

Apache Spark



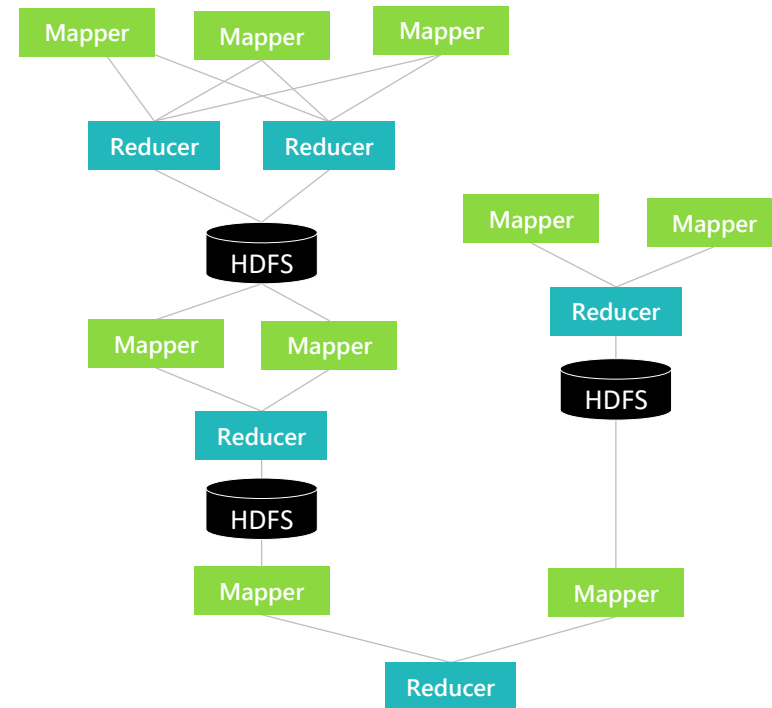
Agenda

- La motivación y nacimiento de SPARK
- Spark Shell y Notebooks
- Arquitectura SPARK
- Spark RDD
- Spark SQL
- Spark ML
- Spark Streaming

Map Reduce

Ventajas

- Framework simple para computación distribuida
- Toda computación debe expresarse como una serie de pasos definidos por dos Operaciones simples:
 - *Mappers*
 - *Reducers*
- Trabajos en batch que pueden expresarse como un data flow simple
- Solidez y Tolerancia a fallos se consiguen por la naturaleza del gráfico acíclico del flujo:
 - Cada ronda de map-reduce es independiente de otras y puede reconstruirse si se pierde
 - La comunicación se gestiona a través de los extremos del flujo



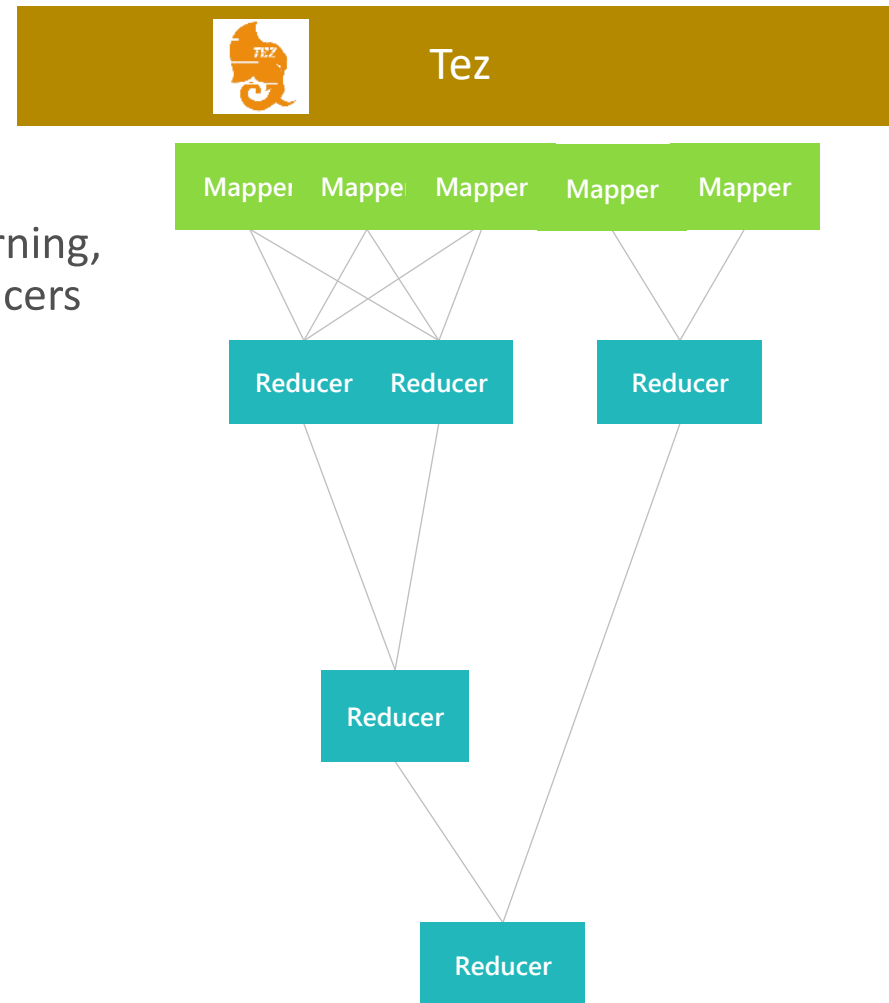
Map Reduce

Desventajas

- Cada paso reduce require escribir a disco
- Para trabajos iterativos, como modelos de Machine Learning, una implementación MapReduce tendría multiples reducers
- No tiene vision global para optimizer el flujo

Mejoras propuestas

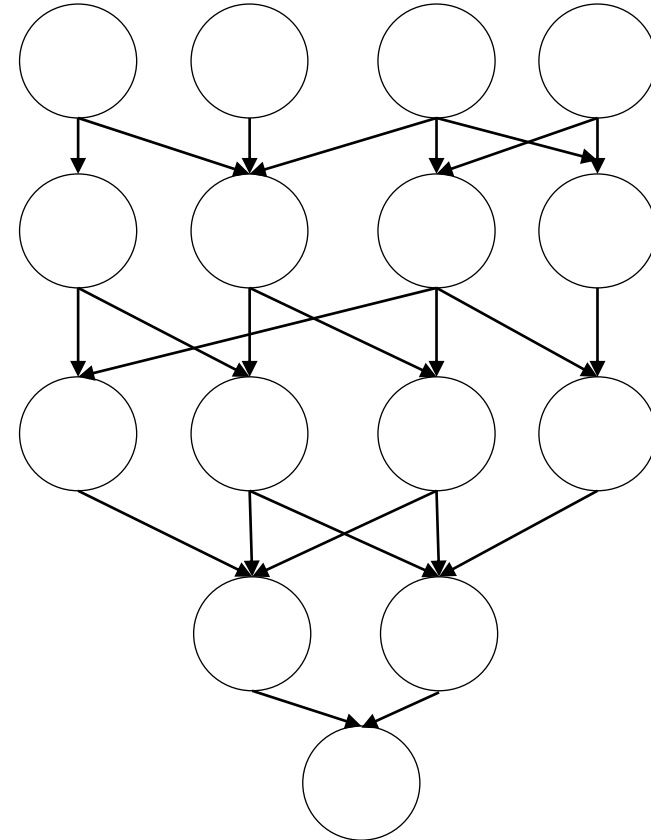
- Stinger/Tez
- Dryad
- Naiad:
 - *Cyclical differential dataflow model*
- Apache Spark



Coordinar Flujo de Datos para la Ejecución de trabajos

► Facilitar Flujos de Datos

- El principal motivo por el que Hadoop tuvo éxito en el ecosistema de procesamiento de datos es porque proporcionaba un framework universal para escribir y ejecutar trabajos distribuidos.
- Sin embargo, un único MapReduce (Hive, Pig, etc) raras veces es suficiente para la mayoría de casos de uso
- Se hace necesario coordinar una serie de trabajos MapReduce
- El principal esfuerzo de ingeniería está en la coordinación del flujo de datos para la ejecución del trabajo
- ¿Podemos hacerlo mejor que MapReduce, que recrea los lectores y escribe a disco para cada iteración en el flujo de datos?



Resultado....

Real-time

StreamInsight

Storm

Naiad

Batch

SCOPE

DryadLINQ

Vowpal Wabbit

Graph

GraphLab

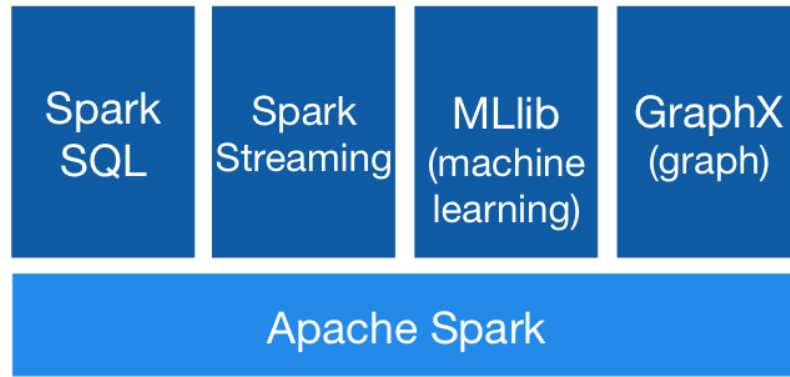
Giraph

El nacimiento de Apache SPARK

- Inspirado directamente por el modelo de flujo de datos cíclico de Naiad y Dryad
- Se inició como un Proyecto de investigación en el AMPLab en UC Berkeley en 2009
- Open Source License (Apache 2.0)
- Es el Proyecto Apache más activo (800+ developers)



Framework Unificado



- Unifica:

- Procesado Batch
- Procesado Tiempo Real
- Stream Analytics
- Machine Learning
- SQL Interactivo

Objetivo: motor unificado entre orígenes de datos, cargas de trabajo y entornos

APIs de desarrollo simplificadas: Scala, Java, Python, R

¿Cuántas APIs tenemos en SPARK?



