**1. Explicación sobre la Creación del RDD**

Para procesar los grandes archivos de datos recibidos, el primer paso es cargarlos en la memoria distribuida del clúster de Spark. Esto se logra creando un **RDD.** La creación del RDD se realizaría utilizando el método textFile del SparkContext (el punto de entrada a la funcionalidad de Spark). Suponiendo que los archivos diarios se reciben en un directorio, el comando sería:

rdd\_transacciones\_raw = sc.textFile("ruta”)

Este comando instruye a Spark para que lea los archivos de texto del directorio especificado y cree un RDD. Cada elemento de este RDD inicial será un string que representa una línea de los archivos originales (una transacción en formato CSV, por ejemplo). El RDD se particiona automáticamente para ser procesado en paralelo por los diferentes nodos del clúster.

**2. Transformaciones y Acciones Seleccionadas**

Una vez creado el RDD con los datos en crudo, se aplican una serie de **transformaciones** para prepararlos y analizarlos.

**Transformaciones Propuestas**

1. **Parseo y Estructuración (map):** El RDD inicial contiene líneas de texto. La primera transformación consiste en convertir cada línea en una estructura de datos con la que se pueda trabajar.
   * rdd\_parseado = rdd\_transacciones\_raw.map(lambda linea: linea.split(','))
2. **Filtrado de Registros (filter):** Para detectar fraudes, es crucial filtrar las transacciones que cumplen ciertos criterios sospechosos, como montos inusualmente altos.
   * rdd\_filtrado = rdd\_parseado.filter(lambda transaccion: float(transaccion[2]) > 10000)
3. **Agregación de Datos (map y reduceByKey):** Para generar reportes, es común agregar datos. Por ejemplo, para contar cuántas transacciones realiza cada usuario, se puede crear un RDD de pares clave-valor.
   * # Se crea un par (ID\_de\_usuario, 1) por cada transacción
     1. rdd\_pares = rdd\_parseado.map(lambda transaccion: (transaccion[1], 1))
   * # Se suman todos los '1' para cada usuario (clave)
     1. rdd\_conteo\_por\_usuario = rdd\_pares.reduceByKey(lambda a, b: a + b)

**Acciones Seleccionadas**

1. **collect():** Esta acción recopila todos los elementos del RDD y los devuelve al programa como una lista.
   * resultados\_finales = rdd\_conteo\_por\_usuario.collect()

Nota: Se debe usar con precaución, ya que si el resultado es muy grande puede agotar la memoria del nodo driver.

1. **saveAsTextFile():** Esta acción guarda el contenido del RDD en archivos de texto en un sistema de archivos distribuido en S3 por ejemplo. Es la opción ideal para resultados grandes que se usarán para generar reportes.
   * rdd\_filtrado.saveAsTextFile("ruta/al/reporte/transacciones\_sospechosas/")

**3. Definición de un Job Spark en este Contexto**

Dentro del contexto de Spark, un **Job** es una unidad de computación de alto nivel que se crea en respuesta a una **acción** sobre un RDD.

Cuando se invoca una acción como collect() o saveAsTextFile(), Spark examina el linaje de transformaciones del RDD que se necesita para calcular el resultado final. A partir de este grafo, Spark crea un plan de ejecución y lo lanza como un Job. Este Job se divide en etapas y estas, a su vez, en tareas que se ejecutan en paralelo en los nodos del clúster.

En este caso, si se ejecuta rdd\_conteo\_por\_usuario.collect(), se lanzaría **un Job de Spark** que incluiría todas las transformaciones previas: la lectura del archivo (textFile), el parseo (map), la creación de pares (map) y la agregación (reduceByKey).

**4. Reflexión sobre el Problema**

**Importancia del Procesamiento Distribuido**

El procesamiento distribuido es fundamental en este caso por tres razones clave:

1. **Escalabilidad:** El volumen creciente de datos hace que un sistema local sea insostenible. Spark permite escalar horizontalmente, añadiendo más máquinas al clúster para manejar más datos, en lugar de depender de una única máquina cada vez más potente y costosa.
2. **Velocidad:** Para procesar los datos rápidamente y detectar fraudes, la paralelización es esencial. Spark divide los millones de registros en particiones y las procesa simultáneamente en múltiples nodos, reduciendo drásticamente el tiempo de análisis.
3. **Tolerancia a Fallos:** Los RDDs son resilientes. Si un nodo del clúster falla durante el cómputo, Spark puede reconstruir automáticamente la partición de datos perdida a partir del linaje de transformaciones, asegurando que el trabajo se complete sin interrupciones.

**Posibles Desafíos Técnicos**

1. **Complejidad de la Infraestructura:** Administrar y configurar un clúster de Spark puede ser complejo. Una solución es utilizar servicios gestionados en la nube (como AWS EMR, Databricks o Google Dataproc) que simplifican esta tarea.
2. **Optimización de Rendimiento:** Escribir código Spark que sea funcional es diferente a escribir código que sea eficiente. El equipo podría enfrentar desafíos como el "data skew" (datos mal distribuidos entre particiones) o el "shuffle" excesivo de datos, que pueden degradar el rendimiento.
3. **Gestión de la Memoria:** Un uso incorrecto de acciones como collect() sobre resultados muy grandes puede causar fallos por falta de memoria en el nodo "driver".