Persistencia y particionado

En este tema trataremos dos aspectos de Apache Spark

- **Persistencia**: cómo guardar DataFrames y RDDs de forma que no tengan que ser recalculados
- **Particionado**: cómo especificar y cambiar las particiones de un DataFrame o RDD

Persistencia

Problema al usar un DataFrame o un RDD varias veces:

- Spark recomputa el RDD y sus dependencias cada vez que se ejecuta una acción
- Muy costoso (especialmente en problemas iterativos)

Solución

- Conservar el DataFrame o RDD en memoria y/o disco
- Métodos `cache()` o `persist()`

Niveles de persistencia (definidos en `pyspark.StorageLevel` y `org.apache.spark.storage.StorageLevel`)

Nivel	Espacio	CPU	Memoria/Disco	Descripción
MEMORY_ONLY	Alto	Bajo	Memoria	Guarda el RDD como un objeto Java no serializado en la JVM. Si el RDD no cabe en memoria, algunas particiones no se *cachearán* y serán recomputadas "al vuelo" cada vez que se necesiten.
MEMORY_AND_DISK	Alto	Medio	Ambos	Guarda el RDD como un objeto Java no serializado en la JVM. Si el RDD no cabe en memoria, las particiones que no quepan se guardan en disco y se leen del mismo cada vez que se necesiten.
DISK_ONLY	Bajo	Alto	Disco	Guarda las particiones del RDD solo en disco.
OFF_HEAP	Bajo	Alto	Memoria	Similar a MEMORY_ONLY_SER pero guarda el RDD serializado usando memoria *off-heap* (fuera del heap de la JVM) lo que puede reducir el overhead del recolector de basura.

Nivel de persistencia

- El nivel por defecto para DataFrames es MEMORY_ONLY
- En Python, los datos siempre se guardan en memoria serializados (usando *pickled*)
  - Es posible especificar serialización (la forma en la que se serializan los datos para mantenerlos en memoria o en disco). Por defecto se utiliza el serializador "Pickle" en Python

Recuperación de fallos

- Si falla un nodo con datos almacenados, el DataFrame o RDD se recomputa
  - Añadiendo `_2` (ó `_3`) al nivel de persistencia (por ejemplo, MEMORY_ONLY_2), se guardan 2 copias del RDD

Gestión de la cache

- Algoritmo LRU (Least Recently Used) para gestionar la cache
  - Para niveles *solo memoria*, los RDDs viejos se eliminan y se recalculan
  - Para niveles *memoria y disco*, las particiones que no caben se escriben a disco

Importante:

- La persistencia debe usarse solo cuando sea necesaria, puesto que puede implicar un coste importante

Persistencia con DataFrames

In [ ]:
```
from pyspark import SparkContext
from pyspark.sql import SparkSession
import os

# Elegir el máster de Spark dependiendo de si se ha definido la variable de entorno HADOOP_CONF_DIR o YARN_CONF
```

```
SPARK_MASTER: str = 'yarn' if 'HADOOP_CONF_DIR' in os.environ or 'YARN_CONF_DIR' in os.environ else 'local[*]'
print(f"Usando Spark Master en {SPARK_MASTER}")

# Creamos un objeto SparkSession (o lo obtenemos si ya está creado)
spark: SparkSession = SparkSession \
  .builder \
  .appName("Mi aplicacion") \
  .config("spark.rdd.compress", "true") \
  .config("spark.executor.memory", "3g") \
  .config("spark.driver.memory", "3g") \
  .master(SPARK_MASTER) \
  .getOrCreate()

sc: SparkContext = spark.sparkContext
```

In [ ]:
```
from typing import Any
import numpy as np
from numpy import dtype, float64, ndarray
from pyspark.sql import Row
from pyspark.sql.dataframe import DataFrame

np_array: ndarray[Any, dtype[float64]] = np.random.random_sample(100000) # generates an array of M random value
row_type = Row("n", "x")
lista: list[Row] = [row_type(i, float(f)) for (i,f) in enumerate(np_array)]
DF1: DataFrame = spark.createDataFrame(lista)

DF1.printSchema()
print("Cacheado: {0}.".format(DF1.is_cached))
print("Nivel sin persistencia: {0}.".format(DF1.storageLevel))
```

