2d5fhg8ep

March 3, 2025

0.1 Películas más votadas

0.1.1 Sin nombre, solo tabla Ratings

[0]:
```
from pyspark.sql import functions as func
from pyspark.sql.types import StructType, StructField, IntegerType, LongType

# Definir el esquema del DataFrame de ratings
schema = StructType([ \
                     StructField("userID", IntegerType(), True), \
                     StructField("movieID", IntegerType(), True), \
                     StructField("rating", IntegerType(), True), \
                     StructField("timestamp", LongType(), True)])

# Cargar el archivo de ratings u-data en un DataFrame, con separador el␣
 ↪tabulador \t
moviesDF = spark.read.option("sep", "\t").schema(schema).csv("dbfs:/FileStore/u.
 ↪data")

# Funciones tipo SQL para agrupar y ordenar
topMovieIDs = moviesDF.groupBy("movieID").count().orderBy(func.desc("count"))

# Top 10
topMovieIDs.show(10)
```

```
+-------+-----+
|movieID|count|
+-------+-----+
|     50|  583|
|    258|  509|
|    100|  508|
|    181|  507|
|    294|  485|
|    286|  481|
|    288|  478|
|      1|  452|
|    300|  431|
|    121|  429|
+-------+-----+
```

```
only showing top 10 rows
```

0.1.2 Con nombre, haciendo inner join con películas (u.item)

[0]:
```
from pyspark.sql.types import StructType, StructField, IntegerType, StringType
from pyspark.sql import functions as F

# Definir el esquema del DataFrame de ratings
esquemaRatings = StructType([
    StructField("UserID", IntegerType(), True),
    StructField("MovieID", IntegerType(), True),
    StructField("Rating", IntegerType(), True),
    StructField("Timestamp", IntegerType(), True)
])

# Cargar el archivo de ratings u-data en un DataFrame, con separador el␣
 ↪tabulador \t
dfRatings = spark.read.csv("dbfs:/FileStore/u.data", sep="\t",␣
 ↪schema=esquemaRatings, header=False)

# Definir esquema para el DataFrame de películas
esquemaPeliculas = StructType([
    StructField("MovieID", IntegerType(), True),
    StructField("Title", StringType(), True),
    StructField("ReleaseDate", StringType(), True),
    StructField("EmptyColumn", StringType(), True),
    StructField("IMDB_URL", StringType(), True),
    StructField("Unknown", IntegerType(), True),
    StructField("Action", IntegerType(), True),
    StructField("Adventure", IntegerType(), True),
    StructField("Animation", IntegerType(), True),
    StructField("Children", IntegerType(), True),
    StructField("Comedy", IntegerType(), True),
    StructField("Crime", IntegerType(), True),
    StructField("Documentary", IntegerType(), True),
    StructField("Drama", IntegerType(), True),
    StructField("Fantasy", IntegerType(), True),
    StructField("FilmNoir", IntegerType(), True),
    StructField("Horror", IntegerType(), True),
    StructField("Musical", IntegerType(), True),
    StructField("Mystery", IntegerType(), True),
    StructField("Romance", IntegerType(), True),
    StructField("SciFi", IntegerType(), True),
    StructField("Thriller", IntegerType(), True),
    StructField("War", IntegerType(), True),
    StructField("Western", IntegerType(), True)
```

```
])

# Cargar el archivo de películas en un DataFrame, con separador |
dfPeliculas = spark.read.csv("dbfs:/FileStore/u.item", sep="|",␣
 ↪schema=esquemaPeliculas, header=False)

# Mostrar las 10 películas con más votos
dfRatingsNombres = dfRatings.join(dfPeliculas,on="MovieID",how="inner")

dfRatingsAgrupados = dfRatingsNombres.groupBy("Title").agg(F.count("Title").
 ↪alias("Ratings")).orderBy("Ratings",ascending=False)
dfRatingsAgrupados.show(10)
```

```
+--------------------+-------+
|               Title|Ratings|
+--------------------+-------+
|    Star Wars (1977)|    583|
|      Contact (1997)|    509|
|        Fargo (1996)|    508|
|Return of the Jed…|    507|
|    Liar Liar (1997)|    485|
|English Patient, …|    481|
|       Scream (1996)|    478|
|    Toy Story (1995)|    452|
|Air Force One (1997)|    431|
|Independence Day …|    429|
+--------------------+-------+
only showing top 10 rows
```

