales.

Para solucionar estas limitaciones apareció Apache Spark, que cambia el modelo de ejecución de tareas, y lo hace más ágil y polivalente.

3. Apache Spark

Como habéis visto en el apartado anterior, MapReduce es un potente algoritmo para procesar datos distribuidos, altamente escalable y relativamente simple. Sin embargo, presenta algunas debilidades en cuanto a rendimiento y versatilidad para implementar algoritmos más complejos. Así, el proyecto de Spark se centró desde el comienzo en aportar una solución factible a estos defectos, mejoró el comportamiento de las aplicaciones que hacen uso del entorno de trabajo MapReduce, y aumentó su rendimiento considerablemente.

Apache Spark es un sistema de cálculo distribuido para el procesamiento de grandes volúmenes de datos y que gracias a su llamada *interactividad* hace que el paradigma MapReduce ya no se limite a las fases *map* y *reduce* y que podáis aplicar muchas más operaciones de utilidad como *mappers*, *reducers*, *joins*, *groups by*, *filter*, etc.

La principal ventaja de Spark respecto a Hadoop es que guarda todas las operaciones sobre los datos en memoria.

Tanto Hadoop como Spark están escritos en Java (y Scala, lenguaje que se ejecuta sobre la máquina virtual de Java) para aprovecharse de las llamadas RMI[4] (Remote Method Invocation) nativas de Java y que se desarrollaron para comunicar de forma eficiente diversos nodos de un clúster. Las llamadas RMI ofrecen un mecanismo que permite invocar un método a un objeto Java remoto, proporcionar un mecanismo simple para la comunicación de servidores en aplicaciones distribuidas. Por medio de RMI, cualquier objeto remoto estará accesible por la red y el programa permanecerá a la espera de peticiones en un puerto TCP. A partir de ese momento, un cliente puede conectarse e invocar los métodos proporcionados por el objeto. Esto es justo lo que hace el nodo *master* en MapReduce para enviar funciones *map()* o *reduce()* a los nodos *worker*. Spark utiliza el mismo sistema para enviar las transformaciones y acciones que se tienen que aplicar a los RDD, el objeto principal que usa Spark para lanzar tareas distribuidas. Entre las principales funcionalidades de Apache Spark destacamos:

(4)https://www.oracle.com/java/technologies/ javase/remote-method-invocation-home.html

- **SparkSQL**: API para trabajar con datos distribuidos utilizando estructuras abstractas, llamadas *dataframes*. Estas son similares a los *dataframes* de R[5] o Pandas[6] de Python.

(5)https://www.r-project.org

(6)http://pandas.pydata.org

- **MLlib**: Spark también ofrece una API para realizar tareas de aprendizaje automático, de la que hablaremos con más detalle en subapartados posteriores.

- **GraphX**: API para desarrollar aplicaciones de análisis de datos basados en grafos (*graph mining*).

- **Spark Streaming**: API para desarrollar aplicaciones de análisis de datos masivos en flujo (*streaming data*).

Además, Spark es compatible con todos los servicios del ecosistema Hadoop.

Este apartado se centrará en el modo interno de funcionamiento de Spark y cómo paraleliza sus tareas, para convertirlo en el entorno más popular de procesado en entornos de datos masivos.

3.1. *Resilient distributed dataset* (RDD)

Para entender el procesado distribuido de Spark, debéis conocer primero uno de sus componentes principales, el *resilient distributed dataset* (RDD en adelante) sobre el que se basa su estructura de paralelización de trabajo.

Un RDD es una entidad abstracta que representa un conjunto de datos distribuidos en un clúster y que representa una capa de abstracción encima de todos los datos que componen vuestro corpus de trabajo, independiente de su volumen, ubicación en el clúster, etc. Las principales características de los RDD son las siguientes:

- Es inmutable, una condición importante porque prohíbe que uno o varios hilos (*threads*) actualicen el conjunto de datos con el que se trabaja facilitando el mantenimiento de la consistencia de los datos.

- Es *lazy loading*, solo se accede a los datos cuando es necesario. Veréis este punto en detalle cuando se presente el modelo de ejecución.

- Puede almacenarse en memoria caché.

El RDD se construye a partir de bloques de memoria distribuidos en cada nodo, y da lugar a las llamadas particiones. Este particionamiento lo realiza Spark y es transparente al usuario, aunque también se puede tener control sobre él. Si los datos se leen del sistema HDFS, cada partición se corresponde con un bloque de datos del sistema HDFS.

El concepto *partición* tiene un peso importante para comprender cómo se paraleliza en Spark. Como habéis visto, cuando se crea un fichero en HDFS, si este es mayor que el tamaño de bloque definido (típicamente 64 MB o 128 MB), se crea una partición para cada bloque (aunque el usuario puede definir el número de particiones por crear cuando está leyendo el fichero, si lo desea). Así, os encontráis con lo siguiente:

- Un RDD es un *array* de referencias a particiones en vuestro sistema.

- La partición es la unidad básica para entender el paralelismo en Spark y cada partición se relaciona con bloques de datos físicos en vuestro sistema de almacenaje o de memoria.

- Las particiones se asignan con criterios como la localidad de datos (*data locality*) o considerando la minimización de tráfico en la red interna.

- Cada partición se carga en memoria volátil (típicamente, RAM) antes de ser procesada.

Un RDD se puede construir a partir de:

- Creación de un objeto «colección» paralelo:

```
> val data = Array(1, 2, 3, 4, 5)
> val distData = sc.parallelize(data)
```

Esto creará una colección paralela en el nodo del controlador, hará particiones y las distribuirá entre los nodos del clúster, concretamente, en su memoria.

