Ejercicio 1

- Objetivo: Analizar tendencias de ventas por categoría de producto utilizando funciones de ventana en Spark SQL.
- Planteamiento: Una empresa de comercio electrónico quiere analizar sus datos de ventas para entender mejor las tendencias por categoría de producto. Necesitan calcular las ventas acumuladas y el promedio móvil de ventas para cada categoría de producto a lo largo del tiempo.

- Crea un DataFrame con las siguientes columnas: fecha, categoria, ventas.
- Utiliza funciones de ventana para calcular:
  - Ventas acumuladas por categoría.
  - Promedio móvil de ventas de 3 días por categoría.
  - Muestra los resultados ordenados por categoría y fecha.
- Ayuda:
  - Utiliza SparkSession para crear el DataFrame.
  - La función Window.partitionBy() te ayudará a definir la ventana por categoría.
  - Las funciones sum() y avg() pueden usarse con over() para cálculos de ventana.
  - Para el promedio móvil, considera usar windowSpec.rowsBetween().

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

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

import org.apache.spark.sql.expressions.Window


object Ejercicio1 {

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

    val spark = SparkSession.builder()

      .appName("Ejercicio1")

      .master("local[*]")

      .getOrCreate()


    import spark.implicits._


    // Crear DataFrame de ventas
```

```
val ventas = Seq(
  ("2023-01-01", "Electrónica", 1000),
  ("2023-01-02", "Ropa", 500),
  ("2023-01-03", "Electrónica", 1500),
  ("2023-01-04", "Hogar", 800),
  ("2023-01-05", "Ropa", 600),
  ("2023-01-06", "Electrónica", 2000),
  ("2023-01-07", "Hogar", 900),
  ("2023-01-08", "Ropa", 700),
  ("2023-01-09", "Electrónica", 1200),
  ("2023-01-10", "Hogar", 750)
).toDF("fecha", "categoria", "ventas")

// Definir ventana por categoría ordenada por fecha
val windowSpec = Window.partitionBy("categoria").orderBy("fecha")

// Calcular ventas acumuladas y promedio móvil
val resultados = ventas
  .withColumn("fecha", to_date($"fecha"))
  .withColumn("ventas_acumuladas", sum("ventas").over(windowSpec))
  .withColumn("promedio_movil", avg("ventas").over(windowSpec.rowsBetween(-1, 1)))

println("Resultados con Ventas Acumuladas y Promedio Móvil:")
resultados.orderBy("categoria", "fecha").show()
```

```
    // Ejemplo adicional: Calcular el rango de ventas por categoría
    val rankSpec = Window.partitionBy("categoria").orderBy(desc("ventas"))
    val resultadosConRango = resultados
      .withColumn("rango_ventas", rank().over(rankSpec))

    println("\nResultados con Rango de Ventas por Categoría:")
    resultadosConRango.orderBy("categoria", "fecha").show()

    // Ejemplo adicional: Calcular la diferencia de ventas con la fila anterior por categoría
    val lagSpec = Window.partitionBy("categoria").orderBy("fecha")
    val resultadosConDiferencia = resultados
      .withColumn("ventas_anterior", lag("ventas", 1, 0).over(lagSpec))
      .withColumn("diferencia_ventas", $"ventas" - $"ventas_anterior")

    println("\nResultados con Diferencia de Ventas con la Fila Anterior:")
    resultadosConDiferencia.orderBy("categoria", "fecha").show()

