Ejercicio 3

- Objetivo: Implementar un sistema de procesamiento de datos de sensores en tiempo real utilizando Spark Structured Streaming.
- Planteamiento: Una fábrica inteligente ha instalado sensores de temperatura en diferentes áreas de la planta. Necesitan un sistema que pueda procesar estos datos en tiempo real y proporcionar estadísticas actualizadas constantemente.
- Configura un StreamingQuery que simule la entrada de datos de sensores.
- Procesa el stream para calcular:
  - Temperatura promedio por sensor en los últimos 5 minutos.
  - Temperatura máxima por sensor desde el inicio del stream.
- Muestra los resultados actualizados cada minuto.
- Ayuda:
  - Utiliza spark.readStream.format("rate") para simular un stream de entrada.
  - La función window() te ayudará a crear ventanas de tiempo para los cálculos.
  - Usa groupBy() con las funciones de agregación avg() y max() para los cálculos requeridos.
  - Configura el outputMode y el trigger en writeStream para controlar cómo y cuándo se actualizan los resultados.

```
import org.apache.spark.sql.SparkSession

import org.apache.spark.sql.functions._

import org.apache.spark.sql.streaming.Trigger


object Ejercicio3 {

  def main(args: Array[String]): Unit = {

    val spark = SparkSession.builder()

      .appName("Ejercicio3")

      .master("local[*]")

      .getOrCreate()


    import spark.implicits._


    // Simular stream de datos de sensores
```

```
val sensorData = spark.readStream
  .format("rate")
  .option("rowsPerSecond", 5)
  .load()
  .withColumn("sensorId", (rand() * 3).cast("int")) // Genera sensorId aleatorio (0, 1, o 2)
  .withColumn("temperature", (rand() * 100).cast("double")) // Genera temperatura aleatoria (0.0 - 100.0)

// Procesar stream
val processedData = sensorData
  .withWatermark("timestamp", "5 minutes") // Define una marca de agua para manejar datos tardíos
  .groupBy($"sensorId", window($"timestamp", "5 minutes")) // Agrupa por sensorId y ventanas de tiempo de 5 minutos
  .agg(
    avg("temperature").as("avg_temp"), // Calcula el promedio de temperatura en la ventana
    max("temperature").as("max_temp")  // Calcula la temperatura máxima en la ventana
  )

// Escribir resultados en consola
val query = processedData.writeStream
  .outputMode("complete") // Especifica que se escriban todos los resultados agregados en cada trigger
  .format("console")      // Escribe la salida en la consola
  .trigger(Trigger.ProcessingTime("1 minute")) // Ejecuta el procesamiento cada 1 minuto
  .start()
```

println("Iniciando consulta de streaming. Los resultados se mostrarán en la consola cada minuto durante 5 minutos.")

query.awaitTermination(300000) // Ejecutar la consulta por 5 minutos (300000 milisegundos)

query.stop() // Detener la consulta de streaming

spark.stop() // Detener la sesión de Spark

}

}

Explicación del Código:

Este programa en Scala utiliza Apache Spark Structured Streaming para procesar un flujo continuo de datos simulados de sensores.

1. **Importaciones:**
   - `org.apache.spark.sql.SparkSession`: Necesario para crear una sesión de Spark.
   - `org.apache.spark.sql.functions._`: Contiene funciones de Spark SQL, incluyendo las necesarias para trabajar con streaming y ventanas de tiempo.
   - `org.apache.spark.sql.streaming.Trigger`: Permite definir cuándo se debe procesar el stream de datos.

2. **Creación de la Sesión de Spark:**
   - Se crea una sesión de Spark de la misma manera que en los ejemplos anteriores.

3. **Simulación del Stream de Datos de Sensores:**
   - `spark.readStream.format("rate").option("rowsPerSecond", 5).load()`: Crea una fuente de datos de streaming utilizando el formato "rate". Este formato genera filas a una velocidad especificada (aquí, 5 filas por segundo), con una columna `timestamp` que indica cuándo se generó cada fila.