API No estructurada

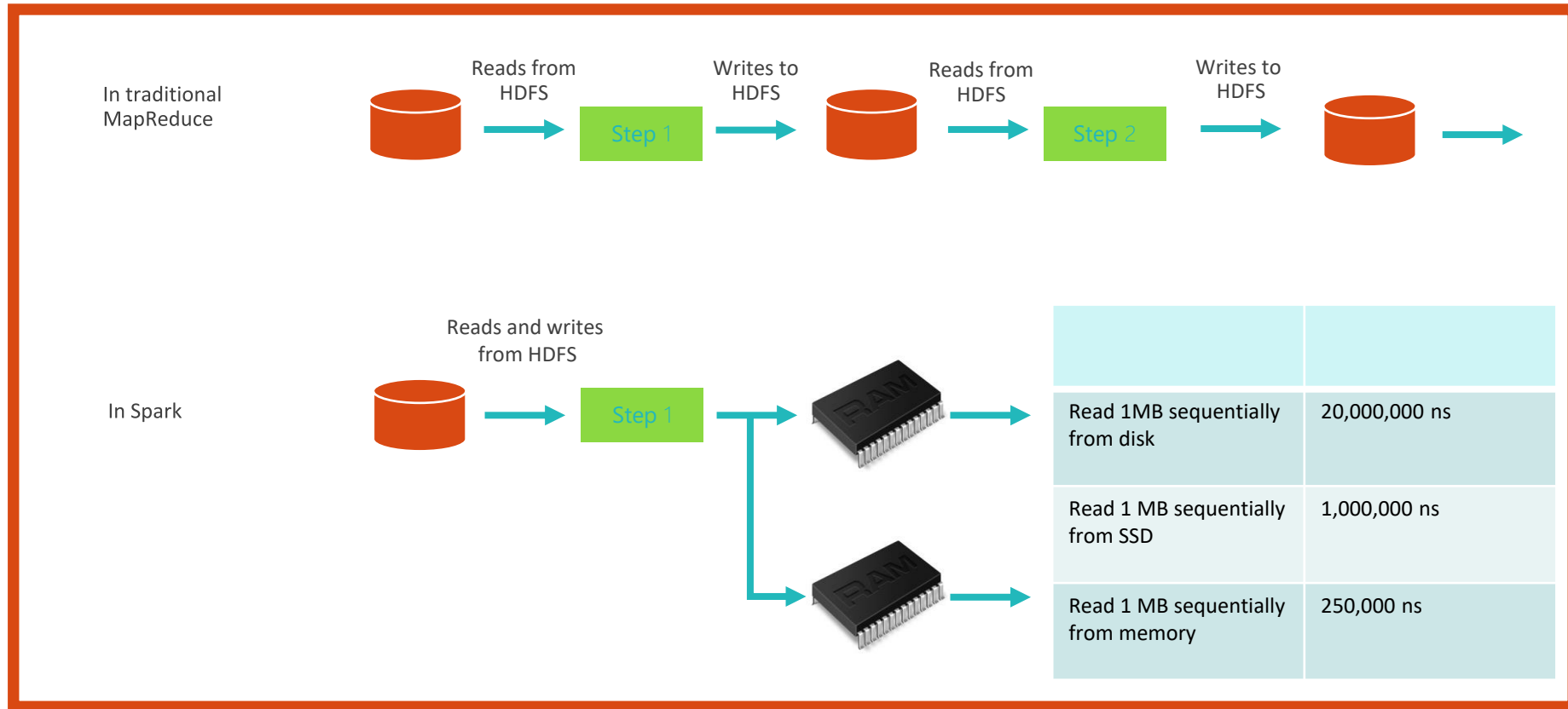
- Manipular objetos Java Raw
- RDDs
- Acumuladores
- Broadcast de Variables

API Estructurada

- Optimizado para tablas
- DataFrames
- DataSets
- Spark SQL

Spark vs. Map Reduce

Datos en memoria compartidos entre los pasos del trabajo



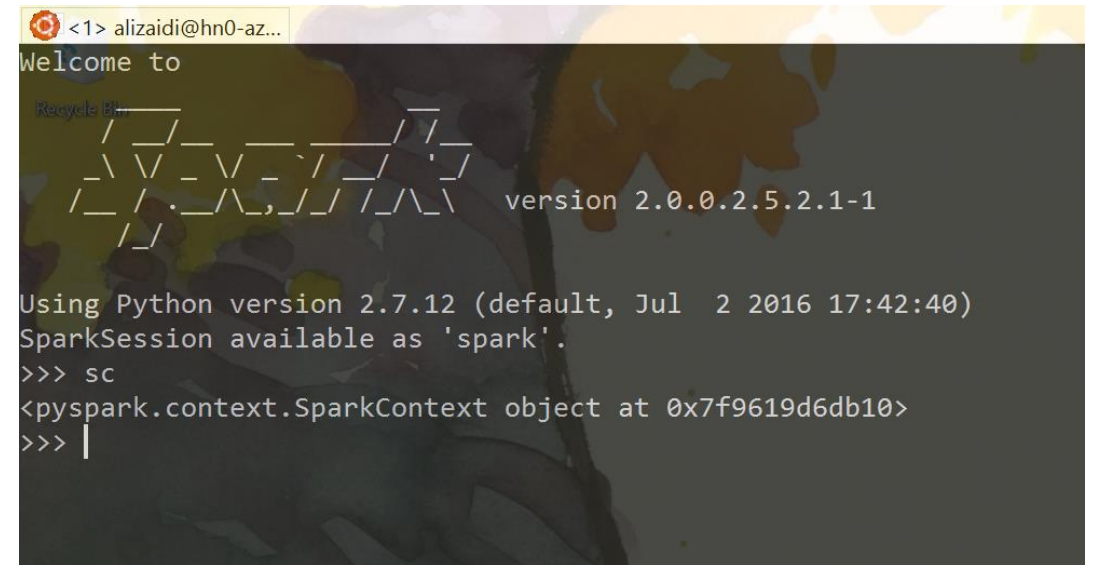
Proceso de Datos con DAG

- Al contrario que MapReduce el motor de procesamiento de datos de Spark no está limitado a mappers y reducers
- Un trabajo Spark puede tener múltiples estados, cada uno de ellos compuesto por varias tareas (“mappers” y “reducers”)
- Los datos se meten en caché y se reutilizan entre iteraciones, reduciendo I/O
- Desarrollado como un motor de procesamiento general para varias cargas de trabajo

Spark-Shell y Notebooks

Spark-Shell

- Disponible en Scala, Python y R



```
<1> alizaidi@hn0-az...  
Welcome to  
Spark version 2.0.0.2.5.2.1-1  
Using Python version 2.7.12 (default, Jul 2 2016 17:42:40)  
SparkSession available as 'spark'.  
>>> sc  
<pyspark.context.SparkContext object at 0x7f9619d6db10>  
>>> |
```

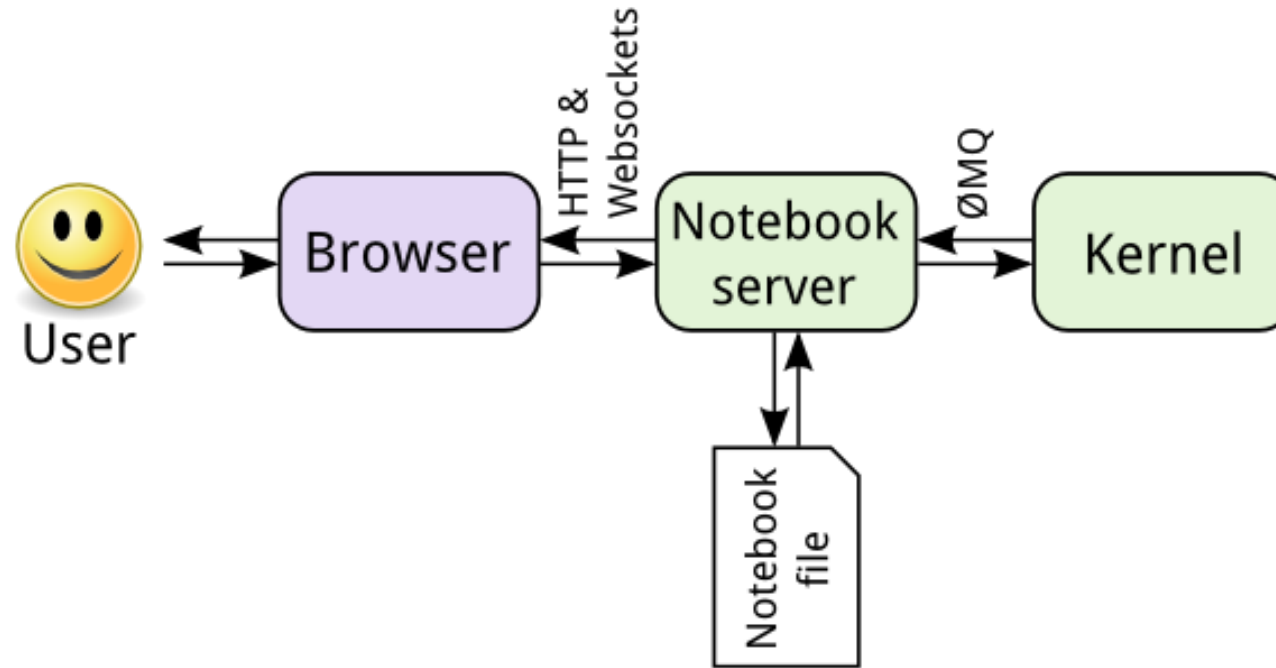
```
val file = sc.textFile("/example/data/gutenberg/davinci.txt")  
  
val counts = file.flatMap(line => line.split(" ")).map(word => (word,  
1)).reduceByKey(_ + _)  
  
counts.collect()
```

Notebooks

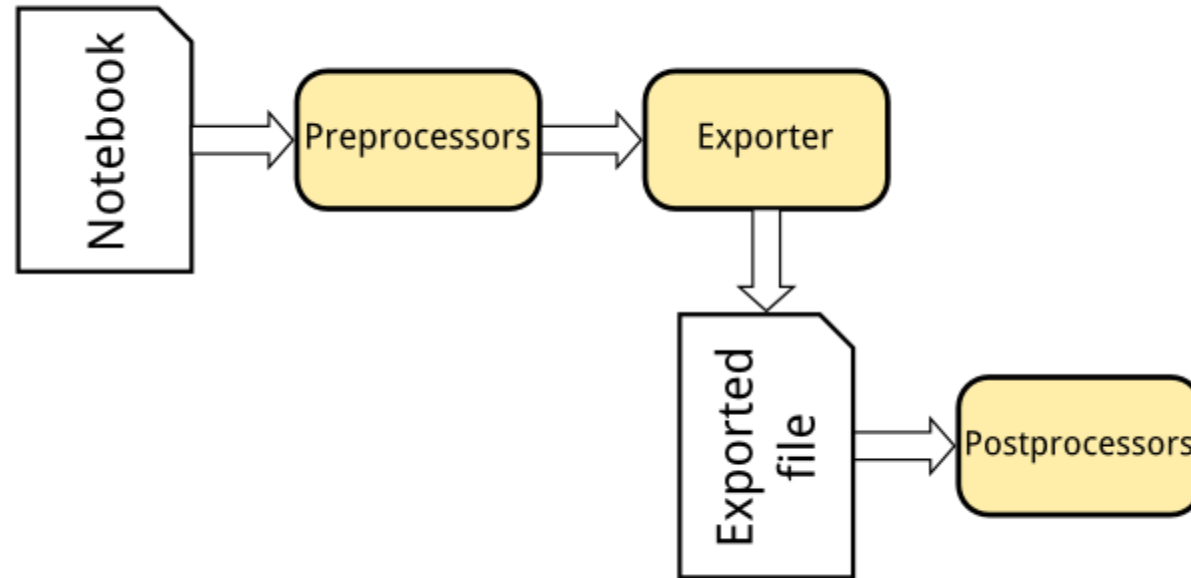


- REPL – Read-Evaluate-Print-Loop
- Prototipo, desarrollo rápido, exploración,...
- Colaboración en equipos

