1 – MapReduce

Clase n°02, a partir de segunda filmina. Paradigma, combiner, múltiples Jobs.

Es un framework para distribuir tareas en múltiples nodos de manera automática, pero también es un *paradigma de programación*. Los problemas se resuelven sin tener acceso a todos los datos. *Ejemplo: acumulado y cuántos en cada nodo para promedio.*

En la fase **map**, se procesa cada dato de entrada para *transformarlos* en un nuevo conjunto intermedio de datos. Luego, en la fase **reduce**, se reúnen resultados intermedios para finalmente *reducirlos* a un conjunto de datos resumidos. Entre las 2 fases existen *shuffle* y *sort*, las cuales no son obligatorias programar para la aplicación.

La unidad de trabajo es **Job**, dividido en una tarea map y una tarea reduce, y controlado por un daemon *JobTracker*, que reside en el nodo master. Los nodos restantes, llamados *TaskTracker*, ejecutan la tarea asignada por el JobTracker (map o reduce) y le reportan el progreso de la misma. *Aclaración: el JobTracker se limita a la gestión.*

1.1 – Entradas e instancias

En los archivos de texto plano (sin tabulaciones), cada línea es un dato a procesar, y la clave (key) es el *offset* de la línea dentro del archivo, no del directorio. Los archivos de entrada se dividen en **splits**, de modo que cada TaskTracker opere sobre un split.

Existen múltiples instancias de la tarea map (*mappers*), intentando que c/u trabaje sobre una copia local del dataset. Cada entrada es una tupla clave-valor, la cual debería ser procesada de manera independiente al resto, y la salida es una lista de 0 o más tuplas.

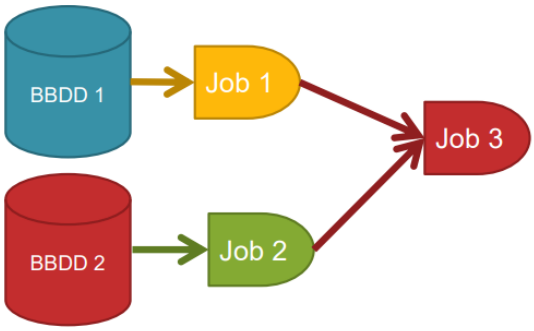
Luego, existe una instancia de la tarea reduce (*reducer*) por cada clave generada por la fase map, donde cada instancia recibe la clave y la lista de valores asociados. Estos valores solo pueden ser iterados mediante un bucle for, sin índices y vuelta atrás.

1.2 – Combiner

Los datos generados por la fase map se guardan en disco para luego ser cargados por un reducer. Dicho volumen de datos debe ser disminuido si la fase reduce tiene una gran carga de trabajo (muchos datos a leer). La *función combiner* se ejecuta a la salida de un mapper, **en el mismo nodo**, como un *reducer* que solo opera con dichos datos.

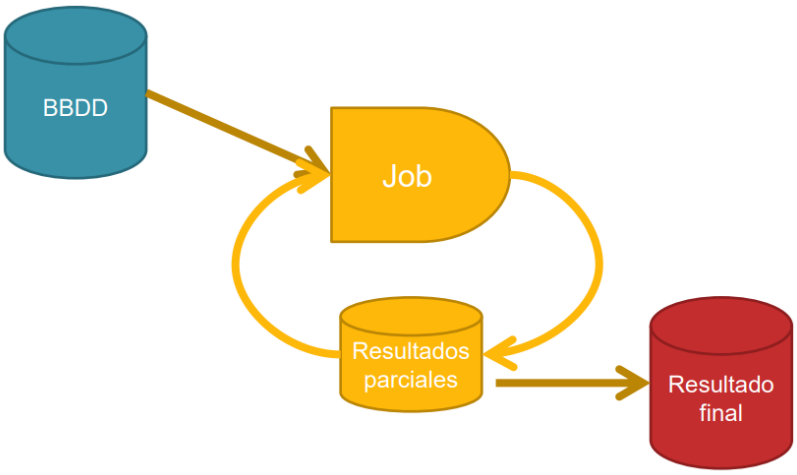
1.3 – Múltiples Jobs

A cada Job se le asigna una tarea map, reduce y combiner opcional; puede darse el caso que varios Jobs compartan una misma implementación map o reduce. El directorio de salida de un Job puede ser el de entrada de un próximo Job. Dichas dependencias se representan mediante un grafo acíclico dirigido (DAG).



