



UNIVERSIDAD  
COMPLUTENSE  
DE MADRID



**ntic**  
master  
**School**

# Apache Spark I

Dr. Pablo J. Villacorta  
Septiembre de 2020



# Agenda

- Motivación
- Introducción a Apache Spark
  - Arquitectura de Spark
- Estructuras de datos fundamentales
  - RDDs
  - DataFrames
- Transformaciones sobre DataFrames



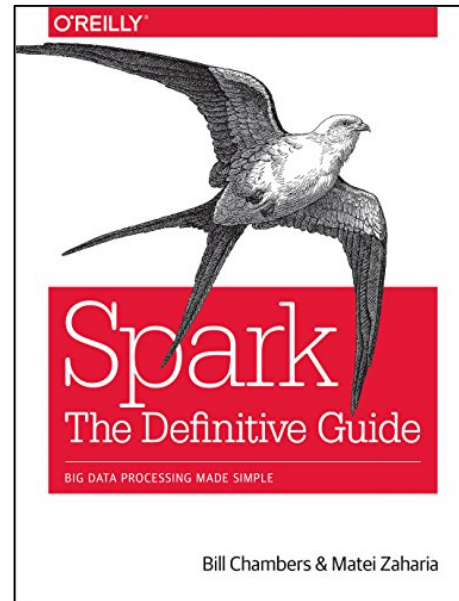
# Usos de Spark actualmente

Algunas de las cosas que las empresas están haciendo con **Apache Spark** actualmente\*:

- **Importar datos** de múltiples fuentes (HDFS, BBDD relaciones y no relacionales) y procesarlos con Spark de manera uniforme y agnóstica a su procedencia
- **Limpieza de datos masivos**
- **Agregación de datos** para crear un data warehouse
- **Exploración de datos masivos**
- **Cálculos y agregaciones en tiempo real**
- **Encontrar patrones** en datos masivos con algoritmos de ML
- Implementar **nuevos algoritmos** de ML
- ...



# 1 Introducción a Apache Spark



# Apache Spark



Apache Spark es un **motor unificado de cálculo** y también un conjunto de **bibliotecas de programación** para **procesamiento paralelo de datos** en clusters de ordenadores

Se considera una herramienta estándar para cualquier **científico de datos y desarrollador Big Data**

Se puede ejecutar en cualquier sitio desde un **portátil** a un cluster de **miles de ordenadores**



# Apache Spark



- **Unificado:** funciona de la misma manera independientemente de que usemos las bibliotecas de Spark desde R, Python, Scala, Java, o SQL.
- El código que Spark realmente ejecuta en paralelo sobre el cluster es único (Spark está escrito en Scala)
- El mismo (único) motor de cálculo es capaz de procesar datos en modo batch, en streaming, de procesar código en SQL, ejecutar algoritmos de Machine Learning, ...



# Apache Spark



- **Motor de cálculo:** de propósito general, con el que podemos programar cualquier cosa
  - Ejecutar consultas SQL en paralelo en el cluster
  - Manipulación/transformación/limpieza de datos
  - ETL (extract-transform-load)
  - Procesamiento de datos en tiempo real
  - Machine learning a gran escala



# Apache Spark



- **Conjunto de bibliotecas** para poder usar Spark desde cualquier programa en **Python, R, Java o Scala**
  - Spark se importa como APIs de programación (bibliotecas) fáciles de usar
  - Al utilizar esas funciones, se ejecutan de manera automática en paralelo en el cluster
  - En este curso veremos **pyspark**, la API de Spark para el lenguaje Python





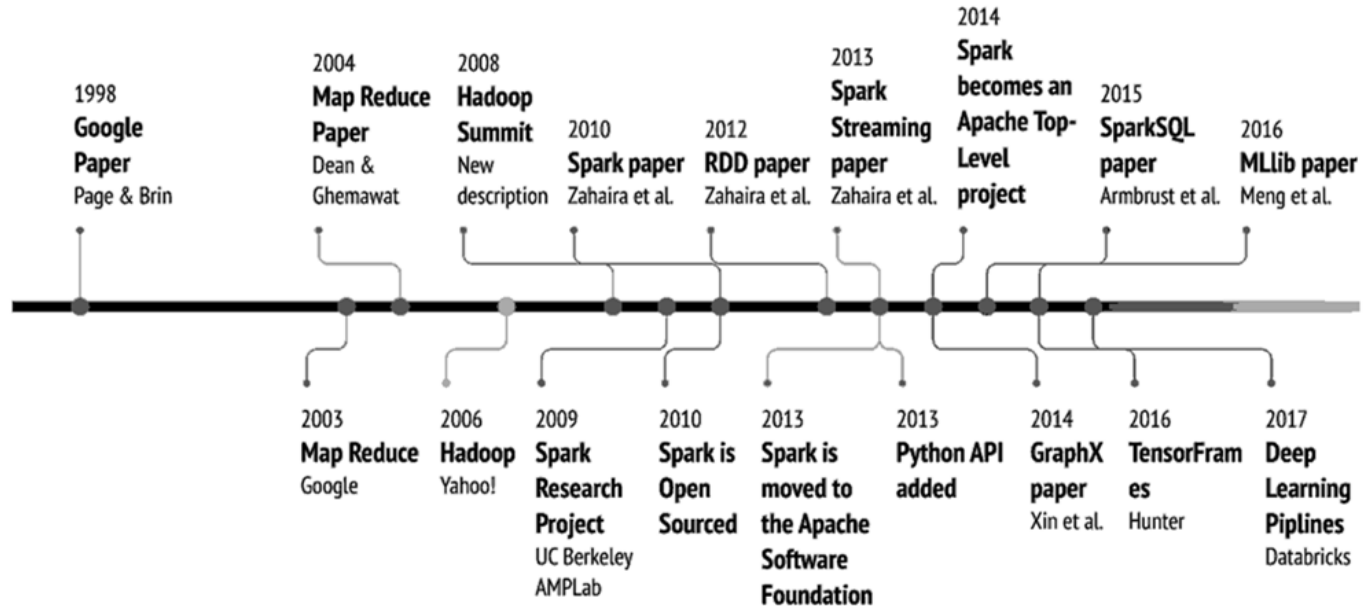
# Apache Spark



- **Procesamiento de datos en paralelo:** de manera automática, el código se ejecuta en paralelo sobre todas las máquinas del cluster
  - Spark abstraer y se ocupa automáticamente de todo lo relativo a redes y comunicación entre las máquinas
  - El programador utiliza las bibliotecas como si estuviera manejando tablas de datos sin nada en especial, casi olvidando que estamos usando un cluster de ordenadores



# Evolución de Spark



By Favio Vázquez



# El origen de Spark

Empezó en 2009 como tesis doctoral de Matei Zaharia (actualmente CTO de Databricks) en **UC Berkeley**, grupo **AMPLab 2009**

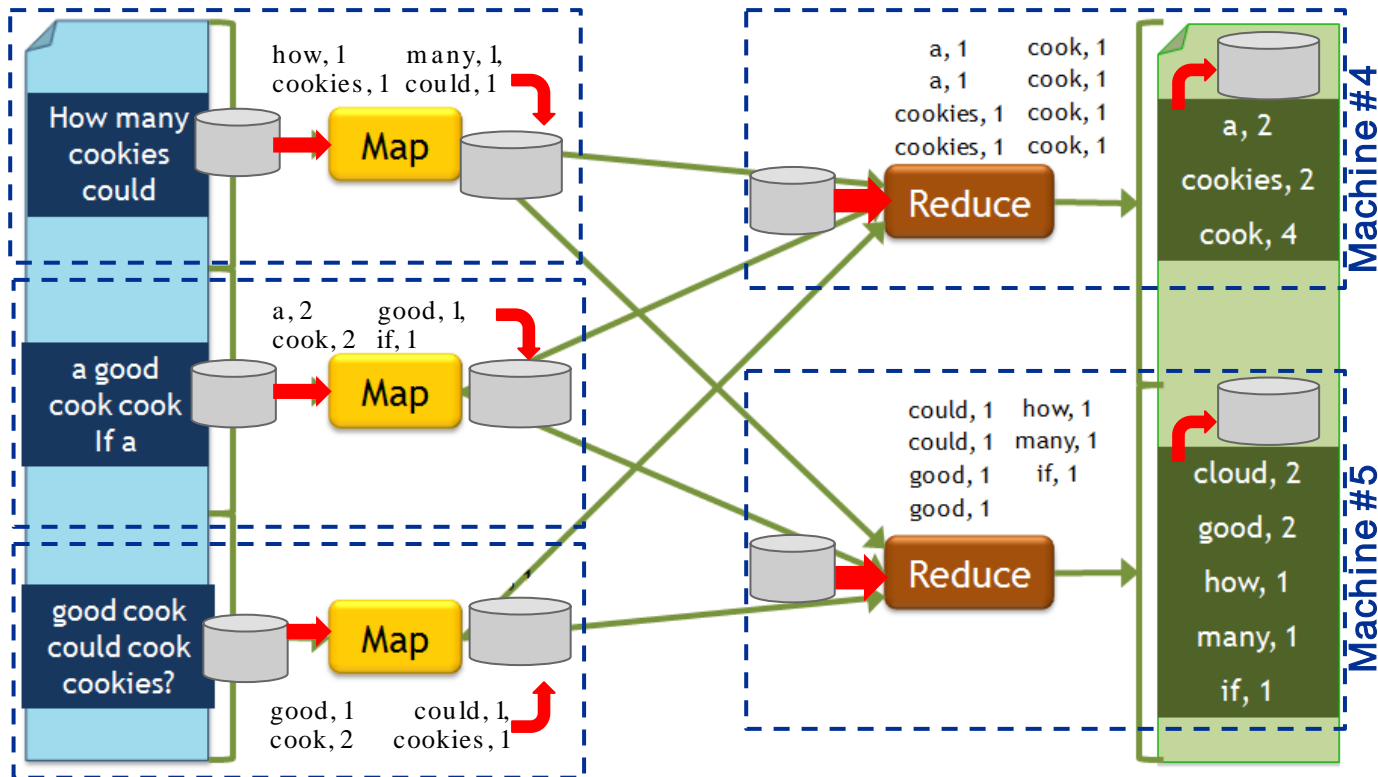
Motivada por los inconvenientes de MapReduce

- **La fase Map** siempre escribe a disco, lo cual es **lento**
- Todo el rato hay **movimiento de datos** entre máquinas (**lento**)
- **La fase Reduce** lee de nuevo de disco
- **Algoritmos de ML:** requieren varias etapas MapReduce encadenadas (**releer de disco en cada iteración**).
- Paradigma de programación **anti-intuitivo**



# Lo que hace MapReduce

Machine #1  
Machine #2  
Machine #3



Disco duro de cada nodo



# El origen de Spark

- Solución: cargamos los datos en trozos (particiones) en la memoria RAM de cada nodo del cluster, hacemos los cálculos en las máquinas en paralelo, y dejamos el resultado en memoria.
- No se mueven datos entre máquinas salvo que sea imprescindible para la operación.
- No se escribe a disco salvo que el programador lo pida explícitamente.

