EVALUACIÓN MÓDULO 7

1. Explica qué son las 5V's de Big Data y por qué son fundamentales en el procesamiento de grandes volúmenes de datos.

Las 5V’s de Big Data son un marco conceptual que describe las características clave y los desafíos inherentes a los conjuntos de datos masivos, estos son:

- **Volumen:** Se refiere a la cantidad masiva de datos generados continuamente. Los sistemas tradicionales (RDBMS) no pueden almacenar o procesar eficientemente petabytes y exabytes de datos. Por esta razón, Big Data requiere sistemas distribuidos como HDFS para el almacenamiento estable.
- **Velocidad:** Es la rapidez con la que los datos se generan, ingieren y procesan. Fuentes como IOT, transacciones financieras o redes sociales generan datos en tiempo real o near-real-time, lo que exige frameworks de procesamiento como Spark Streaming o Flink
- **Variedad:** Indica los diferentes formatos y tipos de datos. Los datos pueden ser, tablas estructuras, no estructurados y semiestructurados. Esta diversidad requiere sistemas flexibles como bases de datos NoSQL (MongoDB, Cassandra) o estructuras como DataFrames en Spark.
- **Veracidad:** Hace referencia a la calidad, confiabilidad y precisión de los datos. Los datos pueden venir con ruido, inconsistencia o sesgo que comprometen su utilidad. Es crucial limpiar y validar los datos para que los análisis y modelos de ML sean confiables y precisos.
- **Valor:** Representa la utilidad que se puede extraer de los datos procesados. El objetivo final no es almacenar datos por almacenar, sino transformarlos en información valiosa para la toma de decisiones estratégicas, como detectar fraudes, personalizar recomendaciones o predecir fallas.

Estas 5V's son fundamentales porque definen los requisitos que debe cumplir una arquitectura de Big Data. Ignorar una de estas dimensiones puede llevar al fracaso de un proyecto. Por ejemplo:

- Un sistema que maneje Volumen pero no Velocidad no será adecuado para detección de fraudes en tiempo real.
- Un sistema que maneje Volumen y Velocidad pero no Variedad no podrá procesar datos de sensores IoT (JSON) junto con imágenes de cámaras.
- Sin garantizar la Veracidad, el Valor de los análisis se ve comprometido generando decisiones erróneas basadas en información deficiente.

2. Describe la diferencia principal entre procesamiento batch y procesamiento en tiempo real. Menciona las tecnologías asociadas con cada tipo de procesamiento.

La diferencia principal radica en el tiempo de procesamiento y la latencia entre la generación del dato y la obtención de un resultado.

- El procesamiento por Lotes o Batch, procesa grandes volúmenes de datos que se han almacenado previamente durante un periodo de tiempo (horas, días). La ejecución se realiza en bloques o "lotes" de datos. Tiene tecnologías asociadas como Apache Hadoop MapReduce, Apache Spark (Core) y Motores SQL Batch.
- El procesamiento en Tiempo Real (Streaming) procesa de forma continua a medida que se generan los datos, casi inmediatamente desde que ocurren los eventos. Posee una latencia de milisegundos cuyo objetivo es el monitoreo, alertas, detección de anomalías, dashboards en tiempo real. Tiene tecnologías asociadas como Apache Spark Streaming, Apache Flink, Apache Kafka Streams y Apache Storm entre otros.

La elección dependerá del caso de uso. El Batch es para análisis profundos sobre datos históricos, mientras que el Streaming es para reaccionar a eventos en el momento que suceden. Frameworks como Spark unifican ambas capacidades

3. ¿Qué es un RDD en Apache Spark? ¿Cuál es la ventaja de usar RDDs sobre bases de datos tradicionales para el procesamiento distribuido?

El RDD (Resilient Distributed Dataset) es la estructura de datos fundamental y de más bajo nivel en Apache Spark. Es es una colección inmutable, tolerantes a fallos y distribuida de objetos que puede ser procesada en paralelo con operaciones "*map", "filter", "reduce"* a través de un clúster.

