Spark: Aspectos avanzados

En este tema trataremos algunos aspectos adicionales de Apache Spark

- Cómo se ejecuta una aplicación Spark
- Uso de variables de broadcast y acumuladores

Ejecución de una aplicación Spark

Veremos conceptos relacionados con la ejecución de un código Spark

- Plan lógico y físico
- Trabajos, etapas y tareas

Plan lógico y físico

A partir de un código de usuario, Spark genera un *plan lógico*

- DAG (denominado *lineage graph*) con las operaciones a realizar
- No incluye información sobre el sistema físico en el que se va a ejecutar
- El optimizador *Catalyst* genera un plan lógico optimizado

A partir del plan lógico optimizado, se elabora un plan físico

- Especifica cómo se ejecutará el plan lógico en el cluster
- Se generan diferentes estrategias de ejecución que se comparan mediante un modelo de costes
  - Por ejemplo, cómo realizar un join en función de las características de los datos (tamaño, particiones, ...)

El plan físico se ejecuta en el cluster

- La ejecución se hace sobre RDDs

Trabajos, etapas y tareas

Una acción genera un *trabajo* (Spark job)

- El job se descompone en una o más *etapas* (stages)
- Las etapas representan grupos de *tareas* (tasks) que se ejecutan en paralelo
  - Cada tarea ejecuta una o más transformaciones sobre una partición
  - Las tareas se ejecutan en los nodos del cluster
- Una etapa termina cuando se realiza una operación de *barajado* (*shuffle*)
  - Implica mover datos entre los nodos del cluster

Pipelining: varios operaciones se pueden computan en una misma etapa

- Operaciones que no impliquen movimiento de datos (pe. `select`, `filter` o `map`)
- La salida de cada operación se pasa a la entrada de la siguiente sin ir a disco

Persistencia en el barajado

- Antes de un barajado, los datos se escriben a disco local
- Permite relanzar tareas falladas sin necesidad de recomputar todas las transformaciones
- No se realiza si los datos a barajar ya han sido cacheados (con `cache` o `persist`)

En el interfaz web de Spark se muestran información sobre las etapas y tareas

- El método `explain()` de los DataFrames o `toDebugString()` de los RDDs muestra el plan físico

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 pyspark.sql.dataframe import DataFrame
from pyspark.sql.functions import sum,col

# Ejemplo para visualizar el plan físico
df1: DataFrame = spark.range(2, 10000000, 2)
df2: DataFrame = spark.range(2, 10000000, 4)
step1: DataFrame = df1.repartition(5)
step12: DataFrame = df2.repartition(6)
step2: DataFrame = step1.selectExpr("id * 5 as id")
step3: DataFrame = step2.join(step12, ["id"])
step4: DataFrame = step3.select(sum(col("id")))

print(step4.collect())
step4.explain()
```

Variables de broadcast

- Por defecto, todas las variables compartidas (no RDDs) son enviadas a todos los ejecutores
  - Se reenvían en cada operación en la que aparezcan
- Variables de broadcast: permiten enviar de forma eficiente variables de solo lectura a los workers
  - Se envían solo una vez

In [ ]:
```
from pyspark import RDD

rdd: RDD[int] = sc.parallelize([1,2,3,4])
x=5
rdd2: RDD[int] = rdd.map(lambda n: x*n)
rdd2: RDD[int] = rdd2.map(lambda n: x-n)
print(rdd2.collect())
```

In [ ]:
```
from operator import add
from pyspark import Broadcast

# dicc es una variable de broadcast
dicc: dict[str, str] = {"a": "alpha", "b": "beta", "c": "gamma"}
bcastDicc: Broadcast[dict[str, str]] = sc.broadcast(dicc)

rdd: RDD[tuple[str, int]] = sc.parallelize([("a", 1),("b", 3),("a", -4),("c", 0)])
reduced_rdd: RDD[tuple[str | None, int]] =  \
    rdd.reduceByKey(add).map(lambda tupla: (bcastDicc.value.get(tupla[0]), tupla[1]))

print(reduced_rdd.collect())
```

Acumuladores

Permiten agregar valores desde los *worker nodes*, que se pasan al *driver*

- Útiles para contar eventos
- Solo el driver puede acceder a su valor
- Acumuladores usados en transformaciones de RDDs pueden ser incorrectos
  - Si el RDD se recalcula, el acumulador puede actualizarse
  - En acciones, este problema no ocurre
- Por defecto, los acumuladores son enteros o flotantes
  - Es posible crear “acumuladores a medida” usando `AccumulatorParam`

In [ ]:
```
from pyspark import Accumulator
from pyspark.sql import Row
from random import randint

# Creamos el DataFrame a partir de una lista de objetos Row
# con enteros aleatorios
l: list[Row] = [Row(randint(1,10)) for n in range(10000)]
df: DataFrame = spark.createDataFrame(l)
```

```python
# Definimos un acumulador
npares: Accumulator[int] = sc.accumulator(0)

# si el número en una fila es par, incrementamos en acumulador
def contarPares(fila: Row) -> None:
    global npares
    if fila["_1"]%2 == 0:
        npares += 1

# Ejecutamos la función una vez por fila
df.foreach(contarPares)

print("Numero de pares: {0}".format(npares.value))
```