     Componentes Clave:

     - **`spark.readStream`**: Inicia el proceso de creación de un DataFrame/DataStreamReader para streaming.
     - **`.format("<fuente_de_datos>")`**: Especifica el tipo de fuente de datos que quieres leer. Las fuentes de datos comunes incluyen:
       - **`"kafka"`**: Para leer datos de Apache Kafka.

    - `"text"`: Para leer archivos de texto (nuevos archivos que aparecen en un directorio).
    - `"csv"`: Para leer archivos CSV (nuevos archivos que aparecen en un directorio).
    - `"json"`: Para leer archivos JSON (nuevos archivos que aparecen en un directorio).
    - `"parquet"`: Para leer archivos Parquet (nuevos archivos que aparecen en un directorio).
    - `"orc"`: Para leer archivos ORC (nuevos archivos que aparecen en un directorio).
    - `"socket"`: Para leer datos de un socket de red (principalmente para pruebas).
    - También se pueden implementar fuentes de datos personalizadas.
- `.options(...)`: Aquí configuras opciones específicas para la fuente de datos elegida. Las opciones disponibles varían según el formato. Por ejemplo:
    - **Kafka:** `bootstrap.servers`, `subscribe`, `subscribePattern`, `assign`, `startingOffsets`, `endingOffsets`.
    - **Fuentes basadas en archivos (text, csv, json, parquet, orc):** `path` (el directorio a monitorizar), `maxFilesPerTrigger`, `latestFirst`, `fileNameOnly`.
    - **CSV:** `header`, `sep`, `inferSchema`.
    - **JSON:** multiLine.
    - **Socket:** host, `port`.
- `.schema(...)` **(Opcional pero Muy Recomendado):** Te permite definir explícitamente el esquema de los datos que estás leyendo. Si bien Spark a veces puede inferir el esquema (por ejemplo, para CSV y JSON), generalmente es **una buena práctica proporcionar un esquema** por las siguientes razones:
    - **Estabilidad:** Evita que tu aplicación de streaming se rompa si el formato de los datos cambia ligeramente.
    - **Rendimiento:** Los esquemas explícitos pueden mejorar el rendimiento.
    - **Control de Tipos de Datos:** Asegura que tus datos se interpreten con los tipos de datos correctos.
- `.load()`: Finaliza la configuración y devuelve un `DataFrame` que representa el flujo continuo de datos.

- `.withColumn("sensorId", (rand() * 3).cast("int"))`: Añade una nueva columna llamada `sensorId`. Utiliza la función `rand()` para generar números aleatorios entre 0 y 1 (excluido), los multiplica por 3 y luego los convierte a enteros, resultando en valores de `sensorId` de 0, 1 o 2. Esto simula datos provenientes de tres sensores diferentes.
- `.withColumn("temperature", (rand() * 100).cast("double"))`: Añade una columna llamada `temperature` con valores aleatorios de punto flotante entre 0.0 y 100.0, simulando lecturas de temperatura.

4. **Procesamiento del Stream:**

- `.withWatermark("timestamp", "5 minutes")`: Define una marca de agua (watermark) en la columna `timestamp`. La marca de agua es un mecanismo para manejar datos que llegan tarde (datos cuyo timestamp está dentro de un cierto límite de tiempo pasado). En este caso, se considera que los datos que llegan más de 5 minutos después de la marca de agua pueden ser descartados para ciertos tipos de agregaciones con estado.

  Implicaciones de `.withWatermark("timestamp", "5 minutes")`:

  - **Datos Tardíos:** Si un evento llega con una marca de tiempo que está más de 5 minutos *antes* del tiempo máximo observado hasta ahora (en la columna "timestamp"), es considerado **demasiado tarde** (late data).
  - **Manejo de Datos Tardíos:** Por defecto, los datos demasiado tardíos se descartan. Sin embargo, puedes configurar Spark para que los envíe a un sumidero (sink) separado utilizando la opción `.trigger(processingTime='...')` en tu `writeStream` y luego utilizando `.withLateWatermarkAllowed(delay)`.
  - **Operaciones de Ventana:** La marca de agua es crucial para las operaciones de ventana (como `groupBy(window(...))`). Define cuándo Spark puede decir con certeza que todos los datos para una ventana específica han llegado (dentro del retraso permitido) y, por lo tanto, puede emitir el resultado de esa ventana.
  - **Agregaciones con Estado:** Para agregaciones con estado (por ejemplo, `groupBy` sin ventanas), la marca de agua ayuda a limpiar el estado de las claves que no han tenido actividad durante un cierto período, aunque esto se configura de manera diferente con `stateTimeout`.