Las ventajas principales sobre RDBMS tradicionales:

- Distribución y paralelismo: los datos se parten en "particiones" y se distribuyen entre los nodos del clúster, permitiendo que las operaciones se ejecuten en paralelo. Una base de datos tradicional monolítica tiene un cuello de botella en un solo servidor.
- Tolerante a los fallos (resilent): reconstruye automáticamente las particiones perdidas usando el lineage, grafo de operaciones que crearon el RDD, es decir, si un nodo falla los datos se re-computan en otro. Las bases de datos tradicionales usan replicación, lo que consume más almacenamiento.
- Flexibilidad de Datos: los RDDs pueden contener cualquier tipo de objeto de Python/java/Scala y no requieren un esquema fijo. Son ideales para datos no estructurados o semiestructurados (logs, texto, JSON crudo). Las base de datos relacionales requieren un esquema rígido definido previamente (DDL)
- Control de Grano Fino: proporcionan control preciso sobre la partición de datos y la ubicación (localidad), optimizando el rendimiento. Las operaciones son transformaciones funcionales (*map, filter, reduce*) que ofrecen más flexibilidad que SQL para lógicas complejas.
- Evaluación Perezosa (Lazy Evaluation): las transformaciones se planifican en un DAG (Directed Acyclic Graph) y se optimizan antes de ejecutarse cuando se llama a una acción (collect, count). Evita cálculos innecesarios y optimiza el procesamiento de manera inherente.

En conclusión Los RDDs son la base que permite a Spark ser un motor de procesamiento distribuido genérico, flexible y tolerante a fallos, superando las limitaciones de escalabilidad y tipo de datos de las bases de datos tradicionales para ciertas cargas de trabajo analíticas y de ETL.

4. Define Apache Spark SQL. ¿Cómo se diferencia de otros motores SQL tradicionales en términos de rendimiento y escalabilidad?

Apache Spark SQL es un módulo de Apache Spark diseñado para procesar datos estructurados y semiestructurados.

Permite ejecutar consultas SQL sobre datos almacenados en diversos formatos y también interactuar con los datos mediante las APIs de DataFrames y Datasets.

Característica	Motor SQL Tradicional (Single-node)	Apache Spark SQL (Distributed)
Arquitectura	**Monolítica**. Ejecuta en un solo servidor. Escala verticalmente (CPU/RAM más potente).	**Distribuida**. Ejecuta en un clúster de múltiples nodos. Escala horizontalmente (agregar más nodos commodity).
Escalabilidad	**Limitada**. Está limitada por la capacidad de hardware de una sola máquina. Llega a un punto donde es prohibitivamente caro o imposible escalar más.	**Masiva.** Puede manejar petabytes de datos distribuyendo el almacenamiento (HDFS, S3) y el procesamiento en decenas, cientos o miles de nodos.
Rendimiento	**Optimizado para OLTP**. Muy rápido para transacciones cortas y consultas puntuales sobre volúmenes pequeños/medios de datos.	Optimizado para OLAP. Más rápido para consultas analíticas complejas (agregaciones, joins) sobre volúmenes masivos de datos, gracias al paralelismo.
Motor de Optimización	**Optimizadores propietarios** (e.g., cost-based optimizer).	**Catalyst Optimizer**. Utiliza un optimizador basado en reglas y costos que transforma y optimiza el plan de ejecución de las consultas, incluyendo *predicate pushdown* y *optimización de joins*.
Procesamiento en Memoria	Generalmente depende del buffer pool en memoria, pero los datos persistentes están en disco.	**Procesamiento en Memoria (Tungsten)**. Puede cachear DataFrames /RDDs completos en memoria para iteraciones ultra-rápidas, evitando E/S a disco.
Tipo de Carga	Transacciones INSERT/UPDATE/DELETE) y consultas simples.	**Análisis y procesamiento por lotes** (ETL) de grandes datasets.

Conclusión: Spark SQL no busca reemplazar a los motores SQL tradicionales para transacciones operacionales (OLTP). Su ventaja radica en la escalabilidad horizontal y el alto rendimiento en consultas analíticas (OLAP) sobre conjuntos de datos extremadamente grandes, integrando además SQL con programas de procesamiento distribuido complejos (ML, Streaming) en una misma plataforma unificada.

Caso Práctico

1. Explica cómo Apache Spark Streaming puede resolver este caso. ¿Qué componentes de Spark serían necesarios?

Apache Spark Streaming es una opción adecuada. Resuelve el problema al permitir el procesamiento de flujos de datos en tiempo real usando el mismo motor y APIs que el procesamiento batch, garantizando consistencia y facilitando el desarrollo.

¿Cómo lo resuelve?

