

Maestría en Análisis y Visualización de Datos Masivos

---

# Ingeniería para el Procesado Masivo de Datos

Ingeniería para el Procesado Masivo de Datos

---

# Tema 1. Introducción a las tecnologías big data

# Índice

## Esquema

### Ideas clave

- 1.1. Introducción y objetivos
- 1.2. La sociedad interconectada: la era del cliente
- 1.3. Definición de las tecnologías big data
- 1.4. Origen de las tecnologías big data
- 1.5. Referencias bibliográficas

### A fondo

El papel del big data en la transformación digital

Historia de Hadoop

Leading digital: turning technology into business transformation

Digital transformation: a model to master digital disruption

## Test

# Esquema

TECNOLOGÍAS BIG DATA PARA PROCESAMIENTO MASIVO DE DATOS	
MOTIVACIÓN	DEFINICIÓN
Compañías digitales y la era del cliente	Sociedad interconectada que genera datos nuevos continuamente
Interacciones digitales del cliente generan datos	Fuentes de datos hoy:
Datos que hablan de clientes para conocerlos	Interacciones entre personas (datos no estructurados)
Mejorar su experiencia y predecir sus necesidades y su comportamiento	Interacción entre persona y máquina (datos estructurados o semiestructurados)
¿Qué define un proyecto como big data?	
Las tres V's: velocidad, variedad, volumen	
Otras herramientas relacionadas, construidas sobre las anteriores, para propósitos específicos. Estudiaremos: Hive, Kafka, Impala	
Simplicidad de resolución con tecnologías big data	

## 1.1. Introducción y objetivos

Empezaremos la asignatura motivando los contenidos que se estudiarán en el resto del temario. Repasaremos las necesidades de la sociedad de la información en la actualidad, una era en la que todos estamos interconectados y somos fuentes de datos. Veremos los retos tecnológicos que esto supone y presentaremos formalmente las tecnologías que los solventan.

Los objetivos que persigue este tema son:

- ▶ Comprender cuáles son las necesidades actuales de procesamiento de datos, sus causas y cómo son solventadas por las tecnologías *big data*.
- ▶ Entender el concepto de clúster de ordenadores y cuáles son las principales tecnologías distribuidas capaces de explotarlo.
- ▶ Conocer las herramientas principales que componen el ecosistema Hadoop, cuál es la finalidad de cada una y cómo se relacionan entre sí.

## 1.2. La sociedad interconectada: la era del cliente

Las tecnologías *big data* surgen para dar respuesta a las nuevas necesidades de la sociedad actual. Vivimos en un mundo interconectado, en el que el 90 % de la información existente, preservada en medios de cualquier tipo, se ha creado en los últimos dos años. El crecimiento de la información producida en el mundo por fuentes de todo tipo, tanto físicas como electrónicas, es exponencial de un tiempo a esta parte. Aunque las estimaciones acerca del volumen divergen, la siguiente gráfica muestra de manera orientativa este fenómeno.

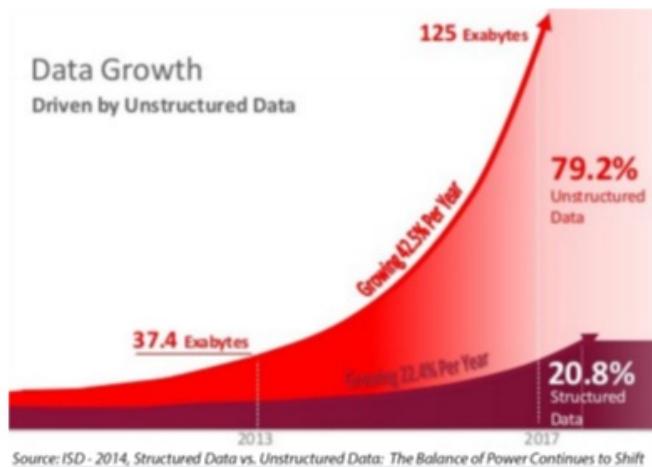


Figura 1. Previsión de crecimiento de los datos generados en todo el mundo. Fuente: [Oracle Machine Learning](#).

Casi el 80 % de los datos que se crean son generados por personas y, por ello, suelen ser datos no estructurados (texto libre, comentarios de personas, tuits, imágenes, sonidos, vídeos). Los 20 % restantes son datos estructurados generados por máquinas [datos de *logs*, sensores, Internet de las cosas (IoT), en general] con el fin de ser procesados generalmente por otras máquinas.

### Fuentes de datos en la actualidad

Existen principalmente tres tipos de situaciones que generan datos en la actualidad:

- ▶ La **interacción entre humanos** a través de un sistema informático que registra información mientras se produce la interacción. Ejemplos claros son el correo electrónico, los foros de Internet o las redes sociales, donde los datos los generamos los humanos al interactuar entre nosotros utilizando dichos medios. Suelen ser datos no estructurados, posteriormente procesados por máquinas.
- ▶ La **interacción entre un humano y una máquina**. El ejemplo más claro es la navegación en Internet: los servidores web generan *logs* con información sobre el proceso de navegación. Lo mismo ocurre al efectuar compras en alguna plataforma web de comercio electrónico o en banca *online*, donde cada una de nuestras transacciones queda registrada y será procesada después con el objetivo de estudiar nuestro comportamiento, así como de ofrecernos productos mejores y más personalizados. Tienden a ser datos estructurados o semiestructurados.
- ▶ La **interacción entre máquinas**. Varias máquinas intercambian información y la almacenan con el objetivo de ser procesada por otras máquinas. Un ejemplo son los sistemas de monitorización, en los que un sistema de sensores suministra la información recibida a otras máquinas para que realicen algún procesado sobre los datos. Al ser la propia máquina quien la genera, suele ser información estructurada, ya que el *software* se encarga de sistematizarla.

Algunas cifras que resumen la cantidad de datos generados gracias a Internet se recogen en la siguiente imagen. Llama especialmente la atención el crecimiento experimentado por empresas como Netflix o Instagram, frente a la estabilización de los gigantes como Google, Facebook o YouTube.

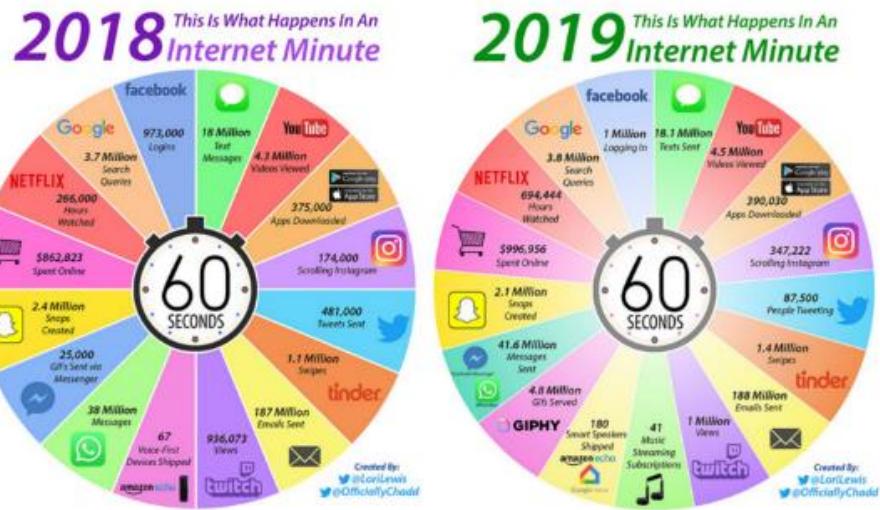


Figura 2. Eventos generados en Internet en un minuto por las empresas digitales. Fuente: Twitter.

## La transformación digital en relación con los datos

La conclusión global a la que llegamos es que el mundo ya ha cambiado, y lo podemos confirmar si examinamos hechos como los siguientes:

- ▶ La empresa que transporta a más personas en el mundo es Uber, que tiene 0 coches físicos.
- ▶ La empresa que más habitaciones reserva es Airbnb, que tiene 0 hoteles físicos.
- ▶ La empresa que más música vende es Spotify, que tiene 0 estudios de grabación.
- ▶ La empresa que vende más películas es Netflix, que tiene 0 estudios.

Con frecuencia, se llevan a cabo más interacciones digitales que físicas entre las personas y las compañías que nos dan servicio, ya sea de suministro de energía, agua o gas; de telecomunicaciones o telefonía; de venta *online* de productos de todo tipo, o incluso de movimientos y servicios bancarios. Estas interacciones están generando, de forma continua y masiva, datos muy valiosos que hablan del comportamiento de los clientes y su análisis permite anticipar qué es lo que estos van a demandar. De hecho, estamos evolucionando más rápido que las propias

compañías, hasta el punto de que se ha abierto una brecha entre las empresas físicas tradicionales y los gigantes digitales, como muestra la figura 2.

Con el objetivo de llenar este espacio, surge la **transformación digital**, que persigue esencialmente tres objetivos:

- ▶ **Centrarse en el cliente**, es decir, pensar continuamente en lo que necesita y en mejorar (personalizar) su experiencia y sus interacciones con la compañía. Esto requiere recabar y analizar grandes cantidades de datos sobre su comportamiento.
- ▶ **Centrarse en canales digitales**, especialmente dispositivos móviles, puesto que las interacciones digitales son las que generan mayor cantidad de datos y, cada vez con más frecuencia, se realizan usando estos dispositivos en vez del PC.
- ▶ **Decisiones guiadas por los datos** (*data-driven*), para lo cual es necesaria la ciencia de (grandes) datos (*big data science*).

## 1.3. Definición de las tecnologías big data

Para acometer estos objetivos, la mayor parte de las tecnologías existentes hasta hace pocos años (principios del siglo xx) no eran suficientes. Ello se debía a la necesidad de procesar, almacenar y analizar datos con ciertas características especiales, las denominadas **tres «v» del big data**:

- ▶ **Volumen:** cantidades de datos lo suficientemente grandes como para no poderse procesar con tecnologías tradicionales.
- ▶ **Velocidad:** flujos de datos que van llegando en tiempo real y tienen que procesarse de manera continua según se van recibiendo.
- ▶ **Variedad:** datos de fuentes diversas, estructuradas y no estructuradas (sean bases de datos relacionales o no relacionales, datos de imágenes, sonido, etc.), que tienen que ser manejados y cruzados de manera conjunta.

*Un proyecto es **big data** cuando implica alguna de las tres «v».*

Una definición más ajustada y realista de un proyecto **big data** sería:

*Un proyecto es **big data** cuando la mejor manera de resolverlo (más rápida, eficiente, sencilla) implica utilizar tecnologías **big data**.*

Podemos definir **big data** como:

*Conjunto de **tecnologías y arquitecturas** para almacenar, mover, acceder y procesar (incluido analizar) datos que eran muy difíciles o imposibles de manejar con tecnologías tradicionales.*

Las causas de esta imposibilidad pueden ser:

- ▶ Cantidades ingentes de datos inimaginables hace unos años.
- ▶ Datos de fuentes diversas, heterogéneas, poco estructuradas, como documentos o imágenes/sonido, que, aun así, necesitamos almacenar y consultar (NoSQL).
- ▶ Datos dinámicos, recibidos y procesados según llegan (flujos de datos o *streams*).

Cabe destacar que, en la definición anterior, se han omitido de manera deliberada palabras como algoritmo, inteligencia, ciencia de datos o cualquier referencia a qué hacer o cómo analizar y explotar dichos datos. Las herramientas *big data* permiten aplicar a datos masivos técnicas que ya existían, pero son tecnologías y no técnicas en sí mismas. Las técnicas de análisis pertenecen al ámbito de la estadística, las matemáticas, las ciencias de la computación y la inteligencia artificial.

La mayoría de estas técnicas y algoritmos han existido desde mucho antes, algunas desde mediados del siglo xx. La sinergia con las tecnologías *big data* consiste en que estas permiten aplicar técnicas de análisis existentes a cantidades de datos mucho mayores, de naturaleza heterogénea. De este modo, logran resultados en menos tiempo y de mucha más calidad, al cruzar datos de diversas fuentes y ser capaces de procesarlos y usarlos para entrenar un algoritmo. Desgraciadamente, se han extendido entre el gran público mitos sobre el término *big data*, tales como los siguientes:



Figura 3. Conceptos erróneos de la finalidad de las tecnologías *big data* extendidos en la sociedad.  
Fuente: Google Images.

## 1.4. Origen de las tecnologías big data

La primera empresa que fue consciente del aumento de los datos que se estaban generando en Internet fue Google, ya que su buscador debe ser capaz de indexar las webs nuevas para que puedan ser encontradas. En los albores del siglo xx, Sanjay Ghemawat, Howard Gobioff y Shun-Tak Leung (2003) publicaron un artículo que se hizo mundialmente famoso, en el cual explicaban el sistema de archivos distribuido Google File System (GFS) que habían desarrollado. Los autores presentaron por primera vez la idea de utilizar ordenadores convencionales conectados entre sí (formando un clúster) para poder almacenar archivos que ocupaban más que un solo disco duro.

A esto se le denomina **commodity hardware**: máquinas no especialmente potentes, similares a las que tienen los usuarios domésticos, pueden conectarse entre sí para trabajar conjuntamente como una sola, a fin de resolver tareas de mayor envergadura. GFS fue la base del sistema de archivos distribuido HDFS que veremos en temas posteriores.

En 2004, Jeffrey Dean y Sanjay Ghemawat publicaron un nuevo artículo, que se popularizó rápidamente, donde explicaban un modelo de programación (MapReduce) aplicable a un clúster de ordenadores para procesar en paralelo archivos almacenados en el sistema de archivos GFS. Google también publicó una biblioteca de programación de código abierto donde implementaba dicho paradigma. Su principal punto fuerte era la abstracción (simplificación) de todos los detalles de *hardware*, redes y comunicación entre los nodos del clúster, para que el usuario pudiera centrarse en el desarrollo de la lógica de la aplicación distribuida de manera sencilla. Durante muchos años, MapReduce fue el estándar de desarrollo de *software big data* a nivel comercial.

En 2009, y motivado por las deficiencias de Hadoop en ciertas tareas, un

investigador llamado Matei Zaharia creó una nueva tecnología *open source* de procesamiento distribuido, llamada Apache Spark, durante la realización de su tesis doctoral en Berkeley. Estudiaremos Spark en profundidad en temas posteriores; por el momento, basta señalar que comparte con MapReduce los principios de ejecutar en un clúster de ordenadores *commodity* y simplificar todos los detalles de redes y comunicación entre los nodos. Desde 2014, MapReduce ha sido reemplazado por Spark en su totalidad. Las herramientas que lo utilizaban como motor de ejecución han sobrevivido gracias a que dicho motor es una pieza intercambiable en muchas de ellas y han sabido adaptarlo a Spark.

## El ecosistema Hadoop

La idea básica que hay tras las tecnologías de procesamiento distribuido es la siguiente:

Es posible procesar grandes cantidades de datos de forma distribuida  
entre varias máquinas interconectadas (clúster), cada una no  
necesariamente muy potente (*commodity hardware*). Si se necesita más  
potencia de cálculo o más capacidad de almacenamiento, basta con  
añadir más máquinas al clúster.

Siguiendo esta filosofía y teniendo como punto de partida el sistema de archivos distribuido GFS (que en Hadoop se transformó en HDFS o Hadoop Distributed File System) y el paradigma MapReduce, se creó **un conjunto de herramientas *open source* para procesamiento distribuido**, cada una con un propósito específico, pero todas interoperables entre sí, que se denomina el **ecosistema Hadoop** (figura 5).



Figura 4. El clúster MareNostrum 4, en el Barcelona Supercomputing Center (BSC). Cada armario se denominarack y cada bandeja es un ordenador completo (nodo).

Sin intentar ser exhaustivos y a título meramente informativo, damos una breve descripción de cada una:

- ▶ **HDFS:** sistema de archivos distribuido que estudiaremos en el tema siguiente.
- ▶ **MapReduce:** paradigma de programación para un clúster de ordenadores (forma de estructurar programas y también biblioteca de programación que se ejecuta sobre el clúster). Actualmente ha caído en desuso.
- ▶ **Flume:** herramienta para tratamiento de *logs*.
- ▶ **Sqoop:** herramienta para migración de grandes cantidades de datos desde bases de datos convencionales a HDFS.
- ▶ **Zookeeper:** coordinador.

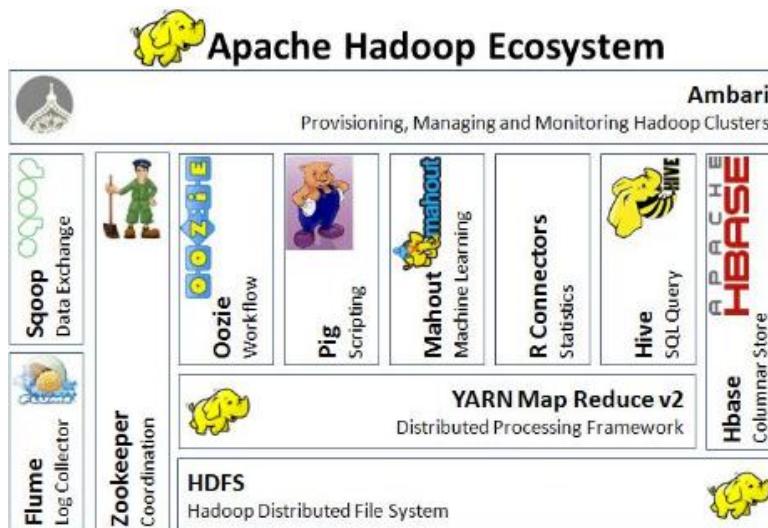


Figura 5. El ecosistema de herramientas *open source* Hadoop para procesamiento distribuido.

- ▶ **Oozie**: herramienta para planificación y ejecución de flujos de datos.
- ▶ **Pig**: herramienta para programar flujos de datos con sintaxis similar a SQL, pero con mayor nivel de granularidad, cuyo procesamiento se efectúa con MapReduce.
- ▶ **Mahout**: biblioteca de algoritmos de *machine learning*. Originalmente programada con MapReduce, tenía un rendimiento pobre, pero actualmente soporta otros *backend* como Spark.
- ▶ **R Connectors**: herramientas para conectar MapReduce con el lenguaje de programación R. En desuso, al igual que MapReduce.
- ▶ **Hive**: herramienta para manejar datos almacenados en HDFS utilizando lenguaje SQL. En su origen, utilizaba MapReduce como motor de ejecución. Actualmente soporta Spark y Apache Tez.
- ▶ **HBase**: base de datos NoSQL de tipo columnar, que permite, entre otras cosas, tener registros (filas) de longitud y número de campos variable.

En este curso, nos centraremos en las siguientes herramientas de Apache, que

constituyen el estándar tecnológico *de facto* en la mayoría de las empresas que utilizan tecnologías *big data*:

- ▶ **HDFS (Hadoop Distributed File System):** sistema de archivos distribuido inspirado en el GFS de Google, que permite distribuir los datos entre distintos nodos de un clúster, gestionando la distribución y la redundancia de forma transparente para el desarrollador que vaya a hacer uso de esos datos.
- ▶ **Apache Hive:** herramienta para acceder mediante sintaxis SQL a datos estructurados que están almacenados en un sistema de archivos distribuido, como HDFS u otros similares. Las consultas SQL son traducidas automáticamente a trabajos de procesamiento distribuido según el motor que se haya configurado, que puede ser MapReduce, Apache Spark o Apache Tez.
- ▶ **Apache Spark:** motor de procesamiento distribuido y bibliotecas de programación distribuida de propósito general, que opera siempre en la memoria principal (RAM) de los nodos del clúster. Desde hace unos años, ha reemplazado totalmente a MapReduce al ser mucho más rápido.
- ▶ **Apache Kafka:** plataforma para manejo de eventos en tiempo real, que consiste en una cola de mensajes distribuida y masivamente escalable sobre un clúster de ordenadores. Estos mensajes pueden ser consumidos por uno o varios procesos externos (por ejemplo, trabajos de Spark).

## Distribuciones de Hadoop

Al estar constituido Hadoop por un conjunto de herramientas diferentes, cada una requiere su propia instalación y configuración en el clúster para poder operar con otras ya instaladas. Este proceso era tedioso y requería bastantes conocimientos. Con el fin de simplificar esta tarea, surgieron las distribuciones de Hadoop, que son conjuntos de herramientas del ecosistema Hadoop empaquetadas juntas, en versiones compatibles y perfectamente interoperables entre ellas, distribuidas con un único *software*, razón por la que no hay necesidad de instalarlas por separado.

Empresas como Cloudera, Hortonworks (estas dos fueron competencia mutua durante mucho tiempo, hasta que acabaron por fusionarse en 2018) o MapR nacieron para crear distribuciones de Hadoop y añadirles, en algunos casos, herramientas propietarias totalmente nuevas, que no pertenecían al ecosistema Hadoop, o incluso modificaciones propias del código fuente original de las herramientas de Hadoop, para solucionar fallos o añadir características avanzadas. Todas las distribuciones de Hadoop de estas empresas tienen versiones *open source* y de pago. La siguiente tabla, incluida a título meramente informativo, compara sus características:

	Cloudera	Hortonworks	MapR
Componentes	Apache modificados y añadidos	Solo Apache oficiales	Apache y añadidos
Versiones	<i>Open source</i> (CDH) y de pago	Sólo 100 % <i>open source</i>	<i>Open source</i> y de pago
Sistema operativo	Linux (Windows vía VMWare)	Linux y Windows	Linux (Windows: VM Ware)
Año de creación	2008	2011	2009
Notas adicionales	Es la más extendida. Certificación muy popular.	Única para Windows, única 100 % <i>open source</i>	La más rápida y fácil de instalar

Tabla 1. Características de herramientas empresariales.

## 1.5. Referencias bibliográficas

Dean, J. y Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1), 107-113. <https://doi.org/10.1145/1327452.1327492>

Ghemawat, S., Gobioff, H. y Leung, S-T. (2003). The Google File System. *A CM SIGOPS Operating Systems Review*, 37(5), 29-43. <https://doi.org/10.1145/1165389.945450>

## El papel del big data en la transformación digital

Newman, D. (2015, 22 de diciembre). The role big data plays in digital transformation.

*Forbes*. <https://www.forbes.com/sites/danielnewman/2015/12/22/the-role-big-data-plays-in-digital-transformation/>

En este artículo, el autor explica de qué manera el *big data* puede ayudar a grandes y pequeñas empresas a minimizar costes, maximizar resultados y, en definitiva, ser competitivas en un entorno cambiante.

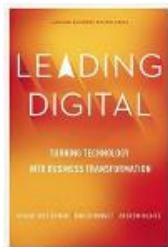
## Historia de Hadoop

Bonaci, M. (2015, 11 de abril). The history of Hadoop. *Medium.com*. <https://medium.com/@markobonaci/the-history-of-hadoop-68984a11704>

Este artículo analiza en profundidad el contexto en el que nació Hadoop y la evolución que ha experimentado.

## Leading digital: turning technology into business transformation

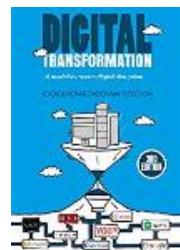
Westerman, G., Bonnet, D. y McAfee, A. (2014) *Leading digital: turning technology into business transformation*. Harvard Business Review Press.



En este manual, los autores ponen el foco en cómo grandes compañías de la industria tradicional están aprovechando las posibilidades de la era digital como estrategia para alcanzar el éxito.

## Digital transformation: a model to master digital disruption

Caudron, J. y Van Peteghem, D. (2018) *Digital transformation: a model to master digital disruption* (3. a edición). BookBaby.



La era digital ha creado multitud de oportunidades para desarrollar nuevos productos y servicios, y abrir nuevas líneas de negocio. Este libro ofrece una metodología para que las empresas que aún no lo han hecho den el salto y aprovechen todas las ventajas que ofrece la transformación digital.

1. En la sociedad actual, la mayoría de los datos que se generan a diario son...

  - A. Datos no estructurados generados por las personas.
  - B. Datos estructurados generados por máquinas.
  - C. Datos estructurados generados por las personas.
2. ¿Qué retos presentan los datos generados por personas en una red social?

  - A. Son datos no estructurados (imágenes, vídeos), más difíciles de procesar.
  - B. Son datos masivos.
  - C. Las dos respuestas anteriores son correctas.
3. El término *commodity hardware* se refiere a...

  - A. Máquinas remotas que se alquilan a un proveedor de *cloud* como Amazon.
  - B. Máquinas muy potentes que suelen adquirir las grandes empresas.
  - C. Máquinas de potencia y coste normales, conectadas entre sí para formar un clúster más potente.
4. Un proyecto se denomina *big data* cuando...

  - A. Solo se puede resolver gracias a las tecnologías *big data*.
  - B. La forma más eficaz y directa de abordarlo implica tecnologías *big data*.
  - C. El problema que resuelve contiene simultáneamente las tres «v».
5. Las tres «v» del *big data* se refieren a:

  - A. Volumen, velocidad y variedad.
  - B. Voracidad, volumen y velocidad.
  - C. Ninguna de las respuestas anteriores es correcta.

- 6.** Lo mejor, si necesitamos más potencia de cómputo en un clúster *big data*, es...

  - A. Reemplazar algunas máquinas del clúster por otras más potentes.
  - B. Aumentar el ancho de banda de la red.
  - C. Añadir más máquinas al clúster y aprovechar todas las que ya había.
- 7.** El sistema de ficheros precursor de HDFS fue...

  - A. GFS.
  - B. Apache Hadoop.
  - C. Apache MapReduce.
- 8.** Una distribución de Hadoop es...

  - A. Un *software* con licencia comercial para clústeres, difundido por Microsoft.
  - B. Un conjunto de aplicaciones del ecosistema Hadoop, con versiones interoperables entre sí y listas para usarse.
  - C. Ninguna de las opciones anteriores es correcta.
- 9.** ¿Qué compañías fueron precursoras de HDFS y MapReduce?

  - A. Google y Microsoft, respectivamente.
  - B. Google, en los dos casos.
  - C. Google y Apache, respectivamente.
- 10.** Definimos *big data* como...

  - A. Todos aquellos algoritmos que se pueden ejecutar sobre un clúster de ordenadores.
  - B. Las tecnologías distribuidas de Internet que posibilitan una sociedad interconectada por las redes sociales.
  - C. Las tecnologías que permiten almacenar, mover, procesar y analizar cantidades inmensas de datos heterogéneos.

Ingeniería para el Procesado Masivo de Datos

---

## Tema 2. HDFS y MapReduce

# Índice

[Esquema](#)

[Ideas clave](#)

[2.1. Introducción y objetivos](#)

[2.2. Introducción a HDFS](#)

[2.3. Arquitectura de HDFS](#)

[2.4. Comandos de HDFS más frecuentes](#)

[2.5. Programación distribuida y MapReduce](#)

[2.6. Referencias bibliográficas](#)

[A fondo](#)

[Guía de usuario HDFS](#)

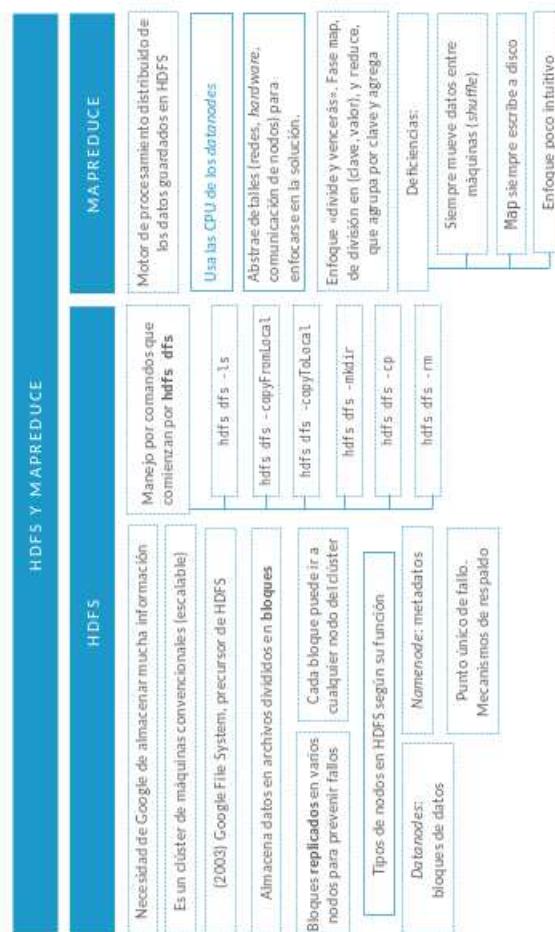
[Glosario de términos de la empresa Databricks, creadores de Apache Spark](#)

[Hadoop: the definitive guide](#)

[Arquitectura de HDFS en detalle](#)

[Test](#)

# Esquema



## 2.1. Introducción y objetivos

En este tema, examinaremos Hadoop Distributed File System, conocido habitualmente como HDFS, que constituye la primera de las tecnologías distribuidas que estudiaremos durante el curso. Ha cambiado relativamente poco con los años y sigue siendo la base de casi todas las demás, puesto que proporciona la capa de almacenamiento persistente (en discos duros) de los datos. En la parte final del tema, veremos el paradigma de programación distribuida MapReduce, propuesto para procesar en paralelo los datos almacenados en HDFS.

Los objetivos que persigue este tema son:

- ▶ Conocer las características propias de HDFS.
- ▶ Entender la arquitectura de HDFS y las operaciones de escritura y lectura.
- ▶ Familiarizarse con los comandos más habituales de HDFS.
- ▶ Comprender el paradigma de programación MapReduce, sus virtudes y sus deficiencias.

## 2.2. Introducción a HDFS

El artículo publicado por Ghemawat, Gobioff y Leung (2003) acerca de Google File System (GFS) fue el germen del sistema de archivos distribuido HDFS (Hadoop Distributed File System), que constituye una parte fundamental del ecosistema Hadoop y el cual se ha seguido utilizando sin apenas cambios desde que se introdujo por primera vez.

En las próximas páginas, veremos:

- ▶ Las características propias de HDFS y sus componentes desde el punto de vista de la arquitectura.
- ▶ Sus peculiaridades como sistema de archivos distribuido frente a sistemas de archivos tradicionales no distribuidos (en una sola máquina).
- ▶ Su funcionamiento interno.
- ▶ Los comandos más habituales utilizados para manejarlo.

Podemos definir HDFS como:

Un sistema de archivos distribuido destinado a almacenar archivos muy grandes con patrones de acceso en *streaming*, pensado para clústeres de ordenadores convencionales.

Si desgranamos esta definición, tenemos que:

**1. Sistema de archivos distribuido:** los archivos se almacenan en una red de máquina, lo cual implica que:

- ▶ Son máquinas *commodity* (*hardware* convencional, que puede fallar). Esto difiere de

los sistemas tradicionales, donde se adquiría un gran ordenador o *mainframe*, mucho más potente de lo que un usuario doméstico poseía en casa, pero con la restricción de que, al ser una sola máquina, el fallo o la falta de capacidad obligaba a reemplazar el *mainframe* por otro más grande.

- ▶ Es escalable: si se requiere más capacidad, basta con añadir más nodos, en lugar de reemplazar una máquina por otra más grande.
- ▶ Necesita incorporar mecanismos *software* de recuperación frente a fallo de un nodo, algo que veremos en la siguiente sección.

**2. Pensado para almacenar archivos muy grandes:** permite guardar archivos mayores que la capacidad del disco de una máquina individual. Es decir, un solo archivo puede ocupar cientos de GB, varios TB e incluso PB, a pesar de que cada disco duro de un nodo tenga una capacidad limitada de 500 GB solamente, por ejemplo.

**3. Archivos con patrón de acceso *write-once, read-many*:** archivos que se crean y después se accede a ellos para ser leídos en numerosas ocasiones. No son archivos cuyo contenido necesitemos reemplazar con frecuencia ni que sean borrados habitualmente. Además, a diferencia de una base de datos, no es importante el tiempo de acceso a partes concretas del archivo (por ejemplo, el fragmento exacto del archivo que contiene el registro con los datos de una persona en concreto), sino que se suele utilizar y procesar el archivo completo en las aplicaciones (modo *batch*, no interactivo). Por este motivo:

- ▶ No soporta modificación de archivos existentes, sino solo lectura, escritura y borrado.
- ▶ No funciona bien para:
  - Aplicaciones que requieran baja latencia a fin de acceder a registros individuales dentro de un archivo.

- Muchos archivos pequeños (generan demasiados metadatos).
- Archivos que se modifiquen con frecuencia.

HDFS es, en realidad, un *software* escrito en lenguaje Java, que se instala encima del sistema de archivos de cada nodo del clúster. El *software* HDFS se encarga de proporcionarnos una abstracción que nos da la ilusión de estar usando un sistema de archivos, pero, por debajo, continúa usando el sistema de archivos nativo del sistema operativo (por ejemplo, en el caso de máquinas Linux, que son las más habituales para instalar HDFS, estará utilizando por debajo el conocido sistema de archivos ext4). La peculiaridad es que nos permite almacenar archivos de manera distribuida, pero manejarlos como si fuese un único sistema de archivos.

Es importante resaltar que, cuando un nodo forma parte de un clúster en el que está instalado HDFS, es posible utilizar tanto el sistema de archivos local del nodo (al que podemos acceder, por ejemplo, abriendo una terminal en ese nodo mediante acceso SSH) como el sistema de archivos HDFS, que existe «además de» cada uno de los sistemas de archivos locales. Para crear datos en HDFS, podemos copiarlos en este *software* desde el sistema de archivos de una máquina concreta si ya existían en ella, o bien pueden generarlos directamente en HDFS como resultados de otra aplicación que se esté ejecutando de forma distribuida en el clúster (por ejemplo, un trabajo de Spark).

La mayoría de los comandos que ofrece HDFS para manejar archivos se llaman igual que los comandos del sistema de archivos de Linux, con el fin de facilitar su uso a usuarios que ya estaban acostumbrados a trabajar con este sistema operativo. No obstante, esto no debe confundirnos: la ejecución de un comando en el sistema de archivos local (por ejemplo, ls, para mostrar el contenido de un directorio de nuestro ordenador) desencadena unas acciones muy diferentes a la ejecución del comando con el mismo nombre, pero en HDFS (precedido por las palabras hdfs dfs).

## 2.3. Arquitectura de HDFS

### Bloques de HDFS

En un dispositivo físico como un disco duro, un bloque físico (o sector) de disco es la cantidad de información que se puede leer o escribir en una sola operación de disco. Habitualmente son 512 bytes. En un sistema de archivos convencional, como, por ejemplo, ext4 de Linux, o NTFS y FAT32 de Windows, un bloque del sistema de archivos es el mínimo conjunto de sectores que se pueden reservar para leer o escribir un archivo. Suele ser configurable (Linux ext4 permite 1 KB, 2 KB, etc.). Generalmente es de 4 KB, y siempre debe contener un número de sectores físicos que sea potencia de 2.

En sistemas de archivos convencionales (p. ej., ext4 de Linux, o NTFS y FAT32 de Windows), los archivos menores que el tamaño de bloque del sistema de archivos siguen ocupando un bloque completo, es decir, se desperdicia una pequeña cantidad de espacio. HDFS tiene su propio tamaño de bloque, que es configurable (por defecto, es de 128 MB), pero los archivos de menos de un bloque de HDFS no desperdician espacio, a pesar de que siempre usan su propio bloque de datos (es decir, nunca se comparte un bloque de HDFS entre varios archivos).

El tamaño de bloque de HDFS tiene varias implicaciones:

- ▶ Un archivo se parte en bloques, que pueden almacenarse en máquinas diferentes. Así se puede almacenar un archivo mayor que el disco de una sola máquina, ya que se almacena troceado.
- ▶ Cada bloque requiere metadatos (que se almacenan en el *namenode*) para mantenerlo localizado y saber de qué archivo forma parte. Si los bloques son muy pequeños, se generan demasiados metadatos, lo cual llena muy rápido el *namenode* y hace más difíciles las búsquedas. Si son muy grandes, limitan el paralelismo de *frameworks* que operan a nivel de bloque (como Spark), en los que varias máquinas

procesan a la vez bloques diferentes de un mismo archivo.

- ▶ Los bloques de datos suelen estar replicados para ofrecer alta disponibilidad y máximo paralelismo: cada bloque está en  $k$  máquinas, donde  $k$  es el factor de replicación. HDFS fija por defecto un factor de replicación de 3, aunque este valor es configurable individualmente para cada fichero mediante el comando `hadoop dfs -setrep -w 3 /user/hdfs/file.txt`.

La siguiente imagen muestra un ejemplo de un archivo `flights.csv` de 500 MB que, al subirlo a HDFS, ha sido particionado automáticamente por este sistema en cuatro bloques: tres de ellos tienen 128 MB (bloques completos, de longitud 134 217 728 bytes) y el cuarto es más pequeño. Cada uno de estos bloques se ha replicado tres veces, también de manera automática. El comando de HDFS que se ha ejecutado después, `fsck`, nos permite consultar los metadatos asociados a este fichero, los cuales describen cómo está almacenado físicamente.

```
[pvillacorta@meetupds1 ~]$ hdfs fsck /SparkMeetup/flights1994.csv -blocks -files
Connecting to namenode via http://meetupds1:50070/fsck?ugi=pvillacorta&blocks=1&files=1&path=%2FSparkMeetup%2Fflights1994.csv
FSCK started by pvillacorta (auth:SIMPLE) from /144.76.3.23 for path /SparkMeetup/flights1994.csv at Thu Apr 18 21:43:27 CEST 2019
/SparkMeetup/flights1994.csv 501558665 bytes, 4 block(s): OK
0. BP-2126204769-          -1526901840376:blk_1073977383_236559 len=134217728 repl=3
1. BP-2126204769-          -1526901840376:blk_1073977384_236560 len=134217728 repl=3
2. BP-2126204769- ....    -1526901840376:blk_1073977385_236561 len=134217728 repl=3
3. BP-2126204769-          -1526901840376:blk_1073977386_236562 len=98905481 repl=3

Status: HEALTHY
Total size: 501558665 B
Total dirs: 0
Total files: 1
Total symlinks: 0
Total blocks (validated): 4 (avg. block size 125389666 B)
Minimally replicated blocks: 4 (100.0 %)
Over-replicated blocks: 0 (0.0 %)
Under-replicated blocks: 0 (0.0 %)
Mis-replicated blocks: 0 (0.0 %)
Default replication factor: 3
Average block replication: 3.0
Corrupt blocks: 0
Missing replicas: 0 (0.0 %)
Number of data-nodes: 3
Number of racks: 1
FSCK ended at Thu Apr 18 21:43:27 CEST 2019 in 1 milliseconds

The filesystem under path '/SparkMeetup/flights1994.csv' is HEALTHY
```

Figura 1. Salida dada por el comando `fsck` para un archivo de 500 MB subido a HDFS.

- ▶ *Rack-awareness*: es posible, aunque no obligatorio, especificar, en la configuración de HDFS, la disposición física (topología) del clúster e indicar qué nodos comparten un mismo *rack* (armario) y cuál es la cercanía entre ellos. De esta manera, HDFS puede decidir mejor en qué nodo del clúster debe colocar cada bloque de un archivo que se vaya a escribir, con el objetivo de minimizar las pérdidas si falla un *rack*

completo, lo cual a veces ocurre debido a que los nodos en un mismo *rack* comparten cierta infraestructura física.

La norma general es no colocar más de una réplica de un bloque en el mismo nodo (esto es obvio), ni más de dos réplicas en nodos del mismo *rack*. A partir de ahí, es posible establecer políticas más complejas para decidir la asignación de bloques de datos a nodos concretos. La topología también influye en la elección de los *datanodes* que servirán cada bloque de datos al leer un fichero, según la cercanía física al cliente a quien serán servidos.

## *Datanodes y namenode*

Cuando se instala HDFS en un clúster, cada nodo puede utilizarse como *datanode* o como *namenode*. El *namenode* (debe haber, al menos, uno) mantiene la estructura de directorios existentes y los metadatos asociados a cada archivo. Por su parte, los *datanodes* almacenan bloques de datos y los devuelven a petición del *namenode* o del programa cliente que está accediendo a HDFS para leer datos.

El *namenode* recibe periódicamente de los *datanodes* un *heartbeat* (por defecto, cada tres segundos, aunque es configurable en la propiedad `dfs.heartbeat.interval`) y un listado de todos los bloques presentes en cada *datanode* (por defecto, cada seis horas, configurable en `dfs.blockreport.*`). El *namenode* es punto único de fallo (SPOF), lo que significa que, sin él, no es posible utilizar HDFS. La razón es que el *namenode* posee la estructura de directorios del sistema de archivos y conoce qué bloques forman cada archivo y dónde se almacenan físicamente. Por ese motivo, se han ideado diferentes mecanismos, tanto de respaldo del *namenode* como de alta disponibilidad, que detallamos ahora. Más adelante explicamos cómo escalar el *namenode* cuando, en el sistema de archivos, ha crecido el número y la complejidad de estos.

## **Respaldo de datos del *namenode***



**Copia** de los **archivos persistentes** de los metadatos a otros nodos o a un sistema de ficheros externos como NFS.

- ▶ **Namenode secundario:** en otra máquina física que no es realmente un *namenode*, un proceso va fusionando los cambios que indica el *log* de edición respecto a la imagen del *namenode*, para tener una fotografía actualizada del estado de los ficheros en el clúster. En caso de fallo del *namenode*, se transfieren a esa máquina los posibles metadatos adicionales que estuviesen replicados en NFS y se empieza a utilizar esa máquina como el *namenode* activo. Esto es un proceso manual que requiere unos treinta minutos, durante los cuales no podemos utilizar HDFS. Al ser tan largo el intervalo de pérdida de servicio, no se considera un mecanismo de alta disponibilidad.

## Alta disponibilidad del *namenode*

Se utilizan un par de *namenodes*, uno de ellos denominado activo y el otro en *stand by*. Ambos comparten un *log* de edición en un sistema de almacenamiento externo y que posee también alta disponibilidad. El *log* va siendo modificado por el *namenode* activo, pero el *namenode* en *stand by* lo va leyendo y va aplicando esos cambios en sus propios metadatos, para estar siempre actualizado respecto al *namenode* activo. Los *datanodes* reportan la información a ambos para monitorizar el estado. En caso de fallo del *namenode* activo, el que se encontraba en *stand by* pasa inmediatamente a activo. Como señalábamos, estará actualizado, puesto que todas las operaciones realizadas por las aplicaciones cliente (las que generan peticiones de lectura o escritura en HDFS) son recibidas siempre por ambos *namenodes*. En la práctica, este proceso apenas lleva un minuto hasta que se restablece el servicio normal gracias al nuevo *namenode*.

## Escalando el *namenode*: *namenodes federados*

Este mecanismo se utiliza cuando el *namenode* se encuentra saturado y cerca de llenarse. No es un mecanismo de protección contra fallos como tal, aunque también

cumple esa función. Consiste en que varios *namenodes* que funcionan a la vez se encargan de directorios distintos del sistema de archivos, sin solapamiento. Por ejemplo: los metadatos relativos a todo el árbol de directorios que cuelga de /user se pueden almacenar en un *namenode*, mientras que todo el árbol que cuelga de /share lo puede hacer en otro. De esta manera, el posible fallo de un *namenode* no afecta en nada a los archivos o directorios que no cuelgan de esa jerarquía. Los *datanodes* sí pueden almacenar indistintamente bloques de archivos de varios *namespaces* (es decir, de varios subárboles).

## Proceso de lectura y escritura en HDFS

### Proceso de lectura

El proceso que se lleva a cabo cuando un comando necesita leer un archivo de HDFS es el siguiente (figura 2):

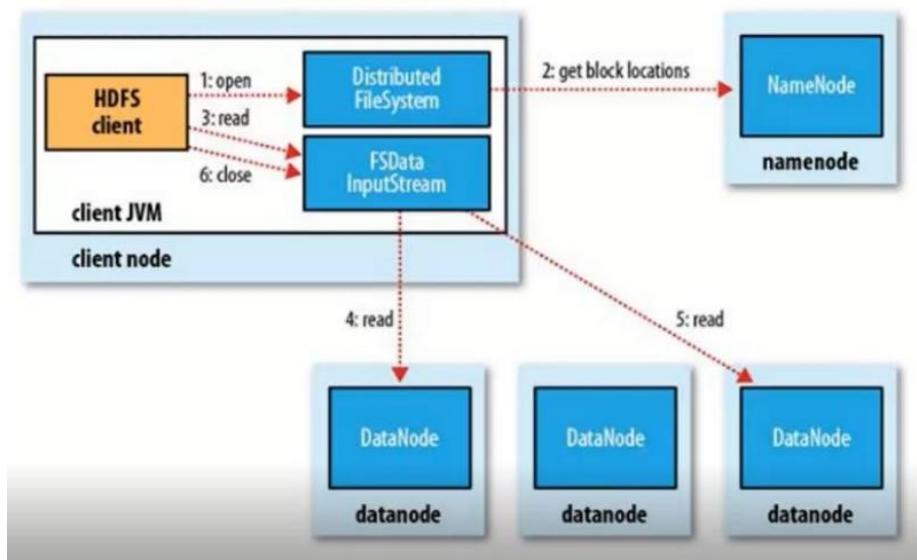


Figura 2. Proceso de lectura de datos de HDFS. Fuente: <https://programmerclick.com/article/8300156830/>

- ▶ El programa cliente (el cual implementa el comando de HDFS que requiere lectura) solicita al *namenode* que le proporcione un fichero. Esto se lleva a cabo mediante una llamada a procedimiento remoto (RPC o *remote procedure call*) desde el nodo

donde se ejecuta el cliente al *namenode*.

- ▶ El *namenode* consulta en la tabla de metadatos los bloques que componen el fichero y su localización en el clúster.
- ▶ El *namenode* devuelve al cliente una lista de bloques, así como los equipos en los que puede encontrar cada uno de ellos.
- ▶ El cliente contacta con los nodos que almacenan los bloques que forman el archivo para ir obteniendo dichos bloques, aunque esto es transparente a la aplicación, ya que simplemente lee datos del objeto `FSDataInputStream` y los percibe como un flujo continuo. Es este objeto el que, ocasionalmente, vuelve a contactar con el *namenode* para obtener la mejor ubicación de los siguientes bloques de datos, según se van necesitando al realizar lecturas.
- ▶ Finalmente, el cliente compone el archivo.

Es importante resaltar que **el *namenode* no devuelve los bloques de datos** (no pasan por él) porque se convertiría en cuello de botella cuando hubiese múltiples clientes leyendo al mismo tiempo. Es el propio cliente (de forma transparente para él, pues se gestiona a través del objeto `FSDataInputStream` que nos ha devuelto la API de HDFS) quien se dirige a los *datanodes* que le ha indicado el *namenode* para solicitarles los bloques de datos que desea leer.

## Proceso de escritura

Lleva a cabo unos pasos análogos (figura 3):

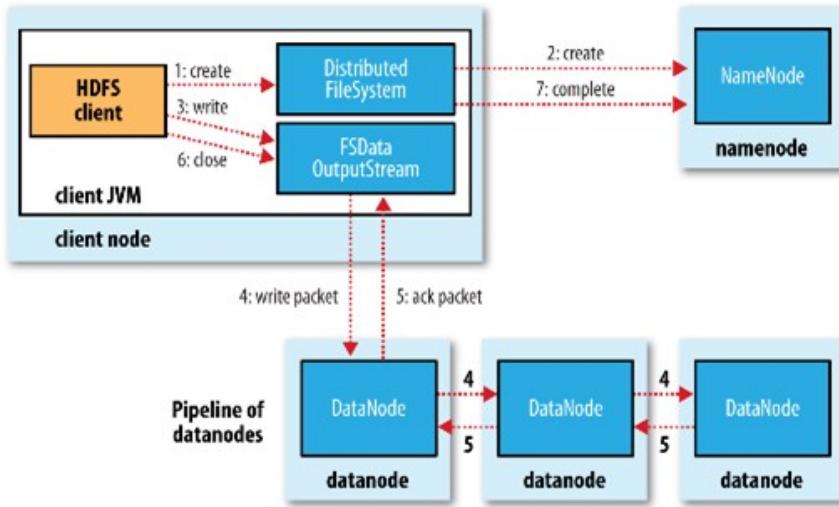


Figura 3. Proceso de escritura de datos en HDFS. Fuente:

<https://programmerclick.com/article/8300156830/>

- ▶ El cliente solicita al *namenode*, mediante RPC, realizar una escritura de un archivo sin bloques de datos asociados.
- ▶ El *namenode* realiza algunas comprobaciones previas, para comprobar que se puede escribir el fichero y que el cliente tiene permisos para hacerlo.
- ▶ El objeto *FSDataOutputStream* devuelto al cliente partitiona el fichero en bloques, los encola y los va escribiendo de forma secuencial. Para cada bloque, el *namenode* le proporciona una lista de *datanodes* en los que se escribirá. Periódicamente volverá a contactar con el *namenode* para obtener la ubicación física en la que ir escribiendo los siguientes bloques de datos.
- ▶ Para cada bloque de datos, el cliente contacta con el primer *datanode* de la lista, a fin de pedirle que escriba el bloque. Este *datanode* se encargará de propagar el bloque al segundo *datanode*, etc.
- ▶ El último *datanode* en escribir el bloque devolverá una confirmación (ack) hacia atrás, hasta que el primer *datanode* mande la confirmación al cliente.
- ▶ Una vez que el cliente ha escrito todos los bloques, envía una confirmación al

*namenode* de que el proceso se ha completado.

Para cada bloque, los *datanodes* (varios, debido al factor de replicación) que lo guardarán forman un *pipeline*. El primer *datanode* escribe el bloque y lo reenvía al segundo, y así sucesivamente. Según se van escribiendo, se devuelve una señal de confirmación *ack*, que también se almacena en una cola. En caso de fallo de algún *datanode* durante la escritura, se establecen mecanismos para que el resto de *datanodes* del *pipeline* no pierdan ese paquete y asegurar la replicación. Son los propios *datanodes* los que gestionan la replicación de cada bloque, sin intervención del *namenode*, para evitar cuellos de botella. Solo se necesita la intervención del *namenode* de forma ocasional, con el fin de preguntarle dónde escribir el siguiente bloque de datos.

## 2.4. Comandos de HDFS más frecuentes

En todos los comandos, se puede usar `hadoop dfs` o `hdfs dfs` indistintamente, aunque la segunda opción está más extendida en la actualidad. Recordemos que lo que estamos haciendo es ejecutar un programa cliente llamado `hdfs` desde cualquiera de los nodos que forman parte de HDFS.

```
[root@sandbox ~]# hadoop fs
Usage: hadoop fs [generic options]
      [-appendToFile <localsrc> ... <dst>]
      [-cat [-ignoreCrc] <src> ...]
      [-checksum <src> ...]
      [-chgrp [-R] GROUP PATH...]
      [-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]
      [-chown [-R] [OWNER][:[GROUP]] PATH...]
      [-copyFromLocal [-f] [-p] [-l] <localsrc> ... <dst>]
      [-copyToLocal [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
      [-count [-q] [-h] [-v] [-t <storage type>]] <path> ...
      [-cp [-f] [-p] [-p[topax]] <src> ... <dst>]
      [-createSnapshot <snapshotDir> [<snapshotName>]]
      [-deleteSnapshot <snapshotDir> <snapshotName>]
      [-df [-h] [<path> ...]]
      [-du [-s] [-h] <path> ...]
      [-expunge]
      [-find <path> ... <expression> ...]
      [-get [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
      [-getfacl [-R] <path>]
      [-getfattr [-R] {-n name | -d} {-e en} <path>]
      [-getmerge [-nl] <src> <localdst>]
      [-help [cmd ...]]
      [-ls [-C] [-d] [-h] [-q] [-R] [-t] [-S] [-r] [-u] [<path> ...]]
```

Es importante resaltar que, a diferencia del sistema de ficheros local de Linux o Windows, HDFS no cuenta con el comando `cd` para acceder a un directorio, ya que no existe el concepto de «directorio actual» o directorio en el que estamos situados. Por ese motivo, todos los comandos requieren especificar rutas completas.

A continuación, se expone un resumen de los comandos más frecuentes:

- ▶ `hdfs dfs -ls /ruta/directorio`: muestra el contenido de un directorio de HDFS.

```
[root@sandbox ~]# hadoop fs -ls /
Found 12 items
drwxrwxrwx  - yarn  hadoop      0 2016-10-25 08:10 /app-logs
drwxr-xr-x  - hdfs  hdfs       0 2016-10-25 07:54 /apps
drwxr-xr-x  - yarn  hadoop      0 2016-10-25 07:48 /ats
drwxr-xr-x  - hdfs  hdfs       0 2016-10-25 08:01 /demo
drwxr-xr-x  - hdfs  hdfs       0 2016-10-25 07:48 /hdp
drwxr-xr-x  - mapred  hdfs     0 2016-10-25 07:48 /mapred
drwxrwxrwx  - mapred  hadoop    0 2016-10-25 07:48 /mr-history
drwxr-xr-x  - hdfs  hdfs       0 2016-10-25 07:47 /ranger
drwxrwxrwx  - spark   hadoop    0 2017-02-02 11:46 /spark-history
drwxrwxrwx  - spark   hadoop    0 2016-10-25 08:14 /spark2-history
drwxrwxrwx  - hdfs   hdfs       0 2016-10-25 08:11 /tmp
drwxr-xr-x  - hdfs   hdfs       0 2016-10-25 08:11 /user
[root@sandbox ~]# █
```

- ▶ hdfs dfs –mkdir /ruta/nuevodirectorio : crea un nuevo directorio, nuevodirectorio , como subdirectorio del directorio /ruta.

```
[root@sandbox ~]# hadoop fs -mkdir /raul
[root@sandbox ~]# hadoop fs -ls /
Found 13 items
drwxrwxrwx  - yarn  hadoop      0 2016-10-25 08:10 /app-logs
drwxr-xr-x  - hdfs  hdfs       0 2016-10-25 07:54 /apps
drwxr-xr-x  - yarn  hadoop      0 2016-10-25 07:48 /ats
drwxr-xr-x  - hdfs  hdfs       0 2016-10-25 08:01 /demo
drwxr-xr-x  - hdfs  hdfs       0 2016-10-25 07:48 /hdp
drwxr-xr-x  - mapred  hdfs     0 2016-10-25 07:48 /mapred
drwxrwxrwx  - mapred  hadoop    0 2016-10-25 07:48 /mr-history
drwxr-xr-x  - hdfs  hdfs       0 2016-10-25 07:47 /ranger
drwxr-xr-x  - root   hdfs      0 2017-02-02 11:48 /raul
drwxrwxrwx  - spark   hadoop    0 2017-02-02 11:48 /spark-history
drwxrwxrwx  - spark   hadoop    0 2016-10-25 08:14 /spark2-history
drwxrwxrwx  - hdfs   hdfs       0 2016-10-25 08:11 /tmp
drwxr-xr-x  - hdfs   hdfs       0 2016-10-25 08:11 /user
[root@sandbox ~]# █
```

- ▶ hdfs dfs –copyFromLocal ruta/local/fichero.txt /ruta/hdfs/ : copia el fichero fichero.txt presente en el sistema de archivos local a la ubicación remota de HDFS situada en /ruta/hdfs/fichero.txt. La ruta del fichero local sí puede ser una ruta relativa, ya que se refiere al sistema de archivos local, por lo que no es imprescindible que empiece por /. hdfs dfs –copyFromLocal <localsrc> <dst>

```
[root@sandbox raul]# ls -la
total 12
drwxr-xr-x 2 root root 4096 Feb  2 11:50 .
dr-xr-x--- 1 root root 4096 Feb  2 11:49 ..
-rw-r--r-- 1 root root  16 Feb  2 11:50 MyBigFile.txt
[root@sandbox raul]# hadoop fs -copyFromLocal MyBigFile.txt /raul/MyBigFileinHDFS.txt
[root@sandbox raul]# hadoop fs -ls /raul/
Found 1 items
-rw-r--r-- 1 root hdfs      16 2017-02-02 11:51 /raul/MyBigFileinHDFS.txt
[root@sandbox raul]# █
```

- ▶ hdfs dfs –copyToLocal /ruta/hdfs/fichero.txt ruta/local : copia un fichero existente en HDFS

(en la ruta remota `/ruta/hdfs/fichero.txt`) al sistema de archivos local (concretamente, a la ubicación de destino `ruta/local/fichero.txt`). Sintaxis: `hdfs dfs –copyToLocal <src> <localdst>`.

```
[root@sandbox raul]# hadoop fs -ls /demo/data/CDR/
Found 2 items
-rwx----- 1 hdfs hdfs    710436 2016-10-25 08:01 /demo/data/CDR/cdrs.txt
-rwx----- 1 hdfs hdfs    68095 2016-10-25 08:01 /demo/data/CDR/recharges.txt
[root@sandbox raul]# hadoop fs -copyToLocal /demo/data/CDR/cdrs.txt myCdrs.txt
[root@sandbox raul]# ls -la
total 708
drwxr-xr-x 2 root root 4096 Feb  2 11:55 .
dr-xr-x--- 1 root root 4096 Feb  2 11:49 ..
-rw-r--r-- 1 root root   16 Feb  2 11:50 MyBigFile.txt
-rw-r--r-- 1 root root 710436 Feb  2 11:55 myCdrs.txt
[root@sandbox raul]#
```

- ▶ `hdfs dfs –tail /ruta/hdfs/fichero.txt`: muestra en pantalla la parte final del contenido del archivo presente en HDFS.

```
[root@sandbox raul]# hadoop fs -tail /demo/data/CDR/recharges.txt
00
6641609561|20130209|094637|3|100
6650359180|20130209|125420|3|100
6638378345|20130209|121231|3|300
6659538250|20130209|191504|3|100
6662032971|20130209|211136|3|500
8333654388|20130209|100458|3|100
6623568405|20130209|121240|3|100
```

- ▶ `hdfs dfs –cat /ruta/hdfs/fichero.txt`: muestra en pantalla todo el contenido del archivo presente en HDFS. Debe usarse con cuidado, ya que lo habitual es que los ficheros sean grandes y el comando imprime todo el fichero. Para paginar la visualización, se recomienda redirigir (con la barra vertical |) la salida de este comando hacia el comando more del sistema de archivos local de Linux: `hdfs dfs –cat /ruta/hdfs/fichero.txt | more`

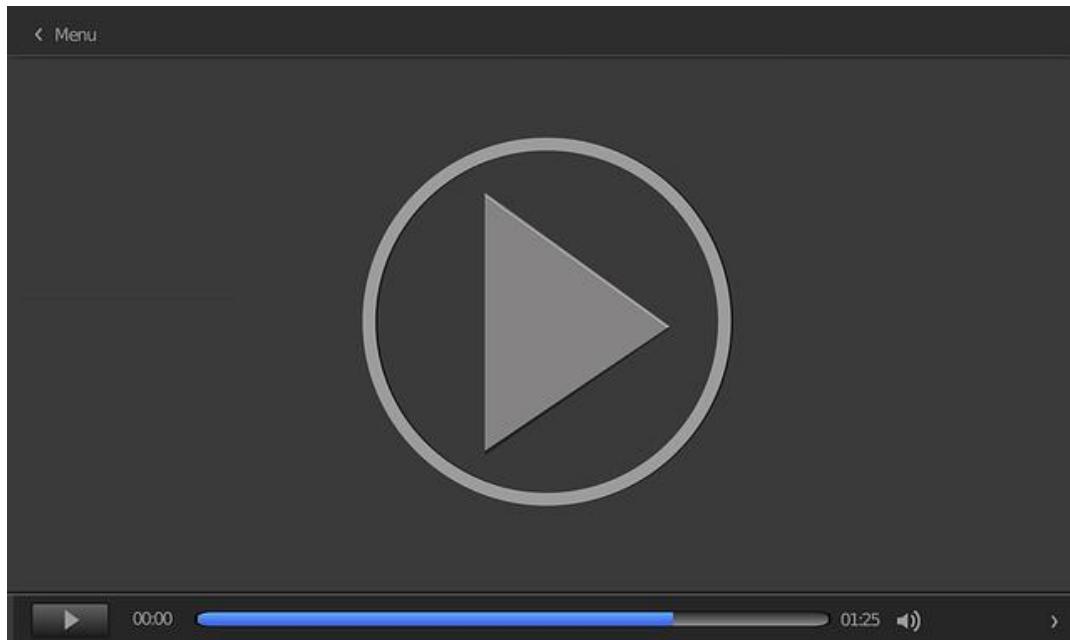
```
[root@sandbox raul]# hadoop fs -cat /demo/data/CDR/recharges.txt | more
PHONE|DATE|CHANNEL|PLAN|AMOUNT
7852121521|20130209|090721|3|100
7642140929|20130209|181648|3|100
7552204414|20130209|224815|3|100
7785846460|20130209|173731|3|100
7972930496|20130209|003527|3|100
7782957598|20130209|200016|3|100
7352795440|20130209|000429|3|100
```

- ▶ hdfs dfs –cp /ruta/hdfs/origen/fichero.txt /ruta/hdfs/ destino/copiado.txt : hace una copia del fichero situado en HDFS, en /ruta/origen/fichero.txt , en la ruta destino (también de HDFS) /ruta/hdfs/destino/copiado.txt . Este comando no interactúa con el sistema de archivos local de la máquina; es una copia de un origen de HDFS a un destino también de HDFS. Sintaxis: hdfs dfs –cp <src> <dst>.
- ▶ hdfs dfs –mv /ruta/original.txt /ruta/nuevo.txt o bien hdfs dfs –mv /ruta/fichero1 /ruta/nueva/ : en el primer caso, renombra un fichero existente en HDFS. En el segundo, lo mueve (sin copiar) de una ubicación de HDFS a otra distinta.
- ▶ hdfs dfs –rm /ruta/fichero.txt : borra un fichero presente en HDFS. Es posible borrar una carpeta completa de forma recursiva (es decir, con todos sus subdirectorios) mediante la opción –r : hdfs dfs –rm –r /ruta/carpeta/ ¡Cuidado!

```
[root@sandbox raul]# hadoop fs -ls /demo/data/CDR/
Found 2 items
-rwx----- 1 hdfs hdfs 710436 2016-10-25 08:01 /demo/data/CDR/cdrs.txt
-rwx----- 1 hdfs hdfs 68095 2016-10-25 08:01 /demo/data/CDR/recharges.txt
[root@sandbox raul]# hadoop fs -cp /demo/data/CDR/cdrs.txt /demo/data/CDR/cdrs2.txt
[root@sandbox raul]# hadoop fs -ls /demo/data/CDR/
Found 3 items
-rwx----- 1 hdfs hdfs 710436 2016-10-25 08:01 /demo/data/CDR/cdrs.txt
-rw-r--r-- 1 root hdfs 710436 2017-02-02 12:01 /demo/data/CDR/cdrs2.txt
-rwx----- 1 hdfs hdfs 68095 2016-10-25 08:01 /demo/data/CDR/recharges.txt
[root@sandbox raul]# hadoop fs -rm /demo/data/CDR/cdrs2.txt
17/02/02 12:01:52 INFO fs.TrashPolicyDefault: Moved: 'hdfs://sandbox.hortonworks.com:8020/user/root/.Trash/current/demo/data/CDR/cdrs2.txt'
[root@sandbox raul]# hadoop fs -ls /demo/data/CDR/
Found 2 items
-rwx----- 1 hdfs hdfs 710436 2016-10-25 08:01 /demo/data/CDR/cdrs.txt
-rwx----- 1 hdfs hdfs 68095 2016-10-25 08:01 /demo/data/CDR/recharges.txt
[root@sandbox raul]#
```

- ▶ hdfs dfs –chmod <permisos> /ruta/hdfs/fichero.txt: es un comando similar a chmod del sistema de archivos de Linux, que se utiliza para cambiar los permisos en un archivo existente en HDFS.
- ▶ hdfs dfs –chown usuario /ruta/fichero.txt : cambia el dueño de un usuario (solo si el usuario que ejecuta la orden tiene permisos para hacerlo).

Para reforzar el conocimiento y la implementación de los comandos de HDFS más frecuentes, visualiza el siguiente vídeo.



---

Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=c9996fae-fbd2-4eb1-ab82-ac6e00b69546>

---

## 2.5. Programación distribuida y MapReduce

Una vez que Google tenía resuelto el problema del almacenamiento de ficheros masivos en el sistema de ficheros distribuido Google File System (GFS), Dean y Ghemawat publicaron un nuevo artículo (2008) sobre cómo aprovechar los *datanodes* para procesar estos ficheros que estaban almacenados de forma distribuida, particionados en bloques. Para ello, desarrollaron un paradigma de programación llamado MapReduce, que consiste en una manera abstracta de abordar los problemas para que puedan ser implementados y ejecutados sobre un clúster de ordenadores. Junto con el artículo, también liberaron una biblioteca de programación en lenguaje Java que implementaba este paradigma.

De forma más precisa, podemos definir MapReduce como:

Modelo abstracto de programación general e implementación del  
modelo como bibliotecas de programación, para procesamiento paralelo  
y distribuido de grandes *datasets*, inspirado en la técnica «divide y  
vencerás».

La gran ventaja que proporciona tanto el modelo como la implementación que suministraron los autores es que **abstira al programador de todos los detalles relativos a hardware, redes y comunicación entre los nodos**, con el fin de que pueda centrarse solamente en el desarrollo de *software* para solucionar su problema.

El punto de partida son archivos muy grandes almacenados (por bloques) en HDFS (en aquel momento, aún era GFS). ¿Podríamos aprovechar las CPU de los *datanodes* del clúster para procesar en paralelo (simultáneamente) los bloques de un archivo? No queremos mover datos (el tráfico de red es muy lento): **llevamos el cómputo (el código fuente de nuestro programa) al lugar donde están**

**almacenados los datos** (es decir, a los *datanodes*, y no al revés, como ocurría en el modelo tradicional de aplicación). Las CPU de los *datanodes* van a procesar (preferentemente) los datos que hay en ese *datanode*.

En el paradigma MapReduce, el usuario solo necesita escribir dos funciones (enfoque «divide y vencerás»):

- ▶ **Mapper:** función escrita por el usuario e invocada por el *framework* en paralelo (simultáneamente) sobre cada bloque de datos de entrada (que generalmente coincide con un bloque de HDFS). De este modo, se generan resultados intermedios, **que se presentan siempre en forma de (clave, valor)** y son escritos también en el disco local.
- ▶ **Reducer:** se aplica en paralelo para cada grupo creado por la función Mapper. En concreto, esta función se llama una vez para cada clave única generada de la salida de la función Mapper, junto con la cual se pasan todos los valores asociados que comparte.

## Ejemplo: el problema de contar ocurrencias de palabras con MapReduce

Supongamos que queremos resolver un problema que consiste en, dado un texto, saber cuántas veces aparece en él cada palabra. Nuestro punto de partida es un fichero que contiene el texto y que está almacenado en HDFS. Por tanto, está particionado en varios bloques, cada uno de los cuales contiene un fragmento del texto. Asumimos que el texto está almacenado en formato ASCII, es decir, el fichero (y, por extensión, cada uno de sus bloques) contiene líneas de texto, tal como se indica en los rectángulos azules de la figura 4.

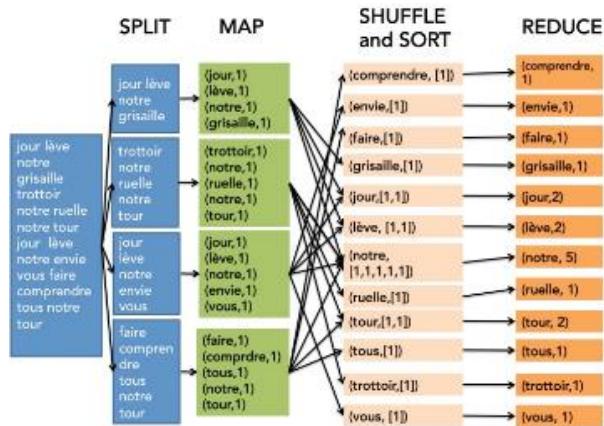


Figura. 4. Representación del problema de contar ocurrencias resuelto con MapReduce. Fuente:  
<https://openclassrooms.com/fr/courses/4297166-realisez-des-calculs-distribués-sur-des-données-massives/4308626-divisez-et-distribuez-pour-regner#/id/r-4362891>

En el modelo MapReduce, el programador (usuario de la API) solamente necesita escribir dos funciones. En primer lugar, una función llamada `map`, que reciba una línea de texto (cada línea de texto de los rectángulos azules, que representan cada uno un bloque de HDFS) y dé lugar, como resultado, a las tuplas que se muestran en los rectángulos verdes, de manera que cada rectángulo azul origina un rectángulo verde.

A continuación, las tuplas que genera la fase `map` son enviadas por la red que conecta los nodos del clúster: la librería de MapReduce, de forma automática y transparente al programador, lleva a cabo la operación `shuffle and sort`, mostrada en rosa. Esta, a partir de los resultados en verde, genera tuplas formadas por una palabra y una lista asociada (más adelante, explicaremos su significado). Lo que está sucediendo es que el propio *framework* está agrupando todas las tuplas que tienen la misma clave y está creando una lista con todos los valores asociados a esa clave común. Esto lo repite por cada clave diferente que encuentre en las tuplas generadas por la función `map` del usuario.

Para cada uno de estos agrupamientos mostrados en rosa, el *framework* invoca

automáticamente a la segunda función que el programador debe escribir: la función `reduce`. Esta es recibida por las tuplas mostradas en rosa (formadas por una palabra y la lista con el número de ocurrencias asociadas a ella) y genera, para cada una de ellas, un resultado como el mostrado en los rectángulos naranjas.

La función `map` que implementa el usuario debe recibir como entrada una tupla (`clave_entrada, valor`). En este problema, `clave_entrada` es el número de línea (ignorado) y `valor` representa una línea completa de texto. Es el *framework* el que, de forma automática, invoca, tantas veces como sea necesario, la función que ha implementado el usuario y le pasa los argumentos que hemos indicado. La invocación y ejecución de la función `map` se lleva a cabo de manera distribuida, en cada nodo del clúster (en los *datanodes*, que es donde residen los bloques de HDFS que contienen los fragmentos del texto que actúa como datos de entrada).

La implementación de la función `map` del usuario para este problema concreto podría dividir la línea de texto en palabras y, para cada palabra que forma parte de la línea de texto, devolver la tupla (`palabra, 1`), que indica que «palabra» ha aparecido una vez. Ejemplo: (`hola, 1`), (`el, 1`), (`el, 1`). La palabra es la `out_key` y el valor siempre es 1.

La función `reduce` que el usuario ha de implementar es también invocada automáticamente por el *framework*. Para ello, se toman como datos de entrada cada una de las claves y la lista de valores asociados a ellas obtenidas en la etapa anterior. Esta lista de valores está formada por el campo `valor` de todas las tuplas que comparten la misma clave, tal como las ha generado la fase `map`. La implementación de la función `reduce` debe agregar resultados (sumar las ocurrencias de una misma palabra). Ejemplo: la función `reduce` recibe (`reduce, [1, 1, 1, 1, 1, 1, 1]`) y da como resultado: (`hola, 7`).

## Inconvenientes de MapReduce

MapReduce permite procesar los datos almacenados de manera distribuida en el

sistema de archivos (por ejemplo, HDFS) de un clúster de ordenadores. Esto ayuda a resolver todo tipo de problemas, pese a que no siempre es sencillo pensar la solución en términos de operaciones `map` y `reduce` encadenadas. En este sentido, se dice que es de propósito general, a diferencia de, por ejemplo, el lenguaje SQL, que está orientado específicamente a realizar consultas sobre los datos. Implementar en SQL algoritmos tales como un método de ordenación o el algoritmo de Dijkstra para encontrar caminos mínimos en un grafo no sería posible. Sin embargo, MapReduce presenta varios inconvenientes, algunos muy serios:

- ▶ El resultado de la fase `map` (tuplas) se escribe en el disco duro de cada nodo, como resultado intermedio. Los accesos a un disco duro son aproximadamente un orden de magnitud (es decir, diez veces) más lentos que los accesos a la memoria principal (RAM) de cada nodo, por lo que estamos penalizando el rendimiento debido a cómo está estructurado el propio *framework*.
- ▶ Después de la fase `map`, hay tráfico de red obligatoriamente (movimiento de datos, conocido como *shuffle*). De hecho, el movimiento de datos forma parte como una etapa obligada en el *framework* de MapReduce. En cualquier aplicación distribuida, el movimiento de datos de un nodo a otro constituye un cuello de botella y es una operación que debe evitarse si es posible.
- ▶ Por otro lado, y relacionado con el punto anterior, para mover datos se necesita, en primer lugar, escribirlos temporalmente en el disco duro de la máquina origen, enviarlos por la red y luego escribirlos temporalmente en el disco duro de la máquina destino (el *shuffle* siempre va de disco duro a disco duro) para, finalmente, pasarlo a la memoria principal de dicho nodo.
- ▶ Estos dos inconvenientes se acentúan especialmente cuando el algoritmo que queremos implementar sobre un clúster es de tipo iterativo, es decir, requiere varias pasadas sobre los mismos datos para ir convergiendo. Este tipo de procesamiento es típico en algoritmos de *machine learning*, que es uno de los que más se puede beneficiar del procesamiento de grandes conjuntos de datos para extraer

conocimiento. Se comprobó que sus implementaciones con MapReduce no eran eficientes debido a la propia concepción del *framework*.