El uso de +1 Job puede deberse a:

- ✓ Dependencia de datos: se calcula una variable que se usa en otra. *Ej: desvío.*
- ✓ Problemas iterativos: se trabaja sobre la salida de un mismo Job. *Ej: Jacobi.*
- ✓ Menor carga del reducer: se resuelve un problema usando procesamientos más simples, aumentando cantidad de reducers. *Ej: página más visitada por usuario.*



1.4 – Relación con SQL

- ✓ GROUP BY: en la fase map, se agrupa usando dicha columna como key.
- ✓ WHERE: en la fase map, se filtran las tuplas que cumplan la condición.
- ✓ SELECT: en map, dichas columnas son key; en reduce, se itera y escribe.
- ✓ SELECT DISTINCT: ídem anterior, pero en reduce, se escribe 1 vez.
- ✓ INNER JOIN: depende de si la fuente son varios maps y si es 1-1 o 1-N. Se puede usar una función map para cada tabla, agregando además un identificador de la tabla de procedencia para que luego en la fase reduce se respete un orden. En el caso 1-N, es necesario modificar las etapas shuffle y sort. *Nota: ver págs. 31-37.*

2 – Spark

Las acciones son aquellas funciones que se aplican sobre una RDD y provocan la ejecución del DAG (vinculación de transformaciones de RDDs) para su resolución.

- ☆ *Rdd.count():* devuelve la cantidad de tuplas que tiene la RDD.
- ☆ *Rdd.countByKey():* devuelve la cantidad de tuplas de cada clave de la RDD.
- ☆ *Rdd.collect():* devuelve todo el contenido, pudiendo ocupar mucha memoria.
- ☆ *Rdd.take(n):* devuelve n tuplas random de alguna partición (minimiza accesos).
- ☆ *Rdd.reduce(t1, t2 → to):* recibe una función para reducir tuplas de a pares, devolviendo otra tupla del mismo formato para seguir reduciendo hasta 1 sola.
- ☆ *Rdd.aggregate(vi, f, g):* reduce una RDD cambiando la estructura de la misma.
  - *vi: tupla de valores iniciales, que tiene formato de tupla resumida.*
  - *f: función del tipo (t1, v) → t2, donde "v" es original y "t" resumida.*
  - *g: función del tipo (t1, t2) → t3, es decir, reduce tuplas resumidas.*

Las transformaciones se ejecutan al realizar acción sobre la RDD. Para múltiples acciones, se persiste la RDD con *rdd.persist()* para evitar procesamiento repetido.

2.1 – Transformaciones

Son funciones que se aplican sobre RDD y devuelve otra RDD. Las del tipo "ByKey" operan sobre RDDs de 2 columnas (clave y valor). Se pueden dividir las transformaciones en **narrow** (ejecutada sobre partición) y **wide** (transfiere entre nodos).

Una wide provoca un cambio de "etapa" o *stage*. Mejora: particionado de RDDs.

- ❒ *Rdd.map(ti → to):* transforma el formato de cada tupla de la RDD. Es narrow.
- ❒ *Rdd.flatMap(ti → list(to)):* transforma una tupla en 1 o más tuplas. Es narrow.
- ❒ *Rdd.filter(t → boolean):* filtra tuplas que cumplen condición. Es narrow.
- ❒ *Rdd.reduceByKey(v1, v2 → vo):* se reducen los valores de cada clave de a pares.
- ❒ *Rdd.aggregateByKey(vi, f, g):* similar a aggregate, pero a valores de cada clave.
- ❒ *Rdd.groupByKey()*: reúne todos los valores de cada clave. Es wide (costosa).
- ❒ *Rdd.sortByKey(asc):* ordena las tuplas de la RDD por clave, sin hacer reducción.
- ❒ *Rdd.join(rdd2):* realiza inner-join con rdd2. Salida: key, (valores1, valores2)
- ❒ *Rdd.cogroup(rdd2):* combinación entre join y groupByKey entre RDDs.
- ❒ *Rdd.distinct():* devuelve la RDD sin tuplas repetidas (ídem dropDuplicates).

