

Procesado de datos

4. Introducción al procesado de datos en streaming

Boni García

<http://bonigarcia.github.io/>
boni.garcia@uc3m.es

Departamento de Ingeniería Telemática
Escuela Politécnica Superior

2020/2021

uc3m | Universidad **Carlos III** de Madrid

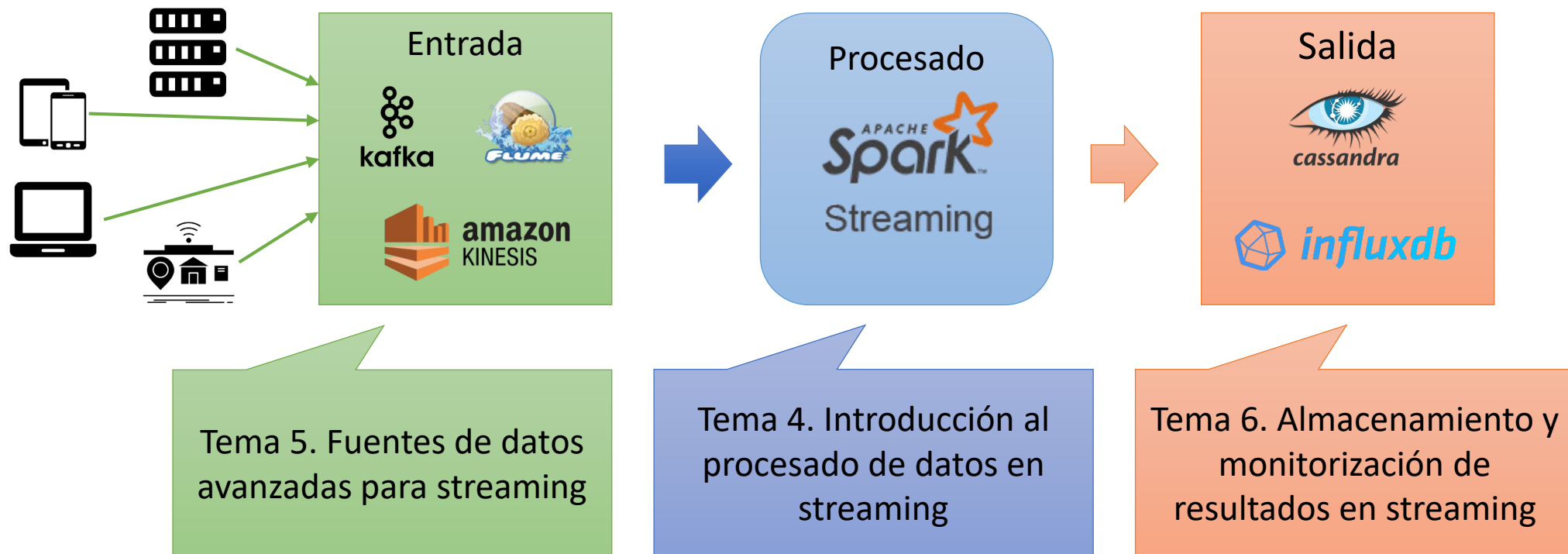


Contenidos

0. Presentación
1. Introducción
2. Apache Spark
3. Spark Streaming
4. Resumen

0. Presentación

- Este módulo de la asignatura está centrado en el procesamiento de datos en tiempo real (**streaming**) y está compuesto por 3 temas:



0. Presentación - Evaluación

- La **evaluación** se realizará mediante evaluación de prácticas
 - 50% cada modulo
 - En cada módulo se tiene en cuentas las prácticas intermedias (20%) y una práctica final (30%)
- Para la realización de las prácticas se proporciona una **máquina virtual** (VM) para ser ejecutada en **Virtual Box** (con soporte a 64 bits)
 - Credenciales de acceso:
 - login: user
 - password: user
 - Se distribuye a través de un fichero OVA (Open Virtualization Appliance):

Instalar desde:

<https://www.virtualbox.org/wiki/Downloads>

https://drive.google.com/file/d/1lnzDIAQOhO_umoRynfn-G1Lp4zLW8k5k/view?usp=sharing

Hay que estar autenticado con nuestra cuenta de UC3M para descargar este fichero

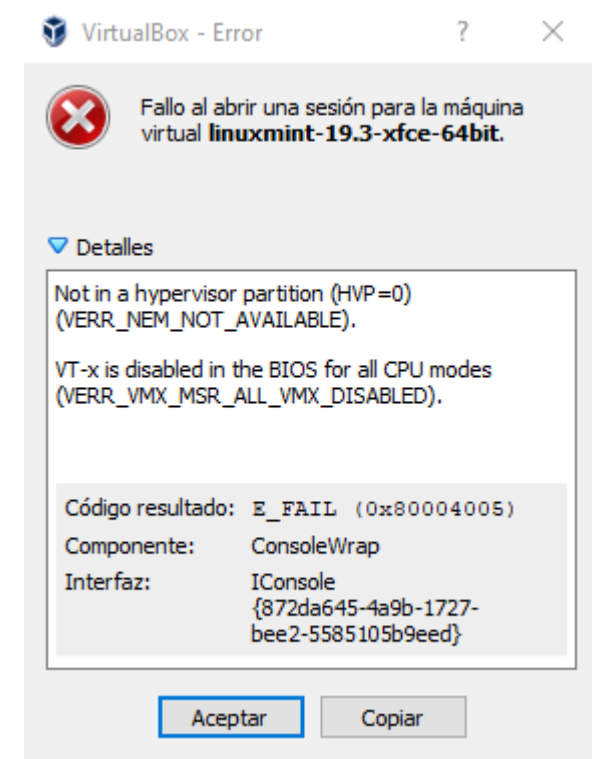
0. Presentación - Máquina virtual

- La VM de la asignatura está basada en un sistema operativo **Linux Mint 20.1** (distribución basada en Ubuntu) de 64 bits y contiene todo el software necesario para el desarrollo de las prácticas:
 - OpenJDK 1.8.0
 - Anaconda 3-2020.11
 - Python 3.7.9
 - Spark/PySpark 2.4.7 (Scala 2.11)
 - Hadoop 2.7.3
 - Flume 1.9.0
 - Kafka 2.4.1 (Scala 2.11)
 - Cassandra 4.0-beta4
 - kafka-python 2.0.2
 - influxdb-client 1.15.0

0. Presentación - Posibles problemas VM

- Problema: VirtualBox devuelve el siguiente error al iniciar la máquina virtual:
Not in a hypervisor partition
(HVP=0)(VERR_NEM_NOT_AVAILABLE).
VT-x is disabled in the BIOS for all CPU
modes (VERR_VMX_MSR_ALL_VMX_DISABLED).
- Causa: No está habilitado el hipervisor a nivel de BIOS
- Solución: Habilitar el hipervisor en la BIOS
 - Esta configuración depende del tipo de máquina
 - La siguiente página recoge formas de hacerlo en diferentes fabricantes (Acer, Asus, Dell...):

<https://2nwiki.2n.cz/pages/viewpage.action?pageId=75202968>



0. Presentación - Posibles problemas VM

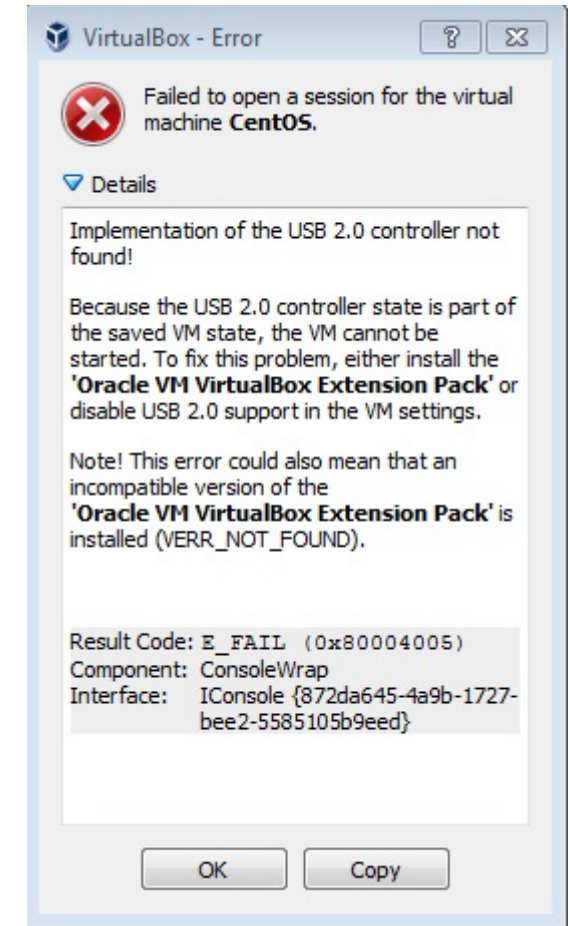
- Problema: VirtualBox devuelve el siguiente error al iniciar la máquina virtual:

Implementation of the USB 2.0 controller not found!

Because the USB 2.0 controller state is part of the saved VM state, the VM cannot be started. To fix this problem, either install the 'Oracle VM VirtualBox Extension Pack' or disable USB 2.0 support in the VM settings.

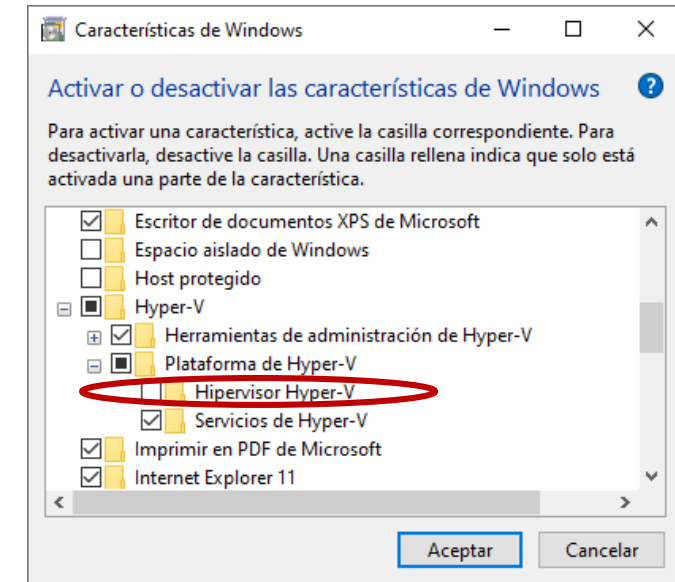
- Causa: Problema para acceder al controlador USB
- Solución: Instalar "Oracle VM VirtualBox Extension Pack":

<https://www.virtualbox.org/wiki/Downloads>



0. Presentación - Posibles problemas VM

- Problema: Imposibilidad de instalar nuevo software en la máquina virtual por fallos al comprobar integridad de datos (checksum, sha) al instalar nuevo software (en Windows 10 como anfitrión)
- Causa: Windows 10 está usando el hipervisor a nivel de sistema operativo
- Solución: Desactivar el uso de Hyper-V Hypervisor en la configuración de Windows

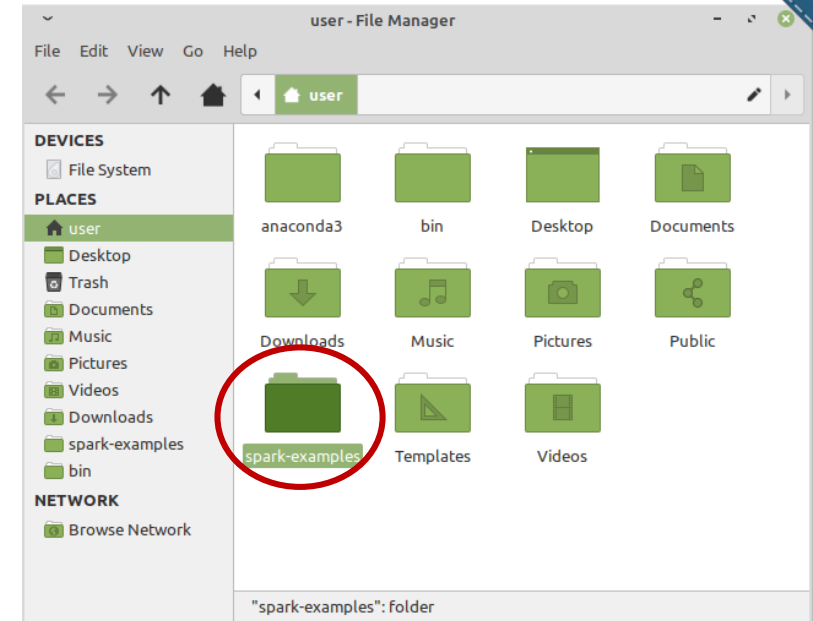


Solución alternativa:

1. Presionar tecla Windows + X → Windows PowerShell (como administrador)
2. Ejecutar comando:
`bcdedit /set hypervisorlaunchtype off`
3. Reiniciar