Arquitectura



Exportación



Demo

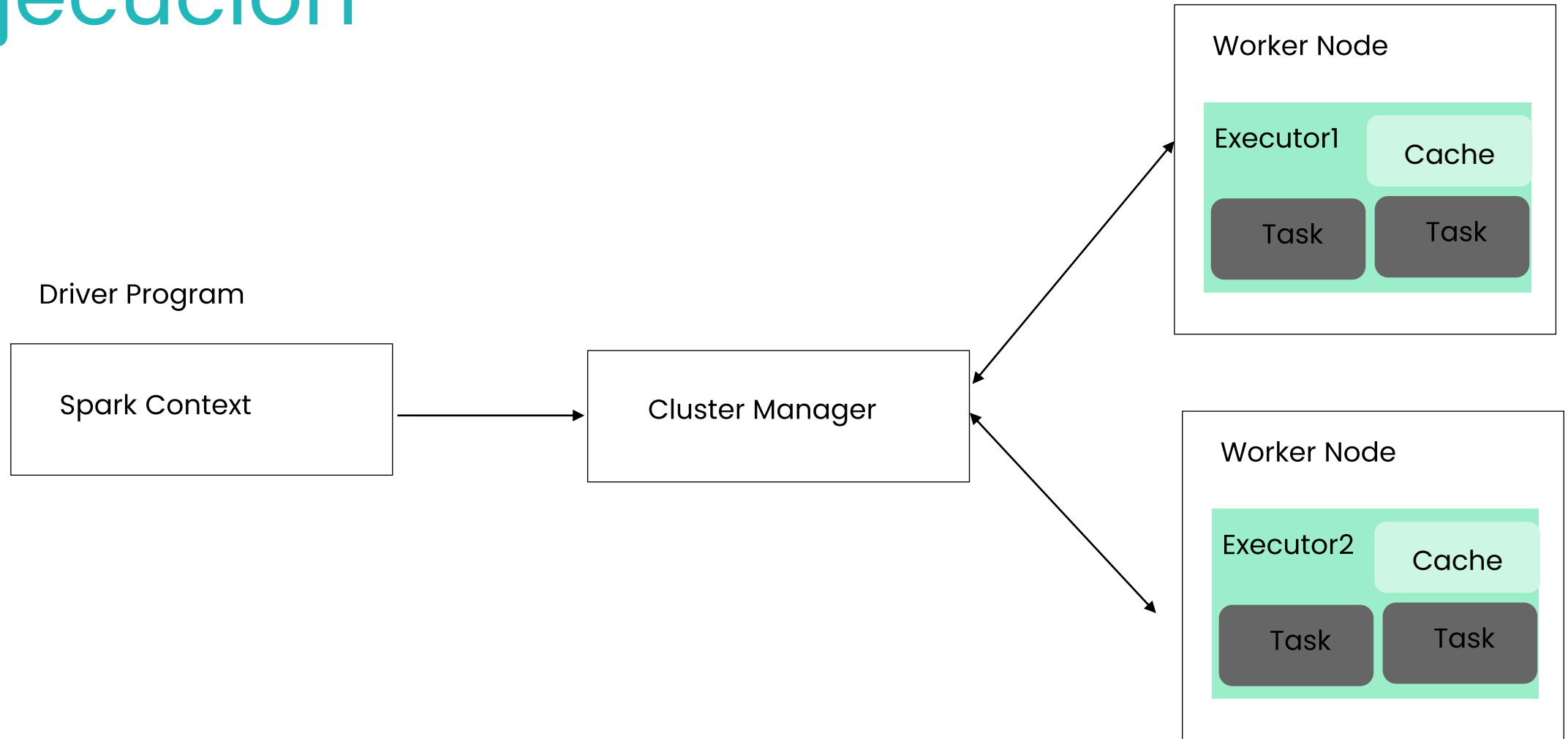
- Spark Shell
- Interactuando con Notebooks

Arquitectura Spark

Tipos de Cluster SPARK

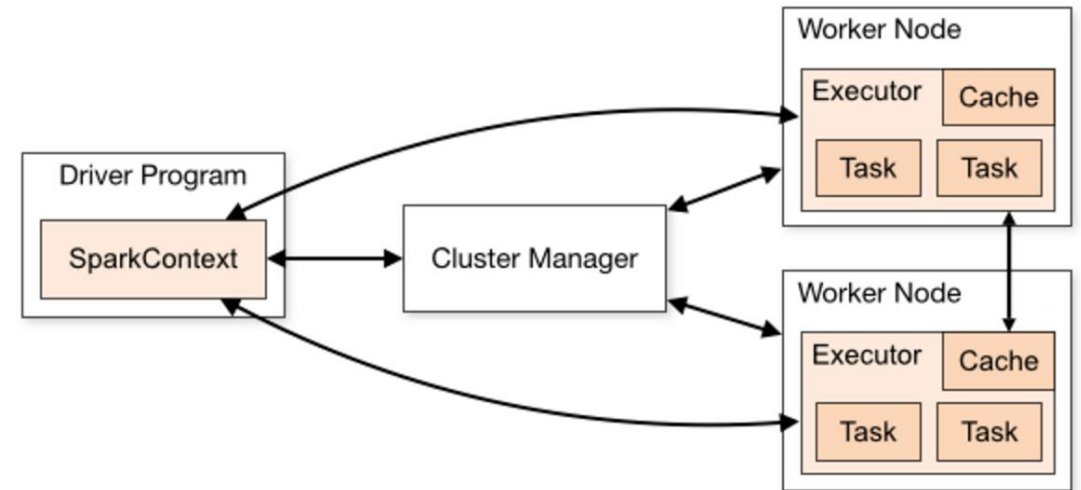
- Podemos utilizar tres tipos de administradores de cluster
 - Spark Standalone
 - Mesos
 - YARN
- El cluster manager provee de recursos
- En el caso de YARN, el gestor de recursos, asignará los contenedores YARN
- Los *executors* ejecutan las operaciones y almacenan los datos
- El código de aplicación se pasa a los *executors*
- El *driver* envía las tareas a los *executors*

Arquitectura SPARK en tiempo de ejecución



Aplicaciones SPARK

- Pueden contener una gran variedad de cargas de trabajo y parámetros
- Pero siempre tienen la siguiente configuración:
 - Un proceso *driver*
 - Mantiene la información sobre la aplicación
 - Responde al programa de usuario
 - Analiza, distribuye y planifica el trabajo entre los *executors*
 - Un conjunto de procesos *executors*
 - Ejecutan el código que les asigna el *driver*
 - Reportan el estado al *driver*

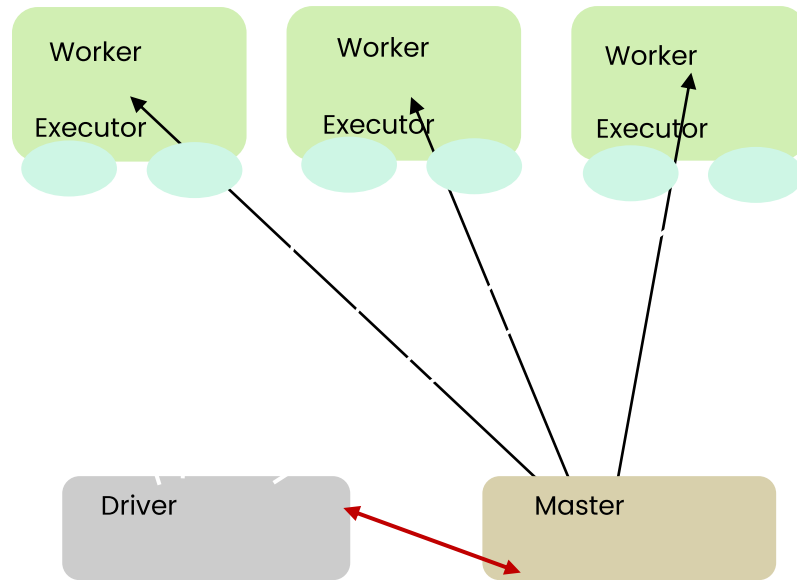


Nota:

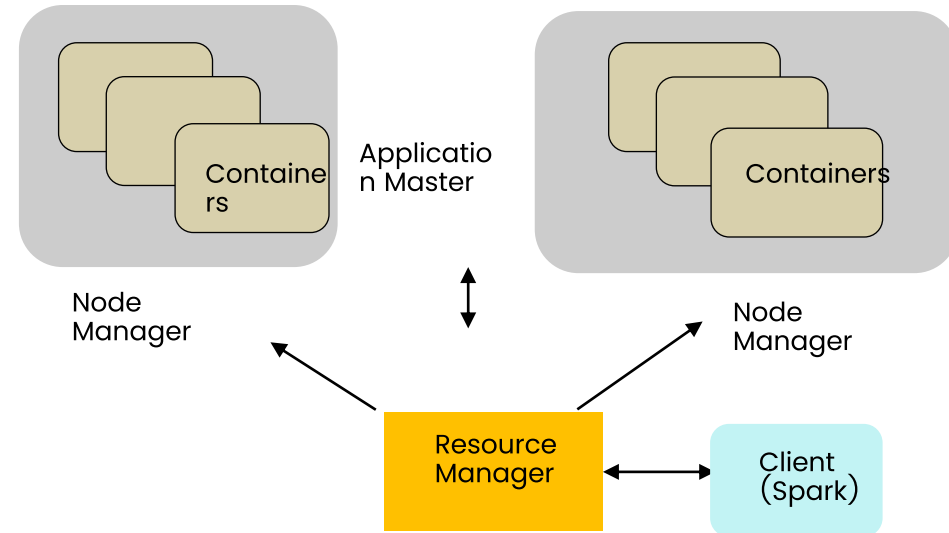
En YARN, tenemos dos modos de configuración:

1. *Cluster-mode*: el driver se ejecuta en el *application master* en uno de los nodos
2. *Client-mode*: El driver se ejecuta en el cliente, necesitamos el AM solo para peticiones YARN

Modos de Despliegue



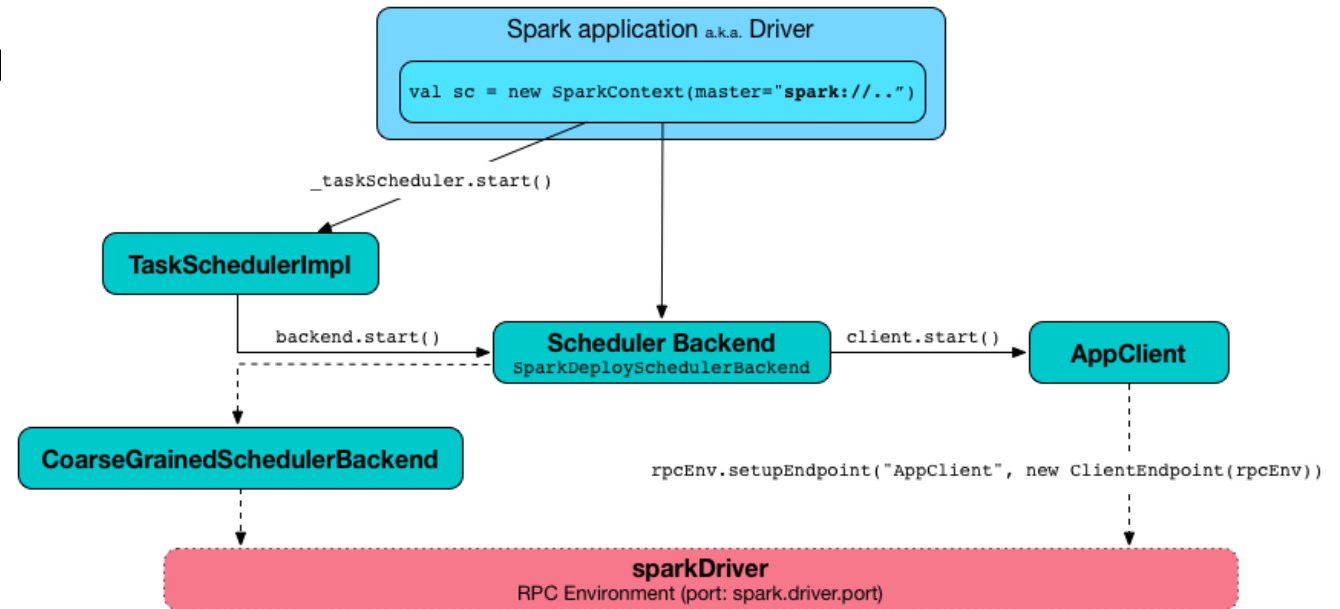
Spark Standalone



Spark en YARN

Standalone

- En este modo Spark usa un *daemon* Master para coordinar con los *workers* que ejecutan los *executors*
- Es el modo predeterminado
- Al lanzar la aplicación podemos decidir cuanta memoria utilizarán los *executors*, así como el número total de cores



