- **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 preprocesamiento, en general, se utiliza la API de Spark SQL. 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 preprocesamiento 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 ML de Spark. En concreto, Spark requiere los formatos mencionados en la Tabla 1.

Problemas de clasificación (clase codificada como número real)

Features	*Label*
[-32.2, 4.5, 1.0, 6.7]	1.0
[-40.8, 2.25, 4.0, 2.3]	0.0

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

Features	*Label*
[-32.2, 4.5, 1.0, 6.7]	0.72
[-40.8, 2.25, 4.0, 2.3]	-4.56

Problemas de *clustering* (no existe la columna *label*)

Features
[-32.2, 4.5, 1.0, 6.7]
[-40.8, 2.25, 4.0, 2.3]

Problemas de recomendación (con filtrado colaborativo)

User	*Item*	*Rating*
2	3	4.5
1	19	3.21

Tabla 1. Formatos de Spark. Fuente: elaboración propia.

Como puedes ver, 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 el 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, puedes consultar la documentación de Spark.

Apache Spark. (s. f.). *Extracting, transforming and selecting features*. https://spark.apache.org/docs/latest/ml-features.html

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`).

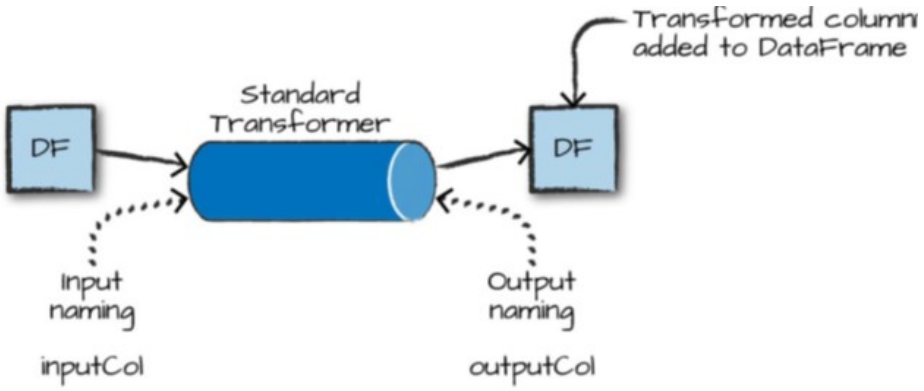

Figura 7. Funcionamiento de un transformador de Spark ML. Fuente: Chambers y Zaharia, 2018.

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 7.

