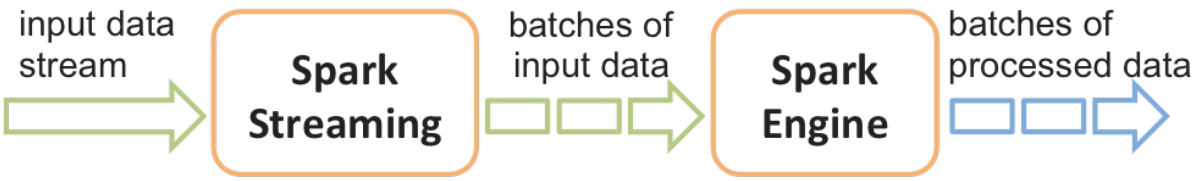
Spark Streaming

Procesamiento escalable, *high-throughput* y tolerante a fallos de flujos de datos.



Entrada desde muchas fuentes: Kafka, Flume, Twitter, ZeroMQ, Kinesis o sockets TCP.

APIs SPARK para Streaming

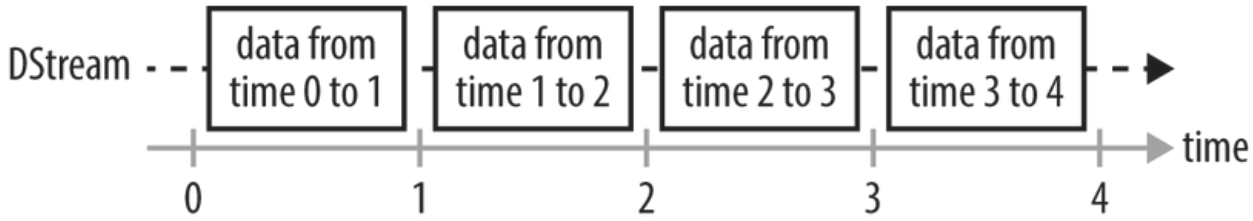
- DStream API (no soportada ya)
  - `API original, basada en RDDs`
- Structured Streaming
  - `Disponible desde la versión 2.2, basada en DataFrames`

Página de Spark Streaming: https://spark.apache.org/streaming/ Documentación principal (de la última versión): https://spark.apache.org/docs/latest/streaming-programming-guide.html

DStream API

Abstracción principal: DStream (*discretized stream*).

- Representa un flujo continuo de datos



Arquitectura *micro-batch*

- Los datos recibidos se agrupan en batches
- Los batches se crean a intervalos regulares (*batch interval*)
- Cada batch forma un RDD, que es procesado por Spark
- Adicionalmente: transformaciones con estado mediante
  - Operaciones con ventanas
  - Tracking del estado por cada clave

Página de Spark Streaming: https://spark.apache.org/streaming/ Documentación principal (de la última versión): https://spark.apache.org/docs/latest/streaming-programming-guide.html

In [ ]:
```
from pyspark import SparkContext
from pyspark.sql import SparkSession
import os

# Elegir el máster de Spark dependiendo de si se ha definido la variable de entorno HADOOP_CONF_DIR o YARN_CONF
SPARK_MASTER: str = 'yarn' if 'HADOOP_CONF_DIR' in os.environ or 'YARN_CONF_DIR' in os.environ else 'local[*]'
print(f"Usando Spark Master en {SPARK_MASTER}")

# Creamos un objeto SparkSession (o lo obtenemos si ya está creado)
spark: SparkSession = SparkSession \
  .builder \
  .appName("Mi aplicacion") \
  .config("spark.rdd.compress", "true") \
  .config("spark.executor.memory", "3g") \
  .config("spark.driver.memory", "3g") \
  .master(SPARK_MASTER) \
```

```
  .getOrCreate()

sc: SparkContext = spark.sparkContext
```

Structured Streaming

Utiliza la API estructurada (DataFrames, DataSets y SQL)

- Lee los datos a medida que llegan al sistema, los procesa y los añade a un DataFrame

Fuentes de datos (*input sources*):

- Apache Kafka
- Ficheros (lee los ficheros en un directorio de forma continua)
- Sockets

Destino de datos (*sinks*):

- Apache Kafka
- Ficheros
- Otras computaciones
- Memoria (para depuración y testing)