Modo YARN

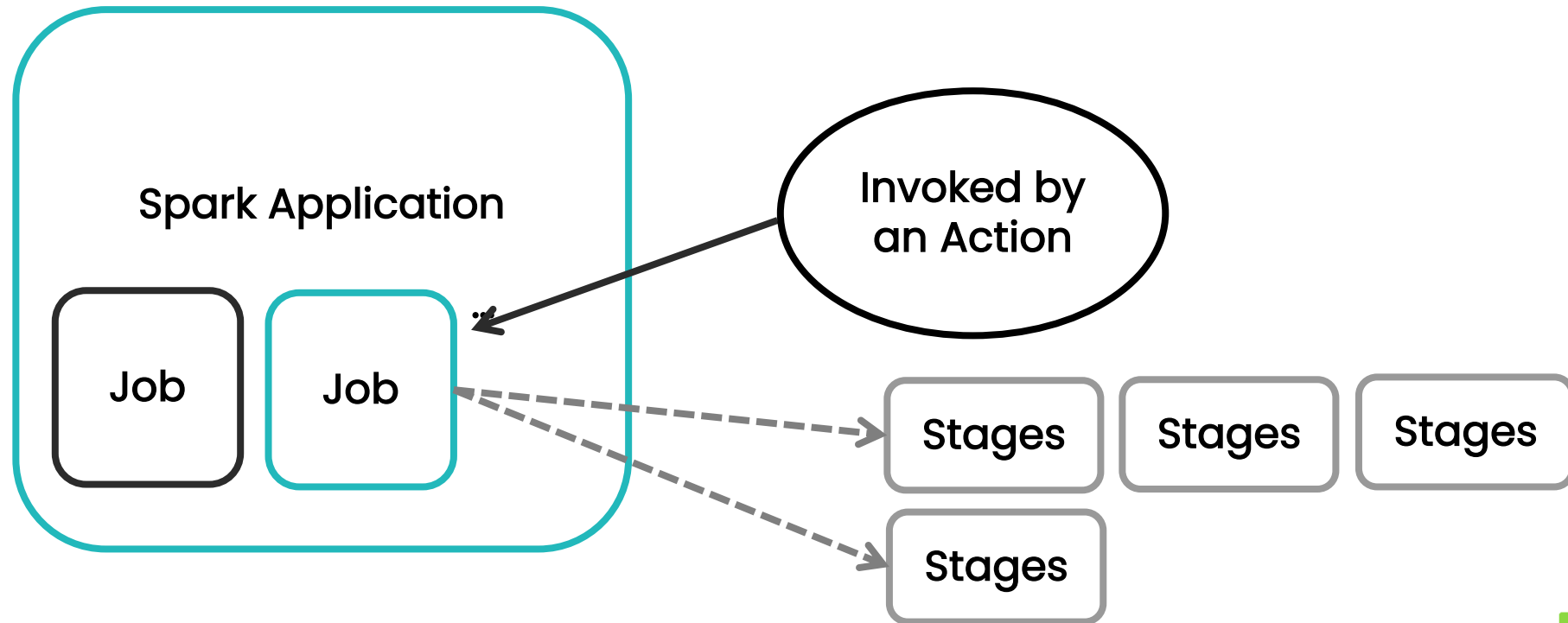
- El ResourceManager de YARN juega el rol de Spark Master
- Los NodeManagers de YARN actúan como *executors*
- Más complejo pero más seguro
- Podemos ejecutarlo en dos modos
 - Cliente – interactivo
 - Cluster – background

Terminología

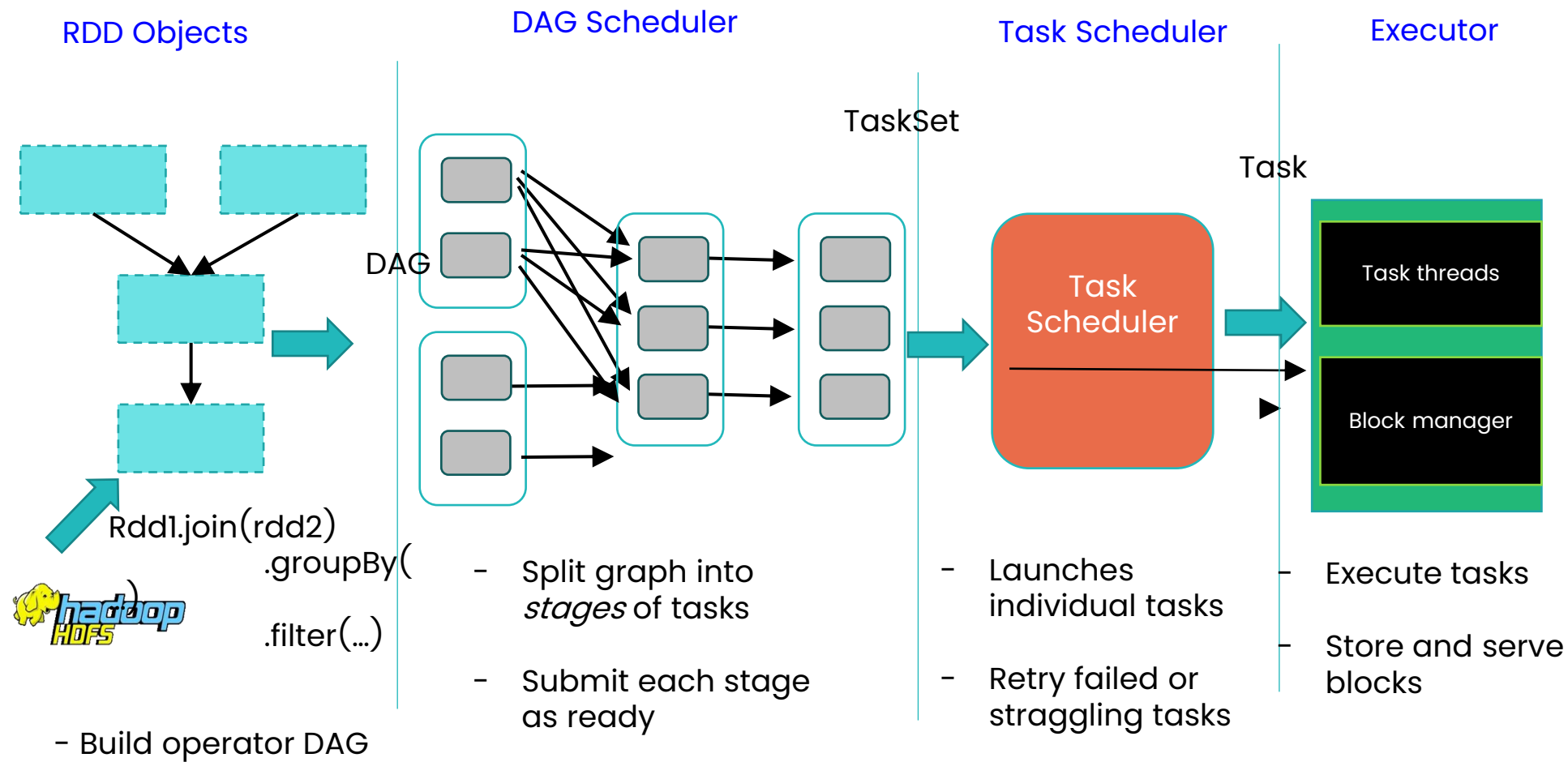
- Job: El trabajo necesario para operar con un RDD
- Stage: Una ronda de trabajo dentro de un *job*, correspondiente a uno o más RDD's en el *pipeline*.
- Tasks: Una unidad de trabajo dentro de un *stage*, correspondiente con una partición RDD
- Shuffle: La transferencia de datos entre *stages*

Ejecución física de una aplicación

Application → Jobs → Stages → Tasks



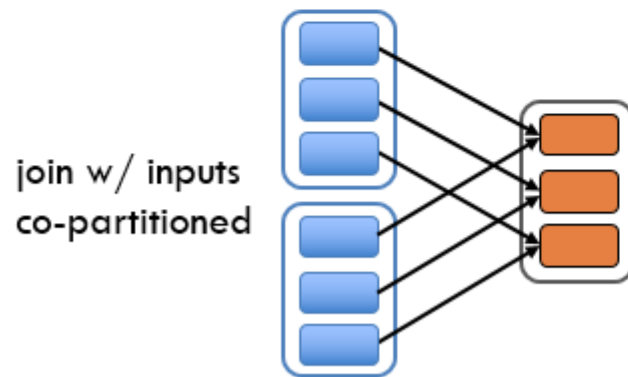
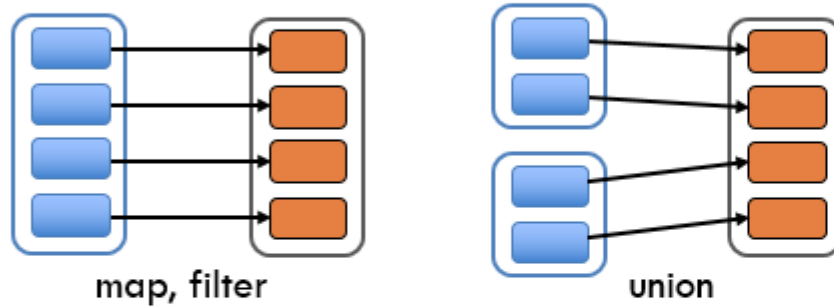
Proceso de Schedule



Dependencias *Narrow* vs. *Wide*

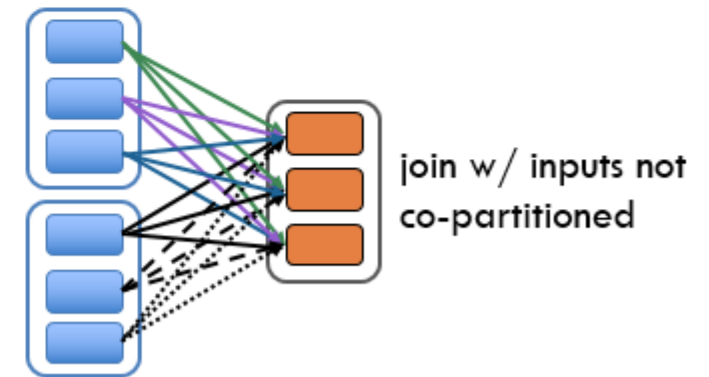
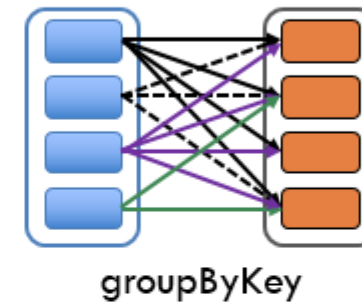
narrow

each partition of the parent RDD is used by at most one partition of the child RDD



wide

multiple child RDD partitions may depend on a single parent RDD partition



SPARK RDD

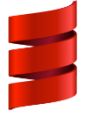
RDD(Resilient Distributed Dataset)