*La memoria RAM es ~10x-100x más rápida que el disco duro*

last_name	phone	email
Burks	(916) 342-8003	burks@yahoo.com
Todd	(917) 234-5004	todd@yahoo.com
Fisher	(917) 234-22234	tameka.fisher@aol.com
Spence	(912) 234-0001	daryl.spence@aol.com
Rice	NULL	crcise@msn.com

RAM memory

Machine #1

last_name	phone	email
Bean	NULL	lbean@hotmail.com
Hays	(917) 234-5004	bobhays@hotmail.com
Duncan	NULL	duncan@yahoo.com
Baldwin	(912) 234-0001	dbaldwin@msn.com
Newmann	NULL	pnewmann@gmail.com

RAM memory

Machine #2

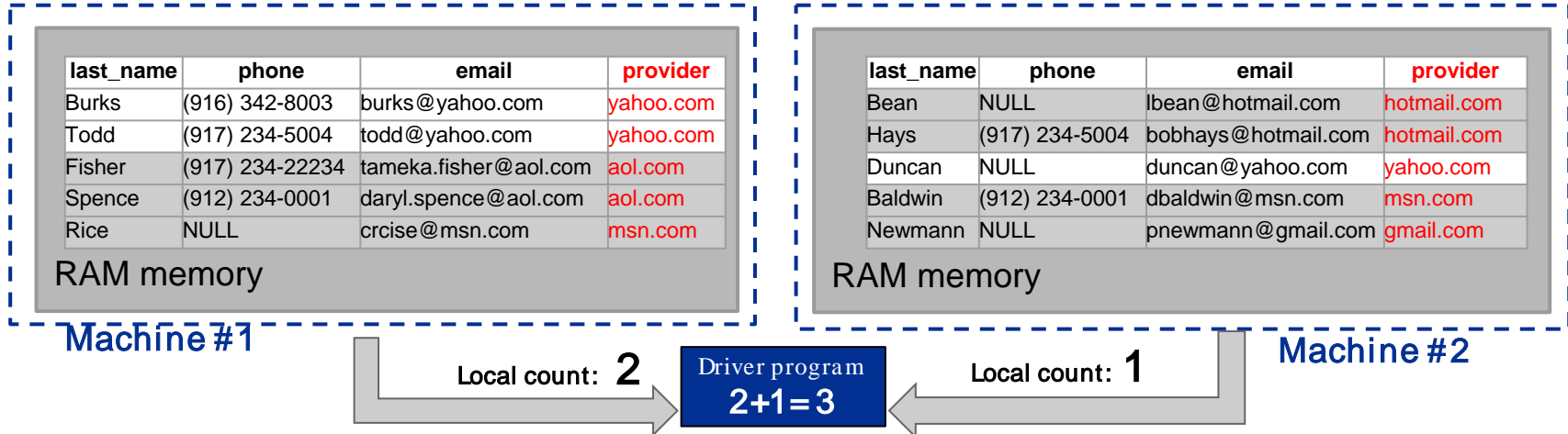


# Procesamiento distribuido en memoria principal

¿Cuántos clientes tienen email de Yahoo?

- Cada nodo carga una porción de datos de un CSV desde HDFS
- Trocean la cadena de la dirección de email por '@'
- Crean nueva columna con el proveedor (parte derecha de @)
- Filtran las filas que tienen 'yahoo.com' y cuentan

Cada nodo hace este procesamiento localmente, en memoria principal



## De nuevo: ¿por qué Spark?

- Si nuestros datos llegan en tiempo real...
  - Python y R no están preparados para datos que llegan en streaming
  - Solución: Spark!
- Si nuestro PC tiene 16 GB RAM pero el dataset tiene 100 GB ...
  - No se puede usar R o Python convencional para procesarlo
    - No podemos ni siquiera cargarlo en memoria
  - Solución: Spark en un cluster con varias máquinas
- Si nuestro PC tiene 16 GB RAM pero el dataset tiene 10 GB ...
  - ¿Puedo procesarlo? Probamos R o Python
  - Es probable que aun así, el ordenador se congele o tarde muchísimo
  - Solución: Spark en un cluster con varias máquinas



## 2 Arquitectura de Apache Spark



# Arquitectura de Spark

**Para utilizar Spark** simplemente importamos una serie de bibliotecas en nuestro programa Python

El programa en sí (**Jupyter notebook**) siempre se ejecuta sobre una sola máquina (por ejemplo nuestro portátil)...

- Python ejecuta el código, línea por línea
- ... pero cuando llamamos a una función de Spark, entonces en esa línea el código se ejecuta de manera distribuida en el cluster
- **Sólo se usa el cluster en esas líneas de código**



## Arquitectura de Spark

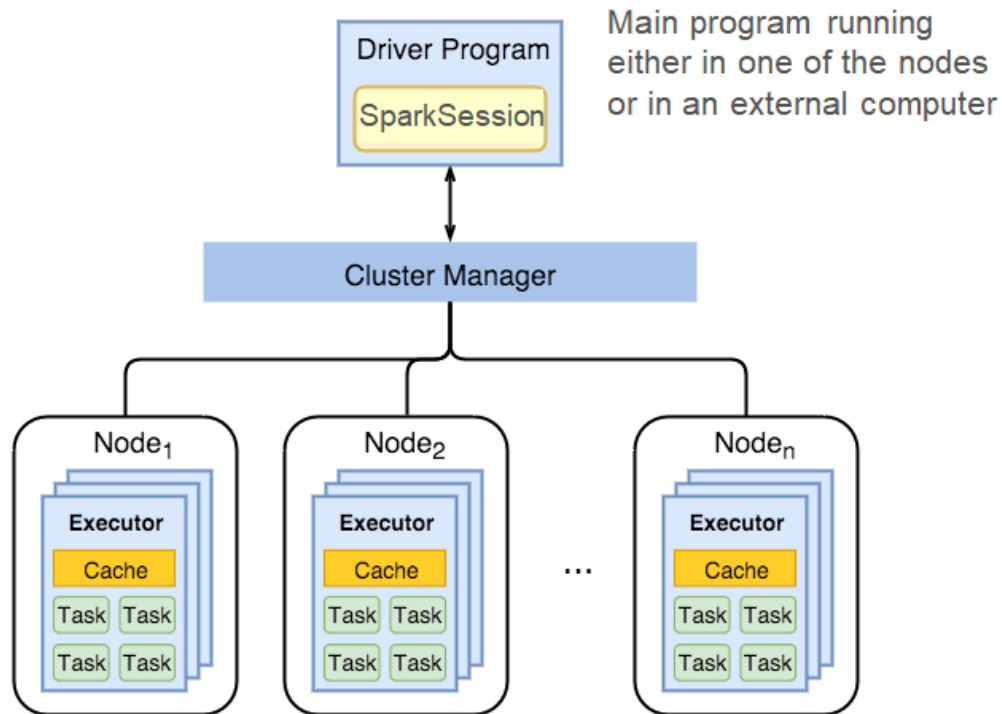
**El programa en Python en el que importamos Spark**, el cual se ejecuta en una sola máquina hasta llegar a alguna función de Spark, se denomina proceso **driver**.

- En nuestro caso, **el driver siempre es el Notebook de Jupyter**

Cada uno de los procesos que se ejecutan en los nodos del cluster (a.k.a. *workers*), a la espera de que el driver les envíe trabajo concreto, se llaman **executors**.



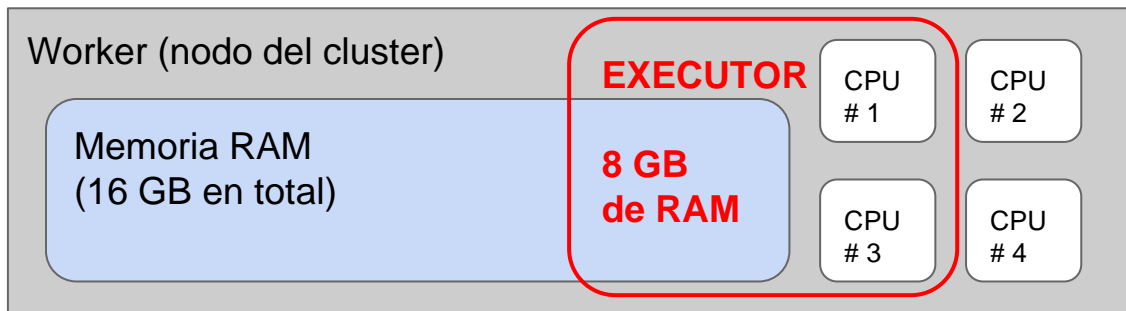
# Arquitectura de Spark



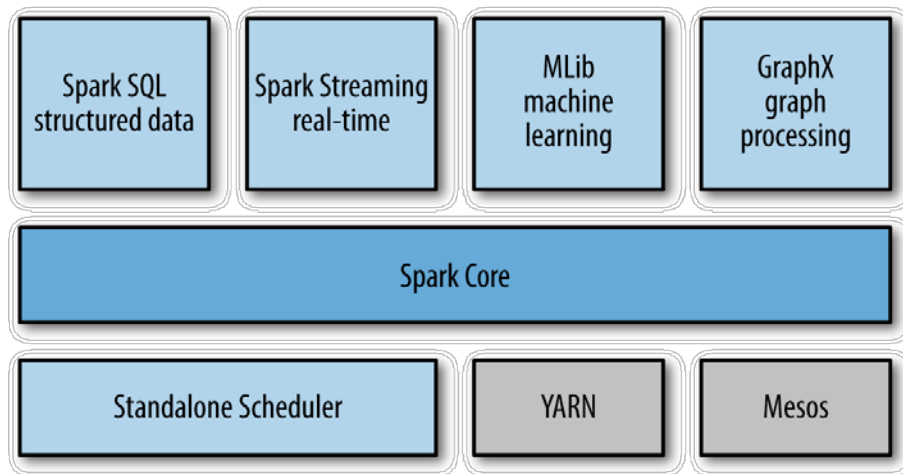
# Executors

**Proceso del sistema operativo** (de la *Java Virtual Machine*, JVM) creado por Spark en cada uno de los nodos del cluster (workers).

A cada ejecutor se le asignan unas pocas CPUs (procesadores) de ese worker, y cierta cantidad de memoria RAM. Esos recursos ya no podrán ser usados por otras aplicaciones que se lancen en ese worker.