- ▶ Finalmente, enfocar cualquier problema en términos de operaciones map y reduce encadenadas no siempre es fácil ni intuitivo para un desarrollador, y la solución resultante puede ser difícil de mantener si no está bien documentada.

## 2.6. Referencias bibliográficas

Dean, J. y Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1), 107-113. <https://doi.org/10.1145/1327452.1327492>

Ghemawat, S., Gobioff, H. y Leung, S-T. (2003). The Google File System. *A CM SIGOPS Operating Systems Review*, 37(5), 29-43. <https://doi.org/10.1145/1165389.945450>

## Guía de usuario HDFS

---

Hadoop Apache. (2020). *HDFS Users Guide*. The Apache Software Foundation. <http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html>

---

Sin duda, en la página oficial de Hadoop Apache podréis encontrar la documentación más actualizada sobre HDFS, con toda la información sobre sus características y su uso.

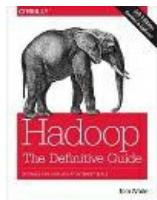
## Glosario de términos de la empresa Databricks, creadores de Apache Spark

Databricks (2021). Hadoop. *Databricks*. <https://databricks.com/glossary/hadoop>

Aquí puedes consultar una definición completa sobre Hadoop. Son también interesantes todos los demás términos contenidos en el glosario, todos relacionados con las tecnologías *big data*.

## Hadoop: the definitive guide

White, T. (2015). *Hadoop: the definitive guide* (4. a edición). O'Reilly.



Guía que incluye las principales tecnologías de Hadoop descritas en detalle. Para este tema, podéis centraros en el capítulo 3. El capítulo 2 es interesante, pero ha quedado obsoleto.

## Arquitectura de HDFS en detalle

Chansler, R., Kuang, H., Radia, S., Shvachko, K. y Srinivas, S. (s. f.). The Hadoop distributed file system. En A. Brown y G. Wilson (eds.), *The architecture of open source applications. Elegance, evolution, and a few fearless hacks*. Aosa Book. <http://www.aosabook.org/en/hdfs.html>



En este capítulo, los autores describen la arquitectura de HDFS y narran su experiencia con este sistema para gestionar 40 *petabytes* de datos empresariales en la compañía Yahoo.

1. ¿Cuánto ocupa en total un archivo de 500 MB almacenado en HDFS, sin replicación, si se asume el tamaño de bloque por defecto?

  - A. Ocupará 500 MB.
  - B. Ocupará 512 MB, que son 4 bloques de 128 MB, y hay 12 MB desperdiciados.
  - C. Ocupará lo que resulte de multiplicar 500 MB por el número de *datanodes* del clúster.
2. ¿Cuál de las siguientes afirmaciones respecto a HDFS es cierta?

  - A. El tamaño de bloque debe ser siempre pequeño para no desperdiciar espacio.
  - B. El factor de replicación es configurable por fichero y su valor, por defecto, es 3.
  - C. Las dos respuestas anteriores son correctas.
3. ¿Qué afirmación es cierta sobre el proceso de escritura en HDFS?

  - A. El cliente manda al *namenode* el fichero, que, a su vez, se encarga de escribirlo en los diferentes *datanodes*.
  - B. El cliente escribe los bloques en todos los *datanodes* que le ha especificado el *namenode*.
  - C. El cliente escribe los bloques en un *datanode* y este envía la orden de escritura a los demás.
4. En un clúster de varios nodos donde no hemos configurado la topología...

  - A. Es imposible que dos réplicas del mismo bloque caigan en el mismo nodo.
  - B. Es imposible que dos réplicas del mismo bloque caigan en el mismo *rack*.
  - C. Las dos respuestas anteriores son falsas.

5. Cuando usamos *namenodes* federados...
  - A. Cada *datanode* puede albergar datos de uno de los subárboles.
  - B. La caída de un *namenode* no tiene ningún efecto en el clúster.
  - C. Ninguna de las respuestas anteriores es correcta.
6. ¿Por qué se dice que HDFS es un sistema escalable?
  - A. Porque reemplazar un nodo por otro más potente no afecta a los *namenodes*.
  - B. Porque un clúster es capaz de almacenar datos a gran escala.
  - C. Porque se puede aumentar la capacidad del clúster añadiendo más nodos.
7. ¿Qué tipo de uso suele darse a los ficheros de HDFS?
  - A. Ficheros de cualquier tamaño que se almacenan temporalmente.
  - B. Ficheros de gran tamaño que se crean, no se modifican, y sobre los que se realizan frecuentes lecturas.
  - C. Ficheros de gran tamaño que suelen modificarse constantemente.
8. La alta disponibilidad de los *namenodes* de HDFS implica que...
  - A. La caída de un *namenode* apenas deja sin servicio al sistema de ficheros durante un minuto antes de que otro entre en acción.
  - B. Es posible escalar los *namenodes* añadiendo más nodos.
  - C. La caída de un *datanode* deja sin servicio al sistema durante unos pocos segundos hasta que este es sustituido.
9. El comando de HDFS para moverse a la carpeta /mydata es...
  - A. hdfs dfs –cd /mydata.
  - B. hdfs dfs –ls /mydata.
  - C. No existe ningún comando equivalente en HDFS.

**10.** ¿Qué inconveniente presenta MapReduce?

- A. No es capaz de procesar datos distribuidos cuando son demasiado grandes.
- B. Entre las fases map y reduce , siempre lleva a cabo escrituras a disco y movimiento de datos entre máquinas.
- C. Es una tecnología propietaria y no es código abierto.

Ingeniería para el Procesado Masivo de Datos

---

## Tema 3. Spark I

# Índice

## Esquema

### Ideas clave

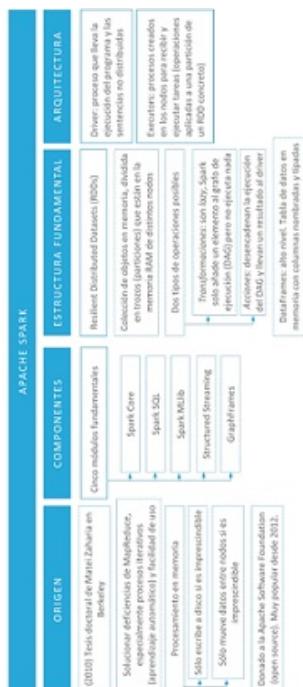
- 3.1. Introducción y objetivos
- 3.2. Apache Spark
- 3.3. Componentes de Spark
- 3.4. Arquitectura de Spark
- 3.5. Resilient distributed datasets (RDD)
- 3.6. Transformaciones y acciones
- 3.7. Jobs, stages y tasks
- 3.8. Ejemplo completo con RDD
- 3.9. Referencias bibliográficas

### A fondo

- Documentación oficial de Apache Spark
- Hadoop: the end of an era
- DATA + AI Summit
- Spark: the definitive guide
- High performance Spark

### Test

# Esquema



## 3.1. Introducción y objetivos

En la última sección del tema anterior, expusimos las deficiencias del paradigma de programación distribuida MapReduce. En la presente lección, conoceremos Apache Spark, un nuevo *framework* de programación distribuida más intuitivo y rápido que nació para tratar de resolverlas.

Los objetivos que persigue este tema son:

- ▶ Entender por qué Spark es superior a MapReduce atendiendo a su diseño.
- ▶ Identificar los módulos que componen Spark y el propósito de cada uno.
- ▶ Conocer la arquitectura y el funcionamiento interno de Spark.
- ▶ Practicar con algunas funciones típicas de procesamiento de datos con Spark.

## 3.2. Apache Spark

En el año 2009, surgió en Berkeley, EE. UU., una nueva propuesta para paliar estas deficiencias, que, poco a poco, ha ido imponiéndose hasta reemplazar completamente a Hadoop (más concretamente, a MapReduce).

**Apache Spark es un motor unificado de cálculo en memoria y un conjunto de bibliotecas para procesamiento paralelo y distribuido de datos en clústeres de ordenadores.**

Analicemos la definición:

- ▶ Es un *motor* de cálculo, esto es, un *framework* de propósito general orientado a resolver cualquier tipo de problema y con el que se puede implementar cualquier algoritmo.
- ▶ *Unificado*: el motor es único e independiente de cómo utilicemos Spark:
  - ▶ Desde la API de DataFrames (para lenguaje R, Python, Java y Scala).
  - ▶ Desde herramientas externas que lanzan consultas SQL contra Spark (Hive, herramientas de *business intelligence* conectadas a Spark como Tableau, PowerBI, Microstrategy, QlikSense, etc.).
  - ▶ Desde una instrucción de la API de programación que recibe una consulta SQL como *string*, tan compleja como sea necesario.
  - ▶ Cualquiera de las opciones anteriores se traduce a un grafo de tareas al que Spark aplica optimizaciones de código automáticamente; por tanto, todos (API, aplicaciones BI, SQL, etc.) se benefician de ellas.
- ▶ *En memoria*: todos los cálculos se llevan a cabo en memoria y solo se escriben

resultados a disco (parciales o finales) cuando el usuario lo indica explícitamente (operación de guardado) o cuando la operación indicada por el usuario requiere forzosamente realizar movimiento de datos entre nodos (*shuffle*, el cual se lleva a cabo desde el disco duro del emisor al disco duro del nodo receptor). Además, el movimiento de datos (y las operaciones de acceso a disco que conlleva) solo se produce cuando es irremediable, pero no de manera obligatoria, como ocurría en MapReduce. Esto consigue un rendimiento muy superior en tareas iterativas (varias pasadas sobre los mismos datos, p.ej., algoritmos de *machine learning*).

### 3.3. Componentes de Spark

Apache Spark consiste en una API (bibliotecas de programación) distribuida, mucho más intuitiva que MapReduce. Al igual que este, abstrae todos los detalles de comunicación de red entre las máquinas del clúster, pero, además, opera de manera similar a las consultas SQL tradicionales, como si los datos fuesen tablas (distribuidas). Esto la hace fácil de usar y de aprender.

Spark ofrece distintas API para cuatro lenguajes: Java (muy tediosa), Scala (la más utilizada en la actualidad por los ingenieros de datos para aplicaciones en producción), Python (paquete de Python PySpark, muy usado por científicos de datos, con una API casi idéntica a la de Scala que, en realidad, no es más que una capa adicional que termina llamando al código de Scala) y R (paquete de R SparkR, con una API muy diferente al resto, más cercana a los comandos típicos de R). En este tema, usaremos PySpark para todos los ejemplos.

La figura 1 muestra los principales componentes de Spark.

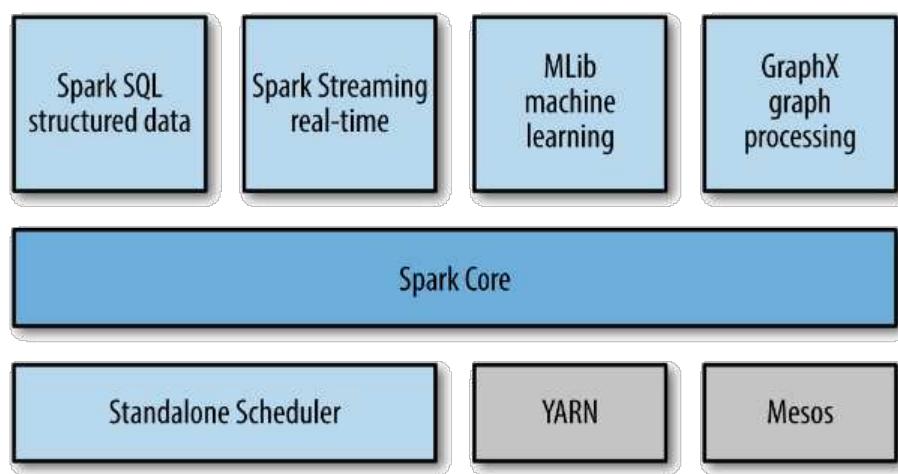


Figura 1. Módulos que componen Apache Spark. Fuente: Chambers y Zaharia 2018.

De estos, el principal es el módulo Spark Core. En él se encuentran las estructuras

de datos fundamentales de Spark, tales como los RDD (Resilient Distributed Dataset), de los que hablaremos más adelante, y las operaciones que se puede llevar a cabo con él. En última instancia, todas las operaciones que se pueden hacer con Spark, independientemente del módulo utilizado o la API donde se encuentren, se traducen automáticamente por parte de Spark a operaciones sobre RDD, que son las únicas que sabe ejecutar el motor.

Los tres módulos inferiores representan tres gestores de recursos de un clúster sobre los que puede ejecutarse Spark. Un gestor de recursos se encarga de asignar máquinas, CPU y memoria principal a Spark, de forma que disponga de un determinado número de nodos y de que, en cada uno, existan suficientes recursos para ejecutar la aplicación que hemos implementado con Spark.

El resto de módulos cumplen las siguientes funciones:

- ▶ Spark SQL y API estructurada es una API con una serie de funciones para manejar tablas de datos distribuidas, estructuradas en columnas con nombre y tipo, denominadas DataFrames. Un DataFrame proporcionan un nivel de abstracción adicional sobre un RDD, al cual recubre. Este módulo incluye también una función para ejecutar sentencias SQL sobre las mismas estructuras de datos distribuidas.
- ▶ Spark Streaming es el módulo para operar de manera distribuida sobre datos en tiempo real, según los vamos recibiendo (*stream* de datos). A partir de Spark 2.0, este módulo ha sido reemplazado por **Spark Structured Streaming**, que simplifica el procesamiento de flujos de datos en tiempo real.
- ▶ Spark MLlib contiene implementaciones distribuidas de algoritmos de Machine Learning.
- ▶ Spark GraphX es el módulo de procesamiento de grafos, representados mediante RDD. Contiene algunos algoritmos de camino mínimo y similares. Actualmente, **este módulo ha quedado obsoleto** y ha sido reemplazado *de facto* por el paquete GraphFrames, que representa un grafo como una pareja de DataFrames, con los

nodos y los arcos respectivamente. Empezó como un proyecto separado de Spark y, finalmente, ha sido integrado en las versiones más recientes.

### 3.4. Arquitectura de Spark

La figura 2 muestra la arquitectura de una aplicación que utiliza Spark al ejecutarse en un clúster. No debemos olvidar que, al escribir una aplicación en Spark, en realidad, estamos escribiendo una aplicación **secuencial** (no paralela) utilizando la biblioteca de Spark para el lenguaje en el que estemos programando (Java, Scala, Python o R). Cuando ejecutamos el código de este programa (proceso que se denomina *driver*), lo hacemos sobre una máquina concreta, que puede ser interna o externa al clúster. Por ejemplo, podríamos escribir un programa en Spark y ejecutarlo en nuestro ordenador portátil, y, desde aquí, el programa driver podría conectarse a un clúster de Spark remoto.

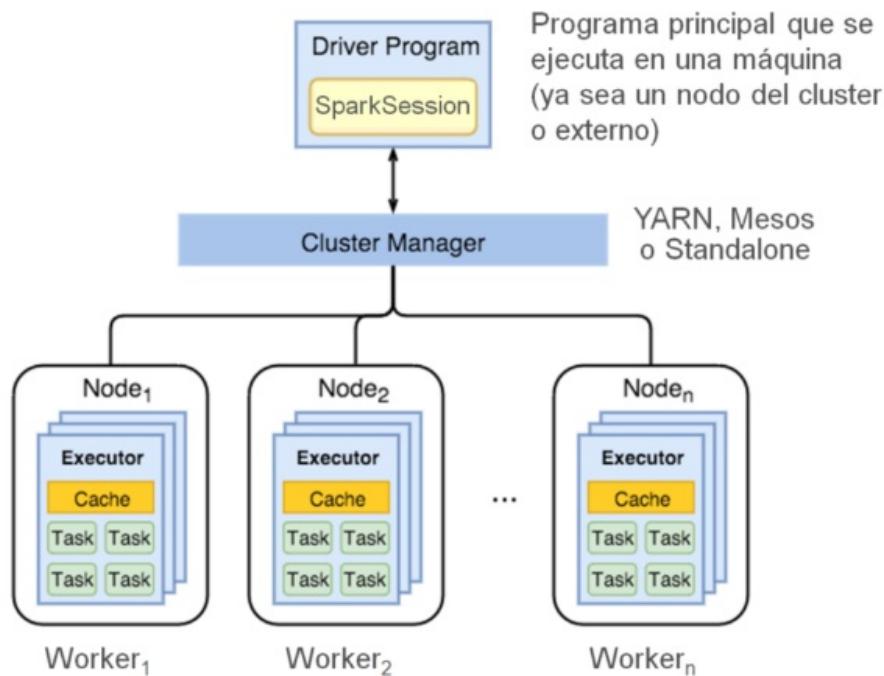


Figura 2. Arquitectura de una aplicación en Spark al ejecutarse en un clúster. Fuente:

<http://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/>

En el transcurso de la ejecución, normalmente necesitaremos crear un objeto denominado `sparkSession`, de la clase `sparkSession` de Spark. En el momento de crear

este objeto, hay que indicar dónde (en qué dirección IP y en qué puerto) existe un clúster de Spark configurado. Ciertas aplicaciones, como, por ejemplo, Jupyter Notebook (cuando está configurado para Spark) o la línea de comandos (intérprete) de Spark, tanto en Scala (*spark-shell*) como en Python (*pyspark-shell*), ya nos dan un objeto `sparkSession` creado al arrancar la aplicación. No obstante, al iniciar el *shell*, debemos pasárle la configuración relativa al clúster para que la `sparkSession` se cree correctamente.

De lo contrario, se estará ejecutando en local, sin conexión con un clúster. De esta manera, la `sparkSession` establece comunicación con el gestor de clúster para poder enviar tareas a los *workers* (nodos del clúster disponibles para Spark). A partir de este momento, podemos utilizar dichos recursos para las sentencias que requieran ejecución distribuida.

Es importante recalcar esto último: **la ejecución de un programa en Spark es secuencial**, en una sola máquina, tal como sería cualquier otro programa en el lenguaje elegido, **excepto cuando el flujo de programa llega a ciertas funciones específicas de Spark que desencadenan ejecución distribuida**, que son la mayoría (pero no todas) de las que forman parte de la API de Spark. Muchas se aplican sobre el objeto `sparkSession` y otras sobre estructuras de datos distribuidas que hayamos ido creando en el programa, como, por ejemplo, RDD o DataFrames de Spark.

En el momento de crear el objeto `sparkSession`, hemos indicado una configuración con el número de nodos, la memoria RAM y el número de cores físicos que necesitamos reservar en cada nodo. En un nodo, estos recursos constituyen lo que se denomina un *executor*: un proceso de la JVM (máquina virtual de Java) que se ejecuta en el nodo y que ocupa los recursos indicados (cores, RAM, disco duro).

El proceso ejecutor es creado por el gestor de clúster cuando arranca nuestra aplicación de Spark y muere cuando la aplicación finaliza (ya sea con éxito o por

alguna condición imprevista que provoca que toda la aplicación termine abruptamente). Cada ejecutor queda preparado para ejecutar *tareas* de Spark, que es la unidad mínima de ejecución de trabajos. Cada tarea requiere un *core* libre para ejecutarse, por lo que, si un *executor* tiene reservados cuatro *cores*, podrá ejecutar cuatro tareas en paralelo. Detallaremos esto más adelante. **Cada uno de los nodos del clúster en los que se crean ejecutores se denomina *worker*.**

### 3.5. Resilient distributed datasets (RDD)

Los RDD constituyen la **abstracción fundamental de Spark**.

Un RDD es una colección no ordenada (*bag*) de objetos, distribuida en  
la memoria RAM de los nodos del clúster.

La colección está dividida en particiones, cada una de las cuales está en la memoria RAM de un nodo distinto del clúster. Al desgranar el nombre, tenemos que:

- ▶ **Resilient** ('resistente', 'adaptable'): es posible reconstruir un RDD que estaba en memoria, a pesar de que una de las máquinas falle, gracias al DAG de ejecución (*directed acyclic graph*, el grafo de ejecución), que veremos más adelante.
- ▶ **Distributed** ('distribuido'): los objetos de la colección están divididos en particiones que están distribuidas en la memoria principal de los nodos del clúster. La colección no está ordenada, por lo que no se puede acceder mediante una posición a objetos individuales.
- ▶ **Dataset**: la colección representa un conjunto de datos que estamos procesando de forma paralela y distribuida, para transformarlos, calcular agregaciones, etc.

La figura 3 representa tres RDD diferentes, distribuidos en la memoria RAM de un clúster de cuatro nodos. No todos los RDD presentan el mismo número de particiones. En la figura, uno de los RDD tiene solo dos, otro tiene tres y otro, cuatro. La idea es similar al almacenamiento de los ficheros en HDFS, donde están distribuidos entre los discos duros de los nodos, con la diferencia de que, en este caso, los RDD están distribuidos en la memoria RAM de los nodos y, además, **no hay replicación de cada partición**.

Si un nodo falla, es posible reconstruir las particiones que estuvieran en ese

momento en su memoria principal gracias al DAG, que mantiene la traza de cómo se construyeron. El DAG es otro mecanismo que proporciona robustez sin necesidad de replicar las particiones de un RDD. Es preciso indicar también que, a pesar de que la figura 3 muestra una sola partición de cada RDD en cada nodo, lo habitual es que, en la memoria de un mismo nodo, haya numerosas particiones de un mismo RDD (de hecho, es normal que los RDD que se van calculando tengan decenas o cientos de particiones).

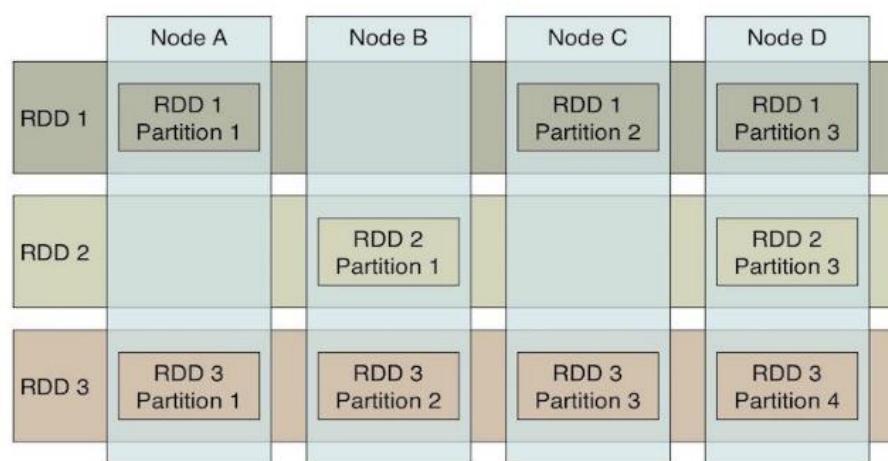


Figura 3. Representación de tres RDD en un clúster de Spark con cuatro *workers*.

En relación con los RDD, cabe destacar dos conceptos:

- ▶ **Inmutabilidad:** el contenido de un RDD no puede modificarse una vez creado. Lo que hacemos es aplicar transformaciones a los RDD para obtener otros nuevos, pero los datos del RDD existente no se alteran. La idea de la ejecución es que, cuando aplicamos una transformación, se ejecuta en paralelo sobre todas las particiones del RDD, de manera transparente al programador, para dar lugar a un nuevo RDD, cuyas particiones son el resultado de aplicar la transformación sobre cada una de las particiones del original.

**Ejemplo:** dado un RDD de números reales, para multiplicar cada elemento por dos, aplicamos una transformación que actúa en cada elemento y lo

multiplica por dos. Spark lleva nuestro código de la transformación (lo serializa y lo envía por la red) a cada uno de los nodos del clúster donde haya particiones de ese RDD y lo ejecuta en ellos para que actúe en cada elemento de esa partición. Todo de manera transparente al programador.

Una vez más, **los datos son el centro, lo más importante; no se mueven salvo que sea imprescindible**. Ciertos tipos de transformaciones no requieren movimiento de datos, pero otros sí, como veremos después.

- ▶ **Partición:** subconjunto de los objetos de un RDD que están presentes en un mismo nodo. **Es la unidad de datos mínima sobre la que se ejecuta una tarea de transformación de manera independiente al resto de particiones.** Idealmente, tendría que haber, al menos, tantas particiones como *cores* físicos (procesadores) disponibles en nuestro clúster. De esta manera, nos aseguramos de que todos los *cores* estarán ocupados, incluso en el caso de que nada más que nuestra aplicación estuviese empleando ese clúster. Lo habitual es que haya muchas particiones de un mismo RDD en cada nodo, en un número muy superior al de procesadores existentes en el ellos. Se recomienda que cada RDD esté dividido en un número de particiones que sea entre el doble y el triple que el número de procesadores del clúster.

Originalmente, los programadores de Spark trabajaban con RDD. Sin embargo, en Spark 1.6, se introdujeron los DataFrames, que definiremos más adelante. Desde Spark 2.0, los propios creadores **recomiendan encarecidamente no utilizar los RDD, sino siempre DataFrames y su API correspondiente**. Aparte de la facilidad de uso, el motivo fundamental es que los DataFrames están sujetos a importantes optimizaciones automáticas de código por parte del analizador de Spark, llamado Catalyst. Los RDD no están sujetos a estas optimizaciones y la ejecución es

sensiblemente más lenta. Todos los ejemplos los presentaremos con PySpark.

### 3.6. Transformaciones y acciones

Podemos realizar dos tipos de operaciones cuando usamos la API de Spark:

- ▶ **Transformación:** operación que se ejecuta sobre un RDD y devuelve un nuevo RDD, en el que sus elementos se han modificado de algún modo. Son *lazy* (perezosas): no se ejecuta nada hasta que Spark encuentra una acción. Mientras tanto, Spark simplemente añade la transformación al grafo de ejecución (el DAG), que mantiene la trazabilidad y permite la *resiliency*.
- El DAG guarda toda la secuencia de transformaciones que se realizaron para obtener cada RDD concreto que se vaya creando en nuestro código.
- Si la transformación no implica *shuffle* (movimiento de datos entre nodos), se denomina *narrow* y cada partición da lugar a otra en el mismo nodo. Hay que recordar que una operación *shuffle* implica una escritura previa de los datos en el disco duro local del nodo emisor y después en el disco local del nodo receptor, de manera transparente al programador.
- ▶ **Acción:** recibe un RDD y calcula un resultado (generalmente, un tipo simple, enteros, *doubles*, etc.) y lo devuelve al *driver* (programa principal, que corre en una máquina). El tipo de dato devuelto al *driver* no es un RDD, sino un tipo nativo del lenguaje que estemos utilizando (Java/Scala/Python/R).
- IMPORTANTE: el resultado de la acción debe caber en la memoria de la máquina donde se está ejecutando el proceso *driver*.
- Una acción desencadena instantáneamente el cálculo de toda la secuencia de transformaciones intermedias y la materialización de los RDD involucrados.
- Una vez materializado un RDD, se aplica la transformación que toque, según indica el DAG, para generar el siguiente RDD y el anterior se libera (no permanece en la memoria RAM, salvo que se indique expresamente mediante el método `cache()`). Un

RDD cacheado permanece materializado en la RAM de los nodos y no es necesario recalcularlo después de que se haya materializado la primera vez.

Por defecto, el punto de partida del DAG son operaciones de lectura de datos, bien desde una fuente de datos como, por ejemplo, HDFS o el sistema Amazon S3, bien desde alguna base de datos (distribuida o no) o similar. Si ningún RDD intermedio ha sido cacheado, cualquier operación que haga referencia a un RDD exigirá reconstruir toda la secuencia de transformaciones previas a dicho RDD, empezando por la lectura de los datos, excepto que alguno de los RDD intermedios haya sido cacheado. Esto hace que la secuencia empiece en el RDD cacheado y no haya que remontarse al origen de datos.

La utilización del DAG sirve para poder reconstruir cualquier RDD de una secuencia de transformaciones, tanto si la necesidad de materializar el RDD se debe a que hacemos referencia a él varias veces a lo largo de nuestro código, como si obedece a que, durante el procesamiento, falla algún nodo y el contenido de su memoria RAM se pierde. Gracias al DAG, es posible reconstruir cualquier partición concreta de cualquier RDD y volver a lanzar la secuencia de transformaciones en otros nodos que sí estén activos.

Existen ciertas operaciones de la API de Spark que **no son transformaciones ni acciones**, como, por ejemplo, `cache()`, sino que sirven para configurar, habilitar o deshabilitar ciertos comportamientos, o para obtener características relativas a la distribución física de un RDD [consultar el número de particiones que tiene, consultar si está cacheado, consultar el esquema (nombres y tipos de las columnas) de un DataFrame, etc.].

## Transformaciones más habituales con RDD

Recordemos que, para todas las operaciones que reciben una función, Spark serializa el código de la función y la envía por red a los nodos, donde es ejecutada.

- ▶ map : recibe como parámetro una función, que se ejecuta sobre cada uno de los elementos del RDD para transformarlos, y devuelve un nuevo RDD con los elementos transformados.
- ▶ flatMap : similar a la anterior, pero, en este caso, la función devuelve un vector de valores para cada elemento. En lugar de generar un RDD de vectores, los aplana para tener un RDD del tipo interior.
- ▶ filter : recibe una función que se aplicará sobre cada elemento del RDD y que deberá devolver un valor booleano (*true*, solo si ese elemento debe ser incluido en el nuevo RDD). Devuelve otro RDD con los elementos que han devuelto *true*.
- ▶ sample : devuelve una muestra aleatoria del RDD del tamaño especificado como parámetro.
- ▶ union : devuelve un RDD con la unión de dos RDD pasados como parámetros.
- ▶ intersection : devuelve la intersección de los dos RDD, es decir, los elementos que están presentes en ambos.
- ▶ distinct : quita los elementos repetidos (retiene cada elemento una sola vez).

Transformaciones específicas para un PairRDD, es decir, RDD de pares (clave, valor):

- ▶ groupByKey : cuando los elementos del RDD son tuplas (grupos de varios elementos ordenados), agrupa los elementos por la clave y considera esta como el primer elemento de la tupla.
- ▶ reduceByKey : similar al anterior, pero se agregan los elementos para cada clave empleando la función especificada como parámetro. Esta debe recibir dos valores y devolver uno, y cumplir las propiedades conmutativa y asociativa.
- ▶ sortByKey : ordena los elementos del RDD por clave.

- ▶ `join` : combina dos RDD de tal modo que se junten los elementos que tienen la misma clave.

## Acciones más habituales en RDD

Por definición, todas las acciones llevan resultados al *driver*, por lo que estos tienen que caber en la memoria del proceso *driver*.

- ▶ `reduce` : ejecuta una agregación de los datos empleando la función especificada como parámetro. Esta agregación se calcula sobre todos los datos, independientemente de que haya o no claves.
- ▶ `collect` : devuelve todos los elementos contenidos en el RDD como una colección del lenguaje (listas en Python y R, *arrays* en Java y Scala). Puede causar una **excepción** por memoria si la lista no cabe en la memoria RAM de la máquina donde está corriendo el *driver*. Se debe usar solo en casos muy controlados.
- ▶ `count` : devuelve el número de elementos contenidos en el RDD.
- ▶ `take` : devuelve los n primeros elementos contenidos en el RDD. En general, no hay garantías de ordenación en un RDD, salvo que se hayan empleado transformaciones como `sortByKey`.
- ▶ `first` : devuelve el primer elemento del RDD. Es equivalente a `take` cuando n=1.
- ▶ `takeSample` : devuelve n elementos aleatorios del RDD.
- ▶ `takeOrdered` : devuelve los n primeros elementos del RDD tras haber realizado una ordenación de todos los elementos contenidos en el mismo.
- ▶ `countByKey` : cuenta el número de elementos en el RDD para cada clave diferente.
- ▶ `saveAsTextFile` : guarda los contenidos del RDD en un fichero de texto.

## Ejemplo de código PySpark con RDD

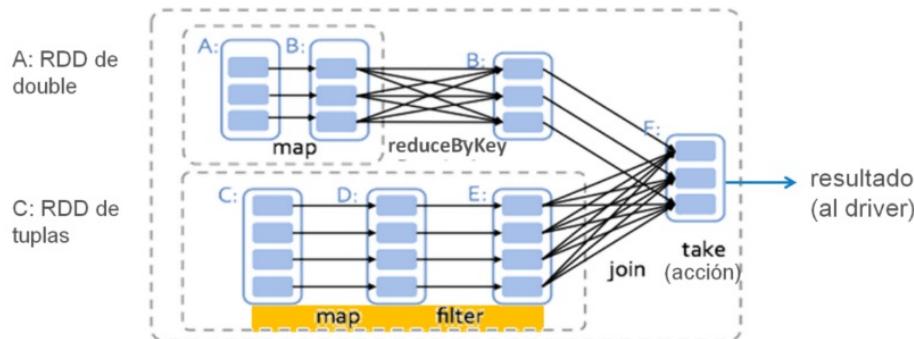


Figura 4. Ejemplo de transformaciones y de la acción take mediante el uso de RDD.

```

func_multiplicar = lambda x: (x, 3*x) # función que devuelve una tupla

A = sc.parallelize([5.0, 3.2, 1.1, -2.4, # distribuimos la lista como
8.9, 4.4, 3.7, 9.1], 3) # un RDD de 3 particiones

B = A.map(func_multiplicar)

B = B.reduceByKey(lambda v1, v2: v1+v2)

C = sc.parallelize([(5.0, 1.0), (1.1, -3)], 4) # lista a RDD de 4 part

D = C.map(lambda tuple: (tuple[0], 2*tuple[1]))

E = D.filter(lambda tuple: tuple[1] > 1)

F = E.join(B)

resultado = F.take(3)# ;ahora es cuando se desencadena el cálculo!

```

Como se observa, ciertas operaciones como join y reduceByKey asumen un RDD de (clave, valor), lo cual no es tan frecuente y todavía se asemeja a MapReduce en la manera de pensar (clave, valor). Para simplificar, existen los DataFrames, similares a tablas.

En el ejemplo anterior, hemos usado la variable `sc`, referida a `SparkContext`, el objeto que, en versiones anteriores, efectuaba la conexión con el gestor de clúster.

Actualmente, el objeto `sparkSession` envuelve (e incluye) a un `SparkContext`. Lo habitual es usar la `sparkSession` para leer datos de una fuente y crear desde ellos un `DataFrame`. No obstante, para poner crear un `RDD` (distribuido) a partir de una lista no distribuida del lenguaje, aún se necesita el objeto `sc`. En el código anterior, estamos creando, en primer lugar, un `RDD` llamado `A`, que la figura 5 representa con tres particiones, a partir de una lista de números reales. Por eso, el `RDD` resultante es un `RDD` de números reales.

Hemos definido una función `func_multiplicar`, que recibe un número y devuelve una tupla formada por el número y el resultado de multiplicarlo por 3. Dicha función la hemos aplicado a cada elemento de `A` mediante el método `map` de Spark. Este método es una **transformación** que aplica a cada elemento del `RDD` nuestra función y devuelve un nuevo `RDD` con el resultado. Para ello, **serializa el código de nuestra función** (en este caso, la función `func_multiplicar`) y lo envía por la red a los nodos, para que, en ellos (en aquellos nodos que contengan alguna partición del `RDD A`), se ejecute dicha función sobre los elementos de las particiones del `RDD` que estén presentes. Hemos llamado `B` al `RDD` resultante, que, en este caso, será un `RDD` de tuplas de dos elementos de tipo `double`, que es lo que devolvía `func_multiplicar`.

Nótese que, para llevar a cabo la transformación `map`, no es necesario movimiento de datos, ya que la función se aplica elemento a elemento sobre los que haya en el nodo y no necesita de otros elementos para calcular el resultado. Se dice que `map` es una transformación **narrow** (estrecha), a diferencia de otras transformaciones que forzosamente requieren movimiento de datos para poder calcular el resultado, llamadas transformaciones **broad**.

Un `RDD` de tuplas de dos elementos se considera, a ojos de Spark, un `PairRDD`, es decir, un `RDD` de (clave, valor). Los `RDD` de (clave, valor) admiten operaciones adicionales, además de las estándar de cualquier `RDD`. Una de ellas es `reduceByKey`, que agrupa elementos del `RDD` que tengan el mismo valor de clave y agrega entre sí sus valores, empleando la función que le pasemos como argumento. Esta función

requiere movimiento de datos entre nodos, ya que existirán tuplas que comparten la misma clave, pero estén en particiones distintas, que, además, posiblemente, también estarán en nodos diferentes. El resultado de esta transformación lo hemos vuelto a asignar a la variable B.

Por otro lado, hemos creado otro RDD en la variable C, que la figura 5 muestra con cuatro particiones, como el resultado de paralelizar una lista de tuplas de números reales. Por tanto, C ya es, desde el comienzo, un PairRDD. Le hemos aplicado, a continuación, una transformación *map*. Dado que C contiene tuplas, la función que aplicamos tiene que saber que el argumento que recibe es una tupla. En nuestro caso, a partir de cada tupla, se devuelve otra cuyo primer elemento es igual y cuyo segundo elemento es el resultado de multiplicar por 2 el segundo elemento original. La sintaxis de *lambda* de Python simplemente sirve para indicar que estamos creando una función anónima, es decir, una función convencional, pero que no necesitamos invocarla desde fuera. Solo la usamos para pasarla como argumento a un método (que espera recibir una función como argumento, tal como le ocurre a *map* de Spark); por eso, no necesitamos darle nombre, aunque podríamos haberlo hecho creando una función convencional con nombre, como se suele hacer con *def* en Python.

El RDD resultante de esta transformación *map* lo almacenamos en la variable D, que también es un PairRDD. Sobre él aplicamos una nueva transformación *filter*, que, al igual que *map*, actúa sobre cada elemento del RDD sin necesitar ningún otro proveniente de otra partición ni de otro nodo para calcular el resultado. De hecho, lo que *filter* lleva a cabo es un filtrado, de manera que el RDD resultante solo contendrá aquellos elementos del RDD original que cumplan cierta condición. La función pasada como argumento a *filter* debe ser capaz de recibir un elemento del RDD (en este caso, una tupla de dos elementos) y siempre ha de devolver un booleano, que indica si ese elemento tiene que formar parte del resultado (*true*) o no (*false*).

A continuación, hemos invocado una transformación *join*, que se aplica a un PairRDD y recibe como argumento otro PairRDD. El resultado es un nuevo RDD que contiene tuplas jerárquicas tales que la clave es común entre una tupla de uno de los RDD y una tupla del otro, y el valor está formado por una tupla con el valor que tenía esa clave en un RDD y el valor que tenía en el otro. El RDD resultante de la operación *join* aplicada a B y E lo almacenamos en la variable F.

Por último, hemos llamado al método *take* sobre el RDD F. Este método es una acción que lleva el resultado al *driver*. Esto implica que el resultado debe caber en él. En este caso, no supone un problema, ya que *take(n)* coge n elementos del RDD y los envía al *driver*, y solo hemos solicitado tres elementos. Además, y lo que es más importante, al ser una acción, desencadena la realización de todas las transformaciones anteriores que estaban pendientes. De hecho, la ejecución de cada una de las líneas de código anteriores a *take* simplemente provocaba que se añadiesen fases al grafo de ejecución (DAG) de Spark, pero no se materializaba ninguna. Se puede comprobar porque la ejecución en Python devuelve inmediatamente el control al intérprete de Python, sin emplear tiempo en llevarla a cabo. El resultado de *take* ya no es un RDD, sino una estructura de datos del lenguaje que se esté manejando. En este caso, es una lista de Python formada por tres elementos del RDD F, es decir, una lista de tres tuplas, que es lo que contiene F.

## 3.7. Jobs, stages y tasks

Un *job* (trabajo) de Spark es todo el procesamiento necesario para llevar a cabo una acción del usuario.

Por ejemplo: `df.count()`, `df.take(4)`, `df.show()`, `df.read(...)`, `df.write(...)` , etc. Cada *job* se divide en una serie de *stages* (etapas).

Un *stage* es todo el procesamiento que puede llevarse a cabo sin mover datos entre nodos.

Cada nodo hace exactamente un procesamiento idéntico, aplicado a diferentes particiones del mismo DataFrame que están procesando todos. Cuando en nuestro código invocamos una operación que implica movimiento de datos (*shuffle*), finaliza un *stage* y se crea otro nuevo. Ej: `df.join(...)`, `df.groupBy(...).agg(...)`.

Una *task* (tarea) es cada una de las transformaciones que forman una etapa. Es la unidad mínima de trabajo de Spark. Más formalmente:

Una tarea de Spark es el procesamiento aplicado por un *core* físico (CPU) a una partición de un RDD.

La siguiente figura representa la división en tareas, etapas y trabajos para el ejemplo de código contenido en el anexo.

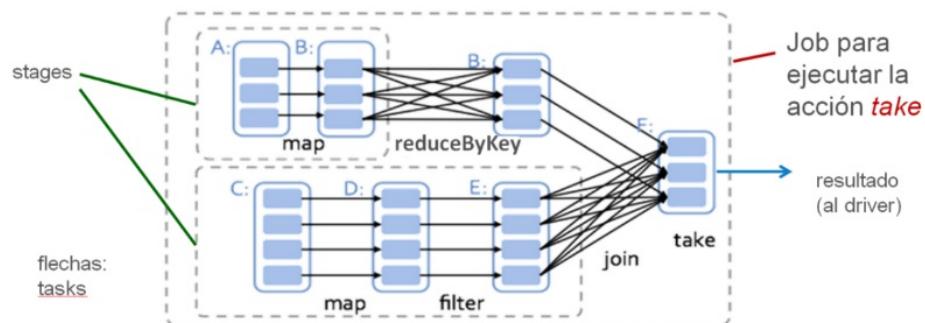


Figura 5. Representación de *jobs*, *stages* y *tasks* en Spark.

### 3.8. Ejemplo completo con RDD

Con el objetivo de comparar las características de la programación mediante RDD con respecto a la facilidad de uso que proporciona el componente que veremos en el siguiente tema (API estructurada y DataFrames), así como de destacar las diferencias con entornos como Apache Hive, que estudiaremos más adelante, vamos a ver, a continuación, un ejemplo completo resuelto en el que usaremos RDD. Posteriormente, resolveremos este mismo ejemplo en el siguiente tema, así como en el tema donde veamos Apache Hive, de forma que se puedan apreciar las similitudes, divergencias, ventajas e inconvenientes de cada uno de los entornos.

Para ello, vamos a usar los ficheros simplificados `flights.csv` y `airport-codes.csv`, almacenados ambos en HDFS bajo el directorio `/user/data`, y cuyas primeras líneas son las siguientes:

```
"year","month","day","origin","dest"  
2014,1,1,"PDX","ANC"  
2014,1,1,"SEA","CLT"  
...  
  
ident,name,iso_country,iata_code  
LELT,Lillo,ES,  
LEMD,Adolfo Suárez Madrid-Barajas Airport,ES,MAD  
...
```

El objetivo del ejemplo es contar cuántos vuelos reciben los diferentes destinos (`dest`) que aparecen en el fichero `flights.csv`. Además, como no conocemos bien los códigos de los aeropuertos de llegada, nos gustaría ver esta agregación del número de vuelos que recibe cada aeropuerto como nombre completo del aeropuerto y número de vuelos que recibe. Para esto, nos ayudaremos de la información que contiene el fichero `airport-codes.csv`, donde se relaciona el código del aeropuerto con su nombre.

```
from pyspark.sql import SparkSession

# En primer lugar, puesto que vamos a trabajar con RDD,
# necesitamos tener acceso al SparkContext.
# Para ello, primero obtenemos la SparkSession,
sparkSession = SparkSession.builder\
    .appName("demo_rdd")\
    .getOrCreate()
# y accedemos a uno de sus atributos, que es el SparkContext.
sparkContext = sparkSession.sparkContext

# Lo siguiente es leer el fichero flights.csv
# Cada línea se va a leer como un string completo:
flights_lines = sparkContext\
    .textFile("hdfs://<ip:port>/user/data/flights.csv")

# Dado que el fichero tiene cabecera (header), necesitamos
# ejecutar estas líneas para ignorarla (la librería de RDD
# no proporciona una opción directa para hacer esto):
header_line = flights_lines.first()
flights_lines = flights_lines\
    .filter(lambda lines: lines != header_line)

# Para comprobar qué contenido hemos leído, ejecutamos la acción take.
# ¡OJO! No hemos usado la acción collect, porque, en ese caso,
# se obtendrían TODOS los registros y podríamos provocar un error por
# quedarnos sin memoria (un fichero almacenado en HDFS probablemente
# esté almacenado ahí, de forma distribuida, porque no cabe
# en un sistema de ficheros local, mucho menos en la memoria de un
# único ordenador):
flights_lines.take(5)
```

Obtenemos los siguientes registros. Fijémonos en que cada uno de los cinco registros pedidos es una cadena de texto (*string*) completa, es decir, no están divididos por los diferentes campos que lo componen. La biblioteca de RDD no proporciona ningún método que permita interpretar la estructura del fichero, sino que sencillamente lee cada línea y la guarda como un registro del RDD. Además, cabe destacar que lo que observamos aquí no es el RDD. Cuando ejecutamos una acción como *collect* o *take* en PySpark, nos devuelve una lista de Python, tal y como se puede apreciar por la delimitación con corchetes de los cinco elementos (*strings*) de la lista.

```
[ '2014,1,1,PDX', "ANC" ,  
'2014,1,1,"SEA", "CLT" ,  
'2014,1,1,"PDX", "IAH" ,  
'2014,1,1,"PDX", "CLT" ,  
'2014,1,1,"SEA", "ANC" ]
```

```
# Por tanto, tenemos que dividir los diferentes  
# campos a mano, aplicando la función split de string.  
# Se separa cada línea utilizando el carácter '...',  
# para cada registro del RDD. Usaremos la transformación map  
# para aplicar una función a cada registro del RDD:  
flights_tuples_rdd = flights_lines.map(lambda line: line.split(','))

# Puesto que es una transformación, todavía no existe el RDD  
# flights_tuples_rdd. No se materializará hasta que se ejecute una  
# acción sobre el RDD:  
flights_tuples_rdd.take(5)
# En este punto, al ejecutar la acción take, es cuando se  
# materializa tanto flights_tuples_rdd como todos los RDD  
# previos necesarios para obtenerlo.
```

Obtenemos la siguiente salida. Ahora sí vemos cada línea separada en sus diferentes campos (año, mes, día, origen, destino) y delimitada en forma de lista de Python. Es decir, tenemos una lista con cinco registros de los RDD, cada uno de los cuales es una lista de *strings* correspondientes a los componentes de cada línea del fichero.

```
[[ '2014', '1', '1', '"PDX"', '"ANC"'],  
['2014', '1', '1', '"SEA"', '"CLT"'],  
['2014', '1', '1', '"PDX"', '"IAH"'],  
['2014', '1', '1', '"PDX"', '"CLT"'],  
['2014', '1', '1', '"SEA"', '"ANC"']]
```

Cabe recordar que, cada vez que ejecutamos una acción, como el último `take`, hay que volver a materializar todos los RDD necesarios desde la lectura de fichero, hasta obtener el RDD sobre el que se ejecuta la acción. Esto puede ser muy costoso en términos de procesamiento. Por tanto, si hay algún RDD sobre el que se necesite aplicar muchas acciones, puede ser interesante guardararlo en la memoria caché de

los *workers*. De esta forma, los RDD guardados en caché no hay que materializarlos en cada acción, sino que son accesibles directamente. Esto supone un compromiso entre la memoria y el procesamiento: los *workers* tienen una cantidad de memoria limitada, por lo que no se pueden cachear todos los RDD que queramos para evitar quedarnos sin memoria en los *workers*. Solo es conveniente cachear aquellos que necesiten materializarse repetidamente.

```
flights_tuples_rdd.cache()
```

```
# Como se puede observar en los resultados previos,
# todos los componentes que representan
# las diferentes líneas de cada sublista
# son de tipo string. Sin embargo, el año,
# el mes y el día estarían mejor representados por un tipo entero.
# Aunque esto no tiene impacto en este ejemplo, vamos a hacer
# un cambio de tipo para mostrar cómo se haría:
flights_tuples_format_rdd = flights_tuples_rdd\
    .map(lambda fields_list :
        (int(fields_list[0]),
         int(fields_list[1]),
         int(fields_list[2]),
         fields_list[3],
         fields_list[4]))  
  
print(flights_tuples_format_rdd.take(5))
```

En los siguientes resultados, podemos ver que ahora año, mes y día son enteros, mientras que origen y destino siguen siendo *strings*. Esta forma de proceder es poco intuitiva, ya que, si tuviéramos una lista con más campos, sería necesario saber qué posición ocupa cada uno para poderle aplicar correctamente el tipo deseado (no se puede hacer por nombre).

```
[(2014, 1, 1, '"PDX"', '"ANC"),
 (2014, 1, 1, '"SEA"', '"CLT"),
 (2014, 1, 1, '"PDX"', '"IAH"),
 (2014, 1, 1, '"PDX"', '"CLT"),
 (2014, 1, 1, '"SEA"', '"ANC")]
```

```
# Ahora queremos agrupar los destinos y contar cuántos
# vuelos recibe cada uno de ellos. Es decir, cuántas veces
# aparece ese destino en el fichero (y ahora RDD) de vuelos.

# Para hacer agrupaciones y agregaciones, necesitamos
# trabajar con PairRDD, es decir, RDD que sean tuplas de
# dos elementos: el primero de ellos será la clave por la que se
# quiera agrupar y el segundo, un valor asociado a la clave.
# En nuestro caso, la clave para cada registro va a ser el destino
# y el valor, 1, para contabilizar como 1 cada vez que
# aparezca dicho destino (y que, al sumarlos, tengamos
# el número de veces que aparece):
flights_dest_counter_rdd = flights_tuples_format_rdd\
    .map(lambda flight_tuple : (flight_tuple[4], 1))
flights_dest_counter_rdd.take(5)
```

Obtenemos un RDD con registros como los que se muestran a continuación: son registros de un *pairRDD*, es decir, un RDD que contiene tuplas de dos elementos, donde el primero será interpretado por Spark como la clave por la que hacer agrupaciones.

```
[('ANC', 1),
 ('CLT', 1),
 ('IAH', 1),
 ('CLT', 1),
 ('ANC', 1)]
```

```
# Con objeto de tener una idea de cómo funciona la agrupación
# de RDD, vamos a asociar todos los valores por clave (destino, en
# nuestro caso) y los vamos a mostrar en una lista. Para ello,
# agrupamos por clave y pedimos que se cree una lista con los valores # de
# cada clave:
flights_dest_grouped_rdd_toshow =
flights_dest_counter_rdd.groupByKey().mapValues(list)

# Mostramos solo un registro del resultado:
print(flights_dest_grouped_rdd_toshow.take(3))
```

Cada registro resultante de la agrupación será una tupla que contiene la clave por la

que se agrupó, así como una lista con todos los valores asociados a esa clave (en nuestro caso, tantos 1 como veces aparece el destino en el fichero):

```
[("ANC", [1, 1, 1, 1, 1, ...]),  
 ("CLT", [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...]),  
 ("ORD", [1, 1, ...])]
```

```
# Ahora sí: nuestro objetivo era contar cuántos vuelos  
# tenían como destino cada uno de los destinos  
# en el fichero flights.csv  
# Para ello, podemos utilizar el pairRDD previo, agrupar por destino y  
# sumar los valores de cada clave:  
flights_dest_total_rdd = flights_dest_counter_rdd\  
    .groupByKey().\  
    .map(lambda tuple_dest : (tuple_dest[0], sum(tuple_dest[1])))  
  
# O también podemos contar cuántos valores tiene asociados cada clave #  
(longitud, len) de la lista:  
flights_dest_total_rdd = flights_dest_counter_rdd\  
    .groupByKey().\  
    .mapValues(len)  
print(flights_dest_total_rdd.take(5))
```

Como podemos ver, el resultado en ambos casos es el mismo:

```
[("ANC", 7149),  
 ("CLT", 1224),  
 ("ORD", 7021),  
 ("PHX", 8660),  
 ("SLC", 5976)]
```

```
[("ANC", 7149),  
 ("CLT", 1224),  
 ("ORD", 7021),  
 ("PHX", 8660),  
 ("SLC", 5976)]
```

```
# Para saber a qué aeropuerto corresponde cada destino,  
# vamos a usar el fichero airport-codes.csv.  
# Procedemos igual que con el fichero flights.csv:  
airports_lines = sparkContext\  
    .textFile("hdfs://<ip:port> /user/data/airport-
```

```
codes.csv")

header_line = airports_lines.first()
airports_lines = airports_lines\
    .filter(lambda lines: lines != header_line)
airports_tuples_rdd = airports_lines\
    .map(lambda line: line.split(','))
airports_tuples_format_rdd = airports_tuples_rdd\
    .map(lambda fields_list : (fields_list[3], fields_list[1]))
print(airports_tuples_format_rdd.take(2))
```

Vemos un par de aeropuertos entre los leídos. Cabe destacar que hemos creado otro pairRDD (obligatorio para el join que queremos hacer), para el que nos hemos quedado solo con dos de los campos de cada línea, y que les hemos dado la vuelta para que el código del aeropuerto sea el primero de la tupla y, por tanto, la clave. Fijémonos en que el formato del código no es igual que en el fichero flights.csv, ya que, en este último, el código está entrecomillado (la lectura de RDD no sabe interpretar que eso son *strings*, por lo que no es necesario el entrecomillado).

```
[('ORD', ' Chicago O'Hare International Airport'),
 ('SFO', ' San Francisco International Airport')]
```

```
# Vamos a quitar las comillas de los destinos en el RDD de vuelos
# para poder hacer el join (si no, los códigos de destino de
# ambos RDD no serán iguales)
# con tuple_flight[0][1:4]. Nos quedamos con
# los caracteres 1-3, ambos inclusive.
# En definitiva, quitamos las comillas (posiciones 0 y 4):
flights_dest_total_rdd = flights_dest_total_rdd\
    .map(lambda tuple_flight : (tuple_flight[0][1:4],
                                tuple_flight[1]))

# Y, ahora sí, podemos hacer un join usando la clave de ambos
# pairRDD:
dest_count_airports_rdd = flights_dest_total_rdd\
    .join(airports_tuples_format_rdd)
print(dest_count_airports_rdd.take(5))
```

El resultado del join con RDD son pairRDD, donde la clave es el código del

aeropuerto y el valor, otra tupla que contiene el valor del primer pairRDD (el número de vuelos) y el del segundo (el nombre del aeropuerto).

```
[('ORD', (7021, "Chicago O'Hare International Airport")),
 ('SFO', (12809, 'San Francisco International Airport')),
 ('IAD', (1344, 'Washington Dulles International Airport')),
 ('KOA', (734, 'Ellison Onizuka Kona International At Keahole Airport')),
 ('JNU', (1282, 'Juneau International Airport'))]
```

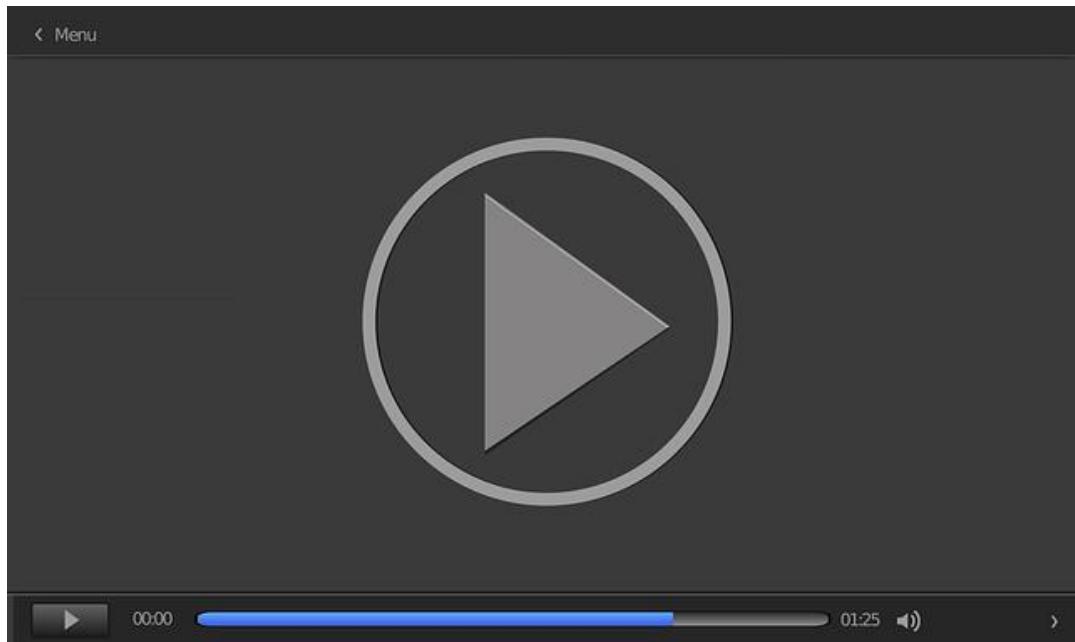
```
# Si lo queremos mostrar como nombre y número de vuelos,
# solo resta elegir esos campos, pero hay que tener en cuenta el
# anidamiento de tuplas (de nuevo, poco intuitivo):
dest_count_airports_rdd = dest_count_airports_rdd\
    .map(lambda tuple_result : (tuple_result[1][1],
                                tuple_result[1][0]))
print(dest_count_airports_rdd.take(5))
```

Por fin, obtenemos el resultado deseado:

```
[("Chicago O'Hare International Airport", 7021),
 ('San Francisco International Airport', 12809),
 ('Washington Dulles International Airport', 1344),
 ('Ellison Onizuka Kona International At Keahole Airport', 734),
 ('Juneau International Airport', 1282)]
```

Este ejemplo bien sirve para demostrar que el manejo de RDD, aunque proporciona gran potencial en el manejo de grandes cantidades de datos, no es demasiado intuitivo, ya que es necesario tener siempre muy presente la estructura interna del RDD y el tipo de datos que contiene, además de la siguiente restricción: necesita pairRDD para poder hacer operaciones de agregación y operaciones entre RDD. En el siguiente capítulo, veremos qué soluciones proporciona Spark para facilitar este tipo de tareas y hacerlas mucho más intuitivas con una API estructurada y Spark SQL.

Para finalizar, puedes ver este vídeo, en el profundizaremos en la utilización de los RDD de Spark desde la herramienta JupyterLab de Dataproc.



---

Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=bf539479-7a1e-4b21-9fea-ac70011af767>

---

Vídeo. Spark RDD.

## 3.9. Referencias bibliográficas

Chambers, B. y Zaharia, M. (2018). *Spark: the definitive guide*. O'Reilly.

## Documentación oficial de Apache Spark

Apache Spark. Página web oficial: <https://spark.apache.org/docs/latest/>

Documentación *online*, detallada y de muy buena calidad sobre el sistema de computación Apache Spark.

## Hadoop: the end of an era

Grishchenko, A. (2019, 23 de marzo). Hadoop: the end of an era [entrada de blog].

*Distributed Systems Architecture.* <https://0x0fff.com/hadoop-the-end-of-an-era/>

Hadoop es uno de los *frameworks* más importantes para el *big data*, cuyo propósito es almacenar grandes cantidades de datos y permitir consultas sobre estos, que se ofrecerán con un bajo tiempo de respuesta. Nació como iniciativa de Apache para dar soporte al paradigma de programación MapReduce, que fue inicialmente publicado por Google. En esta entrada de blog, se propone una interesante reflexión sobre su uso en la actualidad.

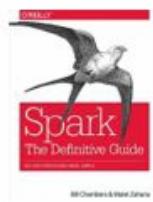
## DATA + AI Summit

DATA + AI Summit. Página web oficial: <https://databricks.com/sparkaisummit>

Sitio web del congreso más famoso de Spark a nivel mundial, del que tienen lugar también versiones más reducidas en cada continente. Está organizado por Databricks, empresa que soporta el desarrollo de Spark.

## Spark: the definitive guide

Chambers, B. y Zaharia, M. (2018). *Spark: the definitive guide*. O'Reilly.



Guía detallada de Spark, en su versión más actualizada. Contiene numerosos ejemplos y muestra exhaustivamente todas las capacidades de Spark.

## High performance Spark

Karau, H. y Warren, R. (2017). *High performance Spark*. O'Reilly.



Manual avanzado sobre cómo escribir código optimizado en Spark.

1. ¿Cuál es la principal fortaleza de Spark?

  - A. Opera en memoria principal, lo que hace que los cálculos sean mucho más rápidos.
  - B. Nunca da lugar a movimiento de datos entre máquinas (*shuffle*).
  - C. Las respuestas A y B son correctas.
  - D. Las respuestas A y B son incorrectas.
2. ¿Qué tipo de procesos se benefician especialmente de Spark?

  - A. Los procesos en modo *batch*, como, por ejemplo, una consulta SQL.
  - B. Los proceso aplicados a datos no demasiado grandes.
  - C. Los algoritmos de aprendizaje automático que dan varias pasadas sobre los mismos datos.
  - D. Las respuestas A, B y C son correctas.
3. ¿Cuál es la estructura de datos fundamental en Spark?

  - A. RDD.
  - B. DataFrame.
  - C. SparkSession.
  - D. SparkContext
4. En una operación de Spark en la que sea necesario movimiento de datos...

  - A. Siempre debemos escribirlos primero en el disco local del nodo emisor.
  - B. No hay acceso al disco local, puesto que Spark opera siempre en memoria.
  - C. Spark nunca provoca movimiento de datos, a diferencia de MapReduce.
  - D. Las respuestas A, B y C son incorrectas.

5. Elige la respuesta correcta: Cuando se ejecuta una transformación en Spark sobre un RDD...
  - A. Se crea inmediatamente un RDD con el resultado de la transformación.
  - B. Se modifica inmediatamente el RDD con el resultado de la transformación.
  - C. Se añade la transformación al DAG, que creará un RDD con el resultado de la transformación cuando se materialice el RDD resultante.
  - D. Se añade la transformación al DAG, que modificará el RDD original con el resultado de la transformación cuando se materialice el RDD resultante.
6. Elige la respuesta correcta: La acción *collect* de Spark...
  - A. No existe como acción; es una transformación.
  - B. Aplica una función a cada fila del RDD de entrada y devuelve otro RDD.
  - C. Lleva todo el contenido del RDD al *driver* y podría provocar una excepción.
  - D. Lleva algunos registros del RDD al *driver*.
7. Elige la respuesta incorrecta: Un PairRDD...
  - A. Es un tipo de RDD que permite realizar tareas de agregación y *joins*.
  - B. Es un tipo de RDD que contiene una tupla con un número variable de componentes.
  - C. Es un tipo de RDD cuyo primer componente se considera la clave y el segundo, el valor.
  - D. Se define como cualquier otro RDD, pero con un formato concreto.

**8.** ¿Qué es un *executor* de Spark?

- A. Cada uno de los nodos del clúster de Spark.
- B. Un proceso creado en los nodos del clúster, preparado para recibir trabajos de Spark.
- C. Un nodo concreto del clúster que orquesta los trabajos ejecutados en él.
- D. Ninguna de las definiciones anteriores es correcta.

**9.** La acción *map* de Spark...

- A. No existe como acción; es una transformación.
- B. Aplica una función a cada fila del RDD de entrada y devuelve otro RDD.
- C. Lleva todo el contenido del RDD al *driver* y podría provocar una excepción.
- D. Lleva ciertos registros del RDD al *driver*.

**10.** Cuando Spark ejecuta una acción...

- A. Se materializan en la memoria RAM de los *workers* todos los RDD intermedios necesarios para calcular el resultado de la acción y después se liberan todos.
- B. Se añade la acción al DAG y no hace nada en ese momento.
- C. Se materializan los RDD intermedios necesarios que no estuviesen ya materializados, se calcula el resultado de la acción y se liberan los no cacheados.
- D. Ninguna de las respuestas anteriores es correcta.

Ingeniería para el Procesado Masivo de Datos

---

## Tema 4. Spark II

# Índice

[Esquema](#)

[Ideas clave](#)

[4.1. Introducción y objetivos](#)

[4.2. DataFrames en Spark](#)

[4.3. API estructurada de Spark: lectura y escritura de DataFrames](#)

[4.4. API estructurada de Spark: manipulación de DataFrames](#)

[4.5. Ejemplo de uso de API estructurada](#)

[4.6. Spark SQL](#)

[4.7. Ejemplo de Spark SQL](#)

[A fondo](#)

[Documentación oficial de Apache Spark](#)

[Hadoop: the end of an era](#)

[DATA + AI Summit](#)

[Spark: the definitive guide](#)

[High performance Spark](#)

[Test](#)

# Esquema

APACHE SPARK – API ESTRUCTURADAS	
	SPARK DATAFRAMES
CONCEPTO	LECTURA/ESCRITURA
DataFrame Spark: tabla [filas y columnas] de datos distribuidos en memoria ~ tabla de BDD relacional	<p>Lectura DataFrames:</p> <ul style="list-style-type: none"> <li>- Fuentes: HDF5, S3, JDBC/ODBC, Kafka, NoSQL</li> <li>- Sin inferir esquema</li> <li>- Con inferencia esquema</li> <li>- Especificando esquema</li> </ul> <p>Internamente es:</p> <ul style="list-style-type: none"> <li>- Una envoltura de un RDD de objetos de tipo Row</li> </ul> <p>Se aconseja uso de DataFrames en lugar de RDD porque...</p> <ul style="list-style-type: none"> <li>- Uso más intuitivo que RDD → reduce errores potenciales</li> <li>- Motor Catalyst: optimiza operaciones con DataFrames</li> </ul>
	<p>ESCRITURA</p> <p>Básicas:</p> <ul style="list-style-type: none"> <li>- read</li> <li>- printSchema</li> <li>- col</li> <li>- select</li> <li>- alias</li> <li>- withColumn</li> <li>- drop</li> <li>- when</li> <li>- lit</li> </ul> <p>Matemáticas y estadísticas:</p> <ul style="list-style-type: none"> <li>- describe</li> <li>- randn/sim/cos/sqrt...</li> </ul> <p>Entidades:</p> <ul style="list-style-type: none"> <li>- unionAll</li> <li>- except</li> <li>- where</li> </ul> <p>Agregaciones:</p> <ul style="list-style-type: none"> <li>- groupBy</li> <li>- agg</li> <li>- count, max, min...</li> <li>- countDistinct</li> </ul>
	<p>TRANSFORMACIONES</p> <p>Permite realizar consultas con formato SQL a DataFrames de Spark</p> <p>Para ello, es necesario:</p> <ol style="list-style-type: none"> <li>1. Registrar el DataFrame como:           <ul style="list-style-type: none"> <li>- tabla o vista</li> </ul> </li> <li>2. Realizar consulta SQL mediante método sql(<code>sql("consulta SQL")</code>)</li> </ol>

## 4.1. Introducción y objetivos

En el tema anterior, estudiamos los conceptos básicos de una nueva tecnología de procesado de grandes cantidades de datos en clúster de equipos, denominada Apache Spark. Este *framework* de programación distribuida nacía con la intención de mejorar en términos de eficiencia, rapidez y programación intuitiva el paradigma de MapReduce.

Tal y como se comentaba en el capítulo previo, Spark es un conjunto de componentes, de los que vimos cómo funciona Spark Core y su estructura de datos principal, los RDD. Otro de los componentes, probablemente el más usado hoy en día, es la API estructurada y su estructura de datos clave, los DataFrames. En este capítulo, estudiaremos en detalle esta API estructurada y la creación y manipulación de DataFrames, así como las ventajas que tiene sobre los RDD. Por último, veremos una API relacionada, Spark SQL, que nos facilita la manipulación de estos DataFrames para aquellos desarrolladores con experiencia previa en SQL.

Teniendo en cuenta todo esto, los objetivos que persigue este tema son:

- ▶ Conocer la API estructurada de Spark y su principal estructura de datos, el DataFrame.
- ▶ Identificar las ventajas de usar DataFrames en lugar de RDD.
- ▶ Conocer Spark SQL, así como sus similitudes y diferencias con la API estructurada.
- ▶ Practicar con algunas funciones típicas de procesamiento de DataFrames, tanto con la API estructurada como con Spark SQL.

## 4.2. DataFrames en Spark

Manejar RDD resulta tedioso cuando los tipos de datos que contienen empiezan a complicarse, ya que en todo momento necesitamos saber exactamente la estructura del dato, incluso cuando son tuplas jerárquicas (tuplas en las que algún campo es, a su vez, una tupla o lista). Sería más conveniente poder manejar los RDD como si fuesen tablas de datos, estructuradas en filas y columnas, lo cual aportaría mayor nivel de abstracción y más facilidad de uso. Esto es lo que nos proporcionan la API estructurada y los DataFrames de Spark:

Un DataFrame de Spark es una tabla de datos distribuida en la RAM de los nodos, formada por filas y columnas con nombre y tipo (incluidos tipos complejos), similar a una tabla en una base de datos relacional

Internamente, un DataFrame no es más que un **RDD de objetos de tipo Row de Spark**, cada uno de los cuales representa una fila de una tabla como un vector cuyos componentes (lo que serían las columnas de una tabla) tienen nombre y tipo predefinido. El esquema (schema) de un DataFrame define el nombre y el tipo de dato de cada una de estas columnas. **Cada DataFrame envuelve (tiene dentro) un RDD**, al que se puede acceder como el atributo `rdd`. Ej.: `variableDF.rdd`. Por eso, los conceptos de transformación y acción se aplican también a DataFrames.

Hay que recordar que los DataFrame de Spark están distribuidos en la memoria RAM de los nodos *worker*, aunque puedan parecerse a una tabla de una base de datos. Por otro lado, el nombre DataFrame es el mismo que el definido en otras librerías de lenguajes como Python (DataFrames del paquete Pandas) o R (data.frame). Si bien el concepto es el mismo (tabla de datos cuyas columnas tienen nombre y tipo), la implementación y manejo no tienen nada que ver, más allá de que los autores eligieron el mismo nombre. Los DataFrames de Spark son un tipo de dato definido

por Spark, están distribuidos físicamente y se manejan mediante la API de Spark.

Existe, en la API de Spark, un método que permite traerse todo el contenido a una sola máquina (el *driver*) y que devuelve un DataFrame de Pandas (no distribuido). Es el método `toPandas`, pero debe usarse con cuidado, ya que, de nuevo, requiere que todo el contenido distribuido en los nodos quepa en la memoria RAM del nodo donde se ejecuta el proceso *driver*. De lo contrario, provocará una excepción `OutOfMemory`.

En las siguientes secciones, vamos a ver las diferentes operaciones que se pueden hacer con DataFrames y la API estructurada. Empezaremos por la lectura y escritura de DataFrames y seguiremos por las diferentes transformaciones que se pueden aplicar a los DataFrames leídos.

## 4.3. API estructurada de Spark: lectura y escritura de DataFrames

Como cabe esperar, lo primero que se necesita para tener un DataFrame son los datos con los que se quiere trabajar y que, por tanto, serán cargados en un DataFrame para su futura manipulación. Spark puede leer información de numerosas fuentes de datos. Para que Spark pueda conectarse a una fuente de datos, debe existir un conector específico que indique cómo obtener datos de esa fuente y convertirlos en un DataFrame. Entre las fuentes de datos más habituales que disponen de dicho conector, podemos encontrar:

- ▶ **HDFS.** Spark puede leer de HDFS diversos formatos de archivo: CSV, JSON, Parquet, ORC y texto plano. No obstante, la comunidad de desarrolladores ha proporcionado mecanismos para leer otros tipos de ficheros, como los XML, entre otros. La terminación indicada en el nombre de archivo no informa a Spark de nada, solo puede servir como pista al usuario que vaya a leer el fichero.
- ▶ **Amazon S3.** Almacén de objetos distribuido creado por Amazon, de donde Spark también puede leer cualquiera de los formatos de fichero nombrados anteriormente.
- ▶ Bases de datos relacionales, mediante conexiones **JDBC u ODBC**. Spark es capaz de leer en paralelo a través de varios *workers* conectados simultáneamente a una base de datos relacional, cada uno de los cuales lee porciones diferentes de una misma tabla. Cada porción va a una partición del resultado. Spark puede enviar una consulta a la base de datos y leer el resultado como DataFrame.
- ▶ Conectores para **bases de datos no relacionales**. Los conectores son específicos, desarrollados por la comunidad o por el fabricante (por ejemplo: Cassandra, MongoDB, HBase ElasticSearch...).
- ▶ Cola distribuida **Kafka** (que veremos en un tema posterior). Para datos que se van

leyendo de un *buffer*.

- ▶ También datos que llegan en **streaming a HDFS** (ficheros que se van creando nuevos en tiempo real).

Aunque hemos hablado de fuentes de datos de lectura, en el caso de querer escribir resultados del DataFrame en almacenamiento persistente, ya sea fichero, base de datos o servicio de mensajería, Spark proporciona los mismos mecanismos de los que hemos hablado para la lectura. A continuación, vamos a ver en detalle cómo realizar las lecturas y escrituras de los datos.