0. Presentación - Ejemplos

- Todo los ejemplos que vemos en este módulo está disponibles en un repositorio **GitHub**
 - Se pueden ejecutar de forma sencilla usando la línea de comandos en la máquina virtual
 - Una copia del repositorio se encuentra ya disponible en la máquina virtual



<https://github.com/bonigarcia/spark-examples>

Contenidos

0. Presentación

1. Introducción

- Big Data
- MapReduce
- Hadoop

2. Apache Spark

3. Spark Streaming

4. Resumen

1. Introducción - Big Data

- El problema del **Big Data** (no se suele traducir este concepto del inglés) ocurre cuando el volumen de datos que necesitamos procesar crecen a una velocidad superior a la capacidad de computo disponible
 - Soluciones: **escalar verticalmente** (usar hardware más potente) o **escalar horizontalmente** (procesar los datos en paralelo usando diferentes máquinas)
- El escalado vertical no es viable económicamente si el volumen de datos es alto (tera/peta bytes), por lo que se suele optar por escalar horizontalmente usando un **clúster**, esto es, un conjunto de máquinas trabajando en paralelo. Sus inconvenientes son:
 - El hardware suele ser de bajas prestaciones (*commodity hardware*) y es propenso a errores (por ejemplo, fallo en celda de memoria o CPU que va más lenta que otras)
 - El software necesario es más complejo. Por esta razón, los frameworks modernos de Big Data dan respuesta a problemas de concurrencia, alta disponibilidad y tolerancia a fallos

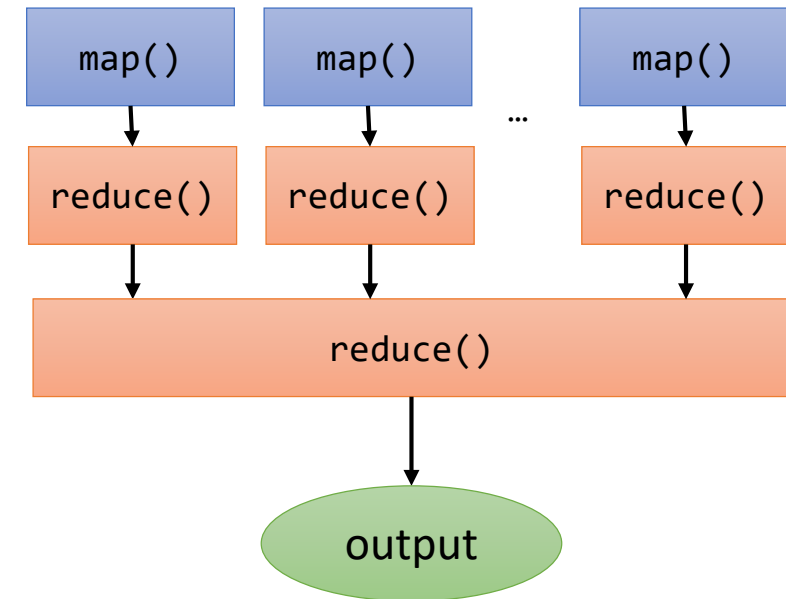
1. Introducción - MapReduce

- Una de las primeras tecnologías desarrolladas para solucionar el problema del Big Data fue **MapReduce**
 - Es un modelo de programación para el procesamiento en paralelo de grandes colecciones de datos en clústeres (escalado horizontal)
 - Fue hecho público por Dean Jeffrey y Sanjay Ghemawat (Google) en 2004 en el artículo “MapReduce: Simplified data processing on large clusters”
- El modelo MapReduce es la ejecución paralela de funciones `map()` y `reduce()` sobre un conjunto de datos

$$[x, y, z].\text{map}(f) \rightarrow [f(x), f(y), f(z)]$$
$$[x, y, z].\text{reduce}(f) \rightarrow w$$

1. Introducción - MapReduce

- Para aplicar el modelo MapReduce, se divide la entrada en conjuntos más pequeños a la cual aplicamos funciones de tipo `map()` y `reduce()` en paralelo
- El/los resultado(s) final(es) se obtiene usando una función `reduce()` al resultado de las partes
- Las ventajas del modelo MapReduce son:
 - Es escalable: Si el volumen de datos de entrada crece, simplemente habrá que añadir más unidades de procesamiento (map-reduce)
 - Es genérico: Podemos reemplazar `map()` y `reduce()` por cualquier función equivalente para poder procesar nuestros datos



1. Introducción - Hadoop

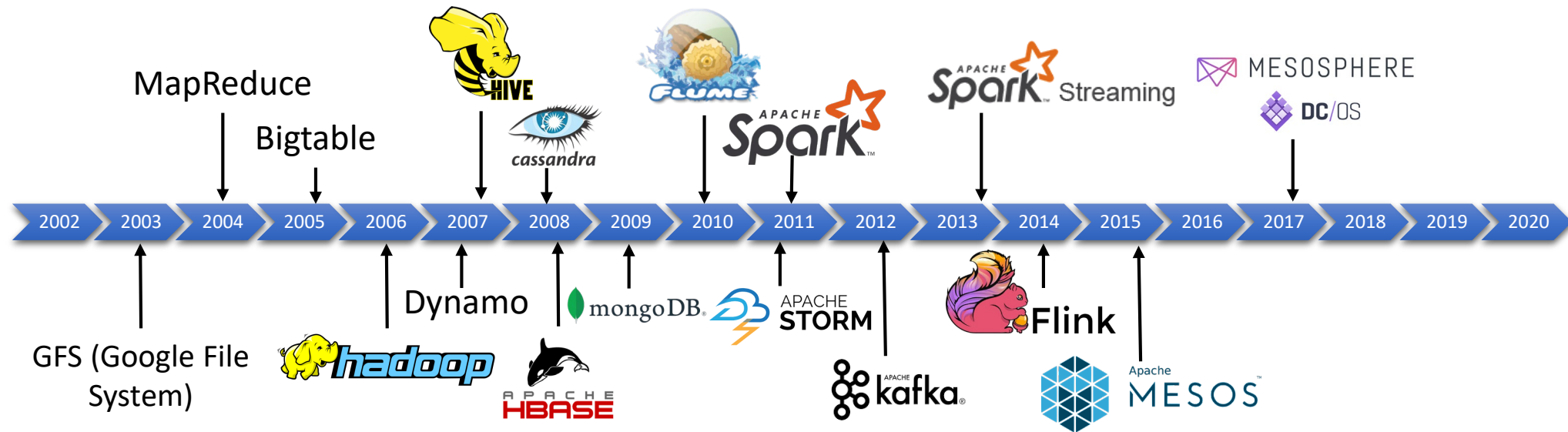
- Una de las primeras aplicaciones de MapReduce fue implementado inicialmente en **Hadoop**
 - Diseñado inicialmente por Doug Cutting, que lo nombró así por el elefante de juguete de su hijo
- Además de MapReduce, Hadoop se basa en Hadoop **HDFS**, que es un sistema de ficheros distribuido:
 - Fue creado a partir de Google File System (GFS)
 - Proporciona tolerancia a fallos mediante replicación
 - Portable a diferentes plataformas (escrito en Java)



<http://hadoop.apache.org/>

1. Introducción - Hadoop

- Después de Hadoop, han surgido multitud de herramientas dentro del ecosistema [Big Data](#)
- En esta asignatura estudiamos **Apache Spark**



Contenidos

0. Presentación

1. Introducción

2. Apache Spark

- Arquitectura
- RDDs
- PySpark
- Databricks
- Jupyter Notebooks
- DataFrames

3. Spark Streaming

4. Resumen

2. Apache Spark

- **Apache Spark** es un framework open-source de uso general para el procesamiento de datos
 - El desarrollo de Spark se inició en la universidad de Berkeley en 2009 y ha continuado desde 2013 en la Apache Software Foundation
 - Spark está escrito en Scala, por lo tanto se ejecuta dentro de la máquina virtual de Java (JVM)
 - Proporciona APIs para Python, Java, Scala, y R
 - Típicamente ejecutaremos Spark en un clúster, aunque se puede ejecutar en una máquina individual



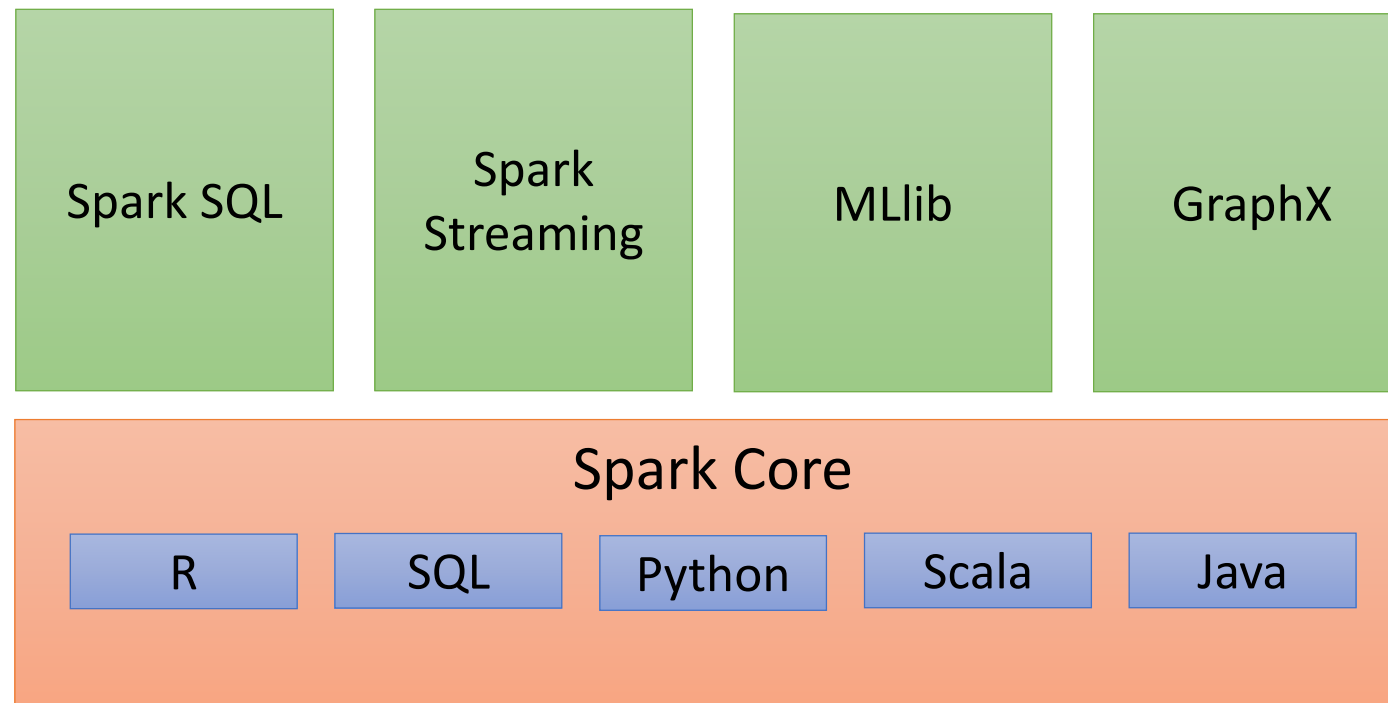
<https://spark.apache.org/>

2. Apache Spark

- Spark se considera una evolución de Hadoop, ya que ofrece entre otras las siguientes ventajas sobre este:
 - Alto **rendimiento**, ya que trabaja con los datos en memoria a diferencia de MapReduce, que utiliza datos almacenados en disco, lo que le permite ser más rápido a costa de un mayor consumo de recursos
 - Ofrece procesamiento en batch y streaming (Hadoop solo batch)
 - Permite análisis avanzado de datos a través de consultas SQL, algoritmos de **Machine Learning** (ML), o procesamiento de gráficos
- Spark puede ser usado para el procesamiento de datos:
 - **Batch**: procesamiento de datos previamente recolectados
 - **Streaming**: procesamiento de datos en tiempo real, es decir, se están recibiendo y tratando constantemente

2. Apache Spark - Arquitectura

- Spark está compuesto por los siguientes módulos:



2. Apache Spark - Arquitectura

- Spark está compuesto por los siguientes módulos:

1. Spark Core

- Ofrece capacidades de **alto rendimiento** para el procesado de datos basado en **MapReduce** trabajando con los **datos en memoria**
- Se encarga de funciones de entrada/salida, recuperación de fallos, o distribución de tareas
- La API proporcionada por el Spark Core se conoce como **RDD** (Resilient Distributed Dataset), y permite el procesado de datos **no estructurados**

En el ámbito del procesado de datos se suele hablar de 3 tipos de datos:

- Datos **no estructurados** (unstructured data). No tienen un formato específico. Se almacenan en múltiples formatos como documentos PDF o Word, correos electrónicos, ficheros multimedia de imagen, audio o video,...
- Datos **estructurados** (structured data): Tienen perfectamente definido la longitud, el formato y el tamaño de sus datos (esquema de datos). Se almacenan en formato tabla, hojas de cálculo o en bases de datos relacionales
- Datos **semiestructurados** (semistructured data). No presenta una estructura perfectamente definida como los datos estructurados pero sí presentan una organización en forma de metadatos (típicamente usando etiquetas o marcadores semánticos). Por ejemplo, HTML, XML o JSON