- Creación de RDD desde fuentes externas, como por ejemplo HDFS o S3 de Amazon. Se crearán particiones por bloque de datos (como se vio en el apartado de HDFS de este módulo) en los nodos donde los datos están físicamente disponibles.

- Al ejecutar cualquier operación sobre un RDD existente. Al ser inmutable, cuando se aplica cualquier operación sobre un RDD existente, se creará un nuevo RDD.

Sobre un RDD se pueden aplicar:

- **Transformaciones**. Una transformación sobre un RDD permite obtener otro RDD, cuyos elementos son el resultado de aplicar dicha transformación a cada uno de ellos. Así, una transformación podría ser una conversión de tipo de datos dentro del RDD o un filtro sobre los elementos que cumplan cierta condición.

- **Acciones**. Consisten en aplicar una operación sobre un RDD y obtener un valor como resultado, que dependerá del tipo de operación. El resultado de una acción no es un RDD.

Desde un punto de vista de modelo de ejecución, no se accede a los datos que componen un RDD hasta que se ejecuta sobre ellos una acción, siguiendo el modelo *lazy loading* ya mencionado. Esto es así ya que las **acciones** son las responsables de devolver resultados a los usuarios, si no le pedís al sistema obtener ningún resultado, ¿para qué es necesario ejecutar ningún proceso de lectura ni ninguna transformación?

Así, cuando se trabaja con RDD a los cuales se aplican **transformaciones**, en realidad se está definiendo una secuencia de procesos que hay que aplicar sobre él, que no se pondrán en marcha hasta la invocación de una acción.

Tomad la siguiente secuencia de comandos en Python para Spark:

```
> data = spark.textFile("hdfs://...")
> words =   data.flatMap(lambda line : line.split(" "))
                  .map(lambda word : (word, 1))
                  .reduceByKey(lambda a, b : a + b)
> words.count()
```

Esta porción de código muestra cómo se carga un RDD (variable *data*) a partir de un fichero ubicado en el sistema distribuido HDFS. A continuación, se aplican hasta tres transformaciones (*flatMap*, *map* y *reduceByKey*). No obstante, ninguna de estas transformaciones se va a ejecutar de forma efectiva hasta la llamada al método *count()*, que os devolverá el número de elementos en el RDD «words». Es decir, hasta que no se ejecute una acción sobre los datos, no se ejecutan las operaciones que os los proporcionan. Si hubiera algún error de programación en las transformaciones *flatMap*, *map* y *reduceByKey*, no se manifestarían en la ejecución hasta la invocación a la acción *count()*.

3.2. Modelo de ejecución Spark

Como hemos indicado, las trasformaciones definen un *pipeline* de tareas sobre un RDD que dará como resultado otro RDD. Esta secuencia de ejecución se basa en el llamado modelo de ejecución de Spark.

La secuencia de tareas que hay que realizar sobre un RDD define un *directed acyclic graph* (DAG, o grafo dirigido acíclico). Este grafo es un modelo de ejecución para sistemas distribuidos que se presenta como una alternativa al paradigma MapReduce, mucho más rígido. Mientras que MapReduce tiene solo dos pasos (*map* y *reduce*), lo que limita la implementación de algoritmos complejos, los DAG pueden tener múltiples niveles que pueden formar una estructura de árbol, con más funciones tales como *map*, *filter, union*, etc.

En un modelo DAG no hay ciclos definidos ni un patrón de ejecución previamente establecido, aunque sí existe un orden o una dirección de ejecución. Apache Spark representa estas secuencias de operaciones mediante un DAG. El DAG de Spark está compuesto por arcos (o aristas) y vértices, donde cada vértice representa un RDD y los ejes representan operaciones que aplicar sobre el RDD.

Cada RDD mantiene una referencia a uno o más RDD originales, junto con metadatos sobre qué tipo de relación tiene con ellos. Por ejemplo, si ejecutáis el código siguiente:

```
> val b = a.map()
```

El RDD *b* mantiene una referencia a su padre, el RDD *a*, llamamos a esta referencia RDD *lineage*.

3.2.1. Tipos de transformaciones

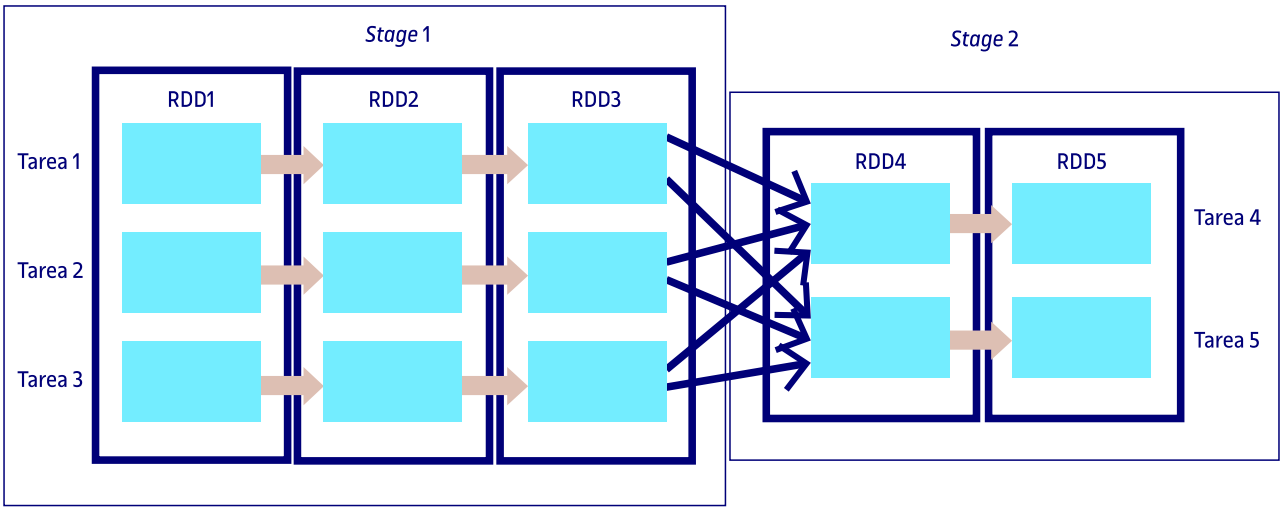
Las transformaciones sobre RDD pueden ser categorizadas como:

- ***Narrow operation***: se utiliza cuando los datos que se necesitan tratar están en la misma partición del RDD y no es necesario realizar una mezcla de dichos datos para obtenerlos todos. Algunos ejemplos son las funciones *filter, sample*, *map* o *flatMap*.

- ***Wide operation***: se utiliza cuando la lógica de la aplicación necesita datos que se encuentran en diferentes particiones de un RDD y es necesario mezclar dichas particiones para agrupar los datos necesarios en un RDD determinado. Ejemplos de *wide transformation* son: *groupByKey* o *reduceByKey*. Estas suelen ser las tareas de agregación y los datos provenientes de diferentes particiones se agrupan en un conjunto menor de particiones.

Al conjunto de operaciones sobre las que se realizan transformaciones preservando las particiones del RDD (*narrow*) se las llama *etapa* (*stage*). El final de un *stage* se presenta cuando se reconfiguran las particiones debido a operaciones de transformación tipo *wide*, que inician el llamado intercambio de datos entre nodos (*shuffling*), iniciando otro *stage*. Ved la figura 2.

Figura 2. Modelo de procesado de RDD en Spark, diferenciando tareas y *stages* en función del tipo de transformación que se va a aplicar

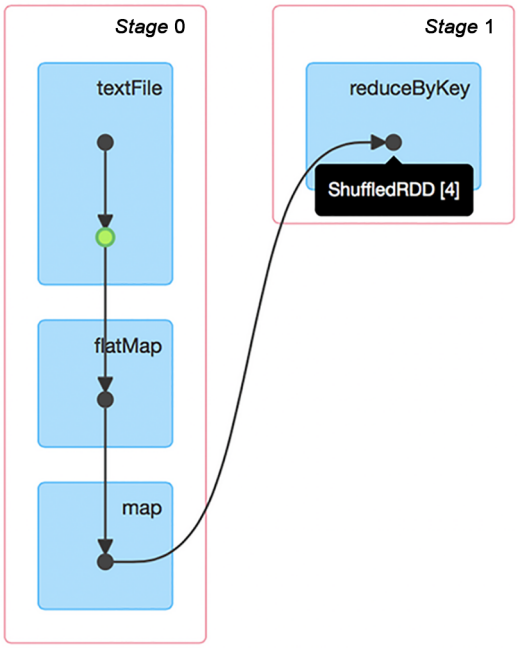


Fuente: Casas Roma, Nin Guerrero y Julbe López (2019)

Las limitaciones de MapReduce se convirtieron en un punto clave para introducir el modelo de ejecución DAG en Spark. Con la aproximación de Spark, la secuencia de ejecuciones, formadas por los *stages*, transformaciones y acciones queda reflejada en un grafo DAG que optimiza el plan de ejecución y minimiza el tráfico de datos entre nodos.

Cuando una acción es invocada para ser ejecutada sobre un RDD, Spark envía el DAG con los *stages* y las tareas que se van a realizar al DAG Scheduler, el cual elabora un plan de ejecución lógica (es decir, RDD de dependencias construidas mediante transformaciones) a un plan de ejecución físico (utilizando *stages* o etapas y los bloques de datos).

Figura 3. DAG de una ejecución en Spark, visualizándose los dos *stages*, diferenciando las *narrow transformations* (*faltMap* y *map*) de la *wide transformation* (*reduceByKey*) que fuerza un cambio de *stage*



Fuente: Databricks. <https://databricks.com/blog/2015/06/22/understanding-your-spark-application-through-visualization.html>

El DAG Scheduler es un servicio que se ejecuta en el *driver* de vuestro programa. Tiene dos tareas principales:

- Determina las ubicaciones ideales para ejecutar cada tarea y calcula la secuencia de ejecución óptima:
  - Primero se analiza el DAG para determinar el orden de las transformaciones con el fin de minimizar el mezclado de datos.
  - Seguidamente se da prioridad a las transformaciones *narrow* en cada RDD.
  - Finalmente se realizan la transformación *wide* a partir de los RDD sobre los que se han realizado las transformaciones *narrow*.
- Maneja los posibles problemas y fallos debido a una posible corrupción de datos en algún nodo: cuando un nodo queda no operativo (bloqueado o caído) durante una operación, el administrador del clúster asigna otro nodo para continuar el proceso. Este nodo operará en la partición particular del RDD y la serie de operaciones que tiene que ejecutar sin haber pérdida de datos.