## Lectura de DataFrames

En general, para leer datos desde Spark, se usa un atributo de la `SparkSession`, `spark.read`, y se especifican diferentes opciones, dependiendo del tipo de fichero por leer:

- ▶ **Formato.** Tal y como se mencionó anteriormente, el formato de fichero puede ser CSV, JSON, Parquet, Avro, ORC, JDBC/ODBC o texto plano, entre otros muchos.
- ▶ **Esquema.** Hay cuatro opciones referentes al esquema:
  1. Algunos tipos de fichero almacenan el esquema junto a los datos (por ejemplo, Parquet, ORC o Avro) y, por tanto, no es necesario indicar ningún esquema adicional.
  2. Para los tipos de fichero que no almacenan el esquema, es posible solicitar a Spark que trate de inferirlo con la opción `inferSchema` activada (`true`). Hay que tener en cuenta que esta opción conllevará más tiempo de lectura, dado que Spark necesita leer una serie de líneas del fichero y analizarlas para tratar de adivinar el esquema.
  3. Podemos pedir a Spark que no trate de inferir el esquema. En este caso, todos los datos se leerán como si fueran texto (`string`).
  4. Cabe la opción de indicar de forma explícita el esquema de los datos esperado, para

evitar un incremento del tiempo de lectura y posibles incorrecciones en la inferencia del esquema por parte de Spark. Veremos estas opciones en detalle, con algún ejemplo, más adelante.

- ▶ **Modo de lectura.** Puede ser; permissive (por defecto), que traduce como null aquellos registros que considere corruptos de cada fila; dropMalformed , que descarta las filas que contienen alguno de sus registros con un formato incorrecto, y failFast , que lanza un error en cuanto encuentra un registro con un formato incorrecto.
- ▶ Existen otra serie de opciones que veremos más adelante, ya que dependen del tipo específico de fichero a leer.

La única información obligatoria es la ruta del fichero que va a ser leído; el resto de opciones son opcionales. Por tanto, la estructura genérica de lectura sería la siguiente:

```
myDF = spark.read.format(<formato>)
    .load("/path/to/hdfs/file") # spark es el objeto SparkSession
# <formato> puede ser "parquet" | "json" | "csv" | "orc" | "avro"
```

En cuanto al esquema, recordemos que es el que describe el nombre y tipo de cada uno de los registros (columnas) de las filas del DataFrame. Como comentábamos, algunos tipos de fichero, como Avro, Parquet u ORC, contienen información respecto a su esquema y, por tanto, no es necesario especificarlo durante su lectura. Sin embargo, con otros formatos, como CSV o JSON, donde el esquema del fichero no está almacenado en este, podemos dejar que Spark infiera el esquema con la opción inferSchema configurada como *true*, indicar que no infiera nada ( *inferSchema* a *false* ) y lea todo como *string*, o bien especificar explícitamente cuál va a ser el esquema concreto esperado.

Cabe recordar una vez más que la opción *inferSchema* indica a Spark que trate de

adivinar el tipo de cada columna, en cuyo caso Spark tratará de inferir lo mejor posible esta información. No obstante, puede darse el caso de que, en una columna que debería ser de tipo entero, falte alguna información y de que Spark, ante la duda, infiera que dicha columna es de tipo texto (*string*). Para evitar este tipo de incorrecciones, es mejor especificar explícitamente el esquema si se conoce de antemano. Un esquema en Spark no es más que un objeto `StructType`, compuesto por un conjunto de `StructFields`. Cada `StructField` representa un registro (columna) de una fila; por tanto, se compone de un nombre (`name`), un tipo (`type`), un booleano que indica si la columna puede contener datos `null` (es decir, datos inexistentes), así como otra información opcional. Por ejemplo, podemos definir en `pyspark` el esquema de un fichero JSON donde cada fila contiene tres campos (columnas) de la siguiente forma:

```
from pyspark.sql.types import StructField, StructType, StringType, LongType

fileSchema = StructType([
    StructField("dest_country_name", StringType(), True),
    StructField("origin_country_name", StringType(), True),
    StructField("count", LongType(), False)
])
```

Una vez que se tiene el esquema definido, solo resta leer los datos utilizando dicho esquema:

```
myDF = spark.read.format("json")
        .schema(fileSchema)
        .load("/path/to/file.json") # spark es el objeto
SparkSession
```

Si no queremos que Spark infiera el esquema ni proporcionar uno nosotros, Spark leerá todas las columnas como *strings*:

```
myDF = spark.read.format("json")
    .options("inferSchema", "false")
    .load("/path/to/file.json") # spark es el objeto SparkSession
```

En caso de querer que Spark infiera el esquema en lugar de especificarlo, habría que usar la opción `inferSchema`, como comentábamos anteriormente:

```
myDF = spark.read.format("json")
    .options("inferSchema", "true")
    .load("/path/to/file.json") # spark es el objeto
SparkSession
```

Por último, cabe mencionar que, para algunos tipos de fichero, suele estar disponible un atajo como `spark.read.<format>("/path/to/file")`, donde `format` puede ser, por ejemplo, `cvs` o `avro`, aunque no todos los formatos lo tienen.

Vamos a ver algunas particularidades de ciertos formatos concretos.

## CSV

Los ficheros CSV son probablemente los más problemáticos, ya que la división de las filas en diferentes registros (columnas) depende de separadores que no siempre se respetan o que son confusos respecto al texto del fichero. Por ejemplo, si los registros están separados por comas, y uno de ellos es de tipo *string* y puede contener dentro comas, cabe la posibilidad de que se produzcan interpretaciones (*parse*) incorrectas. Además de las opciones que ya hemos visto, los ficheros CSV soportan más opciones. Entre las más usadas, destacan:

- ▶ Si el fichero incorpora cabecera, la opción `header` indica si cabe esperar que la primera línea del fichero se corresponda con los nombres de las columnas (`true`) o no (`false`, por defecto).
- ▶ El carácter separador. La opción `delimiter` permite indicar el carácter separador de

registros (columnas) en una fila. Es importante tener en cuenta que **Spark no soporta separadores de más de un carácter**.

Veamos un ejemplo de lectura de CSV con y sin inferencia de esquema, en el que usaremos, además, el atajo `.csv(<path>)` que se comentaba anteriormente.

```
# En df1 se van a leer todas las columnas como si fuesen strings:  
  
df1 = spark.read.option("inferSchema", "false")  
      .csv("/path/hdfs/file")#uso del atajo  
  
# Para evitar que todas las columnas se lean como strings,  
# vamos a indicar el esquema para que cada columna  
# se lea con su tipo correspondiente y se le asigne el nombre deseado:  
  
myschema = StructType([  
    StructField("columna1", DoubleType(), nullable =  
False),  
    StructField("columna2", DateType(), nullable =  
False)  
])  
  
# Pasamos el esquema para que se lean correctamente  
# (el true/false como valor de option se escribe en minúscula):  
  
df2 = spark.read.option("header", "true")\  
      .option("delimiter","|")\  
      .schema(myschema)\  
      .csv("/path/hdfs/file")#uso del atajo
```

## Parquet, ORC, Avro

Para estos tipos de ficheros, la lectura es más directa, ya que basta con ejecutar el siguiente código (no es necesario ni inferir esquema ni especificarlo):

```
myDf = spark.read.parquet("/path/to/file.parquet") # usa atajo
```

## Escritura de DataFrames

La operación de escritura es análoga a la de lectura. En este caso, se usa el atributo `write` de los DataFrames y se indican los siguientes aspectos:

- ▶ **Formato**, con el método `format(<formato>)`, que puede ser cualquiera de los que hemos visto para la operación de lectura.
- ▶ **Modo de escritura**. Puede ser `append`, `overwrite`, `errorIfExists`, `ignore` (si los datos o ficheros existen, no se hace nada).
- ▶ Otras opciones específicas de cada formato.

Así, por ejemplo, si queremos guardar un DataFrame, `df`, en formato CSV, utilizaremos el tabulador como separador y, en modo sobreescritura, en la cabecera en el fichero, escribiremos algo como lo siguiente:

```
df.write.format("csv")  
    .mode("overwrite")  
    .option("sep", "\t")  
    .option("header", "true")  
    .save("path/to/hdfs/directory")
```

Mientras que, si queremos guardarlo en Parquet, introduciremos el siguiente código:

```
df.write.format("parquet") # equivalente en orc, avro y json  
    .mode("overwrite")  
    .save("path/to/hdfs/directory")
```

Hay que tener en cuenta que la ruta especificada donde se guardará la información no es un fichero, sino un directorio (que creará Spark), dentro del cual se escribirá la información del DataFrame en diferentes partes, con la estructura `part-XXXXX-hash.csv` en el caso de CSV. También puede escribirse el resultado en otros destinos de datos muy diversos, siguiendo sintaxis específica. Para más detalles, se puede consultar la web de Spark.



## 4.4. API estructurada de Spark: manipulación de DataFrames

Una vez que sabemos qué es un DataFrame, cómo leer datos para crear DataFrames y cómo escribir la información que albergan en almacenamiento persistente, veamos qué tipo de operaciones podemos realizar sobre estas estructuras de datos distribuidas que nos proporciona Spark.

Recordemos que un DataFrame consistía en un conjunto de filas (en el fondo, envolvían RDD de objetos de tipo `Row`), cada una de las cuales estaba formada por una serie de registros (columnas). La mayoría de las operaciones que ofrece la API estructurada son operaciones sobre columnas (clase `Column`). Spark implementa muchas operaciones entre columnas de manera distribuida (operaciones aritméticas entre columnas, manipulación de columnas de tipo `string`, comparaciones...), que debemos usar siempre que sea posible. Todas ellas son sometidas a **optimizaciones por parte de Catalyst** para ejecutar más rápidamente.

---

Para más información sobre la API estructurada, accede a la documentación oficial: <http://spark.apache.org/docs/latest/api/python>

---

La utilización general de los métodos de la API sigue el siguiente formato: `objetoDataFrame.nombreDelMétodo(argumentos)`. Todas las manipulaciones devuelven como resultado un nuevo DataFrame, sin modificar el original (recordamos que los RDD son **inmutables** y, por extensión, los DataFrames también). Por eso, se suelen encadenar transformaciones: `df.método1(args1).método2(args2)`

### Transformaciones más frecuentes con DataFrames

Supongamos que hemos creado una variable llamada `df` con esta línea de código:

```
df = spark.read.parquet("/ruta/hdfs/datos.parquet")
```

Antes de describir las transformaciones más frecuentes con DataFrames, cabe mencionar que todas ellas están contenidas en la librería de pyspark `pyspark.sql.functions`. Por tanto, es necesario importar esta librería antes de usar cualquiera de estas funciones. Generalmente, se suele importar asignándole el alias `F`, de forma que, posteriormente, nos podremos referir a las funciones como `F.nombreFuncion(argumentos)`.

Una vez que sabemos cómo importar las funciones, veamos cuáles son:

- ▶ `printSchema` imprime el esquema del DataFrame. Esta función es muy útil cuando queremos comprobar que el DataFrame se ha leído de fichero con el tipo de datos esperado.

```
df.printSchema()
```

- ▶ `col("nombreCol")` sirve para seleccionar una columna y devuelve un objeto `Column` sobre el que podemos realizar diferentes transformaciones. Es importante tener en cuenta que no se puede mezclar código SQL (que devuelve DataFrames) con objetos de tipo `Column`.
- ▶ `select` permite seleccionar columnas de diferentes formas:

```
import pyspark.sql.functions as F

#ambas formas de usar select devuelven el mismo resultado:

df.select("nombreColumna").show(5)

df.select(F.col("nombreColumna")).show(5)

# Crear columna con nombre diff y seleccionar esa columna,
# que es el resultado de restar el literal 18 a la columna edad.
```

```
# Mostrar 5 registros de la columna resultante seleccionada:  
  
df.select((F.col("edad")-F.lit(18)).alias("diff")).show(5)
```

- ▶ alias le asigna un nombre a la columna sobre la que se aplica

```
import pyspark.sql.functions as F  
  
df.select(F.col("nombreColumna").alias("nombreNuevo")).show(5)
```

- ▶ withColumn devuelve un nuevo DF con todas las columnas del original más una nueva columna añadida al final, como resultado de una operación entre columnas existentes que devuelve como resultado un objeto Column

```
import pyspark.sql.functions as F  
  
# Crea una nueva columna cuyo nombre es nombreNuevaColumna  
# y que almacena la suma de los valores en las columnas c1 y c2.  
# Devuelve un objeto de tipo Column, del que mostramos 5 registros:  
  
df.withColumn("nombreNuevaColumna", F.col("c1")+F.col("c2")).show(5)
```

- ▶ drop elimina una columna.

```
import pyspark.sql.functions as F  
  
# Ambas formas de utilizar drop dan el mismo resultado  
  
df.drop("nombreColumna")  
  
df.drop(F.col("nombreOtraColumna"))
```

- ▶ withColumnRenamed renombra una columna.

```
import pyspark.sql.functions as F  
  
df.withColumnRenamed("nombreExistenteColumna", "nombreNuevoColumna")
```

- ▶ `when(condición, valorReemplazo1).otherwise(valorReemplazo2)` sirve para reemplazar valores de una columna según una condición que implica a esa o a otras columnas. Si no especificamos `otherwise`, los campos donde no se cumpla la condición se llenarán a `null`. Esta función se utiliza generalmente dentro de `withColumn`:

```
import pyspark.sql.functions as F  
  
df.withColumn("esMayor", F.when("edad>18", "mayor").otherwise("menor"))
```

## Transformaciones matemáticas y estadísticas con DataFrames

Existen numerosas funciones matemáticas y estadísticas disponibles para su aplicación sobre DataFrames. Dichas funciones generan columnas y DataFrames nuevos como, por ejemplo, los siguientes:

- ▶ Columna de números aleatorios:

```
import pyspark.sql.functions as F  
  
# Crea una nueva columna con números aleatorios de una uniforme:  
df = df.withColumn("unif", F.rand())  
  
# Crea una nueva columna con números aleatorios de una normal:  
df = df.withColumn("norm", F.randn())
```

- ▶ DataFrame con estadísticos descriptivos (media, desviación típica, máximo, mínimo...) (método `describe`):

```
df.describe().show()
```

- ▶ Operaciones matemáticas, entre otras:
  - Seno: F.sin()
  - Coseno: F.cos(F.col("nombreColumna"))
  - Raíz cuadrada: F.sqrt(F.col("nombreColumna"))

## Combinaciones y filtrado de DataFrames

Otro grupo de funciones son las relacionadas con la combinación y filtrado de DataFrames. Entre ellas, podemos encontrar:

- ▶ Unión de DataFrames

```
df3 = df1.unionAll(df2)
```

- ▶ Diferencia de DataFrames:

```
df3 = df1.except(df2)
```

- ▶ Filtrado de filas de un DataFrame. Existen dos formas de expresar la condición de filtrado:

```
import pyspark.sql.functions as F

# df3 tendrá las filas de df donde la columna c1 tenga un valor
# mayor que c2
df3 = df.where(F.col("c1")>F.col("c2"))

df3 = df.where("c1 > c2") # Es equivalente a lo anterior
```

## Operaciones de RDD aplicadas a DataFrames

En general, las operaciones que podíamos usar sobre RDD suelen estar disponibles

también para los DataFrames. No olvidemos que un DataFrame no es más que un RDD de objetos de tipo Row y que podemos acceder al RDD que envuelve el DataFrame mediante el atributo rdd :

```
df.rdd.take(5) # Muestra 5 objetos de tipo Row
```

- ▶ map y flatmap sobre el RDD; en este caso, iteramos sobre objetos de tipo Row :

```
<pr><code>def mifuncion(r): # r es un Row y se accede a sus campos mediante
'.'

return((r.DNI, "Bienvenido " + r.nombre + " " + r.apellido))

paresRDD = df.rdd.map(mifuncion) # map sobre un RDD devuelve un RDD

dfRenombrado = paresRDD.toDF("DNI", "mensaje") # lo convertimos a DF
```

- ▶ Otras operaciones que pueden realizarse son: transformaciones como sample, sort, distinct, groupBy ..., y acciones como count, take, first , etc.

A continuación, se muestra un ejemplo un poco más completo del uso de la API estructurada:

```
# el objeto spark (sparkSession) ya está creado en Jupyter. Si no,
# habría que crearlo indicando la dirección IP del máster
# de un cluster de Spark existente:
from pyspark.sql import functions as F
df = spark.read.parquet("/mis/datos/en/hdfs/flights.parquet")
resultDF = df.withColumn("distMetros", F.col("dist")*1000) \
    .withColumn("retrasoCat",
F.when(F.col("retraso") < 15, "poco") \
    .when(F.col("retraso") < 30, "medio") \
    .otherwise("mucho")) # sin otherwise, pondría Null!
    .select(F.col("aeronave"), F.col("origen"),
    F.col("disMetros"),
    F.col("retrasoCat"),
    (F.col("retraso") / F.col("dist")).alias("retrasoPorKm"))
)\\"
```

```
.withColumnRenamed("dist", "distKm")
.where(F.col("aeronave") == "Boing 747") # equivale a .filter()
.where("distMetros > 20000 and origen != 'Madrid'")

# Línea anterior: where pasa una consulta SQL como string.
# En la línea siguiente, hacemos de nuevo lo mismo:
df2 = resultDF.select("retraso, compania")
    .where("aeropuerto like '%Barajas' and retrasoCat = 'poco'")
df2.show() # show es una acción que provoca que se calculen todos los
# DataFrames de las transformaciones anteriores hasta show.
```

## Operaciones de agrupamiento y agregación

El método `groupBy(“nombreCol1”, “nombreCol2”, …)` sobre un DataFrame devuelve una estructura de datos llamada `RelationalGroupedDataset`, que no es un DataFrame y sobre la que apenas se pueden aplicar operaciones. Equivale a la operación `GROUP BY` de SQL, donde se definen grupos para después calcular, para cada uno, un resultado agregado (por ejemplo, la suma, la media, un conteo, etc.) de una o varias variables numéricas.

Se suelen aplicar operaciones como `count()`, que efectúa un conteo del número de elementos de cada grupo, o la función `agg()`, que es la más habitual y realiza, para cada grupo, las agregaciones que le indiquemos sobre las columnas seleccionadas. El resultado solamente contendrá aquellas columnas que se incluyeron como argumentos en el `groupBy` más aquellas que sean mencionadas como argumento de alguna de las operaciones de agregación que incluimos en la función `agg`.

Cuando ejecutamos funciones de agregación sin haber llevado previamente una agrupación con `groupBy`, lo que obtenemos es un DataFrame con una sola fila, que es el resultado de la agregación. El fragmento de código siguiente muestra cómo se utilizan `groupBy` y `agg`.

```
import pyspark.sql.functions as F
newDF = myDF.agg(max(F.col("mycol"))) # devuelve DF de una sola fila
newDF = myDF.groupBy("mycol").agg(F.max(F.col("mycol")))) #Tantas filas
# como valores distintos en mycol
```

```
newDF = myDF.withColumn("complicated", # F.sin: seno. F.lit: constante
    F.lit(2)*F.sin(F.col("colA"))*F.sqrt(F.col("colB")))
newDF = myDF.groupBy("id").agg(F.count("id").alias("countId"),
    F.max("date").alias("maxdate"),
    F.countDistinct("prod").alias("nProd"))
# Sintaxis tipo diccionario para indicar varias agregaciones:
newDF = myDF.groupBy(F.col("someCol"))
    .agg({"existingCol": "min", "otherCol": "avg"})
# Indicamos que, en cada grupo, definido por cada valor de "someCol",
# queremos el mínimo de la columna "existingCol" en dicho grupo y la
# media de los valores de la columna "otherCol". Esto devuelve
# un DF con tantas filas como valores distintos tenga someCol.
```

## 4.5. Ejemplo de uso de API estructurada

En el capítulo anterior, vimos un ejemplo completo de manejo de RDD, con la intención de replicarlo con la API estructurada para observar las diferencias de uso entre ambas API.

Recordemos que, para ello, usábamos los ficheros simplificados `flights.csv` y `airport-codes.csv`, almacenados ambos en HDFS, bajo el directorio `/user/data`, y cuyas primeras líneas son las siguientes:

```
"year","month","day","origin","dest"  
2014,1,1,"PDX","ANC"  
2014,1,1,"SEA","CLT"  
...  
  
ident,name,iso_country,iata_code  
LELT,Lillo,ES,  
LEMD,Adolfo Suárez Madrid-Barajas Airport,ES,MAD  
...
```

El objetivo del ejemplo era contar cuántos vuelos reciben los diferentes destinos ( `dest` ) que aparecen en el fichero `flights.csv`. Además, como no conocemos bien los códigos de los aeropuertos de llegada, queríamos ver esta agregación como nombre completo del aeropuerto y número de vuelos que recibe. Para esto, nos ayudábamos de la información que contiene el fichero `airport-codes.csv`, donde se relacionaba el código del aeropuerto con su nombre. Veamos entonces cómo realizar esta misma funcionalidad usando la API estructurada.

```
# Empezamos obteniendo la SparkSession.  
# Como, en este caso, no vamos a usar RDD, no es necesario  
# obtener el SparkContext:  
from pyspark.sql import SparkSession
```

```
import pyspark.sql.functions as F

spark = SparkSession.builder.appName("ejemplo_DF")\
    .getOrCreate()

# Lo siguiente es cargar los datos de fichero
# en un DataFrame, usando las funciones de lectura
# que vimos anteriormente.
# Concretamente, usamos la opción header, que nos
# permite no tener que hacer código auxiliar para no cargar
# la cabecera como datos, como pasaba con RDD:
flightsDF = spark.read\
    .option("header", "true")\
    .csv("hdfs://<ip:port>/user/data/flights.csv")

# Cuando usamos DataFrames, tenemos dos opciones para obtener
# registros de ellos:
# Usar la función take de RDD:
print(flightsDF.take(5))

# Usar la función show de DataFrames:
print(flightsDF.show(5))
```

Vemos que, cuando usamos la acción `take`, esta devuelve cinco registros del RDD envuelto por el DataFrame. Y recordemos que un DataFrame no era más que una envoltura de un RDD de objetos de tipo `Row`, tal y como podemos comprobar con el resultado de mostrar los registros devueltos por `take`. Por otro lado, la API estructurada nos proporciona un método mucho más visual como es `show`, que muestra la misma información, pero en formato tabla.

```
[Row(year='2014', month='1', day='1', origin='PDX', dest='ANC'),
 Row(year='2014', month='1', day='1', origin='SEA', dest='CLT'),
 Row(year='2014', month='1', day='1', origin='PDX', dest='IAH'),
 Row(year='2014', month='1', day='1', origin='PDX', dest='CLT'),
 Row(year='2014', month='1', day='1', origin='SEA', dest='ANC')]

+---+---+---+---+
|year|month|day|origin|dest|
+---+---+---+---+
|2014|     1|   1|    PDX| ANC|
|2014|     1|   1|    SEA| CLT|
|2014|     1|   1|    PDX| IAH|
```

```
|2014|     1|   1|    PDX| CLT|
|2014|     1|   1|    SEA| ANC|
+----+----+----+----+
only showing top 5 rows
```

```
# Otro método útil que nos proporciona la API estructurada
# es printSchema, para comprobar el tipo de datos de cada
# columna del DataFrame:
flightsDF.printSchema()
```

Con `printSchema`, podemos comprobar el tipo de datos leído por Spark. Como, en este caso, no hemos indicado ni esquema ni valor para `inferSchema`, se aplica por defecto `false` y se lee todo como *string*.

```
root
 |-- year: string (nullable = true)
 |-- month: string (nullable = true)
 |-- day: string (nullable = true)
 |-- origin: string (nullable = true)
 |-- dest: string (nullable = true)
```

```
# Esto nos da pie a convertir el tipo de las columnas
# año, mes y día, y ver así cómo se hace.
flightsDF = flightsDF\
    .withColumn('year', F.col('year').cast(IntegerType()))\
    .withColumn('month', F.col('month').cast(IntegerType()))\
    .withColumn('day', F.col('day').cast(IntegerType()))
```

```
# Ahora, para contar cuantos vuelos llegan a cada aeropuerto de
# destino, basta con agrupar por destino y contar registros por grupo:
flights_destDF = flightsDF.groupBy('dest').count()

# Le cambiamos el nombre a la columna count para ver cómo se hace:
flights_destDF = flights_destDF\
    .withColumnRenamed('count', 'dest_count')

# Ordenamos el DataFrame por número de vuelos en orden
# descendente y mostramos los 5 primeros:
flights_destDF.orderBy(F.desc('dest_count')).show(5)
```

Como vemos, es mucho más sencillo hacer este tipo de manipulaciones con la API estructurada, ya que nos despreocupamos de usar RDD o pairRDD , de la estructura del RDD, etc., y solo tenemos que prestar atención a la estructura de tabla y a las operaciones que queremos realizar.

```
+-----+
|dest|dest_count|
+-----+
| SF0 |      12809 |
| LAX |      10456 |
| DEN |      9518  |
| PHX |      8660  |
| LAS |      8214  |
+-----+
only showing top 5 rows
```

```
# Para mostrar también el nombre de los aeropuertos,
# vamos a cargar el fichero correspondiente en otro DataFrame:
airportsDF = spark.read\
    .option("header", "true")\
    .csv("hdfs://<ip:port>/user/data/airport-codes.csv")

# Y solo queda hacer el join. Del resultado del join, nos quedamos
# con el nombre del aeropuerto (name) y el número de vuelos,
# y ordenamos por número de vuelos en orden descendente:
flights_airports = flights_destDF\
    .join(airportsDF, flights_destDF.dest == airportsDF.iata_code)\\
    .select(F.col('name'), F.col('dest_count'))\
    .orderBy(F.desc('dest_count'))
flights_airports.show(5)
```

Obtenemos los mismos resultados que con RDD, pero de una forma mucho más intuitiva y menos proclive a producir errores.

```
+-----+-----+
|           name|dest_count|
+-----+-----+
|San Francisco Int...|      12809 |
|Los Angeles Inter...|      10456 |
|Denver Internatio...|      9518  |
|Phoenix Sky Harbo...|      8660  |
```

```
|McCarren Internat...|      8214|
+-----+-----+
only showing top 5 rows
```

```
# Por último, para demostrar el uso, escribimos el resultado en ficheros:
flights_airports.write\
    .format('csv')\
    .save("hdfs://192.168.240.4:9000/user/data/flights_dest_airports")
```

Con este ejemplo, comprobamos de primera mano las facilidades que brinda la API estructurada, además de proporcionar las optimizaciones derivadas del uso del motor Catalyst. Queda claro por qué los desarrolladores de Spark recomiendan encarecidamente el uso de esta API, siempre que sea posible, en lugar de la de RDD.

## 4.6. Spark SQL

Spark SQL ofrece una potente opción, que consiste en aplicar operaciones escritas como consultas en lenguaje SQL a DataFrames que se hayan registrado como tablas, sin tener que utilizar la API estructurada paso a paso. Es decir: Spark SQL se integra con la API de DataFrames; así, se puede expresar parte de una consulta en SQL y parte utilizando la API estructurada. Sea cual sea la opción elegida, cabe recordar que se compilará en el mismo plan de ejecución, dado que, como se veía en la descripción de Spark, tiene un motor de ejecución unificado.

Antes de que Spark fuera una herramienta tan usada como lo es hoy en día, la tecnología *big data* que permitía hacer consultas usando lenguaje SQL sobre grandes conjuntos de datos almacenados de forma distribuida era Hive (que se estudiará en un capítulo posterior). Esta herramienta fue muy popular, ya que ayudó a que Hadoop fuera usado por profesionales que no tenían conocimientos suficientes de Java u otros lenguajes de programación para realizar procesamientos de los datos almacenados en HDFS utilizando MapReduce, pero que sí tenían amplios conocimientos en el uso de bases de datos SQL.

Por su parte, Spark comenzó como un motor de procesamiento paralelo de grandes cantidades de datos basado en RDD. Sin embargo, a partir de la versión 2.0, los autores incluyeron un *parser* de consultas SQL (que soportaba tanto ANSI-SQL como HiveQL, el lenguaje SQL de Hive), que dio lugar a una herramienta similar a Hive. Es decir, ofrecía la posibilidad de realizar consultas SQL sobre un conjunto de datos sin necesidad de tener conocimientos de Python, Java o Scala para ello, lo que hacía más accesible el procesamiento de grandes conjuntos de datos. **Spark SQL no reemplaza a Hive**, pero sí que incluye prácticamente toda la funcionalidad que proporciona Hive, atada a que la ejecución por debajo está anclada al motor de procesamiento de Spark.

Cabe aclarar que **Spark SQL** está pensado para funcionar como un **motor de procesado de consultas en batch** (OLTP) y no para realizar consultas interactivas o que necesiten una baja latencia (OLAP). Exactamente igual ocurre con Hive, como se comentará en el capítulo correspondiente.

Spark SQL proporciona su funcionalidad gracias al uso de un catálogo de metadatos, denominado Catalog. Dicho **Catalog** es una **abstracción del metastore**, es decir, del almacenamiento de los metadatos que definen las tablas y vistas (*dataframes* registrados) sobre los que ejecuta las consultas SQL. El Catalog envuelve la complejidad del *metastore* y proporciona una serie de funciones que se pueden ejecutar sobre él (por ejemplo, listar bases de datos, tablas y vistas, y funciones).

El **metastore** representado por el Catalog puede almacenarse bien en memoria (opción ‘in-memory’, por defecto al usar Spark con scripts), o bien en un **metastore** de Hive (opción ‘hive’, por defecto al usar Spark Shell, como con Jupyter Notebooks). En caso de usar la opción ‘hive’, se puede utilizar un *metastore* ya existente y que se especifica mediante configuración, o bien lo crea el propio Spark en caso de no especificar ninguno. En caso de usar un *metastore* de Hive, este metastore se puede acceder desde fuera de Spark, por ejemplo desde Hive o herramientas BI, de forma que es posible por ejemplo manejar y procesar datos con Spark, registrar las tablas resultantes en el *metastore*, para luego consultarlas con Hive o herramientas BI, configurando el acceso al mismo *metastore*. De igual forma, en caso de usar un *metastore* ya existente en el que hubiera tablas registradas, Spark SQL puede acceder y modificar dichas tablas aunque no hayan sido creadas desde Spark SQL.

Existen tres opciones para ejecutar consultas en Spark SQL:

- ▶ **Interfaz de línea de comandos de Spark SQL**. Es una herramienta útil para realizar consultas sencillas en modo local desde la línea de comandos. Para acceder a esta herramienta, basta con ejecutar `./bin/spark-sql` en línea de comandos, en el directorio donde esté Spark instalado.

- ▶ **API de Spark SQL.** El método `.sql()` de la `sparkSession` recibe como argumento una consulta SQL, que puede hacer referencia a cualquier tabla registrada en *metastore*, y devuelve un DataFrame con el resultado de la consulta. Para registrar un DataFrame en el *metastore*, lo cual genera (solamente) los metadatos necesarios, existen varios métodos; entre ellos, podemos mencionar `createOrReplaceTempView`, que la registra solo durante esta sesión, o `write.format('hive').saveAsTable("nombre_tabla")`, que la registra de forma permanente para futuras sesiones. Spark ofrece más métodos para manejar el *metastore*, pero quedan fuera del alcance de este tema. Spark analiza automáticamente la consulta y la traduce a un DAG, exactamente del mismo modo que ocurre al utilizar la API estructurada que hemos visto en las secciones anteriores. Veamos un ejemplo:

```
from pyspark.sql import functions as F
df = spark.read.parquet("/mis/datos/en/hdfs/flights.parquet")
df.createOrReplaceTempView("vuelos") # crear tabla temporal vuelos
resultDF = spark.sql("select *, 1000*dist as distMetros from vuelos")
resultsDF.show() # todas las cols originales más una nueva distMetros
```

Además, la API SQL es totalmente interoperable con la API estructurada, de forma que se puede crear un DataFrame, manipularlo primero con SQL y después con la API estructurada. Es decir, se puede hacer código como el siguiente:

```
from pyspark.sql import functions as F
df = spark.read.parquet("/mis/datos/en/hdfs/flights.parquet")
df.createOrReplaceTempView("vuelos") # crear tabla temporal vuelos
resultDF = spark.sql("select *, 1000*dist as distMetros from vuelos")
    .where("distMetros > 100000") # API SQL + estructurada
resultsDF.show() # Todos los vuelos con distancias >100 km
```

- ▶ **Servidor JDBC/ODBC.** Spark proporciona una interfaz JDBC/ODBC, mediante la cual aplicaciones BI, como Tableau, tienen acceso al *metastore* gestionado por Spark SQL, y pueden lanzar consultas SQL sobre las tablas registradas en dicho *metastore*, que se ejecutarán de forma distribuida sobre el clúster de Spark.

## Tablas en Spark SQL

El elemento de trabajo en Spark SQL son las tablas, equivalente a los DataFrames en la API estructurada. Toda tabla pertenece a una base de datos (*database*) y, si no se especifica ninguna, lo hará a la base de datos por defecto (*default*). Las tablas siempre contienen datos y no existe el concepto de tabla temporal. En su lugar, existen vistas, que no contienen datos. Es importante tener esto en cuenta a la hora de eliminar (*drop*) vistas (no se eliminan datos) y tablas (se eliminan datos).

Otro aspecto que debemos tener en cuenta al crear tablas es si se desea que estas sean gestionadas por Spark (*managed table*) o no (*unmanaged table*). Para entender este concepto, cabe mencionar que una tabla está formada por dos tipos de información: los datos que contiene y los metadatos que la describen. Con esto presente, podemos ver cómo se diferencian las tablas gestionadas y no gestionadas por Spark:

- ▶ **Tablas gestionadas por Spark.** Cuando se guarda un DataFrame como una nueva tabla (usando por ejemplo `saveAsTable` sobre un DataFrame, o `CREATE TABLE`), entonces se crea una tabla gestionada por Spark, ya que son datos nuevos que necesitan almacenarse, y Spark es el encargado de ello. De esta forma, Spark es responsable tanto de los datos como de los metadatos de estas tablas gestionadas, y si borramos estas tablas desde Spark, se borrarán tanto los datos como los metadatos.
- ▶ **Tablas no gestionadas por Spark.** En este caso, Spark gestiona solo los metadatos asociados a la tabla, mientras que nosotros gestionamos los datos, es decir, nosotros controlados dónde se guardan y cuándo se borran. Por tanto, cuando borremos la tabla no gestionada con Spark, solo se eliminarán los metadatos, mientras que los datos quedarán almacenados hasta que nosotros decidamos borrarlos. Podemos crear tablas no gestionadas por Spark de diversas formas. Por ejemplo, cuando se define una tabla desde ficheros ya almacenados en disco, lo que se crea es una tabla no gestionada por Spark, dado que los datos ya existían

previamente, no son datos nuevos creados usando Spark. Esto se consigue especificando un path concreto antes de usar `saveAsTable` (por ejemplo, `df.write.option("path", "/path/to/save").saveAsTable("table_name")`) . Otra opción es utilizar `CREATE EXTERNAL TABLE` , lo que se conoce en muchos foros como tablas externas. Esta opción existe por compatibilidad con Hive, y permite crear tablas no gestionadas por Spark. Para crear una tabla externa se utiliza la consulta `CREATE EXTERNAL TABLE` , tal y como se muestra en el ejemplo a continuación:

```
CREATE EXTERNAL TABLE flights (
  DEST_COUNTRY_NAME STRING, ORIGIN_COUNTRY_NAME STRING)
ROW FORMAT DELIMITED
  FIELDS TERMINATED BY ','
LOCATION 'data/flight_info/'
```

O también desde el resultado de otra tabla:

```
CREATE EXTERNAL TABLE flights
ROW FORMAT DELIMITED
  FIELDS TERMINATED BY ','
LOCATION 'data/flight_info/'
AS SELECT * FROM flights
```

Cuando queremos eliminar una tabla, usamos la consulta `DROPO`. Es importante recordar que, en el caso de tablas gestionadas por Spark, se eliminarán tanto los datos como los metadatos, mientras que, si la tabla no está gestionada por Spark, se eliminarán los metadatos (no se podrá volver a hacer referencia a la tabla eliminada), pero los datos originales no (por ejemplo, si la tabla se creó a partir de un fichero, este quedará intacto).

## Vistas en Spark SQL

Un elemento auxiliar en Spark SQL son las vistas. Una vista especifica un conjunto de transformaciones sobre una tabla existente. Es decir, no son tablas, sino que definen el conjunto de operaciones que se harán sobre los datos almacenados en

cierta tabla para conseguir unos resultados. Las vistas se muestran como tablas, pero no guardan los datos en una nueva localización. Sencillamente, cuando se consultan, ejecutan las transformaciones definidas en ellas sobre la fuente de los datos. Por ejemplo, el siguiente ejemplo crea una vista:

```
CREATE VIEW flights_view AS
    SELECT * FROM flights
    WHERE dest_country_name = 'Spain'
```

La vista no contiene las filas de la tabla *flights* cuyo destino sea *Spain*, sino que únicamente almacena el plan de ejecución necesario para obtener esas filas de la tabla origen, de forma que las pueda mostrar cada vez que sea consultada. Si lo pensamos, una vista no es más que una transformación en Spark que nos devuelve un nuevo DataFrame a partir de otro DataFrame de origen; por tanto, no se ejecutará hasta que se haga una consulta que requiera obtener la vista. El principal beneficio es que evita escribir datos en disco repetidamente (como sería el caso de crear nuevas tablas). Existen diferentes tipos de vistas:

- ▶ **Vistas estándar**, que están disponibles de sesión en sesión, como la que veíamos en el ejemplo previo.
- ▶ **Vistas temporales**, que solo están disponibles en la sesión actual.

```
CREATE TEMP VIEW flights_view AS
    SELECT *
    FROM flights
    WHERE dest_country_name = 'Spain'
```

- ▶ **Vistas globales**, que son accesibles desde cualquier lugar de la aplicación Spark y no pertenecen a ninguna base de datos en concreto, pero se eliminan al final de la sesión.

```
CREATE GLOBAL TEMP VIEW flights_view AS
  SELECT *
    FROM flights
   WHERE dest_country_name = 'Spain'
```

Al crear una vista, podemos indicar que reemplace otra existente:

```
CREATE OR REPLACE TEMP VIEW flights_view AS
  SELECT *
    FROM flights
   WHERE dest_country_name = 'Spain'
```

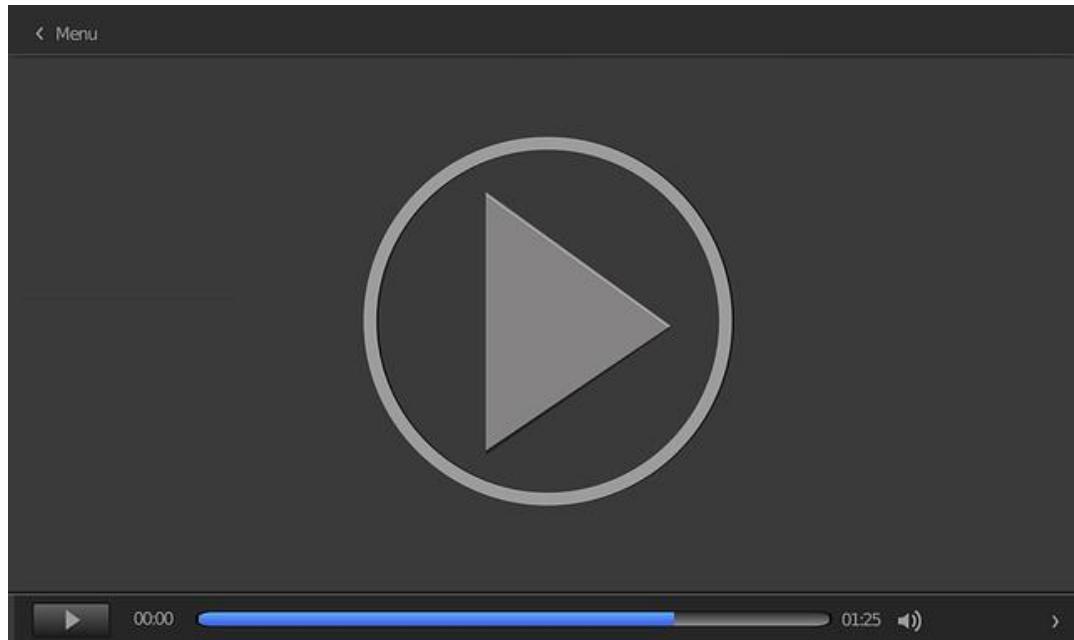
Cuando ejecutamos la sentencia para crear una vista, vemos que no ocurre nada. Recordemos que dicha vista no se crea (no se ejecutan las transformaciones asociadas) hasta que se hacen consultas sobre ella, como, por ejemplo:

```
SELECT *
  FROM flights_view; # ahora es cuando se ejecuta la vista.
```

Respecto a las consultas que se pueden hacer con las vistas, son las habituales de una tabla. Finalmente, podemos descartar una vista usando DROP:

```
DROP VIEW IF EXISTS flights_view;
```

En el vídeo que exponemos a continuación, realizamos una introducción a Amazon Web Services (AWS). Concretamente, vamos a probar el servicio Elastic MapReduce con JupyterLab para ejecutar Spark.



---

Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=2ff50934-cd2e-4338-9285-ac8201622468>

---

## 4.7. Ejemplo de Spark SQL

De nuevo, vamos a partir del ejemplo que hemos visto en el capítulo previo con la API básica y en este capítulo con la API estructurada. En esta ocasión, vamos a reproducirlo usando la API de Spark SQL.

```
# Supongamos que tenemos la SparkSession ya cargada.
# Empezamos por cargar los datos:
flightsDF = spark.read\
    .option("header", "true")\
    .csv("hdfs://<ip:port>/user/data/flights.csv")

# Para el uso de la API estructurada, es primordial registrar
# todos los DataFrames que queramos usar como tablas o vistas:
flightsDF.createOrReplaceTempView('flights')

# Calculamos los vuelos que llegan a cada aeropuerto
# de destino usando una sentencia SQL:
flights_dest_count = spark.sql('SELECT dest, COUNT(dest) AS dest_count FROM
flights GROUP BY dest ORDER BY dest_count DESC')

# De nuevo, para trabajar posteriormente con el resultado
# de esta consulta, necesitamos registrarla como vista:
flights_dest_count.createOrReplaceTempView('flights_dest_count')
flights_dest_count.show(5)
```

Obtenemos idéntico resultado que con la API estructurada. Y no solo eso, sino que cabe recordar que, a pesar de que el código es distinto porque usa API diferentes, al final ambas opciones se traducen en el mismo plan de ejecución, ya que Spark está concebido como un motor de procesamiento unificado.

dest	dest_count
SFO	12809
LAX	10456
DEN	9518
PHX	8660

```
| LAS|      8214|
+---+-----+
only showing top 5 rows
```

```
# Cargamos la información sobre los aeropuertos:
airportsDF = spark.read\
    .option("header", "true")\
    .csv("hdfs://192.168.240.4:9000/user/data/airport-codes.csv")
# Registrarmos el DataFrame como vista:
airportsDF.createOrReplaceTempView('airports')

# Llegados a este punto, solo resta hacer el join:
flights_dest_airports = spark.sql('SELECT a.name, f.dest_count FROM
flights_dest_count f JOIN airports a ON f.dest=a.iata_code ORDER BY
dest_count DESC')
flights_dest_airports.show(5)
```

Como podemos ver, obtenemos una vez más el mismo resultado. La diferencia recae en si el desarrollador tiene más manejo de sentencias SQL o de código .

```
+-----+-----+
|          name|dest_count|
+-----+-----+
|San Francisco Int...|      12809|
|Los Angeles Inter...|      10456|
|Denver Internatio...|       9518|
|Phoenix Sky Harbo...|      8660|
|McCarran Internat...|      8214|
+-----+-----+
only showing top 5 rows
```

## Documentación oficial de Apache Spark

Apache Spark. Página web oficial: <https://spark.apache.org/docs/latest/>

Documentación *online*, detallada y de muy buena calidad sobre el sistema de computación Apache Spark.

## Hadoop: the end of an era

Grishchenko, A. (2019, 23 de marzo). Hadoop: the end of an era [entrada de blog].

*Distributed Systems Architecture.* <https://0x0fff.com/hadoop-the-end-of-an-era/>

Hadoop es uno de los *frameworks* más importantes para el *big data*, cuyo propósito es almacenar grandes cantidades de datos y permitir consultas sobre estos, que se ofrecerán con un bajo tiempo de respuesta. Nació como iniciativa de Apache para dar soporte al paradigma de programación MapReduce, que fue inicialmente publicado por Google. En esta entrada de blog, se propone una interesante reflexión sobre su uso en la actualidad.

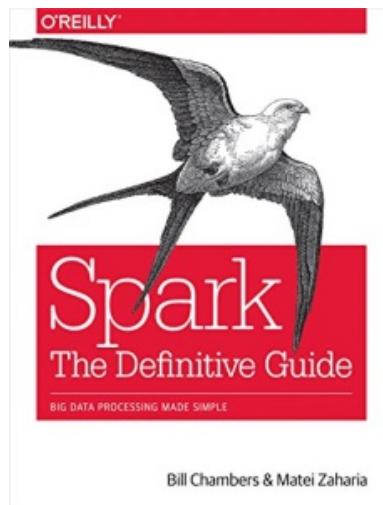
## DATA + AI Summit

DATA + AI Summit. Página web oficial: <https://databricks.com/sparkaisummit>

Sitio web del congreso más famoso de Spark a nivel mundial, del que tienen lugar también versiones más reducidas en cada continente. Está organizado por Databricks, empresa que soporta el desarrollo de Spark.

## Spark: the definitive guide

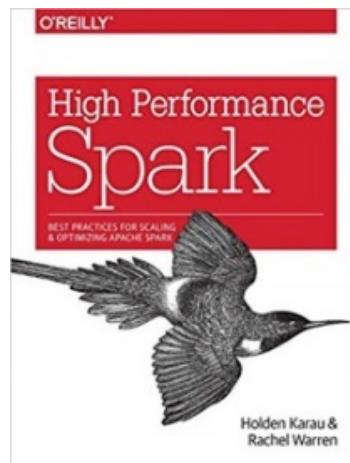
Chambers, B. y Zaharia, M. (2018). *Spark: the definitive guide*. O'Reilly.



Guía detallada de Spark, en su versión más actualizada. Contiene numerosos ejemplos y muestra exhaustivamente todas las capacidades de Spark.

## High performance Spark

Karau, H. y Warren, R. (2017). *High performance Spark*. O'Reilly.



Manual avanzado sobre cómo escribir código optimizado en Spark.

1. Elige la respuesta correcta respecto a los DataFrames de Spark:

  - A. Un RDD es una envoltura de un DataFrame de objetos de tipo Row.
  - B. Un DataFrame es una envoltura de un RDD de objetos de tipo Row.
  - C. Un DataFrame es una envoltura de un objeto de tipo Row que contiene RDD.
  - D. Ninguna de las respuestas anteriores es correcta.
2. Elige la respuesta correcta sobre los DataFrames de Spark:

  - A. Puesto que representan una estructura de datos más compleja que un RDD, no es posible distribuirlos en memoria.
  - B. Puesto que son un envoltorio de un RDD, suponen una estructura de datos que sigue estando distribuida en memoria.
  - C. Son una estructura de datos no distribuida en memoria, al igual que los DataFrames de Python o los data.frames de R.
  - D. Son una estructura de datos distribuida en disco.
3. ¿Qué mecanismo ofrece la API estructurada de DataFrames para leer datos?

  - A. Método *read* de la Spark Session.
  - B. Método *read* del Spark Context.
  - C. No ofrece ningún método, sino que se utiliza la API de RDD para leer datos.
  - D. Método *ingest* de la Spark Session.

**4.** ¿Es obligatorio especificar explícitamente el esquema del DataFrame cuando se leen datos de fichero?

- A. No, porque solo se pueden leer ficheros estructurados como Parquet, que ya contienen información sobre su esquema.
- B. Sí, porque, si no se indica el esquema, Spark no es capaz de leer ficheros CSV, ya que no sabe con qué tipo almacenar cada campo.
- C. No, porque, si no se indica el esquema, Spark guardará todos los campos de los que no sepa su tipo como *strings*.
- D. No, porque, si no se indica el esquema y se intenta leer ficheros sin esquema implícito, Spark lanzará un error.

**5.** Seleccione la respuesta incorrecta: ¿Por qué es aconsejable utilizar DataFrames en Spark en lugar de RDD?

- A. Porque son más intuitivos y fáciles de manejar a alto nivel.
- B. Porque son más rápidos, debido a optimizaciones realizadas por Catalyst.
- C. Porque los DataFrames ocupan menos en disco.
- D. Las respuestas A y B son correctas.

**6.** Tras ejecutar la operación `b = df.withColumn("nueva", 2*col("calif")):`

- A. El DataFrame contenido en `df` tendrá una nueva columna, llamada `nueva`.
- B. Llevaremos al `driver` el resultado de multiplicar 2 por la columna `calif`.
- C. El DataFrame contenido en `b` tendrá una columna más que `df`.
- D. El DataFrame contenido en `b` tendrá una única columna llamada `nueva`.

**7.** ¿Cuál es la operación con la que nos quedamos con el subconjunto de filas de un DataFrame que cumplen una determinada condición?

- A. sample.
- B. filter.
- C. map.
- D. show.

**8.** Las API estructuradas de DataFrames y Spark SQL...

- A. Son API que no se pueden combinar: una vez que se empieza a usar una de ellas, se tienen que hacer todas las tareas con la misma API.
- B. Se pueden aplicar funciones de la API de DataFrames sobre el resultado de consultas de Spark SQL.
- C. Se pueden aplicar el método `sql` para lanzar consultas SQL sobre DataFrames sin registrar.
- D. Ninguna de las opciones anteriores es correcta.

**9.** La transformación `map` de Spark...

- A. No se puede aplicar a un DataFrame porque pertenece a la API de RDD.
- B. Se puede aplicar a un DataFrame porque pertenece a la API estructurada de DataFrames.
- C. Se puede aplicar a un DataFrame porque envuelve un RDD al que se puede acceder mediante el atributo `rdd`.
- D. No existe en Spark; `map` es una acción.

**10.** Para utilizar Spark SQL, es necesario...

- A. Utilizar la función `sql` del objeto `SparkContext`.
- B. Utilizar la función `sql` del objeto `SparkSession`, a fin de ejecutar la consulta SQL sobre el `DataFrame` directamente.
- C. Registrar el `DataFrame` sobre el que se quieran ejecutar las consultas SQL como tabla o vista, antes de ejecutar cualquier consulta.
- D. Ninguna de las respuestas anteriores es correcta.

Ingeniería para el Procesado Masivo de Datos

---

## Tema 5. Spark III

# Índice

[Esquema](#)

[Ideas clave](#)

[5.1. Introducción y objetivos](#)

[5.2. Spark MLlib](#)

[5.3. Spark Structured Streaming](#)

[5.4. Referencias bibliográficas](#)

[A fondo](#)

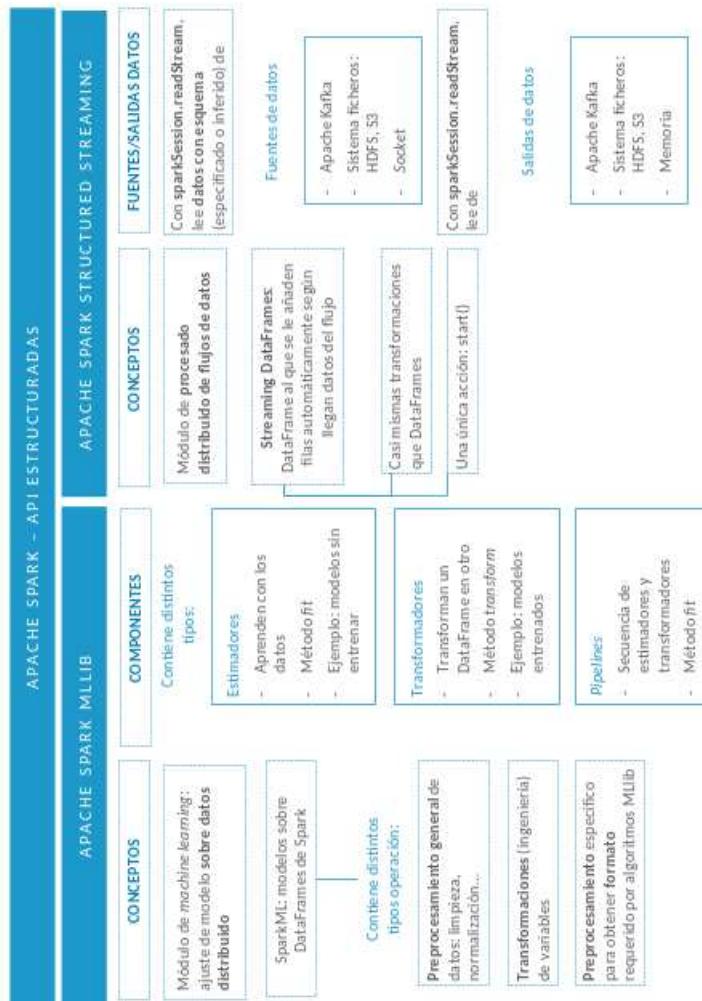
[Documentación oficial de Apache Spark](#)

[Spark: the definitive guide](#)

[Hadoop: the definitive guide](#)

[Test](#)

# Esquema



## 5.1. Introducción y objetivos

Una vez expuestas las características fundamentales de Spark, en este tema nos detendremos en los módulos que proporcionan funcionalidad de más alto nivel. Empezaremos describiendo Spark MLlib y explicaremos los métodos básicos que ofrece para ajustar modelos; a continuación, examinaremos el módulo Spark Streaming para procesar datos en tiempo real. Con esto cerraremos el capítulo dedicado a Spark. Finalizaremos el tema presentando Apache Hive, otra herramienta del ecosistema Hadoop para hacer consultas SQL sobre datos distribuidos, y que puede utilizar como motor de ejecución MapReduce, Spark o Tez indistintamente. Con todo ello, los objetivos que persigue este tema son:

- ▶ Introducir al alumno en los módulos de alto nivel de Spark.
- ▶ Presentar las capacidades de Spark MLlib mediante ejemplos de uso que servirán de base para desarrollar la actividad entregable.
- ▶ Mostrar las ideas básicas y un ejemplo sencillo de Spark Structured Streaming.

## 5.2. Spark MLlib

Spark MLlib es el módulo de Spark para tareas de:

- ▶ Limpieza de datos.
- ▶ Ingeniería de variables (creación de variables desde datos en crudo).
- ▶ Aprendizaje de modelos sobre *datasets* muy grandes (distribuidos).
- ▶ Ajuste de parámetros y evaluación de modelos.

No proporciona métodos para despliegue en producción de modelos entrenados. En la actualidad, es muy frecuente desplegar modelos como microservicios. Estos usan el modelo entrenado para realizar predicciones sobre un nuevo ejemplo, el cual reciben por medio de una llamada HTTP de una aplicación externa que les ha enviado el dato para predecir. Comentaremos más detalles en la próxima sección.

La esencia de MLlib es la implementación de modelos de manera distribuida utilizando Spark. Dichos modelos son capaces de aprender sobre datasets muy grandes almacenados de manera distribuida. No obstante, también es muy frecuente utilizar solo ciertas funcionalidades de MLlib para preprocesar variables en *datasets* masivos, limpiar, normalizar, etc., y, finalmente, para filtrar y pasar el *dataset* resultante (asumiendo que ya no es masivo) al *driver*, a fin de guardarlo en el sistema de archivos local. Posteriormente se puede utilizar una biblioteca de *machine learning* no distribuida, como, por ejemplo, Scikit-learn de Python o paquetes específicos de R como caret o e1071. Tanto Python como R son capaces de leer ficheros no distribuidos en formato texto o CSV, entre otros, y de usarlos como entrada para un algoritmo de aprendizaje automático.

La figura 1 muestra una idea general del ciclo de ajuste de un modelo, junto a las herramientas que proporciona Spark para cada etapa (en color rojo). Además, Spark

tiene una herramienta adicional, llamada *pipelines*, para encapsular todas estas etapas en un solo objeto indivisible y asegurarnos de que los nuevos datos recibidos, y con los que queremos realizar predicciones, pasen por exactamente las mismas etapas de preprocesamiento que el *dataset* estático con el que se entrenó el modelo. Por eso, la figura se refiere a estos pasos como un *all-in-one pipeline*.

La etapa de preprocesamiento incluye, además de las operaciones típicas de un proyecto de *data science*, cierto procesamiento específico de Spark. La finalidad es pasar los datos al formato adecuado de columnas que esperan recibir las implementaciones de cada algoritmo en la API de Spark ML. MLlib incorpora herramientas para esto también.

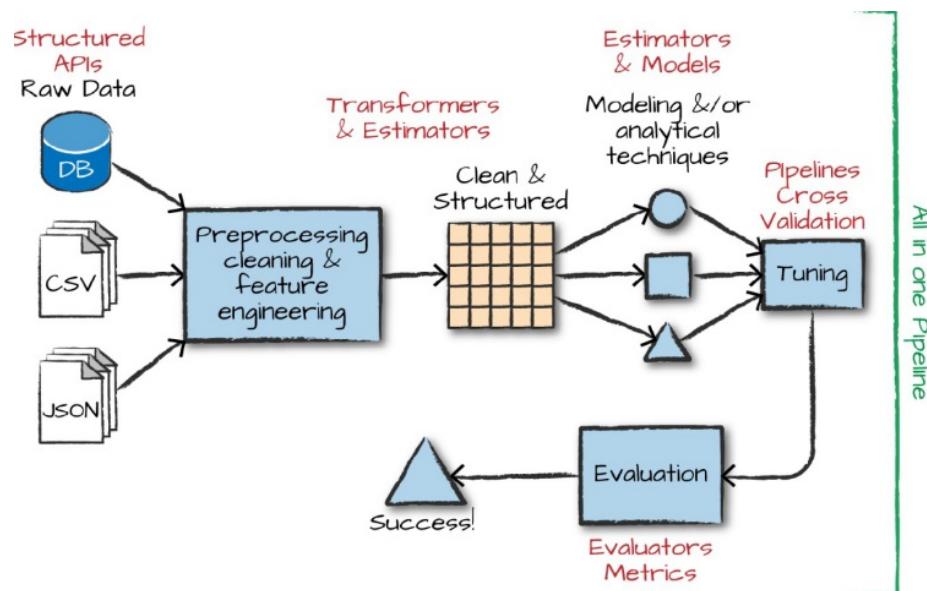


Figura 1. Etapas en el proceso de entrenamiento de un modelo a partir de datos. Fuente: Apache Spark

2.4.5.

## Despliegue de modelos en producción con Spark

Spark no fue concebido para explotación *online* de modelos entrenados, es decir, para dar respuestas rápidas (predicciones) a un ejemplo nuevo que se recibe, por ejemplo, desde un sitio web. Por el contrario, la fortaleza de Spark está en entrenar

modelos en modo *batch (offline)* con conjuntos de datos muy grandes.

Aun así, hay varias formas de aprovechar el modelo entrenado obtenido por Spark:

- ▶ **Entrenar con datos *offline* almacenados en HDFS** (el proceso de creación de variables y entrenamiento llevará cierto tiempo) y, justo después, usar el modelo entrenado para predecir (también en modo *batch*) otro conjunto de datos nuevos. La etapa de predicción es mucho más rápida que la de entrenamiento.
  - Es un enfoque muy frecuente. Es posible cuando los datos que hay que predecir (excepto la columna objetivo) ya se conocen en el momento de entrenar, por ejemplo, series temporales para predecir una ventana a futuro.
  - Las predicciones precalculadas se almacenan en HDFS o en bases de datos indexadas para que sea muy rápido servirlas desde un microservicio.
- ▶ También es posible hacer una predicción en *batch* en otro momento que no sea inmediatamente después del entrenamiento, sino cuando tengamos suficientes datos nuevos (un nuevo lote, considerable) sobre los que predecir.
- ▶ Entrenar, guardar el modelo entrenado y usarlo desde Spark para hacer predicciones una a una. Poco recomendable: lanzar un *job* para cada ejemplo que predecir implica sobrecarga. Balanceo de carga con réplicas del modelo.
- ▶ Entrenar y exportar el modelo a un formato de intercambio. Ejemplo: PMML, para leerlo y explotarlo con otra herramienta no distribuida (Python en especial, aunque también R soporta archivos en formato PMML).
- ▶ Entrenar en modo *online* usando Structured Streaming para recoger datos. Exige reentrenar desde cero, salvo que el algoritmo esté preparado para entrenamiento incremental (modelos de *online learning*, que, en la actualidad, son una minoría).

## Estimadores y transformadores

Antes de describir en detalle la API del módulo Spark MLlib, hay que tener en cuenta que, en la API de Spark (Java/Scala/Python), se distinguen:

- ▶ **Paquete** org.apache.spark.mllib (**en Python:** pyspark.mllib) : API antigua basada en RDD de una estructura llamada LabeledPoint: LabeledPoint(etiqueta, [vector de atributos])
  - . Obsoleta, no debe usarse.
- ▶ **Paquete** org.apache.spark.ml (**en Python:** pyspark.ml) : API actual, sobre DataFrames.
  - En la medida de lo posible, se debe utilizar siempre.
  - Casi todo el contenido del módulo spark.mllib ya está migrado al módulo spark.ml , excepto algunas clases en métricas de evaluación y algún algoritmo de recomendación.

Para la creación de variables y el preprocessamiento, en general, se utiliza la API de Spark SQL que vimos en el tema anterior. No obstante, Spark ML ofrece una transformación en la que puede escribirse código SQL arbitrario e incluir dicha transformación en un *pipeline*. Además, ciertas transformaciones relacionadas con el preprocessamiento estadístico de datos (normalización, estandarización, codificación *one-hot*, etc.) también están disponibles en Spark ML.

Por último, existen varias transformaciones cuyo propósito no es realmente modificar los datos, sino prepararlos (dar a las columnas del DataFrame los tipos adecuados) para la entrada de los algoritmos de Machine Learning de Spark. En concreto, Spark requiere estos formatos:

<b>features</b>	<b>label</b>
[-32.2, 4.5, 1.0, 6.7]	1.0
[-40.8, 2.25, 4.0, 2.3]	0.0

Tabla 1. Problemas de clasificación (clase codificada como número real).

<b>features</b>	<b>target</b>
[-32.2, 4.5, 1.0, 6.7]	0.72
[-40.8, 2.25, 4.0, 2.3]	-4.56

Tabla 2. Problemas de regresión (el *target* ya es un número real).

<b>features</b>
[-32.2, 4.5, 1.0, 6.7]
[-40.8, 2.25, 4.0, 2.3]

Tabla 3. Problemas de *clustering* (no existe columna *label*).

<b>user</b>	<b>item</b>	<b>rating</b>
2	3	4.5
1	19	3.21

Tabla 4. Problemas de recomendación (con filtrado colaborativo).

Como vemos, en todos los algoritmos, los valores de las variables deben presentarse en una sola columna de tipo vector. Por otro lado, si se trata de un problema de aprendizaje supervisado (sea de clasificación o de regresión), la columna *target* debe ser siempre de tipo real (*double*). En el caso de los problemas de clasificación, cada clase se indica con un número real, que ha de empezar en 0.0 y con la parte decimal del número siempre a 0 (es decir, si tenemos un problema con cinco clases, se tienen que codificar como 0.0, 1.0, 2.0, 3.0 y 4.0).

En relación con los problemas de regresión, la columna *target* puede ser cualquier número real. Los nombres de las columnas no tienen por qué ser *features* y *label* como en los ejemplos de arriba, sino que pueden ser cualesquiera, siempre que le indiquemos al algoritmo cómo se llama la columna (de tipo vector) que contiene las *features* en el DataFrame que le pasamos para entrenar y cómo se llama la columna *target*, si la hay.

Si hablamos de algoritmos de recomendación, Spark solo permite filtrado colaborativo. En ese caso, hay que pasarle un DataFrame que contenga, al menos, tres columnas con los identificadores de un usuario, un ítem y el *rating* que ha dado dicho usuario a ese ítem, ya sea implícito o explícito. Los identificadores de usuarios

y de ítems tienen que ser códigos de tipo entero, mientras que el *rating* puede ser un número real en cualquier rango.

Según la documentación oficial de Spark, actualmente nos ofrece las siguientes posibilidades para preprocesamiento, divididas en varios grupos:

- ▶ Extracción: extraer variables a partir de datos en crudo.
- ▶ Transformación: reescalar, convertir o modificar variables.
- ▶ Seleccionar: seleccionar un buen subconjunto de variables de otro más grande.
- ▶ *Locality sensitive hashing* (LSH): combinación de transformaciones de variables con otros algoritmos.

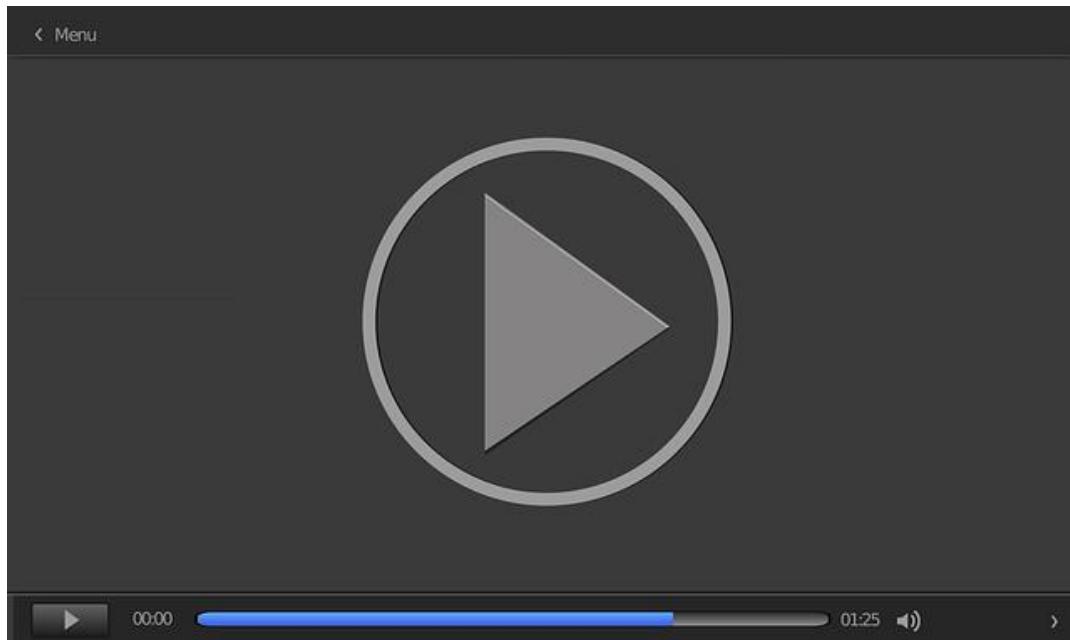
---

Para más detalle consultar la documentación de Spark:

<https://spark.apache.org/docs/latest/ml-features.html>

---

Para finalizar este apartado, en el siguiente vídeo, vamos a mostrar las capacidades de Spark ML en un problema de análisis de sentimiento.



---

Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=1007c11e-1c7e-443a-9dbb-b01100a57e6f>

---

## Transformadores en Spark ML

Un *transformer* es un objeto que recibe como entrada un DataFrame de Spark y uno o varios nombres de columna existentes (por ejemplo, *inputCol*), y los transforma de alguna manera. Su salida es el mismo DataFrame con una nueva columna añadida a la derecha, con el nombre que hayamos indicado en el parámetro correspondiente (generalmente, *outputCol*, pero, a veces, puede ser *predictionCol*).

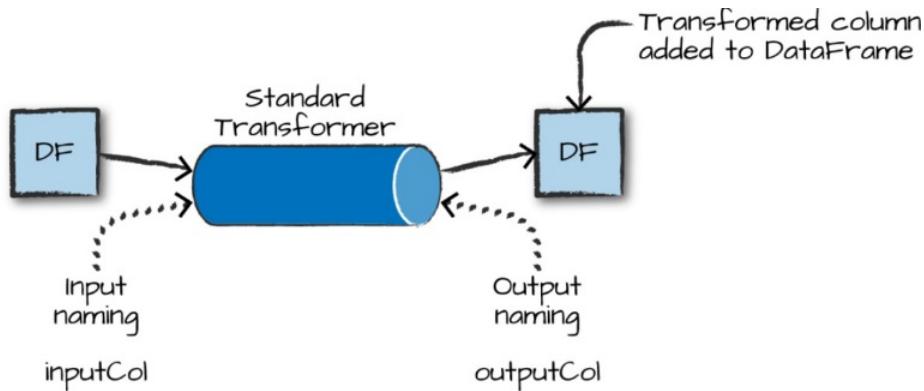


Figura 2. Funcionamiento de un transformador de Spark ML. Fuente: Apache Spark 2.4.5.

La interfaz **Transformer** tiene un único método: `transform(df: DataFrame)`, que recibe un DataFrame y devuelve otro DataFrame. Los transformadores **no necesitan aprender ningún parámetro del DataFrame de entrada**. Simplemente están preparados (tienen toda la información) para transformar un DataFrame, y esto es lo que hacen cuando llamamos a `transform()`, tal como muestra la figura 2.

## Algunos transformadores habituales

- ▶ **VectorAssembler:** recibe varias columnas y las concatena en una sola de tipo vector, de longitud igual al número de columnas que se quieran ensamblar. Necesario para calcular la columna (única) de *features* en los algoritmos de aprendizaje supervisado. Ejemplo:

```
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler

dataset = spark.createDataFrame(
    [(0, 18, 1.0, Vectors.dense([0.0, 10.0, 0.51]), 1.0)],
    ["id", "hour", "mobile", "userFeatures", "clicked"])

assembler = VectorAssembler(
    inputCols=["hour", "mobile", "userFeatures"],
    outputCol="features")
output = assembler.transform(dataset)
print("Assembled hour, mobile, userFeatures to column 'features'")
```

```
output.select("features", "clicked").show(truncate = False)
```



id	hour	mobile	userFeatures	clicked
0	18	1.0	[0.0, 10.0, 0.5]	1.0

id	hour	mobile	userFeatures	clicked	features
0	18	1.0	[0.0, 10.0, 0.5]	1.0	[18.0, 1.0, 0.0, 10.0, 0.5]

- ▶ **Cualquier modelo entrenado:** el resultado de entrenar un modelo sobre un DF es un objeto *model* de la subclase específica del modelo que hayamos ajustado. También es un *transformer*, por lo que es capaz de transformar (hacer predicciones) un DF de ejemplos, siempre que tenga el mismo formato (mismos nombres de columnas y tipos de datos) que el DF que se utilizó para entrenar.
  - Las predicciones se añaden junto a cada ejemplo en una nueva columna.
  - Para facilitar que se mantenga el mismo formato, se suele entrenar un *pipeline* completo y utilizar su salida (*pipeline* entrenado) como transformador.

```
from pyspark.ml.regression import LinearRegression
from pyspark.ml import Transformer

training = spark.read.format("csv")\
    .load("sample_linear_regression_data.csv")

lr = LinearRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)
lrModel = lr.fit(training)
lrModel.__class__ # comprobamos la clase del modelo ajustado
# Devuelve: <class 'pyspark.ml.regression.LinearRegressionModel'>

pred = lrModel.transform(training)
pred.show()
```

## Estimadores en Spark ML

Un *estimator* es un objeto de Spark capaz de realizar transformaciones que primero requieren que ciertos parámetros de la transformación se ajusten (o se *aprendan*) a partir de los datos. Normalmente precisan una pasada previa (o varias) sobre la columna de datos que se desea transformar.

La interfaz **Estimator** tiene un único método: `fit(df: DataFrame)`, que recibe un DataFrame y devuelve un objeto de tipo **model** (el modelo entrenado), que es, además, un **transformer**, tal como explicamos anteriormente. Es importante notar que Spark llama *modelo* a cualquier cosa que requiera un *fit* previo, no solo a los algoritmos de aprendizaje automático que conocemos como modelos propiamente.

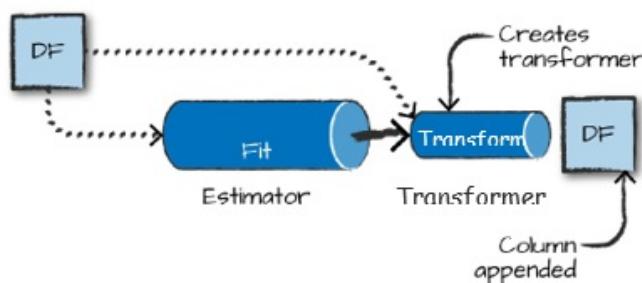


Figura 3. Estimador que genera un transformador. Fuente: Chambers y Zaharia , 2018.

## Estimadores más comunes

- ▶ **StringIndexer:** estimador para preprocessar variables categóricas . Es el más utilizado. Convierte una columna (de cualquier tipo, ya que los valores serán interpretados como categorías) en números reales (*double*), con la parte decimal a 0 y empezando en 0.0. Las categorías se representan mediante 0.0, 1.0, 2.0, etc.
- Además, añade metadatos al DataFrame transformado devuelto por `transform()` , con los que indica que esa columna es categórica y no como cualquier otra columna numérica. Esta información es útil para los algoritmos.

- Los algoritmos que sí soportan variables categóricas (ejemplo: DecisionTree, RandomForest, GradientBoostedTrees ) requieren que estas las pasemos indexadas.
- Los algoritmos que no soportan variables categóricas ( LinearRegression, LogisticRegression ) requieren el uso de OneHotEncoder, tal como veremos.