2. Apache Spark - Arquitectura

2. Spark SQL

- Componente para procesado de datos **estructurados y semiestructurados**
- Ofrece diferentes APIs de alto nivel:
 - **SQL**. Permite la ejecución de consultas a bases de datos (como Apache Hive)
 - **DataFrame**. Son colecciones de datos organizados en **columnas**. Conceptualmente, son equivalentes a una tabla en el modelo relacional
 - **DataSet**. Extensión de la API de DataFrames que añade seguridad de tipos (**type-safe**) a los datos organizados en columnas (solo disponible en Scala y Java)

3. Spark Streaming

- Extensión del Core para procesar **datos en streaming** (en tiempo real)
- Usa una técnica llamada micro-batching (agrupación del flujo de datos en tiempo real en pequeños lotes que son procesados individualmente)

2. Apache Spark - Arquitectura

4. **MLlib** (Machine Learning Library)

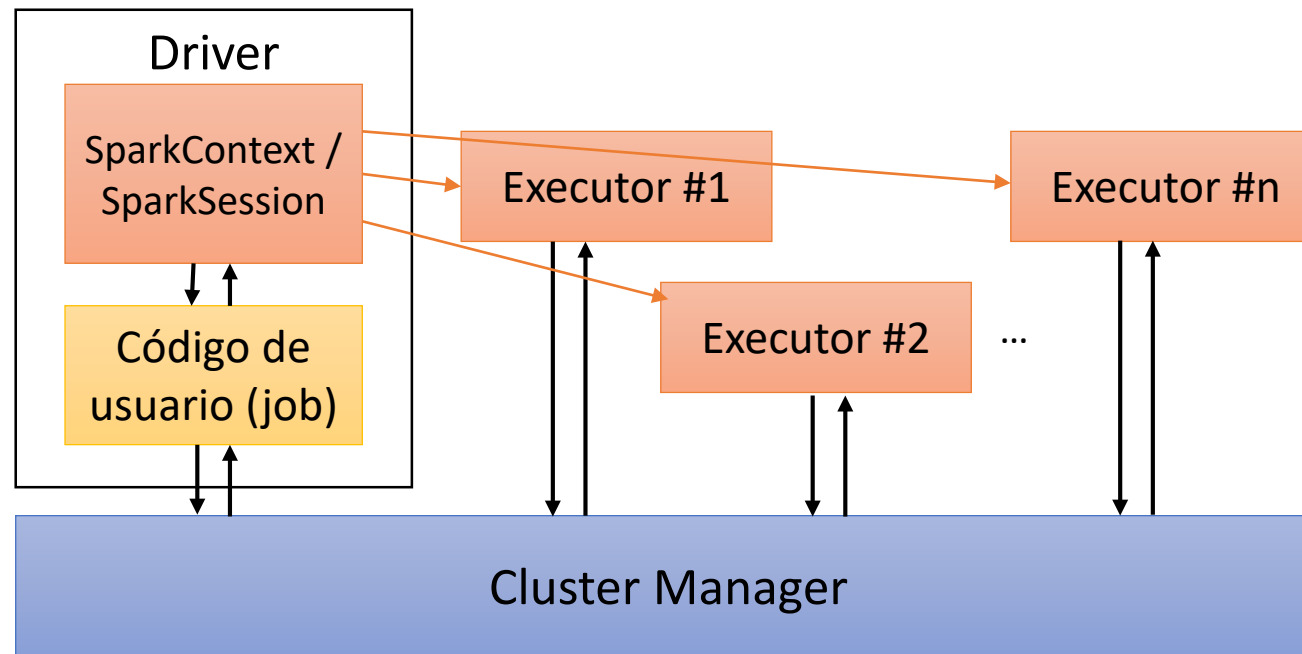
- Librería que implementa diferentes algoritmos de ML sobre clustering, regresión, clasificación y filtrado colaborativo
- Desde la versión 2.0 de Spark, MLlib está basada en la API de DataFrames en lugar de RDD (ya que es más sencilla de usar)

5. **GraphX**

- Librería para el procesamiento de gráficos
- GraphX extiende la API de RDD mediante una nueva abstracción: los grafos dirigidos, en los que podemos representar vértices (nodos) y aristas (arcos)

2. Apache Spark - Arquitectura

- Hay 2 tipos componentes para ejecutar un trabajo (job) en Spark:
 - **Driver**: proceso que coordina la ejecución de un código de usuario a través de diferentes ejecutores
 - **Ejecutor** (executor): proceso encargado de ejecutar una **tarea** específica



2. Apache Spark - Arquitectura

- Cuando se ejecuta Spark en **local**, se utilizan diferentes hilo de ejecución (típicamente se usarán tantos hilos como procesadores lógicos tenga la máquina) para la ejecución del driver y ejecutores
- Cuando se ejecuta Spark en un **clúster**, el driver y los ejecutores se ejecutarán en diferentes nodos
- El **cluster manager** es el componente que gestiona los recursos disponibles. Puede ser de diferentes tipos:
 - Standalone: Sencillo cluster manager incluido con la distribución Spark
 - Apache Mesos: cluster manager open-source (<https://mesos.apache.org/>)
 - Hadoop YARN: Gestor de recursos disponible en Hadoop
 - Kubernetes: Sistema de orquestación de contenedores

2. Apache Spark - Arquitectura

- Hay diferentes formas de ejecutar Spark en un clúster:
 1. En local (on premises), es decir, mantener nuestro propio clúster en máquinas físicas (bare metal) o virtuales (VMs), por ejemplo:
 - Apache Ambari (<https://ambari.apache.org/>) para un clúster Hadoop
 2. Proveedores en la nube (cloud providers), por ejemplo:
 - Amazon EMR (<https://aws.amazon.com/emr/>)
 - Google Dataproc (<https://cloud.google.com/dataproc>)
 3. Productos comerciales (vendor solutions), por ejemplo:
 - Databricks (<https://databricks.com/>):
 - Compañía fundada por los creadores originales de Spark
 - Ofrece una versión gratuita (<https://community.cloud.databricks.com/>)
 - Cloudera (<https://www.cloudera.com/>)

2. Apache Spark - RDDs

- Los RDDs son objetos que permiten manipular colecciones de datos en Spark en **paralelo**:
 - Este proceso en paralelo depende del número de **particiones** (partitions)
 - El número de particiones dependerá de los recursos disponibles (ejecutores)
 - Los RDDs son **inmutables** (no se pueden modificar una vez creados) y siempre están en **memoria**
 - Los RDDs tienen **tipo**, por ejemplo: `RDD[int]`, `RDD[long]`, `RDD[String]`
- Se pueden crear RDDs de tres formas:
 1. Paralelizando una colección de datos existente
 - Por ejemplo, en PySpark: `sc.parallelize(data)`
 2. Desde colecciones de datos externas
 - Por ejemplo, en PySpark: `sc.textFile("file.txt")`
 3. Aplicando transformaciones a otros RDDs

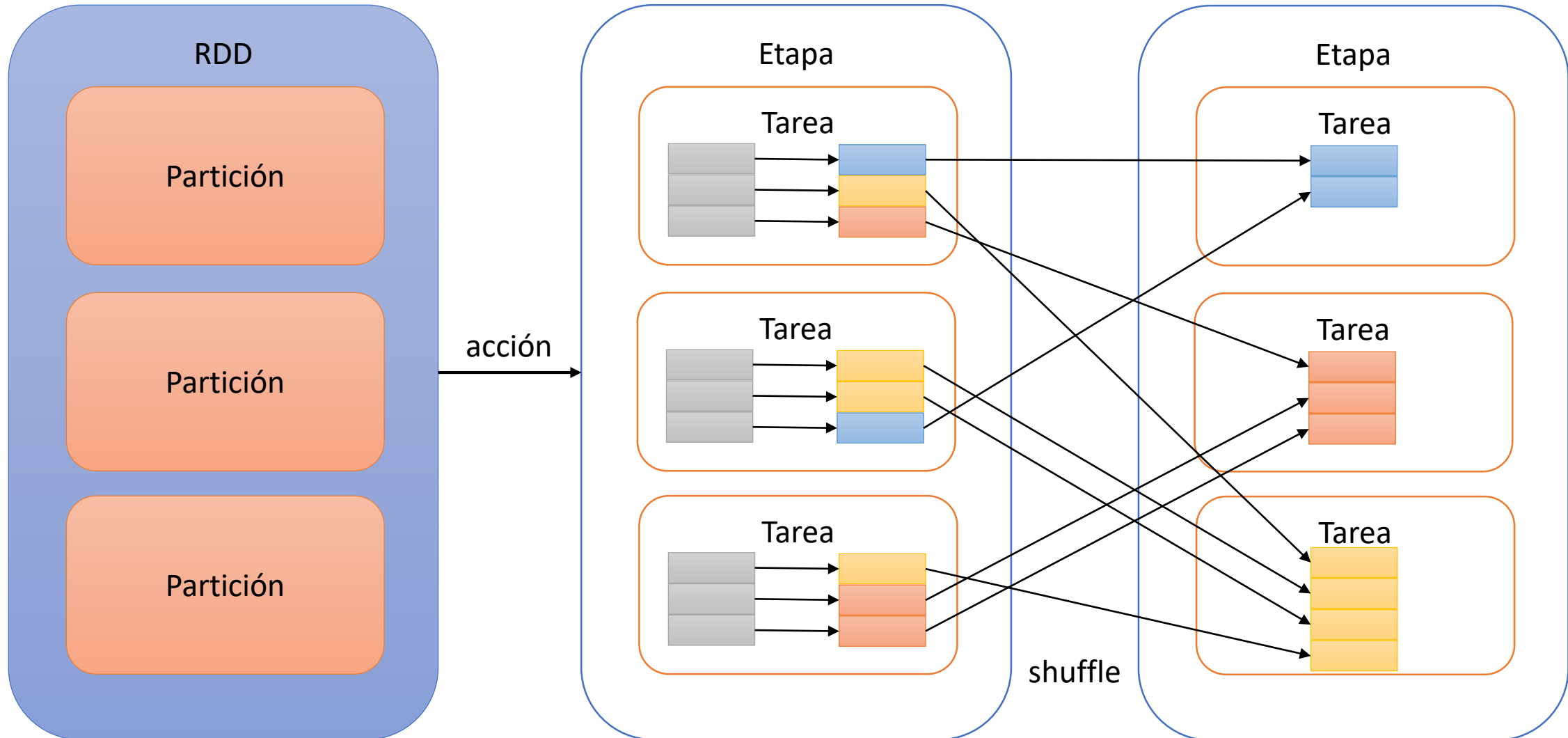
2. Apache Spark - RDDs

- Un trabajo Spark (job) está compuesto por una serie de operaciones sobre objetos RDDs, que puede ser de dos tipos:
 - **Transformaciones:** funciones que produce un nuevo RDD a partir de RDD(s) existentes. Siempre se evalúan de forma perezosa (**lazy evaluation**), es decir, solo se ejecutan cuando se llama a una acción
 - **Acciones:** funciones que se ejecutan sobre los datos contenidos en los RDDs. Permiten mandar resultados desde un ejecutor al driver
- Dependiendo del movimiento de datos entre particiones, se distinguen dos tipos de transformaciones:
 - Estrechas (narrow): Usan datos procedentes de la misma partición
 - Amplias (wide): Usan datos procedentes de diferentes particiones (shuffle)

2. Apache Spark - RDDs

- Un job se ejecuta como un conjunto de **etapas** (stages)
 - Una etapa es la unidad física de ejecución de ejecución en Spark
 - Se crean nuevas etapas en 2 situaciones:
 - Al ejecutar una acción (ResultStage)
 - Al ejecutar una transformación extensa (ShuffleMapStage)
 - Internamente, un job en Spark se organiza usando una estructura de grafo dirigido acíclico (DAG) que recibe el nombre de linaje RDD (**lineage**). Este grafo almacenara una serie de metadatos que proporcionará tolerancia a fallos
- Una etapa está formada N **tareas** (tasks), siendo N el número de particiones del RDD
 - Una tarea es una operación única (por ejemplo, `.map()` o `.filter()`) aplicada a una partición
 - Cada tarea se ejecuta como un hilo independiente (en el mismo o diferente nodo, dependiendo de los recursos disponibles)

2. Apache Spark - RDDs



2. Apache Spark - RDDs

- Algunas de las transformaciones y acciones más comunes son:

Tipo	Operación	Descripción
Transformación estrecha (narrow)	<code>map(f)</code>	Ejecuta una función <code>f</code> a cada elemento
	<code>flatMap(f)</code>	Ejecuta la función <code>map(f)</code> y después aplana los resultados (flatten)
	<code>filter(f)</code>	Selecciona un conjunto de elementos según una función
Transformación amplia (wide)	<code>reduceByKey(f)</code>	Dado un mapa clave-valor de objetos, se devuelve un nuevo mapa agregando los elementos al aplicar una determinada función a las claves
	<code>groupByKey()</code>	Dado un mapa clave-valor de objetos, los agrupa por clave
	<code>union(otherDSS)</code>	Fusiona un DSS con otro DSS
Acción	<code>collect()</code>	Agrega todos los elementos y los envía al driver
	<code>reduce(f)</code>	Agrega todos los elementos en base a una función <code>f</code>
	<code>count()</code>	Cuenta el número de elementos
	<code>foreach(f)</code>	Recorre los elementos y ejecuta una función <code>f</code> a cada elemento