0.1.3 Con un diccionario, UDF y broadcast

[0]:
```
from pyspark.sql import functions as func
from pyspark.sql.types import StructType, StructField, IntegerType, LongType
import codecs

# Genera un diccionario con código y nombre de película
def loadMovieNames():
    movieNames = {}
    lines = sc.textFile("dbfs:/FileStore/u.item")
    collected_lines = lines.collect()  # Convierte el RDD en una lista
    for line in collected_lines:
        fields = line.split('|')
        movieNames[int(fields[0])] = fields[1]
    return movieNames
```

```
# Carga el diccionario en una variable distribuida a todos los nodos con␣
 ↪broadcast
nameDict = spark.sparkContext.broadcast(loadMovieNames())

# Crea el esquema de rating
schema = StructType([ \
                     StructField("userID", IntegerType(), True), \
                     StructField("movieID", IntegerType(), True), \
                     StructField("rating", IntegerType(), True), \
                     StructField("timestamp", LongType(), True)])

# Carga df de ratings
moviesDF = spark.read.option("sep", "\t").schema(schema).csv("dbfs:/FileStore/u.
 ↪data")

movieCounts = moviesDF.groupBy("movieID").count()

# Crea una función definida por usuario (UDF) para buscar nombres de películas␣
 ↪en el diccionario distribuido a partir del código
def lookupName(movieID):
    return nameDict.value[movieID]

lookupNameUDF = func.udf(lookupName)

# Añade la columna nombre de película usando la función UDF
moviesWithNames = movieCounts.withColumn("movieTitle", lookupNameUDF(func.
 ↪col("movieID")))

# Ordena los resultados
sortedMoviesWithNames = moviesWithNames.orderBy(func.desc("count"))

# Muestra los 10 primeros
sortedMoviesWithNames.show(10, False)
```

```
+-------+-----+----------------------------+
|movieID|count|movieTitle                  |
+-------+-----+----------------------------+
|50     |583  |Star Wars (1977)            |
|258    |509  |Contact (1997)              |
|100    |508  |Fargo (1996)                |
|181    |507  |Return of the Jedi (1983)   |
|294    |485  |Liar Liar (1997)            |
|286    |481  |English Patient, The (1996) |
|288    |478  |Scream (1996)               |
|1      |452  |Toy Story (1995)            |
|300    |431  |Air Force One (1997)        |
|121    |429  |Independence Day (ID4) (1996)|
```

```
+-------+-----+----------------------------+
only showing top 10 rows
```

0.1.4 Obtener una lista de nombres de películas y un diccionario con el número de votos en cada puntuación:

[0]:
```
# Nos quedamos con las columnas MovieID y Rating
dfRatings = dfRatings.select("MovieID", "Rating")

# Convertir el DataFrame de ratings a un RDD de filas
rddFilas = dfRatings.rdd

# Convertir el RDD de filas a un RDD de tuplas
rddTuplas = rddFilas.map(lambda fila: (fila[0], (fila[1],1)))

# Función para crear un diccionario con el número de votos para cada puntuación
def crearRatingDict(tuplas):
    RatingDict = {}
    for rating, cont in tuplas:
        if rating in RatingDict:
            RatingDict[rating] += cont
        else:
            RatingDict[rating] = cont
    return RatingDict

# Agrupar por MovieID y agregar las puntuaciones
rddRatingsAgrupados = rddTuplas.groupByKey().mapValues(crearRatingDict)

# Volver a convertir a dataframe y hacer join con películas para obtener el␣
 ↪nombre

# Mostrar 10 películas con su nombre y puntuaciones
```

0.1.5 Superhéroe más relacionado

En cada línea aparece un código de superhéroe y los códigos de otros superhéroes que aparecen con él en algún comic. Puede aparecer repetido en más de una línea.

Ficheros: Marvel-names.txt, Marvel-graph.txt

Obtener cuál es el superhéroe que más relaciones tiene con otros superhéroes.

[0]:
```
from pyspark.sql import functions as func
from pyspark.sql.types import StructType, StructField, IntegerType, StringType

schema = StructType([ \
                     StructField("id", IntegerType(), True), \
```

```
                    StructField("name", StringType(), True)])

names = spark.read.schema(schema).option("sep", " ").csv("dbfs:/FileStore/
 ↪Marvel_names.txt")

lines = spark.read.text("dbfs:/FileStore/Marvel_graph.txt")

# Trim de la columna para eliminar posibles espacios duplicados
connections = lines.withColumn("id", func.split(func.trim(func.col("value")), "␣
 ↪")[0]) \
    .withColumn("connections", func.size(func.split(func.trim(func.
 ↪col("value")), " ")) - 1) \
    .groupBy("id").agg(func.sum("connections").alias("connections"))

mostPopular = connections.sort(func.col("connections").desc()).first()

mostPopularName = names.filter(func.col("id") == mostPopular[0]).select("name").
 ↪first()

print(mostPopularName[0] + " es el superhéroe más relacionado con " +␣
 ↪str(mostPopular[1]) + " relaciones.")
```

```
CAPTAIN AMERICA es el superhéroe más relacionado con 1933 relaciones.
```

0.1.6 Distancia entre superhéroes

Grado de separación entre dos superhéroes, calculándolo a partir de las apariciones conjuntas en un comic.