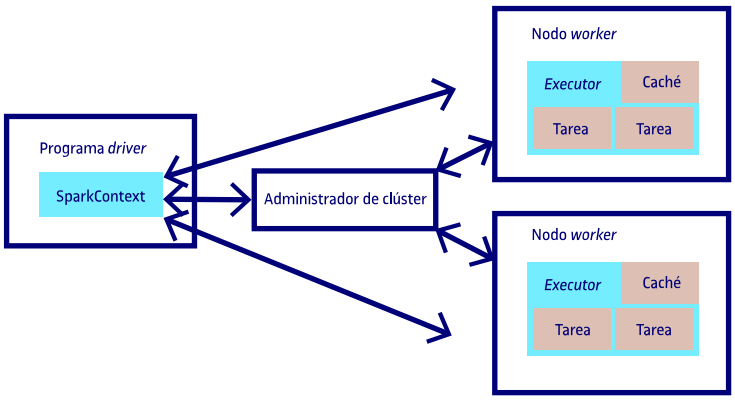
Así, es el DAG Scheduler el que controla la ejecución del DAG para cada trabajo, realiza un seguimiento de qué RDD y salidas de etapa se materializan y encuentra el camino mínimo para ejecutar los trabajos. A continuación, envía las etapas al Task Scheduler.

Las operaciones por ejecutar en un DAG están optimizadas por el DAG Optimizer, que puede reorganizar o cambiar el orden de las transformaciones si con ello se mejora el rendimiento de la ejecución. Por ejemplo, si una tarea tiene una operación *map* seguida de una operación *filter,* el DAG Optimizer cambiará el orden de estos operadores, ya que el filtrado reducirá el número de elementos sobre los que realizar el *map*.

Finalmente, el programa que ejecutará el código es llamado *driver*, y es el encargado de controlar la ejecución, mientras las transformaciones son las operaciones que se paralelizan y se ejecutan de forma distribuida entre los nodos que componen el clúster. Los responsables de la ejecución en los nodos son conocidos como *executors*. Es entonces cuando se accede a los datos distribuidos que componen el RDD. Así, si hay algún problema en alguna de las transformaciones solo será posible detectarla una vez se ejecute la acción que desencadena las transformaciones asociadas.

La figura 4 muestra los conceptos descritos anteriormente.

Figura 4. Arquitectura de funcionamiento de Spark, donde el programa *driver* es el responsable de secuenciar la ejecución, lanzando tareas de forma distribuida a los diferentes *workers* basándose en el modelo y entorno de ejecución (*cluster management*)



Fuente: Apache Spark. <https://spark.apache.org/docs/latest/ cluster-overview.html>

3.3. API Spark

Este apartado describe las transformaciones y acciones básicas y más utilizadas de Spark, aunque para una revisión profunda podéis consultar la API de Python,[7] Scala[8] y Java.[9]

[7] https://spark.apache.org/docs/latest/ api/python/reference/index.html

[8] https://spark.apache.org/docs/latest/ api/scala/org/apache/spark/index.html

3.3.1. Transformaciones

(9)https://spark.apache.org/docs/latest/ api/java/

- ***map***. La función de *map* aplica cualquier función que podamos definir sobre cada elemento de un RDD generando un nuevo RDD. Es una transformación uno a uno. Esto es, el tamaño del RDD resultante es el mismo que el inicial.

Ejemplo

En el RDD {1, 2, 3, 4, 5} si aplicáis la transformación siguiente

```
rdd.map(lambda x:x+2)
```

obtenéis como resultado: (3, 4, 5, 6, 7).

- ***flatMap***. Es una función casi idéntica a la anterior, pero generando una estructura plana. Por ejemplo, un RDD tal como: {[1, 2], [3, 4, 5]} (esto es, cada elemento del RDD es a su vez un *array*), al aplicarle la función `rdd.map(lambda x:x)` generaría la salida {1, 2, 3, 4, 5}.

- ***filter***. La función *filter()* devuelve un nuevo RDD, que contiene solo los elementos que cumplen con una condición booleana. Es una transformación *narrow* porque no mezcla datos de una partición a otras.

Ejemplo

Suponed que el RDD contiene los primeros cinco números naturales (1, 2, 3, 4 y 5) y aplicáis una función que verifica si un valor es un número par (devolviendo un booleano). El RDD resultante después del filtro contendrá solo los números pares, es decir, 2 y 4.

- ***unión***. Con la función *union ()* obtenéis un RDD con los elementos de combinar varios RDD. La regla clave de esta función es que los dos RDD deben ser del mismo tipo.

Ejemplo

```
> rdd_1 = spark.sparkContext.parallelize(((1,"jan",2016),(3, "nov", 2014),(16, "feb", 2014)))
> rdd_2=spark.sparkContext.parallelize(((5,"dic",2014),(17,"sep",2015)))
> rdd_3=spark.sparkContext.parallelize(((6,"dec",2011),(16,"may",2015)))
> rdd_union = rdd1.union(rdd2).union(rdd3)
```

rdd_union contendrá:

```
((1, "jan", 2016), (3, "nov", 2014), (16, "feb", 2014),(5, "dic", 2014),
(17, "sep", 2015),(6, "dec", 2011),(16, "may", 2015)))
```

- ***reduceByKey***. Cuando aplicamos *reduceByKey* en un conjunto de datos clave-valor (K, V), los pares en la misma partición con la misma clave se combinan y finalmente se agregan entre diferentes particiones, con lo que es una transformación *wide*. Como ejemplo ved la figura 1, en la que los pares clave valor se agregan mediante la suma de aquellos valores con la misma clave.

