- 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 . Es 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 o 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**

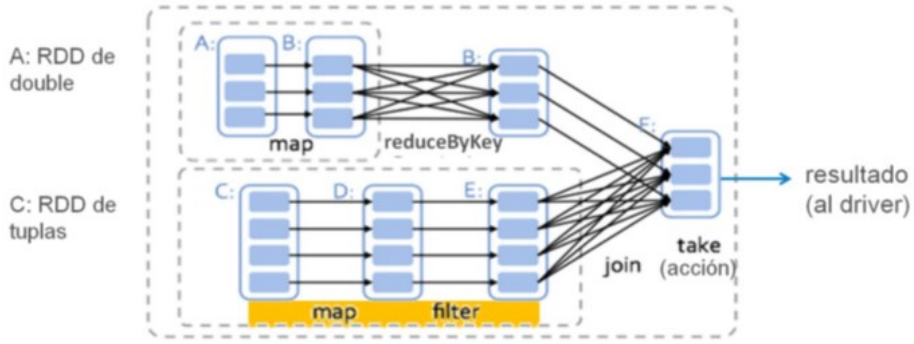


Figura 4. Ejemplo de transformaciones y de la acción take mediante el uso de RDD. Fuente: elaboración propia.

```
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, se ha 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 tres. 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 el 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 del 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 compartan 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 DOS 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.

## Jobs, stages y tasks

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 *job* de Spark es todo el procesamiento necesario para llevar a cabo una acción del usuario. 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. Por ejemplo, `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 núcleo físico (CPU) a una partición de un RDD.

La Figura 5 representa la división en tareas, etapas y trabajos para el ejemplo de código contenido en el anexo.

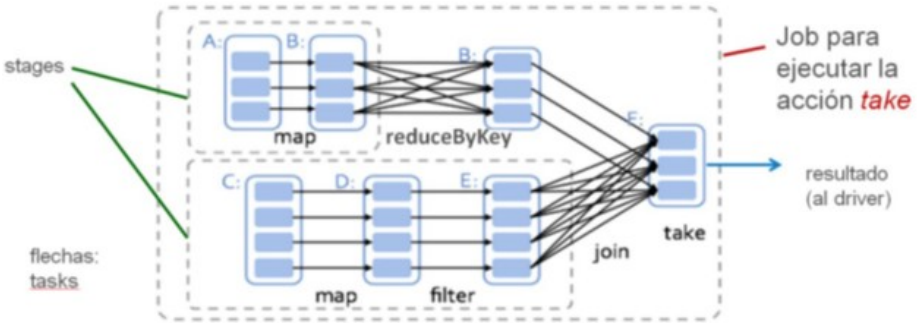


Figura 5. Representación de *jobs*, *stages* y *tasks* en Spark. Fuente: elaboración propia.

## 4.3. Spark II

### 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 que está 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`, por ejemplo, `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 ( `dataframe` es 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 *dataframe* 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`.

A continuación, 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.

## 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, etc.).
- Cola distribuida **Kafka** 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 el 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 los 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:
  - 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.
  - 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.
  - Podemos pedir a Spark que no trate de inferir el esquema. En este caso, todos los datos se leerán como si fueran texto ( `string` ).
  - 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 las 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 el 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 objecto 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, csv o avro , aunque no todos los formatos lo tienen.

Vamos a ver algunas **particularidades** de ciertos formatos concretos:

- 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 (*parser*) 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 *dataframe*s 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 o 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 una sintaxis específica. Para más detalles, puedes consultar la web de Spark.

**API estructurada de Spark: manipulación de *dataframes***

Una vez que sabemos qué es un *dataframe*, cómo leer datos para crear *dataframe*s 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.

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 *dataframe*s 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 *dataframe*s, 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 *dataframe*s) 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 rellenarán 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 *dataframe*s. Dichas funciones generan columnas y *dataframe*s 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 *dataframe*s. Entre ellas, podemos encontrar:

- **Unión de *dataframe*s:**

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

- **Diferencia de *dataframe*s:**

```
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 *dataframe*s. 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`:

```
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 siguiente fragmento de código 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.
```

## Spark SQL

**Spark SQL** ofrece una potente opción, que consiste en aplicar operaciones escritas como consultas en lenguaje SQL a *dataframe*s 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 *dataframe*s; 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.** 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.