Es importante recordar que, al emplear un modelo entrenado de *machine learning* para predecir ejemplos nuevos, primero hay que codificar sus variables categóricas, siguiendo exactamente la misma codificación que se utilizó con los datos de entrenamiento con los que se entrenó el modelo. Por eso, cobra sentido la estructura de *pipeline*, que comentaremos más adelante.

```
from pyspark.ml.feature import StringIndexer
df = spark.createDataFrame(
    [(0,"a"), (1,"b"), (2,"c"), (3,"a"), (4,"a"), (5,"c")], ["id",
"category"])

+---+-----+-----+
| id | category | categoryIndex |
+---+-----+-----+
| 0  | a        | 0.0          |
| 1  | b        | 2.0          |
| 2  | c        | 1.0          |

indexer = StringIndexer(inputCol =
    "category", outputCol = "categoryIndex")
indexed = indexer.fit(df).transform(df)
indexed.show()

# Guardo el transformer ajustado (indexerModel) para usarlo después:
indexerModel = indexer.fit(df)
indexed = indexerModel.transform(df)

# ... resto del código de nuestro programa. Ahora cargamos nuevos
# datos y los codificamos siguiendo exactamente la misma codificación:
indexedNuevo = indexerModel.transform(datosNuevosDF)
```

- ▶ **OneHotEncoderEstimator**: recibe un conjunto de columnas y convierte cada una (de manera independiente) a un conjunto de variables *dummy* con codificación *one-hot*. Cada variable (con n categorías posibles) da lugar a n variables (condensadas en una sola columna de tipo vector), donde, para cada ejemplo, solo una de las n variables tiene valor 1 y el resto son 0. Esto indica cuál es el valor de la categoría presente en ese ejemplo.
- Spark siempre asume que los valores provienen de una indexación previa con StringIndexer: obligatoriamente, la columna de entrada debe estar constituida por números reales con la parte decimal a 0.

```
from pyspark.ml.feature import OneHotEncoderEstimator
df = spark.createDataFrame([
    (0.0, 1.0, 2.0),
    (1.0, 0.0, 3.0), # Spark asume que la 3a col tiene 5 categorías
    (2.0, 1.0, 2.0), # porque el máximo valor que aparece es 4.0;
    (0.0, 2.0, 1.0), # también que vienen de una indexación previa
    (0.0, 1.0, 4.0), # con StringIndexer y, por tanto, empiezan en 0.0
    (2.0, 0.0, 4.0)
], ["categoryIndex1", "categoryIndex2", "categoryIndex3"])

encoder = OneHotEncoderEstimator(
    inputCols = ["categoryIndex1", "categoryIndex2", "categoryIndex3"],
    outputCols = ["categoryVec1", "categoryVec2", "categoryVec3"]
)
model = encoder.fit(df)
encoded = model.transform(df)

# La siguiente línea se utiliza porque vamos a convertir vectores
# sparse a dense, ya que el show() de sparse se ve peor en pantalla

from pyspark.ml.linalg import Vectors, VectorUDT
from pyspark.sql import functions as F

toDenseUDF = F.udf(lambda r: Vectors.dense(r), VectorUDT())

encoded.withColumn("categoryVec1", toDenseUDF("categoryVec1"))\
    .withColumn("categoryVec2", toDenseUDF("categoryVec2"))\
    .withColumn("categoryVec3", toDenseUDF("categoryVec3"))\
```

```
.show()
```

	categoryIndex1	categoryIndex2	categoryIndex3	categoryVec1	categoryVec2	categoryVec3
	0.0	1.0	2.0	[1.0,0.0]	[0.0,1.0]   [0.0,0.0,1.0,0.0]	
	1.0	0.0	3.0	[0.0,1.0]	[1.0,0.0]   [0.0,0.0,0.0,1.0]	
	2.0	1.0	2.0	[0.0,0.0]	[0.0,1.0]   [0.0,0.0,1.0,0.0]	
	0.0	2.0	1.0	[1.0,0.0]	[0.0,0.0]   [0.0,1.0,0.0,0.0]	
	0.0	1.0	4.0	[1.0,0.0]	[0.0,1.0]   [0.0,0.0,0.0,0.0]	
	2.0	0.0	4.0	[0.0,0.0]	[1.0,0.0]   [0.0,0.0,0.0,0.0]	

- ▶ **Cualquier modelo de predicción** (*machine learning*): todos los modelos heredan de **Estimator** y el método `fit(df)` lanza el aprendizaje. Los *estimators* suelen tener muchos parámetros configurables antes de *fit* (en algunos *transformers*, también hay parámetros configurables, pero suelen ser menos).

En el caso de los algoritmos de *machine learning*, los parámetros que se pueden ajustar antes de entrenar el modelo (aparte de los nombres de columnas necesarios) se denominan *hiperparámetros* y afectan a la manera en la que se desarrolla dicho entrenamiento. Por ejemplo, el hiperparámetro que controla la fuerza de la regularización (`lambd`), el número de iteraciones máximo del algoritmo de descenso en gradiente que se aplicará para aprender los parámetros del modelo o el número de árboles que se ajustarán en un algoritmo RandomForest.

## Pipelines en Spark ML

Es frecuente en *machine learning* extraer características de datos en crudo (raw) y prepararlas antes de llamar a un algoritmo de aprendizaje. Sin embargo, puede ser difícil tener control de todos los pasos de preprocesamiento que hemos llevado a cabo al entrenar, para luego replicarlos de manera exacta en otros conjuntos de datos o en el momento de hacer predicciones para nuevos datos con el modelo entrenado. Por ejemplo, a la hora de procesar un documento, hemos de llevar a cabo los siguientes pasos:

- ▶ División en palabras.

- ▶ Procesamiento de palabras para obtener un vector de características numéricas.
- ▶ Preparación de esas características para el formato que requiere el algoritmo elegido en Spark.
- ▶ Finalmente, entrenamiento de un modelo.

Spark nos proporciona un mecanismo para esto, denominado ***pipeline***.

*Pipeline* de SparkML: secuencia de etapas (*estimator* o *transformer*) que se ejecutan en un cierto orden para ir transformando un DataFrame.

En un *pipeline* de Spark, la salida de una etapa es entrada para alguna de las etapas posteriores (no necesariamente la inmediatamente siguiente).

Un *pipeline* es un *estimator*. El método `fit(df)` de un *pipeline* recorre cada etapa: **llama a `transform()` si la etapa es un *transformer* o a `fit(df)` y luego a `transform(df)` si es un *estimator***, pasando siempre el DataFrame *df* tal como esté en ese punto (con las columnas originales más las que le hayan añadido las etapas previas). Es habitual (pero no obligatorio) que la última etapa del *pipeline* sea un algoritmo de *machine learning*, aunque podría haber varios a lo largo de un *pipeline*. Es importante recordar que un mismo objeto (sea un estimador o un transformador) no puede ser añadido como etapa a dos *pipelines* diferentes.

Veamos un ejemplo. La siguiente figura muestra un *pipeline* antes de llamar a *fit*:

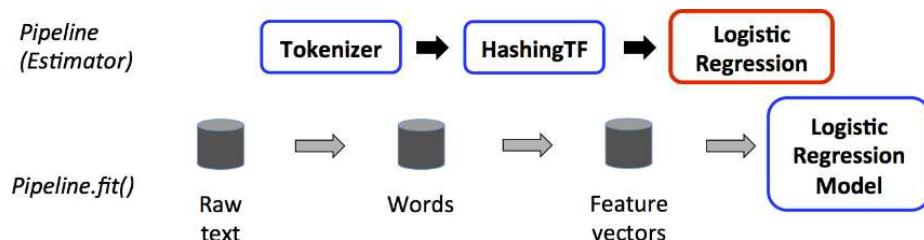


Figura 4. *Pipeline* sin ajustar (arriba) y procesamiento que ocurre al llamar a *fit* (abajo). Fuente: Apache Spark 2.4.5.

Las etapas en azul son transformadores, mientras que las rojas son estimadores. A continuación, vemos el objeto `PipelineModel` (*pipeline* ajustado), donde las etapas que eran estimadores han pasado a ser transformadores. Si ejecutamos el método `transform()` sobre un `PipelineModel`, este irá llamando a `transform` para cada etapa y el DF de la etapa previa devuelto por `transform` pasará como argumento.

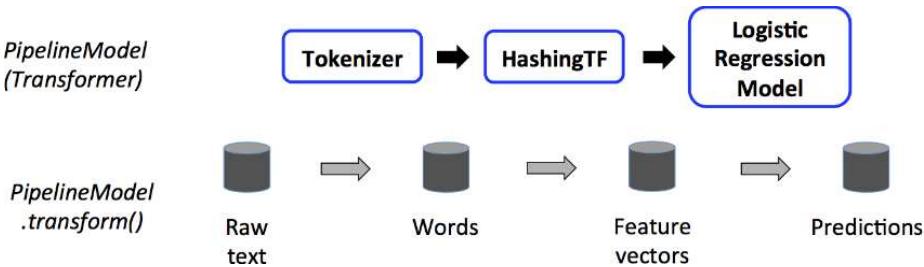


Figura 5. *Pipeline* ajustado (PipelineModel, arriba) y procesamiento que ocurre al llamar a *transform* (abajo). Fuente: Apache Spark 2.4.5.

Lo habitual es llamar a *fit* sobre el objeto *pipeline* una sola vez con los datos de entrenamiento. Esto devuelve un objeto PipelineModel (modelo ajustado), que es un transformador y sobre el que podemos llamar a *transform* tantas veces como queramos, sobre conjuntos de datos nuevos (nunca vistos por el modelo, pero que contengan todas las columnas que esperan cada una las etapas), para realizar predicciones.

El siguiente ejemplo lee un conjunto de datos sobre vuelos y tiempo que han llegado retrasados en minutos, los separa en entrenamiento y test, y actúa sobre los datos de entrenamiento: primero indexa las variables categóricas y las fusiona en una única columna de tipo vector; después estandariza cada variable por separado, binariza la columna *target* (para convertir el retraso en minutos en una variable binaria: retraso sí o no) y ajusta un modelo de regresión logística para predecir si un vuelo tendrá o no retraso. Todas las etapas se añaden a un *pipeline* y se efectúa el procesamiento en el momento de llamar a *fit()* sobre los datos de entrenamiento. Una vez que tenemos el *pipeline* entrenado, lo aplicamos para predecir los datos de los test, los cuales seguirán exactamente las mismas etapas.

```

from pyspark.ml.feature import StringIndexer, VectorAssembler,
Binarizer, VectorSlicer, StandardScaler
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
  
```

```

trainTest = spark.read.parquet("flights.parquet")\
    .randomSplit([0.8, 0.2], 12345)
trainingData = trainTest[0] # Dividimos los datos en train y test:
testingData = trainTest[1] # 80 % para entrenar y 20 % para testear.

monthIndexer = StringIndexer().setInputCol("Month")\
    .setOutputCol("MonthCat")
dayofMonthIndexer = StringIndexer().setInputCol("DayofMonth")\
    .setOutputCol("DayofMonthCat")
dayOfWeekIndexer = StringIndexer().setInputCol("DayOfWeek")\
    .setOutputCol("DayOfWeekCat")
uniqueCarrierIndexer = StringIndexer().setInputCol("UniqueCarrier")\
    .setOutputCol("UniqueCarrierCat")
originIndexer = StringIndexer().setInputCol("Origin")\
    .setOutputCol("OriginCat")
assembler = VectorAssembler()\
    .setInputCols([
        "MonthCat", "DayofMonthCat", "DayOfWeekCat",
        "UniqueCarrierCat", "OriginCat", "DepTime", "CRSDepTime",
        "ArrTime", "CRSArrTime", "ActualElapsedTime", "CRSElapsedTime",
        "AirTime", "DepDelay", "Distance"])
    .setOutputCol("vetorizedFeatures")

# Normalizamos cada variable para tener media 0 y desviación típica 1:
scaler = StandardScaler().setInputCol("vectorizedFeatures")\
    .setOutputCol("features")\
    .setWithStd(True).setWithMean(True)
binarizer = Binarizer().setInputCol("ArrDelay")\
    .setOutputCol("binaryLabel")\
    .setThreshold(15.0)
# Algoritmo de machine learning (Estimator) para clasificación:
lr = LogisticRegression().setMaxIter(10)\
    .setRegParam(0.3)\
    .setElasticNetParam(0.8)\
    .setLabelCol("binaryLabel")\
    .setFeaturesCol("features")

lrPipeline = Pipeline().setStages([
    monthIndexer, dayofMonthIndexer, dayOfWeekIndexer,
    uniqueCarrierIndexer, originIndexer, assembler, scaler,
    binarizer, lr])
pipelineModel = lrPipeline.fit(trainingData) # ajustar modelos
lrPredictions = pipelineModel.transform(testingData) # predecir
lrPredictions.select("prediction", "binaryLabel", "features").show(20)

```



## 5.3. Spark Structured Streaming

En esta sección, veremos una breve introducción a Structured Streaming, el módulo de Spark que ha absorbido al obsoleto Spark Streaming, el cual usaba una estructura de datos llamada DStream, basada en RDD. Actualmente, Spark Structured Streaming utiliza los ***streaming DataFrames***, que conceptualmente son iguales que un DataFrame, pero a los cuales **se les van añadiendo filas automáticamente, en tiempo real, según van llegando**. En realidad, no se implementa de esta manera, pero la analogía es válida para razonar sobre *streaming* DataFrames. Esto permite **disminuir la latencia** con respecto a un proceso *batch* que se ejecute periódicamente, ya que es capaz de hacer el cálculo incremental automáticamente en lugar de recalcular siempre todo el resultado partiendo de 0.

El procesamiento de flujos de datos (*stream processing*) consiste en incorporar continuamente nuevos datos para actualizar en tiempo real un resultado. Es decir, el cálculo se realiza agregando de alguna forma los datos nuevos a los ya existentes. Esta agregación puede incluir descartar en ciertos momentos los datos más antiguos para considerar solo los recibidos en una ventana temporal reciente. Podemos verlo recálculo continuo del resultado, en oposición a lo que ocurre en el procesamiento *batch*, en el que el cálculo se lleva a cabo una sola vez.

Structured Streaming se esfuerza por mantener una API idéntica a los DataFrames. De hecho, el mismo código que calcula la salida para un DataFrame convencional (estático) debería funcionar para un *streaming* DataFrame. Los conceptos de transformación y acción se mantienen, con una salvedad: la única acción disponible en Structured Streaming es la de comenzar un flujo (`start()`), que iniciará el cálculo y lo ejecutará indefinidamente, actualizando resultados periódicamente.

## Fuentes de datos de entradas y salidas permitidas

Spark Structured Streaming permite leer la entrada como flujo de datos desde Kafka (lo veremos en el tema siguiente); desde un sistema de ficheros como HDFS o Amazon S3, en el que una fuente externa va añadiendo ficheros nuevos a un directorio en tiempo real, y desde una fuente *socket* usable solo para desarrollar y testear. La lectura se efectúa con el método `readStream` aplicado al objeto `SparkSession`: `spark.readStream`. Es importante recordar que, en Structured Streaming, **hay que activar explícitamente la opción de inferencia de esquema o bien especificar siempre el esquema del fichero de entrada**, independientemente del formato. Incluso si es un fichero Parquet, que ya contiene dentro el esquema, es necesario especificar este como argumento si no hemos configurado la propiedad `spark.sql.streaming.schemaInference` a `true` en las opciones de Spark. Si ya se dispone de una versión inicial de los datos guardada, se puede realizar una lectura previa a un `DataFrame` convencional (estático), obtener el esquema que se ha leído y usar dicho esquema para el *streaming* `DataFrame`.

La salida puede asimismo escribirse en Kafka, en ficheros y en otras salidas también restringidas para testeo y depuración como, por ejemplo, una salida a «memoria». El **modo de salida** es relevante en estos casos: ¿queremos solamente añadir información con la salida actualizada o reemplazar completamente el fichero de salida generado en cada actualización? Existen tres modos: **añadir**, **actualizar** y **reemplazo completo**.

Veamos un ejemplo de código sencillo. Supongamos que tenemos un directorio de HDFS donde otro proceso externo va creando ficheros en tiempo real, todos con la misma estructura. Cada nuevo fichero incluye información de retrasos sobre un grupo de vuelos que ha aterrizado en diversos aeropuertos recientemente. Asumimos un *dataset* que incluye información sobre los retrasos de vuelos, similar al de ejemplos anteriores. Queremos ver el retraso medio que sufren los vuelos en cada aeropuerto. El código para calcularlo será el mismo que si tuviésemos un

DataFrame estático, pero Spark Structured Streaming irá actualizando automáticamente el fichero de salida agregado en tiempo real.

```
# flights.parquet es una carpeta de archivos, todos con mismo esquema:  
staticDF = spark.read.parquet("/path/to/flights/folder")  
schema = staticDF.schema  
# 1 para que se lea solamente un fichero de la carpeta en cada lectura:  
streamingDF = spark.readStream.schema(schema)\  
    .option("maxFilesPerTrigger", 1)  
\  
    .parquet("/path/to/flights.parquet")  
  
# Usamos la API estructurada como haríamos con un DF convencional:  
resultDF = streamingDF.where("delay > 15").groupBy("airport").count()  
  
# Invocamos la única acción disponible. Nótese que writeStream no es  
query = resultDF.writeStream\ # acción, sino que la acción es start()  
    .queryName("retrasosPorAeropuerto")\ # ;un nombre único!  
    .format("memory")\ # salida a memoria (para pruebas solo)  
    .outputMode("complete")\ # reemplazo completo  
    .start()      # esto desencadena el cálculo en segundo plano  
  
# IMPRESCINDIBLE para evitar que el driver finalice sin esperarnos:  
query.awaitTermination()
```

En el código anterior, Spark procesará un fichero, hará la agregación (conteo) y volcará el resultado a la salida indicada, que, en el caso anterior, es memoria. Inmediatamente después de terminar de procesar un fichero, volverá a leer de la carpeta otro distinto. Se irán leyendo de uno en uno los ficheros de la carpeta, debido a que hemos configurado `maxFilesPerTrigger` a 1. Asumimos que existe un proceso externo que está añadiendo en tiempo real ficheros nuevos a esa carpeta y, por este motivo, Spark los va procesando también en tiempo real.

## 5.4. Referencias bibliográficas

Apache Spark 2.4.5. (s. f.). *Machine Learning Library (MLlib) Guide*.

Chambers, B. y Zaharia, M. (2018). *Spark: the definitive guide*. O'Reilly.

## Documentación oficial de Apache Spark

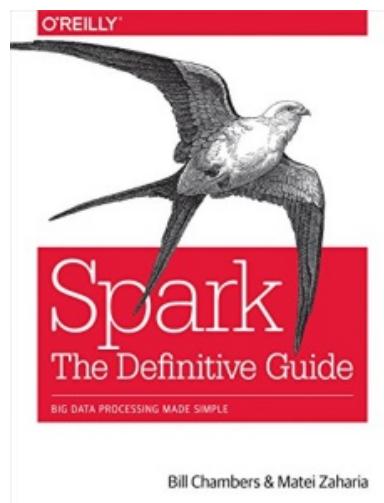
Apache Spark. Página web oficial: <https://spark.apache.org/docs/latest/>

Documentación *online*, detallada y de muy buena calidad sobre el sistema de computación Apache Spark.

## Spark: the definitive guide

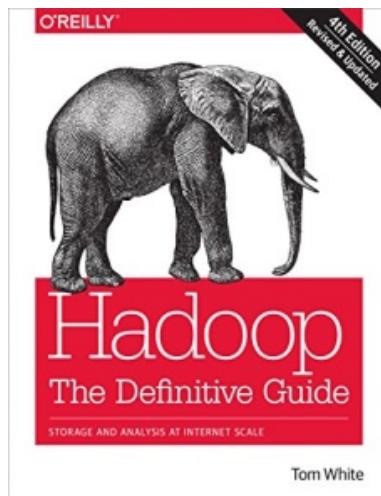
Chambers, B. y Zaharia, M. (2018). *Spark: the definitive guide*. O'Reilly.

Guía detallada de Spark, en su versión más actualizada. Contiene numerosos ejemplos y muestra exhaustivamente todas las capacidades de Spark. Son relevantes los capítulos del 20 al 22, el 24 y el 25.



## Hadoop: the definitive guide

White, T. (2015). *Hadoop: the definitive guide* (4. a edición). O'Reilly.



El capítulo 17 al completo está dedicado a Apache Hive.

1. ¿Qué diferencia Spark MLlib de Spark ML?

  - A. Spark MLlib ofrece interfaz para DataFrames en todos sus componentes, mientras que Spark ML sigue utilizando RDD y ha quedado obsoleta.
  - B. Spark MLlib no permite cachear los resultados de los modelos, mientras que Spark ML sí.
  - C. Spark MLlib es más rápida entrenando modelos que Spark ML.
  - D. Ninguna de las respuestas anteriores es correcta.
2. ¿Qué tipo de componentes ofrece Spark ML?

  - A. Estimadores y transformadores para ingeniería de variables y para normalizar datos.
  - B. Estimadores y transformadores para preparar los datos para el formato requerido por los algoritmos de aprendizaje automático de Spark.
  - C. Solo *pipelines* que no dan acceso a los estimadores internos.
  - D. Las respuestas A y B anteriores son correctas.
3. ¿Cuál es el método principal de un *estimator* de Spark ML?

  - A. El método *fit*.
  - B. El método *transform*.
  - C. El método *estimate*.
  - D. El método *describe*.
4. ¿A qué interfaz pertenecen los algoritmos de *machine learning* de Spark cuando aún no han sido entrenados?

  - A. Transformer.
  - B. Estimator.
  - C. Pipeline.
  - D. DataFrame.

5. ¿A qué interfaz pertenecen los modelos de Spark ML cuando ya han sido entrenados con datos?
  - A. Transformer.
  - B. Estimator.
  - C. Pipeline.
  - D. DataFrame.
  
6. ¿Qué ocurre si creamos un StringIndexer para codificar las etiquetas de una variable en el *dataset* de entrenamiento y después creamos otro StringIndexer para codificar los datos de test en el momento de elaborar predicciones?
  - A. Obtendremos la misma codificación en los dos.
  - B. Da un error, porque un mismo StringIndexer no puede añadirse a dos pipelines.
  - C. Podríamos obtener codificaciones distintas de la misma etiqueta en los datos de entrenamiento y en los de test, lo que falsearía los resultados de las predicciones.
  - D. Ninguna de las respuestas anteriores es correcta.
  
7. ¿Cuál es la estructura principal que maneja Spark Structured Streaming?
  - A. DStreams.
  - B. DStreams DataFrames.
  - C. *Streaming* DataFrames.
  - D. *Streaming* RDD.

- 8.** Spark Streaming permite leer flujos de datos:

  - A. Solo desde tecnologías de ingestión de datos como Apache Kafka.
  - B. Desde cualquier fuente de datos, siempre que contenga un esquema, como, por ejemplo, una base de datos.
  - C. Desde fuentes como Apache Kafka y HDFS, si activamos la inferencia de esquema.
  - D. Las respuestas A, B y C son incorrectas.
- 9.** En Spark Streaming, una vez se ejecuta la acción *start*:

  - A. El *driver* espera automáticamente a que concluya la recepción de flujo para finalizar su ejecución.
  - B. Hay que ejecutar un método para indicar al *driver* que no finalice automáticamente y que espere a que concluya la recepción del flujo.
  - C. Un flujo de datos no tiene fin y, por tanto, el *driver* nunca puede finalizar.
  - D. Ninguna de las opciones anteriores es correcta.
- 10.** ¿Qué acciones pueden realizarse en Spark Structured Streaming?

  - A. `take`.
  - B. `show`.
  - C. `start`.
  - D. `collect`.

Ingeniería para el Procesado Masivo de Datos

---

## Tema 6. Apache Kafka

# Índice

[Esquema](#)

[Ideas clave](#)

- [6.1. Introducción y objetivos](#)
- [6.2. Mensajería publicación/suscripción](#)
- [6.3. Introducción a Apache Kafka](#)
- [6.4. Casos de uso típicos de Kafka](#)
- [6.5. Conceptos fundamentales](#)
- [6.6. Implementación de productores Kafka](#)
- [6.7. Implementación de consumidores Kafka](#)
- [6.8. Referencias bibliográficas](#)

[A fondo](#)

[Kafka: the definitive guide](#)

[Kafka. Documentación oficial](#)

[Test](#)

# Esquema

PARADIGMA MENSAJERIA PUB/SUB	APACHE KAFKA			
	DEFINICIÓN	FUNCIONAMIENTO	PRODUCTOR	CONSUMIDOR
Múltiples productores (P) de información de distinto tipo (T)	Sistema de mensajería pub/sub	Clúster de Kafka	Kafka proporciona API Java para implementar productores y consumidores	
Múltiples consumidores (C) de información. Cada uno suscrito a un tipo de información (T)	Proyecto código libre	Conjunto de Brokeres (servidores Kafka)	Kafka proporciona protocolo bajo nivel; otros lenguajes proporcionan API basado en el protocolo [Python, C++, ...]	
<p><b>Distribuido → Escalable</b></p> <ul style="list-style-type: none"> <li>P_1 (T1, T2)</li> <li>P_2 (T3)</li> <li>P_3 (T1)</li> </ul>		<b>Replicado → Robusto</b> frente a fallos	<b>Se dividen en</b> Particiones, que se replican en los diferentes broteros	Envía mensajes a un topic mediante objeto KafkaProducer <ul style="list-style-type: none"> <li>- Indica mensaje con objeto ProducerRecord</li> <li>- Elige serializador tipos comprobados.</li> <li>- Recomendado Auto</li> <li>- Determina partición</li> <li>- Envía pue de ser fire-and-forget, consumeRecords (mensajes); indica máximo o sincrono o asíncrono</li> </ul>
Bus común de información (mensajes)		Múltiples productores	<b>Partición líder:</b> Es la que recibe los mensajes de los productores y los envía a los consumidores	<ul style="list-style-type: none"> <li>- Consumidor lee de una/varias particiones</li> <li>- Particiones asignadas según número de consumidores en grupo de consumidores</li> <li>- Lee conjuntos de datos</li> </ul>
<ul style="list-style-type: none"> <li>C_A (T1,T3)</li> <li>C_B (T2)</li> <li>C_C (T1)</li> </ul>		Múltiples consumidores	<b>Retención mensajes en disco</b>	<b>Particiones followers:</b> Se mantienen sincronizadas con la partición líder
			<b>Alto rendimiento →</b> transporte rápido grandes volúmenes mensajes	

## 6.1. Introducción y objetivos

En este tema, estudiaremos una herramienta más del ecosistema *big data* en la actualidad: Apache Kafka. Apache Kafka es una tecnología de *software libre* y está auspiciada por la Apache Software Foundation (ASF). Se utiliza sobre todo en casos de uso donde hay que procesar flujos de datos en tiempo real (*streaming*) y ha adquirido un papel fundamental en innumerables empresas hoy en día.

Los objetivos que persigue este tema son:

- ▶ Entender el paradigma de mensajería publicación/suscripción.
- ▶ Entender el concepto de bus de mensajes distribuido y sus casos de uso.
- ▶ Conocer los conceptos fundamentales y el funcionamiento interno de Kafka.

## 6.2. Mensajería publicación/suscripción

Todas las aplicaciones crean datos, ya sean mensajes de log, métricas, actividad de los usuarios, notificaciones, entre otros muchos. Estos informan de diferentes aspectos de la aplicación y ayudan a tomar decisiones y a llevar a cabo ciertas acciones, muchas veces relacionadas con otras partes de la aplicación distintas de aquellas donde fueron generados los datos en sí. Por tanto, a fin de aprovechar estos datos generados para beneficiar y optimizar otros procesos, generalmente es necesario que lleguen a aquellos sistemas que los puedan analizar y usar convenientemente. Por ejemplo, Amazon usa información generada durante la navegación web por su plataforma de compra (en qué productos ha hecho clic el usuario, cuáles ha acabado comprando, etc.) para ofrecer recomendaciones a ese usuario durante su navegación. Es decir, utiliza la información recolectada durante la navegación del usuario para generar en tiempo real recomendaciones ajustadas que lo inviten a comprar más productos. De este modo, mejora el funcionamiento de la plataforma de compra (desde un punto de vista de negocio).

Sin embargo, para que esto sea posible, la información de navegación almacenada debe llegar hasta los sistemas que se encargan del procesado necesario para generar las recomendaciones. Y se requiere que este transporte de información se lleve a cabo a la escala que demanda el volumen de usuarios que navega a la vez en la web de Amazon. Este ejemplo pone de manifiesto la necesidad de ampliar las tecnologías *big data* clásicas de almacenamiento y cómputo con otras capaces de mover datos de unos sistemas a otros para su óptimo aprovechamiento.

Entre los diferentes patrones que existen para este fin, la mensajería basada en el patrón publicación/suscripción (en inglés, *publish/subscribe* o *pub/sub*) es una de las más utilizadas actualmente en diferentes plataformas. Este patrón de transporte de datos se caracteriza por tener emisores (publicadores de información o *publishers*)

de datos (mensajes) que no están dirigidos a ningún receptor en concreto (contrastá, por ejemplo, con el servicio de correo electrónico, donde el emisor envía un mensaje a un receptor o conjunto de receptores específicos). En lugar de ello, el emisor clasifica el mensaje que envía bajo cierta clase (*topic*) y los receptores interesados (suscriptores o *subscribers*) se suscriben a la clase de mensajes que les interesan para poder recibir todos los mensajes asociados a dicha clase.

¿De dónde surge la necesidad de este tipo de sistemas? Generalmente, las aplicaciones constan de diferentes subsistemas que necesitan comunicarse entre sí. En su forma más sencilla, esta comunicación se hace mediante una cola de mensajes o un canal de comunicación entre procesos. Por ejemplo, en una gran aplicación web de compras, podemos tener varios servidores *frontend* que muestran la página web y permiten la interacción de un gran volumen de usuarios con la plataforma de compra. Dichos servidores *frontend* pueden enviar diferentes métricas de la aplicación (en qué productos hace clic cada uno de los diferentes usuarios, cuánto tiempo permanecen viendo cada producto, qué producto incluyen en o eliminan de su lista de deseos, etc.) a un servidor encargado de procesar todos estos datos para generar métricas semanales, mensuales, por producto, etc., tal y como muestra la figura 1.

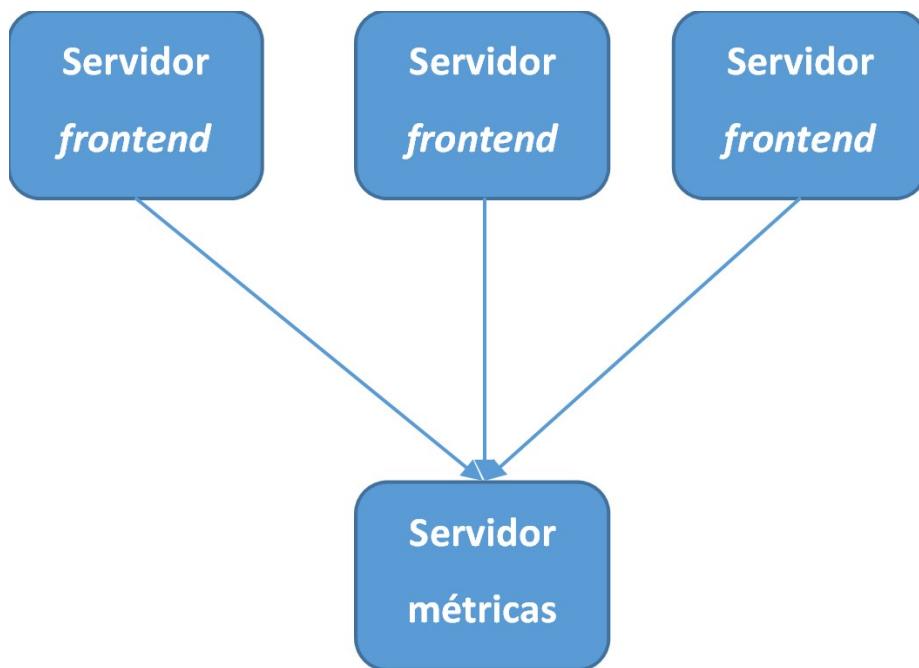


Figura 1. Sistema sencillo: varios servidores *frontend* (web) envían información sobre navegación de los usuarios al servidor de métricas.

Esta solución sería suficiente si no hubiera que conectar más que dos sistemas: el servidor *frontend* con el de procesamiento de métricas. Pero el problema aparece cuando se requiere conectar varios servidores (no solo el *frontend*, sino también el servidor de chat, otro con datos de las valoraciones de los productos, el que gestiona la cesta de la compra, etc.) con un sistema de métricas más complejo, que tiene un servidor de procesamiento y generación de métricas, otro de monitorización activa de métricas para generar notificaciones y alarmas, otro de visualización de métricas, otro de almacenamiento, etc. De repente, la información relacionada con métricas de la aplicación procedente de diferentes sistemas (productores de la información) necesita ser comunicada a otros tantos sistemas diferentes (consumidores de la información), que utilizan esa misma información, pero de diferentes formas (procesamiento, visualización, almacenamiento...). Como vemos en la figura 2, la complejidad aumenta considerablemente, ya que hay que tener en cuenta que cada flecha, que representa el envío de información de cada grupo de productores a cada

consumidor, significa una implementación concreta de dicha conexión con protocolos y formatos de mensaje específicos (y, en ocasiones, distintos).

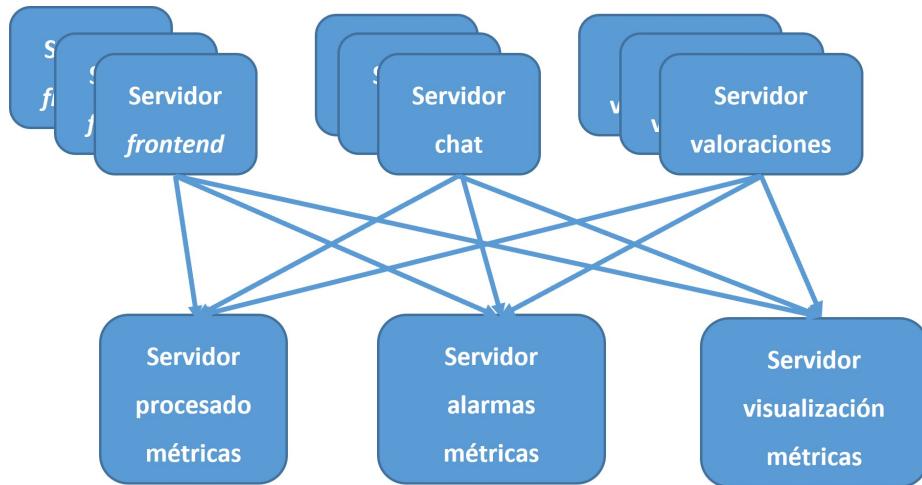


Figura 2. Sistema más complejo: muchos productores de mensajes de métricas envían su información a diferentes consumidores.

Para evitar este incremento exponencial de conexiones, ¿por qué no crear un único sistema donde publicar toda la información relacionada con las métricas, de manera que cualquier otro sistema pueda consultarla en dicho punto central? Así se reduce significativamente la complejidad. Como se observa en la figura 3, disminuye el número de conexiones y todas ellas se realizan con un elemento central; por tanto, son iguales.

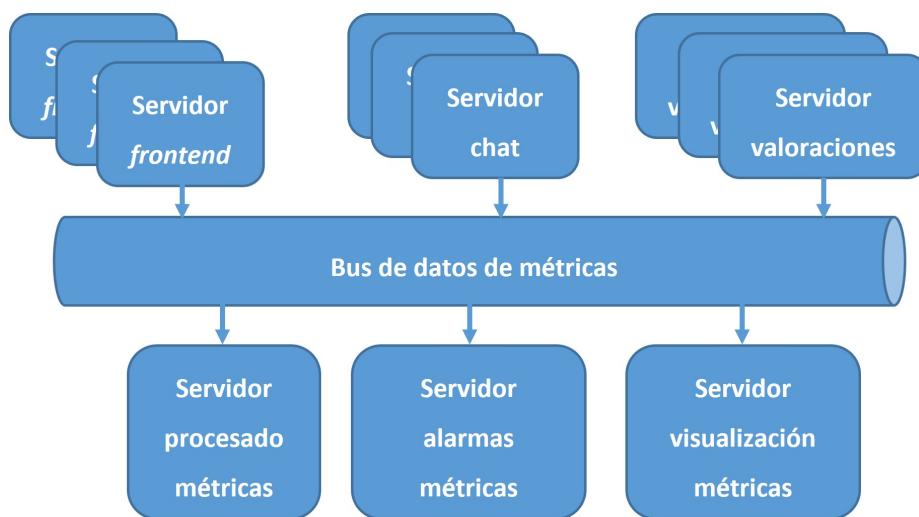


Figura 3. Uso de un bus de mensajes que centraliza la información producida y consumida.

Ahora bien, ¿qué ocurre si este mismo problema se extiende no solo a la información de métricas, sino también a los logs de los diferentes sistemas y a cualquier otro tipo de información interesante para la aplicación? Acabaríamos teniendo un sistema de publicación/consulta para las métricas, otro exactamente igual para los logs, y así sucesivamente, de forma que estaríamos replicando el mismo sistema para diferentes tipos de información. Esto supone un gasto de recursos de mantenimiento de distintos sistemas que, según se puede concluir si se analizan de cerca, hacen el mismo trabajo (independientemente del tipo de información contenida en los mensajes, el sistema de transporte es el mismo) y que es probable que haya que replicar en un futuro con nuevos tipos de información.

De aquí nace la idea de utilizar un único sistema de transporte de información centralizado (denominado a veces bus o canal de datos), que permita a diferentes sistemas publicar diversos tipos de información que posteriormente pueda ser consultada por otros tantos sistemas distintos, y que se pueda escalar según aumenten las necesidades de la aplicación (es decir, según se incremente el volumen de información producida y consumida). Esto es lo que se denomina patrón publicación/suscripción (*pub/sub*), el cual se muestra de forma esquemática en la

figura 4.

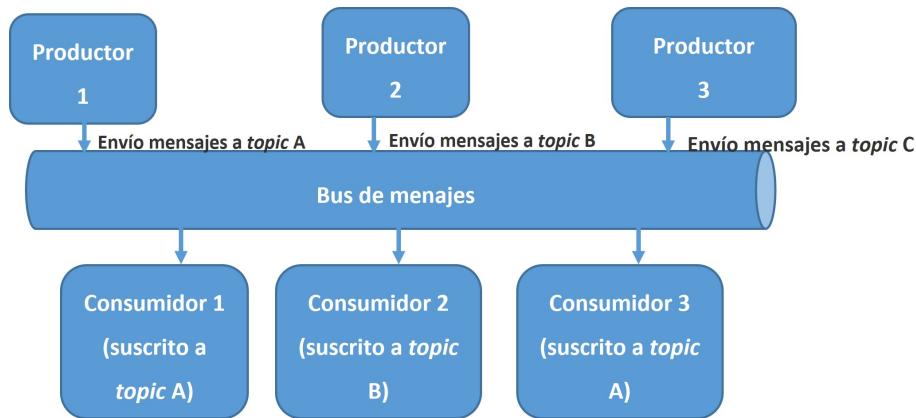


Figura 4. Sistema de mensajería publicación/suscripción.

## 6.3. Introducción a Apache Kafka

Apache Kafka es un sistema de mensajería que sigue el patrón publicación/suscripción (*pub/sub*) descrito en la sección previa. Proporciona un único bus (canal) de datos común, donde varias aplicaciones puedan escribir mensajes y el cual garantice que otras aplicaciones puedan recibirlos leyendo del bus, según los tipos de mensajes que les sean relevantes. Esto se formaliza en la siguiente definición:

Kafka es un bus de datos (también llamado cola de mensajes) distribuido y replicado, basado en el paradigma publicación/suscripción.

Existen varios conceptos en esta descripción que merecen ser examinados con mayor detalle:

- ▶ **Bus.** Un bus de datos es un canal donde las aplicaciones escriben (publican) mensajes, como si fuese un cable de datos de capacidad inmensa. Se llama también cola porque los mensajes se van insertando en un cierto orden y las aplicaciones los van consumiendo (leyendo) según esa misma disposición.
- ▶ **Mensaje.** Se llama también cola de mensajes porque la unidad de información es el **mensaje**, que sería el equivalente a un registro (fila) de una base de datos. Un mensaje es simplemente un *array* de bytes, sin significado para Kafka. Cada mensaje tiene asociado una **clave**, que es otro *array* de bytes sin significado. La clave se utiliza para indicar que el mensaje debe escribirse en una partición concreta del *topic* al que va destinado. Por eficiencia, los mensajes se escriben en Kafka en **lotes** (*batches*), para no generar demasiada sobrecarga en la red. Cuanto mayores son los lotes, más mensajes se pueden gestionar por unidad de tiempo, pero más tarda un mensaje en propagarse al resto de máquinas y aplicaciones consumidoras.

- ▶ **Distribuido y replicado.** Los mensajes se reciben y escriben en el disco local de las máquinas interconectadas; forman así un clúster de Kafka, que opera como un único bus de datos. Además, los mensajes enviados son replicados automáticamente en varias máquinas. Esto aporta robustez frente a fallos o caídas de máquinas, y también un aumento de rendimiento y escalabilidad cuando existan muchas aplicaciones escribiendo y leyendo del bus a la vez.
- ▶ **Publicación/suscripción.** Las aplicaciones que desean recibir mensajes de manera continua se suscriben a un *topic* concreto del bus, de manera que solo les llegarán los mensajes que hayan sido publicados (es decir, escritos por alguna aplicación) en ese *topic*, y ningún otro.
- ▶ **Topics y particiones.** Un *topic* sería el equivalente a una tabla de una base de datos o a un directorio de un sistema de archivos, e indica una agrupación de los mensajes tal que todos están estructurados del mismo modo y pueden interpretarse igual. Una aplicación que escriba o lea del bus debe determinar en qué *topic* desea hacerlo.

Dentro de cada *topic*, se definen particiones, que indican cómo se distribuyen físicamente los datos y que serían equivalentes a los bloques físicos en los que HDFS divide un archivo. Los mensajes se insertan y se leen por orden de llegada, dentro de una misma partición, pero no existe un orden entre particiones distintas de un *topic*.

El particionamiento permite la escalabilidad y la replicación de los datos, de un modo similar a como lo hacen los bloques de HDFS. Así, si aumenta la frecuencia de escrituras en uno o varios *topics*, podemos añadir más máquinas (brókeres) al clúster de Kafka y más particiones al *topic* para aprovecharlas. Al igual que ocurre en HDFS, es posible indicar la organización física (en armarios o *racks*) de las máquinas del clúster de Kafka para decidir cómo ubicar físicamente las réplicas de cada partición de un *topic*.

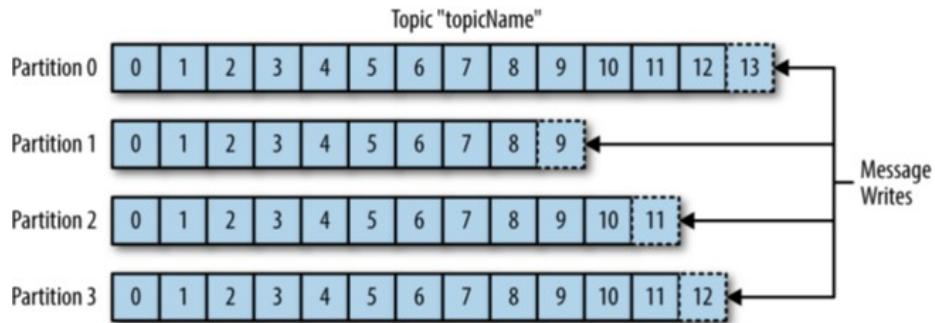


Figura 5. Un *topic* llamado *topicName* con cuatro particiones. Fuente: Narkhede, Shapira y Palino, 2017.

## 6.4. Casos de uso típicos de Kafka

Algunos de los casos de uso más comunes con Kafka son siguientes:

- ▶ **Tracking de actividad en una web:** fue el propósito original de Kafka en LinkedIn. El frontal web envía mensajes a Kafka con la interacción del usuario con la web (clics, modificación de contenido del perfil, etc.). Estos son consumidos por varios sistemas, que monitorizan la información y luego elaboran informes, entrena modelos de *machine learning* o personalizan la página o los resultados en tiempo real, durante la propia navegación.
- ▶ **Mensajería:** varias aplicaciones envían mensajes que después otra aplicación reúne y formatea para mandar por email informes o resúmenes más completos, o información de alertas en tiempo real vía SMS.
- ▶ **Métricas y logs:** múltiples aplicaciones pueden enviar a Kafka desde diversos puntos el mismo tipo de mensajes de *logs*, que luego son utilizados para elaborar informes o análisis en modo *batch*, o para monitorización y creación de alertas en tiempo real, con base en casuísticas más complejas que afectan a varios sistemas.
- ▶ **Historial de cambios de bases de datos:** para replicar los cambios de una base de datos en otros lugares o para mezclar los cambios desde varias bases de datos distintas.
- ▶ **Procesamiento en tiempo real de flujos de datos:** es el caso más típico. Se opera con datos según van llegando para realizar o actualizar agregaciones. Está incluido en los casos anteriores. Con frecuencia, se utilizan *frameworks* de procesamiento distribuido como Spark, Flink, etc.

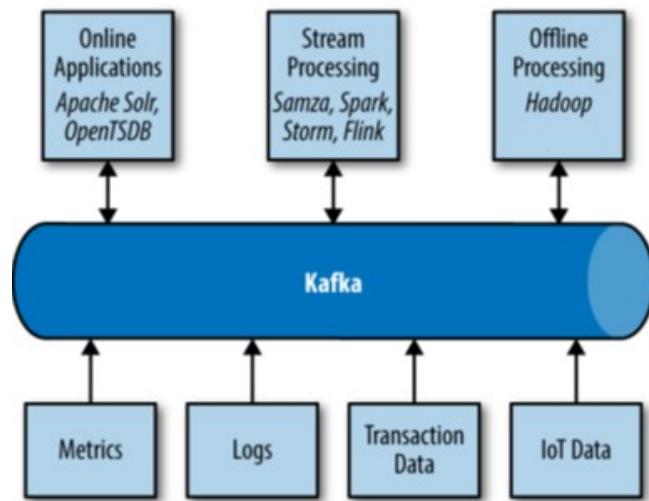


Figura 6. Kafka como bus de datos único, con múltiples productores y consumidores. Fuente: Narkhede, Shapira y Palino, 2017.

La figura 6 representa cómo una compañía podría usar Kafka. Varias aplicaciones distintas escriben en él (recuadros inferiores), es decir, cada una es un productor; mientras que otras aplicaciones leen de Kafka (recuadros superiores), de aquellos *topics* que les son relevantes, por lo que puede decirse que cada una es un consumidor. En ciertos casos, los consumidores son también productores, puesto que vuelven a escribir en Kafka el resultado de haber efectuado cierto procesamiento con el mensaje que habían leído.

## 6.5. Conceptos fundamentales

Kafka nació en 2010 como software interno de LinkedIn, para resolver un problema que no podían solventar con el software existente: un sistema de mensajería con múltiples consumidores al mismo tiempo, en tiempo real, por donde circulen datos de diverso tipo, y con alta disponibilidad, rendimiento y fácil escalabilidad. Se donó en 2011 a la Apache Software Foundation y se convirtió en proyecto top en 2012.

En 2014, varios empleados de LinkedIn fundaron Confluent, empresa que soporta el desarrollo actualmente y que incluye productos relacionados como Kafka Stream y Kafka Connect. Hoy lo utilizan intensivamente todas las grandes compañías: PayPal, Walmart, LinkedIn, Cisco, Netflix, Spotify, Uber, eBay, Amazon, etc.

Vamos a ver algunos detalles y conceptos relacionados con el funcionamiento de Apache Kafka.

### Clúster de Kafka

Mientras que utilizar un clúster de Kafka con un único bróker puede ser suficiente para pruebas de concepto y desarrollos locales, existen numerosos beneficios asociados a tener múltiples brókeres configurados como un clúster. El principal es la capacidad de escalar la carga entre los brókeres que conforman el clúster, así como el uso de replicación de información para evitar pérdidas de datos por fallos potenciales de alguno de los brókeres o durante sus tareas de mantenimiento.

Apache Kafka utiliza otra herramienta del ecosistema Hadoop, llamada **Zookeeper**, para almacenar metadatos sobre el clúster de Kafka (y, en versiones antiguas, también sobre los consumidores). En concreto, Zookeeper posee una lista de los brókeres que son parte de un clúster Kafka en todo momento. Cada bróker tiene un identificador único, con el que se registra en Zookeeper al iniciarse. De esta forma, Zookeeper notifica a un nuevo bróker que intenta unirse al clúster si trata de hacerlo

con un ID que ya está registrado.

El primer bróker que se registra en Zookeeper como parte del clúster de Kafka es designado como bróker controlador del clúster (*controller*). Este se encarga de seleccionar las particiones líderes (concepto que se expondrá más adelante) tan pronto detecta brókeres que se unen o se caen del clúster. Si este bróker controlador pierde la conexión con el clúster, el resto de nodos tratarán de ser controladores. Zookeeper mediará en la pugna por ser controlador y asignará este papel al bróker que primero haga la petición.

## Topics en Kafka y número de particiones

Los *topics* se crean desde la consola (no se gestionan ni desde los productores ni desde los consumidores) y existen varias opciones de configuración para ello. Una de ellas es el número de particiones, que se puede especificar durante la creación del *topic* o confiar en el parámetro *num.partitions* en la configuración del servidor (por defecto, configurada en 1). Es importante tener en cuenta que **el número de particiones de un *topic* únicamente se puede incrementar, nunca disminuir**, por lo que hay planificar de antemano cuántas particiones son óptimas y qué configuración existe en el servidor en caso de que queramos un número de particiones menor que el preestablecido.

Teniendo en cuenta que **el número de particiones va a ser en parte responsable del balanceo de la carga de mensajes en el clúster**, la pregunta que puede surgir en este punto es: ¿qué número de particiones es óptimo al crear un *topic*? En muchos casos, se indica un número de particiones igual o múltiplo del número de brókeres en el clúster para distribuir la carga de forma homogénea. Si se quiere hilar más fino y se tiene información más específica de los requisitos y capacidades del sistema, otros factores que tener en cuenta son la capacidad de envío/recepción de los productores/consumidores frente a la capacidad que se espera del *topic*. Por ejemplo, si se desea alcanzar una tasa de lectura de 1GB/s en un *topic* concreto y

cada consumidor tiene un máximo de procesado de 50MB/s, entonces se necesitarán, al menos, 20 particiones para dicho *topic*, de forma que pueda haber hasta 20 consumidores leyendo del *topic* a la vez (20 consumidores x 50MB/s = 1GB/s).

Otro aspecto que debe configurarse es la **política de retención**, es decir, determinar cuándo eliminar mensajes de un *topic*. Existen dos variantes de esta configuración: por tiempo o por tamaño. La estrategia más habitual es configurar cuánto tiempo se va a mantener cada mensaje grabado en disco y disponible para su consumo. Este parámetro se puede configurar en diferentes escalas (horas, minutos y hasta milisegundos), aunque 168 horas (una semana) es el valor por defecto. La estrategia alternativa es configurar el número total de bytes de mensajes que se mantendrán en disco; en este caso, se configura por partición (no por *topic*). Si se especifican ambos criterios (tiempo y tamaño de partición), los mensajes se eliminarán según se vayan cumpliendo cualquiera de las dos condiciones.

## Replicación de particiones, líder de partición y sincronización de réplicas

Como se comentaba previamente, tener un clúster de Kafka con varios brókeres proporciona la oportunidad de replicar particiones, una de las principales características de este sistema. La replicación es crítica porque garantiza la disponibilidad y durabilidad de los mensajes publicados, aun cuando alguno de los brókeres falle.

Los mensajes de Kafka están organizados en *topics*. Cada *topic* está a su vez dividido en diferentes particiones, cada una de las cuales está replicada en varios brókeres, lo que proporciona redundancia frente a fallos. La cantidad de réplicas o **factor de replicación** (concepto análogo a HDFS) es configurable mediante un parámetro, fijado en 3 por defecto. Existen dos tipos de réplicas de una partición:

- ▶ **La réplica líder.** Cada partición tiene una única réplica designada como líder. Todos los mensajes publicados y consumidos en una partición se harán en la réplica líder.

Es decir, el bróker que contenga la réplica líder de una partición será el encargado de recibir todos los mensajes publicados en dicha partición, así como de enviar todos los mensajes que los consumidores lean de dicha partición (figura 7).

- ▶ **Las réplicas *followers*.** Todas las réplicas de una partición que no son líderes se denominan *followers*. Los brókeres que contienen una réplica *follower* de una partición no escriben mensajes procedentes de productores en la réplica ni sirven mensajes a consumidores de dicha partición. Su única función respecto a la partición de la que contienen la réplica *follower* es replicar los mensajes que se escriben en la réplica líder (procedentes de peticiones de productores) y mantenerse actualizados. En caso de que el bróker con la réplica líder falle, el bróker controlador elegirá un nuevo bróker como líder de la partición de entre los *followers*.

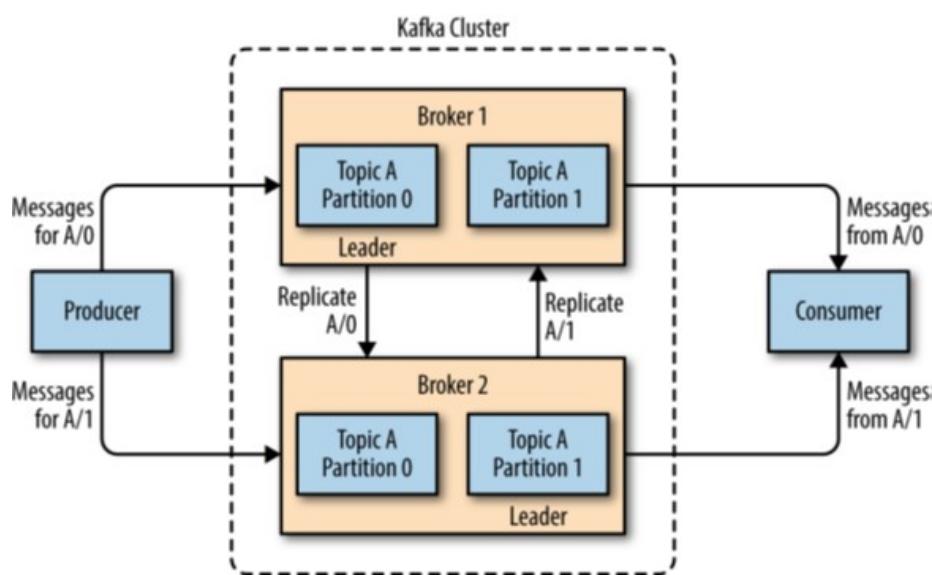


Figura 7. *Topic* con dos particiones replicadas. Cada bróker es líder de una partición. Fuente: Narkhede, Shapira y Palino, 2017.

Para mantenerse sincronizados, los brókeres con réplicas *followers* solicitan regularmente al bróker con la réplica líder los últimos mensajes que este haya recibido. Esas peticiones son un indicativo de cómo de retrasados van en la sincronización. Los retrasos pueden estar causados por la congestión de la red, por caídas ocasionales del bróker, etc. El líder es responsable de saber qué réplicas

*followers* están sincronizadas (*in-sync*) y cuáles no. Aquellas que no han pedido los últimos mensajes a la réplica líder desde hace más de 10 segundos se consideran réplicas no sincronizadas (*out-of-sync*) y dejan de ser valoradas como potenciales sucesoras de la partición líder en caso de que esta falle.

Por otro lado, es importante también decidir cómo repartir las réplicas entre los brókeres. En general, se persiguen tres objetivos:

- ▶ Repartir las réplicas de forma homogénea (por ejemplo, si hay 40 réplicas en total entre todos los *topics* y hay 5 brókeres, cada bróker tendrá 8 réplicas).
- ▶ Asegurar que, para cada partición, cada una de sus réplicas está en un bróker distinto.
- ▶ Si los brókeres saben en qué *rack* están instalados, asignar réplicas de cada partición en *racks* distintos, siempre y cuando sea posible.

## Procesamiento de las peticiones de escritura

En general, la función de un bróker es procesar peticiones enviadas desde productores, suscriptores, brókeres con réplicas *followers* de sus réplicas líder o desde el bróker controlador. Siempre son los clientes (productores, consumidores, o brókeres) los que inician las conexiones y envían las peticiones, y el bróker las procesa y responde a todas ellas. Concretamente, el bróker procesa y responde todas las peticiones recibidas de un mismo cliente en el mismo orden en que las ha recibido, de forma que se garantiza el funcionamiento de Kafka como una cola de mensajes.

Las dos peticiones más habituales son las peticiones de escritura o producción de mensajes (*produce request*) y las peticiones de consumición o lectura de mensajes (*fetch requests*), ambas enviadas al bróker con la réplica líder. Si un bróker con réplica *follower* recibe peticiones para dicha réplica, el cliente que ha realizado la petición recibirá un error. Dado que el responsable de saber a qué bróker enviar la petición es el cliente, existe una petición adicional, la petición de metadatos, que sirve para obtener información de quién es el bróker con la réplica líder de la partición requerida. Cuando un cliente realiza estas peticiones de metadatos, guarda la información recibida en caché para reducir el número de peticiones enviadas.

En cada productor, se configura, de manera independiente, el número de réplicas de una partición que han de recibir un mensaje para considerarlo como escrito con éxito. Que un **mensaje haya sido escrito con éxito** significa que puede ser leído por los consumidores. Hasta que un mensaje no llega a este estado, no puede ser leído por ningún consumidor, como veremos más adelante. El número de réplicas que tienen que recibir el mensaje para considerarlo escrito con éxito se configura en el parámetro ***nacks*** y puede tomar uno de tres valores:

- ▶ *nacks=0*, ni siquiera esperamos a que el líder de la partición lo haya escrito.
- ▶ *nacks=1*, basta con que el líder confirme que lo ha recibido y escrito.

- ▶ *nacks=all*, tienen que recibirla todas las réplicas que estén sincronizadas.

El bróker que recibe un mensaje para una de sus particiones líder lo escribe inmediatamente en disco. Si el valor de *nacks* que venía en la petición es 0 o 1, el bróker devuelve un mensaje de respuesta al productor como que se ha escrito con éxito. Si no, el líder, además, escribe el mensaje provisionalmente en un *buffer* llamado **purgatorio de mensajes**, hasta que recibe las peticiones de las réplicas que desean mantenerse actualizadas, les envía este mensaje y estas confirman su escritura. Es decir, el mensaje escrito en la réplica líder permanece en el purgatorio hasta que el bróker con dicha réplica líder ha recibido confirmación de que el mensaje ha sido escrito en todas las réplicas *followers*. Solo en ese momento el líder devolverá la respuesta de éxito al productor que solicitó la escritura.

## Procesamiento de las peticiones de lectura

Cuando un consumidor desea leer mensajes de un *topic*, envía una petición al bróker que posee la réplica líder de esa partición. Como comentábamos antes, para saber qué bróker tiene la réplica líder de la partición en la que desea escribir, enviará una petición de metadatos si no tiene dicha información en caché. El mensaje de petición de lectura de mensajes contiene varios datos:

- ▶ El *topic*, la partición y el *offset* del mensaje a partir del cual desea consumir (por ejemplo, «envíame los mensajes, empezando en el *offset* 25 de la partición 1 del *topic* “vuelos”»),
- ▶ El tamaño máximo del bloque de mensajes que desea que le sean devueltos. Esto garantiza que el bróker no enviará más datos de los que el cliente es capaz de almacenar en memoria.
- ▶ Opcionalmente, el tamaño mínimo de datos que desea recibir. En este caso, el bróker no le envía ningún mensaje hasta no tener acumulado, al menos, ese mínimo. Este parámetro suele ir acompañado de otro que indica un tiempo máximo de espera, transcurrido el cual el bróker devolverá los mensajes que tenga, aunque no

alcancen el tamaño mínimo indicado. Esto permite reducir procesado y uso de la red cuando los *topics* no manejan una gran cantidad de mensajes.

Si el *offset* especificado existe (no se ha borrado aún del disco), el bróker con la réplica líder de la partición envía directamente desde disco los mensajes solicitados. A pesar de necesitar lecturas y escrituras de disco, el acceso es rápido gracias a la estructura del fichero y a la existencia de índices. Además, Kafka utiliza un mecanismo denominado ***zero-copy***, es decir, envía datos directamente desde disco, sin pasar por un *buffer* en memoria principal. Esto contrasta con el funcionamiento de otros sistemas que utilizan cachés locales antes de enviar datos a los clientes, y permite obtener un mejor rendimiento.

Es importante tener en cuenta que un bróker (líder de una partición) solo devolverá mensajes que hayan sido escritos en todas las réplicas sincronizadas (*in-sync-replicas*). Si no, los mensajes no serán visibles, porque todavía no existirán oficialmente en Kafka y no podrán ser consumidos. Este comportamiento evita riesgos de tener comportamientos inconsistentes. Por ejemplo, si solo el bróker con la réplica líder tiene un mensaje que aún no ha sido enviado a ninguna otra réplica *follower*, podría ocurrir que un cliente leyera dicho mensaje y, acto seguido, el bróker líder fallara; así, cuando otro cliente fuera a leer el mensaje, este no estaría disponible, con lo que se crearía una situación de inconsistencia en relación con la información leída por los diferentes clientes. Al permitir la lectura de mensajes solo cuando están escritos tanto en la réplica líder como en todas las réplicas sincronizadas, se evitan este tipo de situaciones.

Cabe destacar que todos los mensajes enviados por un productor tienen que estar disponibles en todas las réplicas *in-sync* antes de poder ser leídos, independientemente del valor de *nacks* que tenga configurado el productor. Es decir, si un productor tiene configurado *nacks=1*, solo significa que Kafka le indicará que su mensaje ha sido escrito en la réplica líder, pero no está garantizado que este pueda

ser leído por los consumidores todavía, no hasta que no esté escrito también en todas las réplicas sincronizadas.

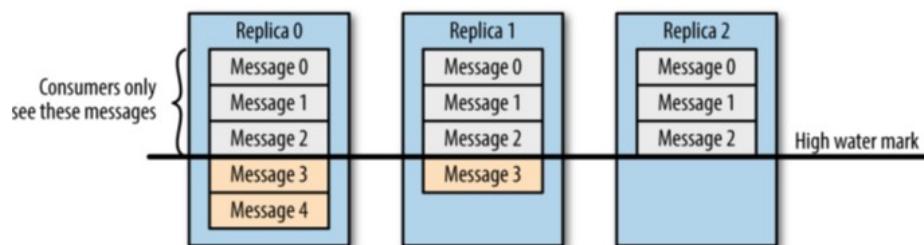
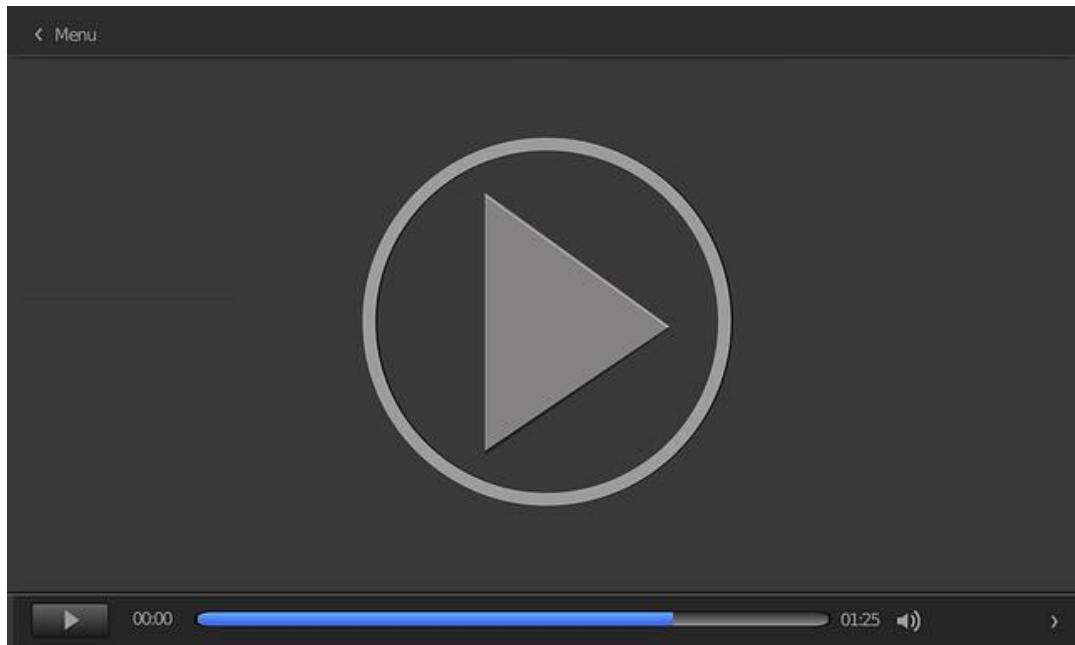


Figura 8. Los consumidores solo ven mensajes que existan en todas las réplicas sincronizadas. Fuente: Narkhede, Shapira y Palino, 2017.

Para finalizar este epígrafe y reforzar los conceptos estudiados, en el siguiente vídeo, vamos a mostrar el funcionamiento de Spark Structured Streaming con Kafka.



---

Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=35980c01-7068-4e8d-8809-aff800fba7eb>

---

## 6.6. Implementación de productores Kafka

Hasta ahora, hemos visto cómo funciona Kafka desde el punto de vista de un observador externo que inspecciona el funcionamiento global del sistema. Sin embargo, para que este funcione, se necesita tener cierto código en las aplicaciones productoras que permita escribir datos en Kafka, y cierto código en las aplicaciones consumidoras para la lectura de estos. A fin de implementar este código, Apache Kafka ofrece una API en lenguaje Java que proporciona la funcionalidad requerida por los productores (escritura de mensajes en *topics*) y consumidores (lectura de mensajes de *topics*). Además de esta API, define también un protocolo a bajo nivel, de manera que cualquier aplicación que lo implemente pueda enviar o recibir mensajes de Kafka. Aprovechando la definición de este protocolo, se han creado diferentes API en otros lenguajes, lo cual permite utilizar Kafka desde, por ejemplo, aplicaciones en Python, C++, etc. Dichas API no forman parte del proyecto original de Kafka y tampoco serán objeto del presente temario, por lo que nos centraremos en la API de Java.

Cuando decimos que un productor escribe un mensaje en un *topic* de Kafka, realmente lo que sucede es que un programa Java implementa la lógica necesaria para ello utilizando la API Java que Kafka proporciona. Dependiendo del caso de uso concreto que estemos resolviendo, deberemos considerar ciertos aspectos de la configuración:

- ▶ ¿Se puede tolerar pérdida de mensajes o son todos críticos? Ejemplo: transacción de tarjeta de crédito, donde no puede perderse ni duplicarse un registro, y que requiere respuesta inmediata de aprobación o denegación (**datos operacionales**).
- ▶ ¿Se puede tolerar que algún mensaje llegue repetido? Ejemplo: información sobre clics en un sitio web para aplicar analítica en modo *batch*, en lo que sería la base de datos **informacional** de la empresa. Para analizar tendencias de comportamiento,

no es relevante perder o tener repetido un mensaje.

- ▶ ¿Hay restricciones de tiempo real en la propagación de mensajes o de cantidad de mensajes por segundo que debe tolerar el bus?

Básicamente, la API Java de Kafka nos ofrece una clase `Java KafkaProducer`, con un método `send` que debemos invocar y que funciona como se muestra en la figura 9. Es necesario configurar ciertas propiedades al crear el objeto `KafkaProducer` sobre el que luego invocaremos el método `send` (para enviar el mensaje), tales como especificar un objeto `Serializer`, que determina cómo convertir el mensaje y la clave en un *array* de bytes que posteriormente puedan ser leídos e interpretados por los consumidores, y una lista de direcciones IP de los brókeres a los que conectarse. Después se ha de crear un objeto `ProducerRecord`, que representa el mensaje que vamos a enviar y que debe incluir, al menos, el *topic* al que queremos enviarlo y el contenido del mensaje. Además, puede especificarse una clave y una partición si va destinado a una concreta.

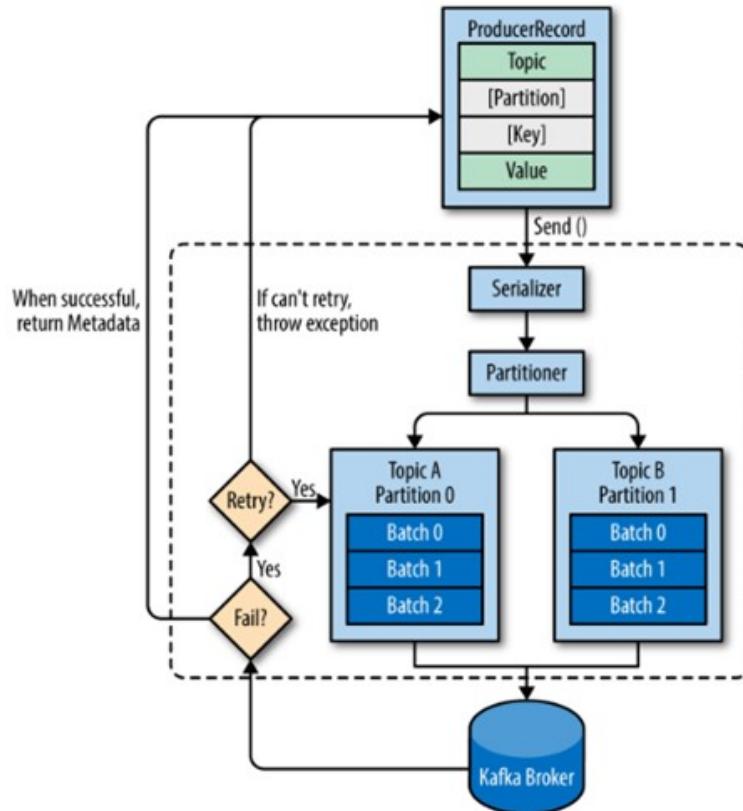


Figura 9. Proceso de escritura de mensajes en Kafka. Fuente: Narkhede, Shapira y Palino, 2017.

Lo primero que hace Kafka es serializar la clave y el valor, es decir, los convierte en **byte arrays** (secuencias binarias de bytes) que pueden ser transmitidos por la red. Si no hemos especificado una partición, hay un objeto, `Partitioner`, que decide a cuál irá, bien en función de la clave que hemos indicado, bien, si no la hay, siguiendo un esquema rotatorio para mantener similares los tamaños de todas las particiones. Si, por el contrario, hemos determinado una partición, irá directamente a ella. A continuación, Kafka añade el mensaje al bloque de mensajes que serán enviados al mismo *topic* y partición. Una hebra separada los va mandando periódicamente a los brókeres apropiados. Cuando el bróker recibe un bloque, envía una respuesta. Si se han escrito con éxito, la respuesta es un objeto `RecordMetadata`, pero, si es errónea, la función `send` puede intentar reenviar antes de devolver un error.

Existen tres maneras posibles de efectuar el envío:

- ▶ **Enviar y olvidar (*fire-and-forget*):** se envía y no se atiende a la posible respuesta.  
Los mecanismos de Kafka tratarán de que sea recibida y, en la mayoría de los casos, será así, pero podrían llegar a perderse mensajes.
- ▶ **Envío síncrono:** el proceso se bloquea a la espera de la respuesta a su envío. Esta será una excepción si no se pudo mandar correctamente.
- ▶ **Envío asíncrono:** en la llamada a send, el usuario incluye un método (más concretamente, un objeto que implemente la interfaz Producer.Callback de Kafka), que será invocado automáticamente por Kafka cuando se reciba la respuesta. En el procesamiento, podemos examinar el éxito o fracaso. Tras llamar a send, el proceso continúa su ejecución sin esperar a la respuesta.

El siguiente código Java muestra un ejemplo del tercer método de envío.

```
private class DemoProducerCallback implements Callback {  
    @Override  
    public void onCompletion(RecordMetadata recordMetadata, Exception e) {  
        if(e!= null) {  
            e.printStackTrace();  
        }  
        } // la siguiente linea crea un mensaje para el topic "CustomerCountry"  
    } // con clave "BiomedicalMaterials" y valor "USA"  
    ProducerRecord<String, String> record = new ProducerRecord<>  
("CustomerCountry",  
    "Biomedical Materials", "USA");  
  
producer.send(record, new DemoProducerCallback()); // enviamos el mensaje
```

Si, en lugar de esto, quisiéramos hacer *fire-and-forget*, no necesitaríamos la clase DemoProducerCallback , sino que, en este caso, el envío se realizaría como producer.send(record) . Finalmente, para efectuar un envío síncrono, de manera que bloqueemos el proceso hasta recibir la respuesta, utilizamos el método get de la clase Future de Java. Dicho método se bloquea en espera de la respuesta, que puede ser una excepción.

```
try {  
    producer.send(record).get();  
} catch (Exception e) {  
    e.printStackTrace();  
}
```

## Serialización: el formato Avro

Cuando queremos enviar como mensaje una instancia de una clase más compleja que hemos definido en nuestro dominio concreto del problema (por ejemplo, un objeto de la clase Cliente que tiene varios campos), no es trivial transformarlo en una representación como *array* de bytes. De este modo, podríamos simplemente convertir cada uno de los campos (DNI, número de cuenta, domicilio...) y enviar el *array* de bytes. Cuando una aplicación consumidora necesitara reconstruir ese objeto a partir de los bytes, tendría que saber exactamente que los 40 primeros bytes representan el DNI y los 80 siguientes, el número de cuenta, etc.