- Son la abstracción principal en Spark
- Las aplicaciones en Spark se escriben en términos de operaciones en conjuntos de datos distribuidos
- El objeto de datos genérico y principal en Spark son los RDD
- Los RDD se construyen manipulando conjuntos de datos distribuidos (objetos HDFS, otros RDDs) a través de una gran variedad de transformaciones paralelas (map, filter, join)
- Las transformaciones se evalúan de forma *perezosa*
- Los RDD hacen un seguimiento del linaje, de modo que pueden reconstruirse en caso de fallo

RDD(Resilient Distributed Dataset)

- Distribuido entre los “workers” de Spark
- Inmutables
 - Si se transforma se genera una nuevo
- Evaluación “perezosa”
 - No se ejecuta en tiempo de definición, sino cuando se evalúe aplicando una acción sobre el RDD
- Para crear un RDD
 - Paralelizando.
 - `val newRDD= sc.parallelize(List(1, 2, 3, 4))`
 - A partir de una fuente de almacenamiento
 - `val newRDD: RDD[Int] = sc.textFile("myValues.txt")`
 - Transformando un RDD
 - `val newRDD: RDD[String] = intValues.map(_.toString)`

Paralelizar una colección



```
val animalsRDD = sc.parallelize(List("panda", "jaguar", "sloth"))
```



```
animalsRDD = sc.parallelize(["panda", "jaguar", "sloth"])
```



```
animalsRDD <- SparkR:::parallelize(sc, list("panda", "jaguar", "sloth"))  
# RDD operations no longer exported in SparkR > 2.0 namespace
```


Crear un RDD desde un Fichero de Texto



```
val fruitsRDD = sc.textFile("/example/data/fruits.txt"))
```

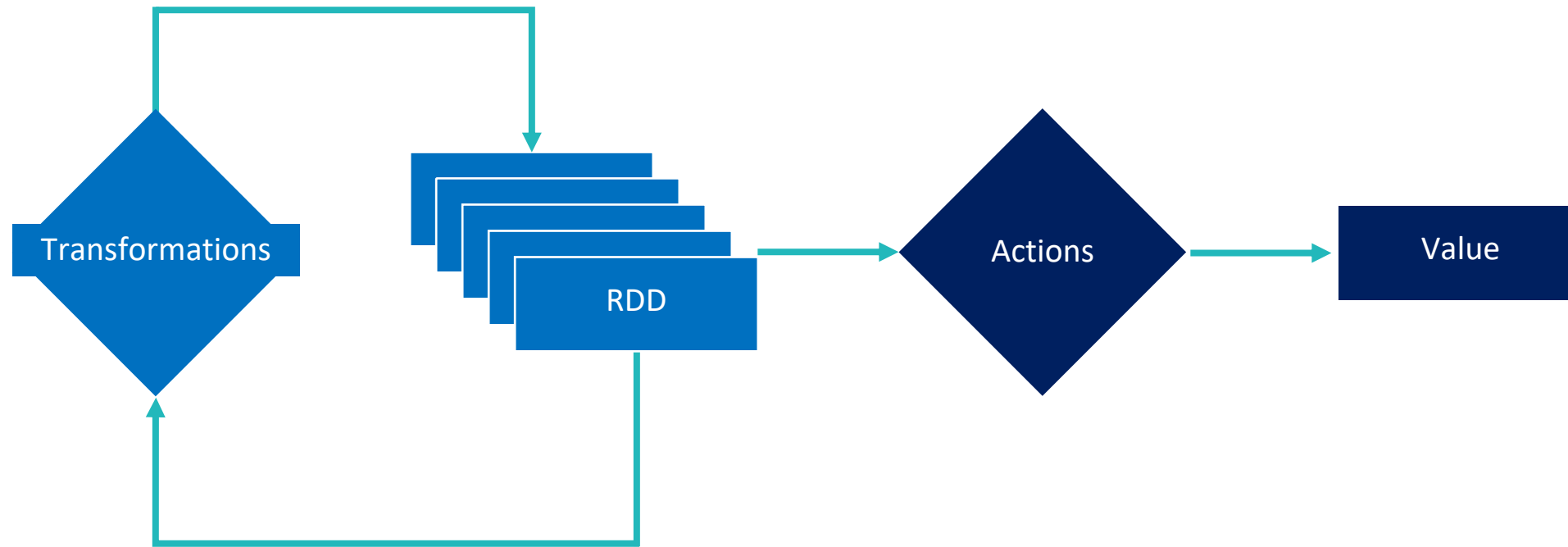


```
fruitsRDD = sc.textFile("/example/data/fruits.txt")
```



```
fruitsRDD <- SparkR:::textFile(sc, "/example/data/fruits.txt")  
# RDD operations no longer exported in SparkR > 2.0 namespace
```

Transformaciones y Acciones



Transformaciones sobre RDD

- map: aplica una función a cada elemento de la colección:
 - `intValues.map(_.toString) // RDD[String]`
- filter: selecciona el subconjunto de elementos que cumplen una determinada expresión booleana:
 - `intValues.filter(_.isOdd) // RDD[Int]`
- flatMap: además de realizar una función map, aplica un método flatten:
 - `textFile.map(_.split(" ")) // RDD[Array[String]]`
 - `textFile.flatMap(_.split(" ")) // RDD[String]`

Acciones sobre RDD

- `count`: nos devuelve el número total de elementos:
 - `sc.parallelize(List(1, 2, 3, 4)).count // 4`
- `collect`: nos vuelca toda la colección distribuida en un array en memoria:
 - `sc.parallelize(List(1, 2, 3, 4)).collect // Array(1, 2, 3, 4)`
 - Ojo, cuidado. Si el RDD es muy grande, podemos tener problemas al volcar toda la colección en memoria.
- `saveAsTextFile`: nos vuelca la información en un fichero de texto:
 - `intValues.saveAsTextFile("results.txt")`
- `Coalesce`

Spark SQL

Spark SQL

- Parte del core de Spark desde la versión 1.0
- Ejecuta consultas SQL / HiveQL, junto con despliegues existentes de HIVE, o los reemplaza



```
SELECT COUNT(*)  
FROM hiveTable  
WHERE hive_udf(data)
```

Spark SQL

- A través del Spark `SQLContext`, solo se soporta el dialecto "sql," un subconjunto de SQL 92.
- A través de `HiveContext`, el dialecto por defecto es "hiveql", correspondiente con el dialecto SQL de Hive. "sql" también está disponible

Pero, dónde están las tablas?

- Ejecutamos consultas SQL a través de los `SQLContext` o `HiveContext`, utilizando el método `sql()`
- El método `sql()` devuelve un `DataFrame`
- Podemos utilizar un motor DBI para utilizar un interfaz SQL diferente
- Podemos mezclar métodos `DataFrame` y consultas SQL en el mismo código
- **Para usar SQL, debemos de:**
 - Consultar una tabla Hive ya persistida o,
 - Generar un alias de table para un `DataFrame`, utilizando `registerTempTable()`

El API de Dataframes

- Es una extensión / abstracción al API de RDD
- Inspirado en los objetos Dataframes de R y Python
- Diseñado desde el principio para soportar cargas de trabajo Big Data
- Facilitar el trabajo con datos sin necesidad de bajar al detalle de los RDD

RDDs colección de elementos genéricos(estructuras internas desconocidas por SPARK).

User	User	User
User	User	User

Colección de objetos con esquema conocido para Spark SQL.

Name	Age	Sex
Name	Age	Sex
Name	Age	Sex
Name	Age	Sex

Características de los Dataframes

- Habilidad para escalar desde los Kb en portátiles a los Pb en un cluster
- Soporte de una gran cantidad de formatos de datos y opciones de almacenamiento
- Integración con el resto de componentes a través de infraestructura Spark de un modo sencillo
- APIs para Python, Java, Scala y R
- Optimización y generación de código a través de un optimizador específico Spark SQL Catalyst

Dataframes vs. RDDs

- Familiar
Para nuevos usuarios familiarizados con otros lenguajes de programación, el API de Dataframes es más familiar
- Facilidad
Para usuarios Spark, es más sencillo de programar que utilizando RDDs
- Optimizados
Obtenemos un mejor resultado debido a una mejor generación de Código y optimización

DataFrames

- Son la abstracción preferida en Spark
- Colección fuertemente tipada de elementos distribuidos
- Construido sobre RDD
 - Son inmutables
 - Llevan registro de las operaciones para reconstruir en caso de fallo
 - Posibilita operaciones en paralelo en colecciones de elementos
- Construimos DataFrames...
 - Paralelizando colecciones existentes
 - Transformando otro DataFrame
 - Obteniendo ficheros de sistemas de almacenamiento

Ejemplo de código SCALA

```
val df = sqlContext.  
  read.  
    format("json").  
    option("samplingRatio", "0.1").  
    load("/Users/spark/data/stuff.json")  
  
df.write.  
  format("parquet").  
  mode("append").  
  partitionBy("year").  
  saveAsTable("faster-stuff")
```

Crear DataFrames desde orígenes de datos

```
>>> df = sqlContext.jsonFile("somejsonfile.json")           //from JSON file*

>>> df = hiveContext.table("somehivetable")                 //from a  Hive Table

>>> df = sqlContext.parquetFile("someparquetsource")         //from a parquet file

>>> df = sqlContext.load(source="jdbc", url="UrlToConnect", dbtable="tablename")//from
JDBC source
```

Crear un DataFrame en Python

```
# The import isn't necessary in the SparkShell or Databricks
from pyspark import SparkContext, SparkConf

# The following three lines are not necessary
# in the pyspark shell
conf = SparkConf().setAppName(appName).setMaster(master)
sc = SparkContext(conf=conf)
sqlContext = SQLContext(sc)

df = sqlContext.read.parquet("/path/to/data.parquet")
df2 = sqlContext.read.json("/path/to/data.json")
```

Operaciones de DataFrames

```
>>> val df = sqlContext.jsonFile("somejsonfile.json")

>>> df.show()                                // Show the contents of the DataFrame

>>> df.printSchema()                        // Print the schema in a tree format

>>> df.select("name").show()                // Select and show the name columns

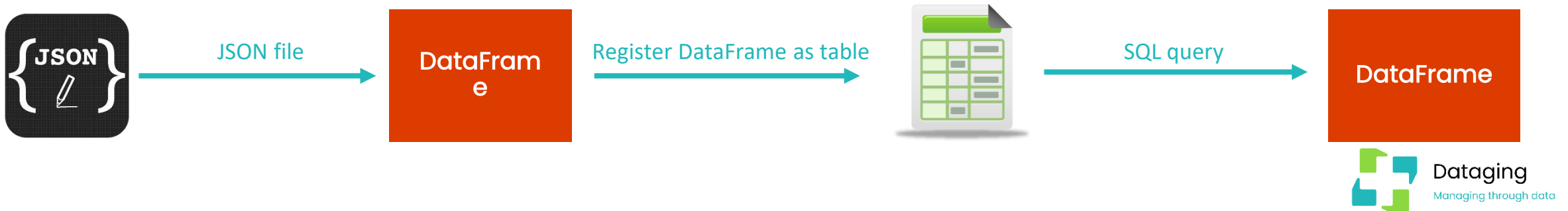
>>> df.select(df("name"),df ("age") +1).show() // Select all but increment the age by 1

>>> df.filter(df("age") > 21).show()        // Select people older than 21

>>> df.groupBy("age").count().show()        // Count people by age
```


Tablas y Consultas

```
// First create a DataFrame from JSON file.  
>>> df = sqlContext.jsonFile("Users.json")  
  
// Register the DataFrame as a temporary table. Temp tables exist only during lifetime of this SQLContext  
instance.  
>>> usertable = sqlContext.registerDataFrameAsTable(df, "UserTable")  
  
// Alternatively, execute a SQL query on the table. The query returns a DataFrame.  
>>> teenagers = sqlContext.sql("select Age as Years from UserTable where age > 13 and age <= 19")
```