<https://spark.apache.org/docs/latest/rdd-programming-guide.html>

<https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/>

2. Apache Spark - PySpark

- **PySpark** es una API Python para Spark, y proporciona una interfaz para objetos RDDs y DataFrames
- La API de **RDDs** está disponible en PySpark a través del objeto **SparkContext**. Para crear este objeto, se puede especificar:
 - **master**: Cadena que define el tipo de ejecución. Puede tomar los valores:
 - URL (para ejecución en clúster), por ejemplo: `spark://HOST:PORT`, `mesos://HOST:PORT`, `k8s://HOST:PORT`, o `yarn` (para clúster YARN identificado por la variable `YARN_CONF_DIR`)
 - `local[*]`: Para ejecutar Spark de manera local dependiendo del número de procesadores lógicos de la máquina (también se puede fijar un número en lugar de `*`)
 - **appName**: Nombre de nuestra aplicación en el clúster (opcional)
 - **conf**: Objeto de configuración de tipo `SparkConf` (opcional)

```
from pyspark import SparkContext  
  
sc = SparkContext(master, appName, conf)
```

<https://spark.apache.org/docs/latest/api/python/pyspark.html>

2. Apache Spark - PySpark

- Para ejecutar un trabajo Spark con PySpark en local, es necesario tener instalado lo siguiente:

1. Máquina Virtual de **Java** 8+ (JRE o JDK)
2. Intérprete **Python**



- La versión 2 de Python (2.7) fue oficialmente descontinuada el 01/01/2020
- La versión recomendada para nuevo código Python es la 3.x
- No obstante, para ejecutar software antiguo desarrollado en Python 2.x, necesitaremos tener instalados ambos intérpretes. Una opción para conseguir cambiar entre varios entornos (2.x y 3.x) es usar Anaconda (<https://www.anaconda.com/>)
 - Es una distribución open-source de utilidades Python y R para big-data, ML, etc.
 - Usa un gestor de paquetes para Python llamado conda (además de pip)

2. Apache Spark - PySpark

- Si ejecutamos nuestro programa Python directamente como aplicación Python necesitaremos tener instalada la librería PySpark:

```
$ pip install pyspark
```

```
$ python my-spark-app.py
```

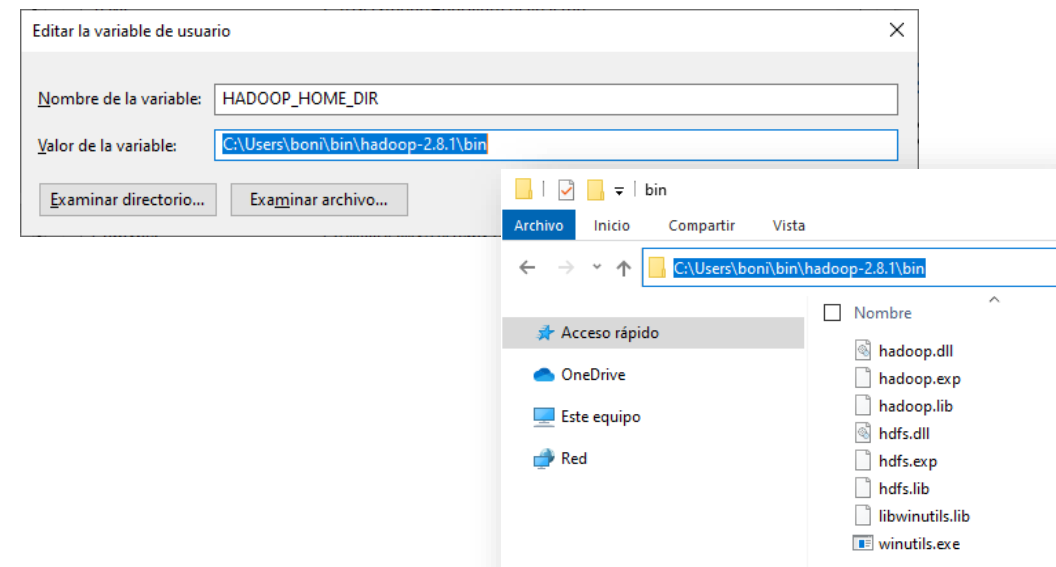
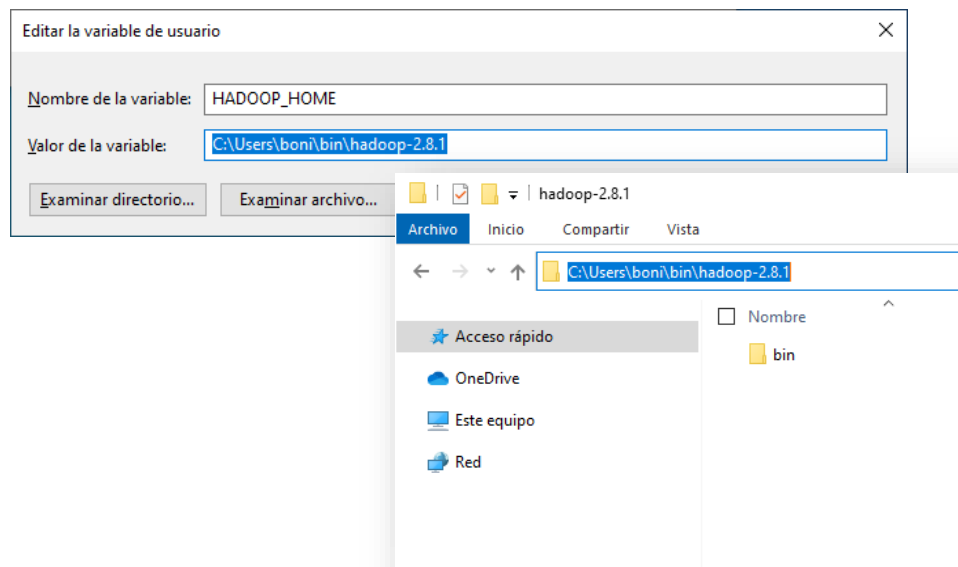
Esta es la forma recomendada para ejecutar los scripts de Python en la máquina virtual

- Alternativamente, podemos descargar la distribución de Spark (<https://spark.apache.org/downloads.html>) y usar el script `bin/spark-submit`:

```
$ spark-submit my-spark-app.py
```

2. Apache Spark - PySpark

- Dado que Spark usa HDFS (el sistema de ficheros de Hadoop), además de Java y Python, es recomendable las utilidades binarias de Hadoop
 - En sistemas Windows (winutils.exe y dll's) se pueden descargar de <https://github.com/steveloughran/winutils>
 - Hay que exportar las siguientes variable de entorno:
 - HADOOP_HOME → Ruta a las utilidades binarias de Hadoop (en esta ruta habrá una carpeta bin que tendrá estas utilidades)
 - HADOOP_HOME_DIR → Ruta a las utilidades binarias de Hadoop



2. Apache Spark - PySpark

Este ejemplo crea un contexto Spark en local para procesar ciertos datos de entrada usando la API de RDDs

La palabra reserva `lambda` en Python se utiliza para definir funciones anónimas

```
double = lambda x: x * 2

# is the same as

def double(x):
    return x * 2
```

```
from pyspark import SparkContext

# Local SparkContext using N threads (N = number of logical processors)
sc = SparkContext(master="local[*]", appName="range-RDD-stdout")

# 1. Input data: list of integers (unstructured batch)
data = range(1, 10001)
print(f"Input data: {len(data)} integers from {data[0]} to {data[9999]}")

# 2. Data processing
# Parallelize input data into a RDD (lazy evaluation) using * partitions
rangeRDD = sc.parallelize(data)
print(f"RDD has been created using {rangeRDD.getNumPartitions()} partitions")

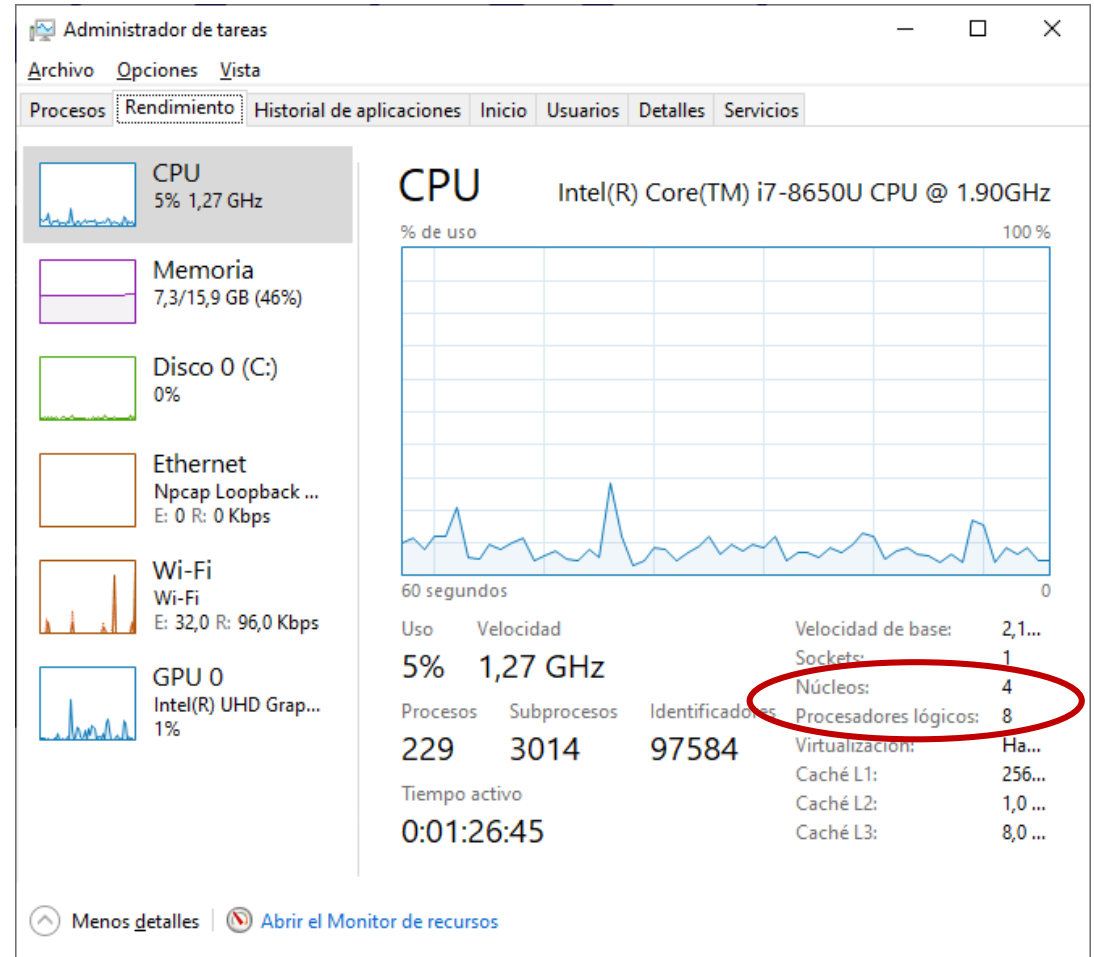
# Process data using lambda functions and collect results:
# 1) subtract 1 to all elements. 2) select those lower than 10.
out = (rangeRDD
      .map(lambda y: y - 1)
      .filter(lambda x: x < 10)
      .collect())

# 3. Output data: show result in the standard output (console)
print(f"The output is {out}")
```

```
Input data has 10000 integers from 1 to 10000
RDD has been created using 8 partitions
The output is [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

2. Apache Spark - PySpark

