Este código está escrito en Scala y utiliza Apache Spark para realizar operaciones distribuidas en un conjunto de datos (RDD). Aquí tienes un desglose de lo que hace cada parte del código:

1. **Crear un RDD a partir de una colección**:

```
val rdd = spark.sparkContext.parallelize(1 to 1000000, 100)
```

   Esto crea un RDD a partir de una colección de números del 1 al 1,000,000, distribuidos en 100 particiones.

2. **Aplicar transformaciones complejas**:

```
val resultRDD = rdd
  .map(x => (x % 10, x))
  .repartitionAndSortWithinPartitions(new HashPartitioner(10))
  .mapValues(x => x * x)
  .reduceByKey(_ + _)
```

   - `map(x => (x % 10, x))`: Transforma cada elemento en un par `(clave, valor)` donde la clave es el residuo de dividir el número por 10.
   - `repartitionAndSortWithinPartitions(new HashPartitioner(10))`: Redistribuye los datos en 10 particiones y ordena los elementos dentro de cada partición.
   - `mapValues(x => x * x)`: Eleva al cuadrado cada valor.
   - `reduceByKey(_ + _)`: Suma los valores para cada clave.

3. **Acción para materializar el resultado**:

```
val result = resultRDD.collect()
```

   Esta acción recopila los resultados del RDD transformado en el controlador.

   // Imprimir los primeros 10 elementos del resultado
   result.take(10).foreach(println)

1. **Crear una secuencia de datos simulando las ventas**:

```
val ventas = Seq(
  Venta(Date.valueOf("2024-09-01"), "ProductoA", 10, 20.5),
  Venta(Date.valueOf("2024-09-01"), "ProductoB", 5, 10.0),
  // ... más datos de ventas ...
)
```

   Esto crea una secuencia de objetos `Venta` con datos de ventas para diferentes productos en distintas fechas.

2. **Crear un Dataset a partir de la secuencia de datos**:

```
val ventasDS = ventas.toDS()
```

   Convierte la secuencia de ventas en un Dataset de Spark.

3. **Realizar operaciones complejas**:

```
val resultadoDF = ventasDS
  .groupBy($"fecha", $"producto")
  .agg(F.sum($"cantidad").as("total_cantidad"), F.sum($"precio" *
$"cantidad").as("total_ventas"))
  .withColumn("promedio_7_dias",
    F.avg($"total_ventas").over(Window.partitionBy($"producto")
      .orderBy($"fecha")
      .rowsBetween(-6, 0)))
  .filter($"total_ventas" > $"promedio_7_dias")
```

   - `groupBy($"fecha", $"producto")`: Agrupa los datos por fecha y producto.
   - `agg(F.sum($"cantidad").as("total_cantidad"), F.sum($"precio" * $"cantidad").as("total_ventas"))`: Calcula la suma de las cantidades y el total de ventas para cada grupo.
   - `withColumn("promedio_7_dias", F.avg($"total_ventas").over(Window.partitionBy($"producto").orderBy($"fecha").rowsBetween(-6, 0)))`: Calcula el promedio móvil de 7 días de las ventas totales para cada producto.
   - `filter($"total_ventas" > $"promedio_7_dias")`: Filtra los registros donde las ventas totales son mayores que el promedio móvil de 7 días.

4. **Mostrar el resultado**:

```
resultadoDF.show()
```

   Muestra el DataFrame resultante.

1. **ear una secuencia de datos simulando los eventos**:

```
val eventos = Seq(
  Evento(Timestamp.valueOf("2024-09-01 10:00:00"), "user1", "compra", 100.0),
  Evento(Timestamp.valueOf("2024-09-01 12:30:00"), "user2", "compra", 200.0),
  // ... más datos de eventos ...
)
```

   Esto crea una secuencia de objetos `Evento` con datos de eventos, incluyendo la marca de tiempo, el usuario, el tipo de evento y el monto.

2. **Crear un Dataset a partir de la secuencia de datos**:

```
val df = eventos.toDF()
```

   Convierte la secuencia de eventos en un DataFrame de Spark.

3. **Procesar el DataFrame**:

```
val dfProcesado = df
  .withColumn("fecha", to_date($"timestamp"))
  .withColumn("hora", hour($"timestamp"))
  .withColumn("es_fin_semana", when(dayofweek($"fecha").isin(1, 7),
true).otherwise(false))
  .groupBy($"fecha", $"es_fin_semana")
  .agg(
    countDistinct($"usuario_id").as("usuarios_unicos"),
    sum(when($"tipo_evento" === "compra",
$"monto").otherwise(0)).as("total_ventas")
  )
  .withColumn("promedio_movil_ventas",
    avg($"total_ventas").over(Window.partitionBy($"es_fin_semana")
      .orderBy($"fecha")
      .rowsBetween(-6, 0)))
```

   - `withColumn("fecha", to_date($"timestamp"))`: Crea una nueva columna `fecha` extrayendo solo la fecha del `timestamp`.
   - `withColumn("hora", hour($"timestamp"))`: Crea una nueva columna `hora` extrayendo la hora del `timestamp`.
   - `withColumn("es_fin_semana", when(dayofweek($"fecha").isin(1, 7), true).otherwise(false))`: Crea una columna booleana `es_fin_semana` que es `true` si la fecha es sábado o domingo.
   - `groupBy($"fecha", $"es_fin_semana")`: Agrupa los datos por fecha y si es fin de semana o no.
   - `agg(countDistinct($"usuario_id").as("usuarios_unicos"), sum(when($"tipo_evento" === "compra", $"monto").otherwise(0)).as("total_ventas"))`: Calcula el número de usuarios únicos y la suma de los montos de las compras para cada grupo.
   - `withColumn("promedio_movil_ventas", avg($"total_ventas").over(Window.partitionBy($"es_fin_semana").orderBy($"fecha").rowsBetween(-6, 0)))`: Calcula el promedio móvil de 7 días de las ventas totales para cada grupo de fin de semana.