- Ingesta Continua: Lee el flujo continuo de transacciones desde una fuente de streaming como Apache Kafka o un socket.
- Procesamiento de Micro-Lotes: Divide el flujo continuo en pequeños lotes de datos (e.g., cada 1 segundo). Cada lote es tratado como un DataFrame distribuido sobre el cual se pueden ejecutar consultas SQL, operaciones de agregación y transformaciones complejas.
- Detección de Patrones: Aplica lógica de negocio (reglas heurísticas) o modelos de Machine Learning (MLlib) en tiempo real sobre cada micro-lote para puntuar cada transacción y identificar patrones fraudulentos (ej: transacciones consecutivas en ubicaciones geográficamente imposibles, montos anómalos para un cliente).
- Salida de Resultados: Escribe continuamente los resultados (alertas de transacciones fraudulentas) en un sink de salida, como una base de datos (Cassandra, PostgreSQL) para que un sistema de alertas las consuma, un dashboard (Kibana, Grafana) o simplemente los muestra en consola para monitoreo.

Componentes de Spark necesarios:

- ★ Spark Core: Para la ejecución distribuida subyacente.
- ★ Spark Structured Streaming: El módulo clave para el procesamiento del flujo de datos.
- ★ Spark SQL: Para definir y ejecutar consultas sobre el flujo de datos (usando DataFrames).
- ★ Spark MLlib (opcional pero altamente recomendado): Para cargar y aplicar un modelo de machine learning previamente entrenado que puntúe las transacciones en tiempo real.

2. Propón una solución usando DStreams y micro-batching. Describe los pasos para configurar el sistema de procesamiento de datos en tiempo real.

Aunque Structured Streaming es la API más moderna, una solución con la API original de DStreams seguiría estos pasos:

- Ingesta de Datos las transacciones se envían a un sistema de mensajería como Apache Kafka, que actúa como un búfer robusto y escalable.
- Creación del DStream: La aplicación Spark Streaming se conecta al topic de Kafka para crear un DStream. Este DStream representa el flujo de transacciones, que se divide internamente en una secuencia de RDDs
- Procesamiento con Estado para detectar fraudes, se necesitan transformaciones con estado que analicen el comportamiento a lo largo del tiempo.
    - ★ Se utiliza una operación como updateStateByKey o mapWithState para mantener un estado por cada Cliente ID.
    - ★ El estado podría ser un contador de transacciones, la suma de montos en la última hora o las ubicaciones recientes.
- Aplicación de Reglas de Fraude se aplican reglas sobre el estado actualizado. Por ejemplo:
    - ★ "Si un cliente realiza más de 5 transacciones en menos de 10 minutos."
    - ★ "Si la ubicación de una nueva transacción está a más de 500 km de la transacción anterior del mismo cliente en menos de 1 hora."
- Generación de Alertas (Sink): Si una transacción activa una regla de fraude, se envía una alerta a un sistema de destino (Sink), como otro topic de Kafka, una base de datos o un sistema de monitoreo.

3. ¿Qué beneficios tendría la integración con Apache Kafka para la ingesta de datos en este escenario?

La integración con Apache Kafka es crítica y proporciona beneficios fundamentales para un sistema de este tipo.

- Desacoplamiento: Kafka actúa como un buffer o cola de mensajes entre los productores de transacciones y los consumidores . Esto evita que una caída o lentitud del procesador de Spark afecte a los sistemas productores de transacciones.
- Tolerancia a Fallos y Durabilidad: Kafka persiste los mensajes en disco y los replica. Si el job de Spark falla y se reinicia, puede reprocesar los mensajes desde el último punto de control (*offset*) conocido, garantizando que no se pierda ninguna transacción (processing guarantees).
- Escalabilidad Horizontal: Tanto Kafka como Spark pueden escalar horizontalmente. Se pueden agregar más particiones al tópico de Kafka y más ejecutores a Spark para manejar un aumento en el volumen de transacciones.
- Alto Rendimiento y Baja Latencia: Kafka está diseñado para manejar millones de mensajes por segundo con muy baja latencia, lo que se alinea perfectamente con el requisito de "miles de transacciones por segundo".
- Soporte para Múltiples Consumidores: Diferentes sistemas pueden consumir el mismo flujo de transacciones para distintos propósitos.

En resumen, Kafka proporciona la robustez, escalabilidad y confiabilidad necesarias para la ingesta de datos en un sistema crítico de detección de fraudes en tiempo real.