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. Por 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", "mobi1e", "userFeatures", "c1icked"])
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)
```

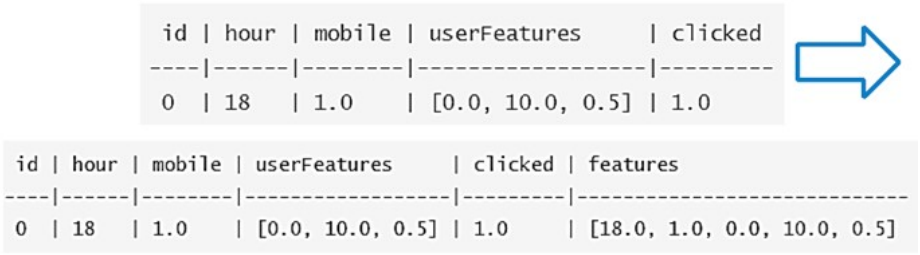

Figura 8. Ejemplo. Fuente: elaboración propia.

Para 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 *fi t* previo, no solo a los algoritmos de aprendizaje automático que conocemos como modelos propiamente.

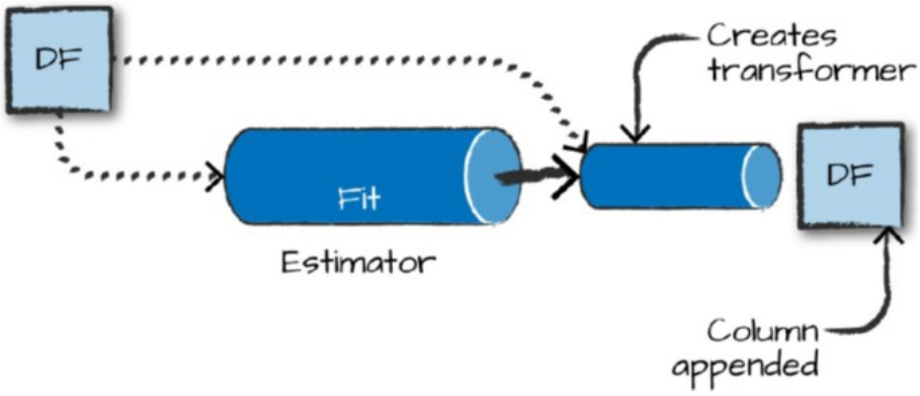

Figura 9. Estimador que genera un transformador. Fuente: Chambers y Zaharia, 2018.

Estimadores más comunes

- `StringIndexer` es un estimador para preprocesar 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"])
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 = indexerModer.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]

Tabla 2. Ejemplo. Fuente: elaboración propia.

- **Cualquier modelo de predicción (ML).** 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 ML, 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 (lambda), 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 ML, 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 es la 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 ML,

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 Figura 10 muestra un *pipeline* antes de llamar a `fit`.

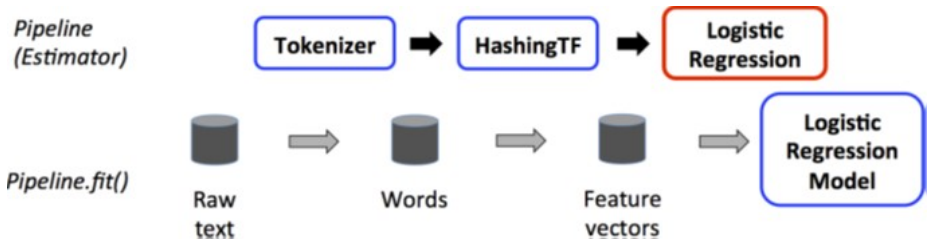

Figura 10. *Pipeline* sin ajustar (arriba) y procesamiento que ocurre al llamar a `fit` (abajo). Fuente: Apache Spark.

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.

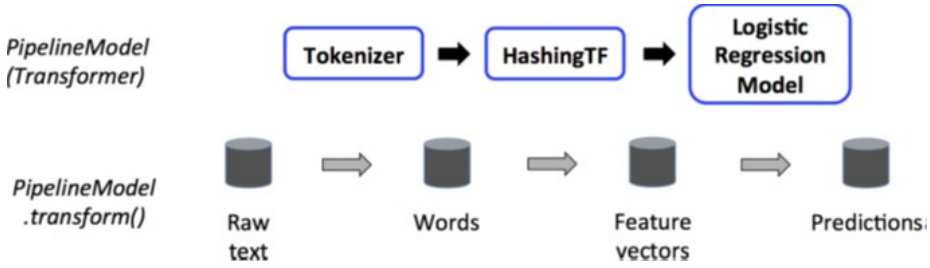

Figura 11. *Pipeline* ajustado (`PipelineModel`, arriba) y procesamiento que ocurre al llamar a *transform* (abajo). Fuente: Apache Spark.

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)
```

Spark Structured Streaming

En esta parte, vamos 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; 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.

4.5. MapReduce

Procesamiento de ficheros distribuidos

Como se vio anteriormente, en la **capa de procesamiento** es donde se tratará tanto aquella información que se ha almacenado en la capa de almacenamiento (procesamiento *batch* o lotes) como la información que llega en tiempo real (procesamiento en tiempo real).