4. **Mostrar el resultado**:

```
dfProcesado.show()
```

Muestra el DataFrame resultante.

1. **Crear un DataFrame simulado para `df1`**:

```
val df1 = Seq(
  (1, "2023-02-01", "Electronica", 1000.0),
  (2, "2023-03-01", "Hogar", 500.0),
  (3, "2022-12-01", "Electronica", 700.0),
  (4, "2023-05-01", "Ropa", 1500.0)
).toDF("id", "fecha", "categoria", "ventas")
```

   Esto crea un DataFrame `df1` con columnas `id`, `fecha`, `categoria` y `ventas`.

2. **Crear un DataFrame simulado para `df2`**:

```
val df2 = Seq(
  (1, "Descuento"),
  (2, "Promoción"),
  (4, "Descuento"),
  (5, "Ofertas")
).toDF("id", "tipo_promocion")
```

   Esto crea un DataFrame `df2` con columnas `id` y `tipo_promocion`.

3. **Realizar el join entre `df1` y `df2` usando broadcast para `df2`**:

```
val resultado = df1.join(broadcast(df2), Seq("id"))
  .filter($"fecha" > "2023-01-01")
  .groupBy($"categoria")
  .agg(sum($"ventas").as("total_ventas"))
  .orderBy($"total_ventas".desc)
```

   - `join(broadcast(df2), Seq("id"))`: Realiza una unión entre `df1` y `df2` en la columna `id`, utilizando `broadcast` para optimizar la unión si `df2` es pequeño.
   - `filter($"fecha" > "2023-01-01")`: Filtra las filas donde la fecha es posterior al 1 de enero de 2023.
   - `groupBy($"categoria")`: Agrupa los datos por la columna `categoria`.
   - `agg(sum($"ventas").as("total_ventas"))`: Suma las ventas dentro de cada grupo y almacena el resultado en `total_ventas`.
   - `orderBy($"total_ventas".desc)`: Ordena los resultados por `total_ventas` en orden descendente.

4. **Analizar el plan de ejecución**:

```
resultado.explain(true)
```

   Muestra el plan de ejecución detallado para entender cómo Spark procesará la consulta.

5. **Materializar el resultado**:

```
resultado.show()
```

   Muestra el DataFrame resultante en pantalla.

1. **Crear un DataFrame simulado para `df`**:

```
val df = Seq(
  (1, "Electronica", 1000.0),
  (2, "Hogar", 500.0),
  (3, "Electronica", 700.0),
  (4, "Ropa", 1500.0)
).toDF("id", "categoria", "ventas")
```

   Esto crea un DataFrame `df` con columnas `id`, `categoria` y `ventas`.

2. **Cachear el DataFrame en memoria y disco**:

```
df.persist(StorageLevel.MEMORY_AND_DISK_SER)
```

   Persiste el DataFrame en memoria y disco utilizando almacenamiento serializado. Esto es útil para mejorar el rendimiento cuando se realizan múltiples operaciones sobre el mismo DataFrame.

3. **Realizar múltiples operaciones sobre el DataFrame cacheado**:

```
val resultado1 = df.groupBy("categoria").agg(sum("ventas").as("total_ventas"))
val resultado2 = df.join(df, Seq("id"))
```

   - `resultado1`: Agrupa los datos por la columna `categoria` y suma las ventas dentro de cada grupo, almacenando el resultado en `total_ventas`.
   - `resultado2`: Realiza una unión del DataFrame consigo mismo basado en la columna `id`.

4. **Mostrar los resultados de las operaciones**:

```
resultado1.show()
resultado2.show()
```

   Muestra los DataFrames resultantes en pantalla.

5. **Liberar el caché cuando ya no sea necesario**:

```
df.unpersist()
```

   Libera la memoria y el espacio en disco ocupados por el caché del DataFrame.

1. **Crear un DataFrame de streaming usando la fuente `rate`**:

```
val streamingDF = spark.readStream
  .format("rate")
  .option("rowsPerSecond", "10")
  .load()
  .selectExpr("CAST(timestamp AS STRING)", "value AS id")
```

   Esto crea un DataFrame de streaming que genera 10 filas por segundo. La columna `timestamp` se convierte a cadena y la columna `value` se renombra a `id`.
2. **Agregar una columna que simule tipos de eventos aleatorios**:

```
val simulatedEventsDF = streamingDF
  .withColumn("tipo_evento", expr("CASE WHEN rand() < 0.5 THEN 'click' ELSE 'view'
END"))
  .withColumn("timestamp", $"timestamp".cast("timestamp"))
```

   - `withColumn("tipo_evento", expr("CASE WHEN rand() < 0.5 THEN 'click' ELSE 'view' END"))`: Agrega una columna `tipo_evento` que asigna aleatoriamente el valor 'click' o 'view'.
   - `withColumn("timestamp", $"timestamp".cast("timestamp"))`: Convierte la columna `timestamp` a tipo `timestamp`.
3. **Procesar el stream**:

```
val procesadoDF = simulatedEventsDF
  .withWatermark("timestamp", "10 minutes")
  .groupBy(window($"timestamp", "5 minutes"), $"tipo_evento")
  .agg(count("*").as("total_eventos"))
```

   - `withWatermark("timestamp", "10 minutes")`: Configura un watermark para manejar datos retrasados.
   - `groupBy(window($"timestamp", "5 minutes"), $"tipo_evento")`: Agrupa los datos en ventanas de 5 minutos y por tipo de evento.
   - `agg(count("*").as("total_eventos"))`: Cuenta el número de eventos en cada grupo.
4. **Especificar el lugar y la forma de salida del stream**:

```
val query = procesadoDF.writeStream
  .outputMode("update")
  .format("console")
  .trigger(Trigger.ProcessingTime("10 seconds"))
  .start()
```

   - `outputMode("update")`: Modo de salida que muestra solo los resultados nuevos o actualizados.
   - `format("console")`: Especifica que la salida se mostrará en la consola.
   - `trigger(Trigger.ProcessingTime("10 seconds"))`: Configura el trigger para procesar los datos cada 10 segundos.
5. **Mantener la aplicación en ejecución hasta que se detenga manualmente**:

```
query.awaitTermination()
```

Mantiene la aplicación en ejecución hasta que se detenga manualmente.