Este proceso es muy frágil y propenso a sufrir errores. Por eso, se suele utilizar algún formato predefinido para la serialización. El más frecuente en Kafka es Avro, en el cual el esquema o estructura de un mensaje o de un archivo de datos se almacena por separado, en un archivo en formato JSON como el siguiente:

```
{"namespace": "customerManagement.avro",  
 "type": "record",  
 "name": "Customer",  
 "fields": [  
     {"name": "id", "type": "int"},  
     {"name": "name", "type": "string"},  
     {"name": "faxNumber", "type": ["null", "string"], "default": "null"}  
 ]  
}
```

Avro presenta la ventaja de que es posible cambiar la estructura de los datos sin cambiar su esquema ni provocar errores en tiempo de ejecución. El esquema tiene

que estar presente en el momento de hacer la lectura (deserialización); no obstante, si intentamos obtener un campo que existe según el esquema, pero no realmente en los datos binarios, simplemente obtendremos *null* para ese campo.

## 6.7. Implementación de consumidores Kafka

Como hemos visto, un consumidor es una aplicación que, mediante el empleo de la API proporcionada por Kafka, es capaz de leer mensajes de un *topic* concreto que es de su interés. Dichos mensajes están organizados en particiones, las cuales se encuentran físicamente distribuidas entre los brókeres. Antes de entrar en los detalles de implementación de los consumidores igual que hicimos en la sección previa con los productores, es necesario revisar un concepto clave en Kafka: los grupos de consumidores. Una vez tengamos claro qué son, por qué son tan importantes y cómo funcionan, pasaremos a ver algunos detalles de implementación con la API Java de Kafka.

### Grupos de consumidores

Es habitual que un consumidor lea mensajes de Kafka, realice algún tipo de procesado con ellos y los almacene o envíe a un nuevo *topic* de Kafka. Tanto el procesado como el almacenamiento son procesos no excesivamente rápidos, que pueden suponer un problema si el ritmo de publicación de mensajes es muy alto y el consumidor no puede seguirlo. Recordemos que Kafka nació en un entorno, LinkedIn, donde el tráfico de mensajes creados era muy alto y, consecuentemente, se requería que los consumidores pudieran procesar ese gran volumen de datos publicados rápidamente a un ritmo adecuado.

En casos como este, resulta necesario un mecanismo para escalar el consumo de mensajes de un *topic*. Para ello, podemos crear varios procesos consumidores que lean del mismo *topic* de forma paralela, lo que permite aumentar el ritmo de consumo de mensajes. Con esta idea en mente nace el concepto **grupo de consumidores**. Generalmente, un proceso consumidor forma parte de un grupo de consumidores concreto. Las particiones existentes en el *topic* al que está suscrito el grupo de consumidores son asignadas por Kafka a sus distintos miembros. Por ejemplo, si

tenemos cuatro particiones en un *topic* y existe un grupo con un único consumidor, este será el encargado de leer de todas las particiones; sin embargo, si el grupo está compuesto por dos consumidores, cada uno solo tendrá acceso a dos de las particiones (figura 10).

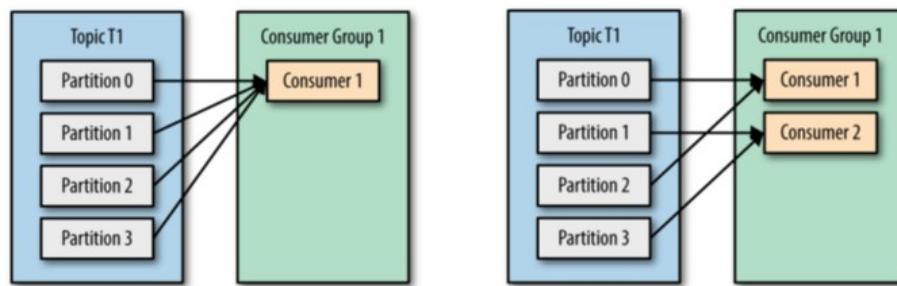


Figura 10. Asignación de particiones de un *topic* a los consumidores de un grupo. Fuente: Narkhede, Shapira y Palino, 2017.

Nótese que, en este caso, no tendría sentido tener más de cuatro consumidores en el grupo, porque un quinto consumidor quedaría ocioso al estar ya asignadas todas las particiones a algún consumidor (figura 11). Sería un supuesto similar a tener más procesadores (CPU) que particiones haya en un RDD que estemos procesando con Spark: alguna CPU quedaría ociosa.

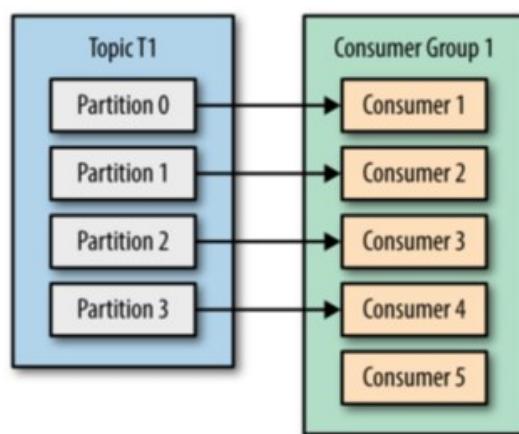


Figura 11. Un consumidor queda ocioso, dado que hay más consumidores en el grupo que particiones. Fuente: Narkhede, Shapira y Palino, 2017.

Otra opción para escalar la lectura de mensajes es crear varios grupos diferentes de consumidores, pertenecientes a aplicaciones distintas, las cuales usan los mismos datos, pero de diversa forma. Por ejemplo, como veíamos al principio del capítulo, los mismos datos de métricas los podían usar unos consumidores para procesarlos y generar reportes, otros para monitorizarlos y crear alarmas, y otros para generar visualizaciones en tiempo real. En estos casos, lo más común es tener un grupo de consumidores para cada aplicación diferente, aunque todos lean del mismo *topic*, como se muestra en la figura 12.

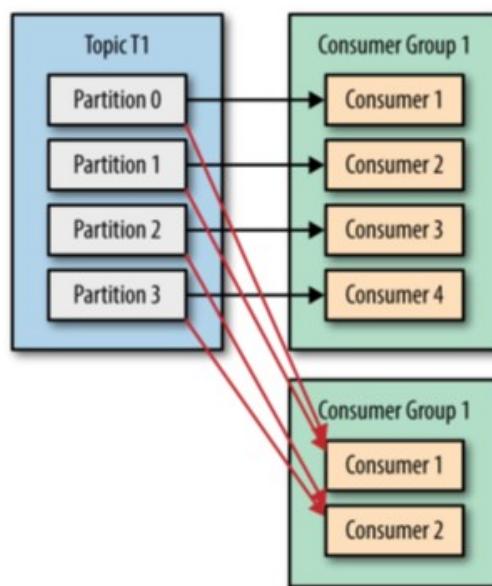


Figura 12. Reparto de particiones cuando hay dos grupos de consumidores. Fuente: Narkhede, Shapira y Palino, 2017.

Cuando se añaden o eliminan consumidores de un grupo, Kafka automáticamente reasigna las particiones, operación conocida como rebalanceo. Aunque resulta muy útil, hace que el grupo de consumidores no pueda leer del *topic* hasta que el proceso finaliza; por tanto, es conveniente evitar rebalanceos que no sean estrictamente necesarios. Para detectar cuándo un consumidor ha dejado de ser parte del grupo de consumidores (porque ha fallado o por cualquier otra razón), existe un *heartbeat* o señal periódica, enviada por los consumidores para confirmar que siguen estando

activos y formando parte del grupo en el que estaban.

## Detalles de implementación de consumidores

Ahora que ya entendemos cómo funcionan realmente los consumidores bajo los grupos de consumidores, veamos algunos detalles de cómo implementarlos. Al igual que ocurre con los productores, también es necesario crear y configurar un objeto `KafkaConsumer` en el que se indiquen, al menos, cuatro propiedades: ubicación de los brókeres (`bootstrap.servers`) , deserializadores de clave y valor (`key.deserializer`, `value.deserializer`) , y, aunque estrictamente es opcional, vamos a considerar también el ID del grupo de consumidores (`group.id`) . Si bien es posible tener un consumidor que no esté dentro de ningún grupo de consumidores, es un caso muy poco habitual. Una vez hemos configurado todos estos parámetros en el objeto de la clase `Properties` , ya podemos crear el objeto consumidor. Para ello, indicamos los tipos de clave (*String*) y valor de mensaje (*String*) que espera recibir.

```
// Código tomado de Narkhede, Shapira y Palino (2017), y adaptado:  
  
// Configuramos las opciones obligatorias del consumidor:  
// ubicación de brókeres, id del grupo,  
// deserializador de clave y valor (deserializador de Strings)  
Properties props = new Properties()  
props.put("bootstrap.servers", "brokerA:9092,brokerB:90902");  
props.put("group.id", "country_counter_visualizer");  
props.put("key.deserializer",  
    "org.apache.kafka.common.serialization.StringDeserializer");  
props.put("value.deserializer",  
    "org.apache.kafka.common.serialization.StringDeserializer");  
  
// Creamos el consumidor, que espera una clave de tipo String y  
// un valor de mensaje de tipo String,  
// y cuya configuración está recogida en props  
KafkaConsumer<String, String> consumer = new KafkaConsumer<String, String>(props);  
  
// Indicamos el topic al que se va a suscribir el consumidor  
// (el topic donde publicaba el productor de la sección previa):  
consumer.subscribe(Collections.singletonList("customerCountries"));
```

El código de la aplicación concreta que estamos desarrollando depende de la lógica de negocio que queramos incluir, pero es habitual que contenga un bucle infinito para recibir mensajes, similar al que se indica más abajo. Sin entrar en detalles, lo que hace básicamente es preguntar a Kafka si hay nuevos mensajes disponibles en ese *topic* mediante el método `poll`, lo cual es necesario para notificar a Kafka que este consumidor sigue activo. El argumento de `poll` es el número de milisegundos que el proceso esperará antes de retornar (100 en este ejemplo) con el número de registros que Kafka le haya proporcionado durante ese intervalo. El resultado es una lista de registros (mensajes), cada uno de los cuales contiene el *topic* y la partición a la que pertenece, el *offset* por el que vamos consumiendo en esa partición y, por supuesto, la clave (`key`) y el valor (`value`) que constituyen el mensaje propiamente.

A continuación, se procesa ese registro ejecutando la lógica de negocio de la aplicación (por ejemplo, insertando información en una base de datos; en este caso, simplemente mostramos por pantalla el valor de cada mensaje). Vemos que el bucle finaliza en caso de que Kafka lance una excepción. En este caso, el código que gestiona la excepción se ocupa de cerrar la comunicación de manera limpia, lo cual lanza internamente un rebalanceo de las particiones asignadas a los consumidores, ya que el proceso deja de existir.

```
// Código tomado de Narkhede, Shapira y Palino (2017), y adaptado:  
try {  
    //Bucle infinito  
    while (true) {  
        // El consumidor pregunta (poll) por nuevos records (mensajes)  
        // del topic al que está suscrito ("customerCountries")  
        ConsumerRecords<String, String> records = consumer.poll(100);  
  
        // Por cada mensaje (record) que Kafka le envía del topic:  
        for (ConsumerRecord<String, String> record: records)  
        {  
            // Imprime por pantalla el valor (value) del mensaje:  
            System.out.println(record.value());  
        }  
    }  
}
```

```
} finally {  
    // Cierra limpiamente las conexiones:  
    consumer.close();  
}
```

## 6.8. Referencias bibliográficas

Narkhede, N., Shapira, G. y Palino, T. (2017). *Kafka: the definitive guide*. O'Reilly.

<https://www.confluent.io/resources/kafka-the-definitive-guide/>

## Kafka: the definitive guide

Narkhede, N., Shapira, G. y Palino, T. (2017). *Kafka: the definitive guide*. O'Reilly.

<https://www.confluent.io/resources/kafka-the-definitive-guide/>

Guía muy completa, que incluye todos los detalles de funcionamiento e implementación. Su descarga está disponible de manera gratuita.

## Kafka. Documentación oficial

Apache Software Foundation (2017). *Apache Kafka. Documentation.*

<https://kafka.apache.org/documentation/>

Documentación detallada sobre conceptos de Apache Kafka y la versión más actualizada de su API. Puede consultarse de manera gratuita.

**1.** ¿Qué es Apache Kafka?

- A. Un sistema de mensajería que utiliza Spark para funcionar.
- B. Un bus de datos distribuido, en el que varias aplicaciones pueden leer y escribir.
- C. Un sistema de colas basado en MapReduce.
- D. Un sistema de data *warehousing*.

**2.** Cuando usamos Kafka...

- A. Cada aplicación elige el tipo de mensajes que desea leer.
- B. Todas las aplicaciones reciben todos los mensajes.
- C. Solo las aplicaciones registradas en Spark pueden acceder al bus.
- D. Cada aplicación puede leer solo un tipo de mensajes.

**3.** ¿Cuál de estas funciones es típica de Kafka?

- A. Transmitir mensajes generados por una aplicación a otras que los utilizan.
- B. Almacenar información accesible para distintas aplicaciones, tal como lo hace una base de datos.
- C. Realizar procesados de flujos de información.
- D. Ninguna de las opciones anteriores es correcta.

**4.** Un *topic* de Kafka...

- A. Es una partición de los datos subyacentes.
- B. Es un conjunto de mensajes que comparten la misma estructura.
- C. Es equivalente a una base de datos.
- D. Es una máquina que almacena cierto tipo de datos.

5. Si en un grupo de consumidores hay más consumidores suscritos a un *topic* que particiones tiene dicho *topic*:
  - A. Kafka reparte los mensajes entre todos consumidores de una misma partición.
  - B. Kafka no permite que esto ocurra y denegará la suscripción al consumidor.
  - C. Uno o más consumidores quedarán ociosos, sin poder consumir mensajes.
  - D. Todos los consumidores reciben mensajes de todas las particiones.
6. Cuando un proceso productor de Kafka utiliza envío asíncrono:
  - A. Se bloquea en espera de la respuesta que confirme que todo ha ido bien.
  - B. Prosigue su ejecución, ya que, al ser asíncrono, no espera respuesta alguna.
  - C. Prosigue su ejecución y Kafka invocará el método que el productor indicó cuando tenga disponible la respuesta.
  - D. Ninguna de las respuestas anteriores es correcta.
7. ¿Qué implica que un bróker contenga la partición líder de un *topic*?
  - A. Que será quien reciba y procese las peticiones de lectura y escritura a esa partición.
  - B. Que decidirá si un consumidor está autorizado para suscribirse al *topic*.
  - C. Que será quien centralice las peticiones de escritura que reciben todos los brókeres que contengan dicha partición.
  - D. Las tres opciones anteriores son correctas.

- 8.** ¿Cuál de las siguientes afirmaciones sobre Kafka es correcta?
- A. Un mensaje en Kafka se elimina según el primer consumidor lo lee.
  - B. Un mensaje en Kafka se elimina siempre tras pasar una semana en el bróker.
  - C. Un mensaje en Kafka se elimina cuando lo han leído el número configurado de consumidores.
  - D. Un mensaje en Kafka se elimina cuando se cumple el tiempo o tamaño configurados.
- 9.** El bróker encargado de supervisar qué brókeres se unen y cuáles dejan el clúster es:
- A. El bróker líder.
  - B. El bróker controlador.
  - C. El bróker sincronizado (*in-sync*).
  - D. Se encarga Zookeeper.
- 10.** Un clúster Kafka está compuesto:
- A. Por uno o varios brókeres, y siempre incluye Zookeeper para gestionar metadatos.
  - B. Por, al menos, dos o más brókeres, y se complementa a veces con Zookeeper para facilitar la gestión de metadatos.
  - C. Por uno o varios brókeres, y se complementa a veces con Zookeeper para facilitar la gestión de metadatos.
  - D. Ninguna de las opciones es correcta.

Ingeniería para el Procesado Masivo de Datos

---

## Tema 7. Hive e Impala

# Índice

[Esquema](#)

[Ideas clave](#)

[7.1. Introducción y objetivos](#)

[7.2. Apache Hive](#)

[7.3. Apache Impala](#)

[7.4. Referencias bibliográficas](#)

[A fondo](#)

[Hadoop: the definitive guide](#)

[Next-generation big data](#)

[Guía Apache Impala](#)

[Test](#)

# Esquema

APACHE HIVE	APACHE IMPALA
CONCEPTO	CONCEPTO
Data Warehouse se sobre clúster que usa motores de procesamiento distribuidos para consultar datos	Herramienta de procesamiento masivo paralelo sobre clúster para consulta de datos.
Consulta datos almacenados en ficheros en HDFS o AWS S3	Consulta SQL de datos almacenados en ficheros en HDFS, S3, AWS S3 y Kudu
Disenado para ser un almacén SQL para consultar datos estructurados y hacer ETL	Mismo formato de metadatos que Hive, pueden compartir catálogo
NO es una base de datos	Diseñada para ser un almacén SQL para consultar datos estructurados y hacer ETL
No está pensado para consultas interactivas, sino para trabajos batch	NO es una base de datos
Facilita la consulta a perfiles business que conocen SQL, pero no lenguajes de programación (Java, Python...)	catalogo: ejecuta en un único nodo, difundiéndole cambios en el catálogo de metadatos.
	statestore: comprueba que cada demonio impala sigue activo e informa cuando alguno se cae
	Impala: ejecuta en todos los nodos del clúster en segundo plano
	Trabaja en memoria
	Tres tipos procesos
	demonio

## 7.1. Introducción y objetivos

En este tema, estudiaremos dos herramientas más del ecosistema *big data* en la actualidad: Apache Hive y Apache Impala. Ambas son herramientas de *software libre* y están auspiciadas por la Apache Software Foundation (ASF). Apache Hive es una herramienta del ecosistema Hadoop para hacer consultas SQL sobre datos distribuidos, que puede utilizar como motor de ejecución MapReduce, Spark o Tez, indistintamente. Por su parte, Apache Impala permite el acceso a diversas fuentes de datos distribuidas de manera uniforme, mediante el lenguaje SQL. A lo largo estas páginas, veremos las principales similitudes y, sobre todo, las diferencias entre ambas herramientas.

Los objetivos que persigue este tema son:

- ▶ Hacer consciente al alumno de la potencia de Apache Hive y su uso desde herramientas externas de inteligencia de negocio (*business intelligence*).
- ▶ Comprender la utilidad práctica de Apache Impala y las necesidades que resuelve.

## 7.2. Apache Hive

En capítulos previos, se han descrito tecnologías *big data* como MapReduce o Spark, que permiten realizar procesados de grandes volúmenes de información. Además, proporcionan gran flexibilidad en los tipos de procesado disponibles y ofrecen la posibilidad clave de escalarlos para poder hacer frente a cantidades de información en continuo crecimiento. Sin embargo, en ambos casos, realizar estos procesados requiere un conocimiento nada despreciable de algunos de los lenguajes de programación soportados por dichas tecnologías, como Python, Java o Scala, entre otros. Esto conlleva que, en muchos entornos empresariales, existan enormes cantidades de información almacenadas, a la espera de ser procesadas para aprovechar su enorme potencial, pero donde no hay profesionales con las habilidades de programación necesarias para poder realizar estos procesados con las tecnologías que hemos visto ahora.

Sin embargo, una habilidad que tiene un público mucho más generalizado es el lenguaje SQL, que igualmente proporciona la capacidad de analizar datos de una base de datos relacional, en este caso para obtener diversos análisis de la información almacenada en una base de datos. ¿Cómo acercar la posibilidad de consultar grandes volúmenes de datos utilizando un lenguaje de consulta ampliamente conocido como SQL, en lugar de lenguajes de programación más complejos y menos conocidos y manejados por los analistas y profesionales que necesitan trabajar con estos datos? Con esta pregunta en mente, el equipo de Jeff Hammerbacher en Facebook creó Apache Hive, una herramienta más en el ecosistema Hadoop para analizar grandes volúmenes de datos almacenados en HDFS mediante un lenguaje similar a SQL, que muchos analistas ya conocían.

Podemos definir Apache Hive como:

*Sistema de **data warehouse** para manejar (leer, escribir, ETL, reporting y análisis), mediante lenguaje SQL, datos almacenados de manera distribuida.*

Fue desarrollado inicialmente por Facebook en el año 2007 y donado a la Apache Software Foundation en 2010 para convertirlo en un proyecto de código abierto (*open source*). Hive proporciona **un armazón SQL** para manejar datos existentes (estructurados) almacenados en diversas fuentes, de las cuales la más frecuente suele ser HDFS, aunque también soporta HBase (base de datos distribuida no relacional) y el sistema de almacenamiento distribuido Amazon S3.

Se trata de una herramienta para acceder fácilmente a los datos mediante el empleo de lenguaje SQL. Está destinado principalmente a analistas de inteligencia de negocio (*business intelligence*) que no dominan la programación en Java (y, por tanto, no tienen facilidad para usar *frameworks* de procesamiento de propósito general como MapReduce o Spark) y a la realización de tareas como ETL, *reporting* y análisis de datos.

El lenguaje de consulta que utiliza se denomina HiveQL y permite ejecutar prácticamente cualquier sentencia de SQL estándar, además de tener soporte para funciones a nivel de fila definidas por el usuario (*user defined functions*, UDF) y tipos de datos complejos.

Apache Hive se utiliza principalmente en modo OLAP (*online analytical processing*), es decir, para realizar análisis en bloque (*batch*) del informacional (datos históricos) de la empresa (agregaciones, *reporting*, procesos de cálculo que no requieren respuesta urgente). No es adecuado para OLTP (*online transactional processing*), ya que no está pensado para manejo interactivo con respuestas rápidas.

## Ejemplo de uso

Para entender un poco mejor cómo Hive permite consultar datos almacenados en un clúster Hadoop usando lenguaje similar a SQL, vamos a ver un breve ejemplo. Tenemos almacenado en HDFS el fichero flights.csv (vamos a suponer una versión simplificada del fichero real que hemos usado en algún otro ejemplo, ahora con solo algunas columnas):

```
> hdfs dfs -ls /user/data
Found 2 items
-rw-r--r-- 3 root supergroup 5950040 2020-10-21 17:22 /user/data/airport-
codes.csv
-rw-r--r-- 3 root supergroup 11244080 2020-10-21 17:22
/user/data/flights.csv
```

El contenido del fichero flights.csv simplificado que vamos a considerar es el siguiente:

```
"year","month","day","origin","dest"
2014,1,1,"PDX","ANC"
2014,1,1,"SEA","CLT"
...
```

```
ident,name,iso_country,iata_code
LELT,Lillo,ES,
LEMD,Adolfo Suárez Madrid-Barajas Airport,ES,MAD
...
```

Tal y como sucede en una base de datos relacional, Hive organiza los datos en tablas. Por tanto, vamos a crear una tabla para gestionar los datos de vuelos:

```
CREATE TABLE flights (year INTEGER, month INTEGER, day INTEGER, origin STRING, dest STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ',';
```

Con esta sentencia, generamos una tabla *sales* con cinco columnas: *year*, *month*, *day*, *origin* y *dest*, cada una con su tipo correspondiente. También indicamos que cada fila está compuesta por valores separados por comas (''). Como se puede comprobar, la sentencia es muy similar a la que podríamos encontrar en una base de datos SQL, aunque la definición del formato de fila indicada por *row format* es particular de Hive. Hasta aquí únicamente hemos definido la tabla, pero aún no tiene datos que gestionar. Para cargar los datos correspondientes, usaremos la siguiente sentencia:

```
LOAD DATA INPATH '/user/data/flights.csv'
INTO TABLE flights;
```

Dicha sentencia solicita a Hive que coloque el fichero especificado en el directorio donde almacena los datos. Es muy importante notar aquí que en ningún momento se ha hablado de crear tablas como lo haríamos en una base de datos relacional, sino que lo que hace Hive es colocar los ficheros que contienen los datos que se desean examinar en su estructura de directorios, para proceder posteriormente a su consulta. Estos ficheros se almacenan *verbatim*, es decir, sin modificación alguna por parte de Hive. El directorio por defecto donde se guardan los ficheros de datos en HDFS es '/user/hive/warehouse', aunque es configurable. Por tanto, tras la sentencia LOAD, se habrá creado un nuevo directorio con el mismo nombre de la tabla (*flights*), que contendrá el fichero de datos:

```
> hdfs dfs -ls /user/hive/warehouse/flights
Found 1 items
-rwxrwxr-x 3 root supergroup 11244080 2020-10-21 17:22
/usr/hive/warehouse/flights/flights.csv
```

Una vez tenemos los datos en Hive (es decir, copiados a su estructura de directorios), podemos realizar consultas sobre ellos. Por ejemplo:

```
SELECT dest, COUNT(dest) AS total_flights_dest
FROM flights
GROUP BY dest;

+-----+-----+
| dest | total_flights_dest |
+-----+-----+
| "ABQ" | 908      |
| "ANC" | 7149     |
....
```

Esta petición, que será fácilmente entendible por aquellos que hayan trabajado previamente con SQL, nos devolverá el total de vuelos (*total\_flights\_dest* significa contabilizar cuántos vuelos aterrizan en cada aeropuerto de llegada, *dest*), agrupados por destino. Pero lo más importante que debemos tener en cuenta es que esta petición no es una consulta SQL al uso, como la de una base de datos relacional, sino que el papel fundamental de Hive es transformar esta petición tan corriente para cualquier analista en un trabajo de procesamiento (ya sea sobre MapReduce, Tez o Spark) de los datos en crudo (recordemos que se consultan los ficheros almacenados en la estructura de directorios de Hive), sin necesidad de programar ningún *script* Java u otros lenguajes de programación, de forma invisible para el analista, y sencillamente devolverle el resultado buscado.

Originalmente, el motor de ejecución de Hive, es decir, el motor de procesamiento distribuido que ejecuta el trabajo (*job*) en el que se ha traducido cada sentencia SQL, se limitaba a MapReduce, pues era el único existente cuando se desarrolló Hive, pero en la actualidad es intercambiable. Hive soporta tres motores posibles: MapReduce, Apache Spark o Apache Tez (activado por defecto). El funcionamiento de Hive, y donde reside la clave de su éxito, consiste en **traducir al usuario, automáticamente y de manera transparente, sus consultas SQL sobre datos distribuidos almacenados en el clúster a trabajos del motor de ejecución** distribuido que se haya configurado (MapReduce, Spark o Tez). Por tanto, la consulta SQL del usuario se puede traducir a un *job* de MapReduce, de Spark o de Tez.

Ya conocemos MapReduce y Spark. Por su parte, **Apache Tez** es un motor de ejecución muy similar a Spark, donde el procesamiento se realiza en memoria, y está orientado especialmente hacia la ejecución de DAG de tareas. Su API es menos amigable que la de Spark y solo puede ejecutarse en el gestor de clúster YARN. Se suele utilizar como motor en aplicaciones de Hadoop como Hive o Pig; no tanto por desarrolladores, para los cuales Spark es más sencillo de manejar y tiene una potencia y rendimiento similares.

## Funcionamiento de Hive

Existen distintas formas de poder usar Hive. La figura 1 muestra tanto los servicios que ofrece como los diferentes clientes (aplicaciones) que pueden interaccionar con él.

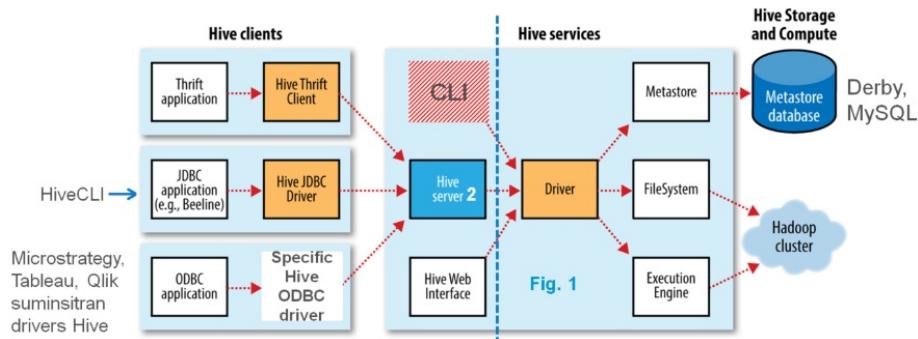


Figura 1. Maneras de utilizar Hive. Fuente: adaptada de White (2015).

Veamos algunos de ellos con mayor profundidad:

- ▶ **Hiveserver2** (o **Thrift Server** en las versiones iniciales). Es un proceso demonio que se ejecuta permanentemente en segundo plano y que escucha peticiones en lenguaje HiveQL. Dicho proceso asume el papel del componente *interface* de la figura 3 y es, por tanto, quien se comunica con el *driver*. El resto de componentes a la derecha de la línea vertical discontinua son los de la figura 3.
- ▶ **CLI (command line interface)**. Versiones anteriores de Hive incluían una línea de comandos con conexión directa al *driver*, pero se eliminó en versiones más recientes; por eso, se marca en rojo.
- ▶ **HWI (Hive web interface)**. Es una interfaz web que sirve como alternativa a la línea de comandos.

Para conectarse al servicio principal, hiveserver2, existen diferentes formas, o clientes:

- ▶ **Cliente Thrift**. Hive se ofrece como un servicio Thrift y, por tanto, se puede interactuar con él usando cualquier lenguaje de programación que soporte dicho lenguaje de definición de interfaz. Hay proyectos que implementan clientes para lenguajes como Python y Ruby. Se pueden consultar los detalles en la [Hive wiki](#).
- ▶ **Driver JDBC**. Las aplicaciones con capacidad para soportar un *driver* JDBC se

pueden conectar a Hive. Por ejemplo, una aplicación Java podría hacerlo usando una dirección JDBC como la siguiente: `jdbc:hive2://host:port/nombre_db`, donde *host* y *port* son la dirección y puerto donde está escuchando Hive, y *nombre\_db* es el nombre de la tabla a la que queremos acceder; o mediante `jdbc:hive2://`, en cuyo caso Hive estaría ejecutándose en la misma JVM que la aplicación que lo invoca. Asimismo, beeline, el intérprete de línea de comandos HiveQL que ofrece Hive, utiliza esta misma interfaz JDBC.

- ▶ **Driver ODBC.** Esta interfaz permite que aplicaciones que soportan el protocolo ODBC puedan conectarse a Hive. Entre ellas, se pueden mencionar numerosas aplicaciones de inteligencia de negocios (*business intelligence*), tales como Tableau, Microstrategy, PowerBI, QlikSense, etc. Los fabricantes de cada una de ellas proporcionan habitualmente un *driver ODBC* propio y ya incorporan Hive como una fuente de datos más (*datasource*) que se puede seleccionar desde la aplicación (figura 2).

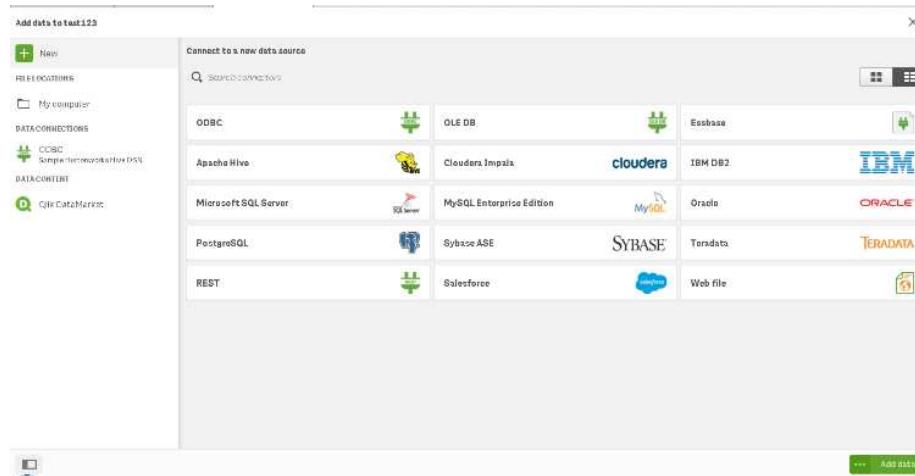
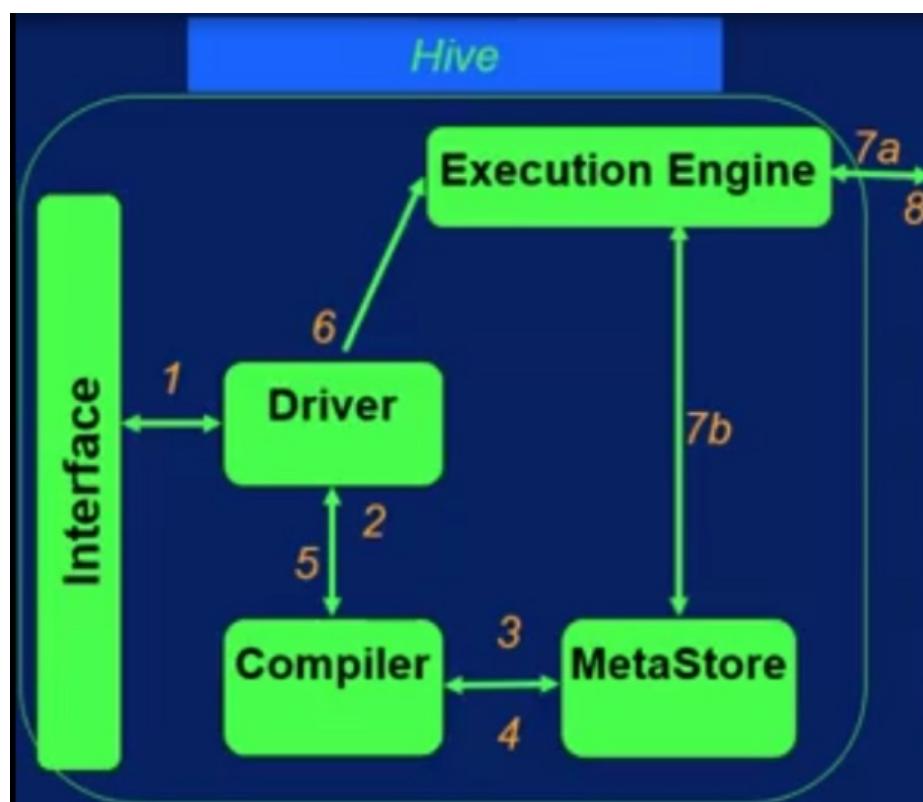


Figura 2. Hive se encuentra entre las fuentes de datos disponibles en la aplicación QlikSense de *business intelligence*.

### Arquitectura de Hive sobre MapReduce

Aunque la parte relativa al motor de ejecución es intercambiable, vamos a examinar la arquitectura original de Hive, cuando las consultas se traducían a trabajos de MapReduce, con el objetivo de entender los pasos que se dan hasta que se ejecuta una sentencia SQL del usuario. Actualmente, lo habitual es que el motor de ejecución sea Spark o Tez. La figura 3 muestra un diagrama de la arquitectura.



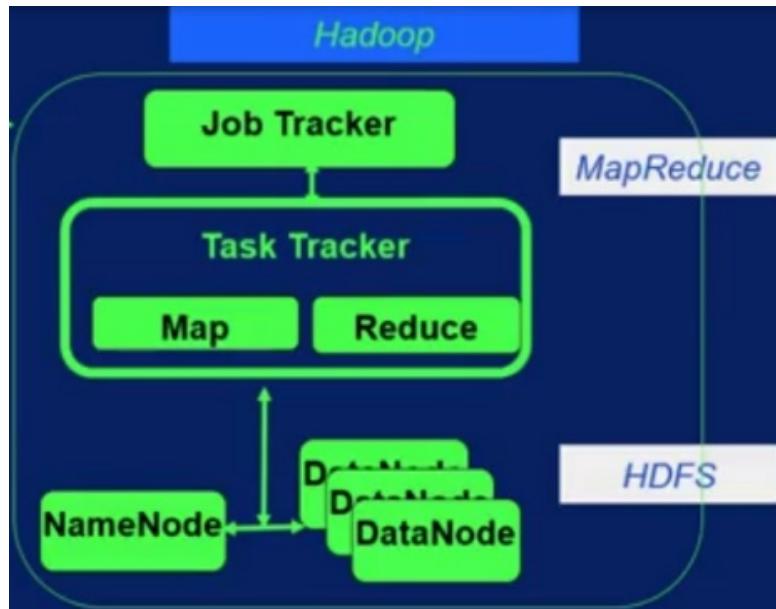


Figura 3. Arquitectura de Apache Hive con MapReduce.

Los módulos de los que se compone son:

- ▶ **Driver**: controlador que recibe sentencias SQL (o, más exactamente, HiveQL) y coordina todo el ciclo de ejecución, recoge resultados de la fase *reduce* y los envía de vuelta a la interfaz desde la que se está usando Hive.
- ▶ **Compilador**: convierte la consulta en un *abstract syntax tree* (AST), comprueba su validez con los datos del *metastore* (que existan las tablas y columnas involucradas en la consulta) y pasa el AST a un DAG optimizado (*stages* y *tasks*) del motor de ejecución concreto que se haya configurado.
- ▶ **Metastore**: es el repositorio central para los metadatos que necesita Hive. Está compuesto de dos partes: el servicio que recupera los metadatos y se los proporciona al *driver*, y la plataforma de almacenamiento de los metadatos como tal. Por defecto, el servicio *metastore* se ejecuta en la misma máquina virtual de Java que el servicio de Hive y contiene, además, una base de datos denominada Derby, que no es más que un fichero almacenado en disco. Esta configuración se denomina ***metastore embebido*** y solo permite tener una única sesión (es decir, un único

cliente que haga peticiones a Hive). Para atender múltiples sesiones a la vez, es necesario utilizar la configuración de **metastore local**. En ella, el servicio de *metastore* se conecta a una base de datos (puede ser cualquier base de datos JDBC, como, por ejemplo, MySQL) que se ejecuta en un proceso distinto.

- ▶ **Motor de ejecución:** envía el plan de trabajos al clúster de Hadoop y también efectúa las operaciones con el *metastore* para actualizarlo si es necesario, según lo que haga la consulta SQL.
- ▶ **Interface:** puede ser desde una CLI (*command line interface*) hasta una aplicación remota de *business intelligence* (por ejemplo, Microstrategy o QlikSense) que se conecta a Hive y envía sentencias SQL que el usuario ha escrito en esa aplicación, como veremos en la siguiente sección.

## Diferencias entre Hive y las bases de datos tradicionales

A pesar de que pueda parecerse en muchos aspectos, ha de quedar claro que **Hive no es una base de datos relacional**, sino un sistema que traduce peticiones (similares a las que se usan en este tipo de bases de datos) a trabajos MapReduce/Spark/Tez, para realizar consultas sobre datos guardados en ficheros de forma distribuida. Esto conlleva importantes diferencias y limitaciones que deben tenerse en cuenta:

- ▶ En bases de datos tradicionales, el esquema de una tabla se comprueba al cargar los datos. Si el formato de los datos cargados no coincide con el esquema definido, no se cargan. Esto se denomina *schema-on-write*, es decir, se comprueba que los datos tienen la estructura definida por el esquema cuando se escriben en la base de datos. El patrón que sigue Hive, sin embargo, es justo el opuesto, denominado ***schema-on-read***. Esto quiere decir que los datos no se comprueban al ser escritos (porque, recordemos, los datos ya están escritos en ficheros cuando Hive los consulta), sino que se comprueban al ser consultados, es decir, al ser leídos. Esto permite que la carga de datos sea muy rápida (tanto como mover o copiar ficheros a una estructura de directorios), ya que no es necesario leer, comprobar y escribir los

datos en el formato interno propio de una base de datos tradicional.

- ▶ Transacciones, *updates* e índices son elementos comunes de las bases de datos relacionales. Sin embargo, en Hive no han aparecido hasta versiones muy recientes porque la tecnología está pensada para escanear tablas (ficheros) completas o realizar consultas sobre grandes partes de ellas. Las inserciones de nuevos datos en forma de ficheros que añadir a una tabla han estado contempladas desde versiones iniciales, pero no así las inserciones a nivel de valor, que solo son posibles desde versiones recientes, ya que no era la filosofía inicial de Hive. Hay que tener en cuenta que HDFS no proporciona mecanismos para modificar ficheros; por tanto, los cambios resultantes de inserciones (*insert*), actualizaciones (*update*) y eliminaciones (*delete*) se tienen que guardar como ficheros de cambios (deltas) y solo funcionan en el contexto de transacciones (también introducidas en versiones recientes de Hive) para garantizar consistencia.

## Gestión de tablas en Hive

Una tabla en Hive está compuesta por los datos (almacenados en HDFS, disco local o AWS S3) y por una serie de metadatos que describen el esquema de la tabla (almacenados y gestionados por el servicio *metastore*). Existen dos tipos tablas dependiendo de quién gestione sus datos:

- ▶ **Tablas gestionadas por Hive (*managed tables*)**. Al crear una tabla en Hive, por defecto, será este el que gestione los datos y, por tanto, los moverá en su directorio. Para cargar datos en una tabla gestionada por Hive, es suficiente con ejecutar una sentencia como la siguiente:

```
CREATE TABLE flights_managed (year INTEGER, month INTEGER, day INTEGER,
origin STRING, dest STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ',';
LOAD DATA INPATH '/user/data/flights.csv'
INTO TABLE flights_managed;
```

Al ejecutar estas dos sentencias, Hive **moverá** el fichero '/user/data/flights.csv' a su estructura de directorios (por defecto, hdfs://user/hive/warehouse/flights\_managed/flights.csv). Es decir, el fichero original desaparecerá del directorio '/user/data' (en el fondo, es un renombrado del fichero). Uno de los aspectos más importantes que deben tenerse en cuenta cuando se usan tablas gestionadas por Hive es que, si eliminamos la tabla con:

```
DROP TABLE flights_managed;
```

Hive eliminará tanto los datos de la tabla (es decir, el directorio /user/hive/warehouse/flights\_managed y todo su contenido, incluido el fichero flights.csv) como los metadatos almacenados en el *metastore*. Y, dado que el fichero de datos fue movido (no copiado) desde su ubicación original a la estructura de directorios de Hive, al eliminarlo de aquí, también perdemos por completo los datos.

- **Existe la alternativa de crear tablas externas.** En ellas, Hive no gestiona los datos (solo los metadatos). Por tanto, si ejecutamos las siguientes dos sentencias:

```
CREATE EXTERNAL TABLE flights_unmanaged (year INTEGER, month INTEGER, day  
INTEGER, origin STRING, dest STRING)  
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY ','  
LOCATION '/user/data/hive_data';  
LOAD DATA INPATH '/user/data/flights.csv'  
INTO TABLE flights_unmanaged;
```

Hive crea los metadatos necesarios en el *metastore* y mueve el fichero con los datos al directorio indicado por LOCATION . La principal diferencia es que, al eliminar una tabla externa (con la misma sentencia DROP que veímos antes), solo se eliminan los metadatos, pero el fichero de datos queda intacto. Por tanto, en casos en los que los datos que van a ser consultados por Hive serán también usados por otras aplicaciones, es conveniente gestionarlos usando tablas externas para evitar

eliminaciones de datos problemáticas.

En general, cabe tener en cuenta que LOCATION indica a Hive dónde buscar los ficheros que contienen los datos de la tabla. Si no se indica, por defecto será el directorio '/user/hive/warehouse/nombre\_tabla'. Por su parte, LOAD mueve los datos del directorio indicado al directorio especificado por LOCATION o, en su ausencia, al directorio por defecto. Asimismo, EXTERNAL indica a Hive que no borre los datos cuando se ejecute la sentencia DROP.

Otro aspecto interesante es la amplia variedad de formatos del fichero de datos que soporta Hive. El más habitual es el de fichero de texto plano ( TEXTFILE ), donde los campos y filas están delimitados por diferentes separadores. Por ejemplo, para crear una tabla cuyos datos están almacenados en un fichero de texto plano, con los campos delimitados por ';' y las filas delimitadas por salto de línea ('\n'), la sentencia de creación de la tabla sería algo como lo que sigue:

```
CREATE TABLE ...
ROW FORMAT DELIMITED
  FIELDS TERMINATED BY ';'
  LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

Puesto que esta es la opción por defecto, lo anterior se puede expresar también de forma más resumida como:

```
CREATE TABLE ...
ROW FORMAT DELIMITED
  FIELDS TERMINATED BY ';' # el separador por defecto es ','
```

Hay que tener en cuenta que los **delimitadores que soporta Hive son de un único carácter**. Si el fichero de datos tiene delimitadores con más de un carácter, es necesario editar el fichero desde fuera de Hive para cambiarlos por otros con un

único carácter antes de intentar crear la tabla correspondiente en Hive.

Hive también soporta otros formatos de fichero como Avro, Parquet, RCFFiles, ORCFiles y Sequence. Por ejemplo, si queremos crear una tabla a partir de datos almacenados en un fichero con formato Avro, se puede usar una sentencia como la siguiente, donde no es necesario especificar delimitadores, puesto que estos tipos de archivo son estructurados. Basta cambiar AVRO por PARQUET, SEQUENCEFILE, RCFILE u ORC para dar cabida al resto de formatos.

```
CREATE TABLE ...
STORED AS AVRO;
```

## Ejecución de una consulta en un clúster de Hive

Vamos a ver un ejemplo breve, pero completo, de uso de Hive para hacer consultas con diferentes tablas. Con este fin, usaremos las tablas simplificadas flights.csv y airport-codes.csv, almacenadas ambas en /user/data y cuyas primeras líneas son las siguientes:

```
"year","month","day","origin","dest"
2014,1,1,"PDX","ANC"
2014,1,1,"SEA","CLT"
...
ident,name,iso_country,iata_code
LELT,Lillo,ES,
LEMD,Adolfo Suárez Madrid-Barajas Airport,ES,MAD
...
```

Vamos a cargar el primero de ellos como tabla externa (no gestionada por Hive, porque lo van a usar más aplicaciones) y el segundo como tabla gestionada por Hive. El objetivo va a ser obtener los diez aeropuertos de destino que han recibido un número mayor de vuelos. En concreto, de esos diez aeropuertos, queremos saber su

nombre y el número de vuelos que han recibido.

```
# Creamos el esquema de la tabla.
# Indicamos los separadores del fichero de texto.
# Indicamos en qué directorio queremos que Hive mueva los datos:
CREATE EXTERNAL TABLE flights (year INTEGER, month INTEGER, day INTEGER,
origin STRING, dest STRING)
ROW FORMAT DELIMITED
  FIELDS TERMINATED BY ','
  LINES TERMINATED BY '\n'
STORED AS TEXTFILE
LOCATION '/user/data/flights_data';

# Esta sentencia nos mostrará la tabla flights:
SHOW TABLES;

# Pero, con esta sentencia, veremos que la tabla está vacía todavía:
SELECT * FROM flights;

# Indicamos que la primera línea es de encabezado:
ALTER TABLE flights SET TBLPROPERTIES ("skip.header.line.count""=1");

# Cargamos los datos en la tabla (los movemos de su ubicación
# actual a la indicada en LOCATION:
LOAD DATA INPATH '/user/ data/flights.csv'
OVERWRITE INTO TABLE flights;

# Ahora la tabla sí tiene datos. ¡OJO! Es importante limitar
# los datos a mostrar a unas pocas líneas; si no, ¡imprimirá todas!
SELECT * FROM flights LIMIT 10;

# Contamos cuántos vuelos llegan a cada aeropuerto de destino.
# Para ello, agrupamos por dest y, por cada grupo, obtenemos
# el código del destino (dest) y el número de filas (COUNT(dest)).
# Creamos una vista en lugar de una tabla, para que no se generen
# datos nuevos, sino para que, cuando se consulte la vista, se
# ejecuten los trabajos necesarios (guardados en la vista) para
# obtener el resultado:
CREATE VIEW IF NOT EXISTS destAirports AS
  SELECT dest, COUNT(dest) AS destCount
    FROM flights
   GROUP BY dest
  ORDER BY destCount DESC;

# Aquí, al consultar los resultados de la vista, es cuando se
# ejecutan los trabajos MapReduce/Tex/Spark (según la
```

```

# configuración) necesarios:
SELECT * FROM destAirports LIMIT 10;

# Nos falta saber el nombre de cada aeropuerto.
# Para eso, cargamos la tabla de info de aeropuertos.
# En este caso, va a ser una tabla gestionada por Hive:
CREATE TABLE airports (ident STRING, name STRING, iso_country STRING,
iata_code STRING)
ROW FORMAT DELIMITED
  FIELDS TERMINATED BY ','
  LINES TERMINATED BY '\n'
STORED AS TEXTFILE;

SHOW TABLES;

SELECT * FROM airports;

ALTER TABLE airports SET TBLPROPERTIES ("skip.header.line.count"="1");

LOAD DATA INPATH '/user/data/airport-codes.csv'
OVERWRITE INTO TABLE airports;

SELECT * FROM airports LIMIT 10;

# Ahora solo falta hacer un join de ambas tablas,
# donde dest de la tabla de vuelos sea igual a iata_code
# de la tabla de aeropuertos, y quedarnos con el nombre
# del aeropuerto y el número de vuelos.
# La concatenación que vemos se debe a que, en la tabla flights,
# el código de los aeropuertos está entrecomillado, y en la
# tabla de aeropuertos, no. Para poder hacer el join, los códigos
# tienen que ser exactamente iguales.
CREATE VIEW IF NOT EXISTS topAirports AS
  SELECT airp.name AS airportName, counts.destCount AS flightsCount
  FROM destAirports counts
  JOIN airports airp
  ON counts.dest = CONCAT("\"", airp.iata_code, "\"");

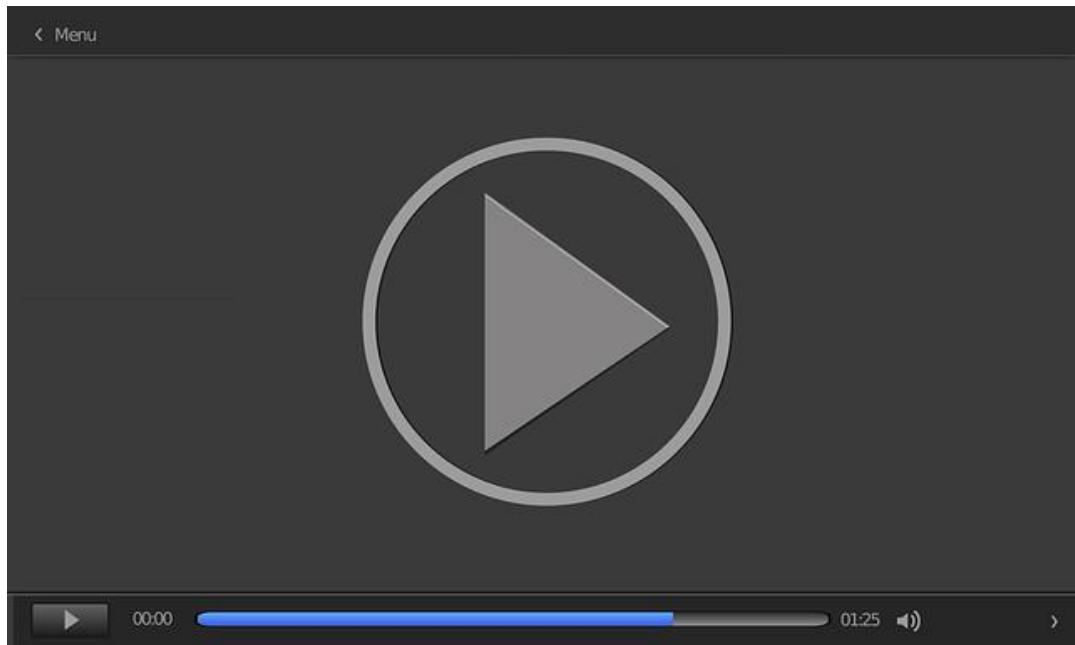
SELECT * FROM topAirports LIMIT 10;

# Por último, vamos a escribir la vista en fichero.
# Para ello, indicamos el DIRECTORIO de salida donde se escribirán
# uno o varios ficheros:
INSERT OVERWRITE DIRECTORY '/user/data/results'
ROW FORMAT DELIMITED

```

```
FIELDS TERMINATED BY '\t'  
STORED AS TEXTFILE  
SELECT * FROM topAirports;  
  
# Solo queda descartar las vistas  
DROP VIEW topAirports;  
DROP VIEW destAiports;  
# y descartar las tablas.  
# Aquí los datos se quedan en HDFS (tabla externa):  
DROP TABLE flights;  
# En este caso, los datos se borran de HDFS  
#(tabla gestionada por Hive):  
DROP TABLE airports;
```

Antes de pasar al siguiente epígrafe, te invitamos a visualizar el siguiente vídeo, en el que vamos a profundizar sobre el sistema *data warehouse* Hive, que, como hemos estudiado, se emplea para manejar datos que están almacenados de forma distribuida utilizando exclusivamente lenguaje SQL.



Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=614f91eb-a5c5-4917-8bf1-b01100a57eff>

## 7.3. Apache Impala

Apache Impala es una herramienta de procesamiento masivamente paralelo (MPP) para ejecutar **consultas interactivas** en lenguaje SQL. Ofrece la posibilidad de acceder a datos almacenados de manera distribuida en HDFS, HBase o en el almacén de objetos distribuidos Amazon S3 (*simple storage service*). También es capaz de escribir en tablas de Apache Kudu. Impala se integra a la perfección en el ecosistema Hadoop, puesto que:

- ▶ Utiliza el mismo *driver ODBC* que Apache Hive, por lo que puede conectarse a herramientas de BI del mismo modo que Hive.
- ▶ Utiliza un lenguaje de consulta (HiveQL) y un formato para sus metadatos similares, por lo que es capaz de acceder al mismo catálogo de metadatos de Hive y, por tanto, puede leer y escribir en tablas de Hive.
- ▶ Soporta formatos de archivo habituales en HDFS, tales como texto delimitado, Parquet, Avro, SequenceFile o RCFile.

Impala fue creado por Cloudera en el año 2012 y liberado poco después a la Apache Software Foundation. A finales de 2017, dejó la fase de incubación y pasó a ser un proyecto oficial. Aunque presenta similitudes importantes con Hive, existen diferencias estructurales y que afectan a la finalidad para la que fueron concebidas:

- ▶ Apache Hive fue diseñado principalmente para trabajos en *batch* (no interactivos) que suelen requerir cierto tiempo para ejecutarse, como, por ejemplo, procesos ETL (*extract-transform-load*). Hive es capaz de traducir para el usuario, de manera transparente, una consulta formulada en lenguaje SQL a un trabajo (*job*) de cualquiera de los motores de ejecución distribuida subyacentes (MapReduce, Apache Spark o Apache Tez). La consulta es ejecutada por dicho motor, lo cual implica que es necesario crear e inicializar un *job* en el momento de ejecutar la

consulta, y esto tiene cierta sobrecarga.

- ▶ Apache Impala no utiliza ninguno de estos motores para sus consultas. Por el contrario, existe una arquitectura propia de Impala, que crea su propia red de procesos (demonios) en las máquinas del clúster y los emplea para acceder a los bloques de datos de HDFS que estén almacenados físicamente en dicha máquina. De esta manera, se va componiendo el resultado de la consulta. Fue concebido desde el principio para consultas muy rápidas, interactivas, sobre grandes cantidades de datos, lo que lo hace apto para exploración de datos masivos y *business intelligence*. Uno de sus puntos fuertes es que opera en memoria, lo cual aumenta la eficiencia. Como los demonios se mantienen siempre en ejecución en el clúster, no hay que inicializar nada al ejecutar una consulta.

Originalmente, Impala no ofrecía soporte para escritura en disco, por lo que, si fallaba alguna de las máquinas del clúster involucradas en la ejecución de una consulta, esta se malograba por completo y debía repetirse desde el principio. En Hive, esto no ocurre, puesto que el propio motor de ejecución es el que proporciona los mecanismos de *resiliency* necesarios. En versiones posteriores, Impala ha añadido también la capacidad de escribir datos en disco para evitar estos problemas.

## Ejecución de una consulta en un clúster de Impala

Antes de ver el proceso de ejecución, introduciremos la terminología básica de Impala. Se denominan «clientes» de Impala a las aplicaciones externas que pueden requerir consultas a Impala, tales como:

- ▶ Hue (interfaz gráfica de administración que incluye un editor gráfico de consultas SQL).
- ▶ Clientes ODBC o JDBC. Gracias a este protocolo estandarizado, las aplicaciones que lo usan están preparadas para establecer una conexión con cualquier base de datos relacional en general. Un ejemplo claro son las aplicaciones de *business intelligence* (Tableau, Microstrategy, PowerBI, etc.) que acceden a bases de datos

relacionales de todo tipo (MySQL, Oracle, etc.) mediante este protocolo.

- ▶ La *shell* de Impala (línea de comandos de Impala similar a la de Hive). Las tareas de administración de Impala también se realizan usando cualquiera de estas aplicaciones.

Recordemos que el *metastore* de Hive contiene información sobre los datos accesibles por Impala, es decir, metadatos acerca de información estructurada existente en HDFS y que puede ser consultada mediante SQL. Las operaciones de creación, modificación o borrado efectuadas vía consultas en SQL conllevan la modificación de dichos metadatos. Estos cambios son automáticamente visibles por todos los *datanodes* en los que se están ejecutando procesos de Impala, gracias al servicio de catálogo introducido en Impala 1.2.

Tanto Hive como Impala admiten también la definición de tablas externas, para las cuales solamente se crean metadatos, pero no hay ningún movimiento o copia de datos hacia los directorios físicos del almacén de Hive o Impala.

El proceso Impala se ejecuta en cada *datanode*, y coordina y ejecuta las consultas SQL. Cualquier instancia de dicho proceso puede recibir una *query* SQL y, en ese caso, se convierte en coordinadora de la *query*, la cual se distribuye después a todos los nodos de Impala para que estos, actuando como *workers*, ejecuten fragmentos de la misma que se pueden llevar a cabo en paralelo.

## Ejecución de una consulta en Impala

1. Un cliente envía una consulta SQL a Impala, que proporciona interfaces estandarizadas para ello. La aplicación puede conectarse al proceso demonio de Impala (llamado `impalad`) de cualquier nodo del clúster, y dicho proceso pasará a ser el coordinador de la consulta.

1. Impala analiza la consulta y determina las tareas que deben ser realizadas por los distintos demonios impalad del clúster. Se planifica la ejecución para optimizar la eficiencia.
2. Los procesos impalad de cada nodo acceden localmente a los servicios de HDFS y HBase para obtener los datos.
3. Cada uno de ellos envía los resultados parciales al demonio que actuaba como coordinador de esa consulta, el cual devuelve el resultado final al cliente.

## Arquitectura de Impala

### El demonio de Impala

Es el componente fundamental de Impala. Se trata de un proceso que corre en cada *datanode* del clúster, con el nombre de *impalad*. El proceso es único, independientemente de que, en un momento dado, el nodo esté actuando como *worker* o como coordinador de una *query* determinada. Lee y escribe datos en archivos, recibe consultas enviadas desde un cliente de cualquier tipo, las paralleliza, distribuye el trabajo a otros demonios del clúster y envía resultados intermedios al coordinador de una *query*. La figura 7 muestra la arquitectura de Impala en un clúster con tres nodos.

Durante el desarrollo de nuestra aplicación o cuando estemos experimentando con una funcionalidad, lo más cómodo será conectarse siempre a la misma máquina y enviar todas las consultas al mismo nodo. En cambio, para flujos de datos que se ejecutan en un entorno de producción, lo más adecuado es balancear la carga, de manera que las consultas que sea necesario ejecutar se envíen cada vez a un nodo distinto, usando el cliente ODBC/JDBC para conectar a nodos físicos diferentes. Impala no proporciona un mecanismo automático para el balanceo de carga ni tampoco para alta disponibilidad, por lo que es el propio usuario quien debe configurar un平衡ador proxy.

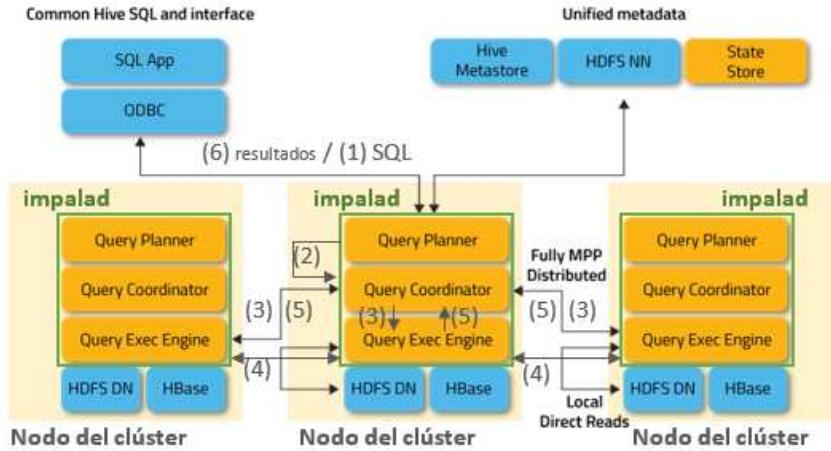


Figura 7. Arquitectura de Impala y pasos para la ejecución de una consulta (entre paréntesis).

Fuente: <http://impala.apache.org>, con elementos añadidos tomados de Kornacker *et al.* (2015).

Las siglas DN hacen referencia a DataNode de HDFS, y las siglas NN, a NameNode.

Los demonios de Impala de los nodos están en comunicación constante con el **Statestore** para confirmar que están activos y pueden aceptar trabajos. Por otro lado, reciben continuamente mensajes de *broadcast* enviados por el demonio **catalogd** (*catalog service*), los cuales indican toda modificación realizada por algún nodo sobre los metadatos (creación, eliminación, modificación de tablas), con el fin de que todos tengan una visión actualizada del catálogo.

## Statestore

Consiste en un proceso demonio llamado `statestored`. Su misión es comprobar que los demonios de Impala están activos. Solo existe una instancia de dicho proceso en todo el clúster. Si uno de los demonios cae debido a un fallo en el *hardware* o en la red, o por cualquier otro motivo, el demonio informa a todos los demonios de Impala de esta situación, a fin de que no envíen trabajos al nodo inaccesible. Dado que su propósito es ayudar cuando se producen errores, no es un proceso crítico para el clúster. En caso de fallo del Statestore (por un problema en el nodo donde está corriendo, por ejemplo), los demonios de Impala continuarán enviándose trabajos

entre sí con normalidad, pero el clúster será menos robusto ante errores que puedan producirse en algún demonio mientras el Statestore esté caído. Por eso, las consideraciones sobre balanceo de carga y alta disponibilidad suelen aplicarse a los impalad , no a este proceso, ya que su caída no ocasiona pérdida de datos.

### **Catalog service**

Es un proceso demonio llamado catalogd , cuya misión es comunicar a los procesos impalad los cambios que ha habido en los metadatos provocados por sentencias SQL. Al igual que ocurre con el Statestore, solo existe una instancia de este proceso en ejecución en todo el clúster. Gracias a él, no es necesario refrescar manualmente el estado de los metadatos. Sin embargo, si se modifica una tabla desde Hive y existe un clúster de Impala utilizando ese mismo catálogo de metadatos, entonces sí es necesario ejecutar manualmente las sentencias REFRESH e INVALIDATE METADATA en un nodo de Impala antes de ejecutar cualquier consulta en él. Esto se debe a que la modificación de los metadatos no se ha realizado desde Impala, sino desde una aplicación externa (Hive), y, por tanto, Impala no puede detectar que los metadatos actualizados difieren de los que conocía cada nodo.

## 7.4. Referencias bibliográficas

White, T. (2015). *Hadoop: the definitive guide*. O'Reilly.

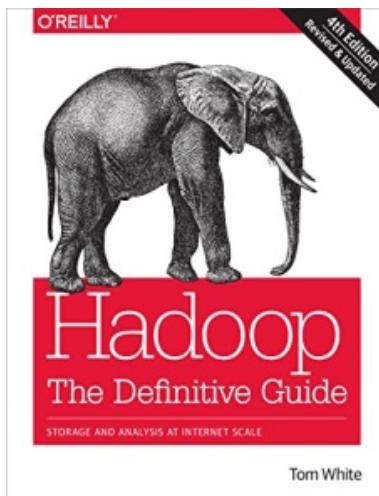
Kornacker, M.; Behm, A.; Bittorf, V.; Bobrovitsky, T.; Ching, C.; Choi, A.; Erickson, J.; Grund, M.; Hecht, D.; Jacobs, M.; Joshi, I.; Kuff, L.; Kumar, D.; Leblang, A.; Li, N.; Pandis, I.; Robinson, H.; Rorke, D.; Rus, S. Russell, J.; Tsirogiannis, D.; Wanderman-Milne, S. y Yoder, M. (2015). *Impala: A Modern, Open-Source SQL Engine for Hadoop*. Actas de la 7a Conferencia Bimensual en Innovative Data Systems Research.

4-7 de enero de 2015. Asilomar, California, EE. UU.

<http://pandis.net/resources/cidr15impala.pdf>

## Hadoop: the definitive guide

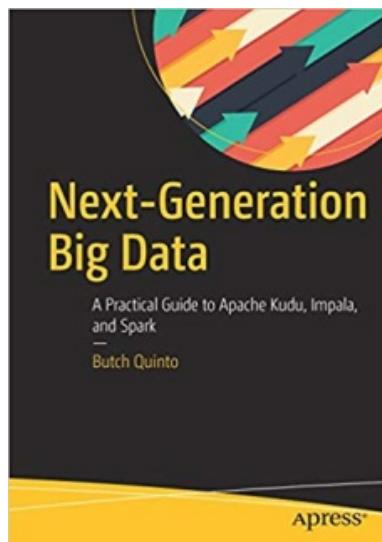
White, T. (2015). *Hadoop: the definitive guide* (4. a edición). O'Reilly.



El capítulo 17 completo está dedicado a Apache Hive.

## Next-generation big data

Quinto, B. (2018). *Next-generation big data: a practical guide to Apache Kudu, Impala, and Spark*. Apress.



Contiene una introducción a Apache Impala, además de otras tecnologías muy actuales.

## Guía Apache Impala

---

Apache Software Foundation. (2019). *Apache Impala Guide.*  
<https://impala.apache.org/docs/build/impala-2.11.pdf>

---

Guía oficial de Apache Impala, escrita por la Fundación Apache para el Software Libre, a la que pertenece el proyecto Impala.

- 1.** Un ejemplo ideal de alguien que puede utilizar Hive es:
  - A. Un analista con conocimientos de SQL que quiere consultar datos estructurados almacenados en HDFS.
  - B. Un programador con conocimientos de MapReduce que quiere consultar imágenes y vídeos.
  - C. Una persona de negocios, con alto conocimiento de Excel, que quiere consultar rápidamente datos masivos guardados en una base de datos relacional como MySQL.
  - D. Los tres casos anteriores son buenos casos de uso.
  
- 2.** Sobre Apache Hive:
  - A. Existen versiones libres y de pago.
  - B. Permite consultar archivos almacenados en HDFS utilizando lenguaje SQL.
  - C. Requiere poseer una base de datos relacional que funcione como respaldo.
  - D. Solo se puede usar como parte de la distribución de Cloudera.
  
- 3.** Hive se define como:
  - A. Una base de datos SQL distribuida.
  - B. Un motor de ejecución distribuido para consultas SQL.
  - C. Una base de datos NoSQL distribuida.
  - D. Un traductor de consultas SQL a trabajos de procesado distribuidos.

**4.** Para usar Hive:

- A. Solo se puede utilizar a través de un intérprete de línea de comandos.
- B. Se puede usar únicamente a través de una conexión JDBC.
- C. Es posible usarlo desde herramientas de BI que dispongan de conector ODBC.
- D. Ninguna de las respuestas anteriores es correcta.

**5.** ¿Cuál de las siguientes afirmaciones sobre Hive es correcta?

- A. Hive siempre utiliza como motor de ejecución Apache Spark.
- B. MySQL puede funcionar como *metastore* de Hive.
- C. Un fichero de texto plano puede funcionar como *metastore* de Hive.
- D. Ninguna de las opciones anteriores es correcta.

**6.** ¿Cuál de las siguientes afirmaciones sobre Hive es correcta?

- A. Cuando se ejecuta la sentencia DROP sobre una tabla, Hive siempre borra los metadatos relacionados con dicha tabla.
- B. Cuando se ejecuta la sentencia DROP sobre una tabla, Hive siempre borra los datos relacionados con esta tabla
- C. Cuando se ejecuta la sentencia DROP sobre una tabla, Hive nunca borra ningún dato ni metadato.
- D. Cuando se ejecuta la sentencia DROP sobre una tabla, Hive siempre borra los datos y metadatos.

7. Señala la respuesta correcta:

- A. Impala está pensado para procesados en bloque (*batch*), mientras que Hive está dirigido a peticiones interactivas.
- B. Impala está dirigido a peticiones interactivas, mientras que Hive está pensado para procesados en bloque (*batch*).
- C. Tanto Impala como Hive están pensados para peticiones interactivas.
- D. Tanto Impala como Hive están pensados para procesados en bloque.

8. ¿Cuál de las siguientes afirmaciones sobre Impala es correcta?

- A. Impala utiliza como motor de ejecución Apache Spark.
- B. Impala utiliza como motor de ejecución Apache tez.
- C. El motor de ejecución de Impala es configurable, igual que en Hive.
- D. Ninguna de las opciones anteriores es correcta.

9. El proceso de Impala que se encarga de ejecutar las consultas del usuario es...

- A. El proceso *statestored*.
- B. El proceso *impalad*.
- C. El proceso *initd*.
- D. El proceso *catalogd*.

10. La manera de ejecutar Impala en un clúster de ordenadores es...

- A. Mediante un proceso que está corriendo en cada máquina y accede directamente a los datos de HDFS de ese nodo.
- B. Mediante el motor de ejecución de Apache Spark que se ejecuta en el clúster y sobre el cual nos proporciona una abstracción SQL.
- C. Mediante las consultas SQL traducidas por Impala al *metastore* de Hive.
- D. Ninguna de las anteriores es correcta.

Ingeniería para el Procesado Masivo de Datos

---

## Tema 8. Cloud computing I

# Índice

[Esquema](#)

[Ideas clave](#)

[8.1. Introducción y objetivos](#)

[8.2. Introducción a cloud computing](#)

[8.3. Ventajas del cloud computing](#)

[8.4. Tipos de nube y servicios en la nube](#)

[8.5. Casos de uso de los servicios en la nube](#)

[8.6. Microsoft Azure](#)

[A fondo](#)

[Cloud computing bible](#)

[Conceptos, tecnología y arquitectura del cloud computing](#)

[Documentación en línea de Microsoft Azure](#)

[Test](#)

# Esquema

CLOUD COMPUTING		MICROSOFT AZURE	
Ventajas:	<ul style="list-style-type: none"> <li>- Virtualización y abstracción de recursos</li> <li>- Alta disponibilidad</li> <li>- Escalabilidad</li> <li>- Elásticidad</li> <li>- Escala global</li> <li>- Tolerancia a fallos</li> <li>- Reducción de costes</li> </ul>	Almacenamiento	<p>Blob Storage</p> <p>Almacenamiento objetos</p> <p>Virtually limitado</p>
Computación	<p>Windows Virtual Machines</p> <p>Linux Virtual Machines</p> <p>Instancia ~ computo ~ servidor</p> <p>Ejecución</p> <p>Capacidad computo</p>	Almacenamiento	<p>SQL Database</p> <p>Base de datos relacional</p> <p>Orientada a OLTP</p>
Modelos despliegue:	<ul style="list-style-type: none"> <li>- Nube pública</li> <li>- Nube privada</li> <li>- Nube híbrida</li> </ul>	Container Instances	<p>File storage</p> <p>Almacenamiento orientado a ficheros</p>
Modelos servicio:	<ul style="list-style-type: none"> <li>- IaaS</li> <li>- PaaS</li> <li>- FaaS</li> <li>- SaaS</li> </ul>	Kubernetes Instances	<p>Queue Storage</p> <p>Almacenamiento orientado a colas mensajes</p>
Big data & analítica		Machine learning & IA	<p>Data Lake Storage</p> <p>Almacenamiento big data</p>
Bases de datos		Machine Learning Service	<p>Synapse Analytics</p> <p>Procesado masivo paralelo (MPP) de datos hasta petabytes</p>
Machine learning & IA		Machine Learning Studio	<p>Stream Analytics</p> <p>Streaming gestionado</p>
Machine learning & IA		HDI insight	<p>CosmosDB</p> <p>No-relacional</p> <p>Soporta MongoDB, Cassandra</p>
Machine learning & IA		Datatrakcs	<p>Table Storage</p> <p>Almacenamiento NoSQL clave/valor</p>
Machine learning & IA		...	<p>Disks</p> <p>Almacenamiento orientado a bloque</p>

## 8.1. Introducción y objetivos

En los temas previos, hemos visto diferentes tecnologías que componen el ecosistema de código libre de *big data*. Hemos trabajado con ellas de forma abstracta, sin incidir realmente sobre dónde están instaladas. La opción más directa es, probablemente, instalarlas en un clúster de equipos locales, alojados en el centro de datos o de procesado de la empresa o establecimiento que usa el clúster. Sin embargo, esta posibilidad no está exenta de inconvenientes, asociados con la complejidad de administrar estos equipos. Con el objetivo de simplificar dicha gestión, se puede acudir a lo que se conoce como servicios en la nube o *cloud computing*.

En las páginas siguientes, se desarrollarán los conceptos relacionados con el *cloud computing*: qué es, qué problemas ataca, por qué cualidades se caracteriza o cuáles son algunos de sus casos de uso principales. Además, en este tema y sucesivos se analizarán de cerca las tres plataformas principales de *cloud computing* hoy en día: Amazon Web Services (AWS), Microsoft Azure y Google Cloud Platform (GCP). Con todo ello, los objetivos que se persiguen son los siguientes:

- ▶ Entender el concepto de *cloud computing*.
- ▶ Comprender de dónde surgen los servicios en la nube y, con ello, qué problemas pretenden atacar.
- ▶ Identificar las principales ventajas del uso de *cloud computing*.
- ▶ Diferenciar los modelos de despliegue de la nube.
- ▶ Diferenciar los distintos modelos de servicios en la nube: IaaS, PaaS, SaaS.
- ▶ Conocer los casos de uso más típicos de *cloud computing*.

- ▶ Explorar los servicios y el funcionamiento de uno de los proveedores de servicios en la nube: Microsoft Azure.

## 8.2. Introducción a cloud computing

La computación en la nube o *cloud computing*, como es más comúnmente conocida, son una serie de servicios relacionados con la computación que se proporcionan a través de Internet. De hecho, la referencia a la nube proviene de la representación más habitual de la red de redes como una nube. Este concepto concentra toda la complejidad subyacente de Internet (protocolos y estándares), necesaria para proporcionar servicios como las páginas web, los buscadores o el correo electrónico. Algo que nos debería ir dando alguna pista de lo que van a significar estos servicios de computación en la nube: una **abstracción de la complejidad de los sistemas, protocolos y tecnologías que existen por debajo y que interactúan para proporcionar toda una serie de servicios**, no solo web en este caso, sino más variados, como veremos más adelante. Cabe considerar también que Internet fue concebido desde el principio como una red de redes redundante y que pudiera hacer frente a fallos. Esta característica permite que los recursos a los que da acceso se pueden **escalar de forma masiva**, otra de las características de los servicios de *cloud computing*.

Se suele decir que son servicios de computación en la nube, pero lo cierto es que van más allá de la mera computación. El abanico de servicios ofrecidos es muy diverso: almacenamiento, bases de datos, redes, *software*, servicios de analítica de datos, inteligencia artificial, servicios para Internet de las cosas (*Internet of Things, IoT*) o para aplicaciones móviles, entre muchos otros. Todos ellos se ofrecen con el objetivo de proporcionar la oportunidad de innovar de forma más rápida, acceder a recursos flexibles y ofrecer economías de escala. Una de las principales ventajas, por ejemplo, es que generalmente **el usuario paga solo por los servicios en la nube que usa**, lo que ayuda a reducir los costes operacionales, tener una infraestructura más eficiente y escalar el sistema según las necesidades de cada momento. Es decir, es posible empezar con sistemas pequeños, para probar cómo

funcionan, y escalarlos después al ritmo que sea necesario.

De forma más técnica, el *cloud computing* se refiere a aplicaciones y servicios que se ejecutan en una red distribuida usando recursos virtualizados, y a los que se accede mediante protocolos y estándares comunes de Internet y redes (de los cuales también hacen uso). Como hemos señalado, los recursos que emplean son virtuales y los detalles de los sistemas físicos se abstraen (esconden) del usuario. Por ejemplo, si accedemos a un servicio de computación, nos dará la sensación de que estamos conectados a un ordenador o servidor físico concreto, pero lo que realmente habrá por debajo es una colección de servidores cuya capacidad de cómputo se «agrega»; en este sentido, lo que se nos ofrece una «porción» de dicha capacidad de cómputo global mediante un servidor virtual: no es un equipo físico real, sino una parte de la capacidad de computación disponible, que se nos presenta dando la ilusión de ser un equipo real.

El *cloud computing* se ofrece generalmente de forma comercial, a través de un proveedor que nos proporciona servicios que teníamos previamente en nuestros propios ordenadores o servidores locales (lo que se conoce como *on-premises*), con la única diferencia de que estos ahora estarán alojados en sus servidores y accederemos a ellos a través de Internet. Como decíamos, el proveedor ofrece estos servicios bajo demanda, en lo que se conoce como *pay-per-use* (es decir, se paga solo por lo que se usa). Esto es una gran ventaja porque, si se necesita ampliar la infraestructura IT, esto es, si en cierto momento necesitamos más capacidad de cómputo, de almacenamiento o de tráfico de datos por un aumento en la demanda de la aplicación que tenemos entre manos, no es necesario comprar todo el *hardware* para lo que puede ser un momento puntual, con la gran inversión que supone de antemano, tanto económica como de recursos para instalar, configurar y mantener dicha infraestructura.

Otra perspectiva de los servicios en la nube va más allá de albergarlos en servidores de terceros y acceder a ellos a través de Internet. En el mundo tecnológico, existen

una serie de acciones que, a la vez que importantes, son también muy comunes para una gran cantidad de negocios y aplicaciones. Toda aplicación o servicio necesita almacenamiento, gestión de la identidad, redes o cierto grado de cómputo. Los servicios en la nube permiten eliminar los trabajos (y, por tanto, los recursos humanos y económicos involucrados) que conllevan este mantenimiento, los cuales, a pesar de su vital importancia, no aportan ningún elemento diferenciador en el negocio, sino que se trata sencillamente de elementos necesarios (en inglés, se suelen denominar *utilities*). De esta forma, los recursos pueden enfocarse en los aspectos que aportan valor en lugar de emplear el tiempo en gestionar estas otras tareas que son necesarias para toda aplicación, pero no diferenciadoras.

Los dos conceptos que posibilitan la existencia de los servicios en la nube, de los cuales se ha hablado previamente en los ejemplos comentados, son:

- ▶ **Abstracción.** La computación en la nube abstrae los detalles de implementación del sistema que proporciona el servicio. Las aplicaciones o servicios se ejecutan en unos sistemas físicos de los que no se conocen las especificaciones, los datos se almacenan en localizaciones desconocidas, la administración de los sistemas se subcontrata a terceros y el acceso a los servicios por parte de los usuarios es ubicuo (esto es, no depende de su localización ni del dispositivo desde el que accedan). Como decíamos anteriormente, al igual que asemejamos Internet con una nube que proporciona una serie de servicios web de los que tampoco sabemos los detalles de infraestructura y sistema que los mantienen en pie, también los servicios de computación, de los que no tenemos detalles a bajo nivel, se representan con este símbolo.
- ▶ **Virtualización.** Los servicios de *cloud computing* virtualizan sistemas mediante la compartición de recursos. Todos los servicios nombrados anteriormente (computación, almacenamiento, bases de datos, etc.) se proporcionan bajo demanda, desde una infraestructura centralizada, de forma que los costes están determinados según el consumo, todos los servicios son usados por múltiples

usuarios a la vez y los recursos se pueden escalar de forma muy ágil.

Un ejemplo de servicio *cloud computing* es el mostrado en la figura 1: una consola de intérprete de comandos de Linux. Pero ¿esta consola se está ejecutando en nuestro ordenador personal o servidor de la empresa, o lo está haciendo en una instancia de cómputo (es decir, el equivalente a un servidor), en la infraestructura de un proveedor de servicios en la nube? A simple vista, no se podría saber, ya que lo que tenemos es un intérprete de línea de comandos, con su funcionalidad asociada, que está ejecutándose en un sistema operativo concreto y en un sistema específico. Aunque podría ser nuestro propio ordenador o servidor, en realidad este intérprete se está ejecutando en una instancia de cómputo EC2 proporcionada por Amazon Web Services, uno de los principales proveedores de servicios en la nube. Dicha instancia abstrae la información del equipo concreto en que se está ejecutando y una miríada de detalles más, y proviene de la virtualización de los recursos que AWS tiene en sus centros de datos, de los cuales nos cede una porción en forma de un servicio de cómputo en la nube.