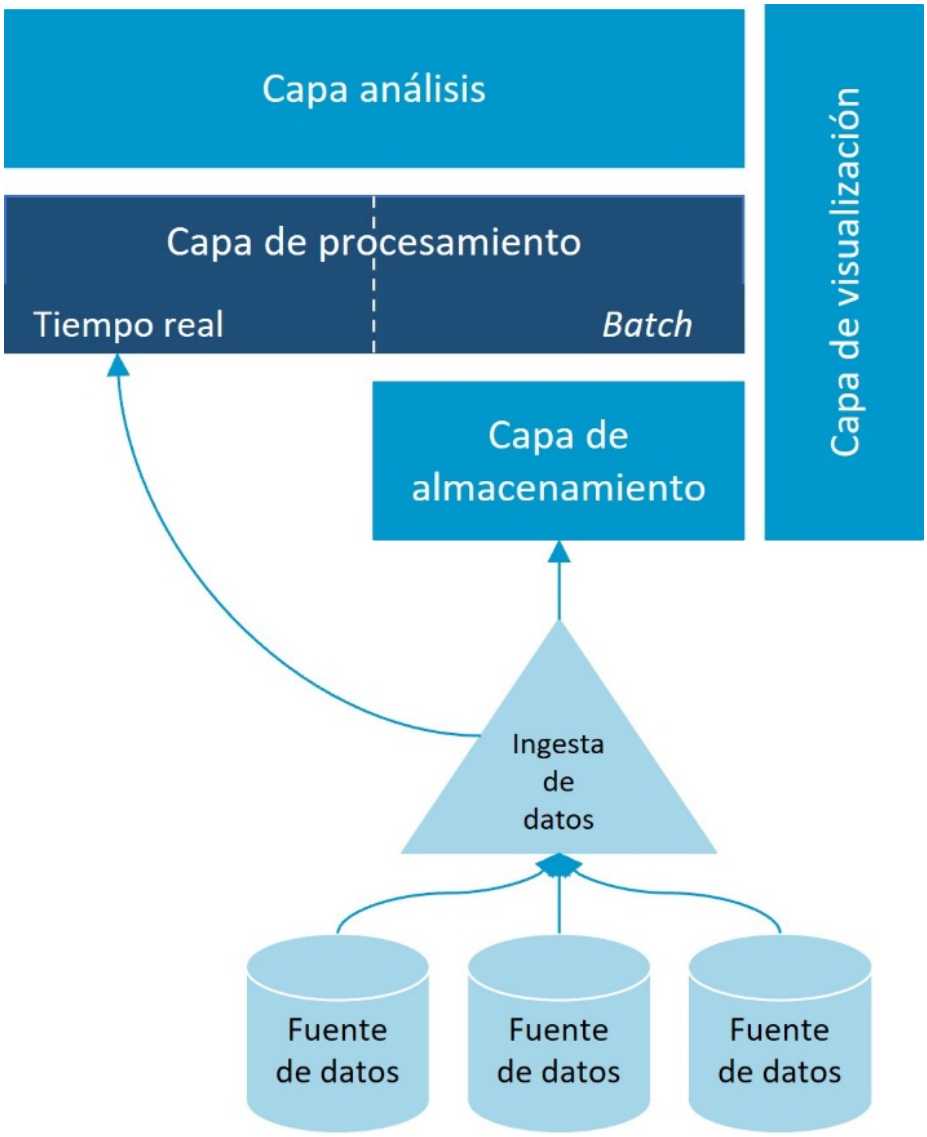

Figura 12. Representación arquitectura *big data*. Capa de procesamiento. Fuente: elaboración propia.

Tema 4. Computación distribuida. Computación paralela

Dentro de la capa de procesamiento ya se vio que había dos tipos: procesamiento por lotes y procesamiento en tiempo real.

Procesamiento por lotes. MapReduce

MapReduce se encarga de automatizar el procesado de los datos en paralelo a través de distintos nodos de un clúster gestionando las tareas necesarias para la realización del trabajo. También proporciona la capacidad de reconducir el proceso en el momento en que detecta que uno de los nodos falla.

MapReduce fue concebido para una función muy concreta, indexar el contenido de cada página web hasta completar todo el universo de la World Wide Web. A medida que se extendieron las aplicaciones de Hadoop a otras lógicas de negocio diferentes empezaron a aparecer ciertos **problemas** con 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 un modelo abstracto de programación general e implementación del modelo como bibliotecas de programación, para procesamiento paralelo y distribuido de grandes volúmenes de datos, inspirado en la técnica «divide y vencerás».