```
boni@ubuntu: ~  
boni@ubuntu: ~$ lscpu  
Architecture:          x86_64  
CPU op-mode(s):        32-bit, 64-bit  
Byte Order:             Little Endian  
CPU(s):                 8  
On-line CPU(s) list:   0-7  
Thread(s) per core:    2  
Core(s) per socket:    4  
Socket(s):              1  
NUMA node(s):          1  
Vendor ID:              GenuineIntel  
CPU family:             6  
Model:                  142  
Model name:             Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz  
Stepping:               10  
CPU MHz:                800.016  
CPU max MHz:           4200.0000  
CPU min MHz:           400.0000  
BogoMIPS:               4224.00  
Virtualization:        VT-x  
L1d cache:              32K  
L1i cache:              32K  
L2 cache:               256K  
L3 cache:               8192K  
NUMA node0 CPU(s):     0-7
```

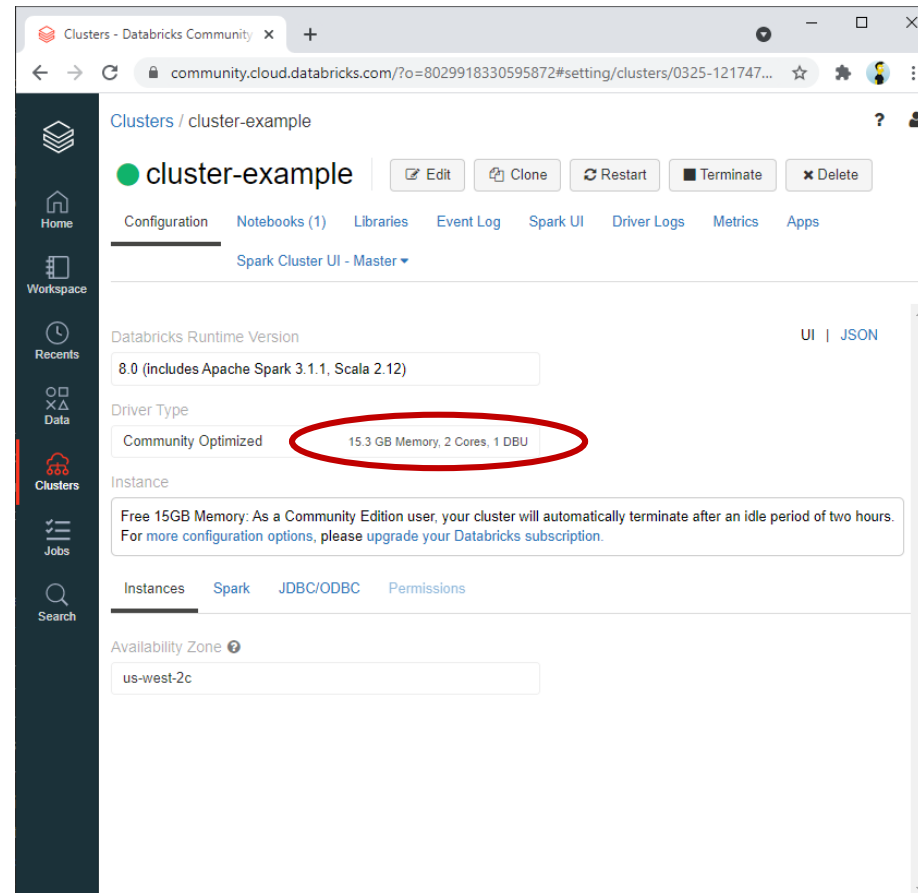


Ejemplo de diferentes formas para chequear el número de procesadores lógicos en Ubuntu y Windows

La diferencia entre el número de núcleos y procesadores lógicos en estos ejemplos es debido al procesamiento multihilo (por ejemplo, Intel *hyper-threading*)

2. Apache Spark - Databricks

- Podemos usar Databricks Community para ejecutar trabajos Sparks en un clúster (de tamaño reducido) de manera gratuita



The screenshot displays the Databricks Community interface for a cluster named 'cluster-example'. The page shows various configuration options and a warning message. A red circle highlights the '15.3 GB Memory, 2 Cores, 1 DBU' specification for the 'Community Optimized' driver type.

Clusters - Databricks Community

community.cloud.databricks.com/?o=8029918330595872#setting/clusters/0325-121747...

Clusters / cluster-example

cluster-example [Edit] [Clone] [Restart] [Terminate] [Delete]

Configuration | Notebooks (1) | Libraries | Event Log | Spark UI | Driver Logs | Metrics | Apps

Spark Cluster UI - Master

Databricks Runtime Version: 8.0 (includes Apache Spark 3.1.1, Scala 2.12) [UI] [JSON]

Driver Type: Community Optimized (15.3 GB Memory, 2 Cores, 1 DBU)

Instance: Free 15GB Memory: As a Community Edition user, your cluster will automatically terminate after an idle period of two hours. For more configuration options, please upgrade your Databricks subscription.

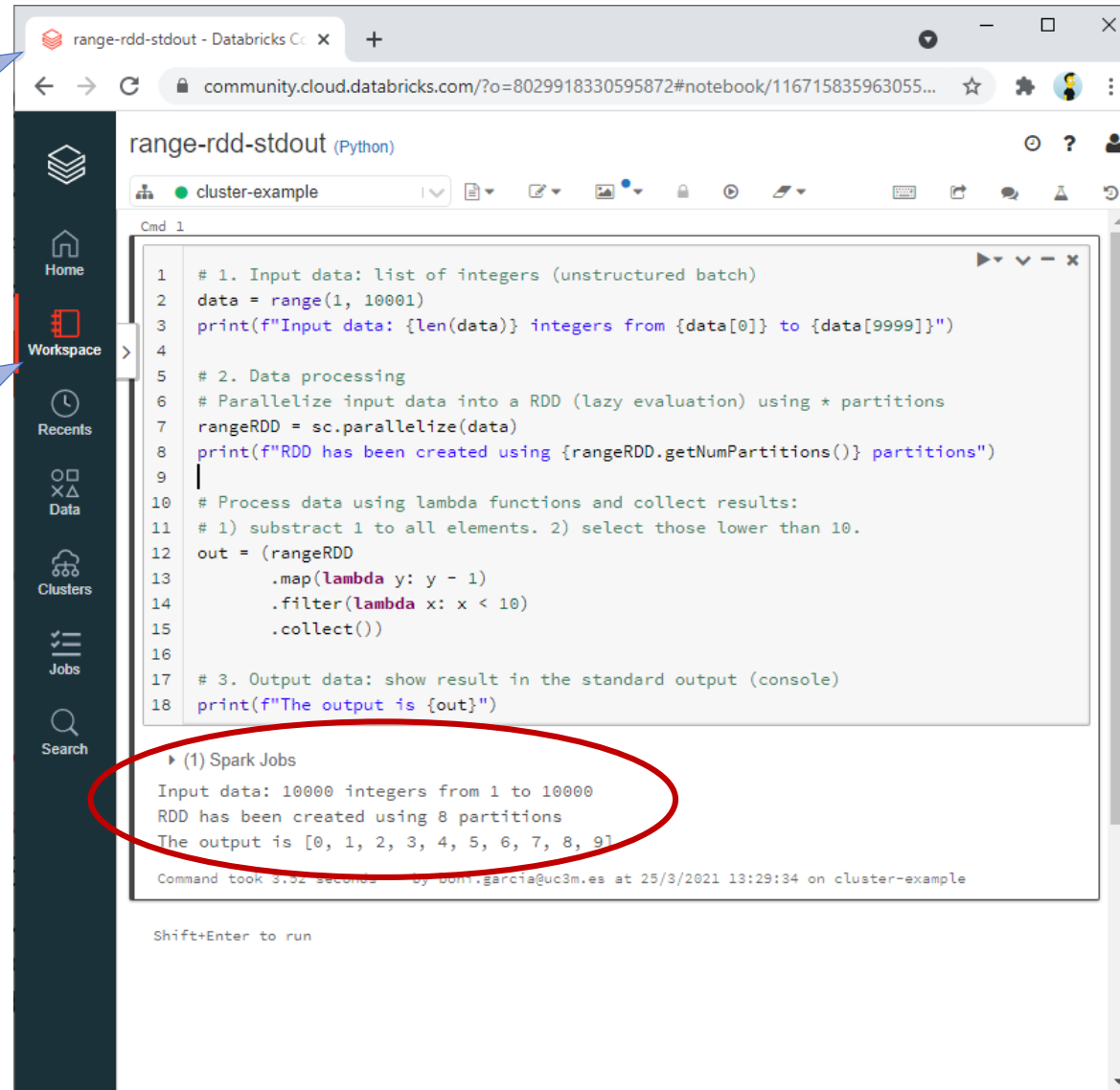
Instances | Spark | JDBC/ODBC | Permissions

Availability Zone: us-west-2c

2. Apache Spark - Databricks

Ejecución del ejemplo anterior (`range-RDD-stdout`) en un Notebook de Databricks Community

Cuando usamos Databricks, el contexto Spark (variable `sc`) está disponible por defecto en el Notebook



The screenshot shows a Databricks Notebook interface. The top navigation bar includes 'Home', 'Workspace', 'Recents', 'Data', 'Clusters', 'Jobs', and 'Search'. The main content area displays a code cell titled 'range-rdd-stdout (Python)'. The code cell contains the following Python code:

```
1 # 1. Input data: list of integers (unstructured batch)
2 data = range(1, 10001)
3 print(f"Input data: {len(data)} integers from {data[0]} to {data[9999]}")
4
5 # 2. Data processing
6 # Parallelize input data into a RDD (lazy evaluation) using * partitions
7 rangeRDD = sc.parallelize(data)
8 print(f"RDD has been created using {rangeRDD.getNumPartitions()} partitions")
9
10 # Process data using lambda functions and collect results:
11 # 1) subtract 1 to all elements. 2) select those lower than 10.
12 out = (rangeRDD
13       .map(lambda y: y - 1)
14       .filter(lambda x: x < 10)
15       .collect())
16
17 # 3. Output data: show result in the standard output (console)
18 print(f"The output is {out}")
```

Below the code cell, the execution output is displayed. A red circle highlights the output text:

```
(1) Spark Jobs
Input data: 10000 integers from 1 to 10000
RDD has been created using 8 partitions
The output is [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The output also includes a timestamp and user information: "Command took 3.52 seconds by boni.garcia@uc3m.es at 25/3/2021 13:29:34 on cluster-example".

2. Apache Spark - Jupyter Notebooks

- También es posible ejecutar jobs Spark en uso Jupyter Notebooks
- Por ejemplo, usando Google Colaboratory (Colab)

```
# Setup
# 1. JDK
!apt-get install openjdk-8-jdk-headless -qq > /dev/null

# 2. Spark
!wget -q https://downloads.apache.org/spark/spark-2.4.7/spark-2.4.7-bin-hadoop2.7.tgz
!tar xf spark-2.4.7-bin-hadoop2.7.tgz

# 3. Envs
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-2.4.7-bin-hadoop2.7"

# 4. PySpark
!pip install -q findspark
import findspark
findspark.init()
```

En primer lugar será necesario configurar las dependencias (Java, Spark, variables de entorno, y PySpark)

2. Apache Spark - Jupyter Notebooks

```
from pyspark import SparkContext

# Local SparkContext using N threads (N = number of logical processors)
sc = SparkContext(master="local[*]", appName="range-RDD-stdout")

# 1. Input data: list of integers (unstructured batch)
data = range(1, 10001)
print(f"Input data: {len(data)} integers from {data[0]} to {data[9999]}")

# 2. Data processing
# Parallelize input data into a RDD (lazy evaluation) using * partitions
rangeRDD = sc.parallelize(data)
print(f"RDD has been created using {rangeRDD.getNumPartitions()} partitions")

# Process data using lambda functions and collect results:
# 1) subtract 1 to all elements. 2) select those lower than 10.
out = (rangeRDD
      .map(lambda y: y - 1)
      .filter(lambda x: x < 10)
      .collect())

# 3. Output data: show result in the standard output (console)
print(f"The output is {out}")
```

```
Input data: 10000 integers from 1 to 10000
RDD has been created using 2 partitions
The output is [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Después, se podrá ejecutar el job Spark en cuestión

2. Apache Spark - DataFrames

- La API de **DataFrames** de Spark nos permite procesar datos estructurados y semiestructurados
 - Inspirada en la API de Python llamada Pandas (<https://pandas.pydata.org/>)
 - Los objetos DataFrame son colecciones de datos **tipados** organizados en **columnas**
 - Internamente, Spark implementa un optimizador para consultas SQL y operaciones con DataFrames llamado Catalyst Optimizer (basado en programación funcional en Scala)
- Podemos crear objetos DataFrame en Spark de las siguientes formas:
 - Cargando datos desde una base de datos
 - Cargando datos desde un fichero externo. Por ejemplo, de un fichero JSON y haciendo inferencia de esquema (infer schema)
 - Desde un objeto DataFrame de Pandas (en Python)
 - Desde un objeto RDD

```
df = spark.createDataFrame(rdd)
```

2. Apache Spark - DataFrames

- La API de DataFrames está disponible en PySpark a través del objeto **SparkSession**
- Para la creación de este objeto se pueden especificar los mismos parámetros que hemos visto en la creación de `SparkSession`, pero esta vez usando el patrón builder):
 - `master`: Forma de ejecutar Spark (en local o usando un clúster)
 - `appName`: Nombre de nuestra aplicación en el clúster (opcional)
 - `conf`: Parejas clave-valor para configuración (opcional)