```
Dropbox — ec2-user@ip-172-31-23-158:~ — ssh -i hadoop_testing.pem ec2-...
or directory
[ec2-user@ip-172-31-23-158 ~]$ ls
HDP_2.6.5_deploy-scripts_180624d542a25.zip  sandbox
assets                                         sandbox-flavor
docker-deploy-hdp265.sh                      test.txt
enable-native-cda.sh                          userid_profile.txt
[ec2-user@ip-172-31-23-158 ~]$ cd sandbox
[ec2-user@ip-172-31-23-158 sandbox]$ cd ..
[ec2-user@ip-172-31-23-158 ~]$ ls
HDP_2.6.5_deploy-scripts_180624d542a25.zip  sandbox
assets                                         sandbox-flavor
docker-deploy-hdp265.sh                      test.txt
enable-native-cda.sh                          userid_profile.txt
[ec2-user@ip-172-31-23-158 ~]$ mkdir my_dir
[ec2-user@ip-172-31-23-158 ~]$ cd my_dir
[ec2-user@ip-172-31-23-158 my_dir]$ ls
[ec2-user@ip-172-31-23-158 my_dir]$ cd ..
[ec2-user@ip-172-31-23-158 ~]$ ls
HDP_2.6.5_deploy-scripts_180624d542a25.zip  sandbox
assets                                         sandbox-flavor
docker-deploy-hdp265.sh                      test.txt
enable-native-cda.sh                          userid_profile.txt
my_dir
[ec2-user@ip-172-31-23-158 ~]$
```

Figura 1. Intérprete de comandos: ¿en un equipo local o en un servicio en la nube?

### 8.3. Ventajas del cloud computing

La computación en la nube supone un cambio bastante drástico respecto a la forma tradicional en que se han venido gestionando los recursos de almacenamiento, computación y, en general, tecnológicos. Sin embargo, han sido las ventajas que se derivan de esta transformación las que han hecho que muchas compañías e individuos se hayan decantado ya por el uso del *cloud computing* para sustituir o complementar su despliegue tradicional de servicios (también conocidos como *on-premises*, es decir, en servidores y equipos propios). A continuación, se describen las principales ventajas asociadas a los servicios de *cloud computing*:

- ▶ **Disponibilidad.** La disponibilidad de un sistema se define como el porcentaje de tiempo que este puede dar respuesta a las peticiones de los usuarios. Un ejemplo habitual de lo que se conoce como alta disponibilidad (uno de los aspectos más deseados en una aplicación) es el conocido 99,99 %. Esta cifra significa que la aplicación garantiza su continuo y correcto funcionamiento, salvo un máximo de cuatro minutos al mes. Para poder proporcionar esta disponibilidad, acordada generalmente por contrato, es necesario replicar los sistemas a escala global y que estos se caractericen por una alta tolerancia frente a fallos, así como por una rápida recuperación si estos ocurren. En otras palabras: los servicios de *cloud computing* son capaces de incrementar la disponibilidad de las aplicaciones que alojan, ya que, gracias a la virtualización de los recursos y la disponibilidad de los mismos a escala global, hacen posible su fácil replicación.
- ▶ **Escalabilidad.** La escalabilidad se puede entender como la capacidad de un sistema de crecer para responder a un incremento de la demanda de peticiones. Esto implica, generalmente, adquirir nuevos equipos con unas capacidades de cómputo, almacenamiento y red mayores; instalarlos y configurarlos; hacer *backup* de la información almacenada hasta el momento, transferirla al nuevo sistema y restaurarla en él, y hacer que todo funcione como en el sistema antiguo, pero

adaptado a las características de mayor capacidad del nuevo. Este proceso conlleva un tiempo nada despreciable y, por tanto, es un problema de cara a alcanzar la alta disponibilidad de la que se hablaba en el punto anterior. Los servicios de *cloud computing* facilitan enormemente esta tarea al proporcionar escalabilidad a las aplicaciones que gestionan, gracias de nuevo a la virtualización. Además, lo hacen de forma transparente para el usuario, gracias a la abstracción de la infraestructura que existe por debajo.

- ▶ **Elasticidad.** No solo es importante poder aumentar la capacidad del sistema según lo hace la carga de trabajo, sino también lo contrario: disminuirla cuando la demanda de carga de trabajo se reduce. De esta forma, se minimiza el desperdicio de recursos, que lleva inevitablemente un sobrecoste asociado. De nuevo, es fácil para una plataforma de *cloud computing* aportar elasticidad, dado que la virtualización y la abstracción permiten ajustar los recursos necesarios tanto para incrementarlos como para reducirlos. Esto posibilita que la mayoría de estos servicios se ofrezcan como una especie de «autoservicio», de forma que incluso grandes recursos de cómputo se pueden suministrar en cuestión de minutos. Esto proporciona a las compañías una gran flexibilidad, ya que elimina o disminuye la necesidad de provisionar adecuadamente de antemano la capacidad del sistema, a veces sin saber realmente el volumen real de carga que va a tener.
- ▶ **Escala global.** Los beneficios de la computación en la nube incluyen la posibilidad de escalar de forma elástica y global. En términos *cloud*, esto significa ser capaz de proporcionar la cantidad adecuada de recursos (ya sea capacidad de cómputo, almacenamiento o ancho de banda de la red) según las necesidades del momento y desde la localización geográfica más adecuada.
- ▶ **Productividad.** Los centros de datos y recursos de computación suelen requerir un trabajo no despreciable de instalación y configuración del *hardware*, actualización y configuración del *software*, y otros trabajos de gestión necesarios a la vez que tediosos. La computación en la nube elimina la necesidad de llevar a cabo muchas de estas tareas, de forma que los equipos técnicos se puedan centrar en otros

objetivos más importantes para las empresas.

- ▶ **Seguridad.** Muchos proveedores de servicios de computación en la nube ofrecen una gran variedad de políticas, tecnologías y controles que refuerzan la seguridad de los servicios proporcionados. De esta manera, ayudan a proteger datos, aplicaciones e infraestructura frente a potenciales amenazas.
- ▶ **Fiabilidad y tolerancia a fallos.** Se refiere a la capacidad del sistema de manejar fallos como cortes de electricidad, cortes en la red, fallos de componentes de hardware (almacenamiento persistente, memoria...). Es decir, cuando ocurre uno de estos fallos, ¿el sistema es capaz de detectarlo y recuperarse rápido, de forma que no afecte a la disponibilidad? La computación en la nube se hace cargo de copias de seguridad, recuperación ante desastres y continuidad de servicio de forma más sencilla y menos costosa, gracias a que los recursos están replicados en múltiples sitios dentro de la red del proveedor en la nube.
- ▶ **Recuperación ante desastres.** Sucesos como cortes de electricidad o de red, incendios o inundaciones pueden comprometer la integridad de un centro de datos. La recuperación frente a desastres es la capacidad que tiene la infraestructura para volver a funcionar normalmente tras la ocurrencia de alguno de estos sucesos dentro de un período de tiempo razonable, así como de recuperar la cantidad de datos perdidos durante el proceso. Es decir, una vez que el sistema es capaz de detectar estos problemas y recuperarse, ¿cómo lo logra? Por ejemplo, si se hacen copias de seguridad cada hora, la pérdida de información será equivalente a los datos generados durante la última hora. Así, la capacidad de recuperación se medirá como el tiempo necesario para cargar los datos de la última copia de seguridad y poner todo el sistema en funcionamiento de nuevo.
- ▶ **Coste.** Hay dos aspectos sobre el coste económico de los sistemas que se ven favorecidos por el uso del *cloud computing*:
  - **Economías de escala.** Es el más obvio y, aplicado al caso que nos ocupa, se traduce en que el coste de la infraestructura IT disminuye con su tamaño. Esto se

debe a que los costes fijos se mantienen y se amortizan entre más unidades: no hay gran diferencia en gestionar 100 o 1000 servidores, y se pueden obtener mejores ofertas al adquirir grandes volúmenes de estos; por tanto, el coste final es menor. Es decir, grandes proveedores como Microsoft, Amazon o Google, al adquirir equipos en grandes volúmenes, van a obtener mejores ofertas que una empresa de menor tamaño, que comprará equipos en un volumen mucho menor.

- **Tipos de gasto.** El segundo aspecto tiene que ver con el tipo de gasto que supone comprar infraestructura directamente frente a contratar servicios de computación en la nube. Cuando se adquieren **equipos físicos**, estos gastos se consideran «gastos de capital» (o *capital expenses*, **CapEx**), es decir, dinero fijo invertido de antemano en elementos, como servidores, que van a reportar, potencialmente, beneficio con el tiempo (*return of investment*). Estos costes, que no son nada despreciables, hay que afrontarlos de antemano, sin saber si realmente se obtendrán esos beneficios esperados de la inversión, lo que supone un riesgo importante que se debe tener en cuenta.

Por otro lado, en los **servicios de computación en la nube**, al pagar según lo que se consume, se consideran operacionales (*operation costs, OpEx*), es decir, gastos del día a día para mantener el negocio en funcionamiento (como la luz, la línea telefónica o el alquiler de una oficina). Estos gastos se derivan del beneficio obtenido mes a mes y, en general, se prefieren frente a los gastos de capital, ya que son variables y no hay que afrontarlos de antemano. Dicho de otro modo, no es necesario hacer una importante inversión inicial sin tener siquiera asegurada su rentabilidad. La computación en la nube elimina los gastos de capital derivados de la compra de servidores, *racks*, redes, climatización, etc., es decir, de toda la infraestructura *hardware* y *software*. Y con ello la consiguiente necesidad de contratar a expertos que la pongan en funcionamiento y la gestionen. Además, evita que existan estos costes para cubrir aumentos de demanda puntuales que no se mantienen en el tiempo y que conllevan un derroche posterior de recursos. La compañía solo paga por el servicio que se consume en cada momento.

## 8.4. Tipos de nube y servicios en la nube

Existen diferentes modelos mediante los que clasificar los sistemas de computación en la nube. Principalmente, podemos utilizar dos tipos de clasificación: por modelos de despliegue y por modelos de servicios.

La figura 2 muestra la relación entre estos dos modelos y algunas características de la computación en la nube. Básicamente, en la base de la figura tenemos el conjunto de recursos (ya sean de computación, de almacenamiento o de cualquier otro tipo) que, mediante su virtualización, son compartidos por todos los usuarios de la nube. Dichos recursos se pueden ofrecer como servicios de infraestructura, plataforma o *software*, cuyas diferencias veremos más adelante. En un escalón por encima, se encuentra el modelo de despliegue de la nube que proporcionan estos servicios, que puede ser pública, privada e híbrida. Veremos también las diferencias entre estos modelos más adelante.

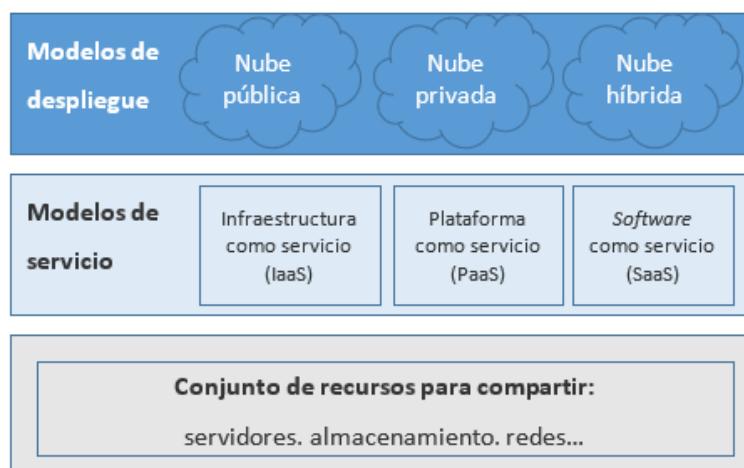


Figura 2. Relación de los modelos de nube.

### Modelos de despliegue de nube

El modelo de despliegue se refiere a la localización donde se aloja la nube, es decir:

dónde se despliega y quién puede usarla. Encontramos tres modalidades:

- ▶ **Nube pública.** Las nubes públicas están alojadas y operadas por un proveedor externo de servicios en la nube, que proporciona sus recursos, como servidores y almacenamiento, a través de Internet. Los servicios en la nube que ofrecen los grandes proveedores (Amazon Web Services, AWS; Google Cloud Platform, GCP, y Microsoft Azure) son ejemplos de nubes públicas. En estos casos, dichos proveedores se convierten en propietarios y gestores de todo el *hardware*, el *software* y la infraestructura auxiliar. Por su parte, los usuarios comparten todos estos elementos de almacenamiento, cómputo y red con otras organizaciones, y acceden a sus servicios y gestionan su cuenta a través de una página web. Cualquiera con una tarjeta de crédito puede contratar servicios de nube pública.
- ▶ **Nube privada.** Se refiere a que los recursos de computación en la red los usa exclusivamente una única empresa u organización. Una nube privada puede estar físicamente en la propia compañía (*on-premises*) o con un proveedor de servicios *cloud* (*off-premises*). De igual forma, la nube puede estar gestionada por el servicio IT de la propia compañía, o bien esta puede decidir pagar un servicio adicional a un proveedor para que albergue y gestione la nube privada. De una u otra manera, todos los servicios e infraestructuras de una nube privada estarán conectados bajo una red privada y no a la red pública de Internet. En caso de estar albergada en otra empresa o gestionada por esta, su servicio de nube privada solo se obtiene mediante contratos específicos, bajo discusión con un representante de la compañía oferente.
- ▶ **Nube híbrida.** Las nubes híbridas combinan una nube pública con una privada, mediante tecnologías que permiten compartir datos y aplicaciones entre ellas. De este modo, proporcionan a las compañías mayor flexibilidad y más opciones de despliegue, y ayudan a optimizar la infraestructura ya existente, así como la seguridad y conformidad con diferentes normas, como la ley general de protección de datos de Europa (GDPR).

Una opción habitual es tener la aplicación a desplegar en una nube privada y aprovechar los servicios de nube pública para escalar cuando es necesario. Combina infraestructura *on-premises* (es decir, el centro de datos propio de la compañía o una nube privada) con una nube pública. Muchas organizaciones eligen esta opción debido a restricciones del negocio como cumplir normativas de gobierno y privacidad de datos, para aprovechar la inversión de su propia infraestructura o para cumplir con los requisitos de baja latencia que necesitan.

La nube híbrida está evolucionando para incluir lo que se denominan *edge workloads* o *edge computing*. *Edge computing* posibilita llevar la computación más cerca de, por ejemplo, dispositivos IoT, donde los datos residen, de forma que la latencia es más baja porque el tiempo desde que el dispositivo envía los datos recabados hasta que se reciben en el sistema de procesado es menor. Por otro lado, la nube híbrida permite que, cuando la demanda de recursos de cómputo y procesado fluctúan (por ejemplo, incrementos de demanda por época de rebajas, Navidad o de declaración de impuestos...), sea posible escalar la infraestructura *on-premises*, mediante el uso de la nube pública de un modo transparente y sin dar acceso a terceros al conjunto completo de datos. De esta forma, las empresas se benefician de la capacidad de escalado, pero mantienen los datos altamente sensibles dentro de sus centros de datos para cumplir con las regulaciones vigentes.

## Modelos de servicios en la nube

Existen diferentes formas de ejecutar código en la nube. Por ejemplo, alguien que tiene en su empresa un servidor Windows puede contratar otro servidor Windows en la nube y ejecutar su código en él. Este es el ejemplo más sencillo de migrar un *software* a la nube. Pero existen más alternativas o modelos de servicios en la nube, que definen los diferentes niveles de responsabilidad de los proveedores de *cloud computing* y de los usuarios. En el fondo, la nube se puede ver como la frontera entre la red, la gestión y las responsabilidades del proveedor de servicios y del cliente. Para hacer esta barrera más concreta y definir dónde se encuentra, los proveedores

ofrecen diferentes servicios, que engloban más o menos responsabilidades, lo que da lugar a los diferentes modelos de servicio.

Los modelos principales son los siguientes: infraestructura como servicio (*Infrastructure as a Service*, IaaS), plataforma como servicio (*Platform as a Service*, PaaS), *serverless* y software como servicio (*Software as a Service*, SaaS). En ocasiones, estas modalidades son denominadas la «pila (stack)» o «modelo SPI» de la computación en la nube, ya que unas se construyen sobre otras. Existen modelos adicionales (como *Storage as a Service* o *Identity as a Service*, entre otros muchos), de los que vamos a ver con algo más de detalle *Function as a Service* (FaaS), también conocido como *serverless computing* o servicios *serverless*, dada su gran disponibilidad actualmente.

- ▶ **Infrastructure as a Service (IaaS).** Esta es la categoría más básica de servicios en la nube. Con IaaS, el usuario «alquila» la infraestructura que necesita a un proveedor de servicios en la nube, en una modalidad de pago por consumo (*pay-as-you-go* o *pay-per-use*): servidores, máquinas virtuales (VM), almacenamiento, redes, sistemas operativos, balanceadores de carga, *firewalls*... En definitiva, todos los componentes que habría que comprar, instalar, configurar, etc., en un centro de datos propio. Toda esta infraestructura como servicio, sin embargo, no hace nada por sí misma, sino que es el usuario quien tendrá que instalar los programas que desee usar, el código del sistema que se va a desplegar, etc.
- ▶ **Platform as a Service (PaaS).** Plataforma como servicio hace referencia a servicios de *cloud computing* que proporcionan un entorno bajo demanda para desarrollo, pruebas, despliegue y gestión de aplicaciones *software*. PaaS está diseñado para facilitar a los desarrolladores, por ejemplo, la creación de una página web o aplicación móvil, sin preocuparse de preparar o gestionar la infraestructura de servidores, almacenamiento, red y bases de datos necesaria para el desarrollo. Por ejemplo, se puede diseñar una aplicación web sin más acciones que subir el código necesario y la configuración requerida, y el proveedor de PaaS se encargará de todo

lo demás. El usuario, por su parte, ni tan siquiera sabe en qué *hardware* se está ejecutando su aplicación web; sencillamente habrá contratado un nivel de servicio (SLA, *service level agreement*) que le asegurará cierto grado de disponibilidad y carga máxima, entre otros, pero no conocerá la infraestructura que hay por debajo para proporcionar dicha aplicación web.

- ▶ **Computación serverless o Function as a Service (FaaS).** Este modelo se solapa en cierta manera con PaaS, en el sentido de que se centra en construir funcionalidades de aplicaciones sin invertir tiempo continuamente en gestionar los servidores y la infraestructura requeridos. El proveedor de la nube se encarga de la configuración, planificación y provisión de recursos necesarios, así como de la gestión del servidor. La principal diferencia se centra en que la arquitectura *serverless* está orientada a eventos, es decir, las funcionalidades se ejecutan solo cuando ocurre un evento o *trigger* específicos. No almacena datos y el entorno se detiene durante el tiempo en el que la función no se ejecuta, para no incurrir en gastos en ese intervalo.
- ▶ **Software as a Service (SaaS).** *Software* como servicio es un modelo para proporcionar aplicaciones *software* a través de Internet, bajo demanda y generalmente bajo una suscripción. Con SaaS, los proveedores de *cloud computing* alojan y gestionan dichas aplicaciones y la infraestructura que hay por debajo, y se encargan de todo el mantenimiento, como actualizaciones de *software* y de seguridad. Los usuarios (no los desarrolladores) se conectan a las aplicaciones también a través de Internet, normalmente desde un navegador en su teléfono móvil, tableta u ordenador. Se trata de aplicaciones cuyo código ya está desarrollado e instalado, y el usuario no puede cambiar nada de las funcionalidades proporcionadas, solo configurar ciertas opciones y utilizarlo. Ejemplos de SaaS incluyen aplicaciones de conferencia *online* (Zoom), seguimiento de la jornada laboral, nóminas y contabilidad, Google Drive, Office365 o Salesforce, entre otras muchas.

En la figura 3, se muestran las fronteras que delimitan qué parte gestiona el

proveedor de servicios de *cloud* y cuál el usuario en cada uno de los modelos descritos, respecto a un despliegue tradicional, es decir, a una aplicación que se ejecuta en los servidores alojados y gestionados por el propio usuario:

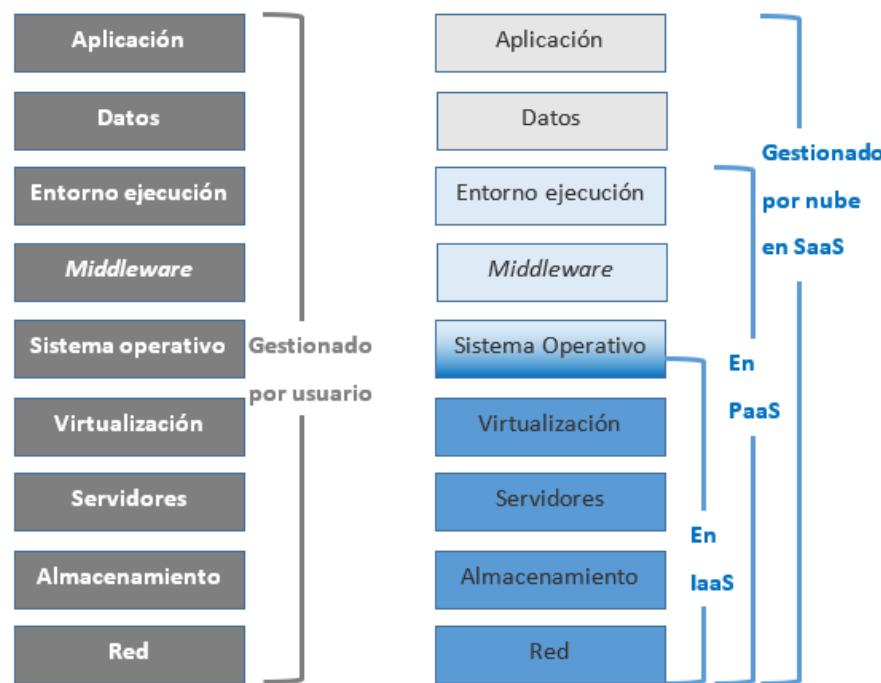


Figura 3. Gestión de recursos en un sistema *on-premises* (izquierda) y en un sistema en la nube, dependiendo del modelo de servicio (derecha).

## 8.5. Casos de uso de los servicios en la nube

Probablemente cualquier persona con un ordenador conectado a Internet esté usando servicios en la nube sin darse cuenta. Por ejemplo, muchos de ellos están detrás de algunos servicios de correo electrónico o de edición de documentos en línea, plataformas de vídeo (series, películas, documentales...) o de música en *streaming*, juegos *online* o en el teléfono móvil, o servicios de almacenamiento y edición de fotos. Los primeros servicios de computación en la nube datan de hace apenas una década, pero ya entonces estaban al servicio de organizaciones diversas, desde pequeñas *startups* hasta corporaciones globales, pasando por agencias gubernamentales o sin ánimo de lucro. A continuación, se mencionan algunos de los casos de uso de los servicios de *cloud computing*:

- ▶ **Construir, desplegar y escalar** de forma rápida **aplicaciones** en la nube nativas, ya sean para web, teléfonos móviles o que ofrezcan una API. Esto es posible gracias a tecnologías en la nube también nativas, como contenedores, arquitecturas de microservicios o como comunicación mediante API.
- ▶ **Crear, almacenar y restaurar copias de seguridad.** Las plataformas de *cloudcomputing* permiten proteger datos de una forma duradera (gracias a su redundancia, ofrecen resistencia a fallos del *hardware*) y más eficiente en costes, aprovechando la escala masiva de estos servicios. Basta con transferir los datos a través de Internet a un sistema de almacenamiento en la nube, que es accesible desde cualquier localización y dispositivo.
- ▶ **Streaming de vídeo y audio.** Las plataformas de computación en la nube permiten a la audiencia conectar con sus recursos de vídeo o audio en *streaming*, gracias a redes de distribución global.
- ▶ Proporcionar **software bajo demanda**. Con la modalidad SaaS, se puede acceder a las últimas versiones de *software* y actualizaciones en cualquier momento y lugar.

- ▶ **Construir y probar aplicaciones.** Como se comentaba anteriormente, una de las grandes ventajas de la computación en la nube es la reducción de costes y del tiempo de desarrollo mediante el uso de la infraestructura que proporciona la nube, que puede ser escalada, tanto para incrementar como para reducir la capacidad según sea necesario.
- ▶ **Análisis de datos.** *Cloud computing* permite unificar datos entre equipos, divisiones y localizaciones en la nube. También utilizar servicios adicionales de analítica de datos o *machine learning*, para descubrir patrones o convertir la miríada de datos almacenados por una empresa en conocimiento útil y aplicable en la toma de decisiones.

Entre los proveedores de servicios de computación en la nube, podemos encontrar Amazon Web Services, Microsoft Azure, Google Cloud Platform o IBM. Vamos a ver en detalle los tres primeros, ya que son los que ofrecen servicios de nube pública y, por tanto, son accesibles para cualquier persona u organización.

A fin de descubrir los servicios en la nube que proporcionan cada uno de estos proveedores, vamos a analizarlos por grupos clásicos, empezando por los más básicos (como cómputo, almacenamiento y red) y pasando después a explorar otros más especializados. Existe un abanico muy amplio de servicios de alto nivel (desarrollo web, móvil, *devops*, *big data*...); aquí nos centraremos en bases de datos, aplicaciones *big data*, servicios relacionados con *machine learning* e inteligencia artificial, ya que son los más cercanos a las tecnologías estudiadas en esta asignatura. De esta forma, examinaremos tanto servicios básicos sobre los que se podrían instalar las tecnologías de código abierto que hemos visto durante el curso (por ejemplo, se podría instalar HDFS, Spark, Kafka, Hive o Impala en una instancia EC2 de AWS, así como cualquiera de las distribuciones Hadoop estudiadas) como las propuestas equivalentes que ofrecen estos proveedores en modo de pago.

## 8.6. Microsoft Azure

Microsoft Azure es una plataforma de computación en la nube que ofrece un conjunto de servicios en continuo crecimiento. Estos permiten construir soluciones para diferentes problemas de negocio, desde servicios para albergar aplicaciones o páginas web hasta virtualización de servidores en los que ejecutar diferentes tipos de aplicaciones, pasando por servicios de almacenamiento de información, bases de datos, gestión centralizada de cuentas de usuario, inteligencia artificial o servicios de soporte de aplicaciones para *Internet of Things* (IoT).

Para gestionar hasta un centenar de estos servicios, Microsoft Azure dispone de Azure Portal (figura 4), la consola web (como alternativa a la consola de línea de comandos) desde la que gestionar, desplegar y/o monitorizar todos los servicios que tenga contratado un usuario.

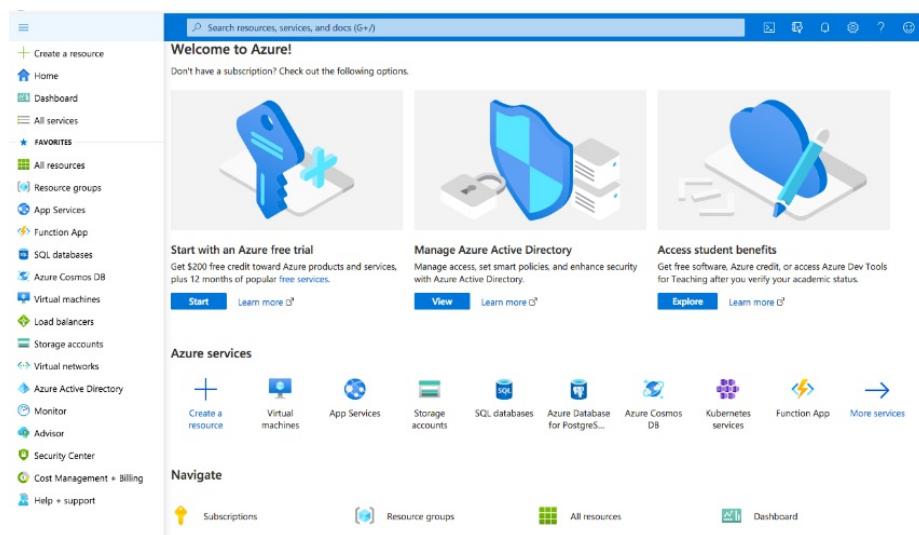


Figura 4. Panel web principal de Microsoft Azure Portal.

Microsoft Azure también dispone de Azure Marketplace, que proporciona al usuario servicios ofrecidos por terceros a través de Microsoft Azure, tales como: plataformas de contenedores *open-source* (como Docker), imágenes de máquinas virtuales (por

ejemplo, distribuciones RedHat), bases de datos más allá de las proporcionadas por el propio Azure, *software* para el desarrollo y despliegue de aplicaciones, herramientas de desarrollo, detección de amenazas, Azure Databricks (para tareas relacionadas con el uso de Spark), Citrix (para servicios de escritorio virtual), etc.

Lo que es ahora Microsoft Azure empezó siendo dos entes por separado. Por un lado, Azure Platform proporcionaba los servicios de infraestructura para alojar aplicaciones web. Por otro, Windows Azure Platform actuaba como un sistema operativo en la nube que proporcionaba una serie de API, a las que se accedía a través del Azure Windows Services Platform API y que permitían hacer uso de los servicios de la plataforma. Por tanto, en Microsoft Azure, se ha venido diferenciando solo entre servicios de infraestructura y de plataforma, además de los servicios de gestión y seguridad, que son transversales (gestión de grupos y roles mediante servicios relacionados con Active Directory, servicios de seguridad, el portal Azure o el Marketplace, entre otros). En la figura 5, se muestra esta clara separación entre servicios de infraestructura y de plataforma, y se incluyen los principales servicios de Microsoft Azure, sin perder nunca de vista que estos son los más usados, aunque la lista completa incluye más de cien. Lo más interesante de la figura es que muestra, de nuevo, la separación entre los servicios IaaS más básicos proporcionados por todos los proveedores (cómputo, almacenamiento, red) y los correspondientes a PaaS (que se muestran encima). Además, en lo que concierne a esta asignatura, es importante que nos fijemos en los servicios englobados en Data y Analytics&IoT.

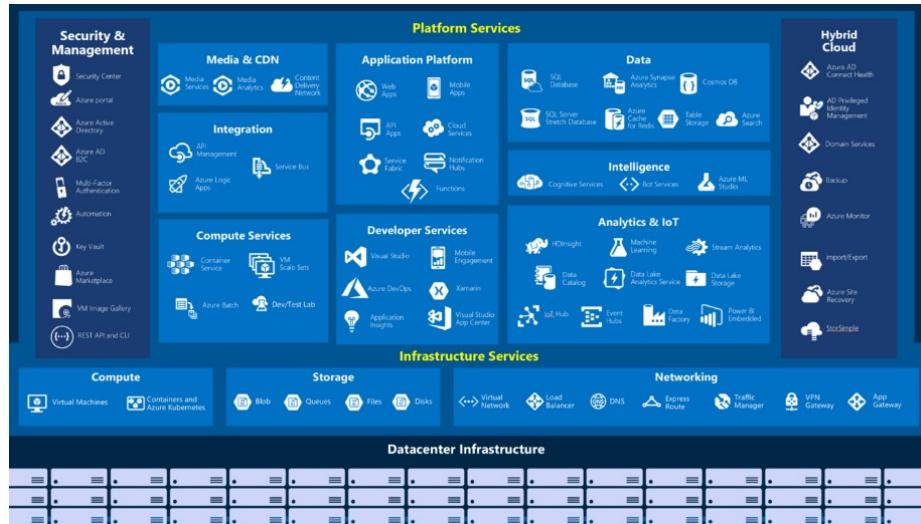


Figura 5. Mapa de los principales servicios en la nube ofrecidos por Microsoft Azure. Fuente: <https://docs.microsoft.com/en-us/learn/modules/welcome-to-azure/3-tour-of-azure-services>

A continuación, vamos a describir los principales servicios, tanto de infraestructura, sobre los que podríamos instalar las tecnologías *big data* que hemos visto en el resto del curso, como los de plataforma, que ya nos ofrecen servicios análogos a las tecnologías *big data* estudiadas, pero que son gestionados parcial o completamente por Microsoft Azure. Estos últimos están englobados en los grupos de datos (*Data*), inteligencia (*Intelligence*) y analítica (*Analytics*) de la figura 5. Además, antes de proceder a describirlos, es conveniente entender conceptos transversales como, por ejemplo, las regiones y zonas de disponibilidad de Microsoft Azure, así como algunos servicios de seguridad y gestión.

## Conceptos y servicios transversales

Microsoft Azure está compuesto a nivel físico por una gran colección de centros de datos repartidos por todo el mundo. Así, cuando un usuario contrata un servicio como una base de datos o una máquina virtual, está usando los equipos físicos presentes en uno o más de estos centros de datos. Para organizarlos y poner a disposición de los usuarios todos los recursos que engloban, Microsoft Azure clasifica todos sus recursos por **regiones**. Una región es un área geográfica que contiene, **al menos**,

**un centro de datos** (aunque puede contener varios, conectados entre sí por redes de baja latencia). Cuando un usuario quiere contratar un servicio de los proporcionados por Microsoft Azure, debe elegir la región desde donde se servirá dicho servicio. Hay que tener en cuenta que no todos los servicios o todas sus características están disponibles en absolutamente todas las regiones. Por tanto, antes de elegir una región, hay que estudiar aspectos como la disponibilidad del servicio, el coste (que puede diferir entre regiones), la latencia o los aspectos legales (por ejemplo, algunas normativas de privacidad de datos no permiten almacenar datos fuera de cierto país o del entorno donde estos fueron creados). La figura 6 muestra las regiones disponibles a fecha de junio de 2020, aunque cabe recordar que estas regiones están en continua expansión.



Figura 6. Regiones disponibles en Microsoft Azure a fecha de junio de 2020.

Tal y como se comentaba previamente, cada región tiene, al menos, un centro de datos, aunque generalmente posee más. Cada centro separado físicamente (e independientes en términos de energía, climatización y red) dentro de una región se denomina ***availability zone (AZ)***. Representa una frontera que aísla recursos, de forma que, si una AZ falla, los servicios proporcionados por la región no se detengan (si están replicados en otras AZ). En general, cada región tiene varias AZ; en muchos casos, al menos tres, aunque existen algunas excepciones. Si la aplicación de un

usuario requiere alta disponibilidad, será necesario especificar qué recursos de los que componen la aplicación necesitan replicación en diferentes AZ para hacer frente a posibles fallos. Es importante determinar estos aspectos, ya que esta replicación supone un coste adicional.

Otro servicio transversal para cualquier aplicación en la nube es la **seguridad**. A este respecto, el primer lema, tanto de Microsoft Azure como de otras plataformas, es que la seguridad es **una responsabilidad compartida**. Si bien el proveedor de servicios en la nube gestiona la infraestructura en diferentes grados (recordemos las diferencias entre IaaS, PaaS y SaaS) y es, por tanto, su competencia la seguridad en esos niveles, el usuario nunca queda totalmente libre de responsabilidad. Por ejemplo, cuando se contratan servicios IaaS, este será el encargado de instalar las actualizaciones de seguridad que van estando disponibles, así como de configurar la red subyacente para que sea segura. En el caso de contratar PaaS, el usuario deja en manos del proveedor ciertas competencias de seguridad relacionadas con la parte IaaS (como la que se acaba de comentar), pero sigue siendo responsable de que la aplicación no se use de forma indebida. Independientemente del nivel del servicio contratado (IaaS o PaaS), **el usuario siempre va a tener la responsabilidad de la seguridad concerniente a:**

- ▶ Los **datos**, ya estén almacenados en bases de datos, en discos asociados a máquinas virtuales o en almacenamiento en la nube. Siempre será responsabilidad del usuario el uso que se haga de ellos, así como su acceso. De especial importancia es tener en cuenta regulaciones como GDPR en cuanto a confidencialidad, integridad y disponibilidad de los datos.
- ▶ Las **cuentas de los usuarios** que pueden acceder a los servicios en la nube contratados por la empresa y, por tanto, configurarlos.
- ▶ La **gestión del acceso** a diferentes recursos. Microsoft Azure proporciona varios servicios, como Azure Active Directory, para autenticar usuarios, gestionar roles y

grupos de acceso, y poder configurar, según estos, diferentes permisos de acceso, lectura y/o edición para cada uno de los distintos servicios. Así, un usuario puede tener permiso de lectura de los datos almacenados en cierta base de datos, pero no de edición; o tener acceso de lectura y edición de los datos de una base de datos, pero no tener permiso de acceso a una máquina virtual.

Como cabe esperar, Azure dispone asimismo de un servicio de seguridad que monitoriza los diferentes servicios contratados en busca de eventos o comportamientos sospechosos (*security center*). También de diversas opciones de encriptación (tanto en reposo como en tránsito), gestión de identidad y acceso con Azure Active Directory, protección de las redes y recursos (Azure Firewall, Azure Application Gateway, Azure DDoS Protection, Azure Advance Thread Protection), entre otras muchas.

## Servicios de cómputo

Azure Compute engloba los servicios de cómputo bajo demanda que ofrece Azure para ejecutar aplicaciones en la nube. Estos pueden ser desde procesadores *multicore* hasta supercomputadores, a los que el usuario tiene acceso a través de máquinas virtuales y contenedores. También existe una tercera opción, conocida como *serverless computing*, que permite ejecutar aplicaciones sin necesidad de definir o configurar ningún tipo de infraestructura. E incluso se puede ir un paso más allá en nivel de abstracción con Azure App Service, para proporcionar directamente aplicaciones web como PaaS.

No obstante, lo más interesante en el campo de *big data* son las dos primeras opciones, máquinas virtuales y contenedores, ya que permiten instalar y ejecutar las tecnologías que hemos visto a lo largo de los capítulos previos. De esta forma, el proveedor de servicios en la nube se ocupa de gestionar la infraestructura subyacente, así que el usuario solo tiene que instalar y configurar las tecnologías *big data* que le interesan, y pagar solo por la infraestructura usada o consumida en cada momento. Veamos en qué consiste cada uno de estos tipos de servicios de cómputo.

- ▶ **Azure Virtual Machines.** Este servicio ofrece máquinas virtuales Windows (Azure Windows Virtual Machine) o Linux (Azure Linux Virtual Machine) alojadas en Azure. Estas máquinas virtuales no son más que emulaciones *software* de un ordenador o servidor físico. Es decir, la percepción del usuario es la de estar trabajando con un ordenador Windows o Linux, sobre el que puede instalar y ejecutar cualquier *software* que podría instalar en un equipo local. Sin embargo, lo que hay realmente por detrás es algo más complejo: una virtualización y compartición de los recursos de Azure para proporcionar los recursos de cómputo requeridos por el usuario, pero mostrados de manera que se abstrae toda esta complejidad, como si de un ordenador físico real se tratara.

Azure proporciona un servicio adicional, denominado Azure Virtual Machine Scale Sets, que se corresponden con el mismo concepto de máquina virtual, pero escalable. Es decir, si la máquina virtual que se ha contratado no puede hacer frente a un aumento de la carga computacional, Azure Virtual Machine Scale Sets le permitirá hacerlo al nivel que requiera la nueva demanda.

- ▶ **Azure Container Instances y Azure Kubernetes Service.** Estos servicios permiten ejecutar aplicaciones alojadas en contenedores (ya sean Docker o Kubernetes) en Azure, sin necesidad de proporcionar servidores o máquinas virtuales. Cabe recordar que un contenedor se diferencia de las máquinas virtuales en que son más ligeros, ya que solo virtualizan el entorno *software* (en lugar de los entornos *software* y *hardware*, como sucede con las máquinas virtuales). Es decir, un contenedor agrupa las librerías y componentes que se necesitan para ejecutar una aplicación y utiliza el sistema operativo sobre el que estas se sustentan. Por ejemplo, si tenemos tres contenedores ejecutándose en una máquina sobre un sistema operativo Linux, los tres contenedores, con todos los componentes y aplicaciones que ejecutan, compartirán el mismo kernel de Linux. Los contenedores están orientados a arquitecturas de microservicios, donde es posible aislar las librerías y componentes que necesita cada servicio y ejecutarlos dentro de un contenedor concreto para

facilitar la integración de todos ellos.

## Servicios de redes

Otro de los servicios básicos es el de conexión de los recursos de cómputo y almacenamiento mediante una red. Aquí se incluye una variedad de opciones para conectar los servicios de Azure, tanto entre ellos como con el mundo exterior. Estos conceptos trascienden el temario de esta asignatura, pero, a fin de completar esta panorámica, enumeraremos, sin entrar en detalle, los principales servicios de red que se pueden encontrar al explorar Microsoft Azure:

- ▶ **Azure Virtual Network.** Conecta máquinas virtuales mediante redes virtuales privadas (**VPN**).
- ▶ **Azure Load Balancer.** Balancea la carga de las peticiones que llegan a y salen de Azure entre diferentes máquinas virtuales, aplicaciones o servicios.
- ▶ **Azure Firewall.** Implementa un *firewall* con alta seguridad y disponibilidad.
- ▶ **Azure DDoS Protection.** Protege las aplicaciones alojadas en Azure de potenciales ataques de denegación de servicio.
- ▶ **Azure ExpressRoute.** Permite conectar la infraestructura de una empresa con la de Azure, mediante un enlace dedicado y seguro con gran ancho de banda.
- ▶ **Azure Network Watcher.** Monitoriza y diagnostica posibles fallos en la red.
- ▶ **Azure Traffic Manager.** Distribuye el tráfico de red entre las regiones globales de Azure,

## Servicios de almacenamiento

Todos los servicios de almacenamiento proporcionados por Azure, denominados Azure Storage, comparten ciertas características comunes: alta durabilidad y disponibilidad gracias al uso de redundancia y replicación de datos, seguridad a través de encriptación de datos y acceso mediante roles de usuario, escalabilidad

virtualmente ilimitada, gestión y mantenimiento por parte de Azure, y accesibilidad desde cualquier sitio mediante una API REST o a través también de API en diferentes lenguajes de programación. Azure Storage puede servir como un servicio de almacenamiento o bien como la base de otros servicios (por ejemplo, de HDInsight o de servicios *big data* basados en el ecosistema Hadoop). Azure Storage proporciona cinco tipos principales de servicios de almacenamiento.

- ▶ **Azure Blob Storage.** Este servicio de almacenamiento es el más barato de entre las opciones disponibles y está orientado a grandes objetos (tales como archivos de texto, de vídeo o de imagen, o archivos binarios) y, en general, a datos no estructurados. Solo permite almacenar archivos, pero no realizar consultas sobre los datos que contienen.
- ▶ **Azure File Storage.** Almacenamiento para ficheros que pueden ser accedidos y gestionados como un servidor de ficheros, pero a nivel global, mediante una URL. Está orientado a reemplazar o complementar servidores de ficheros *on-premises* o NAS. También funciona sistema de almacenamiento, al que puede accederse desde diferentes instancias de máquinas virtuales (es decir, como sistema de almacenamiento compartido entre diferentes máquinas virtuales).
- ▶ **Azure Queue Storage.** Almacenamiento de datos de tipo cola para enviar y entregar mensajes entre aplicaciones, de forma asíncrona. Permite desacoplar los distintos componentes de una gran aplicación, que se pueden enviar mensajes entre ellos.
- ▶ **Azure Table Storage.** Almacenamiento NoSQL que guarda datos no estructurados del esquema de manera independiente. El almacenamiento es de tipo clave-valor, indicado para conservar información como datos de usuario para aplicaciones web, libros de direcciones, información de un dispositivo u otros metadatos. Actualmente, este tipo de almacenamiento es parte de Azure Cosmos DB.
- ▶ **Azure Disks.** Representa el nivel más básico de almacenamiento, en bloques, sin estructura alguna de ficheros. Sirve como base de almacenamiento de arranque de

máquinas virtuales, y cada disco solo se puede usar por la máquina virtual a la que está asociado.

## Bases de datos

La mayoría de las aplicaciones con un mínimo de complejidad necesitan una base de datos que respalde la información que manejan y permita realizar búsquedas rápidas de la información requerida en cada momento. Azure proporciona diferentes servicios de base de datos para almacenar una gran variedad de tipos y volúmenes de información, todas ellas con conectividad global y alta disponibilidad. Entre Algunas de las que podemos encontrar son:

- ▶ **Azure Cosmos DB.** Base de datos distribuida a nivel global, caracterizada principalmente por su baja latencia y alta disponibilidad (99,999 %). Este servicio soporta diferentes bases de datos a través de distintas API: SQL API, MongoDB API o Cassandra API, entre otras. Para escribir los datos en la base de datos correspondiente, existen diferentes opciones, desde servicios específicos de migración de datos (como Azure Data Factory), pasando por el uso de la API del propio Azure Cosmos DB hasta subir y editar ficheros.
- ▶ **Azure SQL Database.** Azure ofrece una base de datos relacional SQL, totalmente gestionada por ella. Está orientada a consultas transaccionales en modo *batch* (OnLine Transaction Processing, OLTP). Esta opción es una alternativa a instalar, por ejemplo, Microsoft SQL Server en una instancia de máquina virtual de Azure, en cuyo caso sería el usuario el que tendría que gestionar la instalación, la configuración y el mantenimiento del servidor. La característica principal de Azure SQL Database es la capacidad de escalar, tanto para incrementar como para disminuir, la capacidad de servir consultas OLTP a la base de datos. Además, también proporciona optimizaciones, disponibilidad global y opciones de seguridad avanzadas (encriptación, protección ante amenazas, autenticación mediante Active Directory, entre otras). Al igual que en el caso anterior, Azure da la posibilidad de introducir datos mediante API para diferentes lenguajes de programación, a través de consultas Transact-SQL o mediante el servicio Azure Data Factory.

- ▶ **Azure Database Migration Service.** Este servicio permite la migración de bases de datos *on-premises* o en otras plataformas *cloud* a Azure.

## *Big data y analítica de datos*

Como se mostraba en la figura 5, Azure proporciona un amplio abanico de tecnologías y servicios relacionados con *big data y analytics*, desde almacenamiento orientado a *big data* con Azure Data Lake Storage hasta analíticas basadas en el ecosistema Hadoop o en sus propias tecnologías. Los ejemplos más representativos de las tecnologías incluidas dentro de este grupo son las siguientes:

- ▶ **Azure Data Lake Storage.** Este sistema de almacenamiento está especialmente orientado al almacenamiento de datos masivos de cualquier tipo y tamaño, y es compatible con el ecosistema Hadoop. Recientemente han actualizado esta tecnología, por lo que en muchos sitios se puede ver referenciada como Data Lake Storage Gen2. Se basa en Azure Blob Storage y en un sistema de ficheros jerárquico, todo ello optimizado para poder gestionar las cantidades masivas de datos esperadas en sistemas *big data*. Su compatibilidad con el ecosistema Hadoop hace que pueda servir como capa de almacenamiento para otras herramientas *big data* de Azure, como Azure Databricks o Azure HDInsight, o para el propio Hadoop, pero sin necesidad de tener que cargar los datos en dichas plataformas, sino directamente desde Azure Data Lake Storage Gen2. Como cabe esperar, entre sus principales características, podemos encontrar: escalabilidad ilimitada y redundancia, tanto a nivel de zona de disponibilidad como geográfica, para poder hacer frente a fallos de servidores.
- ▶ **Azure Synapse Analytics.** Este servicio proporciona un sistema para ejecutar análisis a escala masiva, aunando conceptos de *enterprise data warehousing* (EDW) y *big data analytics*, y moviéndolos a la nube. Aprovecha el procesado paralelo masivo (MPP) para ejecutar consultas complejas, en lenguaje Transact-SQL, de forma rápida, en volúmenes de datos hasta el orden de *petabytes*. Cabe remarcar que el almacenamiento (proporcionado, por ejemplo, por Azure Data Lake Storage)

está separado del procesado (realizado por Synapse Analytics), lo que permite escalar los nodos necesarios para realizar las consultas requeridas. Es decir, se establece de nuevo la separación de almacenamiento y procesado que veíamos en el ecosistema Hadoop, de forma que los recursos necesarios para una y otra tarea son independientes y se pueden escalar de acuerdo con las necesidades de cada una.

- ▶ **Azure Stream Analytics.** Esta tecnología se puede ver como el equivalente de Spark Streaming, pero centrada, en este caso, en el procesado de flujos de datos (*streaming*). Es posible su aplicación tanto para procesar eventos en tiempo real (orientado sobre todo a aplicaciones IoT) como para analizar grandes bloques de datos a intervalos constantes y de forma continua (procesando grupos de datos almacenados en Azure Blob Storage). Para facilitar los procesados, proporcionan un lenguaje declarativo que se asemeja a consultas SQL, mediante el cual se pueden crear consultas temporales complejas y llevar a cabo diferentes análisis.
- ▶ **Azure HDInsight.** Este servicio permite procesar cantidades masivas de datos con clústeres Hadoop gestionados por Azure en la nube. Incluye Apache Hadoop (HDFS y MapReduce), Spark, Kafka, HBase, Storm y Apache Hive. La principal ventaja es que todas estas tecnologías están ya preinstaladas y configuradas para poder escalar automáticamente según sea necesario, lo que ahorra al usuario todo el trabajo de gestión y mantenimiento del clúster, a nivel *hardware* y *software*.
- ▶ **Azure Databricks.** Servicio de analítica de datos mediante Apache Spark, que se puede integrar con otros servicios *big data* proporcionados por Azure. Este servicio está tomando el lugar de otro anterior, denominado Data Lake Analytics. El objetivo principal es crear clústeres de Spark completamente gestionados por Azure, así como proporcionar un entorno de desarrollo interactivo a través de *notebooks*, en los que se puede desarrollar código para Spark mediante algunos de los lenguajes de programación que soporta (R, Python, Scala y SQL).

## Inteligencia artificial

En cuanto a servicios de inteligencia artificial, podemos clasificar los que ofrece Microsoft Azure en dos grupos: servicios a bajo nivel, para desarrollar modelos propios de *machine learning*, y servicios a alto nivel, para usar directamente modelos ya entrenados para aplicaciones muy específicas. Veamos ejemplos de cada uno de estos grupos.

- ▶ **Azure Machine Learning Service.** Azure proporciona este entorno en la nube que permite desarrollar, entrenar, evaluar, gestionar y desplegar modelos de *machine learning*. Tiene funcionalidades de generación automática de modelos y de búsqueda automática de los mejores parámetros para estos. También permite comenzar el entrenamiento del modelo en un ordenador local (*on-premises*) y escalar el entrenamiento y el despliegue después en la nube.
- ▶ **Azure Machine Learning Studio.** Otra opción proporcionada por Azure es un entorno visual, de estilo *drag-and-drop*, para desarrollar, evaluar y desplegar modelos de *machine learning* más sencillos, pero más intuitivos, usando algoritmos y módulos de manejo de datos ya existentes. Podemos interpretar esta herramienta como un paso en la democratización del *machine learning*, que lo hace más accesible a un público menos experto (tal vez un equipo en desarrollo, sin gran experiencia aún en este ámbito), pero que no por ello quiere desperdiciar las oportunidades que ofrece la analítica predictiva.

Un paso más allá en la abstracción del *machine learning* se encuentran los servicios específicos relacionados con inteligencia artificial, que ofrecen funcionalidades a más alto nivel y muy específicas. En este caso, el usuario no diseña ni entrena modelos (y, por tanto, no necesita tener el conocimiento y la experiencia necesarios), sino que usa modelos ya preentrenados y desplegados, listos para ser usados a través de una API intuitiva que abstrae todos los detalles a bajo nivel. Entre los ejemplos más notorios, se encuentran los siguientes:

- ▶ **Vision API.** Algoritmos de procesado de imagen para identificar, poner etiqueta, indexar y moderar imágenes y vídeos.

- ▶ **Speech API.** Servicio para convertir texto hablado en escrito, de verificación mediante voz o para añadir reconocimiento de usuarios por su habla.
- ▶ **Knowledge mapping.** Mapeado de información compleja en datos para resolver datos, como sistemas de recomendación y búsquedas semánticas.
- ▶ **Bing search.** Permite usar la API del buscador Bing.
- ▶ **Natural Language Processing API.** Permite a las aplicaciones procesar lenguaje natural con *scripts* ya preconstruidos, evaluar sentimientos y aprender cómo reconocer lo que el usuario busca.

## Cloud computing bible

Sosinsky, B. (2011). *Cloud computing bible*. Wiley.

Este libro constituye una referencia básica sobre los conceptos más importantes sobre *cloud computing*. También recoge una panorámica de las tres principales plataformas de *cloud computing*: Amazon Web Services, Google Cloud Computing y Microsoft Azure. Sin embargo, dada la velocidad de evolución de los servicios ofertados por dichas compañías, y aunque la información que expone el libro al respecto sirve para comprender los orígenes del *cloud computing* en estas plataformas y su evolución hasta la actualidad, se redirige al lector a la documentación más actual, puesto que los servicios ofertados hoy en día poco tienen que ver ya con los expuestos en el libro.

### Conceptos, tecnología y arquitectura del cloud computing

Erl, T. (2013). *Cloud computing: concepts, technology, and architecture*. Prentice Hall.

Este libro constituye una referencia completa que trata en profundidad los conceptos principales sobre *cloud computing*. Está dirigida a todo aquel que quiera profundizar más allá del uso práctico de los servicios que este ofrece.

## Documentación en línea de Microsoft Azure

Microsoft (2021). Microsoft Azure. <https://docs.microsoft.com/en-us/learn/azure/>

El recurso más completo y actualizado sobre Microsoft Azure es su propia documentación en línea. Dada la rapidez con la que los diferentes servicios proporcionados evolucionan, aparecen y dejan de estar disponibles, es difícil encontrar libros que se mantengan actualizados más allá de unos pocos meses. Por tanto, se recomienda acudir a esta documentación, en la que se pueden encontrar recursos sobre servicios concretos (por ejemplo, Azure SQL Database), así como caminos de aprendizaje (*learning paths*) sobre temáticas más generales (por ejemplo, bases de datos o *data engineering* en Azure) que cubren varios servicios, y las interrelaciones entre ellos.

- 1.** ¿Cómo se puede definir *cloud computing*?
  - A. Es la interconexión de una serie de ordenadores.
  - B. Es el proceso de planificar y ejecutar una serie de tareas
  - C. Son una serie de servicios de computación ofrecidos a través de Internet.
  - D. Ninguna de las respuestas anteriores son correctas.
  
- 2.** ¿Qué modelos de servicio *cloud* existen?
  - A. Público, privado e híbrido.
  - B. IaaS, PaaS y SaaS.
  - C. Microsoft Azure, Google Cloud Platform y Amazon Web Services.
  - D. Servidores de cómputo, almacenamiento y bases de datos.
  
- 3.** ¿Cuál de las siguientes propiedades no es una ventaja de *cloud computing*?
  - A. Coste menor de infraestructura por economías de escala.
  - B. Control total de la infraestructura que soporta los servicios.
  - C. Flexibilidad a la hora de escalar la infraestructura necesaria.
  - D. Alta disponibilidad de los servicios gracias a la replicación.
  
- 4.** ¿Cuál de las siguientes opciones no es un tipo de nube?
  - A. Nube pública.
  - B. Nube privada.
  - C. Nube secundaria.
  - D. Nube híbrida.

5. ¿Qué tipo de servicio no es habitual entre los servicios en la nube?
- A. Máquina virtual
  - B. Máquina física.
  - C. Almacenamiento virtual.
  - D. Interconexión de servicios.
6. ¿Qué dos conceptos hacen posible los servicios de computación en la nube?
- A. Virtualización y disminución de costes.
  - B. Disminución de costes y abstracción.
  - C. Disminución de costes y flexibilidad.
  - D. Abstracción y virtualización.
7. ¿Qué tarea reemplaza el uso de servicios en la nube?
- A. Compra e instalación de servidores.
  - B. Actualización y mantenimiento de servidores.
  - C. Dimensionamiento previo y adquisición de servidores para aumentar la capacidad según los requisitos de las aplicaciones.
  - D. Todas las anteriores.
8. Relaciona los servicios de Microsoft Azure con su temática correspondiente.

Almacenamiento	1	A	Azure HDInsight
Computación	2	B	Azure Cosmos
Bases de datos	3	C	Azure Container Instances
Big data	4	D	Azure Disks

- 9.** Para ejecutar un clúster Hadoop en Microsoft Azure:
- A. Solo se puede usar el servicio HDInsight.
  - B. Es obligatorio contratar una o varias instancias VM e instalar el clúster en ellas.
  - C. Microsoft Azure no permite ejecutar clústeres Hadoop.
  - D. Se puede usar el servicio HDInsight u optar por una alternativa IaaS.
- 10.** Si se quieren utilizar servicios relacionados con *machine learning* en Microsoft Azure:
- A. Es necesario disponer de un equipo de expertos en *machine learning* que entiendan y puedan usar los servicios que provee Microsoft Azure.
  - B. Microsoft Azure no proporciona ningún servicio de *machine learning*. Es necesario contratar un servicio de cómputo sobre el que instalar todo el entorno necesario para desarrollar modelos.
  - C. Existen tanto opciones para conocedores de *machine learning*, que disponen de mayor flexibilidad para construir sus modelos, como servicios de inteligencia artificial que no requieren conocimientos de *machine learning*.
  - D. Microsoft Azure no está diseñado ni orientado a ofrecer servicios de *machine learning* de ninguna forma.

Ingeniería para el Procesado Masivo de Datos

---

## Tema 9. Cloud computing II

# Índice

[Esquema](#)

[Ideas clave](#)

- [9.1. Introducción y objetivos](#)
- [9.2. Amazon Web Services](#)
- [9.3. Regiones y availability zones \(AZ\)](#)
- [9.4. Servicios transversales: seguridad y gestión](#)
- [9.5. Servicios de computación](#)
- [9.6. Servicios de red](#)
- [9.7. Servicios de almacenamiento](#)
- [9.8. Bases de datos](#)
- [9.9. Servicios de big data y analítica](#)
- [9.10. Machine learning e inteligencia artificial](#)

[A fondo](#)

[Documentación en línea de Amazon Web Services](#)

[Test](#)

# Esquema

AMAZON WEB SERVICES (AWS)					
Conceptos	Computación	Almacenamiento	Servicios	Big data & analítica	Machine learning & IA
Regiones geográficas, formadas por 3 o más centros de datos	EC2 Instancia computo ~ servidor Ejecución capacidad computo Elección sistema operativo y middleware (AMI)	S3 Almacenamiento objetos (máx. 5TB/objeto) Bucket ~ directorio Virtualmente ilimitado	RDS Base de datos relacional Varios motores: - Aurora - MySQL - PostgreSQL EBS Almacenamiento orientado a bloque Cada instancia EC2 ligada a un único EBS	EMR Clúster Hadoop gestionado KMS Clúster Kafka gestionado Redshift Consultas OLTP Athens Consultas OLAP DynamoDB No relacional Propietaria Amazon Clave/Valor y documento	SageMaker Entrenar datos, construir, entregar, desplegar modelos machine learning Servicios IA No precisan conocimientos machine learning Transcribe Translate Rekognition Fraud detection Comprehend Polly Personalize ...
Áreas de disponibilidad AZ = centros de datos	ECS Contenedores Docker	EKS Contenedores Kubernetes	EFS Almacenamiento orientado a ficheros EFS compartido entre varias instancias EC2	Glue Trabajos ETL totalmente gestionados	Data Pipeline Trabajos ETL control sobre clúster Kinesis
Responsabilidad	Fargate Orquestación gestionada	Modelo de responsabilidad compartido	DocumentDB No relacional Orientada a documento		Streaming gestionado

## 9.1. Introducción y objetivos

En el tema anterior, definimos los servicios de *cloud computing* como una alternativa a tener aplicaciones instaladas en equipos propios, como pueden las tecnologías *big data* que hemos estudiado en esta asignatura. Para mostrar un primer ejemplo, hicimos una revisión de los recursos que proporciona uno de los grandes proveedores de servicios *cloud*, Microsoft Azure.

Sin embargo, el proveedor por excelencia hoy en día es Amazon Web Services (AWS). Por ello, dedicaremos este tema a explorar los servicios que ofrece y cómo está organizado, de forma que tengamos una idea de cómo desplegar las tecnologías *big data* estudiadas en sus servicios *cloud*, o qué servicios nos ofrece como alternativa para alcanzar los mismos objetivos de gestión (almacenamiento y procesado) de grandes cantidades de datos.

Con esto en mente, los principales objetivos de este tema serán los siguientes:

- ▶ Conocer Amazon Web Services como el principal proveedor de servicios de *cloud computing*.
- ▶ Explorar el abanico de servicios *cloud* que proporciona AWS.
- ▶ Examinar en detalle los servicios IaaS que ofrece AWS y sobre los que se pueden desplegar las tecnologías *big data* estudiadas en la asignatura.
- ▶ Conocer las alternativas PaaS y SaaS centradas en *big data, analytics y machine learning* que ofrece AWS de forma nativa.

## 9.2. Amazon Web Services

Amazon Web Services (AWS) es actualmente la plataforma líder de servicios en la nube, la cual presta sus servicios a grandes empresas (Expedia, Shell), *startups* (Airbnb, Lyft), sector público (Coursera, FDA) y particulares. En la figura 1 se puede observar la posición de cada proveedor de servicios *cloud* según el cuadrante del año 2020 de la consultora Gartner; en él se aprecia cómo AWS tiene un papel dominante, seguido de cerca por Microsoft Azure (que ya exploramos en el capítulo previo) y Google Cloud Services (que exploraremos en el siguiente tema.)



Figura 1. Cuadrante mágico de Gartner para servicios *cloud* de infraestructura (IaaS) y plataforma (PaaS), a fecha de agosto de 2020. Fuente: <https://pages.awscloud.com/GLOBAL-multi-DL-gartner-mq-cips-2020-learn.html?pg=WIAWS>

AWS ofrece un gran abanico de servicios en la nube, mediante una infraestructura global que proporciona acceso desde casi cualquier parte del mundo, alta disponibilidad y seguridad, y un esquema de pago por uso (*pay-as-you-go*). Entre los servicios que ofrece, podemos encontrar los comunes a todas las plataformas de *cloud computing*, como son almacenamiento, procesamiento, redes y seguridad; pero también otros más específicos, relacionados con sectores concretos, como analítica de datos, *devops*, bases de datos, inteligencia artificial, IoT, desarrollo móvil, entre muchos otros. Además, al igual que otras plataformas de *cloud computing*, permite desplegar modelos tanto completamente en la nube como de forma híbrida, es decir, ofreciendo algunos servicios en sus propios servidores y otros en la nube, y dando posibilidad de migrar cuando el usuario decida hacerlo. Además de los servicios que ofrece la propia plataforma AWS, cuenta con el denominado AWS Marketplace, un catálogo digital con numerosos recursos adicionales proporcionados por empresas independientes de Amazon, pero basadas en los servicios en la nube de AWS, como imágenes de máquinas virtuales, herramientas de BI, bases de datos, etc.

Antes de empezar a explorar el catálogo de servicios, es interesante conocer cómo se organizan los recursos en AWS y conocer conceptos importantes como son las regiones y las zonas de disponibilidad, ya que son transversales y afectan a todos los servicios. De igual forma, existen otra serie de servicios de gestión y seguridad que también trataremos brevemente a fin de reflejar su importancia, aunque no entremos en detalle. Después de conocer todos estos aspectos, pasaremos a explorar el abanico de servicios de AWS, los cuales agruparemos por servicios de computación, almacenamiento, redes, bases de datos, *big data* y analítica, y *machine learning*.

## 9.3. Regiones y availability zones (AZ)

Una de las principales características de las plataformas de *cloud computing* en general y de AWS en particular es su disponibilidad global. Esto significa que es posible acceder a sus servicios desde casi cualquier punto del planeta. Por lo tanto, uno de los primeros conceptos que debemos entender es cómo se organizan y distribuyen estos servicios. En el caso de AWS, están desplegados por regiones. Una **región** es un área geográfica autocontenido, que ofrece una serie de servicios de *cloud* (no todos los servicios AWS se ofrecen en todas regiones) basados en una tipología de recursos alojados en dicha región. Cada una está localizada en un país y se rige por un conjunto de leyes determinado. Actualmente, existen 24 regiones en todo el mundo (Londres, Sídney, Montreal, Sao Paulo...) más 4 adicionales que se incorporarán en breve. En la figura 2, puede observarse su localización:



Figura 2. Regiones de AWS a fecha de octubre de 2020. Fuente: <https://aws.amazon.com/about-aws/global-infrastructure/>

Cada región está diseñada para contener todos los recursos necesarios para ejecutar la aplicación que se quiera desplegar en la nube. La pregunta es: ¿qué

región elegir? Básicamente, hay que tener en cuenta cuatro aspectos fundamentales para responder a esta cuestión.

- ▶ El primer aspecto tiene que ver con la **latencia** y, por ende, con la localización de los usuarios de la aplicación. Si estos van a acceder desde Londres, por ejemplo, lo más lógico será desplegar la aplicación en la región de Londres, de forma que la latencia (el tiempo que tardan los datos en viajar desde los recursos AWS hasta los usuarios) sea el menor posible. Si, por el contrario, alojamos nuestra aplicación en la región de Montreal, la latencia para nuestros usuarios de Londres va a ser mucho mayor, ya que la respuesta a cada petición que hagan a la aplicación va a tener que viajar desde Montreal (Canadá) hasta Londres (Reino Unido).
- ▶ La segunda cuestión que se debe tener en cuenta es que no todas las regiones tienen los mismos precios en sus servicios. Dado que en cada región rigen diferentes leyes y la situación financiera es distinta, **los costes de los mismos servicios ofrecidos por AWS en distintas regiones pueden ser muy diferentes**. Estos costes están anunciados y se pueden consultar.
- ▶ La tercera consideración tiene que ver con las **leyes que rigen en cada región**. Por ejemplo, en Europa hay que cumplir con la ley GDPR de protección de datos, por lo que, si una empresa con sede europea quiere desplegar sus servicios en la nube con AWS, será necesario que lo haga en una región europea, ya que, debido a esta normativa, los datos sensibles de sus clientes no pueden alojarse ni enviarse fuera de Europa.
- ▶ Por último, el cuarto aspecto que ha de tomar en cuenta es la **disponibilidad de los servicios**. La mayoría de los servicios de AWS, sobre todo los más básicos, están disponibles en casi cualquier zona. No obstante, antes de tomar una decisión, hay que comprobar que, efectivamente, los servicios que queremos usar estén disponibles en la región que nos interesa o, en caso negativo, buscar otra región que sí los ofrezca.

Pero, a nivel práctico, ¿de qué se compone una región? Una región es una colección

