



Programa de formación MLDS



Ben Chams - Fotolia

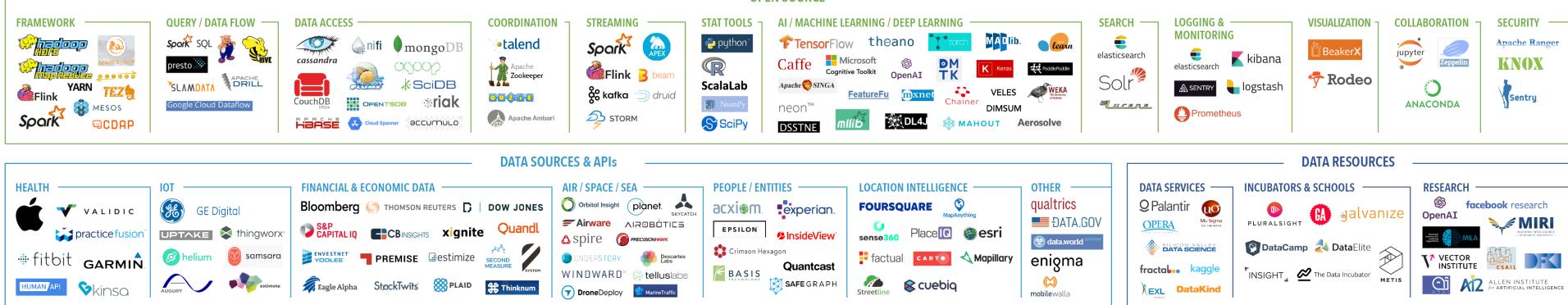
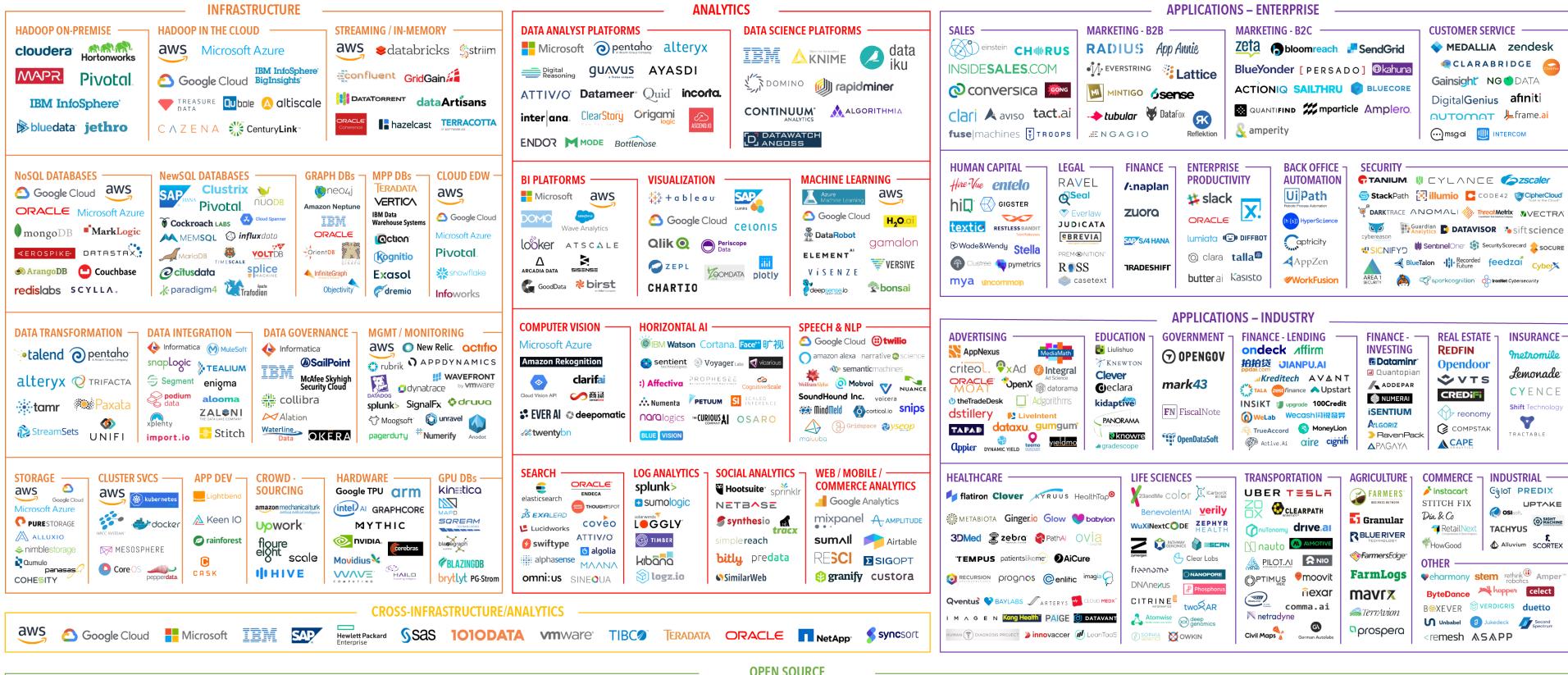
Ben Chams - Fotolia



Módulo BIG DATA Apache Spark

Por
Ing. Jorge Camargo, Ph.D.