    spark.stop()
  }
}
```

Explicación del Código:

1. **Importaciones:**
   - `org.apache.spark.sql.SparkSession`: Para crear la sesión de Spark.
   - `org.apache.spark.sql.functions._`: Contiene funciones de Spark SQL como `to_date`, `sum`, `avg`, `rank`, `lag`, etc.
   - `org.apache.spark.sql.expressions.Window`: Permite definir especificaciones de ventana para realizar cálculos sobre un conjunto de filas relacionadas.

2. **Creación de la Sesión de Spark:**
    - Se crea una sesión de Spark con un nombre de aplicación y se configura para ejecutarse localmente.

3. **Creación del DataFrame de Ventas:**

Cuando te encuentras con `Spark.implicits` en Apache Spark, estás lidiando con un componente crucial que simplifica la manipulación de datos, especialmente cuando trabajas con DataFrames y Datasets. Aquí tienes un desglose:

Concepto Central:

- `Spark.implicits` es un objeto dentro de la `SparkSession` que proporciona conversiones implícitas. Estas conversiones transforman automáticamente objetos Scala (como secuencias o RDDs) en DataFrames o Datasets de Spark, haciendo que tu código sea más conciso.
- Esencialmente, le proporciona al compilador la capacidad de realizar automáticamente ciertas conversiones, por lo que no tienes que escribir código de conversión detallado.

Funcionalidades Clave:

- **Codificadores (Encoders):**
    - Una parte importante de `Spark.implicits` es proporcionar `Encoders`. Los codificadores son esenciales para que Spark comprenda la estructura de tus datos y los convierta a su formato interno. Esto es particularmente importante para los Datasets.
- **Conversiones:**
    - Permite conversiones como:
        - Convertir una `Seq` (secuencia) de Scala en un `Dataset` de Spark.
        - Convertir un RDD (Resilient Distributed Dataset) en un `Dataset`.
        - Proporcionar la capacidad de utilizar la notación "$" para la selección de columnas.
- **Sintaxis Simplificada:**
    - Al importar `Spark.implicits._`, obtienes acceso a estas conversiones implícitas, que agilizan tu código Spark.

Implicaciones Prácticas:

- Cuando ves `import spark.implicits._` en el código Scala de Spark, significa que el programador está habilitando estas conversiones automáticas. Esto es muy común cuando se trabaja con DataFrames y Datasets.
- Simplifica tareas como la creación de DataFrames a partir de datos locales o la conversión de RDDs existentes.

En resumen, `Spark.implicits` es una función de conveniencia que hace que el código Spark sea más legible y eficiente al automatizar tareas comunes de conversión de datos.

    - Se define una secuencia de tuplas que representan datos de ventas (fecha, categoría, ventas).

- `.toDF("fecha", "categoria", "ventas")` convierte esta secuencia en un DataFrame con las columnas especificadas.

4. **Definición de la Ventana (`windowSpec`):**

En el contexto de Spark SQL (y otros sistemas de procesamiento de datos como SQL estándar), una **ventana** se refiere a un **conjunto de filas relacionadas con la fila actual** dentro de un DataFrame o tabla. Este conjunto de filas se define dinámicamente y se utiliza para realizar cálculos sobre él, sin necesidad de realizar agrupaciones tradicionales (`GROUP BY`) que colapsarían las filas.

Imagina que estás viendo una fila específica en tu DataFrame. Una ventana te permite definir un "marco" de filas alrededor de esa fila actual (o incluso en todo el DataFrame o partición), sobre el cual puedes aplicar funciones especiales llamadas **funciones de ventana** o **funciones analíticas**.

Características clave de las ventanas en Spark:

- **Conjunto de filas relacionadas:** La ventana define qué otras filas se consideran relevantes para el cálculo de la fila actual. Esta relación se basa en criterios como particionamiento y ordenamiento.
- **Cálculos sin agregación global:** A diferencia de las funciones de agregación tradicionales (como `SUM`, `AVG`, `COUNT` con `GROUP BY`), las funciones de ventana realizan cálculos para cada fila individualmente, utilizando los datos de la ventana definida para esa fila. El resultado del cálculo se agrega como una nueva columna en el DataFrame, sin reducir el número de filas.
- **Definición flexible:** Puedes definir ventanas basadas en:
    - **Particionamiento (`partitionBy`):** Divide el DataFrame en particiones según los valores de una o más columnas. La ventana se aplica independientemente dentro de cada partición.
    - **Ordenamiento (`orderBy`):** Define el orden de las filas dentro de cada partición (o en todo el DataFrame si no hay particionamiento). El orden es crucial para muchas funciones de ventana que dependen de la posición relativa de las filas.
    - **Marco de ventana (`rowsBetween`, `rangeBetween`):** Define explícitamente qué filas dentro de la partición ordenada se incluyen en la ventana para la fila actual. Puedes definir un número fijo de filas antes y después de la fila actual, o un rango de valores dentro de una columna ordenada.

¿Para qué se utilizan las ventanas en Spark?

Las ventanas son extremadamente útiles para realizar una amplia gama de análisis, incluyendo:

- **Cálculos de rango y clasificación:** Determinar la posición de cada fila dentro de un grupo (por ejemplo, el producto más vendido en cada categoría, el estudiante con la mejor calificación en cada clase).
- **Cálculos acumulativos:** Calcular sumas acumulativas, promedios móviles, etc., a lo largo de una secuencia de datos (por ejemplo, el total de ventas hasta la fecha por región, el promedio de temperatura de los últimos 7 días).
- **Análisis de desplazamiento (Lag y Lead):** Acceder a los valores de filas anteriores o posteriores dentro de un conjunto ordenado (por ejemplo, comparar el precio de una acción con su precio del día anterior).

- **Cálculo de diferencias:** Calcular la diferencia entre el valor de una fila y el valor de la fila anterior o siguiente.
- **Identificación de los primeros o últimos elementos dentro de un grupo:** Encontrar el primer o el último evento dentro de cada categoría.

Ejemplo conceptual:

Imagina un DataFrame con datos de ventas:

categoria	fecha	ventas
A	2023-01-01	100
A	2023-01-02	150
A	2023-01-03	120
B	2023-01-01	200
B	2023-01-02	220

Si definimos una ventana particionada por "categoria" y ordenada por "fecha", al aplicar una función de ventana como `sum("ventas") OVER (windowSpec)`, obtendríamos algo como:

categoria	fecha	ventas	suma_acumulativa
A	2023-01-01	100	100
A	2023-01-02	150	250
A	2023-01-03	120	370
B	2023-01-01	200	200
B	2023-01-02	220	420

Exportar a Hojas de cálculo

Observa cómo la `suma_acumulativa` se calcula dentro de cada categoría (partición) y se va acumulando a medida que avanzamos en las fechas (ordenamiento).

En resumen, una ventana en Spark es un mecanismo poderoso para definir conjuntos de filas relacionadas y realizar cálculos contextuales sin la necesidad de agregaciones que colapsan los datos. Permite realizar análisis sofisticados y obtener información detallada dentro de subconjuntos específicos de los datos.

2. `Window.partitionBy("categoria")`

- **`Window`**: Este es un objeto estático que proporciona métodos para crear especificaciones de ventana. Se encuentra en el paquete `org.apache.spark.sql.expressions`.
- **`.partitionBy("categoria")`**: Este método define cómo se particionan los datos antes de aplicar la función de ventana.
    - **`partitionBy()`**: Agrupa las filas del DataFrame basándose en los valores de la columna especificada (en este caso, "categoria").

- `"categoria"`: El nombre de la columna por la cual se realizará la partición. Todas las filas que compartan el mismo valor en la columna "categoria" formarán una misma partición.
- **Implicaciones de la partición:**
  - Las funciones de ventana se aplicarán de forma independiente dentro de cada partición. Esto significa que los cálculos no "cruzarán" los límites de las particiones.
  - Por ejemplo, si estamos calculando un rango dentro de cada categoría, el rango comenzará desde 1 para la primera fila de cada categoría.

3. `.orderBy("fecha")`

- `.orderBy("fecha")`: Este método define el orden de las filas dentro de cada partición creada por `partitionBy()`.
  - `orderBy()`: Ordena las filas dentro de cada partición según los valores de la columna especificada (en este caso, "fecha").
  - `"fecha"`: El nombre de la columna por la cual se realizará la ordenación. Las filas dentro de cada partición se ordenarán de forma ascendente según los valores de la columna "fecha" de forma predeterminada. Si se desea un orden descendente, se puede usar `orderBy(desc("fecha"))`.
  - **Importancia del ordenamiento:**
    - Muchas funciones de ventana dependen del orden de las filas dentro de la partición. Por ejemplo, las funciones de rango (`rank()`, `dense_rank()`), las funciones de desplazamiento (`lag()`, `lead()`) y las funciones acumulativas (`sum() OVER (ORDER BY ...)`) se ven directamente afectadas por el orden definido.

En resumen, la especificación de ventana `windowSpec` define lo siguiente:

1. **Particionamiento:** Los datos del DataFrame se dividirán en particiones separadas, donde cada partición contendrá todas las filas que tengan el mismo valor en la columna "categoria".
2. **Ordenamiento dentro de cada partición:** Dentro de cada una de estas particiones, las filas se ordenarán según los valores de la columna "fecha" (de forma ascendente por defecto).

¿Para qué se utiliza esta especificación de ventana?

Una vez definida la especificación de ventana `windowSpec`, se puede utilizar en conjunto con diversas funciones de ventana para realizar cálculos sofisticados dentro de cada partición ordenada. Algunos ejemplos comunes incluyen:

- **Calcular el rango o la clasificación de elementos dentro de cada categoría:**

  En Spark SQL (y por lo tanto, en PySpark y Scala Spark), la función `.withColumn()` se utiliza para **agregar una nueva columna a un DataFrame o reemplazar una columna existente.** Es una de las transformaciones más comunes y poderosas para manipular la estructura de tus datos.
  `df.withColumn("rank_dentro_categoria", rank().over(windowSpec))`

- **Calcular la suma acumulativa o el promedio móvil de una métrica dentro de cada categoría a lo largo del tiempo (fecha):**

