1 Transformaciones en Apache Spark

- **map()**: Aplica una función a cada elemento del RDD.

```
rdd = sc.parallelize([1, 2, 3, 4])
mapped_rdd = rdd.map(lambda x: x * 2)
print(mapped_rdd.collect())  # Resultado: [2, 4, 6, 8]
```

- **filter()**: Filtra los elementos del RDD según una condición.

```
rdd = sc.parallelize([1, 2, 3, 4, 5])
filtered_rdd = rdd.filter(lambda x: x % 2 == 0)
print(filtered_rdd.collect())  # Resultado: [2, 4]
```

- **flatMap()**: Similar a `map()`, pero devuelve múltiples valores por entrada.

```
rdd = sc.parallelize([1, 2, 3])
flat_mapped_rdd = rdd.flatMap(lambda x: [x, x * 2])
print(flat_mapped_rdd.collect())  # Resultado: [1, 2, 2, 4, 3, 6]
```

- **groupByKey()**: Agrupa los valores de un RDD por clave.

```
rdd = sc.parallelize([('a', 1), ('b', 2), ('a', 3), ('b', 4)])
grouped_rdd = rdd.groupByKey()
print([(x, list(y)) for x, y in grouped_rdd.collect()])
# Resultado: [('a', [1, 3]), ('b', [2, 4])]
```

- **reduceByKey()**: Agrupa por clave y aplica una función de reducción.

```
rdd = sc.parallelize([('a', 1), ('b', 2), ('a', 3), ('b', 4)])
reduced_rdd = rdd.reduceByKey(lambda x, y: x + y)
print(reduced_rdd.collect())  # Resultado: [('a', 4), ('b', 6)]
```

- **join()**: Une dos RDDs por una clave común.

```
rdd1 = sc.parallelize([('a', 1), ('b', 2)])
rdd2 = sc.parallelize([('a', 3), ('b', 4)])
joined_rdd = rdd1.join(rdd2)
print(joined_rdd.collect())  # Resultado: [('a', (1, 3)), ('b', (2, 4))]
```

- **union()**: Devuelve un RDD que contiene los elementos de ambos RDDs.

```
rdd1 = sc.parallelize([1, 2, 3])
rdd2 = sc.parallelize([4, 5])
union_rdd = rdd1.union(rdd2)
print(union_rdd.collect())  # Resultado: [1, 2, 3, 4, 5]
```

- **distinct()**: Devuelve un RDD con elementos únicos.

```
rdd = sc.parallelize([1, 2, 2, 3, 4, 4, 5])
distinct_rdd = rdd.distinct()
print(distinct_rdd.collect())  # Resultado: [1, 2, 3, 4, 5]
```

- **cartesian()**: Devuelve el producto cartesiano de dos RDDs.

```
rdd1 = sc.parallelize([1, 2])
rdd2 = sc.parallelize([3, 4])
cartesian_rdd = rdd1.cartesian(rdd2)
print(cartesian_rdd.collect())  # Resultado: [(1, 3), (1, 4), (2, 3), (2, 4)]
```

- **coalesce()**: Reduce el número de particiones en el RDD.

```
rdd = sc.parallelize([1, 2, 3, 4, 5], 5)
coalesced_rdd = rdd.coalesce(2)
print(coalesced_rdd.getNumPartitions())  # Resultado: 2
```

- **repartition()**: Aumenta o disminuye el número de particiones.

```
rdd = sc.parallelize([1, 2, 3, 4, 5], 2)
repartitioned_rdd = rdd.repartition(4)
print(repartitioned_rdd.getNumPartitions())  # Resultado: 4
```

- **subtract()**: Devuelve un RDD que contiene los elementos de un RDD que no están en otro.

```
rdd1 = sc.parallelize([1, 2, 3, 4])
rdd2 = sc.parallelize([3, 4])
subtracted_rdd = rdd1.subtract(rdd2)
print(subtracted_rdd.collect())  # Resultado: [1, 2]
```

- **intersection()**: Devuelve los elementos comunes entre dos RDDs.

```
rdd1 = sc.parallelize([1, 2, 3, 4])
rdd2 = sc.parallelize([3, 4, 5])
intersection_rdd = rdd1.intersection(rdd2)
print(intersection_rdd.collect())  # Resultado: [3, 4]
```

