Práctica 1: Datos por Lotes en Streaming y Visualización de datos con PySpark.

CONTEXTO:

La capacidad de procesar y analizar grandes volúmenes de datos en tiempo real se ha convertido en un factor crítico para la toma de decisiones informada. En el ámbito bursátil, comprender con rapidez lo que ocurre en el mercado permite reaccionar ante cambios de volatilidad, rotaciones sectoriales y eventos corporativos.

A pesar de los esfuerzos continuos de los reguladores y del propio mercado por mejorar la transparencia, liquidez y protección del inversor, la volatilidad y los episodios de corrección siguen condicionando el comportamiento del IBEX 35, principal referencia bursátil de España. En 2022–2024, el índice ha atravesado tramos de fuertes subidas y retrocesos, poniendo de manifiesto la necesidad de comprender con mayor profundidad las tendencias y los factores que explican su evolución. Los datos muestran patrones diferenciados por sectores (finanzas, energía, utilities, telecomunicaciones, turismo, inmobiliario), lo que ha llevado a los agentes del mercado a impulsar una nueva iniciativa de información y educación financiera para el público inversor.

Con el fin de responder a este reto, se plantea una iniciativa de análisis que permita identificar tendencias, riesgos y oportunidades a distintos horizontes temporales (intradiario, semanal, mensual y anual).

En bolsa, el medio plazo generalmente abarca un horizonte temporal de uno a cinco años, buscando un equilibrio entre riesgo y rentabilidad, y se caracteriza por una mayor volatilidad aceptable en el corto plazo, permitiendo a los inversores beneficiarse de revalorizaciones moderadas, pero con mayor potencial.

Esta práctica se divide en dos partes: La primera, permite profundizar en operaciones avanzadas que son esenciales para manejar y analizar flujos de datos complejos. La segunda, se enfoca en las estrategias y técnicas para la visualización de datos.

Para la entrega de la práctica es necesario entregar EL PROYECTO COMPLETO con el código ordenado, bien estructurado y funcionando. **La normativa y pautas de la primera parte son las mismas que para la practica 0.**

PRÁCTICA:

Yahoo finance es una librería Python que consulta datos públicos para descargar históricos dividendos, splits, etc., para acciones, índices, ETFs y divisas.

```
import yfinance as yf
df = yf.download(ticker, start=start, end=end, interval=interval)
```

El código anterior, permite descargar datos con yfinance para el ticker indicado, entre start y end, con la granularidad interval (por ejemplo: "1d", "1h", "15m"...).

Devuelve un DataFrame de pandas donde el índice suele ser la fecha/hora (Date/DatetimeIndex) y las columnas son Open, High, Low, Close, Volume.

En este dataframe, “Date” no es una columna, si no un índice. Para convertirlo a columna es necesario usar:

```
df.reset_index(inplace=True)  # Convertir el índice 'Date' a columna
```

Y posteriormente se puede crear un dataframe de spark con todos los datos. Los datos recibidos son:

- Date — date, no nulo. Día de la sesión (formato YYYY-MM-DD).
- Open — double. Precio de apertura del día.
- High — double. Máximo intradía.
- Low — double. Mínimo intradía.
- Close — double- Precio de cierre oficial.
- Volume — int, nulo permitido. Acciones negociadas en el día.
- Ticker — string, no nulo. Símbolo único que representa un activo financiero.

Parquet es un formato de almacenamiento columna-oriented diseñado para análisis de datos a gran escala. A diferencia de los formatos por filas (como CSV), Parquet guarda juntas los valores de cada columna, lo que permite aplicar reducir drásticamente el tamaño en disco y el I/O. Además, soporta anidamiento (arrays, structs, mapas) y tipos complejos, lo que lo hace robusto para datos semiestructurados.

Para almacenar el Dataframe de Pypsark:

```
#Convierte a pandas
df_pandas = df_spark.toPandas()

#Almacena
df_pandas.to_parquet('./data.parquet', index=False)

# Append en vez de sobrescribir
df.write.mode("append").parquet("/ruta/a/destino")
```

Como conoce el esquema (los nombres y tipos de las columnas), los programas que lo leen (Spark) pueden abrir solo las columnas que se necesitan y saltarse el resto, por lo que consigue menores tiempo de lectura y analítica eficiente sobre millones o miles de millones de registros.

```
# Archivo o carpeta Parquet
df = spark.read.parquet("/ruta/a/datos")

# Solo algunas columnas
df = spark.read.parquet("/ruta").select("col1", "col2")
```

A continuación, realiza las siguientes funcionalidades. Indica claramente a que ejercicio pertenece cada sección del código tal y como se especificó en la práctica 0:

- Ej1a: Crea una función que permita descargar el histórico de una acción concreta, facilitando una fecha de inicio y una de fin, y lo almacene en un dataframe de pyspark. Define para cada columna si permite nulos y por qué.