La gran **ventaja** que proporciona tanto el modelo como la implementación que suministraron los autores es que abstrae 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.** Es una 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 MapReduce: 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 **fase *input*** de la Figura 13.

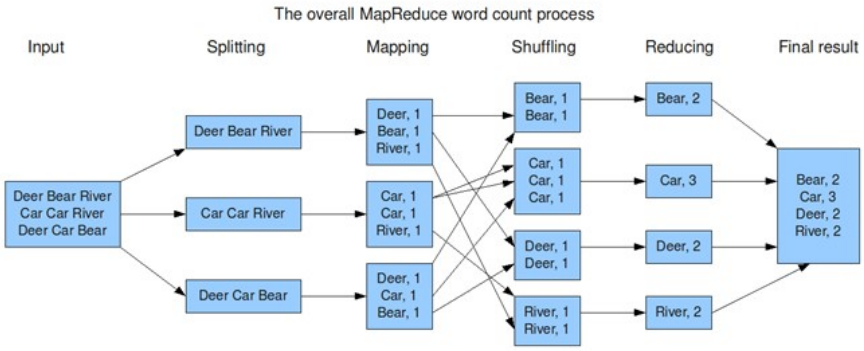

Figura 13. Representación del problema de contar ocurrencias resuelto con MapReduce. Fuente: Lafourcade, 2019.

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 vendrá de la información almacenada en cada bloque de los nodos donde se almacena la información, que representan cada uno un bloque de HDFS. Así de manera paralela se irá leyendo línea a línea en la fase de Mapping, se hará lo que se denomina *splitting* para separar cada palabra de cada frase y con esa información se van generando tuplas (clave/valor) por cada palabra que se recibe de forma que se genere lo siguiente `(palabra_recibida,1)` dando como resultado, las tuplas que se muestran en la **fase de *mapping*.**

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 *shuffling*. En esta fase se generan 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, el *framework* invoca automáticamente a la segunda función que el programador debe escribir: la función `reduce`. Esta es recibida por las tuplas de la fase de *shuffling* (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 la **fase de *reducing*.**

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, por 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). Por ejemplo, la función `reduce` recibe `(hola, [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. 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, leerlos y pasarlos 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 ML, 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. Además, MapReduce trabaja principalmente en Java, aunque tiene alguna librería que permite trabajar con Python también.

- Además, el **código** que hay que generar es muy complicado y requiere muchas líneas de código para una tarea relativamente simple. Por ejemplo, el código que habría que generar para el ejemplo visto anteriormente sería:

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

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

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

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

No se pretende que se aprenda que es lo que está haciendo el código, únicamente se busca que se entienda lo complicado que puede llegar a ser además frente a otras tecnologías y que, para hacer exactamente lo mismo, lo pueden ejecutar todo en una única línea.

Arquitectura de MapReduce

Hay dos versiones claramente diferenciadas a la hora de ver la arquitectura de MapReduce, la versión 1.0 y la 2.0 y posteriores. La **diferencia** principal reside en los servicios utilizados en cada caso y en la mejora de rendimiento que hay en la versión 1 con respecto a la versión 2.

Versión 1 de Hadoop

En la versión original de MapReduce (MapReduce 1) entran en juego dos componentes para realizar la tarea del ejemplo: JobTracker y TaskTracker.

- JobTracker. Es un servicio desarrollado en Java también que se ejecuta en un nodo maestro y es responsable de recoger la petición del cliente y de asignar a los TaskTrackers las tareas concretas que deberán realizar en el proceso. El JobTracker está en el nodo maestro (NameNode), recordemos que es donde se guarda la información de en qué nodo (DataNode) está cada fragmento de cada fichero. Los nodos en los que se van a ejecutar las tareas definidas por el JobTracker serán precisamente los DataNodes, en donde viven los datos. En el caso de que, por cualquier motivo, uno de los nodos fallase el JobTracker se encargaría de redirigir la tarea a otro nodo donde estuviera alojada una réplica de los datos.

