Resumen Spark por los Capos mafia

Teoría:

Entendemos por Big Data a datos que no pueden ser procesados por métodos tradicionales, ya que hablamos de un volumen que requiere ser almacenado en un cluster o datos cuya velocidad de arribo es infinita.

Almacenamiento Distribuido:
Es el encargado de la gestión. cómo y dónde guardar la información en una compu.
En cada equipo del cluster existe un File-system propio y uno global para todo el equipo del cluster, siendo posible mover archivos de un equipo a otro.
El gestor interno del File-System divide en bloques los datos y los almacena dentro de los diferentes equipos del cluster.
Estos datos son manejados por el nodo maestro, el cual 'sabe' de qué forma se repartieron los archivos. Este nodo tiene un backup de toda la wea.
Este nodo maestro se encarga de determinar en qué equipo está el registro que se necesita. Es un capo este nodo, lo demás es al pedo explicar

Cluster:
Conjunto de computadoras que trabajan juntas y pueden ser vistas como un sistema único

Map-Reduce:
Es el procesamiento distribuido de datos utilizando un cluster de computadoras.
Es un modelo de programación para procesar grandes conjuntos de datos
El usuario especifica una función map que procesa un par clave/valor para generar un conjunto intermedio de pares clave/valor.
Se debe especificar también una función reduce que combina todos los valores asociados a la misma clave
Consiste en 3 fases: (Anda a chequearlo a la c...)

- Fase Map:
- Fase reduce
- Fase shuffle and sort
  Tan explicadas abajo padre

Map:
Realiza transformaciones sobre nuestros datos convirtiendo cada registro del archivo en el formato que necesitemos creando un nuevo archivo distribuido.
Debe ser aplicada a cada dato de nuestro set.
Puede ser paralelizada y distribuirse entre las distintas máquinas de un cluster.
Además puede filtrar eliminando los registros que no sirven

Reduce:
Combina los resultados del map.
Es necesario procesar los datos de todas las máquinas del cluster.
Reduce locales en paralelo, y reduce entre máquina mediante etapa de shuffle y sort.
El resultado de un reduce entre dos registros es el input del siguiente reduce.
Diferentes implementaciones:

- reduceByKey:
  El sistema agrupa todos los registros para los cuales la clave es la misma. Requiere que todos los registros de igual clave estén en el mismo equipo que ejecute el reduce.
- reduce:
  Da un único resultado para todo el set de datos. Toma dos valores para dar como resultado la combinación de ambos. Las operaciones deben ser conmutativas y asociativas de modo de poder ejecutarse distribuidas.

Shuffle y Sort:
Mueve la salida de un proceso map a un cierto equipo de tal forma que un reducer pueda procesar sus registros.
Es la fase más costosa del proceso map-reduce.
Mueve datos de un nodo a otro.

Apache spark:
Es el sistema de procesamiento distribuido que vamos a usar.

Arquitectura:

- Comunicación entre un driver y una serie de ejecutores (executors).
- Tareas (jobs) del driver se convierten en tareas para los executors.
- Los resultados de las tareas vuelven al driver.

RDD:

- Estructura de datos que maneja Spark.
- Colecciones particionadas en un cluster.
- Guardados en memoria o en disco.
- Creados a partir de datos externos.
- Reconstruidos automáticamente frente a fallos de máquinas o demoras en un job.

Operaciones en Spark:
Transformaciones:

- Crean un rdd nuevo a partir de otro existente.
- Son **lazy**, es decir, **solo se ejecutan cuando definimos una acción.**
- Se pueden cachear, y a su vez las particiones del rdd.
- Map, Filter, FlatMap, ReducedByKey, GroupByKey, Join (cualquier tipo de join), Distinct.

Acciones:

- Devuelven un valor al driver después de procesar los datos.
- Son las que provocan que se ejecuten las transformaciones.
- Reduce, Collect, Count, Take, TakeOrdered, First, TakeSample, CountByKey

Observación: Si hay algún error en alguna transformación solo va a saltar cuando se ejecute una acción.

Cuando se cachea?
Siempre que una transformación o cadena de transformaciones vaya a utilizarse más de una vez. Es para guardar el resultado, y no tener que hacer la transformación otra vez.

Práctica (lo que le importa a los pibes):

Monstro no te olvides de lo básico:

- Crear spark context
  Spark = SparkSession.builder.getOrCreate()
  sc = spark.sparkContext
- Paralelizamos una colección de python
  Rdd = sc.parallelize(lista, cantidad de particiones). Si no no se especifica la cantidad tiene un valor por default (ver con *sc.defaultParalellism*)
- Leer archivos con textfile
  rdd = spark.sparkcontext.textfile('path')
- Leer archivos con SQLContext
  df = sqlcontext.read.text('path')
  df.rdd Así lo transformo a rdd

Rdd: si sos un macho pecho peludo usa help(nombre de la variable rdd) y te da todas las funciones

.getNumPartitions() **:** Devuelve el número de particiones del rdd
.toDebugString() : Devuelve el conjunto de transformaciones que se aplica

Acciones:

.count(): Devuelve la cantidad de registros del rdd
.take(n) : Devuelve los primeros n registros del rdd
.collect() : Obtiene todos los registros del rdd
(**ojo al piojo**, tenemos que saber que el erdd es acotado)
.First() : Obtiene el primer registro del rdd
.takeOrdered(n , orden) : Obtiene los primeros n registros en base a un orden indicado (**Utilizar de manera acotada**)
(*lambda x: x manera ascendiente, -x manera descendiente*)
*.TakeSample(bool, n)***:** Obtiene una muestra de n registros con o sin reemplazo (repetidas o no)