- `.groupBy($"sensorId", window($"timestamp", "5 minutes"))`: Agrupa el stream de datos por la columna `sensorId` y por ventanas de tiempo de 5 minutos basadas en la columna `timestamp`. Esto significa que se crearán grupos separados para cada sensor y para cada intervalo de 5 minutos.
- `.agg(avg("temperature").as("avg_temp"), max("temperature").as("max_temp"))`: Realiza funciones de agregación en cada grupo:
  - `avg("temperature").as("avg_temp")`: Calcula el promedio de la columna `temperature` dentro de cada ventana y renombra la columna resultante como `avg_temp`.
  - `max("temperature").as("max_temp")`: Calcula el valor máximo de la columna `temperature` dentro de cada ventana y renombra la columna resultante como `max_temp`.

5. **Escritura de los Resultados en la Consola:**
   - `processedData.writeStream`: Inicia la configuración de la escritura del stream procesado.
   - `.outputMode("complete")`: Especifica el modo de salida. En el modo "complete", toda la tabla de resultados agregados se vuelve a escribir en cada trigger. Este modo es adecuado para agregaciones que se actualizan con cada micro-batch.
   - `.format("console")`: Indica que los resultados deben escribirse en la consola.

- `.trigger(Trigger.ProcessingTime("1 minute"))`: Define el trigger para el procesamiento del stream. `Trigger.ProcessingTime("1 minute")` significa que Spark intentará procesar los nuevos datos y generar la salida cada 1 minuto (basado en el tiempo de procesamiento del sistema).
- `.start()`: Inicia la consulta de streaming. Esto devuelve un objeto `StreamingQuery`.

6. **Mantenimiento y Detención de la Consulta:**
   - `println("Iniciando consulta de streaming. Los resultados se mostrarán en la consola cada minuto durante 5 minutos.")`: Imprime un mensaje indicando que la consulta ha comenzado.
   - `query.awaitTermination(300000)`: Bloquea el programa principal y espera a que la consulta de streaming termine o alcance el tiempo de espera especificado (300000 milisegundos = 5 minutos).
   - `query.stop()`: Detiene la consulta de streaming, liberando los recursos asociados.
   - `spark.stop()`: Detiene la sesión de Spark.

Cómo Funciona el Streaming con Ventanas de Tiempo:

Spark Structured Streaming permite realizar cálculos con estado sobre flujos de datos continuos. Las ventanas de tiempo son una forma común de agregar datos en intervalos específicos.

- **Ventana de 5 minutos:** La función `window($"timestamp", "5 minutes")` crea ventanas de tiempo que se agrupan cada 5 minutos. Por ejemplo, una ventana podría ser de 10:00:00 a 10:04:59, la siguiente de 10:05:00 a 10:09:59, y así sucesivamente. Los datos que caen dentro de estos intervalos se agregan juntos para cada `sensorId`.
- **Marca de Agua:** La marca de agua ayuda a lidiar con la latencia en los datos de streaming. Al definir una marca de agua de "5 minutes" en la columna `timestamp`, Spark asume que cualquier dato que llegue más de 5 minutos tarde (en relación con el último timestamp procesado) podría no ser considerado para las agregaciones con estado. Esto es importante para evitar que el estado de la agregación crezca indefinidamente debido a datos muy tardíos.

Cuando ejecutes este código, verás en la consola cómo se imprimen los resultados cada minuto, mostrando el promedio y la temperatura máxima para cada `sensorId` dentro de las ventanas de tiempo de 5 minutos que se van completando. La consulta se ejecutará durante 5 minutos y luego se detendrá.