Utilizamos algoritmo Breadth-first search: recorre un árbol o grafo nivel por nivel, comenzando desde la raíz (o nodo inicial) y explorando todos los nodos vecinos en el nivel actual antes de moverse al siguiente nivel.

BFS es útil para encontrar la ruta más corta en grafos no ponderados y para explorar todos los nodos a una cierta "profundidad" del nodo inicial.

Pasos del algoritmo:

- Inicialización:
  - Coloca el nodo inicial en una cola (queue).
  - Marca el nodo inicial como visitado.
- Proceso de recorrido:
  - Mientras la cola no esté vacía:
    - Saca (dequeue) el nodo al frente de la cola.
    - Procesa el nodo (por ejemplo, imprime su valor).
    - Para cada nodo vecino no visitado:
      - Marca el vecino como visitado.
      - Añade (enqueue) el vecino a la cola.

[0]:
```python
# Ejemplo en Python:
from collections import deque

def bfs(tree, start_node):
    visited = set()
    queue = deque([start_node])
    visited.add(start_node)

    while queue:
        node = queue.popleft()
        print(node)  # Procesa el nodo

        for neighbor in tree[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

# Ejemplo de uso
tree = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': [],
    'E': [],
    'F': [],
    'G': []
}

bfs(tree, 'A')
```

```
A
B
C
D
E
F
G
```

0.2 BFS map-reduce

Objetivo: Encontrar el camino más corto entre un nodo de inicio y un nodo destino en un grafo.
Método: Se utiliza un algoritmo de búsqueda en anchura (BFS) basado en operaciones de map-reduce.
Estrategia:
- Cada nodo se representa con (nodo, ([vecinos], 9999, Pend))
- Lista de vecinos.
- Distancia: inicialmente 9999.
- Estado: Pend (no visitado), Doing (en cola para expandir) y Done (procesado).

- El nodo inicial se representa con distancia 0 y estado Doing: (N1, ([N2, N3], 0, Doing) - En cada iteración se llama a las funciones map y reduce: - Map: - Expande nodos Doing, generando registros para sus vecinos con la distancia incrementada: (N2, ([], 1, Doing)), (N3, ([], 1, Doing)) - Reproduce nodo expandido como finalizado: (N1, ([N2, N3], 0, Done)) - El resto de nodos se mantiene igual. - Reduce: - Combina registros de cada nodo: - Une conjunto de nodos adyacentes. - Mínimo de distancias. - Estado más avanzado. - Se utiliza un acumulador para detectar cuando se alcanza el Nodo destino.

Iteración

Datos Iniciales

Resultado Map

Resultado Reduce

Iteración 1

- N1: ([2,3], 0, Doing)- N2: ([4], 9999, N/A)- N3: ([4,5], 9999, N/A)- N4: ([], 9999, N/A)- N5: ([], 9999, N/A)

```
        <td>Se expande nodo 1 (<b>Doing</b>):<br> - Emite para N2: (2, ([], 1, <b>Doing</b>)
        <td>Agrupación por nodo:<br>
            - <b>N1</b>: (1, ([2,3], 0, <b>Done</b>))<br>
            - <b>N2</b>: Combina (2, ([], 1, <b>Doing</b>)) y (2, ([4], 9999, <b>Doing</b>))
            - <b>N3</b>: Combina (3, ([], 1, <b>Doing</b>)) y (3, ([4,5], 9999, <b>Doing</b>
            - <b>N4</b>: (4, ([], 9999, N/A))<br>
            - <b>N5</b>: (5, ([], 9999, N/A))
        </td>
    </tr>
    <tr>
        <td><b>Iteración 2</b></td>
        <td>Se parte del resultado reduce de Iteración 1:<br> - N1: ([2,3], 0, <b>Done</b>)<
        <td>Se expanden nodos 2 y 3 (<b>Doing</b>):<br>
            - <b>N2</b> (<b>Doing</b>):<br> -- Emite para N4: (4, ([], 2, <b>Doing</b>))<br>
            - <b>N3</b> (<b>Doing</b>):<br> -- Emite para N4: (4, ([], 2, <b>Doing</b>))<br>
        </td>
        <td>Agrupación por nodo:<br>
            - <b>N1</b>: (1, ([2,3], 0, <b>Done</b>))<br>
            - <b>N2</b>: Combina (2, ([], 2, <b>Doing</b>)) y (2, ([4], 1, <b>Done</b>)) → (
            - <b>N3</b>: Combina (3, ([], 2, PE)) y (3, ([4,5], 1, PR)) → (3, ([4,5], 1, <b>
            - <b>N4</b>: Combina (4, ([], 2, <b>Doing</b>)) [de N2] y (4, ([], 2, <b>Doing</
            - <b>N5</b>: Combina (5, ([], 2, <b>Doing</b>)) y (5, ([], 9999, N/A)) → (5, ([]
        </td>
    </tr>
```