# Componentes de Spark



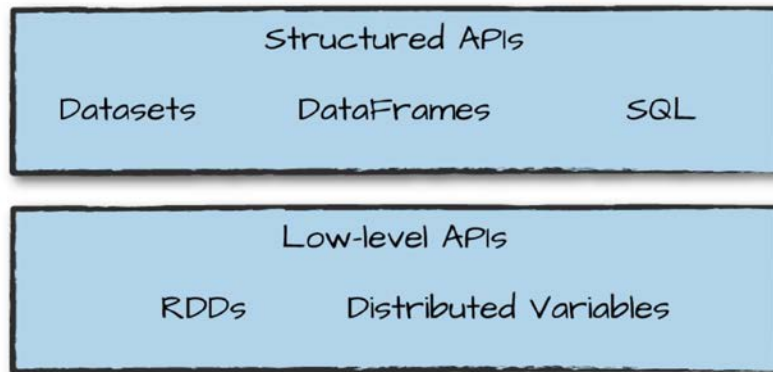
- ▶ Falta un nuevo módulo llamado *Structured Streaming* (introducido en Spark 2.0) que abstrae la mayoría de los detalles de Spark Streaming y simplifica su uso
- ▶ El *core* de Spark incluye los mecanismos de comunicación y gestión de la memoria, tareas, etc y la abstracción principal de Spark: los RDD
- ▶ Yarn y Mesos son dos gestores de un cluster, que permiten planificar tareas de Spark y de otras aplicaciones que comparten el cluster. Spark incluye su propio gestor de cluster, muy sencillo y que solo permite planificar tareas de Spark: *standalone*



# API's de bajo nivel y de alto nivel

Existen dos tipos de **APIs** para interactuar con Spark:

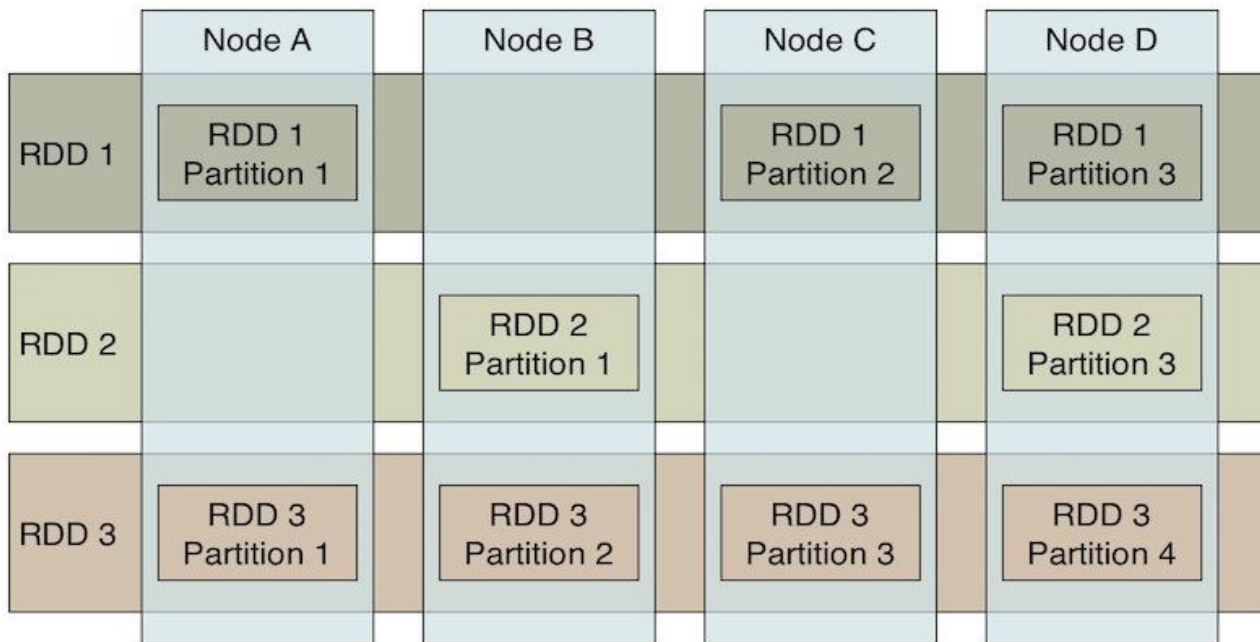
- **API no estructurada** o de bajo nivel (RDDs)
- **API estructurada** o de alto nivel (DataFrames)



# 3 RDDs (Resilient Distributed Datasets)

# RDD: Resilient Distributed Datasets

- **Abstracción fundamental de Spark.** RDD: colección **no ordenada** (bag) de objetos distribuida en memoria entre los nodos del cluster. La colección está dividida en *particiones*, y cada una está en la memoria RAM de un nodo distinto del clúster, **sin replicar**.





# RDD: Resilient Distributed Datasets

- ▶ **Abstracción fundamental de Spark.** RDD: colección **no ordenada** (bag) de objetos distribuida en memoria entre los nodos del cluster. La colección está dividida en *particiones*, y cada una está en la memoria RAM de un nodo distinto del clúster
  - ▶ **Resilient (resistente, adaptable):** es posible reconstruir un RDD que estaba en memoria a pesar de que una de las máquinas falle, gracias al DAG de ejecución
  - ▶ **Distributed (distribuido):** los objetos de la colección están divididos en *particiones* que están distribuidas (sin replicación) en la memoria principal de los nodos del cluster
  - ▶ **Datasets:** la colección representa un conjunto de datos que estamos procesando, transformando, agregando, etc.



# Los RDDs son poco prácticos

Los RDDs son una abstracción de bajo nivel.

La operación más habitual con ellos es *map* que fuerza al usuario a conocer perfectamente la estructura de cada uno de los objetos que están almacenando.

```
customersRDD = spark.sparkContext
    .textFile("/data/customers.csv")
    .map(lambda line: line.split(","))

# Ahora, customersRDD es un RDD de arrays. Cada array tiene
# estructura (nombre, edad, ciudad, país) y yo como programador
# tengo que tener en mente esa estructura exacta

customersRDD.map(lambda array: (array[0], array[3])) # me quedo sólo con españoles
    .filter(lambda array: array[1] == "Spain")
```

Pablo, 24, Barcelona, Spain  
Emma, 32, Alabama, USA  
Tony, 28, Frankfurt, Germany  
Jose, 35, Madrid, Spain

Otras operaciones frecuentes nos obligan a tener un RDD de pares (clave, valor), por ejemplo `reduceByKey`, `join`, ...



# DataFrames: tablas en memoria

Sería mucho mejor poder manipular los datos como si fuesen tablas de una base de datos (con filas y columnas con nombre y tipo)

RDD con 2 particiones

Part #1	[Pablo, 24, Barcelona, Spain] [Emma, 32, Alabama, USA]
Part #2	[Tony, 28, Frankfurt, Germany] [Jose, 35, Madrid, Spain]

DataFrame con 2 particiones

	Name	Age	City	Country
Part #1	Pablo	24	Barcelona	Spain
	Emma	32	Alabama	USA
Part #2	Tony	28	Frankfurt	Germany
	Jose	35	Madrid	Spain

```
customersRDD.map(lambda array:  
    (array[0], array[3]))  
    .filter(lambda array:  
        array[1] == "Spain")
```

```
espDF = customersDF.select("Name", "Country")  
                    .where("Country = 'Spain'")
```



# DataFrames: tablas en memoria

Internamente, un DataFrame no es más que un RDD de un objeto especial llamado *Row* (filas), aunque podemos obviar este detalle y **Manejarlos utilizando la API Estructurada**

Part #2   Part #1

Name	Age	City	Country
Pablo	24	Barcelona	Spain
Emma	32	Alabama	USA
Tony	28	Frankfurt	Germany
Jose	35	Madrid	Spain



Part #2   Part #1

## RDD de Rows

Row(Name=Pablo, Age=24, City=Barcelona, Country=Spain)

Row(Name=Emma, Age=32, City=Alabama, Country=USA)]

Row(Name=Tony, Age=28, City=Frankfurt, Country=Germany)

Row(Name=Jose, Age=35, City=Madrid, Country=Spain)

Documentación oficial del módulo Spark SQL: <https://spark.apache.org/docs/latest/sql-programming-guide.html>

Documentación oficial de la API de pyspark: <https://spark.apache.org/docs/latest/api/python/index.html>



# DataFrames y Spark SQL

- **DataFrame:** tabla de datos distribuida, estructurada en filas y columnas. Cada columna tiene un **nombre** y un **tipo de dato** (entero, real, string, etc, o un tipo definido por el usuario). Se suelen crear leyendo de alguna fuente de datos.
- **Spark SQL:** módulo que define operaciones para manipular DF
  - **API estructurada:** funciones que, al encadenarse, van transformando un DataFrame en otro
  - **SQL:** Spark tiene un analizador SQL que transforma consultas en lenguaje SQL puro a operaciones distribuidas sobre un DataFrame, de manera transparente

```
customers.where("Age > 18").groupBy("Country").count()
```

```
spark.sql("SELECT Country, count(*) FROM customers WHERE Age > 18  
GROUP BY Country")
```



# DataFrames y Spark SQL

- Internamente, Spark siempre ejecuta operaciones sobre RDDs, puesto que es lo único que el motor de cálculo sabe realizar. Cualquier operación es traducida a RDDs.
- Por ese motivo, un DataFrame en realidad envuelve un RDD, ya que es un **RDD de tipo especial de objeto llamado Rows**.
- La API estructurada proporciona operaciones **distribuidas** (transformaciones **lazy**, perezosas) y listas para usar sobre columnas:
  - Seleccionar columnas; filtrar filas según el valor de una o varias columnas
  - Crear una nueva columna con operaciones aritméticas sobre columnas
  - Calcular log/raíz/seno/coseno de una columna existente
  - Crear una columna como una discretización de otra existente
  - Renombrar o eliminar columnas
  - Cambiar el tipo de dato asociado a una columna
  - Funciones de agregación (media, desv típica, min, max...) sobre una columna, o sobre las filas pertenecientes a un grupo definido por una columna categórica



# Operaciones con RDDs y DataFrames

En un RDD se pueden llevar a cabo dos tipos de operaciones

- **Transformaciones**: devuelven un **RDD completamente nuevo** que es el resultado de transformar el RDD original de alguna manera  
*El RDD original **no se modifica** (es **immutable**!)*
- **Acciones**: calculan un resultado (generalmente un objeto simple del lenguaje de programación, como un número, una lista, etc) y **lo envían al proceso driver**  
*El objeto tiene que caber en la memoria de la máquina en la que se está ejecutando el proceso driver.*

Los DataFrames son un tipo especial de RDD, por lo que esto aplica también



# Diferencia fundamental



Una **transformación** es **lazy** (perezosa) y no se ejecuta en ese momento. Spark solo *toma nota* de que hay que hacerla, añadiéndola al grafo de ejecución o **DAG**.

*Ejemplos: withColumn, where, distinct, sort, join, groupBy + agg*

El DAG almacena el **linaje del dato**: qué RDD deriva de otro (como un **árbol genealógico**).