2.2 – Algoritmos iterativos

```
for i in range(5):
    data = data.map(fMap)
    data = data.filter(fFilter)
result = data.collect()
```

En este primer caso, hay una única acción al finalizar el loop, que ejecuta la cadena completa de transformaciones map y filter aplicadas sobre la RDD data.

```
for i in range(5):
    data2 = data.map(fMap)
    data2 = data2.filter(fFilter)
result = data2.collect()
```

En este segundo caso, "map" no se aplica sobre la RDD que resultó de la iteración anterior. Solo ejecuta la última.

```
while error > 0.001:
     data = data.map(fMap)
     data.persist()
     error = data.reduce(fReduce)
result = data.collect()
```

En este tercer caso, si no estuviese el data.persist(), se ejecutaría el DAG desde el inicio hasta el cálculo del error numerosas veces, y también todo hasta el collect().

```
while error > 0.001:
    data2 = data.map(fMap)
    error = data2.reduce(fReduce)
result = data2.collect()
```

Una alternativa a persistir el RDD es no encadenar las transformaciones, ya que hay una acción para cada iteración del loop.

2.3 – Spark SQL

Se trabaja con DataFrames, que consiste en RDDs con esquema (tiene tuplas de datos con nombres y tipo de datos de cada campo) dado por un objeto Row. Al tratarse de una abstracción, todas las transformaciones y acciones de RDDs son aplicables a DF.

Si se desea utilizar SQL puro, se debe aplicar *registerTempTable(nombreTabla)* sobre el DataFrame, de modo que el nombre de tabla se pueda utilizar en las consultas.

Las funciones Window permiten realizar resúmenes por grupo. Se puede utilizar al añadir una nueva columna como agregación, ranking, analítica (lead, lag), etc.

3 – Stream processing

Un dato se recibe, utiliza y descarta. Se usa ventana temporal con los últimos n.

Se deben cuidar 3 aspectos: velocidad, memoria y eficacia; se debe operar cada nuevo dato en el menor tiempo posible, ocupar poca RAM y clasificar con eficacia. Es decir, para cada nuevo dato se debe dar una respuesta antes de la próxima solicitud.

3.1 – Spark streaming

No es stream processing puro, ya que el stream es discreto (DStream) y se guarda en pequeños "chunks" (RDDs) para ser utilizados por pequeños procesos batch. Se especifica el intervalo de tiempo de cada chunk, no menor a 500 ms. Si se desea guardar un chunk antes de ser sobrescrito, se puede persistir mediante *checkpoint(nomBuffer)* y luego mediante la función *updateStateByKey* para obtener la "historia".

A diferencia de Spark para procesamiento batch (no streaming), el DAG solo se ejecuta cuando se da la orden *start()* al *StreamingContext* (las acciones no lo disparan), y dicha ejecución se vuelve a repetir indefinidamente hasta cancelar el Thread.

En base a lo anterior, no se pueden evaluar condiciones al no conocer el valor que arroja una acción hasta ejecución, imposibilitando el uso de toda estructura de control.

Anexo 1 – Parcial de MapReduce 1/1

La ejecución de la función map para cualquier tupla <k1,v1>...

a. ... permite escribir de cero a N tuplas intemedias <k2,v2>
b. ... solo permite escribir una tupla intemedia <k2,v2>
c. ... permite escribir de una a N tuplas intemedias <k2,v2>
d. ... permite escribir cero o a lo sumo una tupla intemedia <k2,v2>

La tarea map lee los datos en forma de pares <k1, v1>
y produce una lista de cero, uno o más pares <k2, v2>.