Ejemplo de *reduceByKey ()*

```
> palabras = ("uno", "dos", "dos", "cuatro", "cinco", "seis", "seis", "ocho",
"nueve", " nueve ", "nueve")

> data = spark.sparkContext.parallelize (palabras) .map (lambda w : (w, 1)).
reduceByKey (lambda a, b: a+b)
```

Resultando en:

```
("uno", 1)
("dos", 2)
("cuatro",1)
("cinco", 1)
("seis", 2)
("ocho", 1)
("nueve", 3)
```

- ***repartition*** y ***coalesce***. Estas transformaciones permiten redimensionar un *dataset* en un número diferente de particiones. Son parecidas, aunque con algunas diferencias. *Repartition* permite redimensionar, ya sea incrementando o reduciendo el número de particiones de un RDD, reconfigurando por completo el RDD. Mientras que la transformación *coalesce()* permite reducir el número de particiones ya existentes a un número determinado, aunque de una forma eficiente, con lo que puede optar por juntar dos particiones moviendo los datos de una de ellas a otra, minimizando la cantidad de datos transferidos por medio de la red. Estas transformaciones son de utilidad cuando se han realizado muchas transformaciones en un RDD tras alguna la ejecución en un *pipeline* de procesado de datos, y resulta finalmente en un RDD altamente particionado, pero con pocos elementos, con lo que Spark debe gestionar muchas pequeñas particiones. En dicho caso, la reducción de particiones os permite operar un RDD de forma más eficiente, y minimizar el tráfico de datos por red.

3.3.2. Acciones

Las funciones ejecutan una tarea específica sobre el RDD, accediendo a los datos. Como ejemplo, mostraremos las tres funciones más habituales.

- ***take(n)***. La función *take* recibe como argumento un número entero *n* y muestra los *n* primeros elementos del RDD. Es importante mencionar que no asume ningún tipo de ordenación, ya que el RDD está formado por particiones, de modo que la agregación de los datos que forman el RDD provenientes de las diferentes particiones no es *a priori* conocido.

Por ejemplo, en el RDD {1, 2, 3, 4, 5} si aplicáis la función *take(3)*:

```
> rdd.take(3)
```

Podríais obtener el resultado: (1,5,4).

- ***count()***. De forma similar, la función *count* devuelve el número de elementos del RDD. Siguiendo con el mismo ejemplo sobre el RDD {1, 2, 3, 4, 5}, si aplicáis la función *count*:

```
> rdd.count()
```

Obtenéis como resultado «5».

- ***saveAsTextFile()***. A modo de ejemplo, mencionamos que un RDD puede almacenarse en disco una vez habéis trabajado con él. La función *saveTextAsFile* guarda el RDD en disco como fichero de texto plano (existen otras funciones para almacenar RDD con diferentes formatos) preservando el número de particiones de que está formado. Por ejemplo, cinco particiones resultan en cinco ficheros distribuidos a través de nuestro sistema de almacenamiento en los diferentes nodos y, cada fichero, a su vez, en diferentes bloques de datos.

4. Arquitecturas *big data*

Una arquitectura *big data* es una estructura software diseñada para realizar las tareas que se han descrito en este módulo: captura e ingestión de datos, análisis tanto en *batch* como en tiempo real y analítica sobre dichos datos.

Existen modelos genéricos de arquitecturas *big data*, tales como Lamda,[10] Kappa[11] o SMACK (Spark, Mesos, Akka, Cassandra and Kafka).[12] En este apartado vamos a describir con cierto detalle el modelo Lambda, ya que cubre las características técnicas esenciales de un sistema *big data*. El modelo Kappa evoluciona el modelo Lambda, centrándose en el procesado en tiempo real, que queda fuera del alcance de este módulo.

(10) http://lambda-architecture.net/

(11) https://www.oreilly.com/radar/questioning-the-lambda-architecture/

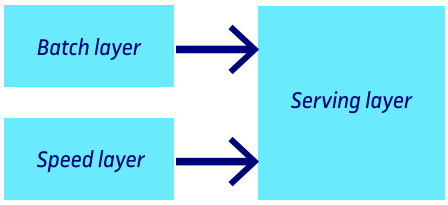
(12) https://www.cuelogic.com/blog/smack-spark-mesos-akka-cassandra-and-kafka-fast-data

4.1. Modelo Lambda

Con tal de resolver los problemas y los requerimientos necesarios ya planteados, se definió la arquitectura Lambda. Es una arquitectura de propósito general, no sujeta ninguna tecnología en particular y que permite explotar datos en entornos *big data*, con baja latencia, escalable y garantizando la tolerancia a fallos y la inmutabilidad de los datos. No obstante, también debe beneficiarse de lo mejor de un sistema tradicional como son modelos de datos complejos que permitan desarrollos verticales a necesidades de negocio concretas.

Una arquitectura Lambda se construye sobre la base de una serie de capas (en adelante *layers*) que satisfacen un propósito concreto, de modo que la combinación de ellas ofrece todas las funcionalidades que un sistema *big data* debe tener. La figura 5 muestra la estructura a alto nivel, de las tres capas.