BIG DATA & AI LANDSCAPE 2018



Agenda



1. Apache Spark
2. Arquitectura
3. Modelo de Programación
4. Interfaces de Datos
5. Transformaciones
6. Acciones
7. Ejemplos



Apache Spark



- Es un motor general para procesamiento de datos a gran escala, rápido y fácil de usar
- Rápido:
 - Ejecuta programas 100x más rápido que Hadoop Map-Reduce en memoria
 - Ejecuta programas 10x más rápido en disco
- Fácil de Usar:
 - Proporciona APIs para Scala, Java, Python y R
 - Provee librerías para:
 - Manipulación de datos con SQL and DataFrames
 - Aprendizaje de máquina con MLlib
 - GraphX
 - Spark Streaming

Apache Spark



- Generalidad:
 - Combina SQL, Streaming y análisis complejos.
- Múltiples modos de ejecución:
 - Hadoop
 - Mesos
 - Standalone
 - Cloud
- Amplio Acceso a datos:
 - HDFS
 - Cassandra
 - Hbase
 - S3 (Amazon)

Apache Spark



- Abstira la complejidad de procesar datos de manera distribuida en clúster.
- Implementa el modelo Map-Reduce
- Motor de ejecución en DAG (Directed Acyclic Graph)
 - Los nodos representan datos (RDDs, DataFrames, Datasets)
 - Los arcos representan transformaciones y acciones sobre los datos