*Calcular un DataFrame aplicando las operaciones de transformación al DF original se denomina **materializar** ese DataFrame en memoria*

Para ejecutar la transformación, múltiples executors aplican la misma transformación a diferentes particiones del RDD original al mismo tiempo

*Cada executor utiliza las CPUs de esa máquina y los datos presentes en ese executor*





# Diferencia fundamental



Una **acción** es una operación que **devuelve un resultado al driver**, sea un número, un array o cualquier otro objeto

*Ejemplos: show, take, collect, write, count*

Una acción **se ejecuta inmediatamente** y provoca la materialización del RDD sobre el cual hemos invocado la acción.

- El propósito de materializar un RDD es aplicar la acción a él, o bien transformarlo en otro RDD para obtener el siguiente descendiente en el linaje.
- Inmediatamente después de que un RDD haya cumplido su propósito, la memoria RAM dedicada a él es liberada.

Para materializar un RDD, es posible que sea necesario llevar a cabo varias transformaciones previas, materializando también sus **ancestros**.



# 4 Transformaciones más frecuentes en DataFrames

## Transformación *select*

- Recibe los nombres de las columnas que queremos seleccionar y devuelve otro DF que solo contiene esas columnas
  - Todas las columnas deben existir!

```
from pyspark.sql import functions as F
transformedDF = flightsDF.select("Year", "Month", "DayofMonth",
                                "ArrTime", "FlightNum")
```

- También podemos usarlo para **crear nuevas columnas al vuelo**:

```
resultDF = flightsDF.select(
    F.col("Year"),
    F.col("Distance"),
    (1.6 * F.col("Distance")).alias("DistKm")
)
```

```
flightsDF.selectExpr("Year", "Distance",
                     "1.6 * Distance as DistKm")
```

Year	Distance	DistKm
2008	2400	3840
2008	3200	5120
2008	1500	2400
2008	4500	7200



## Transformación *sample*

- Recibe el porcentaje de filas que queremos mantener y devuelve otro DF que tiene solo ese porcentaje de forma aleatoria (el porcentaje **no es exacto**).

```
randomSampleDF = flightsDF.sample(0.1) # 10 % de las filas originales
```

- Trabajando con un DF necesito unas pocas filas que reflejen el comportamiento general: muestra aleatoria
  - **Representaciones gráficas** con Python (no Spark) con paquetes de visualización como Seaborn, Matplotlib
  - El DF de Spark completo no cabría en memoria



# Transformación *sample*

- Recibe el porcentaje de filas que queremos mantener y devuelve otro DF que tiene solo ese porcentaje de forma aleatoria (el porcentaje **no es exacto**).

```
randomSampleDF = flightsDF.sample(0.1) # 10 percent of the rows
```

flightsDF (10,000 rows)

Year	Distance	...
2008	2400	...
2008	3200	...
2008	1500	...
2008	4500	....
...	...	...
...	...	....



randomSampleDF  
(aprox. 1,000 filas elegidas al azar)

Year	Distance	...
2008	2400	...
...	...	...
2008	4500	....
...	...	....



## Transformación *filter* / *where*

- Devuelve un nuevo DF que contiene solo las filas que cumplen cierta condición booleana sobre una o más columnas
  - *filter*() y *where*() son exactamente equivalentes
  - El argumento puede ser un **string** con una condición en SQL puro
  - O también una condición booleana utilizando la API de SparkSQL API sobre las columnas

```
transformedDF = flightsDF\  
  .filter("DayOfMonth < 20 or Origin = 'LAX'")\      # SQL puro  
  .filter("Carrier is not null")\                    # SQL puro  
  .filter(~(F.col("FlightNum").isNull()))\          # funciones de SparkSQL  
  .filter(1.6*F.col("Distance") > 5000)\            # funciones de SparkSQL  
  .filter(F.col("Delay") > F.col("DepDelay") + F.col("ArrDelay"))\  
  .where("Delay > 15")                              # SQL puro
```



## Transformación *withColumn*

- Crea una nueva columna de otras existentes. Devuelve un nuevo DF que contiene **todas las columnas existentes más la nueva por la derecha**

```
transformedDF = flightsDF.withColumn("DistKm", 1.6*F.col("Distance"))
```

- Si el nombre corresponde a una columna **ya existente**, estaremos **reemplazando** (*creándola de nuevo*) en el nuevo DF en su misma posición
- Se puede utilizar **cualquier operación entre columnas implementada en Spark SQL** en el paquete `pyspark.sql.functions` (todas ya **distribuidas!**)
  - Se suelen encadenar múltiples operaciones *withColumn* una tras otra.
  - Las nuevas columnas se van creando por la derecha en el mismo orden en el que vamos encadenando las operaciones.



# Transformación *withColumn*

```
from pyspark.sql import functions as F
```

```
transformedDF = flightsDF\  
    .withColumn("DistKm", 1.6 * F.col("Distance")) \ # crear nueva columna  
    .withColumn("TotalDelay", F.col("DepDelay") + F.col("ArrDelay")) \  
    .withColumn("Year", F.lit(2009)) # reemplazar columna existente
```

Year	...	...	DistKm	TotalDelay
2009	...	...	3840	26
2009	...	...	5120	17
2009	...	...	2400	32
2009	...	...	7200	7





## Usando *withColumn* con *when*

- Podemos crear una **nueva columna re-categorizando** otra existente
  - E.g.: descripción cualitativa de los retrasos
- Función ***withColumn*** combinada con la función ***when***, aplicada a una o varias columnas existentes
  - Definimos la *casuística* como operaciones booleanas sobre columnas
  - La nueva columna se añade al DF por la derecha

```
delayCategorizationDF = flightsDF.withColumn("DelaySeverity",  
  F.when(F.col("ArrDelay")<=0, F.col("ArrDelay"))\  
  .when((F.col("ArrDelay")>0) & (F.col("ArrDelay")<=15), "acceptable")\  
  .when((F.col("ArrDelay")>15) & (F.col("ArrDelay")<=30), "annoying")\  
  .when((F.col("ArrDelay")>30) & (F.col("ArrDelay")<=60), "impactful")\  
  .otherwise(F.col("ArrDelay"))
```



## Usando *withColumn* con *when*

```
delayCategorizationDF = flightsDF.withColumn("DelaySeverity",  
  when(col("ArrDelay")<=0, F.col("ArrDelay"))\  
  .when((col("ArrDelay")>0) & (col("ArrDelay")<=15), "acceptable")\  
  .when((col("ArrDelay")>15) & (col("ArrDelay")<=30), "annoying")\  
  .when((col("ArrDelay")>30) & (col("ArrDelay")<=60), "impactful")\  
  .otherwise(F.col("ArrDelay"))  
)
```

Year	...	ArrDelay	...	DelaySeverity
2008	...	14	...	"acceptable"
2008	...	-2	...	"-2"
2008	...	25	...	"annoying"
2008	...	115	...	"115"



# Transformaciones *withColumnRenamed* & *drop*

- Para renombrar columna existente: ***withColumnRenamed***. Devuelve un nuevo DF donde la columna ha sido renombrada
  - La columna debe existir
  - La columna renombrada ocupa la misma posición en el DF resultante
- Para eliminar una columna existente: ***drop***.
  - Si la columna no existe, no da ningún error!
  - Podemos eliminar varias columnas en una misma operación
- Podemos encadenarlas igual que cualquier otra transformación:

```
transformedDF = flightsDF\  
  .withColumnRenamed("Delay", "MinutesLate")\ # nuevo nombre: MinutesLate  
  .drop("Carrier", "FlightNum", "inventedName") # esto es correcto :-)
```



## Transformación *summary*

- Pedimos a Spark que **calcule un resumen estadístico** de columnas numéricas
- Media, desv. típica, min, max y cuartiles (Q1, Q2, Q3)

```
summaryDF = flightsDF.summary()
```

## Transformaciones *distinct* y *dropDuplicates*

- Elimina filas duplicadas (duplicadas = "cuando coinciden en todos los campos")

```
uniqueDF = flightsDF.distinct()
```

- Para contar número de valores distintos de una columna:

```
cuantos = flightsDF.select("Origin").distinct().count()
```

- Si queremos considerar que la fila está duplicada si coincide solo en ciertas columnas que le indicamos como argumento: *dropDuplicates*

```
oneRowPerRouteDF = flightsDF.dropDuplicates(["Origin", "Dest"])
```

```
# Basta que coincidan en Origin y Dest para considerarse duplicadas
```



## Transformación *sort / orderBy*

- Devuelve un DF **cuyas filas están ordenadas en base a una o más columnas** indicadas como argumento, donde la primera de ellas se usa para ordenar y el resto, para terminar de ordenar cuando hay empates en las anteriores.
- Por defecto ordena ascendente

```
sortedDF = flightsDF.orderBy("Origin", "flightDate")
```

```
sortedDF = flightsDF.orderBy("Origin", "flightDate",  
                             ascending = False)
```

```
sortedDF = flightsDF.orderBy(  
    F.col("Origin"),  
    F.col("flightDate").desc()  
)
```



# Funciones de agregación

- Pertenecen a `pyspark.sql.functions`
- Se suelen aplicar dentro de *agg* cuando hemos hecho *groupBy*, para calcular un valor por cada grupo, aunque también pueden aplicarse al DF completo para resumir una columna en un solo valor
- Las más frecuentes: `F.mean`, `F.max`, `F.min`, `F.stddev`, `F.count`, `F.countDistinct`...
- Ejemplo aplicándolas al DF completo: devuelve un DF de una sola fila

```
from pyspark.sql import functions as F
```

```
nDistintosDF = flightsDF.select(      # creamos las columnas al vuelo
    F.min("ArrDelay").alias("minDelayGlobal"),
    F.max("ArrDelay").alias("maxDelayGlobal"),
    F.countDistinct("Origin").alias("nDistintosOrigin"),
    F.stddev("DepDelay").alias("stddevDepDelay")
)
```



# Operaciones con columnas: *cast*

- Podemos **cambiar el tipo de dato** de una columna (existente o nueva)
  - El cambio es desde un **tipo de dato de Spark** a otro
- La función ***cast*** se aplica a un objeto **columna**, no al DF completo, y devuelve otro objeto columna.
- ¡No todas las conversiones están permitidas!
  - E.g.: StringType → IntegerType sólo si es posible parsear los strings
  - IntegerType → BoolType: 0 → False, otro número → True
  - Cualquier tipo → StringType: siempre permitido

```
from pyspark.sql.types import StringType, DoubleType
castedDF = flightsDF\
    .withColumn("DayOfWeek", F.col("DayOfWeek").cast(StringType()))\
    .select(col("DayOfWeek"),
            (3 * F.col("Distance").cast(DoubleType())).alias("Dist3"))
```

- **Primer caso:** reemplazamos una columna por su versión convertida
- **Segundo caso:** creamos una nueva columna Dist3 y la convertimos a doble, "al vuelo" !