Figura 5. Abstracción de las tres capas de la arquitectura Lambda



Fuente: Casas Roma, Nin Guerrero y Julbe López (2019)

4.1.1. *Batch layer*

La capa que se ocupa de generar las vistas precalculadas sobre todos los datos disponibles es la llamada *batch layer*. Esta capa almacena una copia maestra de todos los datos históricos, inmutables e incrementales y sobre ellos es posible

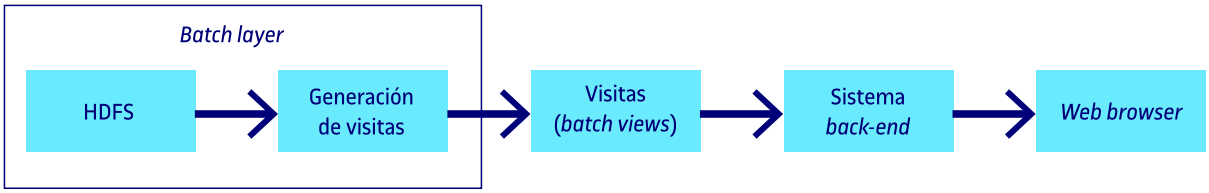
construir cualquier tipo de vista precalculada con más o menos periodicidad. Esta capa es altamente escalable mediante la incorporación de nuevos nodos al clúster.

Sin embargo, existe una limitación en esta capa, y es que la vista no está actualizada a los datos más recientes excepto en el instante en que se empieza a generar. En el lapso de tiempo que se produce en la generación de las vistas, los datos van perdiendo vigencia ya que no se van considerando los datos entrantes mientras se genera dicha vista, solo los que se tenían al empezar el cálculo. Además, es necesario recalcularla constantemente para tenerla lo más actualizada posible. Podríamos decir que la vista *batch* está mayoritariamente desactualizada.

Ejemplo

Un *front-end* correspondiente a una aplicación web que muestra datos relevantes de negocio consultando una vista precalculada sobre un sistema de datos maestros en HDFS. La figura 6 os muestra gráficamente este ejemplo.

Figura 6. Estructura típica de una capa de procesado *batch*



Fuente: Casas Roma, Nin Guerrero y Julbe López (2019)

Así, la aplicación consultaría a un sistema de *back-end* que le proveería de los datos relativos a una consulta y este realizaría las tareas de acceso a datos sobre las vistas precalculadas, no sobre los datos almacenados en HDFS. Como alternativa al uso de vistas, el sistema de *back-end* podría generar las consultas de forma síncrona sobre los datos maestros, pero como se ha comentado anteriormente, el coste computacional y latencia serían tan grandes que el sistema sería inoperativo.

La *batch layer* constituye aquella en la que se almacenan todos los datos de nuestro sistema y ofrece aquellos servicios de procesado para generar las vistas *batch*. Así, el componente más importante de la *batch layer* son los datos *master* (o *master dataset* en adelante), el repositorio donde se almacenan los datos inmutables del negocio.

Desde un punto de vista de procesado, la capa *batch* (*batch processing)* se basa fundamentalmente, al menos inicialmente, en el paradigma MapReduce de Hadoop.

Master dataset

El *master dataset* es el núcleo de datos sobre el que se construye toda la arquitectura, ya que permite reconstruir cualquier vista o agregado sobre ellos en cualquier momento. Así, todas aquellas propiedades destinadas a garantizar su integridad deben ser prioritarias.

Modelo de datos y escalabilidad

El modelo de datos es la manera en la que estructuráis información en unidades semánticas y lógicas. Así, en una base de datos relacional, las tablas, qué contienen y cómo se relacionan responden a un modelo de datos concreto.

Para garantizar la escalabilidad de la capa *batch*, es recomendable desarrollar modelos de datos simples, de modo que cuando se deban añadir nuevos datos, el modelo pueda adaptarse y extenderse de forma ágil. El formato de datos que hay que almacenar en la capa *batch* puede ser diverso, con datos estructurados y no estructurados, sin embargo, almacenar cualquier tipo de datos no es siempre la opción preferible. Si es posible tratar y extraer la información necesaria del contenido no estructurado, esta puede ser la opción deseable dependiendo del cada caso.

> Reglas de dominio y negocio: debéis aplicar unas reglas que permitan normalizar los datos, esto es, un DNI o un código postal siempre van a tener el mismo formato y los procesos de ingesta de datos a vuestro sistema deben implementar y aplicar dichas reglas, con formatos de normalización bien definidos y políticas de calidad de datos bien definidas.

Para poder realizar y reconstruir el ciclo de vida de cualquier entidad en vuestro sistema es recomendable utilizar el modelo de captura de información basada en hechos (*fact-based model*). Esto es, almacenar solo información relativa a aquellos hechos que hacen que una entidad cambie de estado. Capturando dicho cambio y el instante de tiempo en que se ha producido (*timestamp*), así como los datos relativos al cambio de estado que sean necesarios. Es decir, no vais a capturar (registrar) información de dicha entidad si entre dos instantes de tiempo no ha habido ningún cambio. Esto supondría tener dos registros duplicados para instantes de tiempo diferentes, lo que no aporta información significativa y solo desperdicia almacenamiento.

Inmutabilidad y veracidad

Este es un concepto clave y muy diferencial de la arquitectura *big data* respecto a los sistemas relacionales tradicionales, en los cuales la información se actualiza (*update*) frecuentemente. Sin embargo, el *master dataset* almacena nuevos datos, pero estos no pueden borrarse ni modificarse. Así incrementáis la fiabilidad del sistema en caso de que se produzca un error, sea humano o técnico, ya que no se pierde información y cualquier vista puede regenerarse de nuevo.