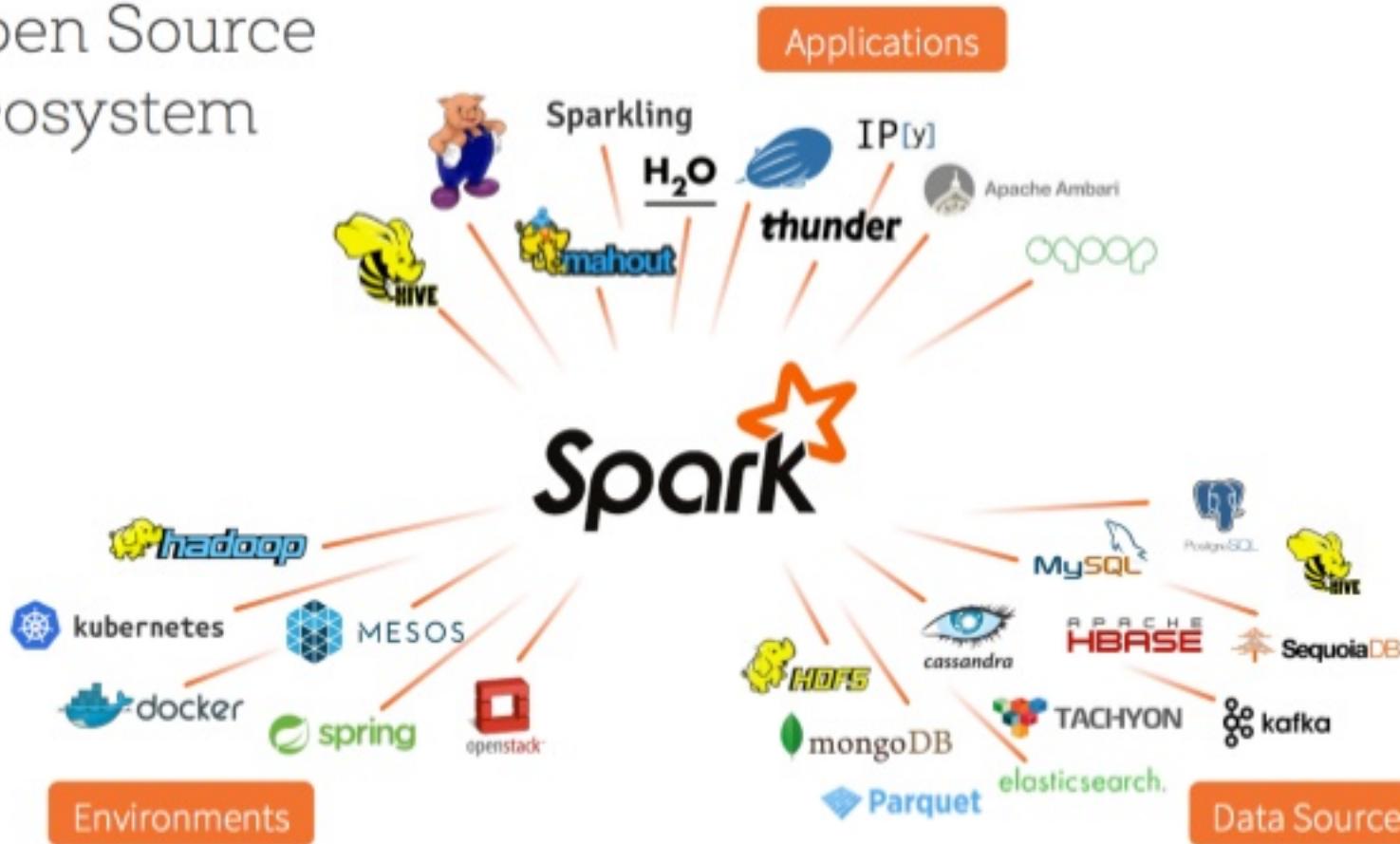
Apache Spark



UNIVERSIDAD
NACIONAL
DE COLOMBIA

- Apache en el ecosistema Open Source

Open Source
Ecosystem



Arquitectura



Spark SQL +
DataFrames

Streaming

MLlib
Machine Learning

GraphX
Graph Computation

Spark Core API

R

SQL

Python

Scala

Java

Spark SQL



- Permite la integración con SQL o a través de la API DataFrame.
- El acceso a los datos es uniforme, sin importar la fuente de datos.
- Integración sencilla con Hive
- Se puede exponer como una base de datos.