```
spark = (SparkSession
        .builder
        .master("local|url")
        .appName("app name")
        .config("key", "value")
        .getOrCreate())
```

2. Apache Spark - DataFrames

- La siguiente tabla muestra algunos de los **métodos** más relevantes que se pueden aplicar a objetos de tipo **DataFrame** en PySpark:

Método	Descripción
<code>select(*columns)</code>	Devuelve un nuevo DataFrame leyendo un conjunto de columnas
<code>filter(condition)</code>	Realiza un filtrado en un DataFrame en base a una condición
<code>groupBy(*columns)</code>	Realiza una agregación de un conjunto de columnas
<code>drop("column")</code>	Devuelve un nuevo DataFrame eliminando una columna existente
<code>count()</code>	Devuelve el número de filas contenidos en el DataFrame
<code>alias("column")</code>	Devuelve un nuevo DataFrame con un nombre de columna determinado
<code>collect()</code>	Devuelve los datos contenidos en el DataFrame (lista de objetos tipo Row)
<code>show()</code>	Muestra por la salida estándar el contenido de los DataFrame
<code>printSchema()</code>	Muestra por la salida estándar el esquema de los DataFrame

2. Apache Spark - DataFrames

- La siguiente tabla muestra algunos de las **funciones** más relevantes que se pueden usar con objetos de tipo **DataFrame** en PySpark:

Método	Descripción
<code>explode()</code>	Devuelve una lista de objetos de tipo fila (Row) para cada uno de los elementos de un array o Lista
<code>split(str, pattern)</code>	Crea una colección de elementos en base a una cadena de entrada y un patrón basado en una expresión regular
<code>lit("column")</code>	Crea un objeto de tipo Column con un nombre determinado
<code>udf(f)</code>	Crea una función definida por el usuario (UDF, User Defined Function), que es una función que extiende las funciones disponibles en Spark SQL para realizar tratamientos con objetos DataFrames
<code>regexp_replace(str, pattern, replacement)</code>	Reemplaza todas las ocurrencias que cumplan una expresión regular (regex) en una cadena de texto

2. Apache Spark - DataFrames

Este ejemplo usa una SparkSession en local para crear un objeto DataFrame partiendo de unos datos en JSON y después realiza algunas operaciones que se muestran por la salida estándar

```
[
  {
    "name": "Michael"
  },
  {
    "name": "Andy",
    "age": 30
  },
  {
    "name": "Justin",
    "age": 19
  }
]
```

```
from pyspark.sql import SparkSession

# SparkSession (for structured data) in local
spark = (SparkSession
         .builder
         .master("local[*]")
         .appName("JSONfile-DataStream-StdOut")
         .getOrCreate())

# 1. Input data: JSON file
df = spark.read.json("../data/people.json", multiLine=True)

df.show() # Displays the content to stdout
df.printSchema() # Displays the schema to stdout

# 2. Data processing: different queries
names = df.select("name")
allInc = df.select(df["name"], df["age"] + 1)
older21 = df.filter(df["age"] > 21)
countByAge = df.groupBy("age").count()

# 3. Output data: show result in the console
names.show()
allInc.show()
older21.show()
countByAge.show()
```

```
+---+-----+
| age|  name|
+---+-----+
| null|Michael|
|  30|  Andy|
|  19| Justin|
+---+-----+
```

```
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)
```

```
+-----+
|  name|
+-----+
|Michael|
|  Andy|
| Justin|
+-----+
```

```
+-----+-----+
|  name|(age + 1)|
+-----+-----+
|Michael|    null|
|  Andy|    31|
| Justin|    20|
+-----+-----+
```

```
+---+-----+
| age| name|
+---+-----+
| 30|Andy|
+---+-----+
```

```
+---+-----+
| age|count|
+---+-----+
| 19|    1|
| null|    1|
| 30|    1|
+---+-----+
```

2. Apache Spark - DataFrames

```
range-rdd-stdout - Databricks Co x +
community.cloud.databricks.com/?o=8029918330595872#notebook/1167158359630559/comman...

range-rdd-stdout (Python)
cluster-example

Cmd 2
1 | sc

Out[5]:
SparkContext
Spark UI
Version
v3.1.0
Master
local[8]
AppName
Databricks Shell

Command took 0.49 seconds -- by boni.garcia@uc3m.es at 25/3/2021 14:14:09 on cluster-example

Cmd 3
1 | spark

Out[6]:
SparkSession - hive
SparkContext
Spark UI
Version
v3.1.0
Master
local[8]
AppName
Databricks Shell

Command took 0.65 seconds -- by boni.garcia@uc3m.es at 25/3/2021 14:14:20 on cluster-example
```

La sesión Spark (variable spark), usada en la API de **Dataframes**, también estará disponible por defecto en los Notebook ejecutados en Databricks

Contenidos

0. Presentación
1. Introducción
2. Apache Spark
- 3. Spark Streaming**
 - Pipeline de datos
 - DStreams
 - Streaming estructurado
 - Usos avanzados
4. Resumen

3. Spark Streaming

- Spark Streaming es una extensión del Spark Core introducida en 2013 que permite procesar **datos en streaming** (en tiempo real) de forma escalable, eficiente, y con tolerancia a fallos
- Los datos en streaming se pueden recibir desde diferentes fuentes:
 - Sistemas software: por ejemplo, logs o estadísticas de uso
 - Sistemas de telemetría: por ejemplo, centrales eléctricas o plantas químicas
 - Dispositivos IoT: por ejemplo, sensores o agentes inteligentes
- Las ventajas de Spark Streaming frente a otras alternativas (como por ejemplo, Apache Storm) son:
 - Proporciona un modelo unificado para batch y streaming (APIs DStream y streaming estructurado)
 - Permite integración con el resto de capacidades de Spark (ML, SQL, análisis)
 - Alto rendimiento, tolerancia a fallos

3. Spark Streaming - Pipeline de datos

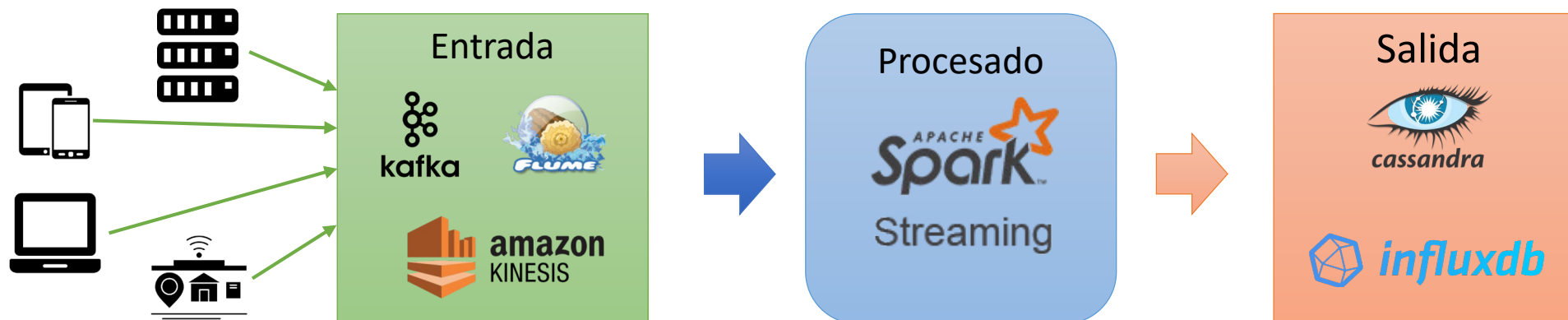
- La secuencia de pasos llevada a cabo en el procesado de datos se conoce como pipeline de datos (**data pipeline**). En procesado batch, es muy común el pipeline en 3 etapas llamado **ETL** (Extract, Transform, Load) :
 1. Entrada de datos, desde una fuente de datos (*data source*)
 2. Procesado de datos, aplicando una serie de funciones a los datos de entrada para generar datos de salida (resultados)
 3. Entrega de resultados, hacia una sistema de ficheros, una base de datos (*database*), un almacén de datos (*data warehouse*), u otro tipo de sistema de salida (*downstream system*)

Los almacenes de datos (por ejemplo, Teradata o Amazon RedShift) proporcionan servicios de almacenamiento y análisis de datos



3. Spark Streaming - Pipeline de datos

- Spark Streaming sigue un proceso equivalente:
 1. Recepción continua de datos de entrada (*data ingestion*), que pueden provenir de diferentes tipos fuentes:
 - Básicas: sistema de ficheros y sockets TCP
 - Avanzadas: Apache Kafka, Apache Flume, y Amazon Kinesis
 2. Procesado con Spark Streaming
 3. Entrega de datos de resultados (datos de salida) para:
 - Almacenamiento: por ejemplo, a un sistema de ficheros o una base de datos
 - Monitorización: visualización (por ejemplo, en un live dashboard), alertas, etc.



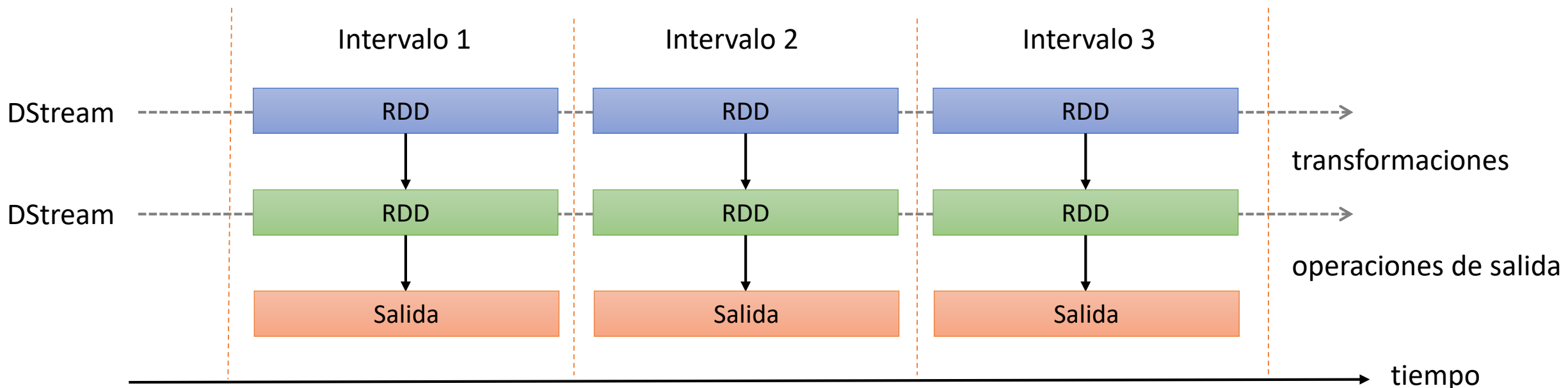
3. Spark Streaming - DStreams

- Spark Streaming usa una técnica llamada **micro-batching**, que consiste en la agrupación de datos en tiempo real en pequeños lotes (*batch*) que son procesados individualmente
- La API proporcionada por Spark Streaming para implementar esta técnica se conoce como **DStream** (Discretized Stream)
 - Representa un conjunto de datos dividido en lotes
 - Internamente es una secuencia de RDDs, de forma que Spark Streaming se pueden integrar con el resto de componentes Spark (Spark SQL, MLlib, etc.)



3. Spark Streaming - DStreams

- Cuando desarrollamos una aplicación con Spark Streaming, un parámetro muy importante será el **intervalo batch**
 - Es el tiempo en lo que se procesan los DStreams como procesos batch
 - Hay que elegir con cuidado el valor de este intervalo de forma que los procesos batch pueden ser procesados dentro del intervalo (dependerá de los requisitos de nuestra aplicación y de los recursos disponibles)



3. Spark Streaming - DStreams

- Cualquier operación que se aplique a un objeto DStream, se realiza a la lista de objetos RDD que contiene
- Spark Streaming clasifica las operaciones en 2 tipos:
 - 1. Transformaciones:** Operaciones que permiten la modificación de DStream de entrada. Hay 2 tipos de transformaciones:
 - a) Sin estado (*stateless*): Si un micro-batch no tiene ninguna dependencia con el anterior. Son equivalentes a las transformaciones de la API de RDD
 - b) Con estado (*stateful*): Se utilizan datos procedentes de micro-batches anteriores
 - 2. Operaciones de salida:** entrega de resultados (*push*) a sistemas de salida. Son equivalentes a las acciones en la API de RDD

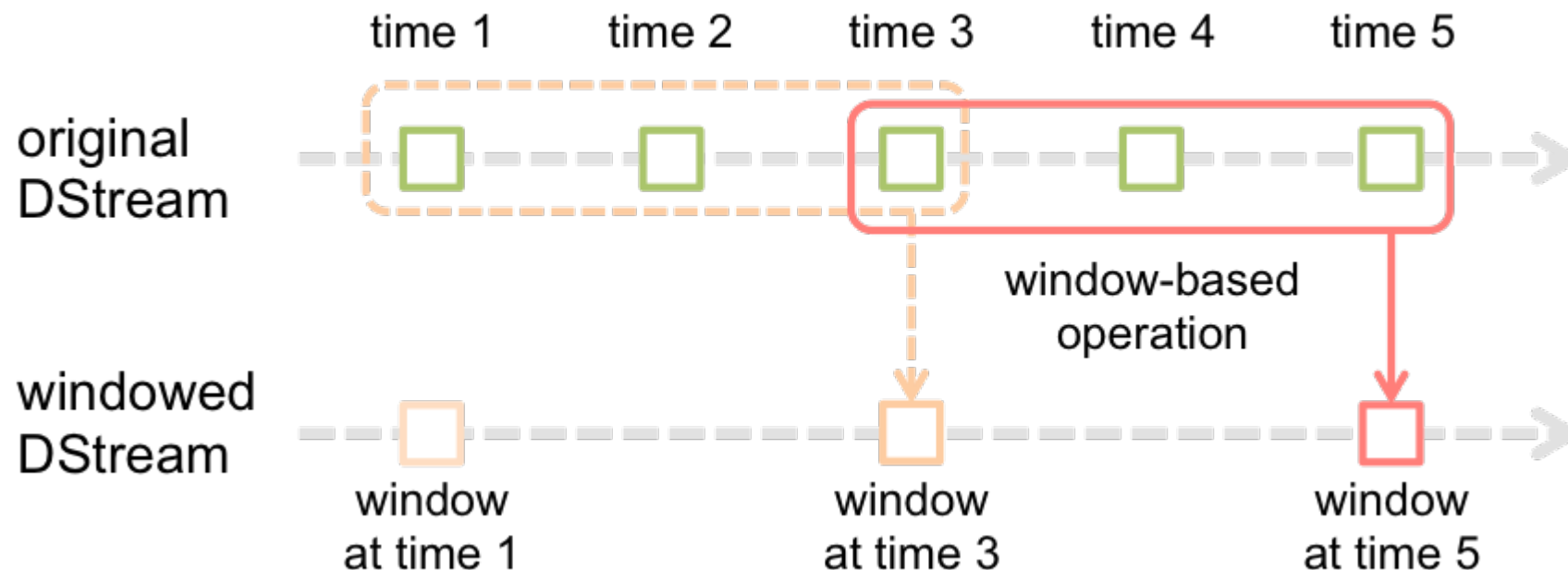
3. Spark Streaming - DStreams

- Algunas de las transformaciones y operaciones de salida más comunes son:

Tipo	Operación	Descripción
Transformación sin estado	<code>map(f)</code>	Ejecuta una función <code>f</code> a todos los elementos
	<code>flatMap(f)</code>	Ejecuta la función <code>map(f)</code> y después aplana los resultados (<code>flatten</code>)
	<code>filter(f)</code>	Selecciona un conjunto de elementos según una función
	<code>reduceByKey(f)</code>	Dado un mapa clave-valor, se devuelve un nuevo mapa al aplicar una determinada función acumulativa a los valores combinados por clave
	<code>countByValue()</code>	Devuelve la frecuencia de aparición de los datos
Transformación con estado	<code>window(length, slideInterval)</code>	Devuelve un DStream que agrupa un conjunto de DStream originales
Operación de salida	<code>pprint()</code>	Muestra los 10 primeros elementos de un DStream en la salida estándar del driver ejecutando la aplicación
	<code>saveAsTextFiles(prefix, [suffix])</code>	Guarda el contenido de un DStream como fichero de texto con el nombre: <code>prefix-TIME_IN_MS[.suffix]</code>
	<code>foreachRDD(f)</code>	Aplica la función <code>f</code> a todos los RDD del DStream

3. Spark Streaming - DStreams

- La transformación `window()` es un ejemplo de transformación con estado porque usa datos procedente de procesos batch anteriores:



3. Spark Streaming - DStreams

- El objeto que nos permite trabajar con DStreams en PySpark se llama **StreamingContext**
- Para crear este objeto, necesitamos especificar:
 - `sc`: Contexto Spark previamente creado
 - `batchInterval`: Intervalo batch (número de segundos en lo que se procesan los DStreams como procesos batch)

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

sc = SparkContext(master, appName, conf)
ssc = StreamingContext(sc, batchInterval)
```