EJERCICIO

1. Implementa una consulta en Spark SQL para analizar un conjunto de datos de transacciones bancarias. La consulta debe identificar las transacciones de mayor valor por cliente, considerando los campos Cliente_ID, Monto, y Fecha.

```
[29] from pyspark.sql import SparkSession
     from pyspark.sql.types import *
     from pyspark.sql.functions import *
     from datetime import datetime, timedelta
     import random

     spark = SparkSession.builder.appName("TransaccionesBancarias").getOrCreate()

     # Definir esquema
     schema = StructType([
         StructField("TransaccionID", StringType(), True),
         StructField("ClienteID", StringType(), True),
         StructField("Monto", DoubleType(), True),
         StructField("Fecha", DateType(), True),
         StructField("TipoTransaccion", StringType(), True),
         StructField("Ubicacion", StringType(), True)
     ])

     # Crear datos de ejemplo (Python list)
     datos_transacciones_str = [
         ("T001", "C001", 15000.50, "2024-01-15", "Transferencia", "Santiago"),
         ("T002", "C001", 2500.00, "2024-01-16", "Retiro", "Santiago"),
         ("T003", "C001", 45000.75, "2024-01-20", "Deposito", "Valparaiso"),
         ("T004", "C002", 8750.25, "2024-01-15", "Transferencia", "Concepcion"),
         ("T005", "C002", 125000.00, "2024-01-18", "Deposito", "Concepcion"),
         ("T006", "C002", 3200.50, "2024-01-22", "Retiro", "Temuco"),
         ("T007", "C003", 67500.30, "2024-01-14", "Transferencia", "La Serena"),
         ("T008", "C003", 890.75, "2024-01-19", "Retiro", "La Serena"),
         ("T009", "C003", 25000.00, "2024-01-25", "Deposito", "Santiago"),
         ("T010", "C004", 180000.90, "2024-01-12", "Deposito", "Antofagasta"),
         ("T011", "C004", 12500.40, "2024-01-17", "Transferencia", "Santiago"),
         ("T012", "C004", 5670.80, "2024-01-23", "Retiro", "Antofagasta"),
         ("T013", "C005", 95000.60, "2024-01-16", "Deposito", "Valdivia"),
         ("T014", "C005", 7850.25, "2024-01-21", "Transferencia", "Puerto Montt"),
         ("T015", "C005", 34500.15, "2024-01-24", "Retiro", "Valdivia")
     ]

     # Convertir strings de fecha a objetos date
     datos_transacciones = [(row[0], row[1], row[2], datetime.strptime(row[3], "%Y-%m-%d").date(), row[4], row[5]) for row in datos_transacciones_str]

     # Crear DataFrame de Spark a partir de la lista y el esquema
     df_transacciones = spark.createDataFrame(datos_transacciones, schema=schema)

     # Crear una vista temporal
     df_transacciones.createOrReplaceTempView("Transacciones")
```

```
# Crear consulta
transaccion_max_por_cliente = spark.sql("""
    SELECT ClienteID, Monto, Fecha
    FROM
        (SELECT ClienteID, Monto, Fecha, ROW_NUMBER() OVER
         (PARTITION BY  ClienteID ORDER BY Monto DESC     ) as rn
        FROM Transacciones                                  ) ranked_transactions
        WHERE rn = 1                                      """)

transaccion_max_por_cliente.show()
```

```
+---------+--------+----------+
|ClienteID|   Monto|     Fecha|
+---------+--------+----------+
|     C001|45000.75|2024-01-20|
|     C002|125000.0|2024-01-18|
|     C003| 67500.3|2024-01-14|
|     C004|180000.9|2024-01-12|
|     C005| 95000.6|2024-01-16|
+---------+--------+----------+
```

Explica cómo optimizamos esta consulta para manejar grandes volúmenes de datos.

1. **Particionamiento**: El particionamiento distribuye los datos por Cliente_ID en diferentes archivos/directorios, permitiendo que Spark procese solo las particiones relevantes.

   Beneficio: Reduce significativamente la cantidad de datos leídos del disco. En lugar de escanear toda la tabla, solo lee las particiones necesarias.

2. **Caching**: Cache solo cuando los datos se reutilizan múltiples veces en la sesión

   Beneficio: Evita releer datos del disco para operaciones subsecuentes. Crítico cuando los datos se usan repetidamente.

3. **Bucketing**: El bucketing pre-distribuye los datos en "buckets" basados en una columna clave, evitando operaciones costosas de shuffle durante JOINs y GROUP BY

   Beneficio: Elimina el shuffle cuando se agrupa por la columna de bucketing, reduciendo dramáticamente el tiempo de ejecución.