- **sample()**: Devuelve una muestra aleatoria de los elementos del RDD.

```
rdd = sc.parallelize([1, 2, 3, 4, 5])
sample_rdd = rdd.sample(False, 0.4)
print(sample_rdd.collect())  # Resultado: [varios valores al azar]
```

2 Acciones en Apache Spark

- **collect()**: Recupera todos los elementos del RDD y los trae al driver.

```
rdd = sc.parallelize([1, 2, 3, 4])
result = rdd.collect()
print(result)  # Resultado: [1, 2, 3, 4]
```

- **count()**: Devuelve el número de elementos en el RDD.

```
rdd = sc.parallelize([1, 2, 3, 4])
result = rdd.count()
print(result)  # Resultado: 4
```

- **take(n)**: Devuelve los primeros `n` elementos del RDD.

```
rdd = sc.parallelize([1, 2, 3, 4, 5])
result = rdd.take(3)
print(result)  # Resultado: [1, 2, 3]
```

- **first()**: Devuelve el primer elemento del RDD.

```
rdd = sc.parallelize([1, 2, 3, 4])
result = rdd.first()
print(result)  # Resultado: 1
```

- **takeSample(withReplacement, num)**: Devuelve una muestra aleatoria de `num` elementos del RDD.

```
rdd = sc.parallelize([1, 2, 3, 4, 5])
result = rdd.takeSample(False, 3)
print(result)  # Resultado: [valores aleatorios de 1 a 5]
```

- **reduce(func)**: Aplica una función de reducción (como una suma) a los elementos del RDD.

```
rdd = sc.parallelize([1, 2, 3, 4])
result = rdd.reduce(lambda x, y: x + y)
print(result)  # Resultado: 10
```

- **countByKey()**: Cuenta los elementos de cada clave en un RDD de pares (clave, valor).

```
rdd = sc.parallelize([('a', 1), ('b', 2), ('a', 3)])
result = rdd.countByKey()
print(result)  # Resultado: {'a': 2, 'b': 1}
```

- **foreach(func)**: Aplica una función a cada elemento del RDD, generalmente para realizar efectos secundarios como escribir en una base de datos.

```
rdd = sc.parallelize([1, 2, 3, 4])
rdd.foreach(lambda x: print(x))
# Imprime: 1, 2, 3, 4
```

- **saveAsTextFile(path)**: Guarda los elementos del RDD como archivos de texto en el sistema de archivos.

```
rdd = sc.parallelize([1, 2, 3, 4])
rdd.saveAsTextFile("/ruta/del/directorio")
```

- **saveAsSequenceFile(path)**: Guarda los elementos del RDD como un archivo de secuencia (clave, valor) en el sistema de archivos (formato Hadoop SequenceFile).

```
rdd = sc.parallelize([('a', 1), ('b', 2)])
rdd.saveAsSequenceFile("/ruta/del/archivo")
```

- **saveAsObjectFile(path)**: Guarda los elementos del RDD en un formato serializado de Java (objeto).

```
rdd = sc.parallelize([1, 2, 3, 4])
rdd.saveAsObjectFile("/ruta/del/archivo")
```

- **takeOrdered(n, key=func)**: Devuelve los primeros `n` elementos, ordenados de acuerdo con la función de clave especificada.

```
rdd = sc.parallelize([5, 3, 1, 4, 2])
result = rdd.takeOrdered(3)
print(result)  # Resultado: [1, 2, 3]
```

- **top(n)**: Devuelve los `n` elementos más grandes en el RDD.

```
rdd = sc.parallelize([5, 3, 1, 4, 2])
result = rdd.top(3)
print(result)  # Resultado: [5, 4, 3]
```

- **aggregate(zeroValue)(seqOp, combOp)**: Agrega los elementos del RDD en paralelo, aplicando una operación secuencial y una operación combinatoria. Es más flexible que `reduce`.

```
rdd = sc.parallelize([1, 2, 3, 4])
result = rdd.aggregate(0)(
    lambda acc, x: acc + x,  # secOp: suma elementos dentro de una partición
    lambda acc1, acc2: acc1 + acc2  # combOp: combina los resultados de las partici
)
print(result)  # Resultado: 10
```

- **fold(zeroValue)(func)**: Similar a `reduce`, pero con un valor inicial (como en `aggregate`).

```
rdd = sc.parallelize([1, 2, 3, 4])
result = rdd.fold(0, lambda acc, x: acc + x)
print(result)  # Resultado: 10
```