- <k1, v1> ➔ list(<k2, v2>)

- ¿Cuál es la tarea del combiner? *Es una función que permite minimizar la cantidad de datos que se deben escribir a la salida de la fase map, y que luego son cargados en la fase reduce. Para ello, se ejecuta en el mismo nodo donde se ejecutó el map, y opera con su salida producida para realizar la reducción de los datos.*

- ¿Cuál es la tarea de la etapa shuffle? *Es la etapa que se encarga de redistribuir los datos, producidos por la fase map, a partir de las claves, de modo que todas las tuplas asociadas a una misma clave sean procesadas por un mismo reducer.*

- ¿Cuál es la tarea de la etapa sort? *Es la etapa que se encarga de que un mismo reducer reciba las tuplas ordenadas por clave.*

Key	Value
4	2
5	2
1	4
3	4

¿Cuál es la salida al ejecutar el siguiente Job?

```
def map(k1, v1, context):
      context.write(v1, k1)


def reduce(k2, v2, context):
      n = 0
      for v in v2:
            n = n + v
      context.write(k2, n)
```

La fase map produce la siguiente salida:

- <2, 4>
- <2, 5>
- <4, 1>
- <4, 3>

Luego, se tienen 2 reducers, uno que trabaja con la clave 2 y otro con la clave 4. Su tarea es sumar los valores recibidos y producir la salida <k2, suma>.

Entonces, la salida del Job:

- <2, 9>
- <4, 4>

Anexo II – Parcial de Spark

a. Una misma partición puede almacenar datos de varias RDDs.
b. Una RDD se divide en particiones y cada partición se almacena en un nodo del cluster.
c. Cada nodo del cluster solo puede almacenar datos de una única RDD.
d. Una RDD solo puede estar formada por una única partición la cual se distribuye entre varios nodos del cluster.

b? Sí. Cada partición va a un nodo y pertenece a una única RDD.

a. Las acciones van armando un DAG el cual es ejecutado por una transformación.
b. Toda transformación y toda acción son ejecutadas inmediatamente por Spark Context.
c. Las transformaciones van armando un DAG el cual es ejecutado por una acción.
d. Las transformaciones se ejecutan solo en RDDs que hayan sido persistidas previamente.

c) Easy!

8) ¿Cuántas veces en total se ejecuta la transformación map en el siguiente script?

```
A = sc.textFile("File")
for i in range(2):
      A = A.map(fMap)
      print(A.reduce(fReduce))
resultado = A.collect()
```

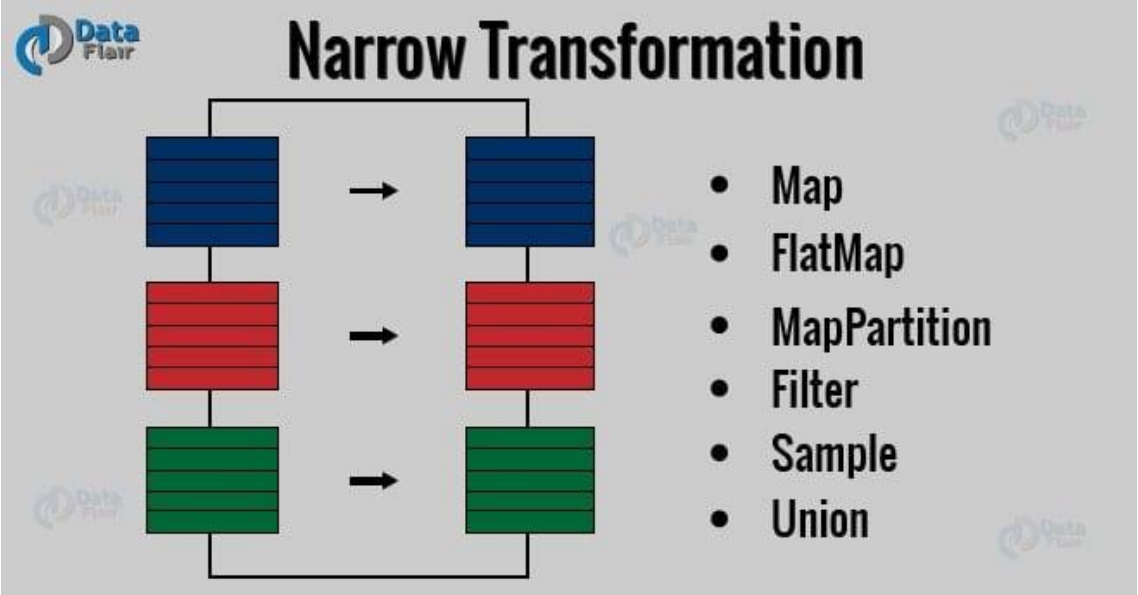
Las acciones son reduce() y collect(), y como no hay persistencia, se ejecuta 1 vez para la primera iteración, luego 2 veces para la segunda, y 3 veces para el collect(), así que en total son 6 ejecuciones de la transformación map.