1. **Crear un DataFrame simulado**:

```
val df = Seq(
  (1, "2023-02-01", "Electronica", 1000.0),
  (2, "2023-03-01", "Hogar", 500.0),
  (3, "2023-02-01", "Electronica", 700.0),
  (4, "2023-05-01", "Ropa", 1500.0)
).toDF("id", "fecha", "categoria", "ventas")
```

   Esto crea un DataFrame `df` con columnas `id`, `fecha`, `categoria` y `ventas`.

2. **Particionamiento por una columna**:

```
val dfParticionado = df.repartition($"fecha")
```

   Reparte las particiones del DataFrame basado en la columna `fecha`.

3. **Particionamiento por múltiples columnas con un número específico de particiones**:

```
val dfMultiParticionado = df.repartition(200, $"fecha", $"categoria")
```

   Reparte el DataFrame en 200 particiones basadas en las columnas `fecha` y `categoria`.

4. **Escritura particionada**:

```
dfParticionado.write
  .partitionBy("fecha", "categoria")
  .bucketBy(10, "id")
  .saveAsTable("tabla_optimizada")
```

   - `partitionBy("fecha", "categoria")`: Escribe los datos particionados por `fecha` y `categoria`.
   - `bucketBy(10, "id")`: Distribuye los datos en 10 "buckets" basados en la columna `id`.
   - `saveAsTable("tabla_optimizada")`: Guarda los datos como una tabla particionada y bucketizada llamada `tabla_optimizada`.

Qué es KryoSerializer?

KryoSerializer es una biblioteca de serialización que Spark puede utilizar para convertir objetos en una secuencia de bytes y viceversa. Es conocida por ser más eficiente y rápida que la serialización predeterminada de Java.

Ventajas de KryoSerializer

1. **Rendimiento Mejorado**: KryoSerializer es más rápida y genera un tamaño de datos serializados más pequeño en comparación con la serialización predeterminada de Java. Esto reduce el tiempo de transferencia de datos entre nodos y mejora el rendimiento general de la aplicación.
2. **Menor Uso de Memoria**: Al generar datos serializados más pequeños, KryoSerializer ayuda a reducir el uso de memoria, lo que es crucial para aplicaciones que manejan grandes volúmenes de datos.
3. **Configuración Sencilla**: Es fácil de configurar en Spark, como se muestra en tu código.

Cómo Configurar KryoSerializer en Spark

En tu código, has configurado KryoSerializer de la siguiente manera:

```
val spark = SparkSession.builder()
  .appName("ConfigurationTuningExample")
  .master("local[*]")
  .config("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
  .getOrCreate()
```

Esta línea:

```
.config("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
```

indica a Spark que utilice KryoSerializer en lugar de la serialización predeterminada de Java.

Registro de Clases (Opcional pero Recomendado)

Para obtener el máximo rendimiento, es recomendable registrar las clases que se van a serializar. Esto se hace para que Kryo pueda manejar estas clases de manera más eficiente:

```
val sparkConf = new SparkConf()
sparkConf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
sparkConf.registerKryoClasses(Array(classOf[TuClase], classOf[OtraClase]))
val spark = SparkSession.builder().config(sparkConf).getOrCreate()
```

En este ejemplo, `TuClase` y `OtraClase` son las clases que se registran para la serialización.

Conclusión

Utilizar KryoSerializer en Spark puede mejorar significativamente el rendimiento de las aplicaciones, especialmente cuando se trabaja con grandes volúmenes de datos. Es una optimización sencilla que puede tener un gran impacto en la eficiencia de la serialización y deserialización de objetos.

¿Qué es la Especulación en Spark?

La especulación es una técnica que permite a Spark detectar y mitigar el impacto de tareas lentas en un trabajo distribuido. Cuando una tarea tarda mucho más en completarse en comparación con otras tareas del mismo trabajo, Spark puede lanzar una copia especulativa de esa tarea en otro nodo. La primera copia de la tarea que se complete se utiliza, y las demás se descartan.

¿Cómo Funciona?

1. **Detección de Tareas Lentas**: Spark monitorea el progreso de todas las tareas en un trabajo. Si detecta que una tarea está tardando significativamente más que las demás, la marca como una tarea lenta.
2. **Lanzamiento de Copias Especulativas**: Spark lanza una copia especulativa de la tarea lenta en otro nodo. Esta copia se ejecuta en paralelo con la tarea original.
3. **Uso del Resultado Más Rápido**: La primera copia de la tarea que se complete se utiliza para el trabajo, y las demás copias se descartan.

Configuración de la Especulación

En tu código, has activado la especulación con la siguiente configuración:

```
.config("spark.speculation", "true")
```

Esto indica a Spark que habilite la especulación para todas las tareas en el trabajo.

Beneficios de la Especulación

- **Mejora del Rendimiento**: Ayuda a reducir el tiempo total de ejecución de un trabajo al mitigar el impacto de tareas lentas.
- **Mayor Robustez**: Aumenta la robustez del trabajo distribuido al asegurar que las tareas lentas no afecten significativamente el rendimiento general.