2. Diseña una solución de machine learning usando MLlib para predecir la probabilidad de que una transacción sea fraudulenta, utilizando características como el Monto, Ubicación y Hora de la transacción. Menciona los pasos para entrenar y evaluar el modelo.

Paso 1: Preparación de datos

```
from pyspark.ml.feature import VectorAssembler, StringIndexer
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator

# Crear características vectorizadas
assembler = VectorAssembler(
    inputCols=["Monto", "UbicacionIndex", "HoraDelDia"],
    outputCol="features"
)

# Indexar ubicaciones categóricas
location_indexer = StringIndexer(inputCol="Ubicacion", outputCol="UbicacionIndex")
```

Paso 2: Entrenamiento del modelo

```
# Dividir datos en entrenamiento y prueba
train_data, test_data = df.randomSplit([0.8, 0.2], seed=42)

# Crear y entrenar modelo de regresión logística
lr = LogisticRegression(featuresCol="features", labelCol="EsFraudulenta")
model = lr.fit(train_data)
```

Paso 3: Evaluación del modelo

```
from pyspark.ml import Pipeline

# Crear pipeline completo
pipeline = Pipeline(stages=[location_indexer, assembler, lr])
pipeline_model = pipeline.fit(train_data)
```

Paso 4: Pipeline completo

```
# Hacer predicciones
predictions = model.transform(test_data)

# Evaluar rendimiento
evaluator = BinaryClassificationEvaluator(labelCol="EsFraudulenta")
auc = evaluator.evaluate(predictions)

print(f"AUC: {auc}")
```

3. Proporciona una implementación básica para procesar un flujo de datos en tiempo real usando Structured Streaming. Explica cómo manejarías los eventos fuera de orden y qué técnicas de watermarking emplearías.

Implementación:

Una watermarking es un umbral de tiempo que define cuánto tiempo se esperarán los eventos tardíos.

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *

spark = SparkSession.builder.appName("FraudDetectionStreaming").getOrCreate()

# Schema de transacciones
schema = StructType([
    StructField("ClienteID", StringType(), True),
    StructField("Monto", DoubleType(), True),
    StructField("Ubicacion", StringType(), True),
    StructField("Timestamp", TimestampType(), True)
])

# Stream desde Kafka
df_stream = spark \
    .readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "transactions") \
    .load() \
    .select(from_json(col("value").cast("string"), schema).alias("data")) \
    .select("data.*")
```

```python
# Watermarking para eventos tardíos
df_with_watermark = df_stream \
    .withWatermark("Timestamp", "10 minutes")

# Agregaciones con ventanas deslizantes
windowed_aggregations = df_with_watermark \
    .groupBy(
        window(col("Timestamp"), "5 minutes", "1 minute"),
        col("ClienteID")
    ) \
    .agg(
        sum("Monto").alias("TotalMonto"),
        count("*").alias("NumTransacciones"),
        countDistinct("Ubicacion").alias("UbicacionesUnicas")
    )

# Detección de anomalías
fraud_detection = windowed_aggregations.filter(
    (col("NumTransacciones") > 10) |
    (col("TotalMonto") > 50000) |
    (col("UbicacionesUnicas") > 3)
)
```

```python
# Salida con manejo de eventos tardíos
query = fraud_detection \
    .writeStream \
    .outputMode("update") \
    .format("console") \
    .trigger(processingTime="30 seconds") \
    .option("checkpointLocation", "/tmp/checkpoint") \
    .start()

query.awaitTermination()
```

Técnicas de watermarking empleadas:

- Watermark de tolerancia: 10 minutos para permitir eventos retrasados por latencia de red.
- Ventanas deslizantes: Análisis en ventanas de 5 minutos para detectar patrones de comportamiento anómalo.
- Manejo de estado: Spark mantiene el estado de las ventanas hasta que el watermark las elimina automáticamente.
- Configuración de output mode: Usar "update" que solo muestra las filas “finales” y no se actualizarán más o "append" que muestra todas las actualizaciones que ocurren en cada trigger a medida que llegan nuevos datos (dentro de la watermark) según los requerimientos de la aplicación.
- Checkpointing: Garantiza exactly-once processing ante fallos

En síntesis esta combinación de **ventanas de tiempo, watermark y output mode** permite que Structured Streaming entregue resultados correctos y aproximados en presencia de eventos tardíos, de manera eficiente y gestionando el estado interno.