El resultado del `Ejercicio3` será una salida continua en la consola que se actualizará aproximadamente cada minuto durante 5 minutos. Esta salida mostrará el promedio (`avg_temp`) y la temperatura máxima (`max_temp`) para cada `sensorId` dentro de las ventanas de tiempo de 5 minutos que se van completando.

Posible Salida (se repetirá cada minuto):

```
-------------------------------------------
Batch: 0
-------------------------------------------
+--------+--------------------+--------+--------+
|sensorId|window              |avg_temp|max_temp|
```

```
+--------+--------------------+--------+--------+
|0       |[2025-03-18 11:08:...|50.234  |98.76   |
|1       |[2025-03-18 11:08:...|45.678  |92.10   |
|2       |[2025-03-18 11:08:...|55.901  |99.99   |
+--------+--------------------+--------+--------+

-------------------------------------------
Batch: 1
-------------------------------------------
+--------+--------------------+--------+--------+
|sensorId|window              |avg_temp|max_temp|
+--------+--------------------+--------+--------+
|0       |[2025-03-18 11:09:...|52.111  |95.43   |
|1       |[2025-03-18 11:09:...|48.333  |90.56   |
|2       |[2025-03-18 11:09:...|58.777  |97.89   |
+--------+--------------------+--------+--------+

-------------------------------------------
Batch: 2
-------------------------------------------
+--------+--------------------+--------+--------+
|sensorId|window              |avg_temp|max_temp|
+--------+--------------------+--------+--------+
|0       |[2025-03-18 11:10:...|49.555  |91.23   |
|1       |[2025-03-18 11:10:...|46.999  |93.67   |
|2       |[2025-03-18 11:10:...|54.123  |98.11   |
+--------+--------------------+--------+--------+

... (y así sucesivamente hasta que pasen 5 minutos)
```

Explicación de la Salida:

- **`Batch: N`**: Indica el número del micro-batch procesado. Dado que el trigger es de 1 minuto, cada batch representa los datos que llegaron durante ese minuto.
- **`sensorId`**: El identificador del sensor (0, 1 o 2).
- **`window`**: Un intervalo de tiempo de 5 minutos. La columna muestra el inicio y el final de la ventana para la cual se calcularon las agregaciones. Por ejemplo, `[2025-03-18 11:08:00.0 UTC, 2025-03-18 11:13:00.0 UTC)` indica una ventana que comenzó a las 11:08:00 y terminó (se completó para la agregación) a las 11:13:00 UTC.
- **`avg_temp`**: El promedio de las lecturas de temperatura recibidas para ese `sensorId` dentro de esa ventana de 5 minutos. Este valor cambiará en cada batch a medida que se completan nuevas ventanas.
- **`max_temp`**: La temperatura máxima registrada para ese `sensorId` dentro de esa ventana de 5 minutos.

Puntos Importantes:

- **Naturaleza Streaming:** La salida no es estática como en los ejercicios anteriores. Continuará apareciendo en la consola cada minuto mientras la consulta de streaming esté activa.
- **Ventanas Deslizantes:** Aunque aquí estamos usando ventanas fijas de 5 minutos, es importante entender que cada vez que el trigger se dispara (cada minuto), Spark evalúa las ventanas que se han completado desde el último trigger.
- **Valores Aleatorios:** Los valores de `avg_temp` y `max_temp` serán diferentes cada vez que ejecutes el programa, ya que los datos de los sensores se generan aleatoriamente.

- **Marca de Agua:** La marca de agua de "5 minutes" asegura que Spark espera hasta cierto punto por datos tardíos antes de finalizar la agregación para una ventana específica.

Después de aproximadamente 5 minutos, el programa llamará a `query.awaitTermination(300000)`, que hará que el programa principal espere a que la consulta de streaming termine. Una vez que el tiempo de espera se agote, se llamará a `query.stop()` y `spark.stop()`, finalizando la aplicación.