- Ej1b: Crea una función que limpie los datos y verifique si hay algún dato erróneo. Comenta tus comprobaciones.

- Ej1c: Crea una función que almacene los datos en formato parquet. Solo se almacenarán los datos del ticker si no existen datos de este ya almacenados.

- Ej1d (opcional): Crea una función que elimine/amplie los datos de un ticker almacenado en un parquet.

- Ej2: Crea una función, que agregue una columna que indique que día de la semana era cuando se tomaron los datos.

- Ej3(opcional): Un gap de apertura es la diferencia entre el precio de apertura de hoy y el cierre de la sesión anterior del mismo valor. Crea una función que permita añadir una columna con este cálculo en %. (Close / Open −1)×100

- Ej4: Obtén los datos, desde enero del 2020 hasta enero de 2025, de los siguientes tickers: BBVA.MC, SAB.MC, IBE.MC, NTGY.MC, TEF.MC, CLNX.MC. Utiliza las funciones anteriores para completar la información de las empresas y guardarlas para no tener que volver a descargarlas. Muestra:
    - Los primeros 5 valores correspondientes al histórico de los tickers.
    - Los primeros 5 valores correspondientes a cada empresa donde se vea calculado el día de la semana.
    - (Opcional): Los primeros 5 valores correspondientes a cada empresa donde se vea calculado el gap.

Posteriormente, se va a obtener datos en tiempo real sobre estas empresas españolas.

- En primer lugar, usa una api gratuita para programar una función que permita obtener el tipo de cambio monetario en tiempo real que hay del Euro respecto al dólar. Por ejemplo:

```
import yfinance as yf

ticker = yf.Ticker(tipo_cambio)
price = ticker.fast_info['last_price']
```

Esta tabla representa una lista con algunos de los tipos de cambios admitidos por el yahoo finance.

Ticker	Descripción
EURUSD=X	Euro frente al Dólar estadounidense.
USDJPY=X	Dólar estadounidense frente al Yen japonés.
EURRUB=X	Euro frente al Rublo ruso
USDCAD=X	Dólar estadounidense frente al Dólar canadiense
EURJPY=X	Euro frente al Yen japonés.

Interpretación del resultado: El Ticker indica la dirección del par para saber cómo leer la cotización: un símbolo como AAA/BBB (por ejemplo, EUR/USD) se interpreta siempre como "1 unidad de la primera divisa (base) vale N unidades de la segunda (cotizada)".

La moneda base es la primera del par (p. ej., EUR en EUR/USD) y la moneda cotizada es la segunda (p. ej., USD en EUR/USD); el precio expresa cuántas unidades de la cotizada obtienes por 1 unidad de la base.

- En segundo lugar, programa una función que obtenga el día y la hora en la que sucede la lectura de ese dato. Por ejemplo:

```
from datetime import datetime

now = datetime.now()

fecha_actual = now.strftime("%Y-%m-%d") # YYYY-MM-DD

hora_actual = now.strftime("%H:%M:%S") # HH:MM:SS
```

Obtén también el día de la semana que es. Weekday, es un método de datetime.date y datetime.datetime que devuelve el día de la semana como número. Con weekday() devuelve el día de la semana como un número (0 para lunes, 1 para martes, etc.).

```
now.weekday()
```

- En tercer lugar, vamos a simular que obtenemos la información de las empresas en tiempo real. **El archivo "datos.py" simula el envío de datos diferentes empresas "en tiempo real". En este archivo, NO HAY QUE MODIFICAR NADA.** Hay que tener en cuenta que los datos son enviados al puerto 8080, y al host "localhost" a través de un socket. La estructura de los datos recibida es la siguiente:
    - Un objeto/diccionario JSON donde la clave es el ticker y su valor es una lista (array). Cada elemento de la lista es un objeto con cuatro campos numéricos:

- price: precio de último/medio desde la última actualización.
- high: máximo desde la última actualización.
- low: mínimo desde la última actualización.
- volume: volumen negociado desde la última actualización.