Consideraciones

- **Uso de Recursos**: La especulación puede aumentar el uso de recursos, ya que se lanzan copias adicionales de las tareas.
- **Configuración Adicional**: Puedes ajustar otros parámetros relacionados con la especulación, como el porcentaje de tareas que deben completarse antes de que se active la especulación (`spark.speculation.quantile`) y el tiempo mínimo que una tarea debe estar en ejecución antes de que se considere para la especulación (`spark.speculation.multiplier`).

Ejemplo de Configuración Adicional

```
.config("spark.speculation.quantile", "0.75")
```

```
.config("spark.speculation.multiplier", "1.5")
```

- `spark.speculation.quantile`: Especifica el porcentaje de tareas que deben completarse antes de que se active la especulación (por defecto es 0.75, es decir, 75%).
- `spark.speculation.multiplier`: Especifica el factor por el cual una tarea debe ser más lenta que la mediana para ser considerada para la especulación (por defecto es 1.5).

La especulación es una herramienta poderosa para mejorar el rendimiento y la robustez de las aplicaciones Spark, especialmente en entornos distribuidos donde las tareas pueden experimentar variabilidad en el tiempo de ejecución.

1. **URL del dataset de Iris**:

```
val url = "https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data"
```

   Especifica la URL desde donde se descargará el dataset de Iris.
2. **Descargar el dataset de Iris**:

```
val irisData = Source.fromURL(url).mkString
```

   Descarga el contenido del dataset desde la URL y lo convierte en una cadena.
3. **Guardar el dataset en un archivo temporal**:

```
val tempFile = new File("iris.csv")
new PrintWriter(tempFile) { write(irisData); close() }
```

   Guarda el contenido descargado en un archivo temporal llamado `iris.csv`.
4. **Especificar los nombres de las columnas**:

```
val irisSchema = Seq("sepal_length", "sepal_width", "petal_length", "petal_width",
"species")
```

   Define los nombres de las columnas del dataset, ya que el archivo original no tiene cabecera.
5. **Leer el CSV usando Spark**:

```
val dfCSV = spark.read
  .option("header", "false")
  .option("inferSchema", "true")
  .option("dateFormat", "yyyy-MM-dd")
  .csv(tempFile.getAbsolutePath)
```

   - `option("header", "false")`: Indica que el archivo no tiene cabecera.
   - `option("inferSchema", "true")`: Indica que Spark debe inferir el esquema automáticamente.
   - `option("dateFormat", "yyyy-MM-dd")`: Configura el formato de fecha (aunque no aplica en este dataset).
   - `csv(tempFile.getAbsolutePath)`: Lee el archivo CSV.
6. **Asignar los nombres de las columnas**:

```
val dfIris = dfCSV.toDF(irisSchema: _*)
```

   Asigna los nombres de las columnas definidos anteriormente al DataFrame.
7. **Modificar el DataFrame**:

```
val dfModificado = dfIris.withColumn("sepal_ratio", $"sepal_length" /
$"sepal_width")
```

   Añade una nueva columna `sepal_ratio` que calcula la relación entre la longitud y el ancho del sépalo.

8. **Mostrar la modificación realizada**:

```
dfModificado.show(10)
```

Muestra las primeras 10 filas del DataFrame modificado

1. **URL del archivo de texto a descargar**:

```
val url = "https://www.gutenberg.org/files/11/11-0.txt"  // "Alice's Adventures in Wonderland" de Lewis Carroll
```

   Especifica la URL desde donde se descargará el archivo de texto.
2. **Descargar el archivo de texto con la codificación UTF-8**:

```
implicit val codec: Codec = Codec.UTF8
val textData = Source.fromURL(url).reader()
```

   Descarga el contenido del archivo de texto desde la URL y lo convierte en un `Reader` utilizando la codificación UTF-8.
3. **Configurar HDFS**:

```
val hdfsUri = "hdfs://localhost:9000"
val hdfsInputPath = s"$hdfsUri/curso/alice.txt"
val hdfsOutputPath = s"$hdfsUri/curso/alice_processed.txt"
```

   Define la URI de HDFS y las rutas de entrada y salida para los archivos en HDFS.
4. **Obtener el sistema de archivos HDFS**:

```
val hadoopConf = new Configuration()
hadoopConf.set("fs.defaultFS", hdfsUri)
val hdfs = FileSystem.get(hadoopConf)
```

   Configura y obtiene el sistema de archivos HDFS.
5. **Crear un archivo en HDFS para escribir en él**:

```
val outputStream: FSDataOutputStream = hdfs.create(new Path(hdfsInputPath))
```

   Crea un archivo en HDFS para escribir el contenido descargado.
6. **Leer y subir el archivo línea por línea**:

```
val bufferedReader = new BufferedReader(textData)
var line: String = null
while ({ line = bufferedReader.readLine(); line != null }) {
  outputStream.writeBytes(line + "\n")
}
bufferedReader.close()
outputStream.close()
println(s"Archivo subido a HDFS: $hdfsInputPath")
```

   Lee el archivo línea por línea y lo escribe en HDFS para evitar transferencias grandes.
7. **Lectura del archivo de texto desde HDFS**:

```
val dfHDFS = spark.read.textFile(hdfsInputPath)
```

   Lee el archivo de texto desde HDFS en un DataFrame de Spark.
8. **Mostrar algunas líneas del archivo leído desde HDFS**:

```
dfHDFS.show(10, truncate = false)
```

Muestra las primeras 10 líneas del archivo leído desde HDFS.

9. **Escritura del DataFrame en un archivo de texto en HDFS**:

```
dfHDFS.write.text(hdfsOutputPath)
println(s"Archivo de texto procesado guardado en HDFS: $hdfsOutputPath")
```

Escribe el DataFrame en un archivo de texto en HDFS.