```
df.withColumn("suma_acumulativa", sum("valor").over(windowSpec))
```

- **Acceder al valor de la fila anterior o siguiente dentro de cada categoría según la fecha:**

```
df.withColumn("valor_anterior", lag("valor", 1).over(windowSpec))
df.withColumn("valor_siguiente", lead("valor", 1).over(windowSpec))
```

- **Identificar el primer o el último elemento dentro de cada categoría según la fecha:**

```
df.withColumn("primer_valor", first("valor").over(windowSpec))
df.withColumn("ultimo_valor", last("valor").over(windowSpec))
```

En conclusión:

La línea de código `val windowSpec = Window.partitionBy("categoria").orderBy("fecha")` crea una especificación de ventana que instruye a Spark SQL a:

- Agrupar las filas del DataFrame por los valores únicos de la columna "categoria".
- Ordenar las filas dentro de cada uno de estos grupos (particiones) según los valores de la columna "fecha".

Esta especificación de ventana se convierte en un componente clave para aplicar funciones de ventana que permiten realizar análisis complejos y obtener información valiosa sobre los datos dentro de contextos específicos definidos por las particiones y el ordenamiento. Es una herramienta poderosa para realizar análisis comparativos, cálculos secuenciales y obtener perspectivas dentro de subconjuntos de datos definidos por una o más columnas.

  - `val windowSpec = Window.partitionBy("categoria").orderBy("fecha")`: Define una especificación de ventana.
    - `partitionBy("categoria")`: Indica que los cálculos se realizarán de forma independiente para cada valor único en la columna "categoria".
    - `orderBy("fecha")`: Especifica el orden de las filas dentro de cada partición (por fecha en este caso).

5. **Cálculo de Ventas Acumuladas y Promedio Móvil:**
   - `.withColumn("fecha", to_date($"fecha"))`: Convierte la columna "fecha" al tipo de dato `Date` para asegurar un ordenamiento correcto.
   - `.withColumn("ventas_acumuladas", sum("ventas").over(windowSpec))`: Calcula las ventas acumuladas para cada categoría, ordenadas por fecha. `sum("ventas").over(windowSpec)` aplica la función de agregación `sum` sobre la ventana definida. Para cada fila, suma las ventas de todas las filas precedentes y la fila actual dentro de su categoría.
   - `.withColumn("promedio_movil", avg("ventas").over(windowSpec.rowsBetween(-1, 1)))`: Calcula el promedio móvil de las ventas para cada categoría. `avg("ventas").over(windowSpec.rowsBetween(-1, 1))` aplica la función de agregación `avg` sobre una ventana que incluye la fila anterior (`-1`), la fila actual (`0`

implícito) y la fila siguiente (`1`) dentro de su categoría y ordenadas por fecha. Para la primera y la última fila de cada partición, la ventana se ajusta.

6. **Mostrar Resultados:**
    - `resultados.orderBy("categoria", "fecha").show()`: Muestra el DataFrame resultante, ordenado por categoría y fecha para facilitar la visualización de los cálculos de ventana.

7. **Ejemplos Adicionales (Añadidos):**
    - **Cálculo del Rango de Ventas por Categoría:**
        - `val rankSpec = Window.partitionBy("categoria").orderBy(desc("ventas"))`: Define una nueva especificación de ventana para calcular el rango dentro de cada categoría, ordenando las ventas de forma descendente (`desc`).