Tablas Hive

```
// crear un HiveContext, que se deriva de SQLContext.

>>> sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)
>>> sqlContext.sql("CREATE TABLE IF NOT EXISTS UserTable (key INT, value STRING)")
>>> sqlContext.sql("LOAD DATA LOCAL INPATH 'user.txt' INTO TABLE UserTable")
// Consultas expresadas en HiveQL.

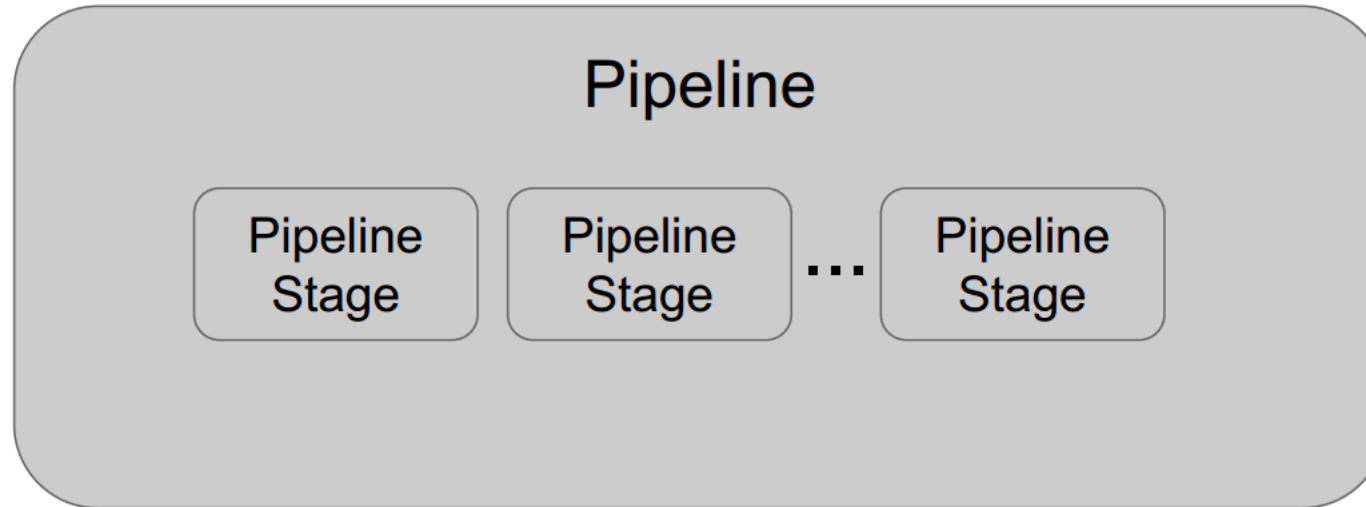
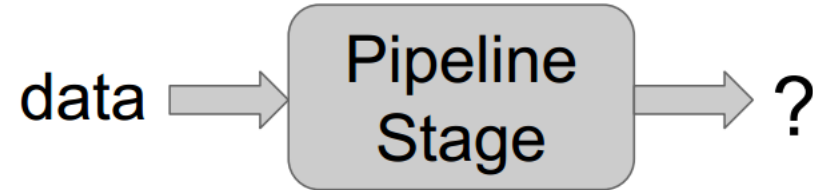
>>> val df = sqlContext.sql("FROM UserTable SELECT Name, Age")
```

Spark Machine Learning

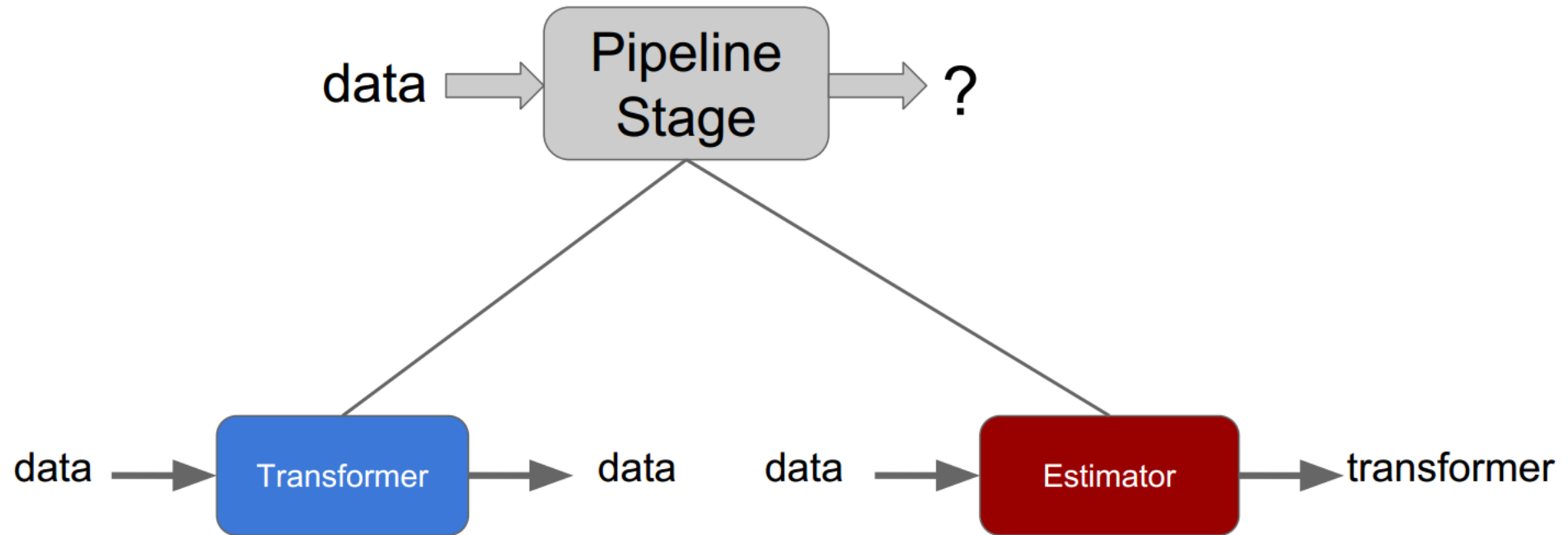
Spark MLlib vs SparkML

- Spark MLlib(spark.mllib)
 - Trabaja con el api RDD
 - En modo mantenimiento desde Spark 2.0
- SparkML (spark.ml)
 - Trabaja con el API DataFrames
 - Más optimizada, mayor número de operaciones y pipelines

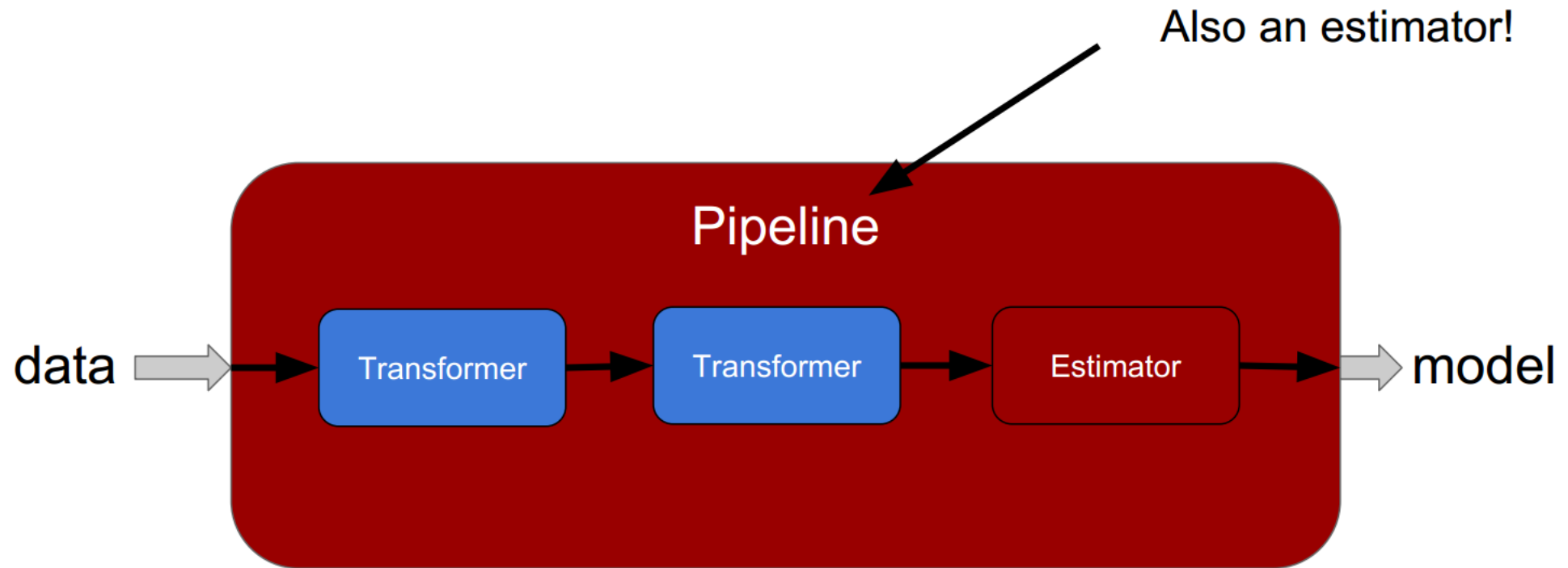
SparkML Pipelines



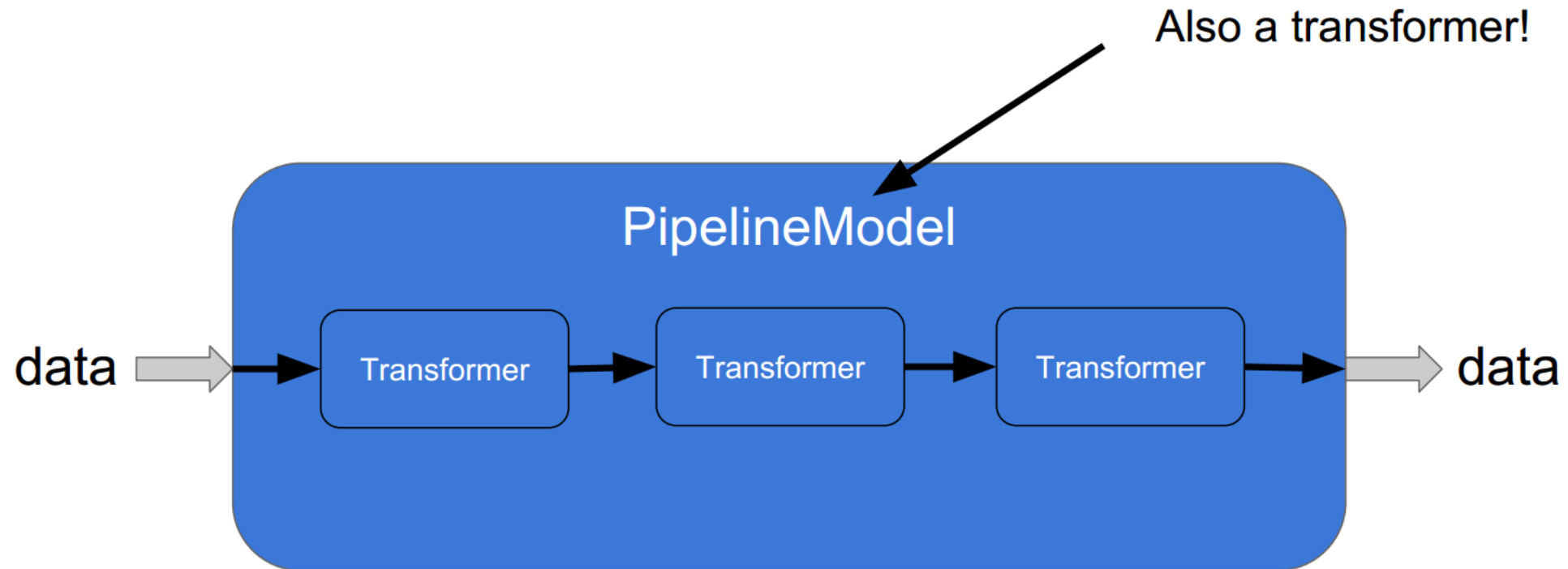
Dos tipos de Etapas



Pipelines son estimadores



Pipelines son también transformadores



ML: Transformador(featurizer)