StreamingContext es una clase de PySpark que se utiliza para configurar y gestionar aplicaciones de Spark Streaming. Spark Streaming, componente de Apache Spark, permite procesar flujos de datos en tiempo real (o casi en tiempo real). Con él se puede consumir datos de fuentes de streaming (Kafka, sockets, etc.), procesarlos con la API de Spark y almacenar los resultados en sistemas persistentes como Delta Lake o bases de datos.

Spark Streaming divide los datos en pequeños intervalos de tiempo ("batches" o lotes) para analizarlos de forma continua.

```
from pyspark.streaming import StreamingContext
```

La variable "spark.storage.replication" determina cuántas copias (réplicas) de los bloques de datos que se reciben se mantienen en memoria o en disco dentro del clúster para tolerar fallos. Si el nodo que almacena un bloque falla, esos datos no estarán disponibles salvo que el valor sea superior a 1.

```
spark.conf.set("spark.storage.replication", "1")
```

Para crear la sesión de streaming, se utiliza la variable de sesión ya generada y se accede a la conexión con el clúster de Spark. Después se crea un objeto StreamingContext(), punto de entrada a las aplicaciones de Spark Streaming (DStream). Un DStream es una abstracción lógica que representa un flujo continuo de datos como una secuencia de RDDs (uno por cada micro-lote).
Este objeto recibe dos parámetros: la conexión con el núcleo de Spark y la frecuencia (en segundos) con la que los datos se agrupan en lotes para su procesamiento.

```
sc = spark.sparkContext

spark.sparkContext.setLogLevel("ERROR")

ssc = StreamingContext(sc, 10)
```

El objeto de conexión establece una comunicación con un socket en un host y un puerto concretos. A medida que llegan los datos al socket, se almacenan como texto en el flujo "líneas". La función "pprint()" es una operación de salida y permite ver la información recibida.

```
lineas = ssc.socketTextStream("host", puerto)

lineas.pprint()
```

Los datos recibidos se guardan en lineas con el formato RDD (Resilient Distributed Datasets), la estructura de datos más básica y fundamental de Spark: colecciones distribuidas que se procesan en paralelo y que pueden contener números, cadenas u objetos. Los DataFrames y los Datasets de Spark se construyen sobre RDDs. Los RDD son inmutables: cada transformación crea un RDD nuevo; el original no se modifica.

En Spark la evaluación es "perezosa": la ejecución se pospone hasta que existe un motivo para materializar resultados. Cuando encadenas transformaciones (map, filter, join, ventana, estado…), Spark no procesa en ese instante; en su lugar, construye un plan que describe qué debe hacerse sobre los RDDs/DStreams cuando corresponda. En streaming, el tiempo avanza en micro-lotes. Al cerrarse cada intervalo, el driver examina el grafo y solo genera *jobs* para los caminos que terminan en una operación de salida (por ejemplo, escribir en almacenamiento).

Por ello, para trabajar con DataFrames es habitual **convertir** cada RDD del flujo. La función "foreachRDD" permite definir una operación que se aplicará a cada RDD que llega en el flujo.

```
lineas.foreachRDD(funcion)
```

Y se define:

```
def funcion(rdd_recibido):
```

Hay que tener en cuenta, que es necesario trabajar con los datos para que tengan el formato adecuado para almacenarse en un Dataframe dado que, por socket, siempre llega texto. Algunas funciones útiles:

- cadena.split(signo): Divide la cadena en una lista de partes, utilizando el signo, espacio o cadena que se quiera como delimitador.
- map(tipo_de_dato, datos_separados): Convierte cada elemento de la lista que se le pasa al tipo de dato indicado.
- datosrdd.map(función): En este caso se aplica la función a cada elemento de datosrdd. Esto transforma cada elemento de datosrdd según la lógica definida en la función.
- tuple: Transforma la lista a una tupla.

Al final de una aplicación de Spark Streaming, siempre se necesita las líneas:

```
ssc.start()

ssc.awaitTermination()
```

- start(): inicia el procesamiento de streaming. Hasta que se llame a ssc.start(), la aplicación de Spark Streaming solo ha configurado el contexto y las operaciones que se realizarán en los datos, pero no comienza a recibir ni procesar los datos.
- awaitTermination(): mantiene el programa activo, bloqueando el hilo principal mientras el StreamingContext sigue ejecutándose. Si no la aplicación terminará inmediatamente después de ejecutar ssc.start().