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.

Existen dos opciones para ejecutar **consultas SQL en Spark:**

- **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 SparkSQL.** El método `.sql()` de la `sparkSession` recibe como argumento una consulta SQL, que puede hacer referencia a cualquier tabla registrada en Hive, y devuelve un *dataframe* con el resultado de la consulta. Para registrar un *dataframe* como una tabla de Hive, lo cual genera (solamente) los metadatos necesarios en el *metastore* de Hive, existen varios métodos; entre ellos, podemos mencionar `createOrReplaceTempView` , que la registra solo durante esta sesión. Spark ofrece más métodos para crear tablas en Hive y 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) se pueden conectar a Spark y realizar consultas de estilo SQL.

**Tablas en Spark SQL**

El elemento de trabajo en Spark SQL son las **tablas,** equivalente a los *dataframe*s 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 no gestionadas por Spark.** Cuando se define una tabla desde ficheros 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).

- **Tablas gestionadas por Spark.** Cuando se guarda un *dataframe* como una nueva tabla (usando, por ejemplo, saveAsTable sobre un *dataframe*), entonces se crea una tabla gestionada por Spark, ya que son datos nuevos que necesitan almacenarse y Spark es el encargado de ello.

- **Tablas externas.** Esta opción existe por compatibilidad con Hive y permite crear tablas no gestionadas por Spark; es decir, los metadatos estarán gestionados por Spark, pero no así los datos. Para ello, 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 `o`. 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;
```

## 4.4. Spark III

### 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 6 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 6 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.

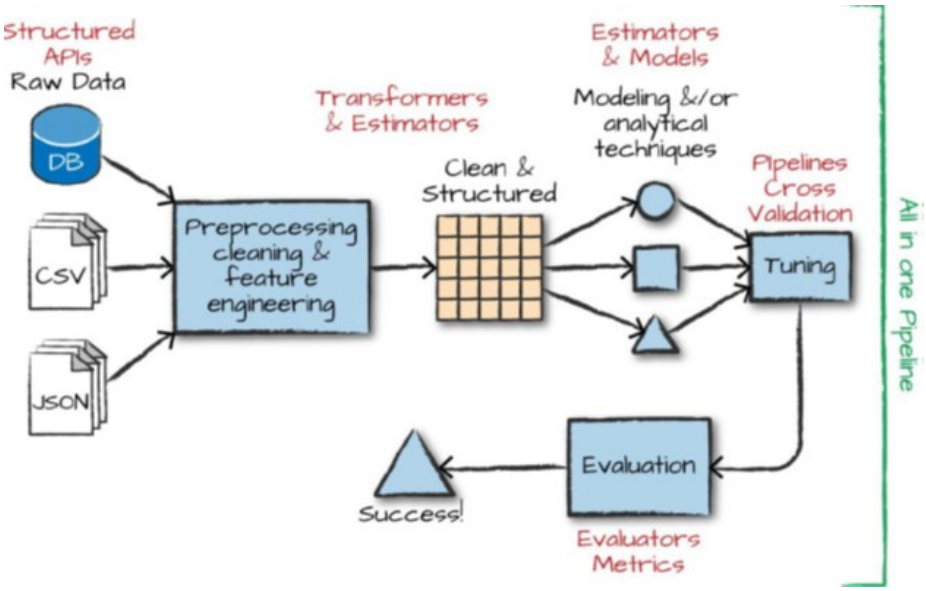


Figura 6. Etapas en el proceso de entrenamiento de un modelo a partir de datos. Fuente: Chambers y Zaharia, 2018.

**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 conozcan 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.** Resulta 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.** Por ejemplo, PMML, para leerlo y explotarlo con otra herramienta no distribuida (Python en especial, aunque también R soporta archivos en formato PMML).