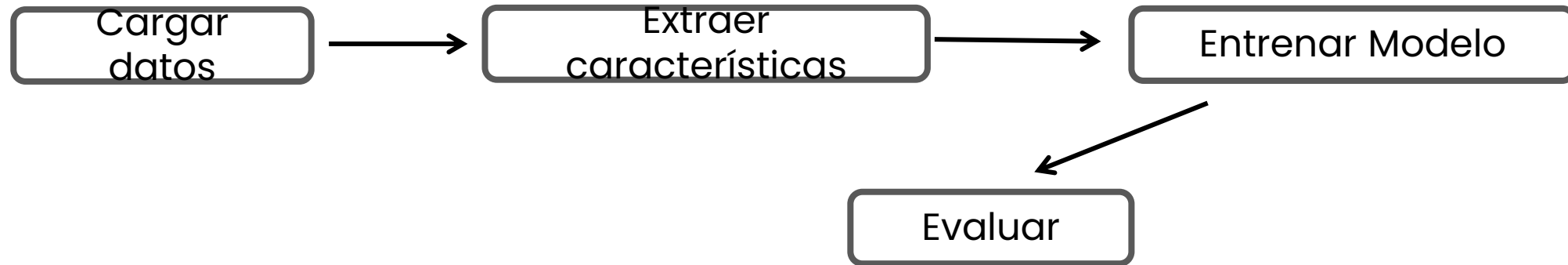
- Un *Transformador* es una clase que transforma un DataFrame en otro DataFrame
- Implementa el método *transform()*
- Ejemplos:
 - HasingTF
 - Bucketizer
 - Binarizer
 - PipelineModel

ML: Estimador

- Un *Estimador* es una clase que a partir de un *DataFrame* genera un *Transformador*
- Implementan el método *fit()*
- Ejemplos
 - IDF
 - CrossValidator
 - MultilayerPerceptronClassifier

ML: Pipelines

- Un *Pipeline* es un estimador que contiene etapas representando un flujo de trabajo reusable
- Las etapas del pipeline son una mezcla de estimadores y transformadores



Ejemplo en Scala: Entrenar un Transformador

- Transformador de características

```
val assembler = new VectorAssembler()  
  .setInputCols(Array("gre", "gpa", "prestige"))  
val df2 = assembler.transform(df)
```

admit	gre	gpa	prestige
no	380.0	3.61	3.0
yes	660.0	3.67	3.0
yes	800.0	4.0	1.0
yes	640.0	3.19	4.0
no	520.0	2.93	4.0



VectorAssembler



admit	gre	gpa	prestige	features
no	380.0	3.61	3.0	[380.0, 3.61, 3.0]
yes	660.0	3.67	3.0	[660.0, 3.67, 3.0]
yes	800.0	4.0	1.0	[800.0, 4.0, 1.0]
yes	640.0	3.19	4.0	[640.0, 3.19, 4.0]
no	520.0	2.93	4.0	[520.0, 2.93, 4.0]

Ejemplo en Scala: Entrenar un Transformador

```
val si = new StringIndexer().setInputCol("admit").setOutputCol("label")
val siModel = si.fit(df2)
val df3 = siModel.transform(df2)
```

admit	gre	gpa	prestige	features
no	380.0	3.61	3.0	[380.0, 3.61, 3.0]
yes	660.0	3.67	3.0	[660.0, 3.67, 3.0]
yes	800.0	4.0	1.0	[800.0, 4.0, 1.0]
yes	640.0	3.19	4.0	[640.0, 3.19, 4.0]
no	520.0	2.93	4.0	[520.0, 2.93, 4.0]

StringIndexer

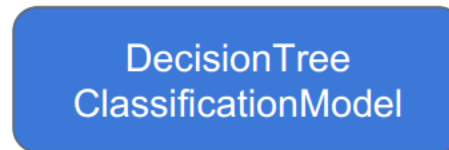
StringIndexerModel

admit	gre	gpa	prestige	features	label
no	380.0	3.61	3.0	[380.0, 3.61, 3.0]	0.0
yes	660.0	3.67	3.0	[660.0, 3.67, 3.0]	1.0
yes	800.0	4.0	1.0	[800.0, 4.0, 1.0]	1.0
yes	640.0	3.19	4.0	[640.0, 3.19, 4.0]	1.0
no	520.0	2.93	4.0	[520.0, 2.93, 4.0]	0.0

Ejemplo en Scala: Entrenar un Transformador

```
val dt = new DecisionTreeClassifier()  
val dtModel = dt.fit(df3)  
val df4 = dtModel.transform(df3)
```

features	label
[380.0, 3.61, 3.0]	0.0
[660.0, 3.67, 3.0]	1.0
[800.0, 4.0, 1.0]	1.0
[640.0, 3.19, 4.0]	1.0
[520.0, 2.93, 4.0]	0.0



features	label	prediction
[380.0, 3.61, 3.0]	0.0	0.0
[660.0, 3.67, 3.0]	1.0	0.0
[800.0, 4.0, 1.0]	1.0	1.0
[640.0, 3.19, 4.0]	1.0	1.0
[520.0, 2.93, 4.0]	0.0	0.0

Construyendo el Pipeline....

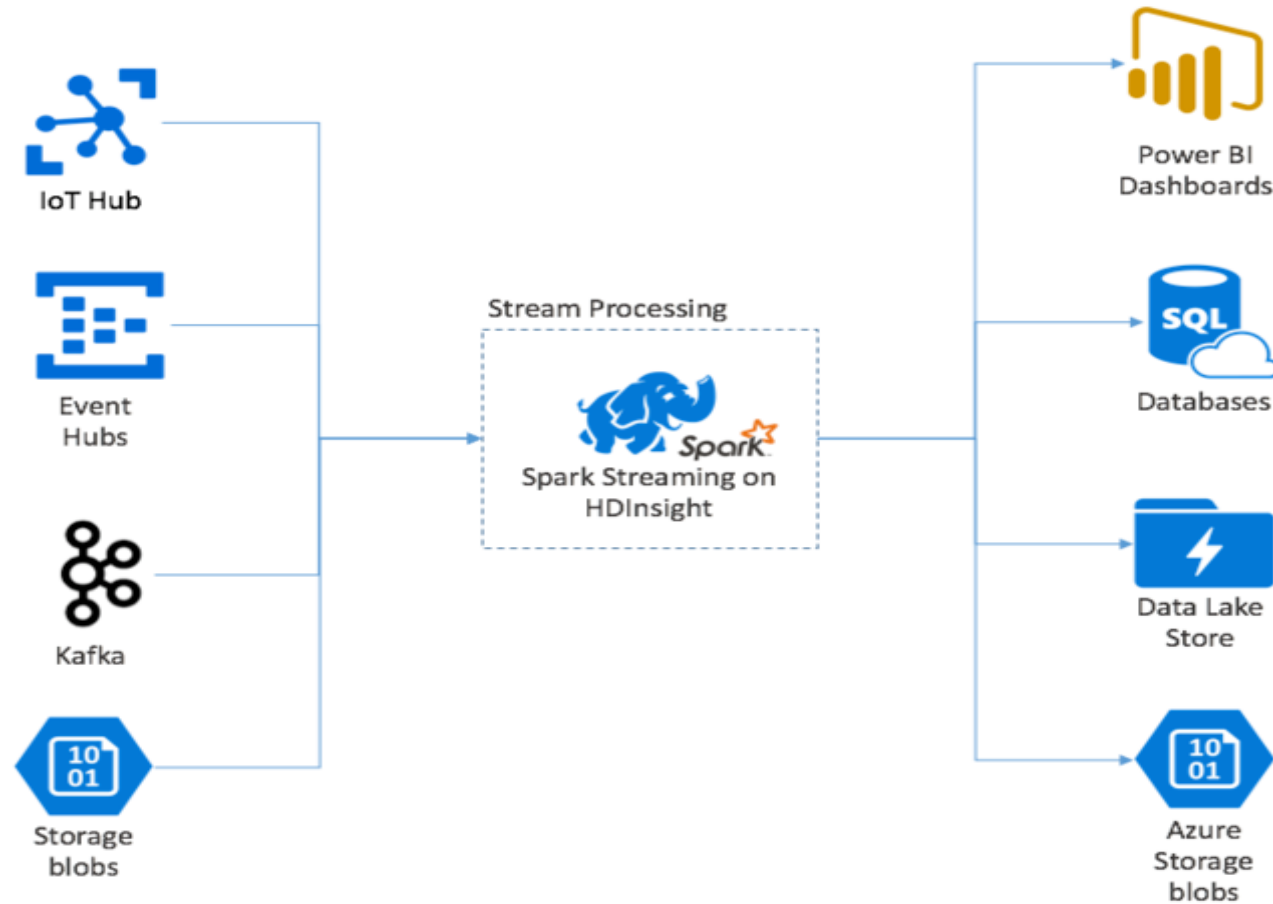
```
val assembler = new VectorAssembler()
assembler.setInputCols(Array("gre", "gpa", "prestige"))
val sb = new StringIndexer()
sb.setInputCol("admit").setOutputCol("label")
val dt = new DecisionTreeClassifier()
val pipeline = new Pipeline()
pipeline.setStages(Array(assembler, sb, dt))
val pipelineModel = pipeline.fit(df)
```