La estrategia más práctica es llevar un contador de micro-lotes vacíos: en cada lote se comprueba si no llegó ningún registro; si ocurre N veces seguidas (umbral), se considera que el flujo está inactivo y se finaliza. También se puede expresar en tiempo de inactividad.

```
ssc.stop()
```

Por último, spark.read permite leer datos en PySpark. Devuelve un DataFrameReader, un objeto con métodos para cargar datos desde múltiples formatos y fuentes, aplicando opciones y un esquema explícito. El resultado de una lectura es siempre un DataFrame. Ejemplos:

```
df = spark.read.json("path/archivo.json o rdd")     # JSON Lines o carpeta
df = spark.read.csv("path/archivo.csv")       # CSV/TSV
df = spark.read.parquet("path/carpeta")       # Parquet (columnar)
df = spark.read.table("db.mitabla")          # Catálogo (Hive/Delta)

df = spark.read.jdbc(url=url, table=table, properties=properties)
```

Para continuar con los ejercicios, ejecuta en un terminal el archivo de “datos.py” para activar el envío de datos (no envía de forma indefinida) y realiza los siguientes ejercicios.

- Ej4: Crea una conexión que permita recibir los datos en tiempo real con StreamingContext.
- Ej5: Recoge TODOS los datos que se están recibiendo sobre las empresas, en formato json, y almacénalos en un DataFrame con la estructura que consideres. Añade la fecha (y cualquier elemento temporal que consideres) y el valor EUR/USD que había en el momento que se recibió el valor de la acción.
- Ej6(opcional): Crea una función que permita calcular el “Open”, y el resto de valores que se han empleado en el Ej1 para este grupo de datos.

PARTE 2:

Los datos están presentes en todos los ámbitos de la vida y una de las vías más potentes para comprenderlos es el canal visual. Mediante gráficos y paneles es posible resumir, explicar y anticipar comportamientos. Antes de explorar formas de construir visualizaciones, conviene repasar algunos fundamentos.

Comprender al usuario

Al diseñar una visualización, es esencial conocer el uso que se le dará y a quién va dirigida. ¿Cuál es su objetivo? ¿Apoyará decisiones estratégicas o tareas operativas? ¿Se trata de una visualización analítica? Identificar si un panel respalda decisiones críticas o si es, por ejemplo, el tablero operativo de un ingeniero de fiabilidad del sitio orientará el diseño hacia una experiencia clara y alineada con los objetivos del usuario. La información presentada "ilumina" a quienes la consultan; la persona que diseña la visualización asume el rol de narradora de datos y, con ello, la capacidad de influir en decisiones relevantes.

Validar los datos

La verificación de los datos utilizados es imprescindible. Comprobar escalas, rangos y otros elementos fundamentales constituye un pilar de cualquier buena visualización. La ejecución de pruebas automatizadas antes del uso en producción aporta una red de confianza y permite que las decisiones basadas en los paneles gocen de mayor credibilidad.

Visualización de datos con cuadernos

A continuación se presentan los principales tipos de gráficos.

Gráficos de líneas: adecuados para puntos que varían en un continuo, como series temporales. Destacan los cambios sutiles a lo largo de periodos extensos.

```
#Una linea
import plotly.express as px
import pandas as pd

stock_prices: pd.DataFrame = px.data.stocks() \
   .melt("date", var_name="listing", value_name="price") \
   .sort_values(["date","listing"])

figure = px.line(
   stock_prices, x="date", y="price",
   color="listing",
   animation_frame="listing",
   title="Acciones tecnológicas 2018 y 2019"
)
figure.show()
```

```
#Varias lineas
import plotly.express as px
import pandas as pd

stock_prices = px.data.stocks() \
   .melt('date', var_name='listing', value_name='price') \
   .sort_values(['date','listing'])

fig = px.line(
   stock_prices, x="date", y="price",
   color="listing",
```

```
    title="Acciones tecnológicas 2018 y 2019"
)
fig.show()
```

Gráficos de barras: útiles para comparar magnitudes y mostrar diferencias entre grupos. Se emplean principalmente con datos categóricos, no con datos continuos.

```
import plotly.express as px
import pandas as pd

tips: pd.DataFrame = px.data.tips()
figure = px.bar(tips, x='size', y='tip')
figure.show()
```