- TaskTracker. Se trata de un demonio (*daemon*) que acepta y ejecuta las tareas asignadas: map, reduce y shuffle. Adicionalmente el TaskTracker envía una señal al JobTracker para advertirle de que está vivo y listo para ejecutar la tarea, así como de indicarle como de «ocupado» se encuentra. Una vez iniciada la ejecución de un proceso mantiene un *feedback* continuo de su progreso, llegando incluso el caso de que si una tarea en un nodo se ralentiza demasiado la manda ejecutar en otro nodo que tenga el mismo bloque de fichero sobre el que se está ejecutando.

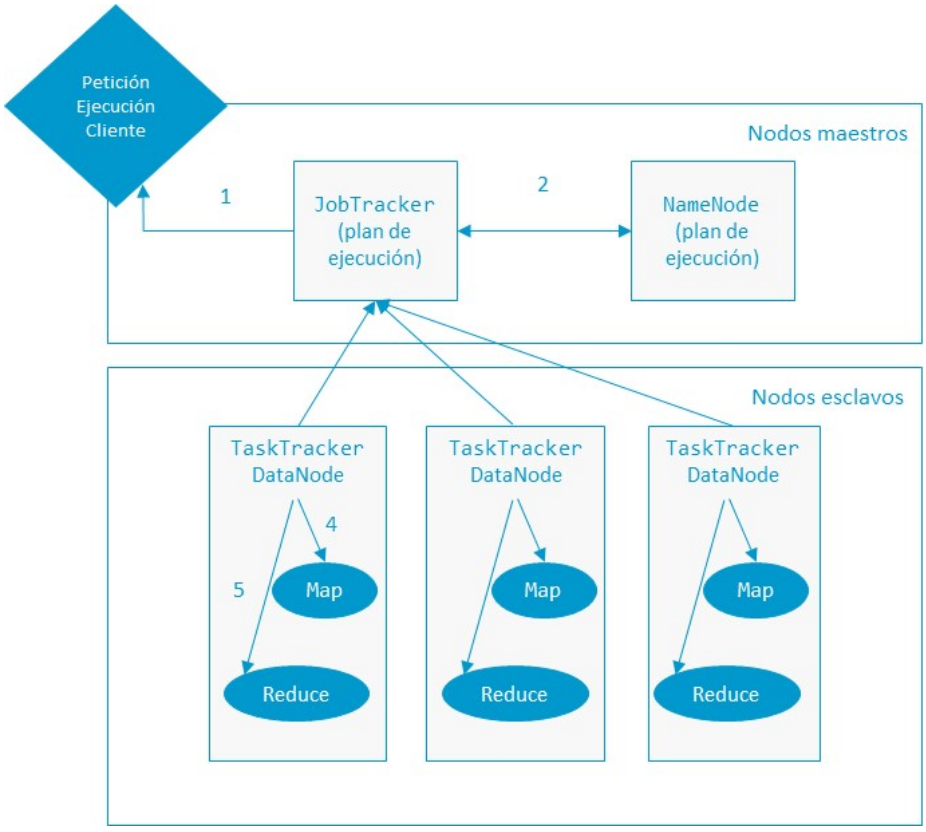

Figura 14. Pasos ejecución MapReduce en versión 1.0 de Hadoop. Fuente: elaboración propia.

- Un cliente, mediante un programa en MapReduce, envía un trabajo a realizar al JobTracker , en el ejemplo anterior era un conteo de las palabras en una serie de frases.

- El JobTracker obtiene del NameNode la ubicación de los bloques de los datos en los DataNodes.

- A partir de ahí el JobTracker manda el programa del cliente en HDFS y asigna las tareas que tienen que realizar los TaskTrackers en los nodos en los que ha localizado los datos.

- El TaskTracker inicia las tareas Map en cada DataNode trasladando el programa del cliente de la ubicación compartida HDFS al nodo donde se van a ejecutar las tareas (los tres nodos que trabajan con cada una de las frases en el diagrama). Cuando se

finaliza la tarea de mapear (Map) en todos los nodos se genera un fichero intermedio a nivel local en el nodo del TaskTracker . Este fichero es el que se pasa a la tarea de Reduce.