Spark Streaming

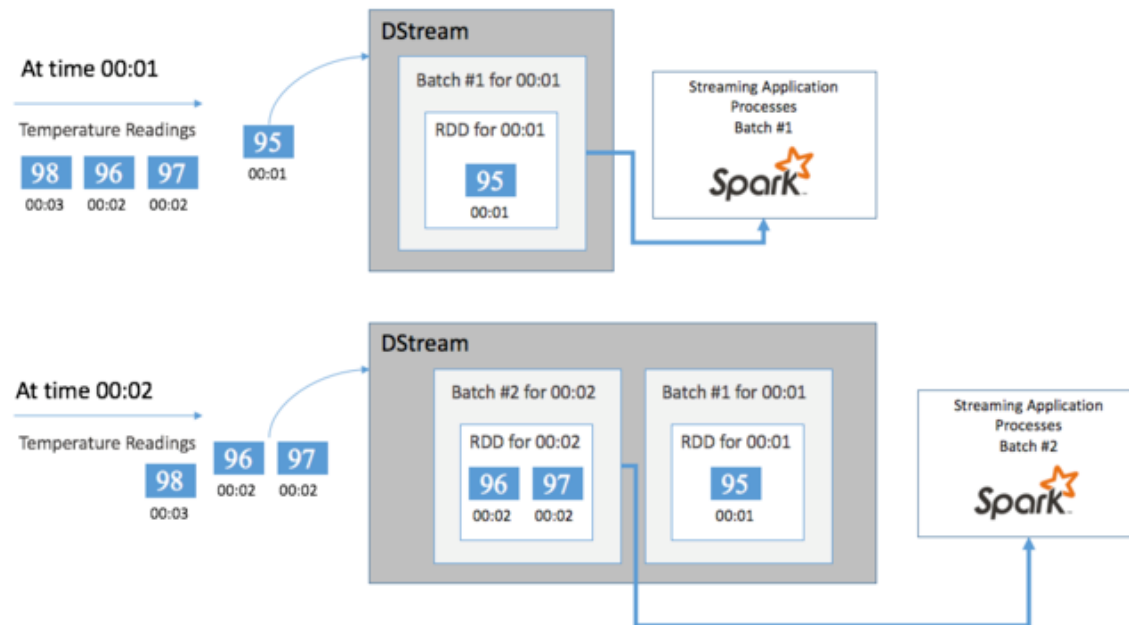
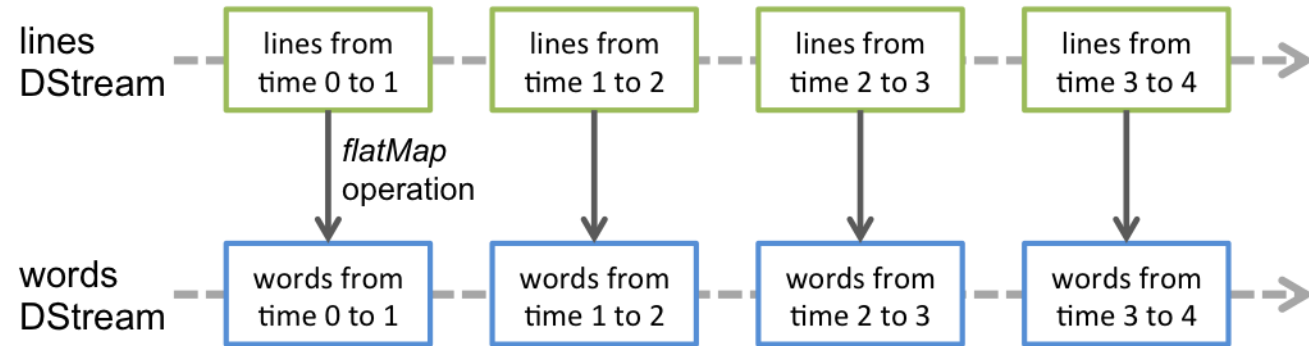
Apache Spark Streaming



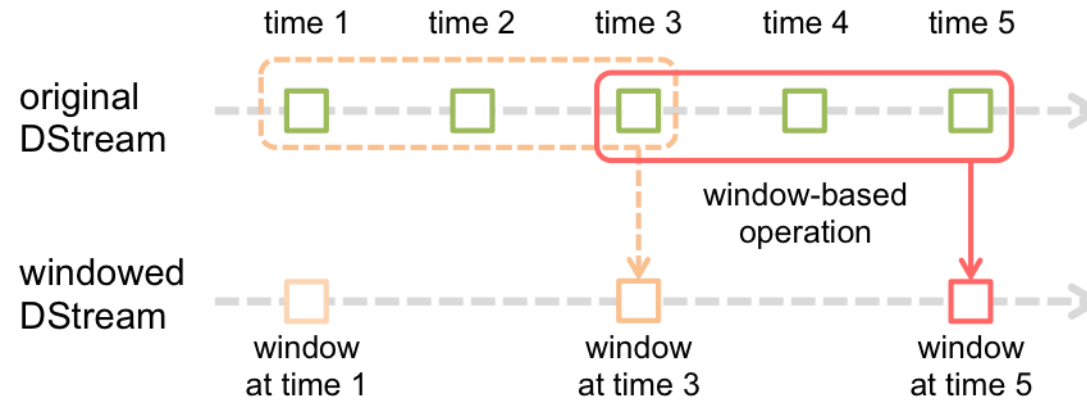
Spark Streaming en HDInsight



Dstreams (Discretized Streams)



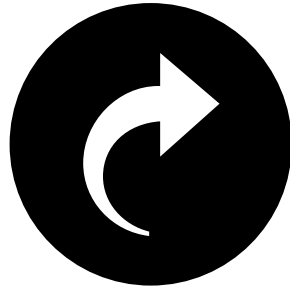
Window



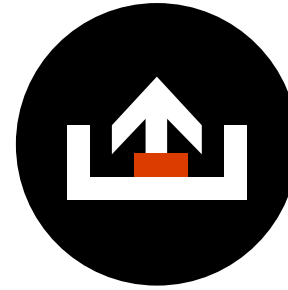
- *window length* – La duración de la Ventana (3 en la figura)
- *sliding interval* – El interval en el que la operación window se realiza (2 en la figura)

Operaciones DStream

- Dos tipos de operaciones: *transformaciones* y *salidas*

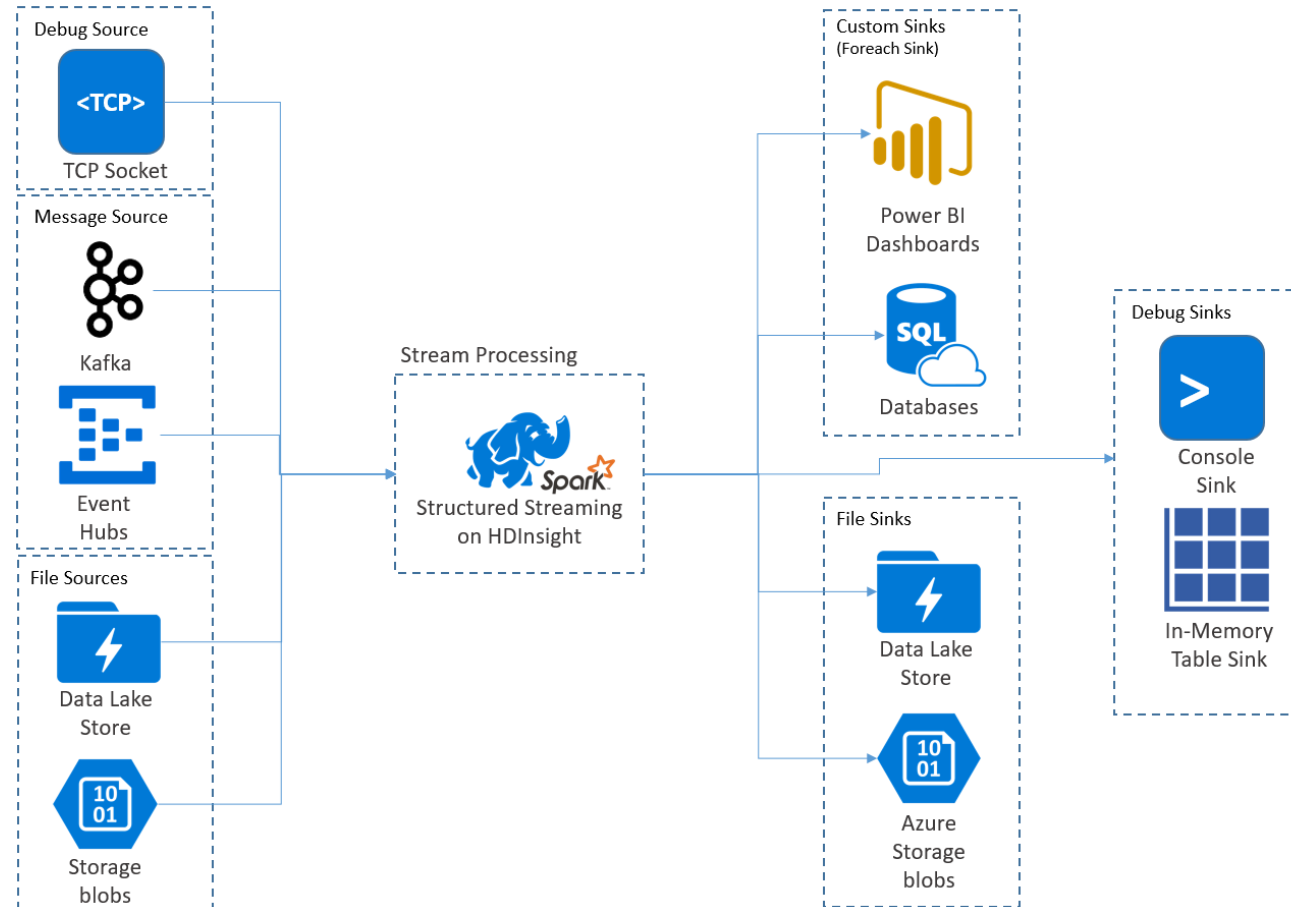


Las transformaciones DStream operan en uno o más Dstreams para crear nuevos con datos transformados (parecido a los RDD)



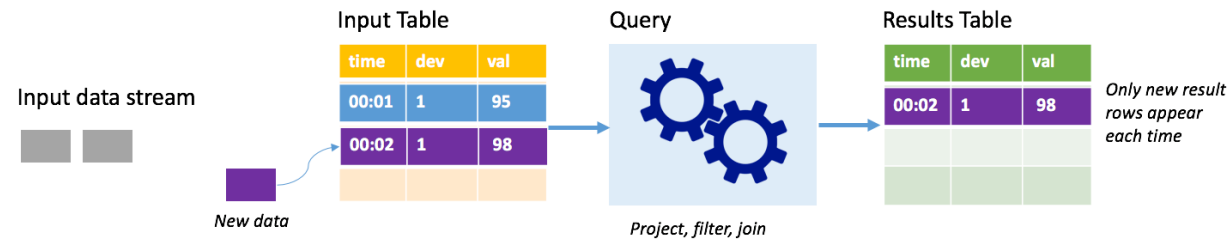
La Operaciones Output escriben datos a una salida de datos externa, como un Sistema de ficheros o una base de datos

Structured Streaming

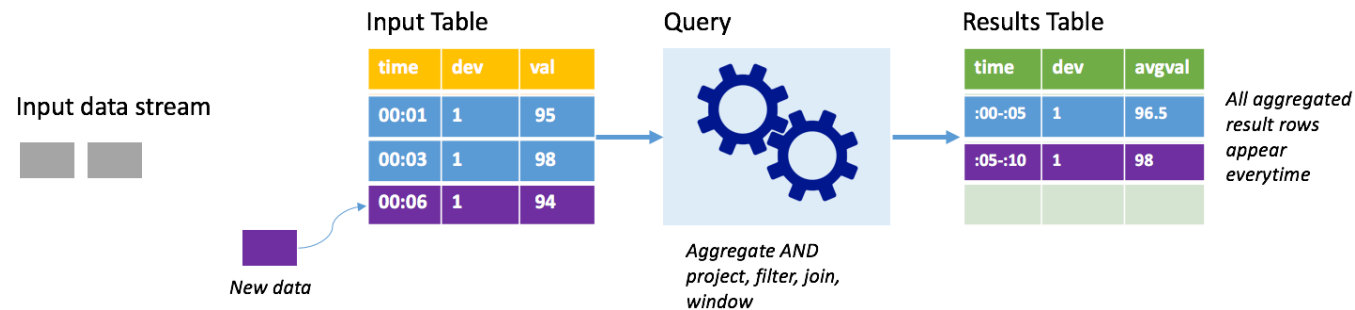


Streams como tablas

- Modo Append



- Modo Completo



Deep Learning

- Soporte Enriquecido de DL
 - Tensor
 - Redes Neuronales
 - Carga de Modelos Caffe, Torch, Keras
- Muy Alto Rendimiento -> Intel MKL
- Escalado