Sin embargo, la inmutabilidad de la información tiene una importante derivada, y es que debéis asegurar que la información es fiable y permanente en el tiempo. Por ejemplo, el valor de una acción en una fecha concreta será siempre el mismo, aunque su valor «actual» pueda ser distinto del que tenía en un momento anterior (el valor más reciente podría almacenarse en una vista

sobre la serie histórica de valores). Este es un requisito clave, ya que una vez un dato entra en el *master dataset*, va a ser difícil (o imposible) actualizarlo. Así, las tareas de ETL para la ingesta de datos en vuestro *master dataset* son clave para asegurar la veracidad e inmutabilidad de los datos. No es recomendable tener que reprocesar datos después de su ingestión en el *batch layer*. Incluso a veces no es posible ya que, por naturaleza y volumen, pueden no estar disponibles para volver a ejecutar la tarea ETL. Asimismo, no es extraño hacer limpieza de datos de poco valor en vuestro *master dataset* si por volumen y criticidad se puede prescindir de ellos, aunque no es, por propia definición, una opción especialmente deseable. En cualquier caso, el modelo y naturaleza de los datos que hay que almacenar (estructurados y no estructurados) son clave a la hora de dimensionar vuestro sistema.

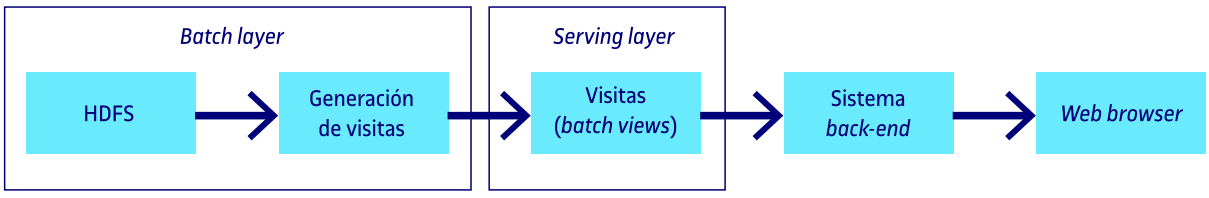
Cuando hablamos de almacenamiento en arquitecturas *big data* es habitual hacer referencia al ***data lake***, que es el repositorio en el cual vais a almacenar toda vuestra información, tanto los datos maestros como todas las posibles vistas sobre ellos. Además, aunque hemos hablado a dos niveles (datos maestros y vistas), es habitual disponer de vistas intermedias sobre los datos maestros y otras vistas sobre ellas, elaborando un modelo de datos de diversas capas sobre el cual lanzar las consultas, realizando sets de vistas específicas de negocio, también llamadas ***datamarts***.

4.1.2. *Serving layer*

Como probablemente ya imagináis, debéis almacenar las vistas en alguna de las capas de vuestra arquitectura, y esta es la *serving layer*. La figura 7 muestra el encaje de la *serving layer* en la arquitectura Lambda.

En ella se almacenan las vistas precalculadas más recientes y, cuando una nueva vista está disponible reemplaza a la que ya teníais en operación. Si algún error ocurre, no se produce el intercambio, de modo que el sistema se mantiene operativo, aunque con datos no actualizados (así aseguramos la fiabilidad del sistema y cumplimos con el requisito de que sea tolerante a fallos).

Figura 7. Ejemplo de la estructura de procesado *batch* con foco en la *serving layer*



Fuente: Casas Roma, Nin Guerrero y Julbe López (2019)

Además, la *service layer* es solo de lectura y en ella no se produce ninguna actualización ni inserción de datos, con lo que la integridad de los datos se mantiene sin posibilidad de corrupción, y mejora, además, la latencia, ya que la lectura de datos es rápida.

Hasta este punto, es importante ver que estas dos capas ya cumplen **casi** todos los requisitos que un sistema *big data* debe cumplir:

- Tolerancia a fallos. Sistemas como Hadoop HDFS incorporan mecanismos de tolerancia a fallos, como hemos descrito en el primer apartado.

- Escalabilidad. Esta capa es escalable horizontalmente con la incorporación de nuevos nodos. Las vistas pueden ser también almacenadas en sistemas distribuidos, teniendo en cuenta que además suelen ser subconjuntos de datos varios órdenes de magnitud más pequeños.

- Generalización, extensión y customización. Esta arquitectura es aplicable a cualquier necesidad de explotación de datos para cualquier ámbito de negocio. Además, nuevas vistas pueden añadirse, modificarse o borrarse. Solo es necesario ejecutar de nuevo el proceso de generación y un conjunto nuevo de vistas estará disponible. Como es lógico, las vistas son tan específicas o generales como sea necesario para responder necesidades generales o específicas del negocio.

- Mantenimiento y depuración. Esta arquitectura permite minimizar el mantenimiento de las vistas, ya que solo van a recalcularse cuando sea necesario y el mantenimiento es solo necesario en la capa *batch*. Sin embargo, sistemas como HDFS son simples de operar, debido a su naturaleza de sistema de ficheros, es muy flexible y con escasa complejidad en comparación con los grandes sistemas de almacenamiento tradicional distribuidos. HDFS hace gran parte del trabajo solo. Asimismo, el disponer de todos los datos maestros permite dar un carácter determinista al sistema como se ha descrito anteriormente.

Llegados a este punto y, como hemos apuntado, ¿qué ocurre en el lapso entre generación de vistas *batch* y el momento presente, cuando vuestro sistema ya no dispone de datos actualizados? Así, la baja latencia es la característica que vuestro sistema todavía no cumple. La *speed layer* soluciona esta limitación.