- El proceso de reduce trabaja con todos los datos recibidos de las tareas de mapeo (Map) y, una vez procesado, escribe el *output* final en el sistema de ficheros HDFS. Al finalizar borra los datos intermedios generados por el TaskTracker en los distintos nodos.

Una de las diferencias claves con otros sistemas de gestión de datos es que, como acabamos de ver, en Hadoop, llevamos el programa hasta el nodo en el que se alojan los datos y no al revés. Este cambio de filosofía es precisamente el que permite la **eficiencia del procesado distribuido** de los datos en Hadoop.

Pero esta arquitectura tenía un problema en la **escalabilidad** y es que el Jobtracker podía llegar a ser un problema, ya que tenía que ocuparse de saber cómo estaban todos los Tasktrackers de «ocupados» para poder mandarles o no más tareas y a la vez tenía que estar pendiente de todas las ejecuciones que se estuvieran realizando para por ejemplo ver si era necesario relanzarlas. Cuando el número de nodos crecía en el clúster, el rendimiento bajaba bastante debido a lo comentado.

MapReduce fue concebido para una función muy concreta, indexar el contenido de cada página web hasta completar todo el universo de la World Wide Web. A medida que se extendieron las aplicaciones de Hadoop a otras lógicas de negocio diferentes empezaron a aparecer ciertos problemas con MapReduce.

Adicionalmente MapReduce no aportaba ninguna **alternativa** para la gestión de los recursos disponibles, solo permitía usar su *framework*; un *framework* que estaba concebido para procesos en lotes (*batch*). Con la aparición de los nuevos modelos de programación, se hizo patente la necesidad de soportar otros paradigmas de programación (Impala, Spark…) y una mejor forma de gestionar los recursos.

Versión 2 de Hadoop. YARN

En la versión 2 de MapReduce, se introduce un nuevo gestor de recursos llamado **YARN** (*yet another resource negociator*). La llegada de YARN sobre MapReduce cambia el funcionamiento de esta arquitectura mejorando bastante su rendimiento.

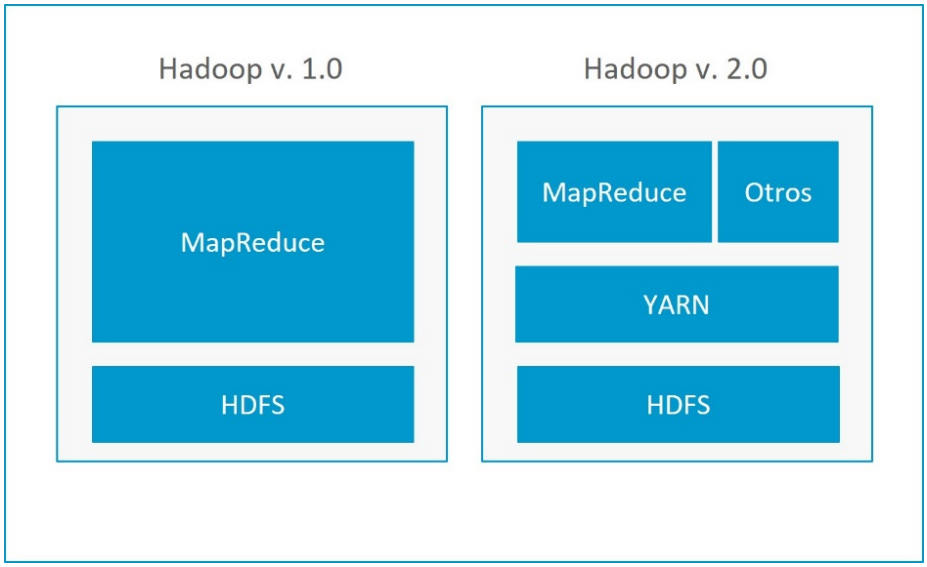

Figura 15. Diferencia arquitectura Hadoop v. 1.0 vs. v. 2.0. Fuente: elaboración propia.