In [ ]:
```
DF1.cache()
print("Cacheado: {0}.".format(DF1.is_cached))
print("Nivel de persistencia por defecto: {0}.".format(DF1.storageLevel))
```

In [ ]:
```
# La persistencia no se hereda en las transformaciones
DF2: DataFrame = DF1.groupBy("x").count()
print("Cacheado: {0}.".format(DF2.is_cached))
```

In [ ]:
```
# Para cambiar el nivel de persistencia, primero tenemos que quitarlo de la cache
DF1.unpersist()

from pyspark import StorageLevel
DF1.persist(StorageLevel.MEMORY_ONLY_2)
print("Cacheado: {0}.".format(DF1.is_cached))
print("Número de particiones: {0}.".format(DF1.rdd.getNumPartitions()))
print("Nuevo nivel de persistencia: {0}.".format(DF1.storageLevel))
```

Persistencia con RDDs

In [ ]:
```
from pyspark import RDD

rdd: RDD[int] = sc.parallelize(range(1000), 10)
print("Cacheado: {0}".format(rdd.is_cached))
print("Particiones: {0}".format(rdd.getNumPartitions()))
print("Nivel de persistencia sin cachear: {0}".format(rdd.getStorageLevel()))
```

In [ ]:
```
rdd.cache()

print("Cacheado: {0}".format(rdd.is_cached))
print("Nivel de persistencia por defecto: {0}".format(rdd.getStorageLevel()))
```

Particionado

El número de particiones es función del tamaño del cluster o el número de bloques del fichero en HDFS

- Es posible ajustarlo al crear u operar sobre un RDD
  - Los RDDs ofrecen un mayor control sobre el particionado
- En DataFrames es posible modificarlo una vez creados
- El paralelismo de RDDs que derivan de otros depende del de sus RDDs padre
- Propiedades útiles:
  - `spark.default.parallelism` Para RDDs, número de particiones por defecto devueltos por transformaciones como `parallelize`, `join` y `reduceByKey`
    - Fijado para un `SparkContext`
    - La propiedad `sc.defaultParallelism` indica su valor
  - `spark.sql.shuffle.partitions` Para DataFrames, número de particiones a usar al barajar datos en transformaciones *wide*
    - Puede cambiarse usando `spark.conf.set`
- Funciones útiles:

- `rdd.getNumPartitions()` devuelve el número de particiones del RDD
- `rdd.glom()` devuelve un nuevo RDD juntando los elementos de cada partición en una lista
- `repartition(n)` devuelve un nuevo DataFrame o RDD que tiene exactamente `n` particiones
- `coalesce(n)` más eficiente que `repartition`, minimiza el movimiento de datos
  - Sólo permite reducir el número de particiones
- `partitionBy(n,[partitionFunc])` Particiona por clave, usando una función de particionado (por defecto, un hash de la clave)
  - Solo para RDDs clave/valor
  - Asegura que los pares con la misma clave vayan a la misma partición

Particiones y RDDs

In [ ]:
```
print("Número de particiones por defecto para RDDs: {0}."
      .format(sc.defaultParallelism))
rdd: RDD[int] = sc.parallelize([1, 2, 3, 4, 2, 4, 1], 2)
pairs: RDD[tuple[int, int]] = rdd.map(lambda x: (x, x*x))

print("RDD pairs = {0}.".format(pairs.collect()))
print("Particionado de pairs: {0}.".format(pairs.glom().collect()))
print("Número de particiones de pairs = {0}.".format(pairs.getNumPartitions()))
```

In [ ]:
```
# Reducción manteniendo el número de particiones
from operator import add
print("Reducción manteniendo particiones: {0}.".format(
       pairs.reduceByKey(add).glom().collect()))
```

In [ ]:
```
# Reducción modificando el número de particiones
print("Reducción con 3 particiones: {0}.".format(
      pairs.reduceByKey(add, 3).glom().collect()))
```

In [ ]:
```
# Ejemplo de reparitición
pairs4: RDD[tuple[int, int]] = pairs.repartition(4)
print("pairs4 con {0} particiones: {1}.".format(
       pairs4.getNumPartitions(),
       pairs4.glom().collect()))
```

In [ ]:
```
# Ejemplo de coalesce
pairs2: RDD[tuple[int, int]] = pairs4.coalesce(2)
print("pairs2 con {0} particiones: {1}.".format(
       pairs2.getNumPartitions(),
       pairs2.glom().collect()))
```