# Operaciones con columnas: *alias*

- Podemos **renombrar un objeto columna** cuando lo estamos creando
  - Si simplemente necesitamos renombrar una columna que ya existe en un DataFrame, tenemos la transformación *withColumnRenamed* sobre el DF
- El método ***alias*** se aplica a un objeto **columna**, no al DF completo
- Lo más habitual es aplicarlo cuando estamos creando una columna, por ej:
  - Dentro de *select* creamos una columna al vuelo, y le damos nombre
  - Dentro de *agg* creamos columnas de agregaciones, y les damos nombres

```
castedDF = flightsDF.select("Origin",\
                             (1.6 * F.col("Distance")).alias("DistKm"))\
    .groupBy("Origin").agg(F.mean("DistKm").alias("AvgKm"))
```





# Transformaciones narrow y wide

Una transformación **narrow** (estrecha) no necesita datos de otros nodos para completarse: se ejecuta 100 % localmente en cada nodo, y devuelve otro DF con el mismo número de particiones.

Ejemplos: where/filter, UDF, withColumn (\*)

Una transformación **wide** (amplia) necesita datos de otros nodos y por tanto, provoca **movimiento de datos (shuffle)** entre nodos del cluster. Por eso suelen ser más lentas. Ejemplos: join, groupBy + agg, orderBy

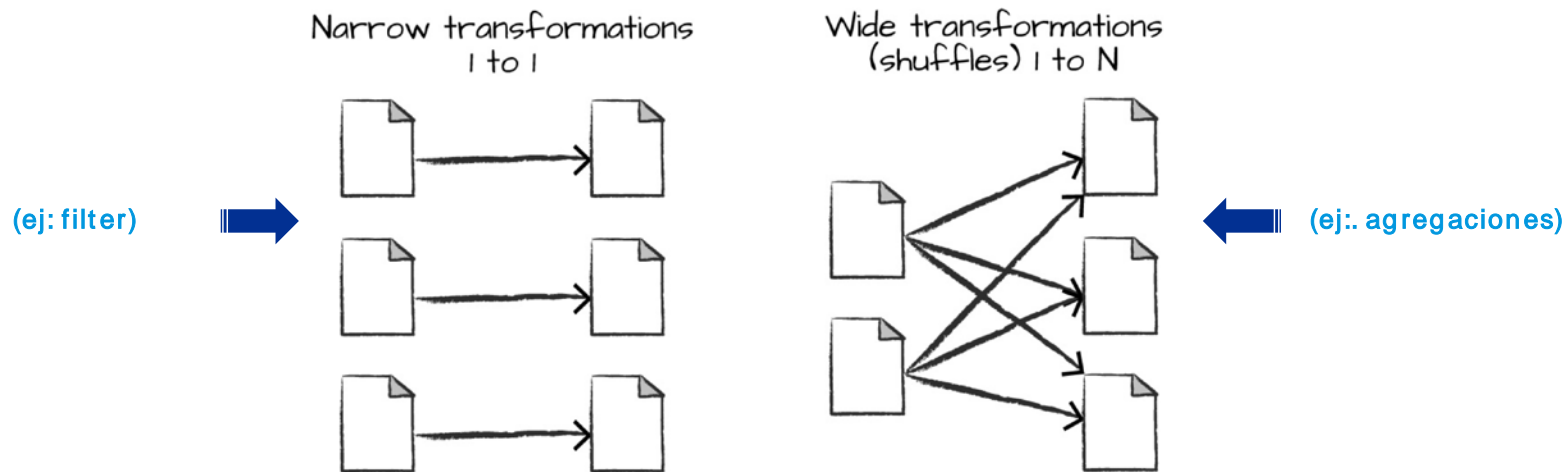
¿Cómo se lleva a cabo el **shuffle (movimiento de datos)** en Spark?

Cuando Spark necesita enviar datos a otro nodo...

- Primero escribe los datos en el disco duro del nodo local (el remitente)
- Los envía por la red desde el disco duro del remitente al disco duro del destinatario
- El nodo receptor los lee de disco duro y los carga en memoria principal



# Transformaciones narrow y wide



# Transformaciones en DataFrames

- Ya conocéis la mayoría de las **transformaciones** que podemos aplicar a un DataFrame (\*)
  - `select`, `where/filter`, `distinct`, `withColumn`, `groupBy` + `agg` or `count`, `join`, Window functions, string functions, UDFs, `map`
  - `randomSplit`, `union`, `intersection`
- Y también la mayoría de **acciones** disponibles (\*):
  - `write`, `textFile` (RDD), `count` sobre el DF completo
  - `show`, `take`, `collect`, `toPandas()`
- Hay operaciones que **no son ni transformaciones ni acciones**:
  - `persist(mode)`: marca el DF para evitar liberarlo tras 1ª materialización
  - `cache()`: atajo para `persist(MEMORY_AND_DISK)`
  - `unpersist()`: marca un DF persistido como no-persistido (liberable)
  - `printSchema()`
  - `columns`: devuelve los nombres de columnas como una lista de Python



## Executor #1

## Executor #2

Los DF no persistidos son expulsados (se libera la RAM que ocupan)

```
flightsDF = spark.read\
    .option(...)\
    .csv("flights.csv")
jfkDF = flightsDF.where(
    "Origin = 'JFK'")
n = jfkDF.count() # 13
```

**Proceso driver**

4+1+0+2+1  
+2+1+1+0+  
1= 13



Worker #1

Worker #2

Los DF persistidos NO son expulsados (no se libera la RAM)

```
flightsDF = spark.read\
    .option(...)\
    .csv("flights.csv")
jfkDF = flightsDF.where(
    "Origin = 'JFK'")
jfkDF.cache() # aún nada
n = jfkDF.count() # 13
```

Proceso  
driver

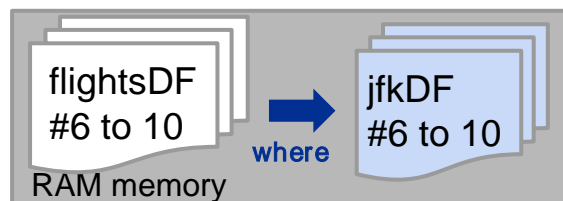
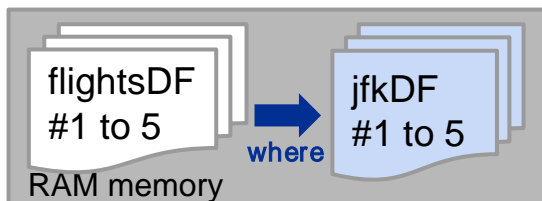
4+1+0+2+1  
+2+1+1+0+  
1= 13



STEP 0



STEP 1



STEP 2



STEP 3



# Persistiendo un RDD/DataFrame

- Un RDD materializado es expulsado de la memoria RAM tan pronto como deja de ser útil, para tener tanta memoria RAM libre como sea posible para futuras operaciones
- **Persistir** un DF significa *marcarlo como permanente* para que Spark no lo libere, y por tanto, no sea necesario recalcularlo cuando haya que volver a usarlo para algo
  - Cada *executor* mantiene las filas de las particiones que le pertenecen.
- Por defecto si no persistimos nada, el **linaje de cualquier RDD empieza en una operación de lectura** de un RDD en la **raíz del DAG** (primer ancestro)
  - Todas las operaciones intermedias se tendrán que ejecutar **una y otra vez**
  - Siempre que aplicamos una acción a un RDD, **Spark tiene que materializarlo, por tanto navega por el DAG hacia arriba hasta que encuentra un RDD persistido o si no, una operación de lectura, y empieza a materializar todos los RDD intermedios desde ese punto.**
- Un RDD puede ser persistido en **memoria**, en **disco**, o combinando **ambas**.
- La operación **cache()** para un DF es un atajo para **persist(MEMORY\_AND\_DISK)**

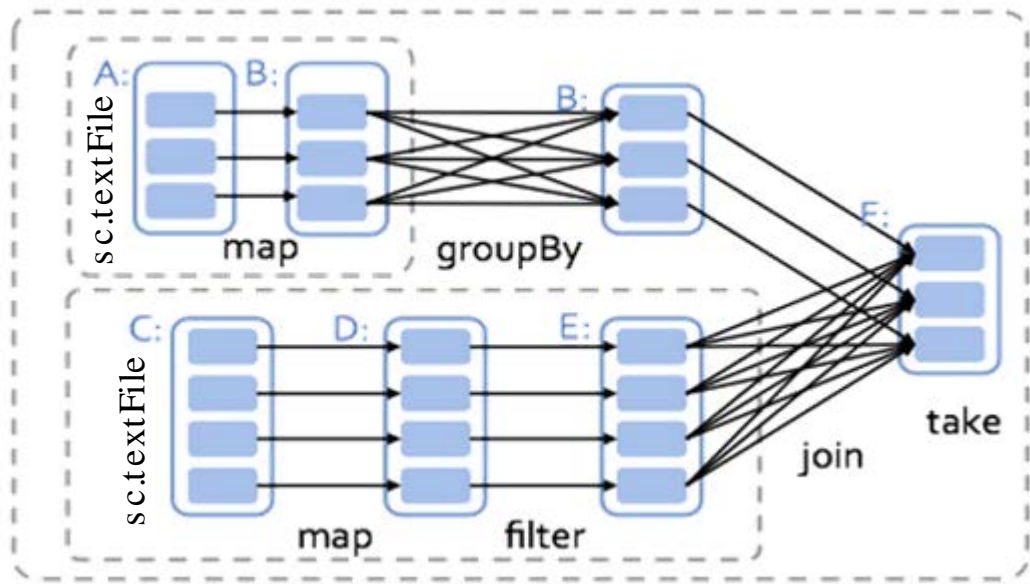


# Ejemplo

```
flightsDF = spark.read\  
    .option("header", "true")\ # no se ejecuta nada,  
    .option("header", "true")\ # pero se añade al DAG  
    .csv("flights_small.csv")  
  
# Transformación: todavía no se ejecuta nada (ni siquiera lectura)  
delayedDF = flightsDF.where("ArrDelay > 15")  
  
# Otra transformación: todavía no se ejecuta nada  
delayedKm = delayedDF.withColumn("DistKM", 1.6 * F.col("Dist"))  
  
delayedKm.show(20) # Esto es una acción: Spark materializa solo  
    # 20 elementos de delayedKm para mostrarlos. Esto requiere  
    # materializar PARCIALMENTE los DF delayedDF y delayedKm  
  
delayedKm.write.csv("delayed.csv") # Otra acción: Spark materializa  
    # el DF delayedKm completo para poder escribirlo. IMPORTANTE: las  
    # operaciones read, where() and withColumn() se ejecutan otra vez!
```



## Ejemplo de DAG con RDDs



```
A=sc.textFile("file.txt")
B = A.map(lambda x:...)
B = B.groupByKey()
C=sc.textFile("file2.txt")

D = C.map(lambda x: ...)
E = D.filter(lambda x: ...)
F = B.join(E)
resultArray = F.take(2)
```

El DAG de Spark permite la **resiliencia** de RDDs: si uno o más workers fallan mientras tienen particiones en memoria, Spark puede recalcular esas particiones en otros execturos **examinando su linaje**