- YARN (*yet another resource negotiator*): **evolución de MapReduce.**
- Fue introducido en **Hadoop 2** para mejorar MapReduce y permitir diferentes tipos de procesamiento de datos: Tez, Spark, MapReduce… no solo *batch.*
- YARN es el encargado de **gestionar un clúster**, lo que quiere decir, que se encarga de gestionar los recursos (memoria, disco, procesos).
- YARN **expone una serie de API** que son aprovechadas por *frameworks* de cómputo, como MapReduce, Spark, etc. (todos ellos pueden ejecutar sobre YARN), es decir ya no es necesario utilizar MapReduce para acceder a HDFS, lo que aprovecharon muchas aplicaciones.
- Tiene **procesos distintos** para el proceso de datos y para la gestión del clúster.

Ejemplo de la vida real que muestra la potencia de Hadoop 2.0 sobre Hadoop 1.0

- Yahoo fue capaz de ejecutar casi el doble de *jobs* al día con YARN que con Hadoop 1.0.
- También experimentaron un aumento considerable en la utilización de la CPU.
- Yahoo incluso llegó a afirmar que la actualización a YARN equivalía a añadir 1000 máquinas a su clúster de 2500 máquinas

En YARN siguen existiendo los dos **procesos:** `Map` y `reduce`, sin embargo, la manera en que se gestionan los recursos del ecosistema es completamente diferente. Para realizar un trabajo YARN, proporciona una nueva arquitectura en la que se modifica sustancialmente la forma en que se distribuyen las tareas entre los nodos. Para ello, YARN incorpora los siguiente nuevos elementos a la ecuación:

- `ResourceManager` (uno por cada clúster) el responsable de organizar los recursos de los nodos esclavos a la medida de las necesidades de cada tarea. Se encarga de iniciar los trabajos (*jobs*) en el clúster maestro a petición de los usuarios (cliente). Puede sonar parecido a lo que hacíamos con el `JobTracker` en la versión uno de MapReduce, pero, como veremos más adelante, hay grandes diferencias.
- `NodeManager` (uno por cada nodo esclavo) sustituye la función que realizaban los `TaskTrackers` en la versión anterior. Son los responsables de iniciar los procesos de una aplicación en particular y además se encargan de solicitar los recursos necesarios al `ResourceManager` garantizando que las tareas asignadas no superen la capacidad de los recursos disponibles en el nodo.
- `Job History Server` (uno por clúster) es un proceso opcional cuya función es mantener un archivo de los logs del trabajo (*job*) ejecutado.

En YARN, el cliente inicia un trabajo (*job*) a través del ResourceManager (gestor de recursos), que se encarga de gestionar los recursos necesarios. Cuando el **gestor de recursos** manda una tarea a un nodo esclavo este necesita asignar una serie de sus recursos para la realización de la tarea. El gestor de recursos es el encargado de asignar los recursos a través de la creación de contenedores (*container*).

Un **contenedor** dedica una parte de la memoria y de la capacidad de la CPU del nodo esclavo para realizar la tarea. En un mismo nodo pueden convivir varios contenedores a un mismo tiempo y, por lo tanto, compartir una parte de los recursos del nodo.

YARN introduce también el concepto de ***application master*** (maestro de aplicaciones), un sistema que controla las aplicaciones, que es otra manera de llamar a los trabajos o las consultas que realizan los usuarios. Hay un maestro de aplicaciones por cada trabajo que se realiza. El maestro de aplicaciones también vive en un contenedor, es decir, también se le asignan unos recursos en el nodo esclavo.

El maestro de aplicaciones es el responsable de solicitar los **recursos** de los contenedores al gestor de recursos. Una vez que este asigna los recursos al contenedor de un nodo esclavo es cuando se empiezan a ejecutar las tareas. El NodeManager se encarga de supervisar el proceso y de validar el *output* resultante. Como ya se ha mencionado un mismo nodo esclavo puede gestionar varios contenedores a la vez, pero cada tarea se ejecutará siempre en un contenedor distinto.

El **maestro de aplicaciones** corre en uno de los nodos esclavos, sin embargo, recurre al gestor de recursos en el nodo maestro para solicitar recursos no solo para el nodo en que se ejecuta, también se encarga de solicitar los recursos (contenedores) para aquellos nodos donde están alojados los bloques de los datos (DataNode) con los que va a trabajar a través del NodeManager correspondiente.