In [ ]:
```
# Particionando por clave
pairs_clave: RDD[tuple[int, int]] = pairs2.partitionBy(3)
print("Particionado por clave ({0} particiones): {1}.".format(
       pairs_clave.getNumPartitions(),
       pairs_clave.glom().collect()))
```

In [ ]:
```
# Usando una función de particionado
def particionadoParImpar(clave):
    if clave%2:
        return 0  # Las claves impares van a la partición 0
    else:
        return 1  # Las claves pares van a la partición 1

pairs_parimpar: RDD[tuple[int, int]] = pairs2.partitionBy(2, particionadoParImpar)
print("Particionado por clave ({0} particiones): {1}.".format(
       pairs_parimpar.getNumPartitions(),
       pairs_parimpar.glom().collect()))
```

Particiones y DataFrames

In [ ]:
```
# Convertimos el RDD en un DataFrame
dfPairs: DataFrame = pairs.toDF()
dfPairs.show()
```

In [ ]:
```
# El DataFrame hereda el número de particiones del RDD
print("Número de particiones del DataFrame: {0}."
      .format(dfPairs.rdd.getNumPartitions()))
```

In [ ]:
```
# Una transformación narrow mantiene el número de particiones
print("Número de particiones tras transformacion narrow: {0}."
      .format(dfPairs.replace(1, 2).rdd.getNumPartitions()))
```

In [ ]:
```
# Una transformación wide no mantiene el número de particiones
print("Número de particiones tras transformacion wide: {0}."
      .format(dfPairs.sort("_1").rdd.getNumPartitions()))
```

In [ ]:
```
# Es posible especificar el número de particiones a usar en la transformación wide
```

```
spark.conf.set("spark.sql.shuffle.partitions", 2)
print("Número de particiones tras transformacion wide: {0}."
      .format(dfPairs.sort("_1").rdd.getNumPartitions()))
```

Trabajando a nivel de partición

Una operación `map` se hace para cada elemento de un RDD (o una `foreach` para cada fila del DataFrame)

- Puede implicar operaciones redundantes (p.e. abrir una conexión a una BD)
- Puede ser poco eficiente

Se pueden hacer `map` y `foreach` una vez por partición:

- Métodos `mapPartitions()`, `mapPartitionsWithIndex()` y `foreachPartition()`

In [ ]:
```
# Ejemplo de mapPartitions
from collections.abc import Iterable

nums: RDD[int] = sc.parallelize([1,2,3,4,5,6,7,8,9], 4)
print(nums.glom().collect())

def sumayCuenta(iter: Iterable[int]) -> list[int]:
    sumaCuenta: list[int] = [0,0]
    for i in iter:
        sumaCuenta[0] += i
        sumaCuenta[1] += 1
    return sumaCuenta

# Llama a la función sumayCuenta una vez por cada partición
# El iterador incluye los valores de la partición
print(nums.mapPartitions(sumayCuenta).glom().collect())
```

In [ ]:
```
# Ejemplo de mapPartitionsWithIndex
def sumayCuentaIndex(index: int, iter: Iterable[int]) -> tuple[str, list[int]]:
    return "Partición "+str(index), sumayCuenta(iter)

# index es el número de partición
print(nums.mapPartitionsWithIndex(sumayCuentaIndex).glom().collect())
```

In [ ]:
```
import os
import tempfile

# Ejemplo de foreachPartition
def f(iter: Iterable[int]) -> None:
    _, tempname = tempfile.mkstemp(dir=tempdir, text=True)
    with open(tempname, 'w') as fich:
        for x in iter:
            fich.write(str(x)+"\t")

tempdir = "/tmp/foreachPartition"

if not os.path.exists(tempdir):
    os.mkdir(tempdir)
    # Para cada partición del RDD se crea un fichero temporal
    # y se escribe en ese fichero los valores de la partición
    # foreachPartitions permite efectos colaterales (que no permite mapPartition)
    # y no devuelve nada
    nums.foreachPartition(f)
```

In [ ]:
```
%%sh
TEMP=/tmp/foreachPartition
echo "Ficheros creados"
ls -l $TEMP
echo
echo "Contenido de los ficheros"
for f in $TEMP/*;do cat $f; echo; echo "========"; done
rm -rf $TEMP
```