# 5 Transformaciones avanzadas



## Eliminar filas con valores faltantes

**myDF.dropna()**: transformación que devuelve un nuevo DF **sin algunas filas** que han sido eliminadas por tener NA o null

# elimina una fila si se encuentra algún **NA** o **null** en cualquiera de las columnas  
cleanDF = flightsDF.dropna() # equivale a dropna("any") por defecto

# quita una fila solamente si todos los valores son NA  
cleanDF = flightsDF.dropna("all")

# quita una fila si hay algún NA en Origin o bien en Dest  
cleanDF = flightsDF.dropna(["Origin", "Dest"])

# quita una fila si **tanto** Origin **como** Dest son **ambos** NA al mismo tiempo  
cleanDF = flightsDF.dropna("all", subset=["Origin", "Dest"])



# Rellenar (imputar) los NA

**myDF.fillna():** transformación que devuelve un nuevo DF con **el mismo número de filas** pero donde los NA han sido **reemplazados por otro valor**

# reemplazamos los NA o valores nulos por 0

```
cleanDF = flightsDF.fillna(0)
```

# reemplazamos NA o valores nulos encontrados en Origin o Dest por 0

```
cleanDF = flightsDF.fillna(0, ["Origin", "Dest"])
```

# reemplazamos NA o nulos en Origin por "JFK", y en Dest por "LAX"

# El argumento es un **diccionario de Python** de (clave, valor) de strings

```
cleanDF = flightsDF.fillna({"Origin": "JFK", "Dest": "LAX"})
```

# Lo mismo con valores enteros

```
cleanDF = flightsDF.fillna({"ArrDelay": 0, "Distance": 1000})
```



# Agregaciones por grupos

- **Transformación** `groupBy(col1, col2...)` seguida de funciones de agregación
  - Consejo: usar *alias()* para dar nombre a las columnas creadas. Si no, Spark les asignará nombres muy poco prácticos por defecto
  - **Atención:** el resultado contendrá solamente las columnas que definen los grupos y las columnas de agregación que hayamos creado. Ninguna columna más.
  - **Las agregaciones** son aplicables a **columnas numéricas**, excepto **count**

```
import pyspark.sql.functions as F
summariesDF = flightsDF.groupBy("Origin", "Dest").agg(
    F.max("Distance").alias("maxDistance"),
    F.mean("ArrDelay").alias("meanArrDelay"),
    F.mean("DepDelay").alias("meanDepDelay"),
    F.stddev("ArrDelay").alias("stddevArrDelay"),
    F.count("*").alias("groupSize"),
    F.countDistinct("Dest").alias("distinctDestinations")
)
```



# Agregaciones por grupos: pivot

- Dividimos en dos partes las columnas que definen los grupos:
  - Una parte las ponemos dentro de `groupBy()`, y otra parte (habitualmente una sola, aunque pueden ser varias) las pasamos como argumento a `pivot()`
  - Los grupos definidos dentro de `pivot` se despliegan como columnas. Si es una sola variable, el resultado tendrá tantas columnas como categorías tenga la variable, multiplicado por el número de operaciones de agregación que haya.

```
import pyspark.sql.functions as F
summariesDF = flightsDF
    .groupBy("Origin", "Dest")\
    .pivot("DayOfWeek").agg(
        F.max("ArrDelay").alias("maxArrDelay"),
        F.mean("ArrDelay").alias("meanArrDelay")
    )

# DayOfWeek tiene 7 categorías, y estamos
# haciendo 2 agregaciones (max y mean) así que
# el resultado tiene 14 columnas + Origin, Dest
```

```
root
|-- Origin: string (nullable = true)
|-- Dest: string (nullable = true)
|-- 1_meanArrDelay: double (nullable = true)
|-- 1_maxArrDelay: string (nullable = true)
|-- 2_meanArrDelay: double (nullable = true)
|-- 2_maxArrDelay: string (nullable = true)
|-- 3_meanArrDelay: double (nullable = true)
|-- 3_maxArrDelay: string (nullable = true)
|-- 4_meanArrDelay: double (nullable = true)
|-- 4_maxArrDelay: string (nullable = true)
|-- 5_meanArrDelay: double (nullable = true)
|-- 5_maxArrDelay: string (nullable = true)
|-- 6_meanArrDelay: double (nullable = true)
|-- 6_maxArrDelay: string (nullable = true)
|-- 7_meanArrDelay: double (nullable = true)
|-- 7_maxArrDelay: string (nullable = true)
```



## Agregaciones por grupos: pivot

```
from pyspark.sql import functions as F
flightsDF.groupBy("Origin", "Dest").pivot("DayOfWeek").agg(
    F.max("DepDelay").alias("maxDepDelay")
).sort("Origin", "Dest").show()
```

Origin	Dest	1	2	3	4	5	6	7
ABE	ATL	8.00	9.00	9.00	8.00	98.00	88.00	92.00
ABE	CLT	70.00	42.00	6.00	8.00	9.00	7.00	84.00
ABE	DTW	94.00	8.00	6.00	74.00	7.00	134.00	8.00
ABE	FLL	null	null	89.00	null	null	81.00	null
ABE	MDT	null	null	null	null	175.00	null	null
ABE	MYR	4.00	-5.00	null	null	85.00	null	null
ABE	ORD	9.00	92.00	92.00	87.00	91.00	90.00	9.00
ABE	PGD	6.00	null	55.00	null	62.00	null	null
ABE	PHL	8.00	48.00	91.00	6.00	4.00	60.00	66.00
ABE	PIE	null	75.00	null	6.00	null	5.00	40.00
ABE	SFB	8.00	9.00	98.00	7.00	93.00	77.00	60.00
ABI	DFW	8.00	9.00	8.00	91.00	84.00	99.00	97.00



## Agregaciones por grupos con ventanas

- Similares a una agregación por grupos, pero se usan para añadir la agregación en una nueva columna del DF original.
- **No modifican el número de filas del DF original** sino que calculan la agregación en cada grupo y la añaden en las filas que pertenecen a ese grupo.
- Suele utilizarse dentro de *withColumn*. Los grupos se definen en el objeto *Window* que hay que crear y que se utiliza para agregar.

```
from pyspark.sql import functions as F
from pyspark.sql import Window
```

```
w = Window().partitionBy("Origin")
flightsExtraInfoDF = flightsDF\
    .withColumn("MeanDelay",
        F.mean("ArrDelay").over(w))
```

ID	Origin	Dest	...	Delay	MeanDelay
2157	LAX	ORC	...	16	13.45
8432	JFK	SFO	...	6	8.32
4532	LAX	ORC	...	12	13.45
9563	JFK	ORC	...	14	8.32



# Transformación join

- Objetivo: juntar dos DataFrames que están semánticamente relacionados por una o más columnas.
- Tenemos que especificar la condición booleana que hace que dos filas (una de cada DF) están relacionadas. Si se cumple la condición, se juntan las filas.
- Hay que especificar el tipo de JOIN: `inner`, `cross`, `outer`, `full`, `full_outer`, `left`, `left_outer`, `right`, `right_outer`, `left_semi`, `left_anti`

```
joinedDF = tableA.join(tableB, on = ["f"], how = "left_outer")
```

a1	a2	f
a	1	010
b	4	011
r	10	012
f	13	013

b1	b2	f
blue	3	023
red	3	012
orange	3	009
yellow	3	013

"left\_  
outer"



a1	a2	f	b1	b2
a	1	010	NULL	NULL
b	4	011	NULL	NULL
r	10	012	red	3
f	13	013	yellow	3





# Transformación join

- El tipo de JOIN por defecto es **INNER**: el resultado solo contiene filas que cumplen la condición en ambos DF
- Si las columnas que se corresponden no tienen el mismo nombre:

```
joinedDF = A.join(B, A.f1 == B.f2) # inner by default
```

A			B		
a1	a2	f1	b1	b2	f2
a	1	010	blue	3	023
b	4	011	red	3	012
r	10	012	orange	3	009
f	13	013	yellow	3	013

"inner"

a1	a2	f1	b1	b2	f2
r	10	012	red	3	012
f	13	013	yellow	3	013




# Transformación join

- Otro tipo muy común de join es el **LEFT JOIN** ("left\_outer")
- Cuando el DF de la izquierda no tiene correspondencia con ningún registro del DF de la derecha, el registro aparecerá en el resultado, pero tendrá todas las columnas del DF de la derecha con valor **null**.

```
joinedDF = A.join(B, on = A.f1 == B.f2, how = "left_outer")
```

A			B		
a1	a2	f1	b1	b2	f2
a	1	010	blue	3	023
b	4	011	red	3	012
r	10	012	orange	3	009
f	13	013	yellow	3	013

"left\_outer"



a1	a2	f1	b1	b2	f2
a	1	010	null	null	null
b	4	011	null	null	null
r	10	012	red	3	012
f	13	013	yellow	3	013



# User-Defined Functions (UDFs)

- Mecanismo para convertir una función convencional escrita en Python, R, Java o Scala en una función que puede ser llamada por los executors sobre cada fila de una columna concreta de un DF de Spark.
- Primero escribimos la función en Python que procesa **un solo elemento**, y luego la envolvemos en un objeto UDF que permite invocarla sobre una columna completa.
- Spark automáticamente **envía el código de nuestra función a los executors** para que la invoquen una vez por cada fila. Análogo a la función *map* pero con DF.
- Cada executor invoca a nuestra función sobre las porciones (particiones) de datos presentes en el executor.