1. **URL del archivo Parquet a descargar**:

```
val url = "https://d37ci6vzurychx.cloudfront.net/trip-data/yellow_tripdata_2022-
01.parquet"
```

   Especifica la URL desde donde se descargará el archivo Parquet.
2. **Descargar el archivo Parquet**:

```
val localFilePath = "yellow_tripdata_2022-01.parquet"
downloadFile(url, localFilePath)
println(s"Archivo Parquet descargado en: $localFilePath")
```

   Descarga el archivo Parquet desde la URL y lo guarda en una ruta local.
3. **Leer el archivo Parquet local**:

```
val dfParquetLocal = spark.read.parquet(localFilePath)
println("DataFrame leido del archivo Parquet local:")
dfParquetLocal.show(5)
println(s"Numero de filas: ${dfParquetLocal.count()}")
println("Esquema del DataFrame:")
dfParquetLocal.printSchema()
```

   - Lee el archivo Parquet local en un DataFrame de Spark.
   - Muestra las primeras 5 filas del DataFrame.
   - Imprime el número de filas y el esquema del DataFrame.
4. **Configurar HDFS**:

```
val hdfsUri = "hdfs://localhost:9000"
val hdfsInputPath = s"$hdfsUri/curso/yellow_taxi_2022-01.parquet"
val hdfsOutputPath = s"$hdfsUri/curso/yellow_taxi_2022-01_processed.parquet"
```

   Define la URI de HDFS y las rutas de entrada y salida para los archivos en HDFS.
5. **Obtener el sistema de archivos HDFS**:

```
val hadoopConf = new Configuration()
hadoopConf.set("fs.defaultFS", hdfsUri)
val hdfs = FileSystem.get(hadoopConf)
```

   Configura y obtiene el sistema de archivos HDFS.
6. **Copiar el archivo Parquet a HDFS**:

```
hdfs.copyFromLocalFile(new Path(localFilePath), new Path(hdfsInputPath))
println(s"Archivo Parquet copiado a HDFS: $hdfsInputPath")
```

   Copia el archivo Parquet desde la ruta local a HDFS.
7. **Lectura del archivo Parquet desde HDFS**:

```
val dfParquetHDFS = spark.read.parquet(hdfsInputPath)
println("DataFrame leido desde HDFS (Parquet):")
```

```
dfParquetHDFS.show(5)
```

Lee el archivo Parquet desde HDFS en un DataFrame de Spark y muestra las primeras 5 filas.

8. **Realizar una operación simple: filtrar las filas donde `trip_distance` > 5.0**:

```
val dfFiltered = dfParquetHDFS.filter($"trip_distance" > 5.0)
println("DataFrame filtrado (viajes con distancia > 5.0 millas):")
dfFiltered.show(5)
```

Filtra las filas del DataFrame donde la distancia del viaje (`trip_distance`) es mayor a 5.0 millas y muestra las primeras 5 filas del DataFrame filtrado.

9. **Guardar el DataFrame filtrado en un nuevo archivo Parquet en HDFS**:

```
dfFiltered.write.parquet(hdfsOutputPath)
println(s"DataFrame filtrado guardado en HDFS: $hdfsOutputPath")
```

Guarda el DataFrame filtrado en un nuevo archivo Parquet en HDFS.

1. **Definición de la función**:

```
def downloadFile(url: String, localFilePath: String): Unit = {
```

La función `downloadFile` toma dos parámetros: `url` (la URL del archivo a descargar) y `localFilePath` (la ruta local donde se guardará el archivo).

2. **Abrir la conexión y los flujos de entrada y salida**:

```
val connection = new URL(url).openConnection()
val inputStream = connection.getInputStream
val outputStream = new FileOutputStream(new File(localFilePath))
```

- `connection`: Abre una conexión a la URL especificada.
- `inputStream`: Obtiene un flujo de entrada desde la conexión.
- `outputStream`: Crea un flujo de salida para escribir el archivo en la ruta local.

3. **Leer y escribir el archivo en bloques**:

```
try {
  val buffer = new ArrayByte
  var bytesRead = inputStream.read(buffer)
  while (bytesRead != -1) {
    outputStream.write(buffer, 0, bytesRead)
    bytesRead = inputStream.read(buffer)
  }
} finally {
  inputStream.close()
  outputStream.close()
}
```

- `buffer`: Crea un búfer de 4096 bytes para leer el archivo en bloques.

- `bytesRead = inputStream.read(buffer)`: Lee datos en el búfer desde el flujo de entrada.
- `while (bytesRead != -1)`: Continúa leyendo mientras haya datos disponibles.
- `outputStream.write(buffer, 0, bytesRead)`: Escribe los datos leídos en el flujo de salida.
- `inputStream.close()` y `outputStream.close()`: Cierra los flujos de entrada y salida en el bloque `finally` para asegurar que se cierren incluso si ocurre una excepción.

Esta función es útil para descargar archivos de manera eficiente y segura, asegurando que los recursos se liberen adecuadamente después de la operación.

1. **Crear un DataFrame de ejemplo**:

```
val data = Seq(
  ("Alice", "Sales", 3000),
  ("Bob", "Sales", 4000),
  ("Charlie", "Marketing", 4500),
  ("David", "Sales", 3500),
  ("Eve", "Marketing", 3800)
).toDF("name", "department", "salary")
```

   Esto crea un DataFrame `data` con columnas `name`, `department` y `salary`.

2. **Definir una ventana particionada por departamento y ordenada por salario**:

```
val windowSpec = Window.partitionBy("department").orderBy("salary")
```

   Define una ventana de particionamiento que agrupa los datos por `department` y los ordena por `salary`.