3. Spark Streaming - DStreams

- La secuencia típica de un job con DStreams en PySpark es:
 1. Crear contexto de streaming (por ejemplo, `ssc`)
 2. Definir entrada de streaming creando objetos `DStream`
 3. Definir transformaciones y operaciones de salida en los objetos `DStream`
 4. Comenzar la recepción de datos: `ssc.start()`
 5. Esperar a la terminación del proceso: `ssc.awaitTermination()`
 6. Terminar el proceso mediante (típicamente con Ctrl+C)

3. Spark Streaming - DStreams

- Las fuentes de datos **básicas** en PySpark se crean usando los siguientes métodos del contexto de streaming (objeto `StreamingContext`):

Fuente	Descripción	Código Python
Sistema de ficheros	Lectura de datos disponibles en un sistema de ficheros compatible con la API HDFS, como por ejemplo HDFS, S3, o NFS (en la API de Python solo se pueden leer ficheros de texto)	<code>ssc.textFileStream(dataDirectory)</code>
Sockets TCP	Entrada de texto a través de una conexión TCP (usada para pruebas)	<code>ssc.socketTextStream("hostname", port)</code>

3. Spark Streaming - DStreams

- Este ejemplo crea un **DStream** cuya entrada es un socket TCP por el que recibe texto
- Cada 5 segundos se procesan los lotes
- El procesamiento de cada lote (batch) consiste en contar las palabras de cada línea y mostrar el resultado por la salida estándar

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

# Local SparkContext
sc = SparkContext(master="local[*]", appName="Socket-DStream_WordCount-StdOut")
sc.setLogLevel("ERROR")

# StreamingContext with a batch interval of 5 seconds
ssc = StreamingContext(sc, 5)

# 1. Input data: create a DStream that receives data from a socket
stream = ssc.socketTextStream("localhost", 9999)

# 2. Data processing: word count
words = stream.flatMap(lambda line: line.split(" "))
pairs = words.map(lambda word: (word, 1))
wordCount = pairs.reduceByKey(lambda x, y: x + y)

# 3. Output data: show result in the console
# Print the word count of each RDD generated in this DStream
wordCount.pprint()

ssc.start() # Start the computation
ssc.awaitTermination() # Wait for the computation to terminate
```

x = valor acumulado
y = valor actual

3. Spark Streaming - DStreams

- Para ejecutar este ejemplo usaremos NetCat (nc), una utilidad de línea de comandos que permite realizar conexiones TCP/UDP, tanto como cliente como servidor

En sistemas UNIX-like (Mac OS, Linux) NetCat es una herramienta estándar de la consola de comandos

```
$ nc -lk 9999  
hello world
```

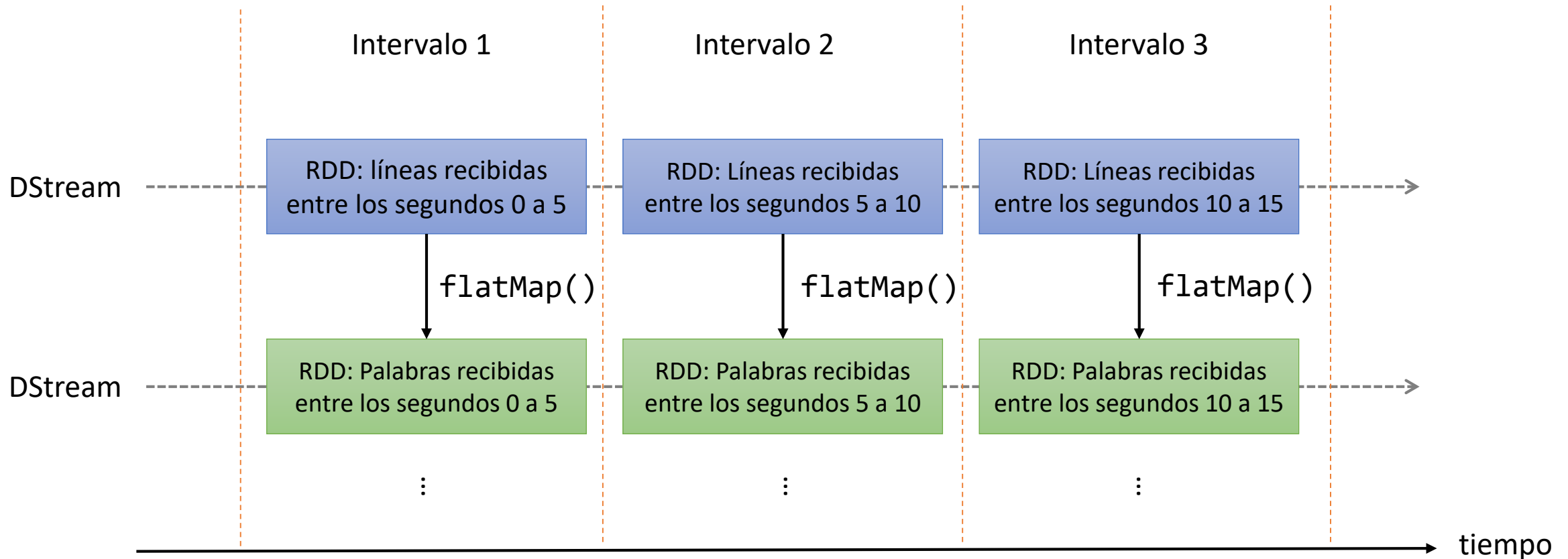
En sistemas Windows hay que instalar NetCat como herramienta adicional de consola de comandos (por ejemplo de esta [página](#))

```
$ nc -L -p 9999  
hello world
```

```
$ python socket-dstream_wordcount-stdout.py  
...  
-----  
Time: 2020-03-25 10:13:00  
-----  
-----  
Time: 2020-03-25 10:13:05  
-----  
-----  
( 'hello', 1 )  
( 'world', 1 )  
-----  
-----  
Time: 2020-03-25 10:13:10  
-----
```

3. Spark Streaming - DStreams

- En este ejemplo, se crean un RDD cada 5 segundos que es procesado mediante `flatMap()`, `map()`, y `reduceByKey()`



3. Spark Streaming - DStreams

- Este ejemplo crea un **DStream** cuya entrada son ficheros de texto de una carpeta del sistema de ficheros
- Cada 5 segundos se procesan los RDDs
- Se cuentan las palabras de los ficheros, y se escriben los resultados en la salida estándar

```
import sys

from pyspark import SparkContext
from pyspark.streaming import StreamingContext

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print(f"Usage: {sys.argv[0]} <input-folder>",
              file=sys.stderr)
        sys.exit(-1)

    # Local SparkContext and StreamingContext (batch interval of 5 seconds)
    sc = SparkContext(master="local[*]",
                     appName="FileSystem-DStream-StdOut")
    sc.setLogLevel("ERROR")
    ssc = StreamingContext(sc, 5)

    # 1. Input data: create a DStream that read text files from the file system
    inputfolder = sys.argv[1]
    stream = ssc.textFileStream("file://" + inputfolder)

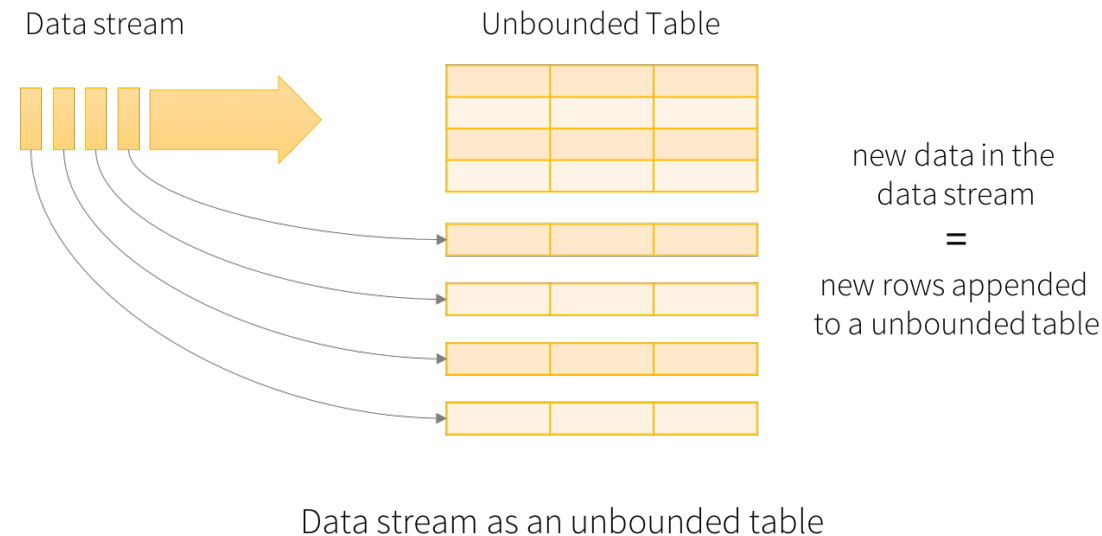
    # 2. Data processing: word count
    wordCount = (stream.flatMap(lambda line: line.split(" "))
                 .map(lambda word: (word, 1))
                 .reduceByKey(lambda x, y: x + y))