Histogramas: equivalentes a barras para datos continuos; suelen mostrar frecuencias (por ejemplo, distribución de ventas).

```
import plotly.express as px
import pandas as pd

tips: pd.DataFrame = px.data.tips()
figure = px.histogram(
    tips, x="tip", nbins=25,
    color_discrete_sequence=['green']
)
figure.show()
```

Diagramas de dispersión (scatter): revelan relaciones y posibles correlaciones entre dos variables.

```
import plotly.express as px
import pandas as pd

iris: pd.DataFrame = px.data.iris()
figure = px.scatter(
    iris, x="sepal_width", y="sepal_length",
    hover_data=["petal_width"],
    color="species", size="petal_length"
)
figure.show()
```

Gráficos circulares: resultan adecuados para representar porcentajes de un total.

```
import plotly.express as px
import pandas as pd

# Dataset de ejemplo
```

```
tips = px.data.tips()

# Porcentaje de cuentas por día
por_dia = tips.groupby("day").size().reset_index(name="cuentas")

fig_pie = px.pie(
   por_dia,
   names="day",
   values="cuentas",
   title="Distribución de cuentas por día",
   hole=0.0  # 0.4 para donut
)
fig_pie.update_traces(textposition="inside", textinfo="percent+label")
fig_pie.show()

# Versión donut
fig_donut = px.pie(
   por_dia,
   names="day",
   values="cuentas",
   title="Distribución de cuentas por día (donut)",
   hole=0.4
)
fig_donut.update_traces(textposition="inside", textinfo="percent+label")
fig_donut.show()
```

Gráficos de burbujas: permiten mostrar relaciones entre tres o más valores numéricos colocando puntos en un plano X-Y y variando el tamaño de cada marca.

```
import plotly.express as px
import pandas as pd

population = px.data.gapminder()
figure = px.scatter(
   population.query("year==1952"),
   y="lifeExp", x="gdpPercap",
   size="pop", color="continent",
   hover_name="country",
   log_x=True, size_max=80
)
figure.show()
```

A continuación, se llevará a cabo un análisis visual de los resultados con el objetivo de identificar patrones, comparar tendencias y detectar posibles anomalías de forma intuitiva. Este análisis se apoyará en gráficos, y resúmenes visuales que faciliten la comprensión rápida y la toma de decisiones informadas. Para enriquecer la interpretación y acelerar la exploración de hipótesis, se habilitará el uso de herramientas de inteligencia artificial (IA).

Para resolver los siguientes ejercicios, se señalará en el código —igual que en la PARTE 1— el fragmento correspondiente a cada apartado. Además, se entregará un informe en PDF que incluya:

- los gráficos de cada ejercicio,
- la argumentación de los datos seleccionados para contrastar las hipótesis,
- la justificación de los enfoques utilizados (porque se han seleccionado ese gráfico), y
- un análisis profundo de los resultados.

Importante: los gráficos de cada ejercicio deben presentarse de forma independiente (una figura por ejercicio o, si procede, varias figuras separadas dentro del mismo ejercicio).

- Ej7a. Las noticias y resultados no son el único factor que afecta al precio de una acción: el sentimiento humano también influye. El Friday effect es una estacionalidad documentada en finanzas conductuales (aversión al riesgo antes del fin de semana, rebalanceos, flujos corporativos, etc.). Objetivo: comprobar si los retornos diarios de los viernes son estadísticamente distintos (mayores o menores) a los del resto de días laborables.
- Ej7b (opcional). Analiza el mismo fenómeno durante el periodo vacacional de verano. Investiga el contexto y justifica si, con el periodo de datos disponible, se observa alguna tendencia.
- Ej7c (opcional). Investiga si existen otras estacionalidades, eventos de calendario (ej., fin/principio de mes, festivos, efecto lunes) o elementos (por sectores) que puedan influir en el precio.
- Ej8. Analiza la relación entre precio y volumen. Muestra la evidencia gráfica y comenta los resultados (ej., correlación, volumen relativo, confirmación de movimientos).
- Ej9. Muestra un gráfico de los datos recibidos en tiempo real y añade algún cálculo adicional relevante para entender cómo está variando la acción.
- Ej10 (opcional). Los gaps recogen información overnight (noticias, sentimiento de fin de semana). Determina si tienden a revertir (cerrar el hueco) o a continuar y representa gráficamente el resultado para orientar decisiones de entrada en apertura.