.reduce(lambda a b...) : Obtiene un solo registro, combinando el resultado en base a una función dada
.countByKey() : Cuenta ocurrencias de registros para cada clave.
(Solo con registros que tengan clave y elemento)

Transformaciones:

*.map(Lambda x: ...)***:** Transforma cada registro en base a una función dada
Por lo general se usa para generar nuevos rdd aplicando cambios a sus “columnas”
.filter(Lambda x: ...): Filtra registros en base a la función dada
*.flatMap(función)***:** Similar al map pero cada registro puede generar 0, 1 o más registros
*.reduceByKey(Lambda ...)***:** Combina los registros para una misma clave a una función de reduce, la cual debe ser **Conmutativa y Asociativa**
*.groupByKey()***:** Agrupa los registros para cada clave. Se obtiene todos los registros para cada clave.
Solo se debe utilizar si es necesario la info de cada registro y es acotado
*.distinct()***:** Elimina registros duplicados. Puede utilizarse para obtener palabras únicas
.*sortByKey()***:** Ordena los registros por clave

Útiles:

.split: Divide en palabras (anda a chequearlo a la)

Transformaciones entre dos RDD:

Unión: Realiza la unión entre dos RDD (No importa si los registros son iguales o no)
union = rdd1.union(rdd2)
intersection: Realiza la intersección entre dos rdd de registros.
interseccion = rdd1.intersection(rdd2)
subtract: Elimina del primer rdd los que estén en el segundo rdd
sub = rdd1.substract(rdd2)
joins: Se combinan dos rdd en base a las claves de los registros. Junta cada registro del rdd con cada registro del rdd2 que tengan la misma clave

inner join: Cuando tengo un set de datos del tipo (k,v) (k,w) devuelve (k,(v,w)) con todos los pares de elementos para cada clave(Las claves que están en común en ambos rdd)
rdd1.join(rdd2)
left outer join: Cuando tengo un set de datos del tipo (k,v) (k,w) devuelve (k,(v,w)) asegurando que todos los datos de la izq van a estar en el resultado del join
rdd1.leftOuterJoin(rdd2)
right outer join: Idem el de arriba pero con la derecha
rdd1.rightOuterJoin(rdd2)
outer/full join: Cuando tengo un set de datos del tipo (k,v) (k,w) devuelve (k,(v,w)) asegurándose que todos los datos estén aunque las claves no concuerden
rdd1.fullOuterJoin(rdd2)
broadcast join: Permite mantener una variable de solo lectura cacheada en cada

una de las máquinas del cluster
Sc.broadcast(rdd)

Transformaciones sobre las particiones:

.glom(): Junta los registros de cada partición en una lista
.mapPartitions(función): Devuelve un nuevo rdd aplicando una función a cada partición del rdd
.repartition(n): Reshuffle los datos en el rdd de forma aleatoria para crear más o menos particiones y balancearlas. Hace un shuffle de todos los datos por la red
.coalesce(n): Decrementa la cantidad de particiones del rdd. No hace un shuffle por defecto, solo pasa datos de una partición a otra. **No quedan balanceadas**.
.repartitionAndSortWithinPartitions(): Reparticionamos un rdd de acuerdo a un particionador y ordena los registros en base a su clave (los registros deberán tener clave)

Ascending

```
[ ] rdd.map(lambda x: (x, x)).repartitionAndSortWithinPartitions(2).glom().collect()
```

```
[ ] rdd.map(lambda x: (x % 3, x)).repartitionAndSortWithinPartitions(2).glom().collect()
```

```
[ ] rdd.map(lambda x: (x % 3, x)).repartitionAndSortWithinPartitions(2, ascending=False).glom().collect()
```

PartitionFunc

```
[ ] rdd.map(lambda x: (x * 2, x)).repartitionAndSortWithinPartitions(2).glom().collect()
```

```
[ ] rdd.map(lambda x: (x * 2, x)).repartitionAndSortWithinPartitions(2, partitionFunc=lambda x: (x % 3)).glom().collect()
```

Persistencia y Cache

.Cache(): Cachea un RDD intermedio que va a ser utilizado varias veces de modo de evitar tener que ejecutar todas las transformaciones cada vez.
.saveAsTextFile('nombre.txt'): Guarda un RDD a disco en un archivo de texto.
.saveAsPickeFile('nombre.file'): Guarda un RDD a disco en un archivo con los datos serializados.

Como Sacar el Promedio, por Mateo Rojas:

Tenemos un rdd de la forma: *{'x','y','z'}* y queremos sacar el promedio de y, donde x es el index.
Mapeamos: *rdd.map(lambda x: x[0], (x[1], 1))*
Queda: *{'x',('y',1)}*
Sumamos: *rdd.reduceByKey(lambda x,y: x[1][0] + y[1][0], x[1][1] + x[1][1])*
Queda: *{'clave', ('valor_total','cantidad_de_valores')}*
Y ahora calculamos el promedio: *rdd.map(lambda x: x[0],(x[1][0]/x[1][1]))*
Queda: *{'clave', 'promedio'}*

Contar Apariciones, por Mateo Rojas:

Supongamos que queremos meter en un dataframe del estilo:
{'x','HOLA!!!'} una variable que indique cuantos '!' hay
Tenemos que mapear y hacer lo siguiente:
rdd.map(lambda x: (x[0], x[1].count('!')))
Queda *{'x','cant_de_!_por_linea'}*

Sugerencias, por Mateo Rojas:

- Filtrar antes que todo
- NO usar sort by key, usar tarkeOrdered()