    # 3. Output data: show result in the standard output
    wordCount.pprint()

    ssc.start() # Start the computation
    ssc.awaitTermination() # Wait for the computation to terminate
```

3. Spark Streaming - Streaming estructurado

- Spark Streaming también permite enviar y recibir datos estructurados usando DataFrames (**streaming estructurado**)
- El flujo de datos se va añadiendo a una tabla ilimitada (***unbounded table***) en la que se van añadiendo nuevas filas según llegan datos



3. Spark Streaming - Streaming estructurado

- Para **crear** un DataFrame en streaming será necesario invocar el método **readStream** de una sesión Spark (`SparkSession`). Habrá que especificar el formato de la entrada:
 - **socket**: Datos recibidos de una conexión TCP (para pruebas)
 - **rate**: Datos generados en base a un contador de números enteros y una marca de tiempo (para pruebas)
 - **kafka**: Datos recibidos de Apache Kafka
 - **file**: Datos disponibles en un sistema de ficheros compatible con la API HDFS, como por ejemplo HDFS, S3, o NFS
- El **procesado** se hará en base a los mismos método y funciones que se usan para el procesado batch de DataFrames (`select()`, `filter()`, etc.)
- Para obtener **resultados**, hay que generar una **consulta** invocando el método **writeStream** del objeto DataFrame
 - Podemos realizar la consulta de forma indefinida (hasta recibir la señal de terminación) invocando el método `awaitTermination()`

3. Spark Streaming - Streaming estructurado

- En la invocación de `writeStream` (consulta) se generará una **tabla de resultados**. En esta invocación, como mínimo hay que especificar:
 1. **Modo** para la generación esta tabla (método `outputMode()`):
 - `append`: Se generan solo las filas nuevas desde la última vez (modo por defecto)
 - `update`: Se generan solo las filas que han cambiado desde la última vez
 - `complete`: Se genera la tabla completa (solo válido si hay agregación de datos)
 2. **Formato** de tabla de resultados (método `format()`):
 - `console`: Para la salida estándar
 - `kafka`: Para Apache Kafka
 - `org.apache.spark.sql.cassandra`: Para Apache Cassandra

3. Spark Streaming - Streaming estructurado

- Este ejemplo crea un **DataFrame en streaming** usando una secuencia de pruebas (**rate**) cada segundo
- No realiza ningún tipo de procesamiento (muestra por la secuencia según se genera)

```
Batch: 0
-----+-----+
|timestamp|value|
+-----+-----+
-----+-----+

Batch: 1
-----+-----+
|          timestamp|value|
+-----+-----+
|2021-04-08 17:27:...|  0|
+-----+-----+

Batch: 2
-----+-----+
|          timestamp|value|
+-----+-----+
|2021-04-08 17:27:...|  1|
+-----+-----+

Batch: 3
-----+-----+
|          timestamp|value|
+-----+-----+
|2021-04-08 17:27:...|  2|
+-----+-----+
```

```
from pyspark.sql import SparkSession

# Local SparkSession
spark = (SparkSession
        .builder
        .master("local[*]")
        .appName("Rate-DataFrame_Nothing-StdOut")
        .getOrCreate())

# 1. Input data: test DataFrame with sequence and timestamp
df = (spark
     .readStream
     .format("rate")
     .option("rowsPerSecond", 1)
     .load())

# 2. Data processing: nothing

# 3. Output data: show results in the console
query = (df
        .writeStream
        .outputMode("append")
        .format("console")
        .start())

query.awaitTermination()
```

3. Spark Streaming - Streaming estructurado

- Este ejemplo crea un **DataFrame en streaming** usando una secuencia de pruebas (**rate**) cada segundo
- Realiza un filtrado de los valores, eliminando los valores impares

```

-----
Batch: 0
-----
+-----+-----+
|timestamp|value|
+-----+-----+
+-----+-----+

-----
Batch: 1
-----
+-----+-----+
|          timestamp|value|
+-----+-----+
|2020-04-16 13:22:...|  0|
+-----+-----+

-----
Batch: 2
-----
+-----+-----+
|timestamp|value|
+-----+-----+
+-----+-----+

-----
Batch: 3
-----
+-----+-----+
|          timestamp|value|
+-----+-----+
|2020-04-16 13:22:...|  2|
+-----+-----+

```

```

from pyspark.sql import SparkSession

# Local SparkSession
spark = (SparkSession
        .builder
        .master("local[*]")
        .appName("Rate-DataFrame_Filter-StdOut")
        .getOrCreate())

# 1. Input data: test DataFrame with sequence and timestamp
df = (spark
      .readStream
      .format("rate")
      .option("rowsPerSecond", 1)
      .load())

# 2. Data processing: filter odd values
even = df.filter(df["value"] % 2 == 0)

# 3. Output data: show results in the console
query = (even
        .writeStream
        .outputMode("append")
        .format("console")
        .start())

query.awaitTermination()

```

3. Spark Streaming - Streaming estructurado

- Este ejemplo crea un **DataFrame en streaming** usando un socket TCP como entrada de datos
- No realiza ningún tipo de procesado, y muestra los resultados por la salida estándar según llegan

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import length

# Local SparkSession
spark = (SparkSession
        .builder
        .master("local[*]")
        .appName("Socket-DataFrame_Nothing-StdOut")
        .getOrCreate())

# 1. Input data: DataFrame representing the stream of input lines from socket
df = (spark
      .readStream
      .format("socket")
      .option("host", "localhost")
      .option("port", 9999)
      .load())

# 2. Data processing: nothing

# 3. Output data: show result in the console
query = (df
        .writeStream
        .outputMode("append")
        .format("console")
        .start())

query.awaitTermination()
```

3. Spark Streaming - Streaming estructurado

- Este ejemplo modifica el anterior para realizar un procesado en el que se añade una nueva columna a la tabla de resultados con la **longitud** de las cadenas de texto recibidas por el socket

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import length

# Local SparkSession
spark = (SparkSession
        .builder
        .master("local[*]")
        .appName("Socket-DataFrame_Length-StdOut")
        .getOrCreate())

# 1. Input data: DataFrame representing the stream of input lines from socket
df = (spark
     .readStream
     .format("socket")
     .option("host", "localhost")
     .option("port", 9999)
     .load())

# 2. Data processing: add column with the length of each line
dfLen = df.withColumn("length", length(df["value"]))

# 3. Output data: show result in the console
query = (dfLen
        .writeStream
        .outputMode("append")
        .format("console")
        .start())

query.awaitTermination()
```

3. Spark Streaming - Streaming estructurado

- Este ejemplo modifica el anterior para realizar un procesado que consiste en **contar las palabras** de cada línea de texto que llega por el socket y mostrar la tabla de resultados **completa** por la salida estándar
 - Se modifica el valor del parámetro `spark.sql.shuffle.partition`, que determina el número de particiones a usar en operaciones group o join (valor por defecto 200)

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode, split

# Local SparkSession
spark = (SparkSession
        .builder
        .master("local[*]")
        .appName("Socket-DataFrame_WordCount-StdOut")
        .config("spark.sql.shuffle.partitions", "2")
        .getOrCreate())

# 1. Input data: DataFrame representing the stream of input lines from socket
df = (spark
     .readStream
     .format("socket")
     .option("host", "localhost")
     .option("port", 9999)
     .load())

# 2. Data processing: word count
words = df.select(
    explode(
        split(df.value, " ")
    ).alias("word")
)
wordCount = words.groupBy("word").count()

# 3. Output data: show result in the console
# Print the word count in "complete" mode (entire table)
query = (wordCount
        .writeStream
        .outputMode("complete")
        .format("console")
        .start())

query.awaitTermination()
```

3. Spark Streaming - Streaming estructurado

- Para ejecutar este ejemplo volemos a usar NetCat (nc)

UNIX-like

```
$ nc -lk 9999
cat dog
dog dog
owl cat
dog
owl
```

Windows

```
$ nc -L -p 9999
cat dog
dog dog
owl cat
dog
owl
```

```
$ python socket-dataframe_wordcount-stdout.py
```

```
...
```

```
Batch: 0
```

```
+---+---+
|word|count|
+---+---+
+---+---+
```

```
Batch: 1
```

```
+---+---+
|word|count|
+---+---+
| dog|    3|
| cat|    1|
+---+---+
```

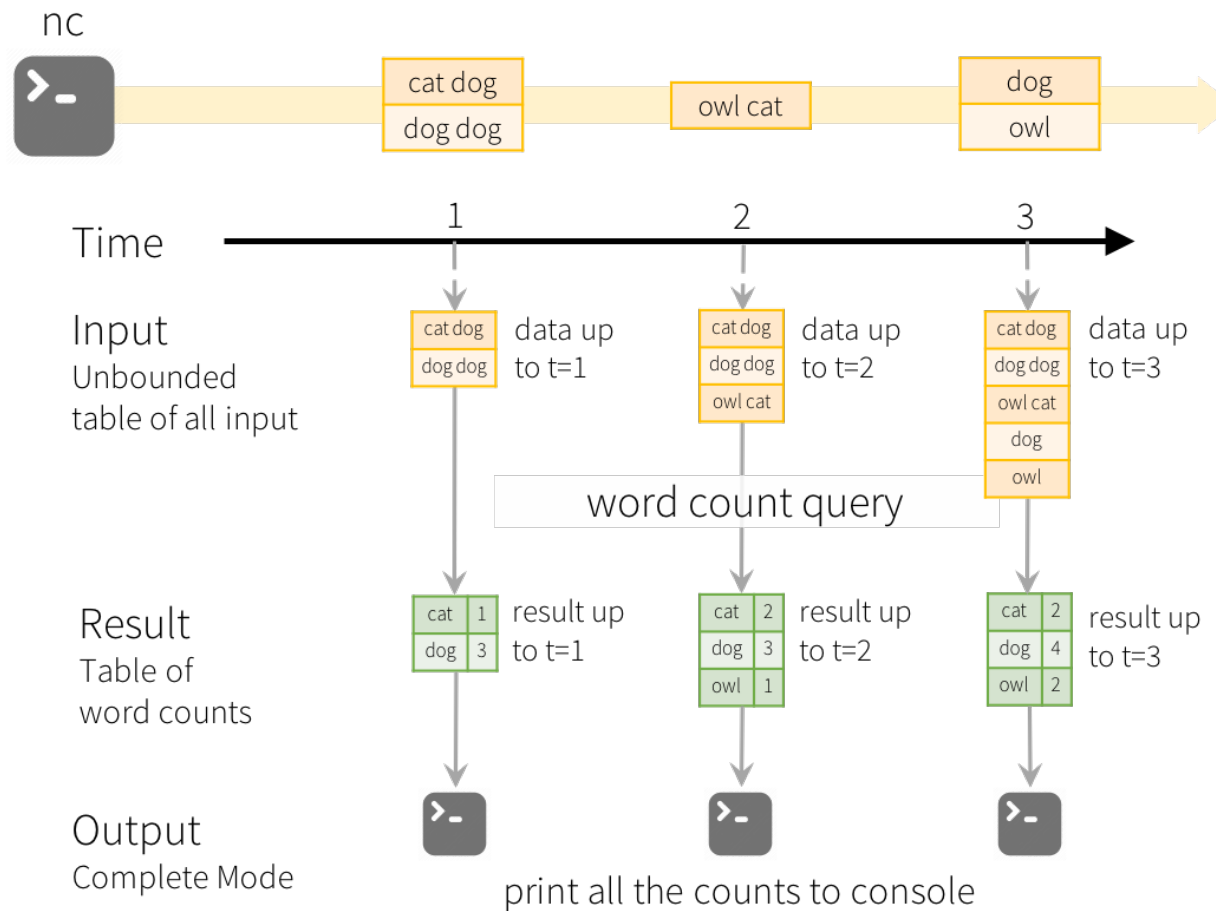
```
Batch: 2
```

```
+---+---+
|word|count|
+---+---+
| dog|    3|
| cat|    2|
| owl|  1|
+---+---+
```

```
Batch: 3
```

```
+---+---+
|word|count|
+---+---+
| dog|    4|
| cat|    2|
| owl|  2|
+---+---+
```

3. Spark Streaming - Streaming estructurado



3. Spark Streaming - Usos avanzados

- Las siguientes características (features) también están disponibles en Spark Streaming, pero no las vemos en clase:
 - Operaciones con MLlib
 - Ajuste de rendimiento
 - Desplegar aplicaciones en clúster
 - Monitorización de aplicaciones
 - Tolerancia a fallos
 - Caché y checkpoints
 - Acumuladores y variables broadcast
 - ...

<https://spark.apache.org/docs/latest/>

Contenidos

0. Presentación
1. Introducción
2. Apache Spark
3. Spark Streaming
4. Resumen

4. Resumen

- **Apache Spark** es un framework de alto rendimiento que permite el procesado de datos no estructurados (mediante la API de **RDDs**) y estructurados (mediante la API de **DataFrames**)
- **Spark Streaming** es una extensión que permite procesar datos en tiempo real (streaming) mediante una técnica llamada micro-batching
- La abstracción fundamental en Spark Streaming se conoce **DStream**, que es un conjunto de RDDs que son procesados con Spark
- Spark Streaming también permite procesar DataFrames dinámicos en base a un flujo de datos (**streaming estructurado**)
- La API **PySpark** permite el desarrollo de aplicaciones Spark en Python, tanto para procesados batch como en streaming