3. **Calcular el rango y el salario acumulado dentro de cada departamento**:

```
val result = data.withColumn("rank", rank().over(windowSpec))
                 .withColumn("cumulative_salary", sum("salary").over(windowSpec))
```

   - `withColumn("rank", rank().over(windowSpec))`: Añade una columna `rank` que calcula el rango de cada empleado dentro de su departamento basado en el salario.
   - `withColumn("cumulative_salary", sum("salary").over(windowSpec))`: Añade una columna `cumulative_salary` que calcula el salario acumulado dentro de cada departamento.

4. **Mostrar el resultado**:

```
result.show()
```

   Muestra el DataFrame resultante con las nuevas columnas `rank` y `cumulative_salary`.

Este código es útil para realizar análisis avanzados utilizando ventanas de particionamiento en Spark, lo que permite calcular métricas como el rango y el salario acumulado dentro de grupos específicos.

Este código está escrito en Scala y utiliza Apache Spark para definir y utilizar una UDF (User Defined Function) para categorizar salarios. Aquí tienes un desglose de lo que hace cada parte del código:

1. **Definir una UDF para categorizar salarios**:

```
val categorizeSalary = udf((salary: Int) => {
  if (salary < 3500) "Low"
  else if (salary < 4500) "Medium"
  else "High"
})
```

   Define una UDF llamada `categorizeSalary` que toma un salario como entrada y devuelve una categoría de salario ("Low", "Medium" o "High") basada en el valor del salario.
2. **Registrar la UDF para uso en SQL**:

```
spark.udf.register("categorizeSalary", categorizeSalary)
```

   Registra la UDF `categorizeSalary` para que pueda ser utilizada en consultas SQL.
3. **Crear un DataFrame de ejemplo**:

```
val data = Seq(
  ("Alice", 3000),
  ("Bob", 4000),
  ("Charlie", 5000)
).toDF("name", "salary")
```

   Crea un DataFrame `data` con columnas `name` y `salary`.
4. **Usar la UDF en una transformación de DataFrame**:

```
val result = data.withColumn("salary_category", categorizeSalary($"salary"))
result.show()
```

   - `withColumn("salary_category", categorizeSalary($"salary"))`: Añade una nueva columna `salary_category` al DataFrame `data` utilizando la UDF `categorizeSalary`.
   - `result.show()`: Muestra el DataFrame resultante con la nueva columna `salary_category`.
5. **Usar la UDF en una consulta SQL**:

```
data.createOrReplaceTempView("employees")
spark.sql("SELECT name, salary, categorizeSalary(salary) as salary_category FROM
employees").show()
```

   - `data.createOrReplaceTempView("employees")`: Crea una vista temporal llamada `employees` a partir del DataFrame `data`.
   - `spark.sql("SELECT name, salary, categorizeSalary(salary) as salary_category FROM employees").show()`: Ejecuta una consulta SQL que selecciona las columnas `name`, `salary` y la categoría de salario calculada utilizando la UDF `categorizeSalary`, y muestra el resultado.

Este código demuestra cómo definir y utilizar una UDF en Apache Spark tanto en transformaciones de DataFrame como en consultas SQL.

Este código está escrito en Scala y utiliza Apache Spark para trabajar con datos complejos, incluyendo arrays y mapas. Aquí tienes un desglose de lo que hace cada parte del código:

1. **Crear un DataFrame con datos complejos**:

```
val data = Seq(
  (1, Array("apple", "banana"), Map("a" -> 1, "b" -> 2)),
  (2, Array("orange", "grape"), Map("x" -> 3, "y" -> 4))
).toDF("id", "fruits", "scores")
```

   Esto crea un DataFrame `data` con columnas `id`, `fruits` (un array de cadenas) y `scores` (un mapa de cadenas a enteros).
2. **Operaciones con arrays**:

```
val withArrayLength = data.withColumn("fruit_count", size($"fruits"))
```

   Añade una nueva columna `fruit_count` que contiene el tamaño del array `fruits` utilizando la función `size`.
3. **Operaciones con mapas**:

```
val withMapValue = data.withColumn("score_a", $"scores".getItem("a"))
```

   Añade una nueva columna `score_a` que extrae el valor asociado con la clave "a" en el mapa `scores` utilizando la función `getItem`.
4. **Explotar (explode) un array**:

```
val explodedFruits = data.select($"id", explode($"fruits").as("fruit"))
```

   Utiliza la función `explode` para descomponer el array `fruits` en múltiples filas, creando una nueva columna `fruit` que contiene cada elemento del array.
5. **Mostrar los resultados**:

```
withArrayLength.show()
withMapValue.show()
explodedFruits.show()
```

   Muestra los DataFrames resultantes de las operaciones anteriores.

Este código demuestra cómo trabajar con estructuras de datos complejas en Spark, como arrays y mapas, y cómo realizar operaciones comunes como calcular el tamaño de un array, extraer valores de un mapa y descomponer arrays en múltiples filas.

Este código está escrito en Scala y utiliza Apache Spark junto con HDFS para descargar, leer, procesar y guardar un archivo Parquet. Aquí tienes un desglose de lo que hace cada parte del código:

1. **URL del archivo Parquet a descargar**:

```
val url = "https://d37ci6vzurychx.cloudfront.net/trip-data/yellow_tripdata_2022-01.parquet"
```

   Especifica la URL desde donde se descargará el archivo Parquet.

2. **Descargar el archivo Parquet**:

```
val localFilePath = "yellow_tripdata_2022-01.parquet"
downloadFile(url, localFilePath)
println(s"Archivo Parquet descargado en: $localFilePath")
```

   Descarga el archivo Parquet desde la URL y lo guarda en una ruta local.

3. **Leer el archivo Parquet**:

```
val dfTaxi = spark.read.parquet(localFilePath)
dfTaxi.printSchema()
dfTaxi.describe().show()
```

   - Lee el archivo Parquet local en un DataFrame de Spark.
   - Muestra el esquema del DataFrame y algunas estadísticas básicas.