de lo que se conoce como **zonas de disponibilidad o availability zones (AZ)**, que básicamente son centros de datos (o *datacenters*). **En cada región, hay, al menos, dos availability zones, separadas geográficamente, aisladas e interconectadas con una red de fibra propia de baja latencia**, alto ancho de banda y gran redundancia. Por tanto, aunque físicamente sean centros de datos separados, lógicamente funcionan como una única área, de forma que las aplicaciones que se ejecutan en una región lo hacen en todos los centros de datos asociados a esta, de forma simultánea. También puede existir una base de datos maestra en una *availability zone* conectada a otra en una *availability zone* distinta, en estado de espera ante fallos.

De esta forma, la región es robusta ante desastres naturales u otros problemas potenciales que pueden aparecer en un centro de datos, ya que existen otros para respaldarlo. Por tanto, ofrece no solo escalabilidad, sino también alta disponibilidad de los servicios. Otro aspecto interesante es que los recursos o servicios contratados en una región no se replican en otras regiones a no ser que se especifique explícitamente, lo que implica que se facturan por separado. Es decir, si tenemos una base de datos que queremos tener replicada en dos regiones para aumentar la disponibilidad y fiabilidad, pagaremos lo correspondiente a cada una de las regiones donde esté desplegada.

## 9.4. Servicios transversales: seguridad y gestión

Antes de conocer los diferentes servicios clásicos de cómputo, almacenamiento, etc., es recomendable tener en cuenta algunos detalles sobre servicios transversales como la seguridad y la gestión de los recursos contratados. En este sentido, AWS se centra en cinco puntos para garantizar la seguridad de las aplicaciones alojadas en su nube:

- ▶ **Protección de los datos.** Uno de los principales objetivos de seguridad es proteger los datos, las cuentas y los trabajos alojados en la nube de cualquier acceso no autorizado. Para ello, AWS proporciona diferentes mecanismos de encriptación de datos, gestión de claves y certificados (AWS Key Management System, AWS KMS; AWS Secrets Manager; AWS Certificate Manager); detección de amenazas o comportamientos sospechosos, o protección específica de datos sensibles (Amazon Macie).
- ▶ **Gestión de identidad y acceso.** AWS ofrece una serie de herramientas con el objetivo de gestionar tanto la identidad de los usuarios que acceden a los servicios contratados como los permisos de los roles que tienen asignados. La principal es AWS Identity & Access Management (AWS IAM), y también son de utilidad Amazon Cognito y AWS Organizations.
- ▶ **Protección de la infraestructura.** Además de proteger los centros de datos a nivel físico, AWS también facilita recursos destinados a proteger las aplicaciones a nivel de red, mediante mecanismos como el filtrado de peticiones basado en IP o en el contenido HTTP, el filtrado de tráfico malicioso (AWS Web Application Firewall, WAF), la gestión de reglas impuestas por un *firewall* (AWS Firewall Manager) o la protección frente a ataques DDoS (AWS Shield).
- ▶ **Detección de amenazas y monitorización continua.** Diversos servicios de monitorización y detección de eventos sospechosos están disponibles en AWS,

como AWS Config y AWS CloudTrail para configuración y monitorización del uso de API, o Amazon GuardDuty para la detección de amenazas.

- ▶ **Privacidad de los datos.** El servicio AWS Artifact proporciona informes sobre el cumplimiento de las mejores prácticas en cuanto a manejo de datos sensibles, así como sugerencias sobre cómo mejorar aspectos que puedan dar lugar a un uso de datos peligroso.

Además de los servicios nombrados previamente, uno de los pilares de AWS (así como de otros proveedores, como ya vimos con Microsoft Azure) es el **modelo de responsabilidad compartida en seguridad y cumplimiento de regulaciones**.

Mientras que AWS aligera parte de la carga operativa al trabajar y gestionar desde el sistema operativo hasta la capa física de los equipos que conforman la infraestructura, el usuario sigue siendo responsable de otras tantas partes del sistema, que serán unas u otras dependiendo de los servicios contratados. En el caso de AWS, estas responsabilidades se dividen de la siguiente forma:

- ▶ **AWS** tiene las competencias sobre la **seguridad de la nube (security of the cloud)**. Es decir, es responsable de proteger la infraestructura sobre la que se ejecutan los servicios *cloud* contratados por el usuario.
- ▶ **El usuario** es responsable de la **seguridad en la nube (security in the cloud)**. No obstante, el alcance de esta responsabilidad dependerá del servicio concreto contratado. Por ejemplo, en una instancia EC2, al tratarse de un servicio IaaS, el usuario será responsable de configurar y gestionar tareas relacionadas con la seguridad de dicha instancia (instalar actualizaciones de seguridad, asegurar el software que instale en dicha instancia, etc.). Por otro lado, si elige servicios PaaS gestionados por Amazon (como, por ejemplo, Amazon S3 o Amazon DynamoDB), no tendrá que preocuparse por el sistema operativo o *middleware* instalado, sino tan solo de configurar correctamente la gestión de acceso (lectura/modificación) de usuarios a la información contenida en dichos servicios o de la encriptación de esta.

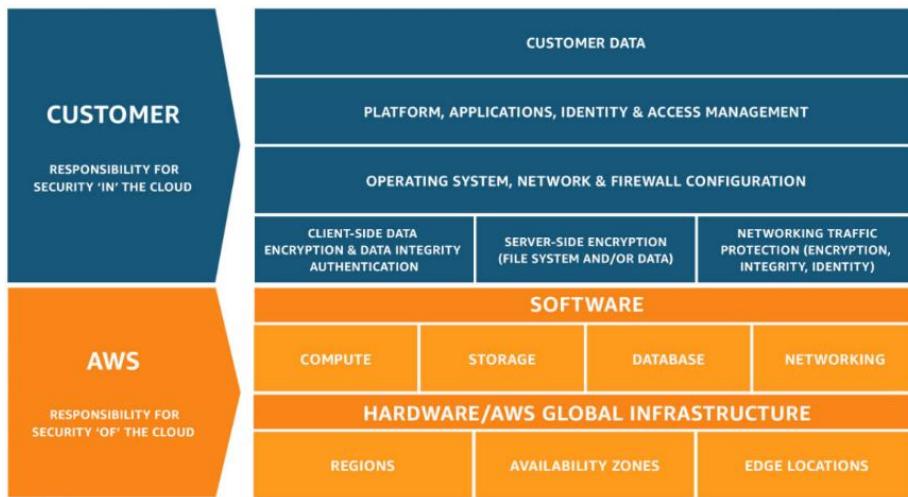


Figura 3. Modelo de responsabilidad compartida de AWS. Fuente:

<https://aws.amazon.com/compliance/shared-responsibility-model/>

Una vez que conocemos los conceptos y las ideas transversales, pasamos a explorar algunos de los recursos que ofrece AWS: desde servicios orientados a infraestructura (IaaS) hasta despliegue de aplicaciones web, servicios relacionados con IoT, DevOps o servicios para aplicaciones de teléfonos móviles, entre otras muchas categorías. Dada la envergadura de la lista completa, vamos a hacer una revisión de los principales, agrupándolos en los conjuntos clásicos de servicios (computación, almacenamiento y red). A esta clasificación, añadiremos tres grupos más, enfocados en temáticas relacionadas con el *big data* y la analítica de datos: bases de datos, analítica y *machine learning*. De esta forma, se pretende dar una visión de los servicios de infraestructura sobre los que se podrían desplegar las tecnologías *big data* estudiadas en los temas previos (HDFS, Spark, Hive, Kafka...), así como una panorámica de los servicios gestionados por AWS que nos proporcionan estas mismas tecnologías o alternativas propietarias, de manera que podamos establecer similitudes y diferencias entre todas ellas.

## 9.5. Servicios de computación

Entre los servicios de cómputo que se pueden encontrar en AWS, tenemos Amazon **Elastic Compute Cloud (EC2)**, que es el equivalente a un servidor físico, pero en la nube (IaaS); y opciones como **Elastic Container Service (ECS)**, que permite usar contenedores Docker, o **Elastic Kubernetes Service (EKS)**, en el caso de querer usar la tecnología Kubernetes (PaaS). Existen más servicios de computación interesantes, como AWS Lambda, que ofrece computación *serverless* (FaaS) (recordemos que esta modalidad permitía tener funciones que se disparan ante ciertos eventos o intervalos), o AWS Lightsail (SaaS), que va un paso más allá en la gestión y ofrece la opción de desplegar aplicaciones web, tales como páginas web, sin que el usuario se tenga que preocupar de nada más que de rellenar la información que quiere mostrar, ya que AWS se encarga de gestionar toda la infraestructura y el *software* necesarios. Para nuestro propósito de despliegue de tecnologías *big data*, los servicios más interesantes son AWS EC2 y las tecnologías de contenedores, que se describen con algo más de detalle a continuación.

- ▶ **EC2** permite disponer de servidores virtuales bajo demanda. Cada servidor virtual se llama **instancia** EC2, y con ella es posible hacer prácticamente cualquier cosa que se puede realizar con un servidor físico tradicional: instalar y desplegar *software*, configurar opciones, desarrollar código, etc. Se pueden adquirir tantas instancias como se desee. Cada una proporciona toda la flexibilidad necesaria para arrancarlas, pararlas, o eliminarlas en cualquier momento, de forma que, al parar o eliminar una instancia, esta deja de consumir recursos y, por tanto, de incurrir en costes. Cuando se crea una instancia EC2, se configuran principalmente dos aspectos:
- **Sistema operativo.** Cada instancia EC2 soporta un amplio rango de sistemas operativos dentro del conjunto de Linux y Windows. Para seleccionar el sistema operativo que estará instalado en una instancia EC2, elegiremos lo que se llama

**Amazon Machine Image (AMI).** Esta AMI contiene información sobre qué instalar y cómo configurar la instancia EC2, en relación tanto con el sistema operativo como con las aplicaciones que se quieran añadir. Una misma AMI se puede aplicar a múltiples instancias EC2, de forma que tengamos varias instancias con la misma configuración inicial.

- **Capacidad de cómputo, memoria y red.** Cada instancia se puede personalizar eligiendo la capacidad de cómputo, el tamaño de la memoria RAM y las características de red, es decir, si tiene más o menos ancho de banda y, por tanto, mayor o menos latencia al conectarse a la red. Existen muchos tipos de instancias EC2 predefinidas con una serie de características de capacidad de cómputo, memoria y red, que se adaptan a diferentes tipos de problemas, cada una con un coste acorde a las capacidades que proporciona. Estos tipos de instancias están agrupadas por nombres similares y atienden a optimizaciones para diferentes casos de uso. Por ejemplo, las instancias de tipo G están optimizadas para aplicaciones de uso intensivo de gráficos o vídeo. Se pueden encontrar configuraciones de propósito general, optimizadas para cómputo o uso de memoria intensivos, con aceleradores *hardware* de cómputo u optimizadas para almacenamiento (para ver toda la familia de tipos de instancia, se recomienda consultar el siguiente [enlace](#)). Para cada una de las familias, además, existen diferentes configuraciones. Por ejemplo, la figura 4 muestra las diferentes opciones de instancias T2 de propósito general.

Instance	vCPU*	CPU Credits / hour	Mem (GiB)	Storage	Network Performance
t2.nano	1	3	0.5	EBS-Only	Low
t2.micro	1	6	1	EBS-Only	Low to Moderate
t2.small	1	12	2	EBS-Only	Low to Moderate
t2.medium	2	24	4	EBS-Only	Low to Moderate
t2.large	2	36	8	EBS-Only	Low to Moderate
t2.xlarge	4	54	16	EBS-Only	Moderate
t2.2xlarge	8	81	32	EBS-Only	Moderate

All instances have the following specs:

- Intel AVX†, Intel Turbo†
- t2.nano, t2.micro, t2.small, t2.medium have up to 3.3 GHz Intel Scalable Processor
- t2.large, t2.xlarge, and t2.2xlarge have up to 3.0 GHz Intel Scalable Processor

Figura 4. Especificaciones de los diferentes tipos de instancia T2. Fuente: <https://aws.amazon.com/ec2/instance-types/>

Sin embargo, la elección del tipo de instancia no quiere decir esta que sea inamovible. Si en algún punto la configuración *hardware* de la instancia deja de ser adecuada para nuestros propósitos, podemos modificarla en la consola de control o de forma programática, con una llamada a cierta API. Estos cambios se pueden hacer manualmente, así como de forma automática. Toda esta infraestructura proporciona muchas posibilidades, gracias a la rápida disponibilidad de diferentes tipos de servidores que se pueden crear y eliminar en cualquier momento.

- ▶ **ECS** es un servicio que permite ejecutar y escalar en AWS aplicaciones desde

contenedores Docker, mientras que **EKS** es su análogo para contenedores Kubernetes. Recordemos que los contenedores permiten empaquetar un sistema operativo con el conjunto de *software* y librerías (con su versión correspondiente) necesarios para ejecutar cierta aplicación o servicio, y que dichos contenedores pueden ser desplegados en cualquier sitio. Con este último fin, AWS ofrece varios servicios, como los mencionados ECS y EKS, pero también AWS Fargate, que permite ejecutar los contenedores sin necesidad de gestionar los servidores o clústeres.

## 9.6. Servicios de red

Para conectar los recursos contratados en AWS, existen diferentes servicios de red. El primero del que vamos a hablar es **Virtual Private Cloud (VPC)**. Una VPC sirve para aislar la aplicación de un usuario del resto de aplicaciones que se ejecutan en AWS. Es como un marco que aísla todos los recursos que usa la aplicación dentro de una caja, de la que nada sale ni entra sin consentimiento expreso. Este consentimiento expreso no deja de ser un filtro por IP, puerto, protocolo u otra información. Además, una VPC se puede dividir en **subredes**, que generalmente se usan para determinar accesos y aislar diferentes tipos de tráfico en distintas subredes (por ejemplo, para separar *frontend* de *backend* o de capa de datos).

Para crear una VPC, lo habitual es comenzar por determinar la región en la que se quiere desplegar y el rango de direcciones IP que va a utilizarse. Dentro de la VPC, se define una subred seleccionando un subrango de entre el rango de direcciones IP elegido para la VPC. Por último, una vez completado este paso, se crea una instancia EC2 conectada a la subred. Cabe señalar que una VPC engloba todas las zonas de disponibilidad (AZ) de la región donde está desplegada, por lo que se pueden añadir subredes en diferentes AZ. No obstante, cada subred debe estar dentro de una única AZ y no puede extenderse a más una.

En cada región, AWS proporciona una VPC por defecto, que ofrece 65 536 direcciones IP, y una subred para cada AZ de la región, con 4096 IP por cada subred. La tabla de rutas asociada con la VPC por defecto tiene una ruta pública, la cual estará conectada a un *gateway* que dará acceso a Internet.

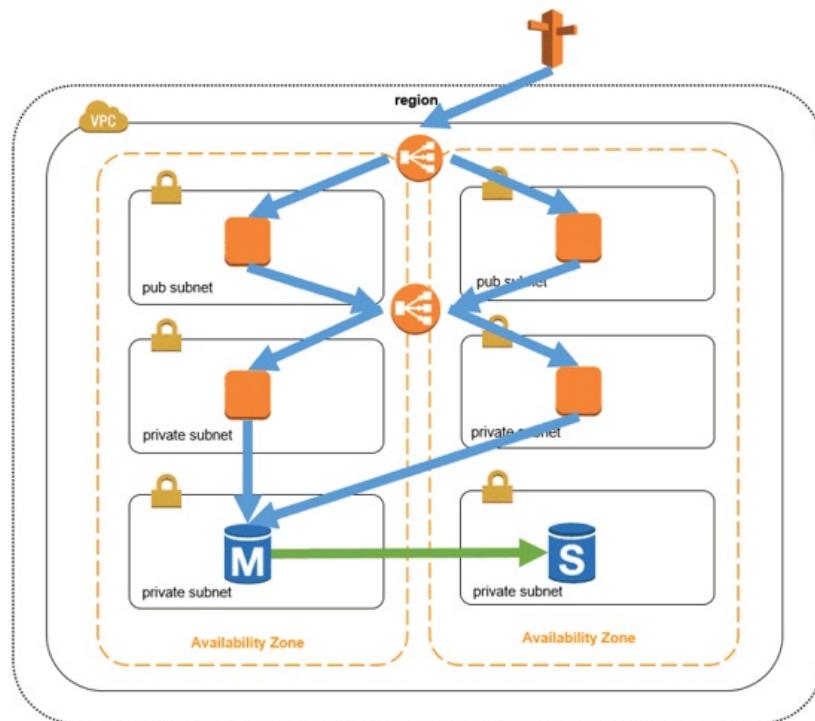


Figura 5. Uso de VPC y subredes para aislar recursos críticos, proporcionar redundancia y propiciar el balanceo de carga. Fuente: <https://aws.amazon.com/blogs/startups/scaling-on-aws-part-2-10k-users/>

Para dar un ejemplo de lo que permiten las VPC y las subredes en cuanto a aislamiento de recursos, nos fijaremos en la figura 5. En ella tenemos una aplicación desplegada en una región de AWS, dentro de su VPC particular. Dicha VPC se extiende a las dos AZ que la componen y que van a servir para aumentar la disponibilidad y el balanceo de carga a la aplicación. Cada AZ tiene tres subredes: una pública, es decir, conectada mediante un *gateway* a Internet (para que los usuarios puedan acceder a la aplicación); y dos privadas, lo que significa que no pueden ser accedidas desde Internet, solo desde otras subredes dentro de la VPC. Dentro de las subredes, vemos una serie de instancias EC2 (cuadrados naranjas), así como una base de datos máster (M) y sincronizada (S) en *standby* para añadir respaldo de seguridad.

Las subredes públicas engloban los componentes de la aplicación que proporcionan

e l *frontend*, para que los usuarios puedan acceder a este. Por otro lado, el componente encargado de la parte *backend* está en una subred privada y la capa de datos (base de datos, en este caso), en otra subred privada diferente. Además, vemos replicación de la estructura de una AZ en otra AZ, para proporcionar mayor disponibilidad y capacidad de procesado de peticiones. Esta separación en subredes nos permite que los usuarios solo tengan acceso al *frontend* a través del *gateway* y del balanceador de carga que exponen las subredes públicas. Por otro lado, un segundo balanceador de carga permite repartir las peticiones desde el *frontend* a las diferentes instancias del *backend*, que proporcionan más capacidad para procesarlas. Finalmente, estas dos instancias de *backend* acceden a la base de datos máster (la única activa, ya que la réplica sincronizada solo proporciona respaldo ante fallos).

## 9.7. Servicios de almacenamiento

El almacenamiento es una de las partes críticas de la arquitectura de cualquier aplicación. Construir y mantener un repositorio de datos es complejo y caro, y consume mucho tiempo. Por ello, al igual que ocurría con la capacidad de cómputo, se quiere evitar que la capacidad estimada esté por debajo o por encima de la necesaria, y también tener la posibilidad de incrementar dinámicamente la capacidad de almacenamiento según sea necesario. AWS cuenta con diversos servicios de almacenamiento, todos ellos ofrecidos a través de Internet de forma duradera y fiable. Cada uno está destinado a diferentes casos de uso o necesidades. A continuación, se presentan los tres tipos más empleados:

- ▶ **Amazon Elastic Block Store (EBS)** está orientado a volúmenes de bloques, para usarlo en las instancias EC2 como almacenamiento persistente. Puede verse como el *disco duro* de una instancia EC2, donde estará instalado el *software* y copiados los ficheros necesarios para que esta realice su función. Por tanto, **cada servicio EBS estará asociado a una única instancia EC2**. Cada volumen EBS se replica automáticamente dentro de una AZ para protegerla de posibles fallos, lo que mejora la disponibilidad y durabilidad. Los volúmenes EBS ofrecen la consistencia y baja latencia que necesitan las instancias EC2 para cargar y gestionar los datos necesarios.

EBS proporciona un rango de opciones que permiten optimizar el funcionamiento y coste del servicio de almacenamiento. Estas opciones se dividen en dos categorías: almacenamiento **SSD**, para trabajos transaccionales (baja latencia, para bases de datos y volúmenes de arranque de EC2), o **HDD** (*hard disk drive*), para trabajos intensivos que necesitan un alto ancho de banda (MapReduce o procesado de *log*, que no requieren tanto una baja latencia como una capacidad grande de procesado). EBS permite aumentar dinámicamente la capacidad, ajustar los parámetros y cambiar el tipo de volumen sin perjuicio en el rendimiento. El coste va a depender de

la capacidad y del tipo de volumen, así como de la región.

- ▶ **Amazon Simple Storage Service (Amazon S3)** almacena datos como objetos (ficheros) dentro de recursos llamados ***buckets***. Se pueden almacenar tantos objetos como sean necesarios dentro de un *bucket*, cada uno con un tamaño de hasta 5TB; también es posible guardar, acceder y eliminar objetos. Como medida de control y auditoría de acceso y modificación, el sistema permite controlar los *logs* de acceso y la lectura de cada *bucket* y de los objetos concretos. De igual forma, se puede elegir la región donde se almacena cada *bucket* para optimizar la latencia, minimizar costes o cumplir con requisitos regulatorios.
- ▶ **Amazon Elastic File System (EFS)** proporciona almacenamiento para archivos en la nube, parecido a un NAS. Ofrece una interfaz sencilla que permite crear y configurar sistemas de ficheros. Está diseñado para facilitar el acceso masivo paralelo compartido a miles de instancias EC2. Cuando se monta un sistema de ficheros EFS en una instancia EC2, provee una interfaz del sistema de ficheros estándar que posibilita la integración de EFS con las aplicaciones y herramientas existentes. **Múltiples instancias EC2 pueden acceder a un mismo sistema de ficheros EFS** al mismo tiempo, de forma que se erige como una fuente de datos común para aplicaciones y servicios que se ejecutan en distintas instancias EC2.

Amazon S3 es, tal vez, la opción más típica para multitud de aplicaciones y, por ello, le vamos a prestar un poco más de atención. Decíamos que S3 permite almacenar objetos (ficheros) en *buckets* (algo similar a un directorio, cuyo nombre debe ser único a nivel global). S3 es la parte de almacenamiento de muchos servicios de *machine learning* en AWS (en SageMaker, en concreto). De igual forma, es muy útil crear un DataLake con S3, porque no tiene límite de tamaño, pero sí una **durabilidad del 99,99999 %, y permite independizar el almacenamiento de los servicios de computación** (EC2, Athena, Redshift, Spectrum, Rekognition, Glue...). Además, tiene una arquitectura centralizada con almacenamiento orientado a objetos, lo que significa que **todos los datos son accesibles desde un único**

punto y soporta casi cualquier formato de fichero (CSV, JSON, Parquet, ORC, Avro, Protobuf).

Existen diferentes **tipos de almacenamiento S3**, o lo que se conoce como S3 Storage Tiers, en los que el coste se abarata según disminuye el acceso a los datos guardados. En la tabla 1, se muestra una comparativa de sus características.

- ▶ Amazon S3 **Standard**. Almacenamiento de propósito general.
- ▶ Amazon S3 **Standar-Infrequent Access (IA)**. Para datos que no se acceden de forma frecuente.
- ▶ Amazon S3 **One Zone-Infrequent Access**. Para datos que no se acceden de forma frecuente y que, además, se almacenan en una única AZ (por tanto, estamos dispuestos a perderlos en caso de algún fallo en el centro de datos donde se guarden).
- ▶ Amazon S3 **Intelligent Tiering**. Amazon decide en qué tipo de almacenamiento S3 (*tier*) guardar los datos que se le entreguen, para optimizar el coste frente a los requisitos del sistema.
- ▶ Amazon **Glacier**. Para archivar datos que van a ser accedidos de forma muy ocasional o incluso excepcional.

Tipos de almacenamiento S3					
	Standard	Standard IA	1 AZ – Standard IA	Intelligent Tiering	Glacier
Durabilidad	99,999999999 %				
Disponibilidad	99,99 %	99,9 %	99,5 %	99,90 %	NA
Tolerancia a fallos	>=3	>=3	1	>=3	>=3
Uso	Uso frecuente	Uso poco frecuente		Asignación inteligente	Archivo datos

Tabla 1. Tipos de almacenamiento S3. Por «Tolerancia a fallos» se entiende el número de AZ (*datacenters*) que tienen que fallar a la vez para perder los datos.

Existen una serie de **reglas para el ciclo de vida de los datos**, para ahorrar en costes. Por ejemplo, se empieza por almacenar datos en la opción Standard cuando se usan de forma habitual, después pasan al almacenamiento Standard IA cuando dejan de accederse a menudo y, por último, pasan a almacenarse en Glacier cuando apenas se acceden. Es posible **programar acciones de transición**, es decir, reglas que determinan a qué clase de almacenamiento mover los datos en cada momento. Así, se pueden configurar los objetos para que se almacenen en Standard IA 60 días después de su creación y en Glacier a los 6 meses. O incluso de pueden programar reglas de caducidad, para eliminar los archivos una vez haya pasado un tiempo determinado (por ejemplo, cuando se quiere eliminar ficheros de *log* 6 meses después de su creación). Como se ve, existen muchos recursos para gestionar el almacenamiento de los datos, a fin de ahorrar en costes tanto como sea posible.

Cabe mencionar también que **es posible encriptar objetos en S3 hasta con cuatro opciones** distintas: SSE-S3 (Server Side, Encryption, con claves gestionadas por AWS), SSE-KMS (con claves gestionadas a través de Key Management Service, con seguridad adicional), SSe-C (cuando es el usuario el que gestiona sus propias claves) y CSE (Client Side Encryption, es el cliente el responsable de encriptar los datos fuera de AWS antes de enviarlos a almacenar en S3). En *machine learning*, generalmente se usa SS3-S3 y SSE-KMS. Por otro lado, S3 da opción de aplicar medidas de seguridad basada en acceso por usuario o en acceso por recurso (a nivel *bucket* y a nivel objeto). Otro aspecto que considerar es la necesidad, por seguridad, de que todo el tráfico se quede dentro de la VPC (en lugar de salir a la web pública). Es decir, es más que aconsejable asegurar que los servicios privados, como AWS SageMaker, puedan acceder a S3 dentro de la VPC. Es lo que ocurría en la figura 5 con las instancias EC2 y la capa de datos, que podría ser S3 en lugar de una base de datos. De esta forma, se evita que datos sensibles viajen a través de la red pública de Internet y se aumenta el nivel de privacidad de estos.

## 9.8. Bases de datos

Otra de las herramientas clave para muchas aplicaciones y que ofrecen los proveedores de servicios en la nube son las bases de datos. Una de las alternativas para utilizar bases de datos con AWS es el servicio de computación EC2 descrito anteriormente, que, como veíamos, no es más que un servidor virtual en la nube en el que se puede instalar cualquier *software*. Por tanto, basta con instalar la base de datos que elijamos y ejecutarla en la instancia EC2. El problema de esta opción es que, aunque la infraestructura *hardware*, es decir, la instancia EC2, está gestionada por AWS, será responsabilidad del usuario gestionar la base de datos como tal (instalación, actualizaciones de seguridad, configuración, mantenimiento...). Para evitar estos inconvenientes, AWS ofrece otras opciones, de forma que el usuario no tenga que preocuparse por gestionar ni la arquitectura *hardware* ni tampoco la base de datos en sí. Únicamente habrá de encargarse del diseño de los esquemas de la base de datos, así como de la gestión y el uso de los datos que quiera introducir en ella.

Entre las decisiones que sí que tiene que tomar el usuario a la hora de usar estos servicios PaaS, una de las más importantes es si la base de datos va a ser para desarrollo o producción. Ambos conceptos están directamente relacionados con la opción de desplegar la base de datos, bien en una única zona de disponibilidad (AZ), que es más económico, bien en modo multi-AZ para proporcionar alta disponibilidad, que es más costoso, pero más interesante cuando se trata de una base de datos en producción. También hay que decidir sobre encriptación y *backups*, monitorización, si queremos actualizar a nuevas versiones o quedarnos en la versión actual, protección con eliminación o sobre control de acceso. A continuación, se describen los principales servicios de bases de datos de que dispone AWS.

- ▶ **Amazon Relational Database Service (Amazon RDS)** proporciona diferentes tipos

de bases de datos relacionales gestionadas por AWS. Al ser un servicio PaaS, el usuario puede crear, operar y escalar la base de datos de su elección directamente, sin necesidad de provisionar *hardware*, configurar la base de datos, actualizarla o realizar *backups*, manteniendo siempre la alta disponibilidad y la capacidad de redimensionar la base de datos para ajustarla a las necesidades de cada momento. Solo se tiene que preocupar de diseñar los esquemas de la base de datos, decidir si desea encriptación y quién tiene permisos para acceder y usar los datos, así como de gestionar estos últimos.

Amazon RDS ofrece seis motores de base de datos: **PostgreSQL**, **MySQL**, **MariaDB**, **Oracle**, **Microsoft SQL Server** y **Amazon Aurora**. Esta última es una base de datos nativa en la nube, compatible con MySQL y PostgreSQL, que mejora las prestaciones de los motores originales. Concretamente, Amazon Aurora es hasta cinco veces más rápida que MySQL estándar y hasta tres veces más que PostgreSQL. Proporciona alta disponibilidad (incluso quince réplicas de lectura de baja latencia), posibilidad de *backups* continuos en S3 y replicación de los datos en distintas AZ.

Además de los diferentes motores de base de datos, AWS dispone de un servicio adicional, **AWS Database Migration Service (AWS DMS)**, para ayudar a migrar bases de datos, tanto *on-premises* como en otras plataformas en la nube, a una base de datos en AWS.

- ▶ En el apartado de bases de datos NoSQL, AWS ofrece **DynamoDB**, un servicio de base de datos no relacional rápida y flexible, con latencia por debajo de los 10 ms a cualquier escala. Al igual que las bases de datos relacionales del apartado anterior, también está gestionada por AWS y soporta modelos de almacenamiento tanto de documento como de clave-valor. Por tanto, tampoco en este caso hay que preocuparse ni por la gestión de la infraestructura *hardware* ni por los pormenores de la administración de la base de datos en sí, sino únicamente de definir y crear tablas, determinar los datos que van a componer la base y concretar las necesidades de

tráfico de estos. En RDS, lo más importante al crear la base de datos es determinar la capacidad, mientras que, en DynamoDB, lo que se define son los requisitos de rendimiento en términos de rapidez en la transferencia de datos. Y en cualquiera de los dos casos, esta especificación inicial de necesidades puede escalarse a medida que estas cambien.

AWS se encarga de particionar los datos en diferentes servidores para poder escalar el tamaño y aumentarlo según sea necesario, sin que exista un límite. DynamoDB se encarga, además, de sincronizar réplicas de los datos en tres centros (AZ) de una región para asegurar redundancia y disponibilidad, así como para que un fallo en una AZ no afecte al funcionamiento de la base de datos.

- ▶ **Amazon ElastiCache.** Este servicio permite construir y ejecutar en la nube **bases de datos en memoria** compatibles con **Redis o Memcached**. Este tipo de almacenamiento hace posible el uso intensivo de datos y la obtención de los datos consultados con baja latencia gracias a que estos se guardan en memoria. Algunos casos de uso de este tipo peculiar de base de datos son cachés, almacenamiento de datos de sesión, aplicaciones de juegos o analíticas en tiempo real, entre otros.
- ▶ **Amazon DocumentDB.** Esta base de datos de documentos no relacional, que dispone de compatibilidad con **MongoDB** y que está completamente gestionada por Amazon. Ofrece como principales características rapidez, escalabilidad y **alta disponibilidad (99,99 %)**. Soporta trabajos de MongoDB y permite almacenar y consultar datos **JSON**. Para conseguir la alta disponibilidad y aumentar la eficiencia, DocumentDB permite tener hasta quince réplicas de lectura de una misma base de datos, independientemente del tamaño de esta, mientras que los datos como tales se pueden replicar hasta en seis copias repartidas en tres AZ.

Además de las bases de datos ofrecidas como PaaS y mencionadas previamente, cabe recordar que, si fuera preciso proporcionar un servicio de base de datos en la nube que AWS no tiene en su catálogo, o si se quiere usar una que sí ofrece AWS, pero teniendo control total de instalación y gestión sobre ella, siempre es posible usar

instancias EC2 o contendores ECS/EKS. De esta manera, será el propio usuario el que gestione la instalación y el despliegue de la base de datos sobre dichos servicios IaaS.

## 9.9. Servicios de big data y analítica

AWS ofrece un gran abanico de servicios relacionados con el mundo *big data* y la analítica de datos, desde clústeres Hadoop gestionados hasta aplicaciones individuales que se asemejan en funcionalidad a las que se han visto a lo largo de la asignatura: Hive, Spark Streaming, Kafka... Como siempre, cabe recordar que se pueden utilizar instancias EC2 (IaaS) para instalar las tecnologías del ecosistema Hadoop como si de un clúster local se tratara. Las opciones que analizaremos en esta sección son alternativas o complementos que aligeran la gestión de instancias y el montaje del clúster para que el usuario solo tenga que usar los servicios (PaaS y SaaS).

### Amazon Elastic MapReduce (Amazon EMR)

El servicio más cercano a las tecnologías estudiadas en los temas previos es **Amazon Elastic MapReduce (Amazon EMR)**. Se trata de una plataforma que proporciona tecnologías Hadoop sobre instancias EC2, pero sin que el usuario se tenga que preocupar de elegir y gestionar dichas instancias ni de crear el clúster por sí mismo, lo que redunda en un ahorro de tiempo y recursos.

Ofrece algunas de las **herramientas más usadas del ecosistema Hadoop: HDFS, YARN, Tez, Zookeeper, Spark** (incluyendo todos sus *frameworks*: Spark Streaming, Spark SQL, MLlib y GraphX), **HBase, Presto, Flink, Hive**, entre otras varias (accede al siguiente [enlace](#) para ver una lista completa y actualizada). Además proporciona **EMR notebooks**, el equivalente a los Jupyter Notebooks, pero orientados a EMR y, como cabe esperar, con muchos puntos de integración con otros servicios AWS (las instancias EC2, que son los nodos del clúster; VPC para configurar la red virtual que conecte los nodos, S3 para almacenar datos de entrada y salida, CloudWatch para monitorizar funcionamiento y configurar alarmas, IAM para configurar permisos, CloudTrail para auditar las peticiones que se han hecho a un servicio y Data Pipeline

para programar los inicios y paradas del clúster).

Un clúster EMR tiene algunas peculiaridades:

- ▶ Existe **un único nodo denominado máster**, que está presente siempre en cualquier clúster y que no es más que una instancia EC2 que se encarga de su gestión (monitoriza los nodos activos y caídos, reparte tareas y datos, y lleva a cabo un seguimiento de las tareas ejecutadas).
- ▶ Puede haber **uno o más core nodes**, que son los nodos que componen el sistema de ficheros HDFS y pueden, además, ejecutar tareas. Permiten ser escalados, tanto para aumentar como para disminuir el tamaño del clúster.
- ▶ Puede haber **cero o más task nodes**, que son nodos que únicamente ejecutan tareas, pero no almacenan ninguna información (no son parte de HDFS), por lo que no hay riesgo de pérdida de datos si se eliminan. Por tanto, son buenos candidatos para ejecutarse en *spot instances* (instancias EC2 temporales con coste más reducido, que AWS puede detener en cualquier momento en que requiera los recursos de dicha instancia), los cuales reducen considerablemente el coste.

Hay varias formas de ejecutar un clúster EMR:

- ▶ **Transient clúster.** Destinado a casos de uso donde existe una secuencia predefinida de tareas por realizar, las cuales, una vez finalizadas, hacen que el clúster se termine y elimine automáticamente, lo que supone un ahorro en costes.
- ▶ **Long-running clúster.** Precisa terminar y eliminar el clúster manualmente cuando no se quiere usar más, lo que lo hace ideal para realizar consultas *ad hoc* o experimentar con datos que no se sabe muy bien de antemano cómo van a ser.

Cuando se ejecuta un clúster, se eligen los servicios que se van a usar y se pueden añadir ***spot instances*** (instancias EC2 más baratas, pero que AWS puede parar en cualquier momento y que duran, como máximo, un día) en ciertos momentos para ejecutar grandes tareas, o usar **instancias EC2 reservadas** (IR, con contrato

mínimo de un año, durante el que la capacidad de cómputo está reservada y garantizada) para tareas a largo plazo, a fin de ahorrar costes. De cualquier forma, nos podemos conectar al nodo máster para ejecutar tareas y ordenarlas desde la consola.

Como decíamos, HDFS está disponible en EMR (el mismo que hemos estudiado en los primeros temas), pero hay que tener en cuenta que este HDFS es temporal, lo que significa que, en cuanto el clúster se detiene, los datos almacenados en HDFS desaparecen con él (porque el clúster, esto es, las instancias EC2 se paran y eliminan). Por tanto, es peligroso utilizar HDFS con EMR, ya que nos arriesgamos a perder datos si el clúster se detiene o a incurrir en un alto coste por mantener su ejecución continua, aunque solo sea para preservar HDFS y, con él, los datos que almacena. Por ello, AWS ofrece tres alternativas para almacenar datos:

- ▶ **EMR File System (EMRFS)**, un sistema que permite acceder a S3 como si fuera HDFS.
- ▶ El **sistema de ficheros local**, que no está distribuido y es menos eficiente. Se usa generalmente en el nodo máster.
- ▶ **EBS para HDFS**, donde EBS se emplea para almacenar los datos como si fuera HDFS, pero sin necesidad de tener instancias EC2 ejecutándose continuamente.

Cabe mencionar que EMR se factura por horas de uso. A ello hay que sumar el cobro adicional por el uso de las instancias EC2 que componen el clúster, por lo que debe tenerse cuidado con los costes incurridos. Finalmente, hay que destacar que EMR está protegido por sistemas de seguridad como: políticas y roles IAM, Kerberos para autenticación con gran nivel de seguridad, SSH para la comunicación entre los usuarios y las instancias del clúster EMR.

## [Amazon Managed Streaming for Apache Kafka \(Amazon MSK\)](#)

Otro servicio muy cercano también a las tecnologías vistas en clase, pero en su

modalidad gestionada por AWS, es **Amazon Managed Streaming for Apache Kafka (Amazon MSK)**. Como su propio nombre indica, aligera la complejidad de montar el clúster Kafka y deja al usuario con la única tarea de escribir las aplicaciones (productores y consumidores) Kafka a través de su API nativa.

## AWS Glue

Pasando ya a servicios similares a los que hemos visto anteriormente, pero propiedad de AWS, podemos mencionar **AWS Glue**, enfocado en **tareas ETL** (*extract, transform, load*). Proporciona un editor visual que señala las rutas donde encontrar los datos que se van a analizar, tanto en AWS como en otras fuentes accesibles mediante JDBC. AWS Glue se encarga de descubrir dichos datos y metadatos asociados (esquemas de tablas), y de almacenar toda esa información en AWS Glue Data Catalog, a fin de realizar las tareas ETL sobre ellos o para que otros muchos servicios AWS puedan acceder y consultar dichos datos. Cuando se indica una ruta de fuente de datos a AWS Glue (que puede ser JSON, CSV o Parquet, entre otros), este la inspecciona para extraer su esquema y lo anota en el catálogo. Posteriormente, se pueden solicitar transformaciones sobre los datos recogidos en este.

**AWS Glue Data Catalog** es el repositorio de metadatos sobre todos los datos de una cuenta AWS. La idea es que pueda indexar automáticamente todos los esquemas de Amazon S3, Amazon RDS y Amazon Redshift, así como bases de datos desplegadas en instancias EC2. Además, se ayuda de *crawlers* para construir el catálogo, así como las fuentes de datos mencionadas previamente. Los *crawlers* analizan todos los datos almacenados en la cuenta para inferir esquemas y particiones, y funcionan con ficheros JSON, Parquet o CSV. También es posible programarlos para que se ejecuten en ciertos momentos o bajo demanda.

Por su parte, **Glue ETL** es el componente que permite transformar, limpiar y enriquecer datos antes de realizar ningún análisis. El código ETL se escribe en

Python o Scala, mientras que las transformaciones se ejecutan en una plataforma *serverless* de Spark. Todas ellas (*jobs*) se pueden programar gracias a **Glue Scheduler** o automatizar bajo la ocurrencia de eventos por medio de **Glue Triggers**. Entre las transformaciones proporcionadas por Glue ETL, podemos encontrar: Bundled Transformations, para eliminar registros nulos o vacíos (DropFields, DropNullFields); funciones *filter* (filtro), *join* (unión de igualdad) o *map* (mapeo); y Machine Learning Transformations, como FindMatches ML, para encontrar registros duplicados o coincidentes en un conjunto de datos, aunque estos no tengan identificador único y los campos no coincidan exactamente (deduplicar). También conversiones de formatos (CSV, JSON, Avro, Parquet, ORC, XML) y cualquier transformación de Apache Spark (por ejemplo, K-Means).

## AWS Data Pipeline

**AWS Data Pipeline** es un servicio para gestionar dependencias entre tareas, reintentos y notificaciones de errores durante tareas de ETL. Los destinos de los datos pueden ser S3, RDS, DynamoDB, Redshift y EMR, mientras que las fuentes de datos pueden ser servicios de AWS o estar *on-premises*. Por supuesto, ofrece alta disponibilidad.

Un caso de uso posible es el siguiente. Imaginemos que tenemos una base de datos RDS que contiene datos sobre los que queremos aplicar cierto algoritmo de *machine learning*. Para esto, necesitamos mover los datos de RDS a S3, a fin de usar posteriormente Amazon SageMaker (el servicio para construir modelos *machine learning*, que veremos más adelante). Con este objetivo, se puede crear un *pipeline* con AWS Data Pipeline, que generaría las instancias EC2 necesarias para mover los datos desde RDS, DynamoDB u otras fuentes a S3. Es decir, Data Pipelines orquesta y mueve todos los datos necesarios.

A simple vista, podría parecer el mismo servicio que Glue, pero existen ciertas diferencias. La principal es que, con Glue ETL, usamos Spark, que se centra en el

proceso ETL aprovechando el catálogo de datos de Glue, sin preocuparse por configurar o gestionar recursos (servidores). Por su parte, Data Pipeline es un servicio de orquestación que ofrece más control sobre el entorno, los recursos de computación que ejecutan el código y sobre el código en sí, que no tiene por qué ser Spark. Esto permite acceder a las instancias EC2 o EMR creadas por el *pipeline* y configurar y gestionar a más bajo nivel que con Glue.

## Redshift

**Redshift** es un sistema de *data warehousing* orientado a columna, que permite realizar **analíticas interactivas** utilizando **lenguaje SQL**. Es un sistema OLAP, (OnLine Analytical Processing), es decir, su foco principal son las consultas complejas dirigidas a aplicaciones de BI (Business Intelligence) y de reporte, y está optimizado para consultas de sólo lectura. Necesita cargar los datos desde S3, o utilizar Redshift Spectrum para hacer las consultas directamente sobre S3 sin necesidad de cargar los datos.

Es un sistema que hay que provisionar de antemano. La diferencia con RDS, además de que RDS es almacenamiento en base de datos relacional mientras que Redshift lee los datos desde S3 sin almacenarlos en ningún sitio adicional, es que RDS está orientado a almacenamiento por filas (mientras que Redshift está orientado a columnas) y RDS está orientado a OLTP (OnLine Transaction Processing), es decir, a consultas transaccionales que permiten no sólo leer, sino también insertar, actualizar y eliminar datos, y que generalmente proporcionan respuestas rápidas.

## Amazon Athena

**Amazon Athena** es un servicio *serverless* (es decir, no hace falta provisionar la arquitectura *hardware* que hay por debajo, sino que AWS se encargará de ello) para realizar consultas interactivas a datos almacenados en S3, sin necesidad de cargarlos. Soporta muchos formatos (CSV, JSON, ORC, Parquet o Avro) y puede trabajar con datos estructurados, semiestructurados y no estructurados. Por debajo,

está funcionando una plataforma de código libre llamado Presto.

Se puede usar, por ejemplo, para realizar consultas sobre *logs* de interacciones con páginas web o sobre datos intermedios almacenados en S3 antes de cargarlos en Redshift, o para analizar *logs* almacenados en S3, procedentes de otros servicios de AWS. También permite realizar todas estas consultas desde *notebooks* de Jupyter, Zeppelin o RStudio, así como la integración con QuickSight (servicio de visualización de AWS) o con herramientas de visualización que soporten los protocolos ODBC o JDBC. Algunos casos de uso contraindicados son la generación de reportes o visualizaciones con una gran carga de formato (para eso está pensado QuickSight), o la realización de tareas ETL (para lo que ya existe Glue).

Un flujo de trabajo habitual puede ser el siguiente: tenemos una serie de datos en S3, cuyo esquema ha sido extraído por AWS Glue y guardado en su catálogo, y Athena puede hacer uso de ellos para visualizarlos después en QuickSight, por ejemplo.

Como cabe esperar, en el plano de la seguridad, Athena está protegido también por políticas de control de acceso, encriptación en S3 (en reposo) y uso de TLS (*transport layer security*) en la comunicación entre Athena y S3, de forma que los datos están también encriptados en tránsito.

## Amazon Kinesis

Una alternativa a Apache Kafka y Spark Streaming es **Amazon Kinesis**, una tecnología propietaria y gestionada por AWS. Representa una herramienta crucial para aplicaciones como procesado de *logs*, métricas, IoT, flujos de clics y procesamiento de flujos de datos en general (*streaming*). A pesar de ser un servicio de ingestión de datos en tiempo real, estos se siguen replicando de forma síncrona en tres AZ para evitar perder datos en caso de fallos.

Kinesis alberga varios servicios bajo su paraguas:

- ▶ **Kinesis Streams.** Ingestión de datos a gran escala con baja latencia.
- ▶ **Kinesis Analytics.** Realiza analíticas en tiempo real de flujos de datos (Kinesis Streams, Kafka, Amazon MSK) usando consultas SQL (parecido a Spark Structured Streaming).
- ▶ **Kinesis Firehouse.** Carga flujos de datos en S3, Redshift, ElasticSearch y Splunk.
- ▶ **Kinesis Vídeo Streams.** Servicio para *streaming* de vídeo en tiempo real y analíticas asociadas.

Podemos ver una arquitectura sencilla compuesta por estos servicios en la figura 6. Como ya sabemos, es posible tener diferentes flujos de datos en tiempo real, ya sean dispositivos IoT, clics en una aplicación web, métricas y *logs*, etc. En el caso del ejemplo, el flujo de datos proviene de un medidor de pulso, cuyos datos son gestionados por otros servicios denominados AWS Greengrass y AWS IoT (estos quedan fuera del alcance del temario, pero podemos verlos como flujos de datos genéricos). Estos flujos de datos se adquieren a través de Kinesis Streams (como ocurría con Apache Kafka).

A partir de aquí, podemos ver dos ramas de acción. En la rama inferior, los datos ingeridos en bruto se almacenan en un *bucket* S3 a través de Kinesis Firehouse. Estos son posteriormente procesados mediante tareas ETL por AWS Glue, que almacena el resultado en otro *bucket* S3. Por último, Amazon Athena realiza consultas SQL sobre los resultados de las operaciones ETL, mientras que las analíticas finales se muestran en Amazon Quicksight, la herramienta de visualización de AWS (que también queda fuera del alcance del temario). En la rama superior, por su parte, Amazon Kinesis Analytics realiza la media del pulso sobre los datos recibidos y envía el resultado a otro flujo de datos consumido por Amazon Lambda. Amazon Lambda compara esta media continua con información del paciente (almacenada en DynamoDB) y envía mensajes de alerta a través de Amazon SNS en caso de que el pulso rebase cierto límite, que depende de los atributos del

paciente (edad, patologías previas, etc.).

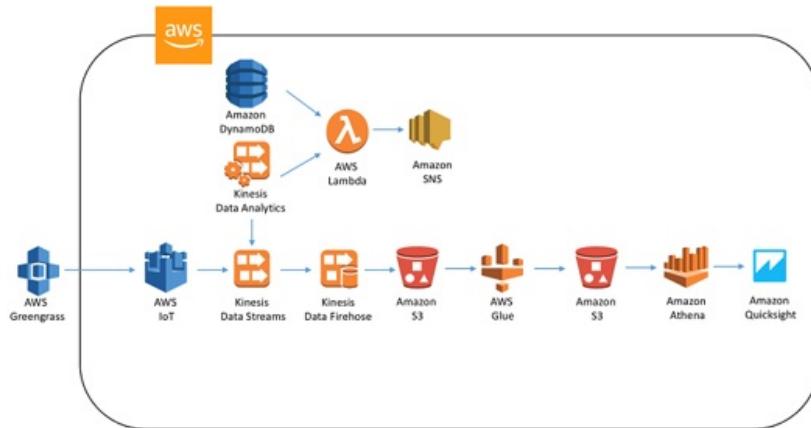


Figura 6. Ejemplo de flujo de datos gestionado con tecnologías AWS, entre ellas, AWS Kinesis. Fuente: <https://aws.amazon.com/blogs/big-data/how-to-build-a-front-line-concussion-monitoring-system-using-aws-iot-and-serverless-data-lakes-part-1>

Viendo un poco más en detalle el funcionamiento de cada uno de los servicios bajo el paraguas de Kinesis, los flujos en **Kinesis Streams** están divididos en *shards* o particiones (como las particiones de Apache Kafka), adonde diferentes productores inyectan datos y desde donde múltiples consumidores se abastecen de los datos de los *shards* a los que están suscritos. Aunque es un servicio gestionado por AWS, cada *shard* tiene que estar provisionado de antemano; es decir, se requiere cierta planificación para conocer cuántos *shards* vamos a tener y cuál es la capacidad de cada uno.

Cada registro insertado en Kinesis puede ser de hasta 1 MB y, por defecto, los registros se guardan durante 24 horas, aunque es posible configurarlos con un máximo de 7 días, de forma que permite reprocesar los datos en caso necesario. También escribir código propio tanto para los consumidores como para los productores. Asimismo, trabaja casi en tiempo real (unos 200 ms para la opción clásica y alrededor de 70 ms para la opción mejorada). Al igual que ocurría en Apache Kafka, diferentes aplicaciones (consumidores) pueden consumir de un

el mismo flujo. Por otro lado, una vez que los datos se insertan en Kinesis, no se pueden eliminar (son inmutables). Los productores pueden enviar 1 MB/s como máximo o 1000 mensajes/s por *shard*. Por su parte, cada *shard* puede servir hasta 2 MB/s de mensajes (agregado en todos los consumidores que lean de dicho *shard*) o cinco llamadas a la API.

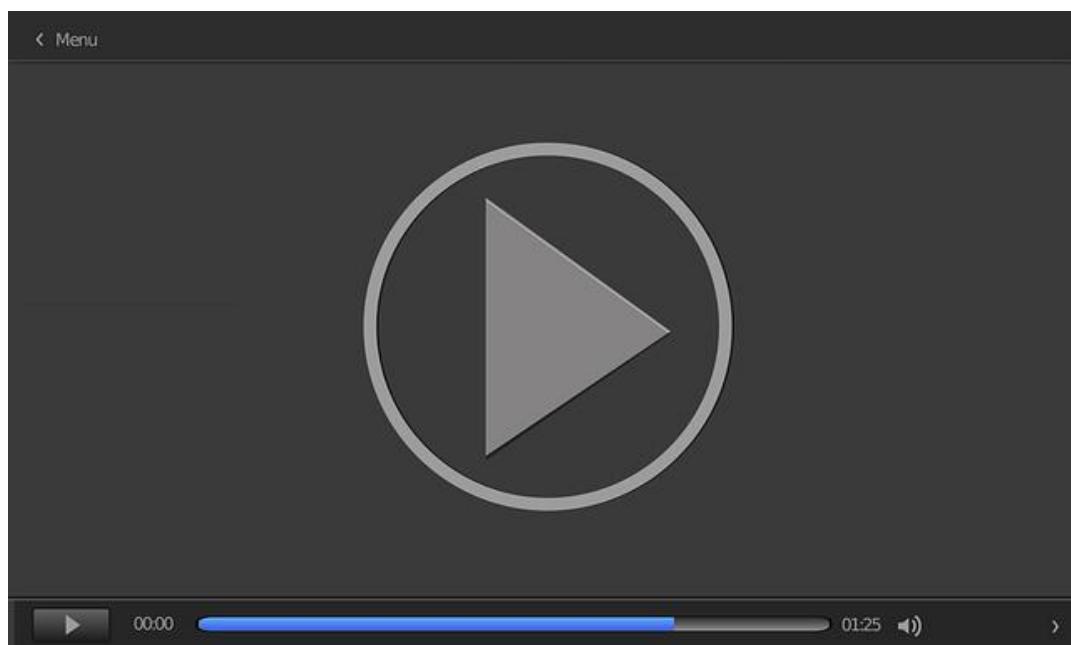
**Kinesis Data Firehouse** es un servicio operado por AWS, que puede gestionar datos casi en tiempo real (con un mínimo de 60 s de latencia para bloques de datos que no están completos), procedentes de flujos en Redshift, Amazon S3, ElasticSearch y Splunk. Ofrece escalado automático y soporta muchos formatos de mensaje, además de ser capaz de convertir datos CSV y JSON a Parquet y ORC, así como de comprimir a GZIP, ZIP y SNAPPY. También es posible hacer transformaciones de datos a través de AWS Lambda. El coste se basa solo en el volumen de datos que atraviesa Kinesis Firehouse, independientemente de las transformaciones que se hagan sobre ellos.

**Kinesis Data Analytics** puede tomar datos tanto de Amazon MKS, Kinesis Data Streams o Kinesis Data Firehouse, y realizar diferentes análisis de estos. Para ello, utiliza consultas SQL como, por ejemplo, `SELECT STREAM itemId, count(*) FROM SourceStream GROUP BY itemId;`). Las consultas devuelven un flujo de salida, que puede ir hacia Kinesis Data Streams, S3, Kinesis Data Firehouse o Redshift. Los casos de uso más típicos de Kinesis Analytics son:

- ▶ Streaming ETL, para seleccionar columnas y realizar transformaciones sencillas en los datos del flujo.
- ▶ Reporte continuo de métricas, por ejemplo, para tener una visualización de resultados que se actualiza continuamente.
- ▶ Analíticas en tiempo real, para buscar ciertos criterios según llegan los datos y crear alertas o filtros.

Con Kinesis Analytics, se paga solo por los recursos consumidos, aunque su coste es bastante alto. Está totalmente gestionado por AWS, permite escalar automáticamente su capacidad y está protegido por mecanismos de seguridad a través de roles de usuario.

Para finalizar este epígrafe, os proponemos visualizar el siguiente vídeo. En él, vamos a ver el entrenamiento y despliegue de modelos en Amazon Web Services (AWS) en un clúster de EMR, utilizando SageMaker.



---

Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=d665ce96-164c-4007-ab94-ac8b00c81329>

---

Vídeo. Despliegue de modelos con AWS, EMR y SageMaker (I).

## 9.10. Machine learning e inteligencia artificial

Los servicios de *machine learning* e inteligencia artificial se están convirtiendo en un elemento esencial de muchas aplicaciones y todos los proveedores de servicios los incluyen entre su oferta. En el caso de AWS, lo hace a dos niveles. En el nivel más bajo, ofrece servicios para diseñar y entrenar modelos propios, monitorizarlos, llevar un seguimiento de experimentos o incluso dejar que sea el propio servicio el que se aventure a encontrar el mejor modelo y los parámetros asociados. Todo esto es lo que ofrece el servicio AWS Sagemaker y sus diferentes componentes, que veremos con algo más de detalle.

A un nivel mucho más alto, que no requiere conocimientos específicos de *machine learning*, sino únicamente de los requisitos de la aplicación, se encuentran numerosos servicios de inteligencia artificial enfocados en distintos aspectos: texto, voz o fraude, entre otros. Como en los casos anteriores, todos estos recursos mencionados hasta ahora tienen que ver con la oferta de servicios gestionados por Amazon.

AWS proporciona asimismo la opción IaaS, donde existen instancias EC2 configuradas con *hardware* específico (como GPU), diseñadas para poder desplegar el entorno de desarrollo del modelo que prefiera el usuario. En cuanto a los servicios PaaS, por su especificidad, veremos con algo más de detalle aquellos que se comentaban al principio, tanto a bajo como a alto nivel.

Como se mencionaba anteriormente, Amazon SageMaker es un servicio gestionado por AWS que brinda la oportunidad de construir, entrenar y desplegar modelos de *machine learning*, sin que el usuario deba preocuparse por la infraestructura necesaria para ello. Aunque su uso precisa de conocimientos de *machine learning*, facilita enormemente el desarrollo de modelos gracias a los diversos componentes integrados, que cubren todo el ciclo de vida, desde el diseño hasta la monitorización

del modelo resultante.

- ▶ **Amazon SageMaker Ground Truth** ayuda en la construcción de conjuntos de datos correctamente etiquetados para entrenar el modelo. Proporciona acceso a los servicios de etiquetado de Amazon Mechanical Turk, así como flujos de trabajo e interfaces para facilitar dicho proceso.
- ▶ **Amazon SageMaker Studio** es un entorno de desarrollo integrado (IDE) web para realizar todo el proceso de construcción, entrenamiento y despliegue de modelos. Permite cargar datos, usar *notebooks*, entrenar y ajustar modelos, comparar resultados de diferentes experimentos y, finalmente, desplegar modelos en producción desde una única herramienta. Los *notebooks* mencionados son Amazon SageMaker Studio Notebooks, es decir, *notebooks* de Jupyter con acceso a S3 para leer los datos de entrenamiento, las librerías necesarias (Spark, scikit-learn, numpy, pandas, TensorFlow) y una amplia variedad de modelos preconstruidos, y con capacidad de desplegar instancias tanto para entrenar como de producción.

Para crear tareas de entrenamiento, hay que indicar la URL del *bucket* S3 donde se encuentran los datos de entrenamiento, los recursos de computación que vamos a necesitar para entrenar (qué tipo de instancias EC2, con GPU, etc.), la URL de S3 para el *bucket* donde se guardarán los datos de salida y la ruta al ECR donde se ejecutará el código. Para entrenar, podemos optar por utilizar algoritmos ya disponibles, Spark MLlib, código Python propio en TensorFlow o MXNet, imágenes Docker propias o algoritmos que se pueden adquirir en AWS Marketplace.

- ▶ **Amazon SageMaker Autopilot** ofrece la posibilidad de construir modelos *machine learning* automáticos (denominados en algunos sitios como autoML). Es capaz de inspeccionar los datos en bruto, aplicar procesados de variables, probar diferentes algoritmos y combinaciones de estos, y reportar los resultados de los experimentos para justificar la elección de la mejor combinación. Además, proporciona el acceso al usuario para visualizar los resultados de todos los pasos.

- ▶ **Amazon SageMaker Experiments** permite explorar los diferentes experimentos diseñados (combinaciones de conjunto de datos, algoritmos y parámetros) junto con los resultados del entrenamiento, a fin de decidir qué modelo elegir.
- ▶ **Amazon SageMaker Model Monitor** proporciona monitorización del funcionamiento del modelo y del potencial desvío de los nuevos datos respecto a los del entrenamiento, una vez desplegado el modelo en producción.
- ▶ Respecto al despliegue, Amazon SageMaker ofrece varias posibilidades: **Amazon SageMaker Components for Kubeflow Pipelines** para orquestar todo el proceso; **instancias específicas como Amazon Inf1**, que poseen chips optimizados para el proceso de inferencia (predicción) del modelo desplegado, o Amazon SageMaker Neo para desplegar modelos *edge* (más cercanos a donde se van a necesitar las predicciones), entre otras muchas opciones.

En la figura 7 podemos observar qué componentes se pueden usar en cada paso del ciclo de vida de un modelo de *machine learning*.



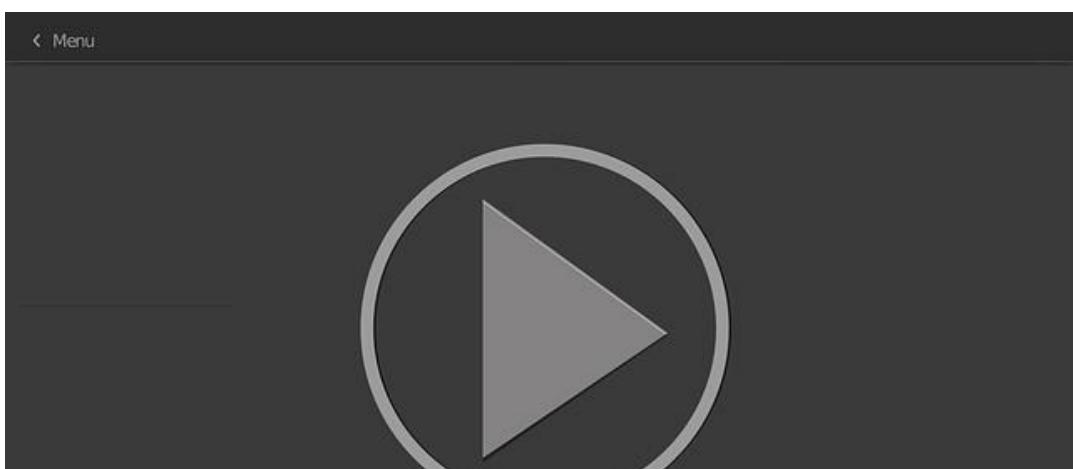
Figura 7. Servicios AWS SageMaker para cada paso del ciclo de vida de un modelo de *machine learning*.

Fuente: <https://aws.amazon.com/sagemaker>

En cuanto a los servicios de inteligencia artificial que no precisan conocimientos por parte del usuario, sino sencillamente el uso de la API correspondiente, encontramos también una larga lista. Algunos de los principales son:

- ▶ Amazon **Comprehend**: para procesado de lenguaje natural, que permita extraer conocimiento útil y relaciones entre entidades en textos no estructurados.
- ▶ Amazon **Translate**: ofrece traducción en múltiples lenguas en tiempo real.
- ▶ Amazon **Polly**: es el servicio de traducción texto a voz (*text-to-speech*).
- ▶ Amazon **Transcribe**: es el servicio opuesto a Polly, es decir, traducción voz a texto (*speech-to-text*).
- ▶ Amazon **Lex**: proporciona agentes conversacionales (*chatbots*).
- ▶ Amazon **Rekognition**: analiza imágenes y vídeos para catalogar elementos, automatizar flujos de trabajo y extraer significado de ellos.
- ▶ Amazon **Personalize**: como su propio nombre indica, personaliza la experiencia de los consumidores usando *machine learning* y el conocimiento obtenido de su web de compra.
- ▶ Amazon **Fraud Detector**: identifica operaciones *online* potencialmente fraudulentas, basándose en el conocimiento y la experiencia adquiridos en su web de compraventa.

Finalizamos el tema con dos vídeos que completan el propuesto en el epígrafe anterior. En el primero, vamos a productivizar un *pipeline* con EMS.



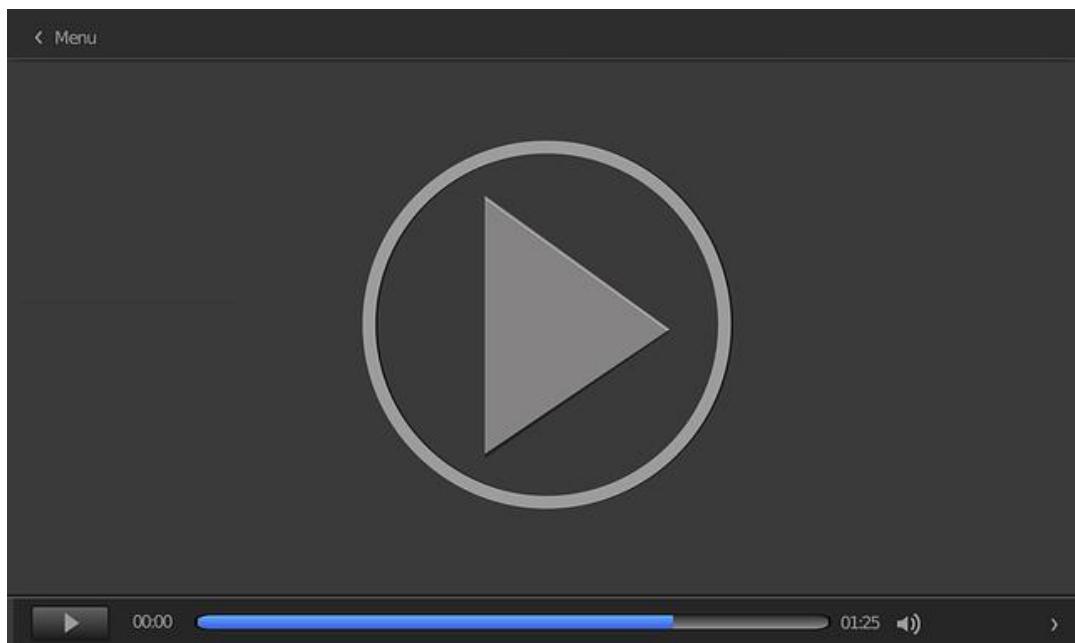


Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=a7148bee-123c-4967-9264-ac8b00e48e7e>

Vídeo. Despliegue de modelos con AWS, EMR y SageMaker (II).

A continuación, una vez que ha terminado de entrenar el *pipeline* de Spark, procedemos a explicar cómo desplegarlo en Amazon SageMaker.



Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=1cacf69d-f051-4218-89e6-ac8b014f040c>

Vídeo. Despliegue de modelos con AWS, EMR y SageMaker (III).



## Documentación en línea de Amazon Web Services

Amazon Web Services (2021). *AWS Documentation*. <https://docs.aws.amazon.com/>

Amazon Web Services (2021). *Introducción a AWS. Aprenda los aspectos fundamentales y comience a crear en AWS ahora.* <https://aws.amazon.com/getting-started>

El recurso más completo y actualizado sobre Amazon Web Services es su propia documentación en línea. Dada la rapidez con la que los diferentes servicios proporcionados evolucionan, aparecen y dejan de estar disponibles, es difícil encontrar libros que se mantengan actualizados más allá de unos pocos meses. Incluye tanto documentación sobre cada uno de los servicios concretos como caminos de aprendizaje que muestran la forma en que los diferentes recursos pueden interactuar para conseguir objetivos concretos, como servicios de *data science*, DevOps o Database Administrator.

1. ¿Cómo se distribuyen los recursos de la infraestructura de AWS?

  - A. Se dividen en zonas, que, a su vez, tienen dos o más subzonas.
  - B. Se dividen en regiones, que, a su vez, engloban dos o más zonas de disponibilidad.
  - C. Se dividen en zonas de disponibilidad, que, a su vez, contienen dos o más centros de datos.
  - D. No existe ninguna división, todos los recursos son globales e indistinguibles.
2. Una empresa quiere utilizar los servicios de AWS para almacenar datos personales y sensibles de sus clientes. ¿Cuál es el elemento más limitante a la hora de determinar dónde almacenar dichos datos?

  - A. El coste, ya que, dependiendo de dónde se almacenen dichos datos, este puede ser mayor o menor.
  - B. La latencia, ya que se tardaría mucho en obtener los datos si están almacenados lejos de donde se realiza la consulta.
  - C. La legislación, porque, al ser información sensible, solo se pueden almacenar en lugares muy concretos para no incurrir en delitos.
  - D. Todos los elementos anteriores tienen la misma importancia y hay que tenerlos en cuenta por igual.

- 3.** Una de las mayores ventajas de usar AWS como proveedor de servicios *cloud* es:

  - A. Que tiene responsabilidad sobre toda la infraestructura, los servicios y los datos necesarios para desplegar nuestras aplicaciones.
  - B. Que tiene responsabilidad sobre toda la seguridad concerniente a nuestra aplicación, desde el *firewall* al control de acceso.
  - C. Que tiene la responsabilidad sobre la infraestructura y garantiza ciertos niveles de servicio al respecto.
  - D. Todas las afirmaciones previas son correctas.
- 4.** ¿Qué afirmación sobre las instancias de cómputo EC2 es incorrecta?

  - A. Permiten elegir la imagen (AMI) que instalar en ellas de entre una colección predefinida o una proporcionada por el usuario.
  - B. Poseen un conjunto predeterminado de configuraciones de cómputo, memoria y red, de donde escoger obligatoriamente la configuración predefinida que se deseé.
  - C. Se pueden contratar tantas instancias EC2 como se deseé.
  - D. Cada instancia EC2 está ligada a un servicio EBS para almacenamiento persistente.
- 5.** ¿Cómo se interconectan los servicios AWS que contrata un usuario?

  - A. Mediante una red global que comparten todos los servicios contratados por todos los usuarios en AWS.
  - B. Mediante una red propia del usuario que conecta las direcciones IP de los servicios contratados.
  - C. Los distintos servicios contratados son independientes y autocontenido, por lo que no necesitan ni pueden comunicarse con otros.
  - D. Mediante el servicio de interconexión AWS VPC.

- 6.** Indica qué caso de uso no es propio de S3:
- A. Sistema de arranque de una instancia EC2.
  - B. Almacenamiento de ficheros accesibles desde instancias EC2.
  - C. Almacenamiento de ficheros accesibles desde un navegador web.
  - D. Almacenamiento de archivo de ficheros de escaso acceso.
- 7.** Se quiere desplegar una base de datos relacional de forma rápida y que no suponga una carga de mantenimiento para el departamento de IT, más allá de la gestión de los datos contenidos. ¿Qué servicio AWS escogerías?
- A. Instancia EC2 e instalación de MySQL.
  - B. AWS RDS.
  - C. AWS DynamoDB.
  - D. AWS ECS e instalación de MySQL.
- 8.** ¿Cuál de las siguientes opciones es la mejor para desplegar un servicio de almacenamiento distribuido en AWS?
- A. Varias instancias EC2 sobre las que el usuario instala un clúster Hadoop, que incluye HDFS.
  - B. Varios contenedores ECS sobre los que el usuario instala un clúster Hadoop, que incluye HDFS.
  - C. Un clúster EMR, con su propio sistema de almacenamiento distribuido.
  - D. Un clúster EMR, con el sistema de almacenamiento HDFS.
- 9.** Amazon SageMaker es un servicio de AWS destinado a:
- A. Construir y entrenar modelos de *machine learning* desde cero.
  - B. Utilizar modelos de *machine learning* preconstruidos.
  - C. Realizar consultas interactivas sobre grandes conjuntos de datos.
  - D. Catalogar todos los datos existentes en los diferentes servicios AWS.

- 10.** Si se quieren manejar flujos de datos en tiempo real, ¿qué servicio AWS no sería adecuado?
- A. Amazon Kinesis Streams.
  - B. Amazon MSK.
  - C. Amazon Redshift.
  - D. Instancias EC2 con Kafka instalado.

Ingeniería para el Procesado Masivo de Datos

---

## Tema 10. Cloud computing III

# Índice

## Esquema

### Ideas clave

- 10.1. Introducción y objetivos
- 10.2. Google Cloud Platform
- 10.3. Regiones y zonas
- 10.4. Servicios transversales: seguridad y gestión
- 10.5. Servicios de computación
- 10.6. Servicios de red
- 10.7. Servicios de almacenamiento
- 10.8. Bases de datos
- 10.9. Servicios de big data y analítica
- 10.10. Machine learning e inteligencia artificial

### A fondo

Documentación en línea de Google Cloud Platform

### Test

# Esquema

GOOGLE CLOUD PLATFOR M (GCP)	
Conceptos	Servicios
Organización	
Computación	
Almacenamiento	
Bases de datos	
Big data & analítica	
Machine learning & IA	
Regiones geográficas	Cloud SQL Base de datos relacional Varios motores: - MySQL - PostgreSQL - Microsoft SQL Server
Formadas por 2 o más Zonas = centros de datos	Cloud Storage Almacenamiento objetos Bucket ~ directorio virtualmente ilimitado Modalidades por tiempo almacenamiento
Incrementa disponibilidad y tolerancia a fallos	Persistent Disk Almacenamiento orientado a bloque Cada instancia VM ligada a un único disco Puede ser compartido
Responsabilidad	Google Kubernetes Engine (GKE) Clúster contenedores Kubernetes Modelo de responsabilidad compartido
	Dataproc Clúster Hadoop gestionado
	BigQuery Consultas OLAP
	Dataflow Trabajos ETL totalmente gestionados
	DataLab Jupyter Notebooks integración con otros servicios
	Cloud Pub/Sub Streaming gestionado
	Cloud Firestore No relacional Orientada a documento
	Filestore Almacenamiento orientado a ficheros Compartido entre varias instancias VM
	AI Platform Requiere conocimientos Data Labelling BigQuery Datasets AutoML Training AI Explanations Vizier Prediction Pipelines