[0]:
```
startHeroID = 5306 #SpiderMan
targetHeroID = 14  #ADAM 3,031

# Acumulador para indicar que hemos encontrado el objetivo en el recorrido␣
 ↪horizontal del árbol
```

```
hitCounter = sc.accumulator(0)

def convertToBFS(line):
    fields = line.split()
    heroID = int(fields[0])
    connections = []
    for connection in fields[1:]:
        connections.append(int(connection))

    status = 'N/A' # No alcanzado
    distance = 9999

    if (heroID == startHeroID):
        status = 'Doing'  # En proceso
        distance = 0

    return (heroID, (connections, distance, status))


def createStartingRdd():
    inputFile = sc.textFile("dbfs:/FileStore/Marvel_graph.txt")
    return inputFile.map(convertToBFS)


def bfsMap(node):
    heroID = node[0]
    data = node[1]
    connections = data[0]
    distance = data[1]
    status = data[2]

    results = []

    if status == 'Doing':
        for connectionHero in connections:
            # Añade un nodo por cada conexión
            results.append((connectionHero, ([], distance + 1, 'Doing')))


            # Si se alcanza el nodo buscado, se incrementa el contador
            if connectionHero == targetHeroID:
                hitCounter.add(1)

        # Añade el nodo recibido como procesado
        results.append((heroID, (connections, distance, 'Done')))

    else:
```

```
        # Añade el nodo recibido sin procesar
        results.append((heroID, (connections, distance, status)))

    return results

def bfsReduce(data1, data2):
    connections1 = data1[0]
    connections2 = data2[0]
    distance1 = data1[1]
    distance2 = data2[1]
    status1 = data1[2]
    status2 = data2[2]

    # Unifica las dos listas de conexiones
    connections = list(set(connections1 + connections2))

    # Se queda con la distancia menor
    distance = min(distance1, distance2)

    # Se queda con el estado más avanzado
    status_priority = {'N/A': 0, 'Pend': 1, 'Doing': 2, 'Done': 3}
    status = status1 if status_priority[status1] > status_priority[status2]␣
 ↪else status2

    return (connections, distance, status)


#Programa principal
iterationRdd = createStartingRdd()

iterationRdd.collect()

for iteration in range(1, 10):
    print("Procesando iteración # " + str(iteration))

    # Expande nodos Doing, generando registros para sus vecinos con la␣
 ↪distancia incrementada. El nodo expandido se añade como finalizado.
    # El resto de nodos se queda igual.
    # Si se alcanza el nodo buscado, se incrementa el acumulador para indicar␣
 ↪que hemos terminado.

    mapped = iterationRdd.flatMap(bfsMap)

    # Se ejecuta la acción mapped.count() para forzar la evaluación del RDD y␣
 ↪la actualización del acumulador
    print("Procesando " + str(mapped.count()) + " valores.")
```

```
    if (hitCounter.value > 0):
        print("Se ha localizado el objetivo. Ramas paralelas en las que se ha␣
↪alcanzado: " + str(hitCounter.value))
        break

    # Reducer combina registros de cada id, uniendo los nodos adyacentes, y␣
↪dejando el número de pasos menor y el estado más avanzado
    iterationRdd = mapped.reduceByKey(bfsReduce)
```

```
Procesando iteración # 1
Procesando 8330 valores.
Procesando iteración # 2
Procesando 220615 valores.
Se ha localizado el objetivo. Ramas paralelas en las que se ha alcanzado: 1
```