        - `.withColumn("rango_ventas", rank().over(rankSpec))`: Calcula el rango de cada venta dentro de su categoría. Las filas con el mismo valor de ventas recibirán el mismo rango, y el siguiente rango se saltará los valores correspondientes.
        - Se muestra el DataFrame con la columna "rango_ventas".
    - **Cálculo de la Diferencia de Ventas con la Fila Anterior por Categoría:**
        - `val lagSpec = Window.partitionBy("categoria").orderBy("fecha")`: Define una especificación de ventana similar a `windowSpec` para acceder a la fila anterior.
        - `.withColumn("ventas_anterior", lag("ventas", 1, 0).over(lagSpec))`: Utiliza la función `lag` para obtener el valor de la columna "ventas" de la fila anterior dentro de la misma categoría y ordenada por fecha. El segundo argumento (`1`) indica que se acceda a la fila que está una posición antes, y el tercer argumento (`0`) es el valor predeterminado si no hay una fila anterior (por ejemplo, para la primera fila de cada categoría).
        - `.withColumn("diferencia_ventas", $"ventas" - $"ventas_anterior")`: Calcula la diferencia entre las ventas actuales y las ventas de la fila anterior.
        - Se muestra el DataFrame con las columnas "ventas_anterior" y "diferencia_ventas".

8. **Detener la Sesión de Spark:**
    - `spark.stop()`: Cierra la sesión de Spark y libera los recursos.

Cómo Funciona el Cálculo con Ventanas:

Las funciones de ventana en Spark SQL permiten realizar cálculos que involucran un conjunto de filas relacionadas con la fila actual. La especificación de la ventana define qué filas se incluyen en este conjunto (la "ventana").

- **`partitionBy`**: Divide el DataFrame en particiones basadas en los valores de la columna especificada. El cálculo de la ventana se realiza de forma independiente dentro de cada partición.
- **`orderBy`**: Define el orden de las filas dentro de cada partición. Esto es importante para funciones como `sum` acumulativo o promedios móviles.
- **`rowsBetween`**: Define el rango de filas que se incluirán en la ventana para cada fila actual.
    - `-1`: Indica la fila anterior a la actual.
    - `1`: Indica la fila siguiente a la actual.
    - `0`: Indica la fila actual.
    - `Window.unboundedPreceding`: Indica todas las filas desde el principio de la partición hasta la fila actual.
    - `Window.unboundedFollowing`: Indica todas las filas desde la fila actual hasta el final de la partición.

En este ejemplo, el promedio móvil se calcula considerando la venta del día anterior, la venta del día actual y la venta del día siguiente dentro de la misma categoría. Para la primera y última fecha de cada categoría, la ventana se ajusta, ya que no habrá una fila anterior o siguiente respectivamente.

Los ejemplos adicionales ilustran otras funciones de ventana útiles como `rank` para asignar rangos basados en valores dentro de una partición y `lag` para acceder a los valores de filas precedentes.

El resultado del `Ejercicio1` sería una tabla impresa en la consola que mostraría los datos de ventas originales junto con dos columnas adicionales calculadas utilizando funciones de ventana: `ventas_acumuladas` y `promedio_movil`. Además, como añadí ejemplos adicionales, también se mostrarían tablas con el `rango_ventas` y la `diferencia_ventas` con respecto a la fila anterior.

Aquí te presento una posible salida, basada en los datos de entrada y los cálculos definidos:

Resultados con Ventas Acumuladas y Promedio Móvil:

```
+----------+-----------+------+-----------------+--------------+
|fecha     |categoria  |ventas|ventas_acumuladas|promedio_movil|
+----------+-----------+------+-----------------+--------------+
|2023-01-01|Electrónica|1000  |1000             |1000.0        |
|2023-01-03|Electrónica|1500  |2500             |1250.0        |
|2023-01-06|Electrónica|2000  |4500             |1500.0        |
|2023-01-09|Electrónica|1200  |5700             |1566.6666666666667|
|2023-01-04|Hogar      |800   |800              |800.0         |
|2023-01-07|Hogar      |900   |1700             |850.0         |
|2023-01-10|Hogar      |750   |2450             |850.0         |
|2023-01-02|Ropa       |500   |500              |500.0         |
|2023-01-05|Ropa       |600   |1100             |550.0         |
|2023-01-08|Ropa       |700   |1800             |600.0         |
+----------+-----------+------+-----------------+--------------+
```

Explicación de las Columnas:

- **`fecha`**: La fecha de la venta.
- **`categoria`**: La categoría del producto vendido.
- **`ventas`**: El monto de la venta.
- **`ventas_acumuladas`**: La suma de las ventas hasta esa fecha dentro de la misma categoría (ordenado por fecha).

- **`promedio_movil`**: El promedio de las ventas considerando la fila anterior, la fila actual y la fila siguiente dentro de la misma categoría (ordenado por fecha). Para la primera y última fila de cada categoría, el promedio se calcula con las filas disponibles.

Resultados con Rango de Ventas por Categoría:

```
+----------+-----------+------+-----------------+--------------+------------+
|fecha     |categoria  |ventas|ventas_acumuladas|promedio_movil|rango_ventas|
+----------+-----------+------+-----------------+--------------+------------+
|2023-01-06|Electrónica|2000  |4500             |1500.0        |1           |
|2023-01-03|Electrónica|1500  |2500             |1250.0        |2           |
|2023-01-01|Electrónica|1000  |1000             |1000.0        |3           |
|2023-01-09|Electrónica|1200  |5700             |1566.6666666666667|4           |
|2023-01-07|Hogar      |900   |1700             |850.0         |1           |
|2023-01-04|Hogar      |800   |800              |800.0         |2           |
|2023-01-10|Hogar      |750   |2450             |850.0         |3           |
|2023-01-08|Ropa       |700   |1800             |600.0         |1           |
|2023-01-05|Ropa       |600   |1100             |550.0         |2           |
|2023-01-02|Ropa       |500   |500              |500.0         |3           |
+----------+-----------+------+-----------------+--------------+------------+
```

Explicación de la Columna Adicional:

- **`rango_ventas`**: El rango de cada venta dentro de su categoría, basado en el monto de las ventas (de mayor a menor).

Resultados con Diferencia de Ventas con la Fila Anterior:

```
+----------+-----------+------+-----------------+--------------+--------------+-----------------+
|fecha     |categoria  |ventas|ventas_acumuladas|promedio_movil|ventas_anterior|diferencia_ventas|
+----------+-----------+------+-----------------+--------------+--------------+-----------------+
|2023-01-01|Electrónica|1000  |1000             |1000.0        |0.0           |1000.0           |
|2023-01-03|Electrónica|1500  |2500             |1250.0        |1000.0        |500.0            |
|2023-01-06|Electrónica|2000  |4500             |1500.0        |1500.0        |500.0            |
|2023-01-09|Electrónica|1200  |5700             |1566.6666666666667|2000.0        |-800.0           |
|2023-01-04|Hogar      |800   |800              |800.0         |0.0           |800.0            |
|2023-01-07|Hogar      |900   |1700             |850.0         |800.0         |100.0            |
|2023-01-10|Hogar      |750   |2450             |850.0         |900.0         |-150.0           |
|2023-01-02|Ropa       |500   |500              |500.0         |0.0           |500.0            |
|2023-01-05|Ropa       |600   |1100             |550.0         |500.0         |100.0            |
|2023-01-08|Ropa       |700   |1800             |600.0         |600.0         |100.0            |
+----------+-----------+------+-----------------+--------------+--------------+-----------------+
```

Explicación de las Columnas Adicionales:

- **`ventas_anterior`**: El valor de la columna "ventas" de la fila anterior dentro de la misma categoría (ordenado por fecha). Para la primera fila de cada categoría, el valor es el predeterminado que definimos en la función `lag` (que era 0).
- **`diferencia_ventas`**: La diferencia entre el valor de "ventas" de la fila actual y el valor de "ventas" de la fila anterior.

Es importante notar que el orden de las filas dentro de cada categoría está determinado por la columna "fecha" debido a la cláusula `orderBy("fecha")` en la definición de la ventana.