# User-Defined Functions (UDFs)

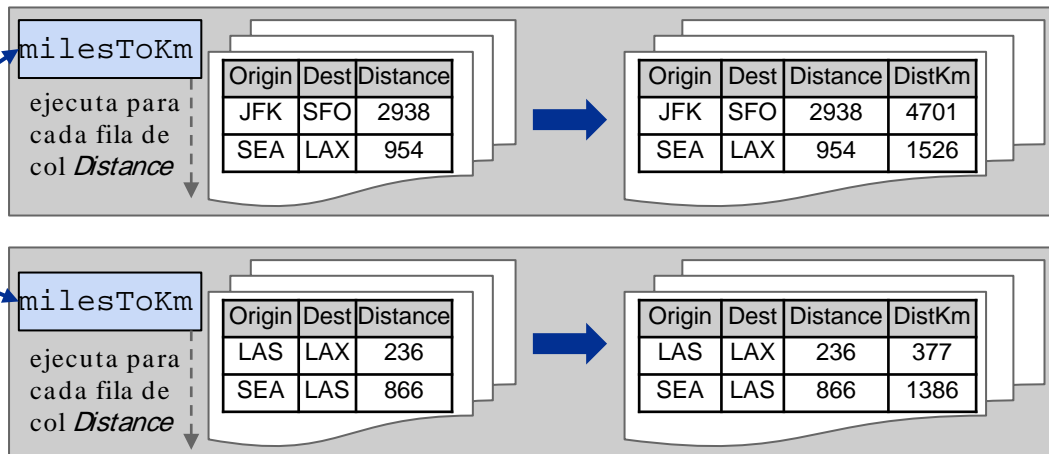
```
from pyspark.sql import functions as F
from pyspark.sql import types as T
def milesToKm(miles):
    return(1.6*miles) # multiplica un número por 1.6

toKmUDF = F.udf(milesToKm, T.DoubleType()) # envolvemos nuestra función

resultDF = flightsDF.withColumn("DistKm", toKmUDF(F.col("Distance")))
```

envía el código de la  
función a executors

milesToKm  
Programa  
driver



Executor #1  
Executor #2



# Vistas y consultas SQL puras

- Es posible crear una **vista temporal** de un DataFrame
  - Esto solamente genera metadatos en el catálogo de Hive
  - Existen solo mientras no finalice el driver (volátiles)
- La vista puede utilizarse como una tabla y **consultarse en lenguaje SQL puro**
  - También puede cruzarse con otras tablas persistentes de Hive
  - Para escribir una consulta SQL se usa el método **sql** de la SparkSession
  - La consulta es una **transformación** y su resultado es un DataFrame

```
flightsDF.createOrReplaceTempView("flights")  
resultDF = spark.sql("select max(ArrDelay)  
                      from flights group by Origin")
```

- Para salvarlo como tabla persistente de Hive, creando físicamente los datos:  
`flightsDF.write.saveAsTable("flightsTable")`



# 6 Pyarrow y PandasUDF



# Apache Arrow

**Tecnología para representar datos en memoria principal** de manera común a varios lenguajes diferentes, de forma que un dato puede ser creado en la RAM desde un lenguaje y leído e interpretado por otro, **sin etapa intermedia de traducción** o copia de datos de un lenguaje a otro

- Zero-copy
- Rapidez: representación columnar optimizada para procesadores SIMD (optimización de consultas analíticas en memoria, vectorización al máximo)
- Flexibilidad: permite comunicación entre numerosos lenguajes
- Estandarización: participan desarrolladores de los proyectos open source más utilizados



# Apache Arrow

	session_id	timestamp	source_ip
Row 1	1331246660	3/8/2012 2:44PM	99.155.155.225
Row 2	1331246351	3/8/2012 2:38PM	65.87.165.114
Row 3	1331244570	3/8/2012 2:09PM	71.10.106.181
Row 4	1331261196	3/8/2012 6:46PM	76.102.156.138

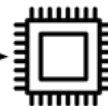
Traditional Memory Buffer

Row 1	1331246660
	3/8/2012 2:44PM
	99.155.155.225
Row 2	1331246351
	3/8/2012 2:38PM
	65.87.165.114
Row 3	1331244570
	3/8/2012 2:09PM
	71.10.106.181
Row 4	1331261196
	3/8/2012 6:46PM
	76.102.156.138

Arrow Memory Buffer

session_id	1331246660
	1331246351
	1331244570
	1331261196
timestamp	3/8/2012 2:44PM
	3/8/2012 2:38PM
	3/8/2012 2:09PM
	3/8/2012 6:46PM
source_ip	99.155.155.225
	65.87.165.114
	71.10.106.181
	76.102.156.138

```
SELECT * FROM clickstream  
WHERE session_id = 1331246351
```



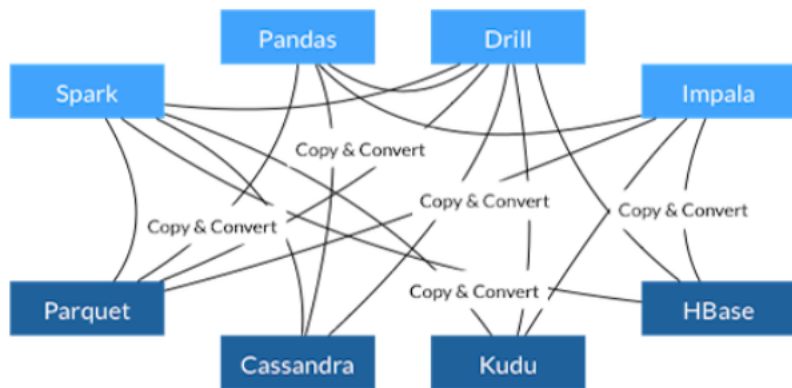
Intel CPU



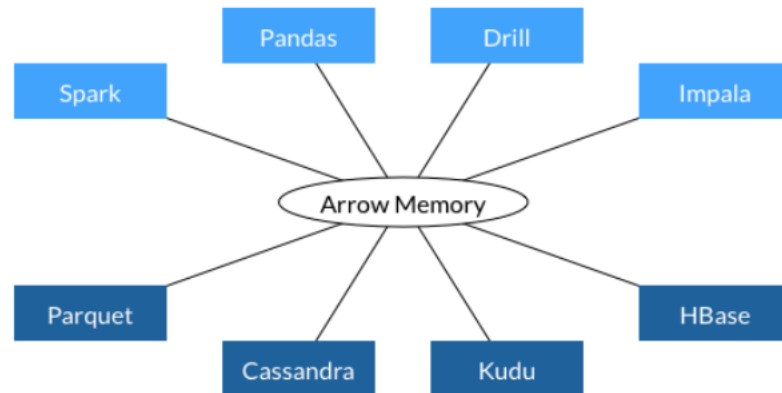


# Apache Arrow

## Advantages of a Common Data Layer



- Each system has its own internal memory format
- 70-80% computation wasted on serialization and deserialization
- Similar functionality implemented in multiple projects

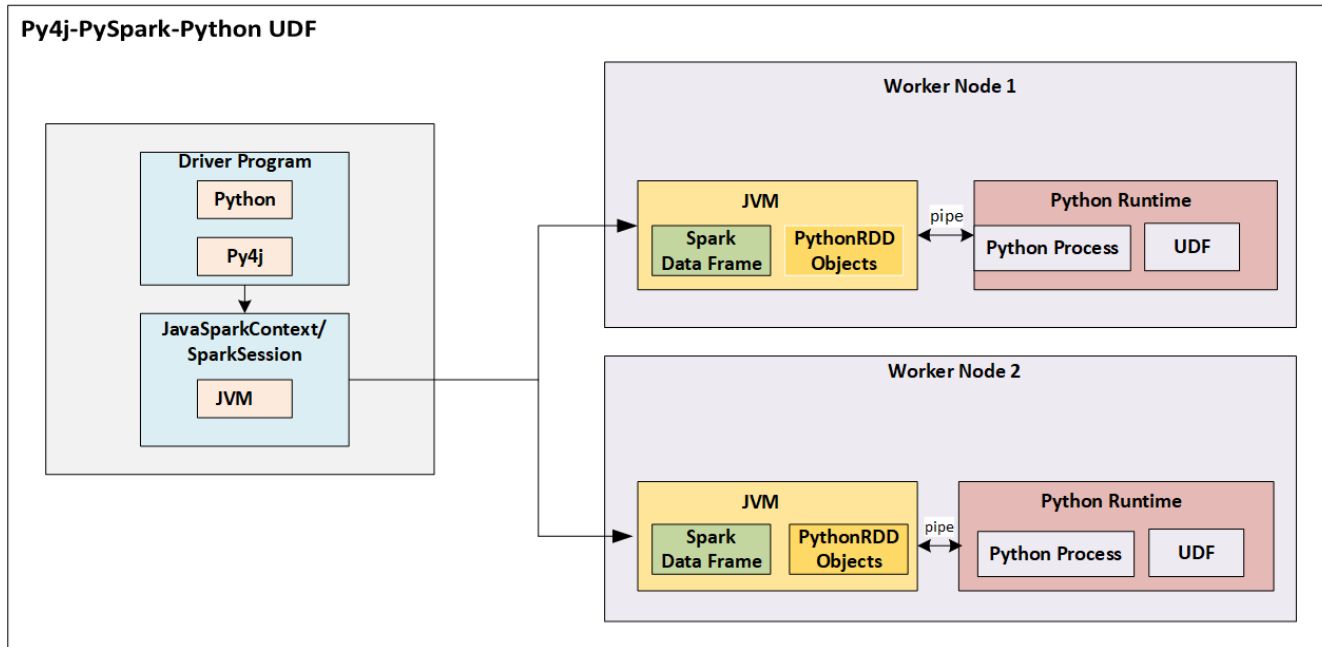


- All systems utilize the same memory format
- No overhead for cross-system communication
- Projects can share functionality (eg, Parquet-to-Arrow reader)



# Ejecución de una UDF estándar en pyspark

**Para ejecutar código Python en los executors:** se crea un proceso de la JVM y otro proceso del intérprete de python en cada executor



## PandasUDF (UDF vectorizadas)

**Objetivo:** eliminar la etapa intermedia (*pipe*) de copia y traducción de los datos entre la JVM (Java) y el intérprete de Python

- Mediante la biblioteca de Apache Arrow para Python, llamada **pyarrow**
- Idea básica: los datos (por trozos) de un DataFrame de Spark en la JVM pueden ser representados de una manera especial que puede ser leída sin traducción como un **DataFrame de la biblioteca Pandas** de python. Las columnas son objetos Pandas.Series
- Lo que se representa son trozos del DataFrame: columnas completas como pd.Series, o bien grupos completos (definidos por groupBy) como pd.DataFrame. **Cada serie o cada grupo se lleva completo a un executor**



# PandasUDF (UDF vectorizadas)

**Utilizan el paquete pyarrow:** tiene que estar instalado en todos los workers

- **PandasUDF escalares:** se utilizan en *withColumn* o *select* . Debemos escribir una función de Python asumiendo que sus argumentos son `pd.Series` y devolver otra `pd.Series` de la misma longitud. Spark convierte trozos de columnas en series y concatena las series devueltas.
- **PandasUDF por grupos:** se utilizan con *groupBy().apply()*. Debemos escribir una función de Python asumiendo que su **único** argumento es un `pandas.DataFrame` (dataframe de Pandas) con **todas las filas y todas las columnas de un grupo**, y debe devolver otro `pd.DataFrame`.
- **PandasUDF de agregación por grupos:** se utilizan con *groupBy().agg()*. Se invocan para cada grupo, y reciben una serie con todos los valores de una columna en ese grupo. Devuelven un solo valor resultante de agregar los valores de la serie.
- Cada grupo completo será llevado a un solo executor y allí será transformado a un DF de pandas local al executor: cada grupo tiene que caber en la memoria de un executor.