4. **Cachear el DataFrame para mejorar el rendimiento de operaciones subsecuentes**:

```
dfTaxi.persist(StorageLevel.MEMORY_AND_DISK)
```

   Persiste el DataFrame en memoria y disco para mejorar el rendimiento de las operaciones subsecuentes.

5. **Realizar algunas transformaciones y consultas**:

```
val dfProcessed = dfTaxi
  .withColumn("trip_duration_minutes",
    (unix_timestamp($"tpep_dropoff_datetime") - unix_timestamp($"tpep_pickup_datetime")) / 60)
  .withColumn("price_per_mile", when($"trip_distance" > 0, $"total_amount" / $"trip_distance").otherwise(0))
  .withColumn("day_of_week", date_format($"tpep_pickup_datetime", "EEEE"))
```

   - Añade una columna `trip_duration_minutes` que calcula la duración del viaje en minutos.
   - Añade una columna `price_per_mile` que calcula el precio por milla.
   - Añade una columna `day_of_week` que extrae el día de la semana de la fecha de recogida.

6. **Crear una vista temporal para usar SQL**:

```
dfProcessed.createOrReplaceTempView("taxi_trips")
```

7. **Realizar una consulta SQL compleja**:

```
val resultSQL = spark.sql("""
  SELECT
    day_of_week,
    AVG(trip_duration_minutes) as avg_duration,
    AVG(trip_distance) as avg_distance,
    AVG(total_amount) as avg_amount,
    AVG(price_per_mile) as avg_price_per_mile
  FROM taxi_trips
  WHERE trip_distance > 0 AND trip_duration_minutes BETWEEN 5 AND 120
  GROUP BY day_of_week
  ORDER BY avg_amount DESC
""")
resultSQL.show()
```

8. **Análisis adicional usando la API de DataFrame**:

```
val topPickupLocations = dfProcessed
  .groupBy("PULocationID")
  .agg(
    count("*").as("total_pickups"),
    avg("total_amount").as("avg_fare")
  )
  .orderBy(desc("total_pickups"))
  .limit(5)
topPickupLocations.show()
```

9. **Guardar resultados en formato Parquet**:

```
resultSQL.write.mode("overwrite").parquet("taxi_summary_by_day.parquet")
topPickupLocations.write.mode("overwrite").parquet("top_pickup_locations.parquet")
```

10. **Liberar el caché**:

```
dfTaxi.unpersist()
```

11. **Detener la sesión de Spark**:

```
spark.stop()
```

12. **Definición de la función `downloadFile`**:

```
def downloadFile(url: String, localFilePath: String): Unit = {
  val connection = new URL(url).openConnection()
  val inputStream = connection.getInputStream
  val outputStream = new FileOutputStream(new File(localFilePath))

  try {
    val buffer = new ArrayByte
    var bytesRead = inputStream.read(buffer)
    while (bytesRead != -1) {
      outputStream.write(buffer, 0, bytesRead)
      bytesRead = inputStream.read(buffer)
    }
  } finally {
```

```
    inputStream.close()
    outputStream.close()
  }
}
```

Este código demuestra cómo descargar, leer, procesar y guardar datos en formato Parquet utilizando Apache Spark y HDFS.

Este código está escrito en Scala y utiliza Apache Spark para descargar, leer, procesar y analizar un conjunto de datos de rutas de vuelos. Aquí tienes un desglose de lo que hace cada parte del código:

1. **URL del conjunto de datos de vuelos**:

```
val url =
"https://raw.githubusercontent.com/jpatokal/openflights/master/data/routes.dat"
```

   Especifica la URL desde donde se descargará el archivo CSV con los datos de rutas de vuelos.

2. **Descargar el archivo CSV**:

```
val localFilePath = "routes.csv"
downloadFile(url, localFilePath)
println(s"Archivo CSV descargado en: $localFilePath")
```

   Descarga el archivo CSV desde la URL y lo guarda en una ruta local.

3. **Leer el archivo CSV**:

```
val dfRoutes = spark.read
  .option("header", "false")
  .option("inferSchema", "true")
  .csv(localFilePath)
  .toDF("airline", "airline_id", "source_airport", "source_airport_id",
        "destination_airport", "destination_airport_id", "codeshare",
        "stops", "equipment")
dfRoutes.printSchema()
dfRoutes.show(5)
```

   - Lee el archivo CSV en un DataFrame de Spark.
   - Asigna nombres a las columnas del DataFrame.
   - Muestra el esquema del DataFrame y las primeras 5 filas.

4. **Persistir el DataFrame en memoria y disco**:

```
dfRoutes.persist(StorageLevel.MEMORY_AND_DISK)
```

5. **Definir una UDF para categorizar las rutas basadas en el número de paradas**:

```
val categorizeRoute = udf((stops: Int) => {
  if (stops == 0) "Direct"
  else if (stops == 1) "One Stop"
  else "Multiple Stops"
})
```

6. **Aplicar la UDF y realizar algunas transformaciones**:

```
val dfProcessed = dfRoutes
  .withColumn("route_type", categorizeRoute($"stops"))
  .withColumn("is_international",
              when($"source_airport".substr(1, 2) =!=
$"destination_airport".substr(1, 2), true)
```

```
            .otherwise(false))
```

Este fragmento de código realiza transformaciones en el DataFrame `dfRoutes` para añadir dos nuevas columnas: `route_type` e `is_international`. Aquí tienes un desglose de lo que hace cada parte del código:

- **Añadir la columna `route_type`**:

```
.withColumn("route_type", categorizeRoute($"stops"))
```

Utiliza la UDF `categorizeRoute` para categorizar las rutas basadas en el número de paradas (`stops`). La nueva columna `route_type` tendrá valores como "Direct", "One Stop" o "Multiple Stops".