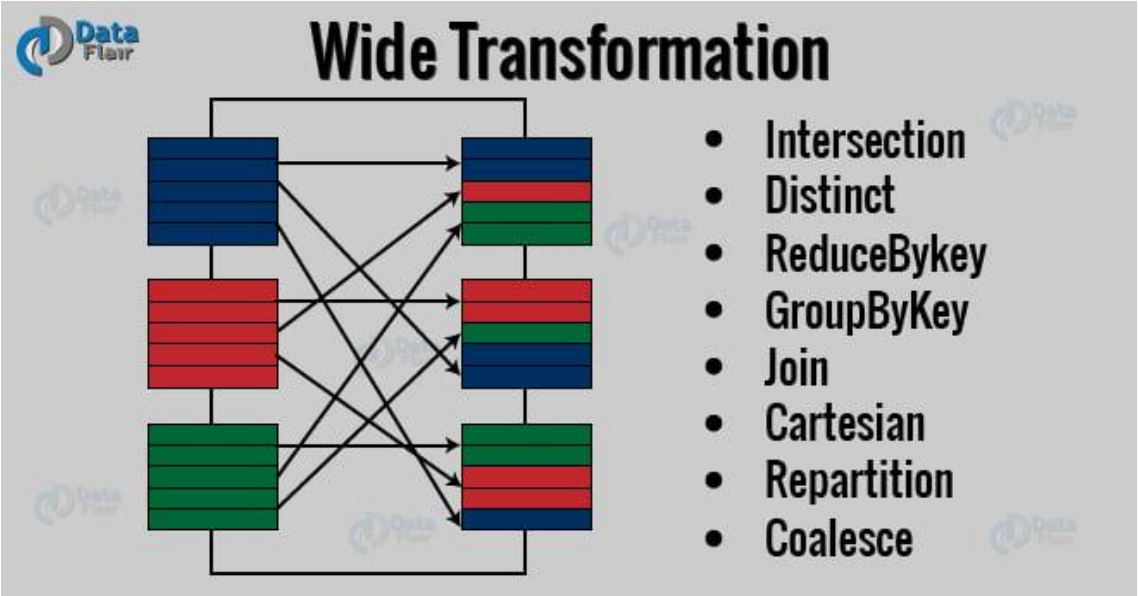
9) Dibuje el DAG que resulta de la ejecución del siguiente script etapas se ejecutará.

```
A = sc.textFile("Caso B1")
B = A.map(fMap1)
C = B.filter(fFilter1)
C = C.map(fmap2)
A = sc.textFile("Caso B2")
B = A.map(fmap3)
D = C.join(B)
D = D.filter(fFilter2)
final = D.reduce(fReduce1)
```

No veo trampas acá, se ejecutan todas las transformaciones, y hay un cambio de etapa solo en **join**, ya que requiere transferencia de datos entre diferentes nodos, es decir, es tipo wide.

Entonces, 2 etapas.


Data Flair
Narrow Transformation
Map
FlatMap
MapPartition
Filter
Sample
Union


Data Flair
Wide Transformation
Intersection
Distinct
ReduceBykey
GroupByKey
Join
Cartesian
Repartition
Coalesce