Ejemplo: procesar los ficheros en el directorio `by-day/`

In [ ]:
```
%%sh
wget -q https://raw.githubusercontent.com/dsevilla/tcdm-public/refs/heads/24-25/datos/by-day.zip
unzip by-day.zip
# Vemos el formato de un fichero
ls by-day/
head by-day/2010-12-01.csv
```

Procesamos un fichero como DataFrame

In [ ]:
```
# Creamos un DataFrame normal con los datos de uno de los ficheros
from pyspark.sql.dataframe import DataFrame

dfEstatico: DataFrame = spark.read\
                  .format("csv")\
                  .option("header", "true")\
                  .option("inferSchema", "true")\
                  .load("by-day/2010-12-01.csv")
print("Número de particiones del DataFrame = {0}.".
      format(dfEstatico.rdd.getNumPartitions()))

dfEstatico.show()
```

In [ ]:
```
# Obtenemos un DataFrame con la compra por hora y por cliente durante ese día
from pyspark.sql.functions import window, col, desc

# Pedimos que se usen 4 particiones (opcional)
spark.conf.set("spark.sql.shuffle.partitions", "4")

dfCompraPorClientePorHoraEstatico: DataFrame =\
             dfEstatico.select(
                                col("CustomerId"),
                                (col("UnitPrice")*col("Quantity")).alias("total_cost"),
                                col("InvoiceDate"))\
                       .groupBy(col("CustomerId"), window(col("InvoiceDate"), "1 hour"))\
                       .sum("total_cost")

print("Número de particiones del DataFrame = {0}.".
      format(dfCompraPorClientePorHoraEstatico.rdd.getNumPartitions()))

print("Número de filas del DataFrame = {0}.".
      format(dfCompraPorClientePorHoraEstatico.count()))

dfCompraPorClientePorHoraEstatico.show(15, False)
```

Procesamos todos los ficheros en Streaming

In [ ]:
```
# Definimos un DataFrame en Streaming que toma como fuente de datos
# los ficheros en el directorio by-day/
# Indicamos que se lea 1 fichero en cada activación

dfStreaming: DataFrame = spark.readStream\
                    .schema(dfEstatico.schema)\
                    .option("maxFilesPerTrigger", 1)\
                    .format("csv")\
```

```
                    .option("header", "true")\
                    .load("by-day/*.csv")
print(type(dfStreaming))
```

In [ ]:
```
# A partir del anterior, obtenemos la compra por hora y por cliente
dfCompraPorClientePorHoraStreaming: DataFrame = \
            dfStreaming.select(
                               col("CustomerId"),
                              (col("UnitPrice")*col("Quantity")).alias("total_cost"),
                               col("InvoiceDate"))\
                       .groupBy(col("CustomerId"), window(col("InvoiceDate"), "1 hour"))\
                       .sum("total_cost")

print(type(dfCompraPorClientePorHoraStreaming))
```

In [ ]:
```
from pyspark.sql.streaming.readwriter import DataStreamWriter

# Creamos un objeto DataStreamWriter para escribir los valores del DataFrame previo
# Los valores se escriben a una tabla en memoria
# El modo de escritura es "complete": se reescribe la salida entera
# Los datos se pueden acceder a traves de la tabla compras_por_hora
# Se leen los datos de entrada cada segundo

dswConsultaCompras: DataStreamWriter = dfCompraPorClientePorHoraStreaming\
                    .writeStream\
                    .format("memory")\
                    .queryName("compras_por_hora")\
                    .outputMode("complete")\
                    .trigger(processingTime='1 seconds')
print(type(dswConsultaCompras))
```

In [ ]:
```
# Métodos definidos para un DataStreamWriter
[method_name for method_name in dir(dswConsultaCompras)
 if callable(getattr(dswConsultaCompras, method_name))]
```

In [ ]:
```
# Iniciamos el acceso a los datos de entrada
dswConsultaCompras.start()
```

In [ ]:
```
# Vamos mostrando la tabla cada segundo
from time import sleep

for x in range(20):
    spark.sql("""
            SELECT *
            FROM compras_por_hora
            ORDER BY `sum(total_cost)` DESC
            """).show(5, truncate=False)
    sleep(1)
```