# PandasUDF escalares

```
import pandas as pd
from pyspark.sql.functions import col, pandas_udf
from pyspark.sql.types import LongType

# Creamos la función de python que multiplica dos series (dos objetos pd.Series) entre sí
def multiply_func(a, b):
    return a * b

multiply = pandas_udf(multiply_func, returnType = LongType())

# Probamos la función con datos de Pandas locales (cada columna de un df de Pandas es una serie)
x = pd.Series([1, 2, 3])
print(multiply_func(x, x))
# 0    1
# 1    4
# 2    9
# dtype: int64

# Creamos un DF de Spark (distribuido) a partir de un dataframe de Pandas
df = spark.createDataFrame(pd.DataFrame(x, columns=["x"]))

# Ejecutamos la función como si fuese una UDF convencional, aunque es una UDF vectorizada
df.select(multiply(F.col("x"), F.col("x")).alias("pairwiseProduct")).show()
```



# PandasUDF por grupos

```
from pyspark.sql.functions import pandas_udf, PandasUDFType

df = spark.createDataFrame( [(1, 1.0), (1, 2.0), (2, 3.0), (2, 5.0), (2, 10.0)], ("id", "v"))

# Anotación @pandas_udf para envolver nuestra función como UDF vectorizada. Tenemos que indicar la
# estructura del pandas.DataFrame devuelto por la función, bien en un string o en un StructType

@pandas_udf("id long, v double", PandasUDFType.GROUPED_MAP)
def subtract_mean(pdf):
    # pdf es un pandas.DataFrame (DataFrame de pandas, no distribuido)
    v = pdf.v
    return pdf.assign(v=v - v.mean())

df.groupby("id").apply(subtract_mean).show()
```

```
# +---+---+
# | id|  v|
# +---+---+
# | 1| 1.0|
# | 1| 2.0|
# | 2| 3.0|
# | 2| 5.0|
# | 2|10.0|
# +---+---+
```

```
# +---+---+
# | id|  v|
# +---+---+
# | 1|-0.5|
# | 1| 0.5|
# | 2|-3.0|
# | 2|-1.0|
# | 2| 4.0|
# +---+---+
```



# PandasUDF de agregaciones por grupos

```
from pyspark.sql import Window
```

```
df = spark.createDataFrame( [(1, 1.0), (1, 2.0), (2, 3.0), (2, 5.0), (2, 10.0)], ("id", "v"))
```

```
@pandas_udf("double", PandasUDFType.GROUPED_AGG)
```

```
def mean_udf(v):  
    return v.mean()
```

```
# Primer caso de uso de la agregación: en un groupBy().agg()
```

```
df.groupby("id").agg(mean_udf(df['v'])).show()
```

id	mean_udf(v)
1	1.5
2	6.0

```
# Segundo caso de uso de la agregación: dentro de una ventana
```

```
w = Window().partitionBy('id').rowsBetween(Window.unboundedPreceding, Window.unboundedFollowing)
```

```
df.withColumn('mean_v', mean_udf(df['v']).over(w)).show()
```

id	v	mean_v
1	1.0	1.5
1	2.0	1.5
2	3.0	6.0
2	5.0	6.0
2	10.0	6.0



# 7 Fuentes de datos en Spark





# Lectura y escritura de DataFrames

La **lectura y escritura de DataFrames** de todo tipo de fuentes de datos se realiza mediante una API uniforme: es otra de las **grandes fortalezas** de Spark, ya que existen conectores para casi cualquier fuente de datos

- Ficheros en Amazon S3, GoogleCS, HDFS como CSV, JSON, XML, formato libsvm, BBDD relacionales, no-SQL como MongoDB, Cassandra, Hbase, ElasticSearch, ...)

**DataFrameReader:** el objeto **spark.read** realiza las lecturas.

- Podemos configurar varias opciones mediante option("key", "value")
- Existe una opción genérica .format("formato") donde "formato" es "csv", "jdbc", "mongo", "parquet", "json" ... y termina con el método .load("/path/to/file")
- Los formatos frecuentes tienen un atajo para no usar load, ej: csv("...")
- Siempre devuelve un DataFrame, independientemente de la fuente de datos
- Podemos pasarle el esquema explícito en el momento de hacer la lectura, con la opción .schema(myschema)
- Para ficheros CSV podemos indicar qué hacer con las filas mal formadas



# Lectura y escritura de DataFrames

**Formato general:**

```
myDF = spark.read.format("format")\  
    .option("key", "value")\  
    .schema(mySchema)\  
    .load("/path/to/file")
```

```
spark.read \  
  .format("csv")\  
  .option("mode", "FAILFAST")\  
  .option("header", "true") \  
  .option("inferSchema", "true")\  
  .option("path", "flights_jan08.csv")\  
  .load()
```

```
DataFrame[Year: int, Month: int, DayofMonth: int, DayOfWeek: int, DepTime: string, CRSDepTime: int, ArrTime: string, CRSArrTime: int, UniqueCarrier: string, FlightNum: int, TailNum: string, ActualElapsedTime: string, CRSElapsedTime: int, AirTime: string, ArrDelay: string, DepDelay: string, Origin: string, Dest: string, Distance: int, TaxiIn: string, TaxiOut: string, Cancelled: int, CancellationCode: string, Diverted: int, CarrierDelay: string, WeatherDelay: string, NASDelay: string, SecurityDelay: string, LateAircraftDelay: string]
```



# Lectura y escritura de DataFrames

**Formato general:**

```
myDF = spark.read.format("format")\
    .option("key", "value")\
    .schema(mySchema)\
    .load("/path/to/file")
```

```
from pyspark.sql.types import StructType, StructField, StringType, IntegerType
flightsSchema = StructType([ # creamos un esquema
    StructField('Origin', StringType(), nullable = True),
    StructField('ArrDelay', IntegerType(), nullable = True),
    ...
])
flightsDF = spark.read.schema(flightsSchema)\
    .option("mode", "DROPMALFORMED")\
    .csv("/tmp/flights_jan08.csv")
```

Read mode	Description
permissive	Sets all fields to null when it encounters a corrupted record and places all corrupted records in a string column called <code>_corrupt_record</code>
dropMalformed	Drops the row that contains malformed records
failFast	Fails immediately upon encountering malformed records

# Otro ejemplo: lectura de una base de datos MongoDB

```
chinaPortsDF = spark.read.format("mongo")\
    .option("uri", "mongodb://localhost/myschema.mytable")\
    .load("/path/to/file")
```



# Lectura y escritura de DataFrames

**DataFrameWriter:** el objeto **myDF.write** realiza las escrituras.

- Opciones mediante `.option("key", "value")`, y opción `.format("formato")` donde casi todos los formatos frecuentes tienen ya un **atajo** para no tener que usarlo. Termina con el método **save("/path/to/file")** que, cuando es HDFS, **crea un nuevo directorio**.
- En ese directorio **se crean tantos ficheros como particiones tenga el DataFrame**.
- Opciones interesantes:
  - **mode**("modo") siendo "modo" : "overwrite" ó "append" ó "error" (por defecto), ó "ignore"
  - **partitionBy()** combinado con parquet, crea estructura de directorios en base a una variable categórica que acelera operaciones de filtrado cuando leamos de nuevo el fichero.
  - **saveAsTable()** en lugar de `save()` al final, para guardar el DF físicamente en el formato y ubicación donde esté configurado Hive (generalmente HDFS como Parquet), y añadir los metadatos correspondientes al metastore de Hive.



# Lectura y escritura de DataFrames

**DataFrameWriter:** el objeto **myDF.write** realiza las escrituras.

```
flightsWeekendDF.write.partitionBy("Origin")\  
    .format("parquet")\  
    .save("/path/new/folder")  
  
# los dos últimos se pueden sustituir por .parquet("/path/folder")  
# En el directorio se crean subcarpetas llamadas "Origin=JFK",  
# "Origin=LAX", etc y dentro, los datos de esa partición  
  
# Salvamos físicamente los datos en una tabla Hive en Parquet  
flightsWeekendDF.write.saveAsTable("flightsWeekends")
```



## Lectura/escritura de BBDD relacionales

- Spark puede leer de cualquier BBDD relacional, como MySQL, PostgreSQL, Oracle, SQL Server...mediante el protocolo habitual **JDBC**
- Spark hace predicate pushdown: una operación de filtrado de Spark puede traducirla automáticamente en un filtrado nativo del datasource. Hay reglas de cuándo será capaz y cuándo no. También con archivos Parquet!
- Los executors de Spark se conectan en paralelo a la base de datos para leer simultáneamente porciones distintas de la misma tabla al mismo tiempo.



# Federación de datos

- Al igual que se puede construir una base de datos federada como una base de datos virtual formada por BBDD heterogéneas, de forma transparente al usuario, es posible usar [Spark para leer fuentes diversas a DataFrames](#), y después manipularlos y cruzarlos con código Spark, sin importar su origen.
- Spark actúa como una capa de abstracción común y lleva a cabo las operaciones que no puedan ser traducidas a consultas nativas



# Federación de datos

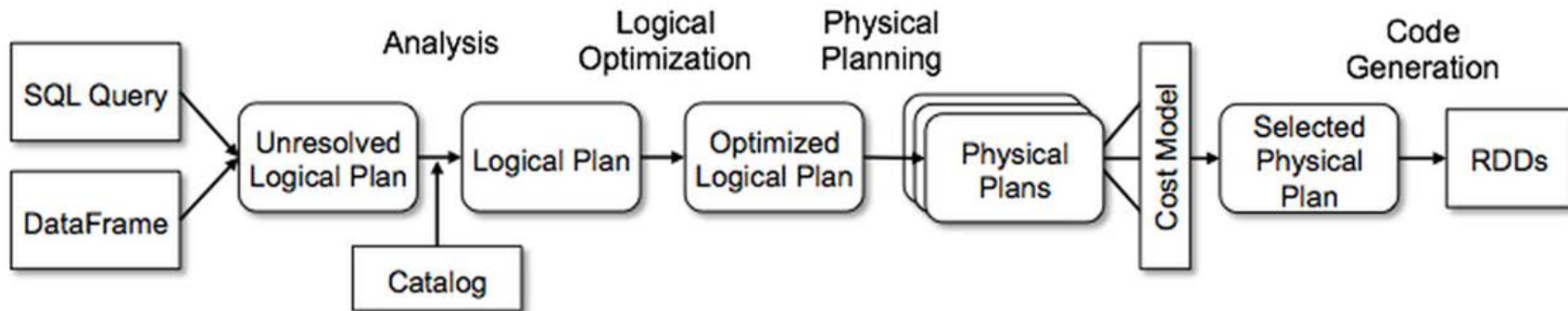




# 8 Fases de ejecución



# Fases de ejecución en Spark



```
scala> accountsDataset.select($"balance").filter($"balance" > 0).explain(true)
== Parsed Logical Plan ==
'Filter ('balance > 0)
+- Project [balance#210]
   +- LocalRelation [name#209, balance#210]

== Analyzed Logical Plan ==
balance: double
Filter (balance#210 > cast(0 as double))
+- Project [balance#210]
   +- LocalRelation [name#209, balance#210]
↓
== Optimized Logical Plan ==
Project [balance#210]
+- Filter (balance#210 > 0.0)
   +- InMemoryRelation [name#209, balance#210], true, 10000, StorageLevel(disk, memory, deserialized, 1 replicas)
      +- LocalTableScan [name#94, balance#95]
```