- **Añadir la columna `is_international`**:

```
.withColumn("is_international",
            when($"source_airport".substr(1, 2) =!=
$"destination_airport".substr(1, 2), true)
            .otherwise(false))
```

  - `when($"source_airport".substr(1, 2) =!= $"destination_airport".substr(1, 2), true)`: Comprueba si los dos primeros caracteres del código del aeropuerto de origen (`source_airport`) son diferentes de los del aeropuerto de destino (`destination_airport`). Si son diferentes, la ruta se considera internacional y se asigna `true`.
  - `.otherwise(false)`: Si los códigos de los aeropuertos son iguales, la ruta no es internacional y se asigna `false`.

7. **Crear una vista temporal para usar SQL**:

```
dfProcessed.createOrReplaceTempView("flight_routes")
```

8. **Realizar un análisis complejo usando SQL**:

```
val routeAnalysis = spark.sql("""
  SELECT
    airline,
    route_type,
    COUNT(*) as route_count,
    SUM(CASE WHEN is_international THEN 1 ELSE 0 END) as international_routes
  FROM flight_routes
  GROUP BY airline, route_type
  ORDER BY route_count DESC
""")
routeAnalysis.show()
```

9. **Utilizar funciones de ventana para análisis más avanzado**:

```
val windowSpec = Window.partitionBy("airline").orderBy($"route_count".desc)
val topRoutesByAirline = routeAnalysis
  .withColumn("rank", dense_rank().over(windowSpec))
  .filter($"rank" <= 3)
```

```
  .orderBy($"airline", $"rank")
topRoutesByAirline.show()
```

10. **Realizar un análisis de conectividad de aeropuertos**:

```
val airportConnectivity = dfProcessed
  .groupBy("source_airport")
  .agg(
    countDistinct("destination_airport").as("destinations"),
    sum(when($"is_international", 1).otherwise(0)).as("international_connections")
  )
  .orderBy($"destinations".desc)
airportConnectivity.show()
```

Este fragmento de código realiza un análisis de la conectividad de los aeropuertos utilizando el DataFrame `dfProcessed`. Aquí tienes un desglose de lo que hace cada parte del código:

- **Agrupar por `source_airport`**:

```
.groupBy("source_airport")
```

Agrupa los datos por el aeropuerto de origen (`source_airport`).

- **Calcular agregaciones**:

```
.agg(
  countDistinct("destination_airport").as("destinations"),
  sum(when($"is_international", 1).otherwise(0)).as("international_connections")
)
```

  - `countDistinct("destination_airport").as("destinations")`: Cuenta el número de aeropuertos de destino distintos para cada aeropuerto de origen y lo almacena en una nueva columna llamada `destinations`.
  - `sum(when($"is_international", 1).otherwise(0)).as("international_connections")`: Suma el número de rutas internacionales para cada aeropuerto de origen. Si la ruta es internacional (`is_international` es `true`), se cuenta como 1; de lo contrario, se cuenta como 0. El resultado se almacena en una nueva columna llamada `international_connections`.

- **Ordenar por el número de destinos**:

```
.orderBy($"destinations".desc)
```

Ordena los resultados por el número de destinos (`destinations`) en orden descendente.

11. **Guardar los resultados en formato Parquet**:

```
routeAnalysis.write.mode("overwrite").parquet("route_analysis.parquet")
topRoutesByAirline.write.mode("overwrite").parquet("top_routes_by_airline.parquet"
)
airportConnectivity.write.mode("overwrite").parquet("airport_connectivity.parquet"
)
```

Este fragmento de código guarda los resultados de los análisis en formato Parquet. Aquí tienes un desglose de lo que hace cada parte del código:

- **Guardar el análisis de rutas en formato Parquet**:

```
routeAnalysis.write.mode("overwrite").parquet("route_analysis.parquet")
```

Guarda el DataFrame `routeAnalysis` en un archivo Parquet llamado `route_analysis.parquet`. El modo `overwrite` asegura que si el archivo ya existe, será sobrescrito.

- **Guardar los tipos de rutas principales por aerolínea en formato Parquet**:

```
topRoutesByAirline.write.mode("overwrite").parquet("top_routes_by_airline.parquet"
)
```

Guarda el DataFrame `topRoutesByAirline` en un archivo Parquet llamado `top_routes_by_airline.parquet`. El modo `overwrite` asegura que si el archivo ya existe, será sobrescrito.

- **Guardar el análisis de conectividad de aeropuertos en formato Parquet**:

```
airportConnectivity.write.mode("overwrite").parquet("airport_connectivity.parquet"
)
```

Guarda el DataFrame `airportConnectivity` en un archivo Parquet llamado `airport_connectivity.parquet`. El modo `overwrite` asegura que si el archivo ya existe, será sobrescrito.

Guardar los resultados en formato Parquet es una buena práctica porque Parquet es un formato de almacenamiento columnar que es eficiente en términos de espacio y rendimiento, especialmente para consultas analíticas.

12. **Liberar el caché y detener la sesión de Spark**:

```
dfRoutes.unpersist()
spark.stop()
```

13. **Función auxiliar para descargar el archivo**:

```
1. def downloadFile(url: String, localFilePath: String): Unit = {
2.   val connection = new URL(url).openConnection()
3.   val inputStream = connection.getInputStream
4.   val outputStream = new FileOutputStream(new File(localFilePath))
5.
6.   try {
7.     val buffer = new ArrayByte
8.     var bytesRead = inputStream.read(buffer)
9.     while (bytesRead != -1) {
10.        outputStream.write(buffer, 0, bytesRead)
11.        bytesRead = inputStream.read(buffer)
12.      }
13.    } finally {
14.      inputStream.close()
15.      outputStream.close()
16.    }
```

```
17.  }
```