4.1.3. *Speed layer*

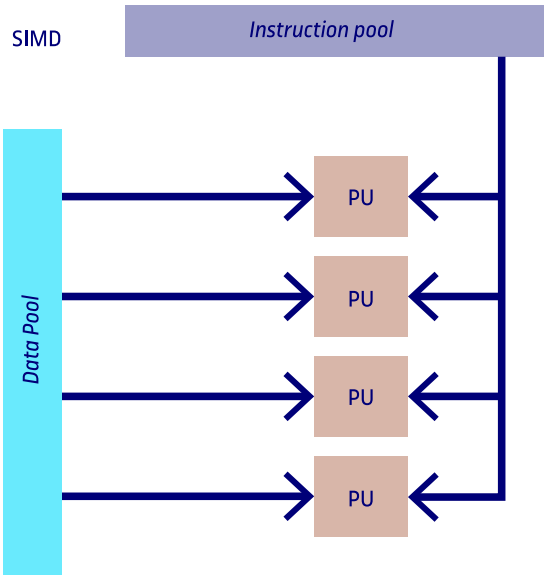
La *speed layer* permite al sistema disponer de información sobre los datos que han entrado en él mientras se regeneran las vistas *batch*, manteniendo todos los datos actualizados. Así, en esta capa las vistas (*real-time views*) se actualizan de forma incremental solo con los nuevos datos que van entrando en el sistema, esto va en contraposición a la *batch layer*, que recalcula las vistas de forma global. Las visitas *batch* contienen toda la información de vuestro sis-

tema hasta cierto instante de tiempo y las vistas ofrecidas por la *speed layer* contienen los datos desde ese momento hasta el momento presente. Ambas capas se complementan y permiten realizar cualquier tipo de consulta sobre todo el ecosistema de datos disponible.

5. Arquitecturas basadas en GPU

Como hemos comentado en apartados anteriores, los servidores con GPU se basan en arquitecturas *many-core*. Estas arquitecturas disponen de una cantidad muy grande de núcleos (procesadores) que realizan una misma operación sobre múltiples datos. En computación esto se conoce como SIMD (*single instruction, multiple data*). SIMD es una técnica empleada para conseguir un gran nivel de paralelismo en cuanto a datos y consiste en instrucciones que aplican una misma operación sobre un conjunto más o menos grande de datos. Es una arquitectura en la que una única unidad de control común despacha las instrucciones a diferentes unidades de procesamiento. Todas estas reciben la misma instrucción, pero operan sobre diferentes conjuntos de datos. Es decir, la misma instrucción es ejecutada de manera sincronizada por todas las unidades de procesamiento. La figura 8 describe de forma gráfica esta idea.

Figura 8. Estructura funcional del concepto «una instrucción, múltiples datos»



Fuentes: <https://es.wikipedia.org/wiki/SIMD> y <https://www.sciencedirect.com/topics/computer-science/single-instruction-multiple-data>

El uso de estas arquitecturas hace posible definir la idea de GPGPU o computación de propósito general sobre procesadores gráficos que permite realizar el cómputo de aplicaciones que tradicionalmente se ejecutaban sobre CPU en GPU de forma mucho más rápida.

Lenguajes de programación como CUDA, desarrollado por NVDIA, permiten utilizar una plataforma de computación paralela con GPU para realizar computación general. Con dicho lenguaje, ingenieros y desarrolladores pueden acelerar las aplicaciones informáticas mediante el aprovechamiento de la poten-

cia de las GPU de una forma más o menos transparente, ya que, aunque CUDA se basa en el lenguaje de programación C, dispone de API y librerías para una gran cantidad de lenguajes como por ejemplo Python o Java.

Bibliografía

Barlas, G. (2014). *Multicore and GPU Programming. An Integrated Approach*. Boston, MA: Elsevier.

Bengfort, B.; Kim, J. (2016). *Data Analytics with Hadoop*. Boston, MA: O'Reilly Media.

Biery, R. (2017). *Introduction to GPUs for Data Analytics. Advances and Applications for Accelerated Computing*. San Francisco, CA: O'Reilly Media.

Casas, J.; Nin, J.; Julbe López, F. (2016). *Big data. Análisis de datos masivos*. Barcelona: Editorial UOC.

Dunning, T.; Friedman, E. (2015). *Real-World Hadoop*. Boston, MA: O'Reilly Media.

Gebali, F. (2011). *Algorithms and Parallel Computing*. Nueva York, NY: John Wiley & Sons, Inc.

Kamburugamuve, S.; Wickramasinghe, P.; Ekanayake, S. *et al.* (2017). «Anatomy of machine learning algorithm implementations in MPI, Spark, and Flink». *The International Journal of High-Performance Computing Applications* (vol. 32, n.º 1, págs. 61-73). <https://doi.org/10.1177/1094342017712976>.

Sitto, K.; Presser, M. (2015). *Field guide to Hadoop: an Introduction to Hadoop, its Ecosystem, and Aligned Technologies*. Boston, MA: O'Reilly Media.

White, T. (2015). *Hadoop: The Definitive Guide. Storage and Analysis at Internet Scale* (4.ª ed.). Boston, MA: O'Reilly Media.

Zaharia, M.; Karau, H.; Konwinski, A. *et al.* (2015). *Learning Spark: Lightning- Fast Big Data Analysis*. Boston, MA: O'Reilly Media.

Zaharia, M.; Chambers, B. (2017). *Spark: The Definitive Guide, Big Data Processing Made Simple*. Boston, MA: O'Reilly Media.