	Servicios IA No precisan conocimientos Vision Video Translation NLP DialogFlow Text-to-Speech Speech-to-text Recommendation

## 10.1. Introducción y objetivos

En los temas previos, se ha definido el concepto de *cloud computing* y se han explorado dos de los principales proveedores de computación en la nube, Microsoft Azure y Amazon Web Services, junto con el catálogo de servicios que ofrecen, con especial foco en aquellos relacionados con *big data*, analítica y *machine learning*.

Para completar esta panorámica general sobre el *cloud computing*, en el presente capítulo profundizaremos en el tercer proveedor principal de servicios *cloud*: Google Cloud Platform. Al igual que con los dos vistos anteriormente, se va a detallar, en primer lugar, la organización de los recursos y servicios transversales. A continuación, se examinarán los servicios a nivel infraestructura (IaaS), para ver después explorar los de plataforma y aquellos más relacionados con las tecnologías *big data*, de analítica y *machine learning*.

De esta forma, los objetivos a perseguir en este tema son los siguientes:

- ▶ Conocer Google Cloud Platform (GCP) como uno de los principales proveedores de servicios de *cloud computing*.
- ▶ Explorar el abanico de servicios *cloud* que proporciona GCP.
- ▶ Estudiar en detalle los servicios IaaS que ofrece GCP y sobre los que se pueden desplegar las tecnologías *big data* estudiadas en la asignatura.
- ▶ Conocer las alternativas PaaS y SaaS centradas en *big data*, analítica y *machine learning* que ofrece GCP de forma nativa.

## 10.2. Google Cloud Platform

Google Cloud Platform (GCP) nació de la necesidad de Google de ejecutar sus aplicaciones (Gmail, Drive...) a la escala que solicitaban los usuarios. Ello condujo al diseño e implementación de una plataforma que tuviera la capacidad de gestionar todas estas aplicaciones, a las que acceden cada día millones de usuarios a escala global y con diferentes requisitos de uso. Posteriormente, dicha plataforma se ha extendido para atender las demandas no solo de Google, sino también del público general, en un esquema de pago por uso. Basta con reflexionar un momento sobre los requerimientos que precisa una empresa como Google, a fin de ofrecer sus servicios a millones de usuarios con un nivel de calidad razonable (latencia, capacidad de cómputo y de almacenamiento, ingesta de datos, gestión de recursos...), para hacernos una idea del potencial de Google Cloud Platform.

Así, Google Platform ofrece un gran abanico de recursos en la nube mediante una infraestructura global que proporciona acceso desde casi cualquier parte del mundo, alta disponibilidad y seguridad, y un esquema de pago por uso (*pay-as-you-go*). Entre los servicios que ofrece, podemos encontrar los comunes a todas las plataformas de *cloud computing*, como son almacenamiento, procesamiento, redes y seguridad, y otros más específicos, relacionados con sectores concretos como analítica de datos, *DevOps*, bases de datos, inteligencia artificial, IoT y desarrollo móvil, entre muchos otros. Además, al igual que otras plataformas de *cloud computing*, permite desplegar modelos tanto completamente en la nube como de forma híbrida, combinando la ejecución de algunos servicios en sus propios servidores y otros en la nube, y dando la posibilidad de migrar cuando el usuario decida hacerlo.

Antes de profundizar el catálogo de servicios, es interesante conocer cómo se organizan los recursos de Google Cloud, así como conceptos como las regiones y

las zonas, ya que son transversales y afectan a todos los servicios. También existen otra serie de herramientas de gestión y seguridad que exploraremos, aunque sin entrar en detalle. Más adelante, pasaremos a explorar algunos de los servicios de Google Cloud; comenzaremos por servicios de computación, almacenamiento y redes, y seguiremos por bases de datos, *big data*, analítica y *machine learning*.

### 10.3. Regiones y zonas

Al igual que Azure y AWS, GCP también tiene disponibilidad global, es decir, sus servicios pueden ser accedidos desde prácticamente cualquier punto del mundo, para lo cual se requiere una distribución de los recursos que los hacen posibles. Con este fin, las localizaciones donde GCP ofrece sus servicios están divididas en **regiones y zonas**.

Una región es un área geográfica independiente, que suele ofrecer una baja latencia en los servicios accedidos (por debajo del milisegundo). Está compuesta por zonas, donde GCP tiene desplegados una serie de recursos (un centro de datos). La característica más importante de una zona es que se trata de un dominio único de fallo dentro de una región, es decir: si una zona falla, no afecta a las demás. Por tanto, para ofrecer aplicaciones tolerantes a fallos y con alta disponibilidad, es aconsejable desplegarlas en varias zonas de una región. Si, además, queremos ir un paso más allá y hacer las aplicaciones tolerantes a fallos en una región entera (por ejemplo, a causa de un desastre natural), GCP ofrece la posibilidad de desplegarlas en varias regiones.

Más concretamente, GCP ofrece sus servicios y recursos en tres modalidades:

- ▶ Recursos **zonales**. En esta modalidad, los servicios están desplegados en una única zona, por lo que, si esta deja de estar disponible, los servicios también dejarán de funcionar.
- ▶ Recursos **regionales**. Los servicios desplegados en esta modalidad están replicados en más de una zona dentro de una única región, de forma que aumenta la disponibilidad respecto a los recursos zonales.
- ▶ Recursos **multiregionales**. Algunos servicios gestionados por Google están desplegados no solo en varias zonas dentro de una región, sino también en varias

regiones. Esta opción optimiza la disponibilidad y la eficiencia, pero es también más costosa. No todos los servicios disponen de esta opción, solo unos pocos, que se pueden consultar en la documentación de GCP para tener la lista más actualizada.

Existen múltiples regiones repartidas a lo largo de todo el mundo y su número está en constante crecimiento. La figura 1 muestra las regiones a fecha de octubre de 2020, pero es conveniente consultar la documentación de GCP para conocer la relación de regiones y zonas más actualizada.



Figura 1. Regiones actuales (azul) y futuras (blanco) de GCP, a fecha de octubre de 2020. Fuente: <https://cloud.google.com/about/locations>

Al igual que ocurría con AWS, en GCP es necesario tener en cuenta ciertos aspectos a la hora de elegir la región o regiones donde desplegar los servicios que se desean ofrecer:

- ▶ **La latencia.** Se trata de elegir la región más cercana a los usuarios que vayan a utilizar los recursos y servicios en la nube de GCP, para que la distancia sea menor y, por tanto, la latencia experimentada también.
- ▶ **El coste** de los servicios en cada región, ya que, en función de la legislación, los impuestos u otras limitaciones en las diferentes zonas, el despliegue de recursos y servicios será más o menos gravoso, lo que también hará variar el precio para el

usuario que contrata los servicios en la nube correspondientes.

- ▶ La **legislación** vigente en relación con la localización de datos sensibles o con cualquier otro aspecto relativo al uso de servicios IT. Cabe recordar que leyes como, por ejemplo, el reglamento GDPR imponen ciertos requisitos estrictos en el uso, la distribución y el almacenamiento de datos sensibles de personas. Por tanto, es de capital importancia tener esto en cuenta a la hora de elegir la región donde desplegar servicios de almacenamiento o procesado de dicha información.
- ▶ La **disponibilidad** de los servicios, puesto que no todos los servicios están disponibles en todas las regiones. Para una lista actualizada de la disponibilidad de los servicios en las diferentes regiones, se recomienda consultar la documentación más actualizada de la plataforma en <https://cloud.google.com/about/locations>.

## 10.4. Servicios transversales: seguridad y gestión

**Una de las principales prioridades** sobre la que se sustentan y promocionan los servicios ofrecidos por GCP es la **seguridad**, basada en una infraestructura protegida por capas, control y monitorización inteligentes. En este ámbito, se pone especial énfasis en la privacidad y la transparencia, por lo que se delega siempre en el usuario (el que contrata los servicios) el control, el acceso y la gestión de los datos que estos manejan. De hecho, los servicios ofrecidos por GCP comparten todas las medidas de seguridad que toma Google para ejecutar sus propias aplicaciones en la infraestructura *cloud*. A continuación, repasaremos los principales aspectos de seguridad y gestión que rigen de forma transversal el catálogo de servicios que exploraremos posteriormente en el tema.

- ▶ **Seguridad operacional.** La seguridad es parte integral de las operaciones de Google. Para ello, busca de forma activa amenazas y vulnerabilidades de seguridad mediante diversas herramientas propias y comerciales. También está alerta sobre ataques *malware* que comprometan la integridad de las cuentas o provoquen robos de datos y accesos no autorizados a su red, mediante mecanismos de monitorización, prevención, detección y eliminación de este *software* malicioso. Entre los muchos aspectos monitorizados, están el tráfico de la red, las acciones de los empleados, así como bases de datos de vulnerabilidades. Cuando se detecta algún incidente de seguridad, existen procesos perfectamente definidos para gestionar potenciales efectos en la confidencialidad, integridad o disponibilidad de los sistemas.
- ▶ **Tecnología con la seguridad como prioridad desde su diseño .** En GCP, la seguridad y protección de los datos están entre los principales criterios de diseño. Como se comentaba previamente, la infraestructura está asegurada por capas. Empezando por las medidas de protección en los centros de datos (tarjetas de acceso, alarmas, seguridad del recinto, medidas biométricas, detección de intrusos,

*logs de acceso, informes de actividad, grabaciones de actividad, guardas de seguridad, fuertes restricciones de acceso a las instalaciones que salvaguardan los datos, a las que menos del 1 % de los empleados de Google puede acceder, etc.).*

Por otro lado, los centros de datos disponen de fuentes de alimentación redundantes, sistemas de climatización que mantienen la temperatura en rangos adecuados o sistemas de detección de incendios, entre otras medidas para asegurar la integridad de los recursos físicos. A nivel de red, cuentan con su propia red de fibra y con cables submarinos que les permiten ofrecer servicios de muy baja latencia a nivel global. En este sentido, se aseguran de que solo servicios y protocolos autorizados accedan y usen dicha red, mediante *firewalls* y listas de control de acceso, o rutado de tráfico a través de Google Front End (GFE), que permite la monitorización y detección de ataques de denegación de servicio (DDoS). Asimismo, los centros de datos soportan la seguridad de los datos en tránsito mediante el uso del protocolo TLS (proporcionado por servicios como Cloud Load Balancer o por el uso de túneles seguros con Cloud VPN). A nivel global, como comentábamos al principio del tema, los servicios se ofrecen en diferentes zonas y regiones para poder proporcionar tolerancia a fallos y desastres naturales.

- ▶ **Uso de datos.** Como ya se vio en otros proveedores, los datos son responsabilidad del usuario que contrata los servicios *cloud*, no de Google en este caso. Es decir: los datos son del usuario y, por tanto, Google no puede tener acceso a ellos ni procesarlos para fines de publicidad o venta a terceros.
- ▶ **Acceso de datos y restricciones.** Aunque estén almacenados en los mismos servidores físicos, Google se asegura de aislar a nivel lógico los datos de los diferentes usuarios. Existen diversos roles administrativos con privilegios específicos para acceder a los datos dentro de un proyecto GCP, de forma que distintos miembros del equipo que participan en el proyecto tengan acceso a los datos que necesiten según su rol.
- ▶ **Cumplimiento de regulaciones.** Dadas las diferentes regulaciones que rigen en las

distintas regiones, Google se actualiza constantemente para cumplir con las que apliquen en cada caso.

Los servicios de GCP, al igual que ocurría con AWS y Azure, se rigen por un **modelo de responsabilidad compartida en cuanto a seguridad se refiere**, que depende del servicio en concreto. La figura 2 representa las responsabilidades del usuario y del proveedor, según si el servicio está desplegado en las dependencias del usuario o si se ha optado por utilizar alguno de estos modelos de servicio: IaaS, PaaS o SaaS.

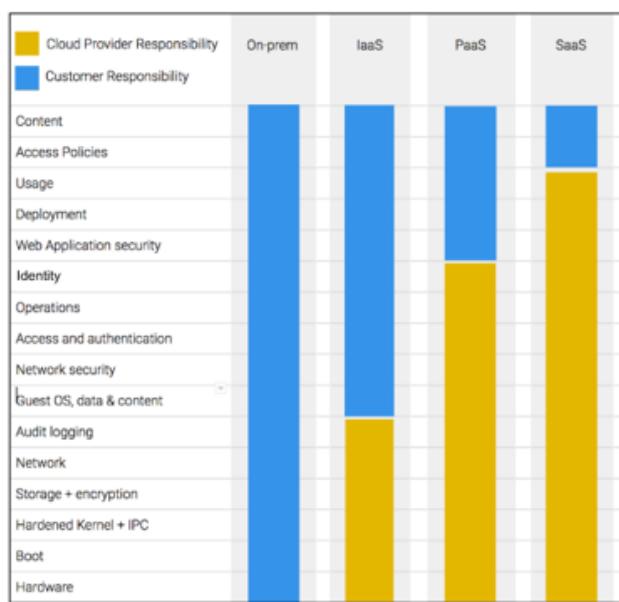


Figura 2. Modelo de responsabilidad compartida de la seguridad del sistema en GCP. Fuente: <https://services.google.com/fh/files/misc/google-cloud-security-foundations-guide.pdf>

Como parte del esquema de responsabilidad compartida, Google Cloud proporciona una serie de mecanismos de seguridad que implementa por sí mismo, así como unas guías y recomendaciones para que sea lo más sencillo posible para los usuarios cumplir con sus obligaciones en esta materia. Esta documentación se denomina **security blueprints** y forma parte de la jerarquía de seguridad de Google Cloud, la cual considera cuatro capas de seguridad:

- ▶ **Seguridad de la infraestructura** principal de Google Cloud. La infraestructura de recursos de Google proporciona seguridad a nivel de almacenamiento, privacidad de los datos, comunicación entre servicios, comunicación con usuarios por Internet, operaciones y administración.
- ▶ **Seguridad en los servicios y productos** de Google. Los productos de Google Cloud están diseñados con la seguridad como base e incluyen mecanismos como control de acceso, segmentación de servicios y protección de datos.
- ▶ Guías básicas de seguridad (**foundation security blueprints**). Estos principios son una de las guías que proporciona Google Cloud para ayudar al usuario en la construcción de su despliegue en la nube y cubre los siguientes aspectos:
  - **Estructura de una organización en Google Cloud**. Cada organización que contrata servicios de Google Cloud existe como un ente denominado **organización (organization)**, que gestiona todos los recursos y políticas desde un único punto. Cada organización consta de una jerarquía de carpetas o apartados (**folders**), cada uno de los cuales engloba diferentes **proyectos** (la unidad de trabajo en GCP) bajo un concepto común, y donde cada proyecto se define como un conjunto de **recursos**. Esta jerarquía está compuesta por los siguientes *folders*: *bootstrap*, *common*, *production*, *non-production* y *development*. La función de los *folders* es unificar una serie de políticas, es decir, establecer ciertas constantes para la configuración de recursos y aplicarlas consistentemente en todos los proyectos dentro de dicho *folder*. La figura 3 muestra un ejemplo de organización, donde se pueden ver diferentes proyectos englobados en sus *folders* correspondientes, cada uno de los cuales contiene un conjunto de recursos diferentes (redes, claves, aplicación...).
  - **Jerarquía de recursos y despliegue**. Los recursos se despliegan mediante un *pipeline* que permite la automatización y ofrece la posibilidad de estandarizar la revisión, aprobación y vuelta atrás de cada despliegue.
  - **Autenticación y autorización**. Google Cloud proporciona servicios de gestión de

identidad, como Google Cloud Identity, para administrar la autenticación y el acceso de usuarios; y mecanismos de federación de identidad mediante Active Directory y Single Sign-On. También dispone de Cloud IAM, que permite definir grupos con diferentes roles, permisos de acceso y uso de los recursos.

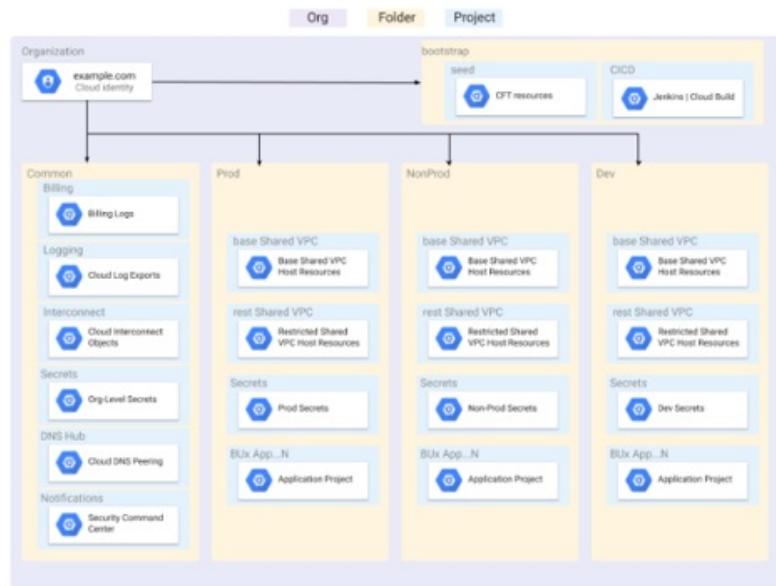


Figura 3. Jerarquía de *folders* de una organización. Fuente: <https://services.google.com/fh/files/misc/google-cloud-security-foundations-guide.pdf>

- ▶ • **Redes(segmentación y seguridad).** Google Cloud dispone de conexiones dedicadas para interconectar nubes híbridas con recursos *on-premises* y en su nube pública. Cuenta también con la herramienta Shared Virtual Private Cloud (VPC) para alojar cada proyecto y facilitar así la gestión de los recursos de red que este requiere. Asimismo, con el objetivo de proteger los recursos conectados en la red y aislarlos de otros proyectos, proporciona, entre otros recursos, *firewalls*, Cloud DNS o VPC Service Controls, este último para ofrecer un perímetro de protección para servicios que almacenan información muy sensible.
- **Gestión de secretos.** Existe un servicio Google Secret Manager para almacenar diferentes claves, API keys, contraseñas, certificados y otros datos sensibles, cuyo

acceso está gestionado a través de Cloud IAM.

- **Logging.** Google Cloud dispone de Google Cloud Logging, que permite agregar todos los *logs* generados en los servicios contratados por la organización.
  - **Controles de investigación (*detective controls*).** Google Cloud utiliza la telemetría de la propia plataforma para detectar configuraciones no recomendables, vulnerabilidades y comportamientos sospechosos. Para ellos dispone de servicios como Security Command Center, Chronicle, Cloud Asset Inventory o Cloud Logging.
  - **Configuración del consumo.** Existen mecanismos para la monitorización del uso de los recursos contratados en Google Cloud y de generación de alertas para evitar consumos inesperados.
- **Guía de seguridad en trabajos y aplicaciones específicos(*workload and applications blueprints*).** Además de los principios básicos anteriores, Google Cloud proporciona recomendaciones adicionales, de carácter más específico, para diseñar, construir y operar sobre los trabajos y aplicaciones ejecutados en la nube (por ejemplo, ofrece el Cloud Healthcare Data Protection Toolkit, más específico del campo sanitario).

Una vez que conocemos los conceptos e ideas transversales, pasamos a explorar algunos de los servicios que ofrece este proveedor. Al igual que ocurre en todos los proveedores de *cloud computing*, existen una serie de bloques fundamentales sobre los que se construyen los servicios *big data* ofrecidos por Google Cloud. Estos bloques fundamentales son: seguridad, computación, almacenamiento y redes. Además de estos servicios básicos, GCP proporciona múltiples otros, entre ellos un conjunto centrado en *big data* y *machine learning* que eliminan parte de la complejidad de gestionar la infraestructura y los recursos que hay por debajo. Vamos a explorar los principales en las siguientes secciones.

## 10.5. Servicios de computación

Al igual que en otras plataformas, uno de los servicios básicos principales es el de computación. Podemos encontrar opciones gestionadas por Google (pertenecientes al modelo PaaS o SaaS, es decir, *serverless*, sin necesidad de preocuparse por la gestión de la infraestructura subyacente) con **AppEngine** u orientadas a eventos y procesados puntuales con **Cloud Functions**. También otras en las que podemos elegir el *hardware* que estará por debajo y gestionar el servicio hasta el nivel del sistema operativo con **Compute Engine** (IaaS). Por último, existen soluciones intermedias, en las que Google permite configurar un clúster de máquinas, ejecutando un clúster Kubernetes con **Kubernetes Engine (GKE)** o contenedores Docker con **Cloud Run**, y alojando diferentes aplicaciones o sistemas dentro de contenedores.

Con estas opciones, podemos hacer frente a distintos casos de uso, como alojar y servir una página web (desde páginas web estáticas hasta opciones más sofisticadas que necesiten平衡adores de carga y escalado, aplicaciones web con muchas dependencias u opciones más gestionadas como Heroku o Engine Yard), desplegar un clúster Hadoop para aplicaciones *big data* o servir un modelo de *machine learning* desarrollado con TensorFlow.

Vamos a ver con más detalle las dos opciones que más nos interesan por su capacidad para poder desplegar clústeres de tecnologías *big data* en ellas: Google Compute Engine (y servicios relacionados) y servicios de contenedores.

### [Google Compute Engine \(GCE\)](#)

**Google Compute Engine (GCE)** es el servicio que proporciona servidores virtuales en los que el usuario debe elegir el *hardware* que habrá por debajo (tipo de máquina, tamaño de disco y de memoria, capacidad de la red...) y qué sistema operativo quiere que tenga instalado. Por tanto, la configuración, administración y gestión

*software* queda en manos del usuario, pero se elimina la necesidad de comprar las máquinas o instalar el sistema operativo y el resto de herramientas básicas, cuya gestión es responsabilidad de GCP.

Para crear una instancia de GCE, llamadas **instancia de máquina virtual (VM)**, es necesario elegir el tipo de máquina. Esto determina, entre otros aspectos:

- ▶ La memoria.
- ▶ La CPU (que puede ser *standard*, *high-memory*, *high CPU*, *shared-core* para tareas poco intensivas, GPU...).
- ▶ El tipo y tamaño del disco para almacenamiento persistente (cuyo tipo puede ser estándar, SSD, local SSD o *cloud storage*, cada cual con sus características de capacidad, velocidad, redundancia y precio).
- ▶ La zona donde se desplegará la instancia.
- ▶ Opciones de *firewall*.

Aunque se puede especificar la configuración concreta deseada, es posible elegir también **configuraciones preestablecidas para diferentes propósitos**: general (máquinas E2, N1, N2, N2D), optimizada para cómputo intensivo (máquinas C2), optimizada para altos requisitos de memoria (máquinas M2) u optimizada con *hardware* de aceleración (máquinas A2). Durante la configuración de las opciones de la instancia, GCP muestra el precio estimado de esta. Además del almacenamiento elegido, todas las instancias tienen un pequeño disco persistente, para tareas básicas de la máquina como el arranque.

Existen distintos tipos de imágenes que se pueden instalar en la instancia, es decir, diferentes sistemas operativos con ciertas herramientas preinstaladas. El usuario tiene la posibilidad de usar imágenes públicas (preconstruidas) o privadas, las cuales construye y proporciona por sí mismo.

Cada VM pertenece a una única red privada virtual (VPC), donde todas las instancias de una misma VPC se comunican mediante red cableada (más rápido), mientras que las instancias de diferentes VPC se comunican a través de Internet (más lento). Por tanto, si necesitamos tener varias instancias interconectadas para interaccionar entre ellas, es conveniente que estén en la misma VPC, a fin de aumentar la velocidad del tránsito de los datos de unas a otras.

## Alternativas a las *virtual machines*: servicios de contenedores

GCP proporcionan varias alternativas específicas a las máquinas virtuales de *cloud computing*:

- ▶ **Preemptible instances.** Son instancias con un coste reducido, pero que pueden finalizar y desaparecer en cualquier momento (con 30 segundos de preaviso, para permitir limpiar/guardar cualquier recurso que se quiera mantener), en caso de que GCP necesite los recursos para otros fines. Por tanto, es una opción interesante para aplicaciones tolerantes a fallos si tenemos en cuenta que estas instancias tienen una duración máxima de 24 horas, que la probabilidad de terminación depende del día y la zona, y que no se ejecutan durante actualizaciones, reinicios o tareas de mantenimiento, sino que, en estos casos, finalizan automáticamente.
- ▶ **Sole-tenant nodes.** Este recurso proporciona servidores de GCE dedicados solo a los trabajos del usuario que la contrata. Ofrece las mismas opciones que las VM de GCE en términos de capacidad de cómputo, memoria, sistemas operativos, etc., pero en servidores dedicados exclusivamente al usuario. Es ideal para requisitos estrictos de seguridad o cuando es necesario tener un *hardware* dedicado específico, es decir, sobre el que ejecutar *software* con licencia ligada a un único *hardware*.
- ▶ **Contenedores.** Si la aplicación que se quiere ejecutar en la nube tiene diferentes componentes (servidor web, base de datos, etc.), cada uno con gran cantidad de dependencias, puede ser más sencillo separar todos estos componentes en distintos contenedores. Esto es lo que se llama una arquitectura orientada a servicios o

microservicios, y, para su implementación, se usan tecnologías de contenedores como Docker o Kubernetes.

Para ejecutar estos contenedores, se podrían usar instancias VM de GCE instalando Docker o Kubernetes, y haciendo el despliegue de forma manual, gestionada por el usuario. Sin embargo, GCP proporciona dos servicios específicos, llamados **Google Kubernetes Engine (GKE)** y **Google Run**, que eliminan parte de la complejidad de instalar y gestionar un clúster Kubernetes o contenedores Docker, respectivamente. Estas herramientas dejan únicamente en manos del desarrollador el despliegue de los contenedores que forman su aplicación y ocultan todos los detalles por debajo. La principal característica de Kubernetes es que permite crear un clúster con nodos sobre los que se instalen diversos contenedores.

Las **principales ventajas de usar contenedores** es lo que se conoce como *componentización* o microservicios, y consisten en:

- ▶ Aislamiento de capa de aplicación, datos y procesado de otras capas.
- ▶ Portabilidad, dado que el contenedor está autocontenido y, por tanto, es fácil llevarlo a otro sistema y desplegarlo allí.
- ▶ Velocidad de despliegue, pues los contenedores son ligeros (solo virtualizan el sistema operativo, no el *hardware*) y, en consecuencia, son rápidos de desplegar.
- ▶ Orquestación, que permite crear clústeres de forma sencilla.
- ▶ Registro de imágenes, de forma que se pueda tomar una imagen preconstruida. GCP ofrece su propio servicio de registro de contenedores con **Google Container Registry**.
- ▶ Flexibilidad, ya que se pueden mezclar fácilmente contenedores con otros sistemas *on-premises*.

Los contenedores tienen las mismas opciones de almacenamiento que las instancias

VM de GCE, con la diferencia de que sus discos son efímeros: cuando un contenedor se termina, todos los datos guardados en el disco se eliminan y, al reiniciar, se empieza desde cero. Si no queremos este comportamiento, podemos usar *gcePersistentDisk* para crear discos persistentes que mantengan los datos de los contenedores.

**Google Cloud Run** va un paso más allá y ofrece contenedores Docker, pero abstrayendo por completo toda la gestión de la infraestructura y permitiendo escalar automáticamente según lo requieran las necesidades de la aplicación en cada momento.

## 10.6. Servicios de red

Google Cloud dispone de un servicio denominado **Virtual Private Cloud (VPC)**, que representa una versión virtual de una red física de interconexión, en este caso, de servicios *cloud*. Esta VPC proporciona interconexión entre instancias VM de GCE, clúster GKE, instancias de AppEngine, así como otros componentes de GCP. Dispone asimismo de mecanismos de balanceo de carga de tráfico y túneles para conectar redes *on-premises* con la red en la nube, y es capaz de distribuir tráfico desde平衡adores de carga externos hacia el *backend*. Para grandes organizaciones, también es posible definir una red **shared VPC**, que permite que múltiples proyectos puedan usar una única VPC compartida por todos ellos.

Una red VPC consiste en una o más particiones de direcciones IP, llamadas **subredes**. Cada subred está asociada a una región y debe existir, al menos, una dentro de una red VPC para que esta pueda empezar a usarse. Por defecto, cuando se crea un proyecto, también **se instaura una red VPC (auto mode VPC)**, que, a su vez, establece una subred por cada región. No obstante, un proyecto puede contener varias redes VPC. Las redes VPC son recursos globales que no tienen asociado ningún rango de direcciones IP, mientras que las subredes que las componen son recursos regionales, cada uno de los cuales cuenta con un rango de direcciones IP asignado. Al crear un recurso (ya sea de cómputo, como los vistos en la sección previa, o de cualquier otro tipo), es necesario seleccionar una subred (más su red asociada) y una zona (más su región asociada, que determinará las subredes que se pueden elegir).

Una de las principales ventajas de las redes VPC y las subredes es que permiten definir reglas mediante un **firewall**, para controlar el tráfico que puede entrar y salir de los recursos conectados a cada subred. Así, los recursos conectados a una VPC se pueden comunicar con los de otra VPC, siempre que las reglas de *firewall* lo

permitan, mediante tráfico IPv4 (las redes VPC no permiten tráfico *broadcast*, *multicast* ni IPv6 dentro de la red). Por defecto, dichas reglas admiten casi todo el tráfico saliente de las instancias conectadas a una VPC, y bloquean casi todo el entrante. Esta configuración inicial es modificable dependiendo de las redes en juego y del tráfico que se quiera permitir en la red.

Podemos ver un ejemplo de red VPC y subredes en un proyecto de GCP en la figura 4. Como se observa, dentro de un proyecto, existe una red VPC con tres subredes, una de ellas en una región y otras dos en otra región diferente. La subred 1 contiene dos instancias VM en una única zona. La subred 2 contiene otras dos instancias VM, también en una misma zona, aunque diferente de la subred 1. Por último, la subred 3 contiene una instancia VM en una zona y otra instancia VM en otra zona diferente. Esto permite, lógicamente, aislar diferentes recursos, así como incrementar la disponibilidad y la eficiencia mediante la réplica de estos en diferentes zonas, pero también a través del uso de balanceadores de carga que gestionen las peticiones entrantes a los recursos (subredes) que estén disponibles en cada momento.

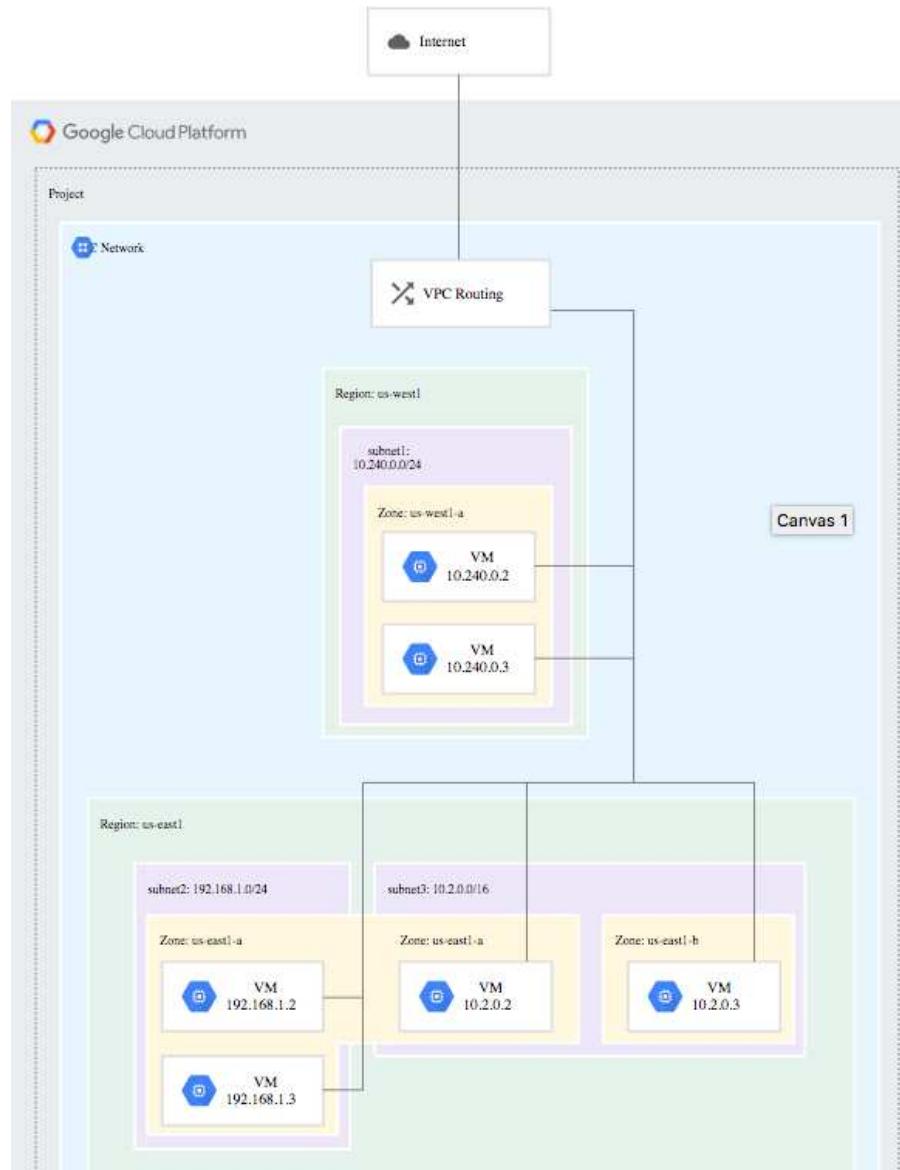


Figura 4. Ejemplo de VPC dentro de un proyecto GCP. Fuente: <https://cloud.google.com/vpc/docs/vpc>

## 10.7. Servicios de almacenamiento

Existen diversos tipos de almacenamiento en GCP, adecuados a distintas necesidades: desde almacenamiento en bloque para instancias VM hasta almacenamiento en la nube para objetos inmutables como imágenes y vídeos, pasando por diversos tipos de bases de datos que veremos en la siguiente sección. Además, GCP proporciona un abanico de servicios de transferencia de datos desde diferentes fuentes a GCP. A continuación, detallarán las tres opciones principales de almacenamiento:

- ▶ **Persistent Disk** es un servicio orientado a bloques, cuya principal utilidad es la de servir de almacenamiento persistente para instancias VM y para contenedores, para lo que se necesita que el disco persistente y la instancia de cómputo estén en la misma zona. Proporciona opciones de replicación y disponibilidad multirregión; aumento de capacidad de almacenamiento durante el uso; encriptación de datos, tanto en tránsito como en reposo, y una separación de los recursos de cómputo respecto de los de almacenamiento, lo que aporta mucha flexibilidad al sistema final. Además, a diferencia de otros proveedores, en el caso de GCP, varias instancias VM pueden leer datos de un mismo recurso Persistent Disk. Entre las opciones de almacenamiento en bloques que proporciona GCP, encontramos:
  - Discos persistentes HDD, ideales para acceso secuencial y con un coste menor.
  - Discos persistentes SSD, con gran eficiencia en el acceso aleatorio.
  - Discos SSD locales, es decir, discos físicos reales unidos a una instancia VM en particular.
- ▶ **Cloud Storage** está orientado al almacenamiento ilimitado de objetos que pueden ser accedidos a nivel global. Ofrece una durabilidad del 99,99999999 % y baja latencia. El almacenamiento está organizado en *buckets*, cada uno de los cuales tiene un nombre único en todo GCP y una localización asociada. La elección de la

zona donde estará desplegado el servicio de almacenamiento dependerá de los requisitos de latencia, de la tolerancia a fallos mediante replicación multizona o multirregión, o del cumplimiento de normativas de datos. También dispone de distintos mecanismos de seguridad, como permisos de acceso mediante Cloud IAM, logs de acceso y gestión, y diferentes opciones de encriptación de los datos.

Dependiendo de la frecuencia de acceso y de la duración del almacenamiento, Cloud Storage ofrece diferentes opciones:

- ▶ • **Standard multi-regional**, para acceso frecuente desde cualquier parte del mundo. Ofrece con 99,95 % de disponibilidad y redundancia en, al menos, dos regiones separadas por un mínimo de 100 millas (160 km). Es la opción más cara.
- **Standard regional**, para acceso frecuente desde una única zona del mundo, con 99,9 % de disponibilidad. Está indicado para realizar analíticas de datos o para datos usados en instancias VM.
- **Nearline**, para acceso, como mucho, una vez al mes, con una disponibilidad del 99,0 %. Se compromete a que los datos estarán almacenados, al menos, durante treinta días. El coste mayor está asociado a la consulta de los datos.
- **Coldline**, para acceso de una vez al año como máximo. Ofrece también 99,0 % de disponibilidad y misma velocidad de acceso que en otros tipos, pero con alto coste de obtención de datos y un compromiso de mantenimiento de datos almacenados de, al menos, noventa días. Está pensado para *backups* (recuperación ante desastres o mantenimiento de datos archivados que no se sabe si se volverán a usar o no).
- **Archive**, para acceso muy ocasional. Requiere un compromiso de mantenimiento de datos almacenados durante, al menos, 365 días. Está orientado, por ejemplo, al archivo de datos regulados.

La siguiente tabla muestra un resumen de las principales características de estos

tipos de almacenamiento. Además, GCP dispone de un servicio llamado **Object Lifecycle Management (OLM)**, que permite configurar transiciones de datos de opciones más a menos costosas, según la frecuencia de acceso disminuye con el tiempo.

Tipos de almacenamiento Cloud Storage					
	Standard multi-regional	Standard regional	Nearline	Coldline	Archive
Durabilidad	99,99999999 %				
Disponibilidad	99,9 %	99,95 %	99,0 %	99,0 %	NA
Tolerancia a fallos	>=2	1	>=2	>=2	>=2
Uso	Uso frecuente (aplicaciones web, móvil...)	Datos almacenados, al menos, 30 días	Datos almacenados, al menos, 90 días	Datos almacenados, al menos, 365 días	

Tabla 1. Características de los tipos de almacenamiento que ofrece Cloud Storage.

Veremos más adelante que, en GCP, HDFS es sustituido por Cloud Storage para evitar tener un *namenode* ejecutándose continuamente en una instancia VM, con el consecuente coste prohibitivo que esto tendría.

- ▶ **Filestore** es un sistema de almacenamiento de ficheros en la nube, equivalente a un NAS (Network Attached Storage), para instancias VM y GKE. Permite escalar hasta cientos de TB. Se trata de un servicio totalmente gestionado por Google Cloud, que se encarga de las operaciones necesarias a fin de desarrollar un sistema de almacenamiento de archivos conectado en red, para la compartición de datos entre distintas instancias VM y GKE.

## 10.8. Bases de datos

Yendo un paso más allá en la abstracción en el campo del almacenamiento de ficheros, vamos a analizar las opciones de bases de datos que proporciona GCP. Recordemos que tenemos dos grandes grupos de bases de datos, relacionales y NoSQL; también merece la pena tener en cuenta las bases de datos en memoria (Redis, cachés y similares). Podemos usar instancias VM o GKE para instalar y desplegar la base de datos que nos interese; en este caso, su gestión (instalación, actualización, escalabilidad, etc.) queda en nuestras manos. No obstante, GCP ofrece un abanico de bases de datos gestionadas por Google Cloud, que ahorran al usuario todo este trabajo y lo dejan a cargo solo del almacenamiento y la administración de los datos en sí. En este apartado, veremos las opciones principales de bases de datos de los grupos mencionados, en su opción totalmente gestionada por Google Cloud, así como sus características y casos de uso más habituales.

- ▶ **Cloud SQL** es un sistema totalmente gestionado por Google Cloud, que permite desplegar bases de datos relacionales **PostgreSQL, MySQL y SQL Server** en la nube. Es un recurso zonal, que puede replicarse (con alta consistencia entre réplicas) para aumentar la disponibilidad. Está orientado a OLTP, es decir, a consultas transaccionales en **batch**. No es, sin embargo, adecuado para OLAP, por el gran número de comprobaciones de consistencia que realiza en cada consulta. La característica más interesante de esta opción es que ofrece **escalado horizontal**, es decir, proporciona mayor capacidad de almacenamiento, más instancias y/o mayor replicación según sea necesario.

Un servicio relacionado es **Transfer Service**, que ayuda a transferir datos desde diversas fuentes hasta Cloud Storage. Las fuentes desde donde se pueden transferir datos son AWS (por ejemplo, un *bucket* de S3), sitios HTTP/HTTPS, ficheros locales u otro *bucket* de Cloud Storage.

- ▶ **Cloud Spanner** es un servicio similar a Cloud SQL, pero con un **motor propietario de Google**. Funciona como recurso regional o global y ofrece un 99,999 % de disponibilidad. Está orientado asimismo a consultas *batch* (OLTP) y proporciona alta consistencia entre réplicas, además de escalado horizontal según crezcan las necesidades del sistema.
- ▶ En el apartado de bases de datos NoSQL, una de las principales opciones es **Bigtable**, indicada para casos de uso que requieran lecturas de baja latencia y gran capacidad de escritura y escalabilidad, como, por ejemplo, cargas analíticas muy altas o aplicaciones de baja latencia. Es también un recurso regional o global, con una latencia del orden de unos pocos milisegundos. Bigtable es una base de datos orientada a columnas, indicada para hacer escaneos rápidos de claves secuenciales. En este sentido, es necesario diseñar cuidadosamente la estructura de las claves, ya que aquellas similares se almacenan en posiciones adyacentes. De este modo, si tenemos una serie de datos a los que queremos acceder en conjunto de forma rápida, habremos de asignarles las claves de tal manera que esa información se almacene de forma contigua.

BigTable es bastante parecido a Hbase: ambas bases de datos se orientan a columnas y están muy indicadas para datos dispersos (*sparse*), ya que no malgastan espacio cuando no se tienen todos los datos de cierto registro. De forma muy general, BigTable es básicamente una versión gestionada por GCP de HBase, pero con algunas particularidades. Proporciona escalabilidad horizontal, sin que la disponibilidad decaiga y sin necesidad de gestionar la configuración y el mantenimiento de la base de datos. Asimismo, permite tener muchas familias de columnas (hasta cien) antes de que el nivel de funcionamiento mengüe. Soporta almacenamiento denormalizado, permite únicamente operaciones CRUD y solo mantiene ACID a nivel de fila.

- ▶ **Cloud Firestore** es una de las bases de datos NoSQL orientada a documentos con cierto esquema (XML o HTML). Sus principales características son el soporte de

transacciones ACID y una garantía de disponibilidad del 99,999 % gracias a la replicación multirregión. Generalmente, este tipo de base de datos no está pensada para consultas interactivas (OLAP) ni transaccionales (OLTP), sino para búsquedas rápidas de claves y de sus correspondientes valores. Su principal ventaja es que la velocidad de respuesta de los resultados depende del tamaño de estos y no del de la base de datos. Es decir, la consulta será más lenta si devuelve 10 filas que si devuelve 1 fila, pero será indiferente a si la base de datos tiene 10 entradas o 10 millones de entradas. La clave para conseguir este comportamiento es que los índices son rápidos de leer, pero lentos para escribir. Por tanto, esta opción no está indicada para escritura intensiva de datos.

Cloud Firestore está, en cierto modo, más cerca de las bases de datos relacionales que otras bases de datos NoSQL, porque ambas soportan transacciones atómicas y usan índices para realizar búsquedas rápidas. No están indicadas para datos no estructurados (en cuyo caso, BigTable es mejor); ni para analíticas/BI (OLAP); ni para guardar grandes objetos, mayores de 10 MB (para eso está Cloud Storage); ni para aplicaciones que necesiten muchas escrituras o actualizaciones en columnas importantes. Por el contrario, conviene su uso cuando se necesita escalado para mejorar el funcionamiento de lectura y para documentos jerárquicos que se pueden almacenar como clave/valor. La razón por la que es tan escalable es que está completamente indexada, pero eso también implica que las actualizaciones (*updates*) son muy lentas, las operaciones *join* no son posibles y no permite filtrar resultados basados en una subconsulta ni incluir más de una condición en los filtros con desigualdades.

- ▶ **Cloud Memorystore** es la opción perteneciente al grupo de las bases de datos en memoria. Es un servicio zonal, totalmente gestionado por Google Cloud, que proporciona una base de datos Redis o Memcached escalable y con alta disponibilidad (99,9 %). Se caracteriza por la velocidad de respuesta, por debajo del milisegundo.

## 10.9. Servicios de big data y analítica

Como se mencionaba previamente, GCP ofrece un amplio abanico de servicios gestionados más allá de la infraestructura y las bases de datos. Nos centraremos en aquellos relacionados con las tecnologías de *big data* y analítica, para acabar en el siguiente apartado con los servicios de *machine learning*.

Al igual que AWS, GCP ofrece varios servicios relacionados con *big data* y analíticas, desde desplegar clústeres Hadoop gestionados por GCP hasta herramientas propietarias de GCP con funcionalidades muy similares a las de las tecnologías del ecosistema Hadoop que se han visto en los temas previos. Todo ello sin perder de vista que siempre es posible instalar dichas tecnologías en instancias VM de Google Cloud Engine o en contenedores GKE.

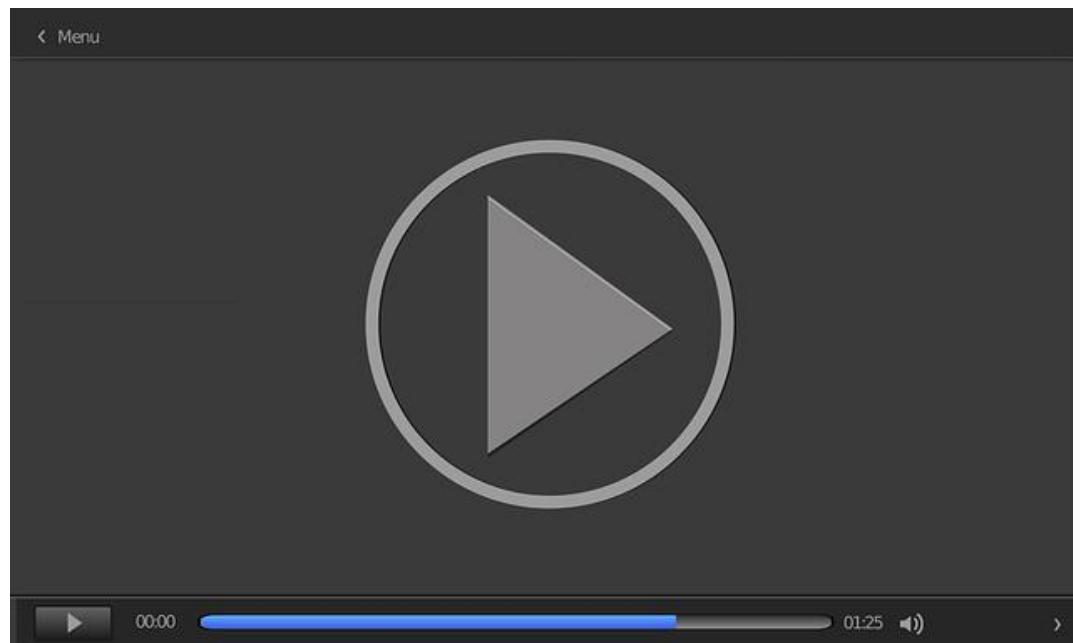
- ▶ **Dataproc** ofrece la posibilidad de desplegar **clústeres Hadoop totalmente gestionados por GCP**, que incluyen varias tecnologías como las propias de Hadoop (HDFS, MapReduce y YARN), **Zookeeper**, **Hive**, **Pig** y **Spark**. Aunque Dataproc incluye HDFS, en la práctica, se usa **Google Cloud Storage** para evitar tener un *namenode* activo continuamente en una instancia VM, cuyo coste sería prohibitivo. El hecho de estar totalmente gestionado por GCP permite crear un clúster, usarlo y terminarlo sin tener que gestionar nada, lo cual es ideal para mover desarrollos que se ejecutaban en Hadoop directamente a Google Dataproc. Además, se puede desplegar un clúster, escalarlo o terminarlo en 90 segundos o menos.

Para crear el clúster, se pueden elegir diferentes tipos de máquina basadas en instancias VM. Un clúster necesita, al menos, un nodo máster y dos nodos *worker*. Los clústeres de Dataproc pueden contener instancias **preemptible**, ya que son buenas opciones para ejecutar cargas de trabajo puntuales y permiten reducir costes, pero siempre hay que tener en cuenta que no deben ser usadas para almacenamiento, solo para procesados. Como cabe esperar, una de las principales

ventajas de Dataproc es el autoescalado, incluso mientras se están ejecutando tareas, lo cual puede lograrse añadiendo o eliminando nodos *worker*, o añadiendo almacenamiento.

Además de las tecnologías Hadoop, Dataproc ofrece una fácil integración con otras herramientas de GCP como BigQuery, Cloud Storage o BigTable, lo que hace de la combinación de todas ellas una completa plataforma de datos.

En el siguiente vídeo, profundizaremos sobre la utilización de la herramienta Dataproc y el espacio de almacenamiento que nos ofrece.



---

Accede al vídeo:

<https://unir.cloud.panopto.eu/Panopto/Pages/Embed.aspx?id=cc848a08-769f-4753-96c0-ac6d010a1c04>

---

Vídeo. Introducción a Dataproc.

- **Dataflow** es un **sistema ETL** (similar a lo que podría hacer Spark o Flink), similar a

Spark semánticamente. Las últimas versiones están basadas en **Apache Beam**, un *framework* para procesar grandes cantidades de datos (tanto en modelo *batch* como en *streaming*), que permite desarrollar *pipelines* de transformaciones. Así, mediante este *software*, Dataflow es capaz de fragmentar y procesar en paralelo conjuntos de datos por lotes de gran tamaño o transmisiones de datos en vivo de gran volumen. El resultado acaba siendo un grafo dirigido acíclico (*directed acyclic graph*, DAG).

Esta idea de representar las transformaciones como un DAG se usa en muchas tecnologías (Flink, Apache Beam, TensorFlow, Spark), porque permite ordenar topológicamente los pasos (es decir, realizar las transformaciones de forma óptima) y se presta a la parallelización. Un *pipeline* en Dataflow es una tarea única, potencialmente repetible de inicio a fin, que encapsula una serie de procesados que aceptan datos de fuentes externas, los transforman para proporcionar algún tipo de resultado y producen una salida. Dicho *pipeline* está definido en el programa *driver*, mientras que las computaciones como tal se ejecutan en un *backend* de nodos *worker*. La fuente y el sumidero de datos pueden ser muy diversos, como objetos en Cloud Storage o tablas de BigQuery.

- ▶ **BigQuery** es un sistema de **warehousing** que permite tener una interfaz SQL para realizar consultas sobre grandes conjuntos de datos almacenados en ficheros o tablas (*data warehouse*); es decir, no es una base de datos relacional ni NoSQL. Por tanto, semánticamente, es similar a Hive, pero BigQuery es mucho más rápido, ya que depende de un sistema propietario para realizar las consultas. Por otro lado, la latencia es un poco mayor que en BigTable y Firestore, por lo que es preferible elegir estos para requisitos de baja latencia. No tiene propiedades ACID y, por tanto, no se puede utilizar para consultas transaccionales (OLTP), pero sí es apto para **consultas interactivas** como analíticas de datos, BI o *data warehouse (OLAP)*.

Mientras que las bases de datos relacionales se sustentan generalmente en el *schema-on-write* (es decir, se comprueba si el esquema definido coincide con los datos cuando estos se escriben en la base de datos), Hive y BigQuery lo hacen en el

**schema-on-read**, porque comprueban si esquema y datos se alinean al leer dichos datos en una consulta (hay que tener en cuenta que estos dos sistemas no los escriben, sino que hacen consultas sobre datos ya almacenados en ficheros en almacenamiento persistente). BigQuery es, además, capaz de **inferir el esquema** de los datos en algunos casos, lo cual es clave. Las bases de datos clásicas son las propietarias de los datos que hay guardados en ellas; sin embargo, en el caso de Hive y BigQuery, los datos pueden pertenecer a estos sistemas o estar gestionados por sistemas externos, de manera que, en ciertos casos, pueden no saber qué esquema tienen o si este ha cambiado. Por tanto, resulta necesario que puedan inferirlo.

El modelo de datos está compuesto por lo que se denomina **datasets**. Un *dataset*, que solo puede pertenecer a un único proyecto de Google Cloud, es un conjunto de tablas y vistas. Las tablas contienen registros (*records*) con filas y columnas (*fields*). Existen columnas anidadas y repetidas. El esquema se puede especificar en el momento de la creación o cuando se cargan los datos inicialmente, así como actualizarse tras crear la tabla. Existen dos tipos de tablas: nativas (almacenadas en BigQuery) y externas (no gestionadas por BigQuery, sino por alternativas como BigTable, Cloud Storage o Google Drive).

BigQuery ofrece la opción de inferir el esquema mientras se cargan datos o se hacen consultas de datos externos. Lo que hace es seleccionar un fichero aleatorio en la fuente de datos y escanear hasta cien filas como ejemplo representativo del esquema. Luego examina los *fields* y trata de asignar un tipo de datos a cada uno en función de los valores que encuentra en la muestra de filas escaneadas. Puede cargar datos en bloque (CSV, JSON, Avro, GCP Datastore Buckups) o por flujo (para *logs* o *dashboards* en tiempo real). Otras opciones de carga de datos son *datasets* públicos (una alternativa muy interesante por la gran cantidad de *datasets* de datos disponibles de lo más diverso), *datasets* compartidos o ficheros de *log*.

- ▶ **Cloud Pub/Sub** es un sistema de transporte de mensajes disponible en GCP. Es un

*middleware* de mensajes, capaz de gestionar (recibir/enviar) mensajes de muchos a muchos de forma asíncrona y desacoplando escritores (publicadores) de leedores (suscriptores). Al igual que en Apache Kafka, existen aplicaciones que publican mensajes en un **topic** y otras que se suscriben y reciben los datos publicados en él. La suscripción es una **cola** (un flujo de mensajes) desde el sistema publicador al suscriptor. La figura 5 muestra el funcionamiento de las dichas colas.

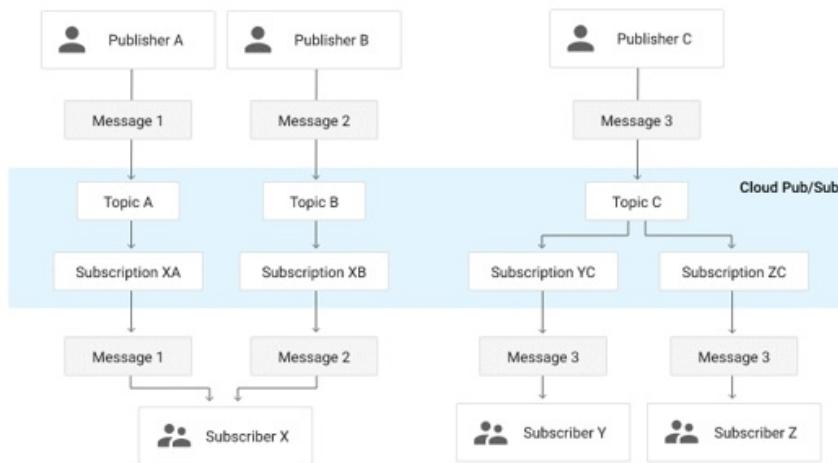


Figura 5. Funcionamiento de Google Pub/Sub como sistema de colas. Fuente:

<https://cloud.google.com/pubsub/docs/overview>

Un **mensaje** es una combinación de datos y atributos enviados por un publicador a un *topic*. Los atributos son pares clave-valor enviados junto con los datos. Los mensajes son persistentes hasta que se entregan y/o se notifica su recepción. Existe una cola por suscripción (como se ve en la figura 5) y el mensaje se queda en esa cola hasta que es consumido. Los suscriptores, por su parte, pueden usar mecanismo *pull* (mediante peticiones HTTPS) o *push* (mediante webhook).

Cloud Pub/Sub es capaz de integrarse con el resto de tecnologías GCP y de transportar mensajes desde/hacia diferentes servicios de esta plataforma, tal y como se muestra en la figura 6.

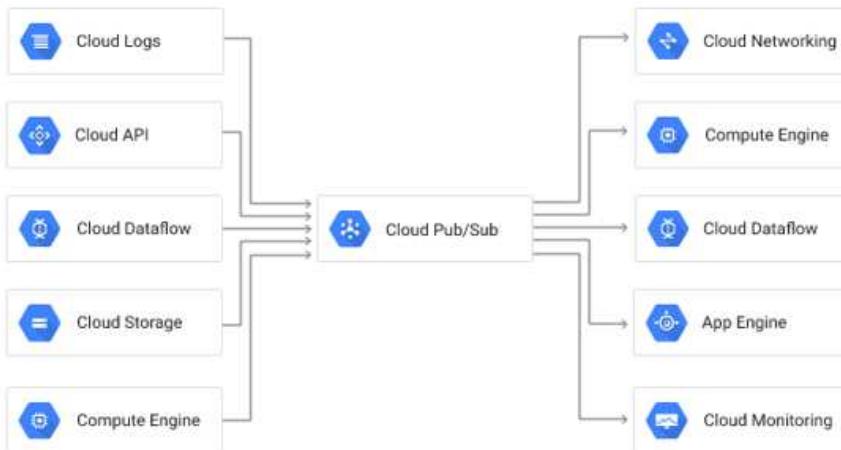


Figura 6. Integración de tecnologías GCP con Pub/Sub.

Fuente: <https://cloud.google.com/pubsub/docs/overview>

Algunos casos de uso pueden ser: balanceo de cargas de computación en clústeres, procesado asíncrono de órdenes, distribución de notificaciones de eventos, refresco de cachés distribuidas, *logging* en múltiples sistemas de forma simultánea o *streaming* de datos entre diferentes sistemas.

- ▶ **Cloud Datalab** es un servicio de GCP que proporciona Jupyter *notebooks*, los cuales permiten explorar, visualizar, analizar y transformar datos. Estos *notebooks* se ejecutan como contenedores y en una instancia VM. Datalab facilita el proceso de creación de dicha instancia, la ejecución del contenedor correspondiente y la conexión entre el navegador web desde donde se maneja el *notebook* al contenedor. Además, también facilita la integración y el acceso a otros servicios Google Cloud como BigQuery, Google Machine Learning Engine, Dataflow o Cloud Storage.
- ▶ **Cloud Dataprep** es un servicio gestionado por una empresa externa, Trifacta, para explorar y transformar grandes cantidades de información en bruto, de muy diverso tipo, en conjuntos de datos limpios y estructurados, que puedan servir para su posterior análisis y procesado, tal y como se muestra en la figura 7. Dataprep puede leer desde diversas fuentes de datos, tanto procedentes de GCP (como BigQuery y Cloud Storage) como externas; realizar las transformaciones necesarias para limpiar

y homogeneizar los datos, y pasarlos posteriormente a fases de almacenamiento y análisis.



Figura 7. Arquitectura de un caso de uso de Cloud Dataprep. Fuente: <https://cloud.google.com/dataprep>

Todo el abanico de servicios descritos previamente proporciona los medios para construir diferentes casos de uso, como los que se muestran en la figura 8: procesado de tareas *batch*, exploración y minería de datos, procesado y almacenamiento de flujos de datos, entrenamiento de modelos *machine learning*, uso de *machine learning* para producir predicciones o *data warehousing*.

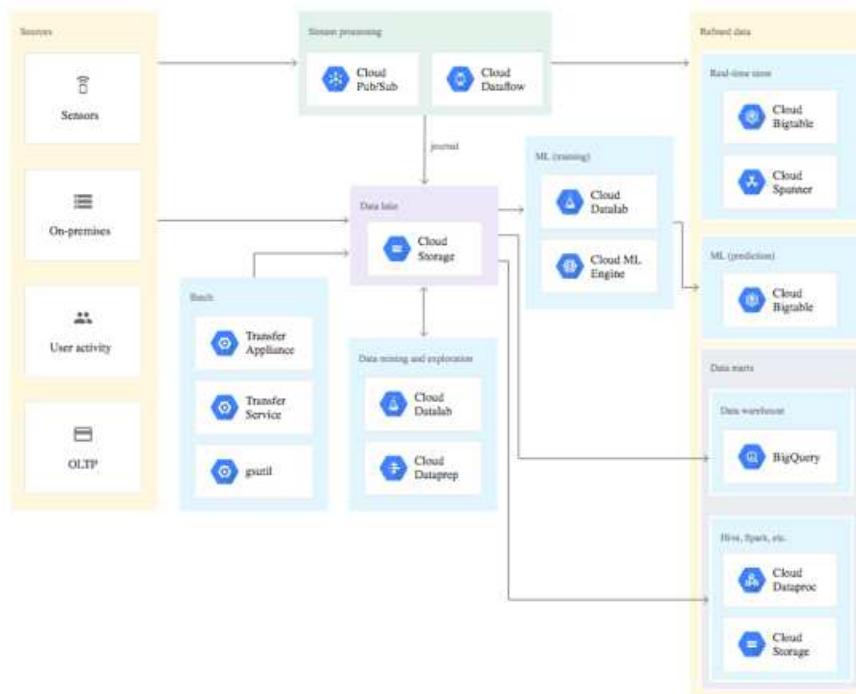


Figura 8. Diferentes casos de uso relacionados con analíticas y *big data* de los servicios Google Cloud. Fuente: <https://cloud.google.com/solutions/build-a-data-lake-on-gcp>

## 10.10. Machine learning e inteligencia artificial

Los servicios de *machine learning* e inteligencia artificial son uno de los diferenciadores de Google Cloud, ya que TensorFlow, desarrollado por Google, es el claro ganador cuando hablamos actualmente de desarrollar soluciones de *machine learning* y, sobre todo, de *deep learning*, lo que da cierta ventaja competitiva al integrarlo con el resto de servicios *cloud*.

Al igual que ocurría con Azure o AWS, Google Cloud también propone servicios de *machine learning* a diferentes niveles, según el conocimiento y la experiencia en la construcción, y el uso de modelos *machine learning* requeridos para poder usarlos. En este caso, podemos dividir los servicios en dos grandes grupos: AI Platform y Cloud AI Building Blocks.

### AI Platform

Bajo el paraguas de **AI Platform** se agrupa un completo conjunto de servicios que abarcan todo el proceso de preparación de datos, entrenamiento, validación, despliegue y mantenimiento (MLOps) de modelos de *machine learning*. Estos recursos precisan que el usuario que los contrate tenga conocimientos y experiencia con estas tecnologías. Los servicios incluidos en AI Platform y algunos complementarios para abarcar hasta el último detalle en el flujo de trabajo son los que se muestran en la figura 9 y que, a continuación, describimos.



Figura 9. Servicios de AI Platform.

- ▶ **AI Platform Data Labeling Service** permite contratar a personas que etiqueten con gran precisión y exactitud colecciones de datos para usarlos después en el entrenamiento de modelos de *machine learning*. Cabe recordar que es de vital importancia que los datos usados en entrenamiento tengan un correcto etiquetado, que no desvíe el aprendizaje del algoritmo, para no caer en el conocido «*trash in, trash out*». Es decir: si los datos de entrenamiento no son precisos, las predicciones no podrán ser acertadas.
- ▶ **BigQuery Datasets.** Como se comentaba en la sección anterior, BigQuery dispone de una serie de *datasets*, esto es, conjuntos de tablas y vistas con enormes cantidades de información, bien creadas por el propio usuario, bien tomadas de los *datasets* públicos que ofrece este servicio. En uno y otro caso, se trata de datos interesantes para el entrenamiento de los modelos de *machine learning*.
- ▶ **Cloud AutoML** es un servicio bastante diferenciador, que facilita la construcción y el despliegue de modelos personalizados sin necesidad de tener grandes conocimientos de *machine learning*. Es decir, proporciona una serie de modelos preconstruidos, que acaban de entrenarse con los datos concretos del problema que

se quiere atacar. Existen diferentes modalidades de AutoML dependiendo del tipo de dificultad: AutoML Translation, Tables, Text & Document, Vídeo Intelligence y Vision.

- ▶ **AI Platform Notebooks** son instancias JupyterLab; en concreto, instancias de Deep Learning VM, un tipo de instancias VM optimizadas para desarrollar y entrenar modelos de *deep learning*, con las librerías apropiadas ya preinstaladas y listas para su uso, y que permiten escalar los recursos necesarios para adaptarse a las diferentes demandas.
- ▶ **AI Platform Training** es un servicio mediante el cual es posible entrenar modelos TensorFlow o *scikit-learn* en la nube, en contenedores. Esto permite tener una estructura distribuida, con un nodo máster que gestiona la tarea de entrenamiento y tantos nodos *worker* como se necesiten para entrenar de forma paralela, de manera que el proceso sea más ágil.
- ▶ **Explainable AI** ayuda a validar los modelos construidos en la etapa previa, proporcionando un conjunto de herramientas para entender e interpretar las predicciones realizadas por los modelos de *machine learning*. Esto permite revisar el comportamiento del modelo y comprobar posibles aspectos a mejorar.
- ▶ **AI Platform Vizier** es un servicio de optimización (en contraposición a la herramienta Explainable AI) que ajusta los hiperparámetros de los modelos de *machine learning* a través de pruebas con diferentes heurísticos y experimentos.
- ▶ **AI Platform Prediction** está en el plano de despliegue y permite alojar los modelos ya entrenados y optimizados para servir las peticiones de predicción que reciban, tanto en modo bloque (*batch*) como en modo *online*. Soporta versionado de modelos y escalado del servicio para ajustarse al volumen de peticiones.
- ▶ **AutoML Vision Edge** también permite desplegar modelos, en este caso, procedentes del servicio AutoML Vision.
- ▶ **TensorFlow Enterprise** proporciona un nivel de gestión y soporte de TensorFlow para requisitos empresariales.

- ▶ **AI Platform Pipelines** facilita el campo de MLOps para automatizar y gestionar flujos de trabajo con *machine learning*, que engloban la preparación de datos, el entrenamiento, la validación y el despliegue. Pipelines utiliza Kubernetes con TensorFlow para proporcionar y orquestar dichos flujos de trabajo.
- ▶ **Continuous evaluation** permite monitorizar la entrada y salida de los modelos desplegados, para detectar posibles derivas en la precisión del modelo, debidas potencialmente a cambios en la distribución estadística de los datos de entrada.
- ▶ **Deep Learning VM Image y Deep Learning Containers** son servicios de computación orientados a desarrollar modelos de *deep learning*, que incluyen hardware de aceleración con GPU y librerías asociadas. Estas instancias y contenedores son compatibles con JupyterLab, que oculta toda la complejidad que hay por debajo y permite al desarrollador centrarse en la construcción y el entrenamiento de los modelos, y olvidarse de la gestión de la infraestructura que los soporta.

## Cloud AI Building Blocks

La segunda modalidad de servicios *machine learning* está centrada en proporcionar API que no requieren ningún conocimiento sobre el tema, sino que se pueden aplicar directamente en el problema a solventar. Estas API, denominados Cloud AI Building Blocks, incluyen diferentes opciones:

- ▶ **Vision**: para clasificar imágenes o detectar objetos en ellas.
- ▶ **Video**: para clasificar vídeo o detectar objetos en ellos.
- ▶ **Translation**: para traducir entre diferentes lenguas.
- ▶ **Natural Language**: para clasificar textos, detectar entidades o extraer la estructura o significado.
- ▶ **Dialogflow**: permite desarrollar agentes conversacionales (*chatbots*).

- ▶ **Cloud Text-to-Speech API:** proporciona un servicio de traducción de texto a voz.
- ▶ **Cloud Speech-to-Text API:** proporciona un servicio de traducción de voz a texto
- ▶ **Recommendations AI:** permite crear recomendaciones basadas en datos a gran escala.
- ▶ **Cloud Inference API:** identifica correlaciones en datos de series temporales.

## Documentación en línea de Google Cloud Platform

Google Cloud (2020). *Google Cloud Products.* <https://cloud.google.com/products>

Google Cloud (2020). *Primeros pasos con Google Cloud.* <https://cloud.google.com/docs>

El recurso más completo y actualizado sobre Google Cloud Platform es su propia documentación en línea. Dada la rapidez con la que los diferentes servicios proporcionados aparecen, evolucionan y dejan de estar disponibles, es difícil encontrar manuales que se mantengan actualizados más allá de unos pocos meses. Incluye documentación tanto sobre cada uno de los servicios concretos como sobre conceptos más generales, que ayudan a entender los principales casos de uso y mejores prácticas.

- 1.** ¿Cómo se organizan los recursos, los servicios y las políticas de seguridad que contrata y configura un usuario u organización en Google Cloud?

  - A. En *folders*, que contienen proyectos.
  - B. En proyectos, que contienen *folders*.
  - C. En proyectos y zonas.
  - D. En zonas, que contienen diferentes *folders*.
- 2.** Si queremos aumentar la disponibilidad de un servicio GCP, ¿qué debemos hacer?

  - A. Desplegarlo en la región más cercana a su uso.
  - B. Desplegarlo en una región que no presente problemas legales con la información que gestiona.
  - C. Desplegarlo como recurso regional o multirregional.
  - D. Desplegar una instancia VM que esté siempre ejecutándose.
- 3.** Elige la respuesta incorrecta:

  - A. GCP proporciona una serie de servicios de AI bajo AI Platform, para usuarios no expertos en el dominio, los cuales quieran usar AI en sus aplicaciones sin desarrollar ningún modelo.
  - B. GCP proporciona una serie de servicios de AI bajo AI Platform, para usuarios expertos en el dominio que quieran usar AI en sus aplicaciones desarrollando sus propios modelos.
  - C. GCP proporciona una serie de servicios de AI bajo Cloud AI Building Blocks, para usuarios no expertos en el dominio, los cuales quieran usar AI en sus aplicaciones sin desarrollar ningún modelo.
  - D. Entre los servicios AI para uso directo, se pueden encontrar herramientas de clasificación de imágenes o vídeo, o traductores entre diferentes idiomas.

4. En cuanto a la seguridad, ¿qué esquema sigue GCP?
- A. Un esquema de seguridad compartida, donde GCP se hace siempre cargo de todos los niveles, excepto de los datos.
  - B. Un esquema de seguridad compartida, donde GCP se hace cargo de ciertos niveles, que dependen del servicio desplegado.
  - C. El usuario debe hacerse cargo de la seguridad de todo el sistema, que sigue un modelo de cuatro capas.
  - D. Un esquema de seguridad compartida de cuatro capas, donde el usuario solo se hace cargo de la capa a más alto nivel y Google Cloud, de todos los aspectos de las otras tres.
5. ¿Qué opción es más interesante para ejecutar tareas cortas y no críticas, que se podrían repetir si fuera necesario?
- A. Instancias VM normales.
  - B. *Preemptible instances* VM.
  - C. *Sole-tenant VM instances*.
  - D. *One use VM instances*.
6. Una empresa quiere almacenar los datos históricos de las nóminas de los empleados, con el único objetivo de hacer frente a una posible auditoría en los cinco años siguientes al pago de cada nómina. ¿Qué opción de almacenamiento de GCP es la más adecuada en cuanto a acceso y coste?
- A. Cloud Storage Coldline.
  - B. BigTable.
  - C. Cloud Storage Archive.
  - D. Cloud Persistent Disks.

- 7.** Cuando se crea un proyecto en Google Cloud, ¿cómo se interconectan los servicios que engloba?

  - A. Se crea una VPC, que contiene el rango de direcciones IP que se asignan los servicios.
  - B. Hay que definir siempre manualmente las subredes de una VPC para tener disponibles direcciones IP que asignar a los servicios.
  - C. Se crea automáticamente una subred dentro de la VPC, que contiene el rango de direcciones IP disponibles para asignar a los servicios.
  - D. Un proyecto solo se puede interconectar con otro, pero los servicios dentro de un proyecto no se interconectan y, por tanto, no se necesitan direcciones IP.
  
- 8.** Elige la respuesta incorrecta: Si quisieramos desplegar un clúster Hadoop en GCP, podríamos...

  - A. Usar varias instancias VM configuradas manualmente como clúster e instalar las herramientas del ecosistema Hadoop deseadas.
  - B. Usar el servicio Dataproc.
  - C. Usar el servicio Dataflow.
  - D. Usar varios contenedores GKE configurados manualmente como clúster e instalar las herramientas del ecosistema Hadoop deseadas.
  
- 9.** ¿A qué base de datos de código libre se asemeja BigTable?

  - A. Es un motor propietario único de Google, muy diferente a cualquier otra base de datos existente.
  - B. MongoDB.
  - C. Cassandra.
  - D. HBase.

**10.** Relaciona cada servicio GCP con el que sería su equivalente en proyectos Apache:

Dataproc	1
Cloud Pub/Sub	2
Dataflow	3
BigQuery	4

A	Apache Spark
B	Hadoop
C	Impala
D	Apache Kafka