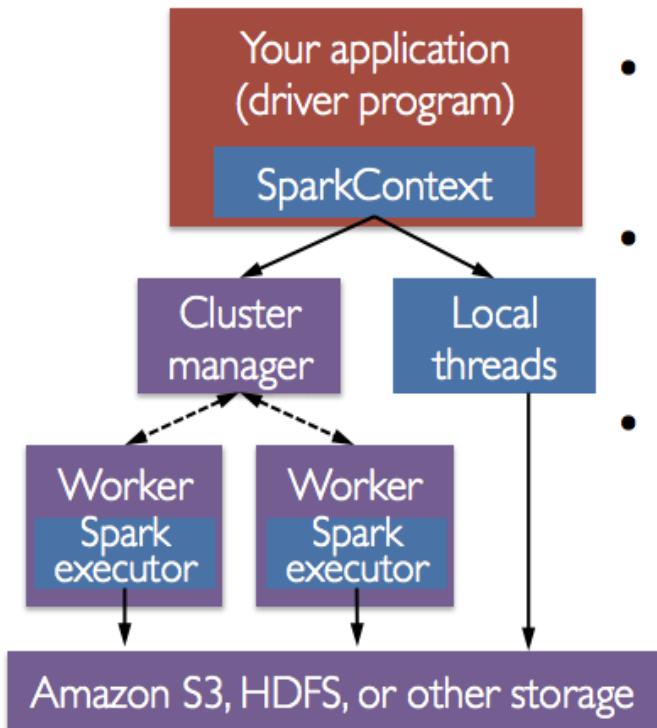
Spark MLLib



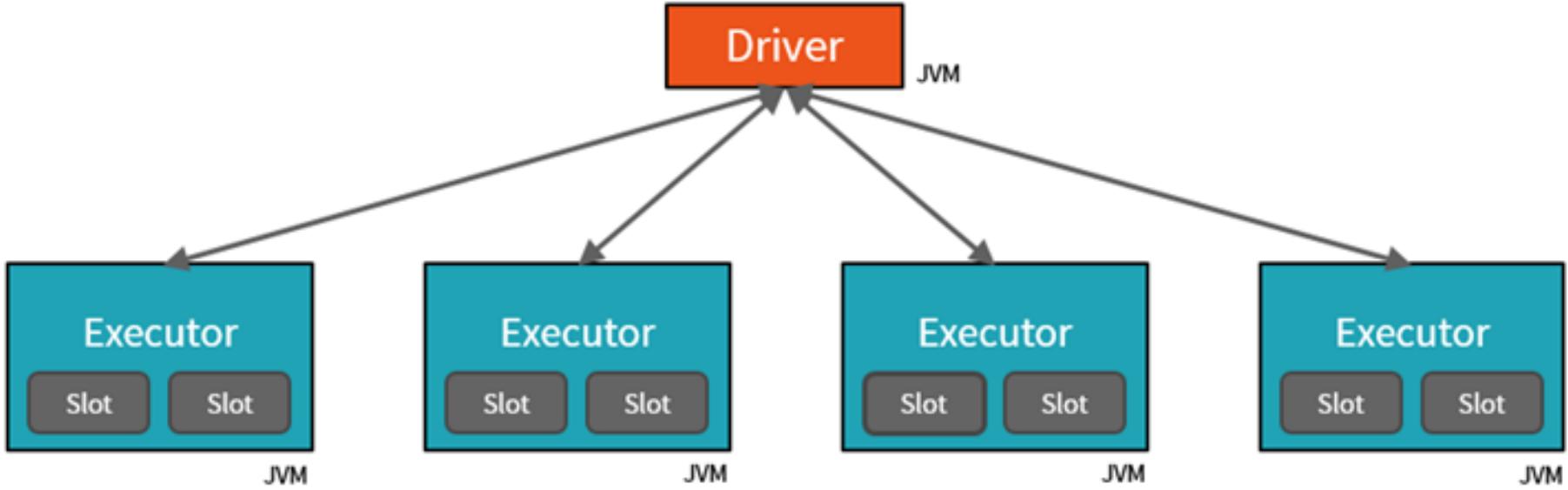
- Cuenta con una integración con Numpy y las APIs de Spark.
- Algoritmos de ML: Random Forest, Logistic Regression, Kmeans, entre otros.
- Herramientas para flujos de trabajo en ML: Pipelines, transformación de características, evaluación de modelos, guardado de modelos.

Arquitectura

- Spark utiliza una arquitectura Maestro-Esclavo
- La aplicación principal se ejecuta en el nodo maestro (*driver program*)
- Las tareas distribuidas se ejecutan en los nodos esclavos (*workers*)
- *SparkContext* mantiene y coordina la conexión al clúster

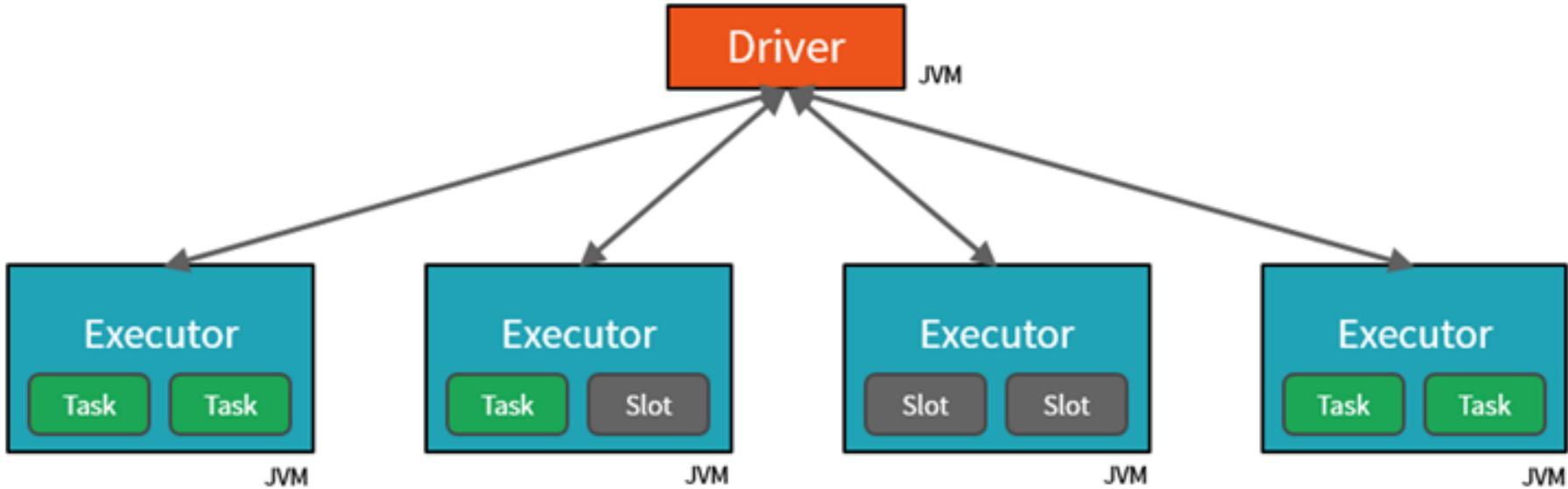


Arquitectura



- El driver envía tareas a los espacios disponibles en los ejecutores.

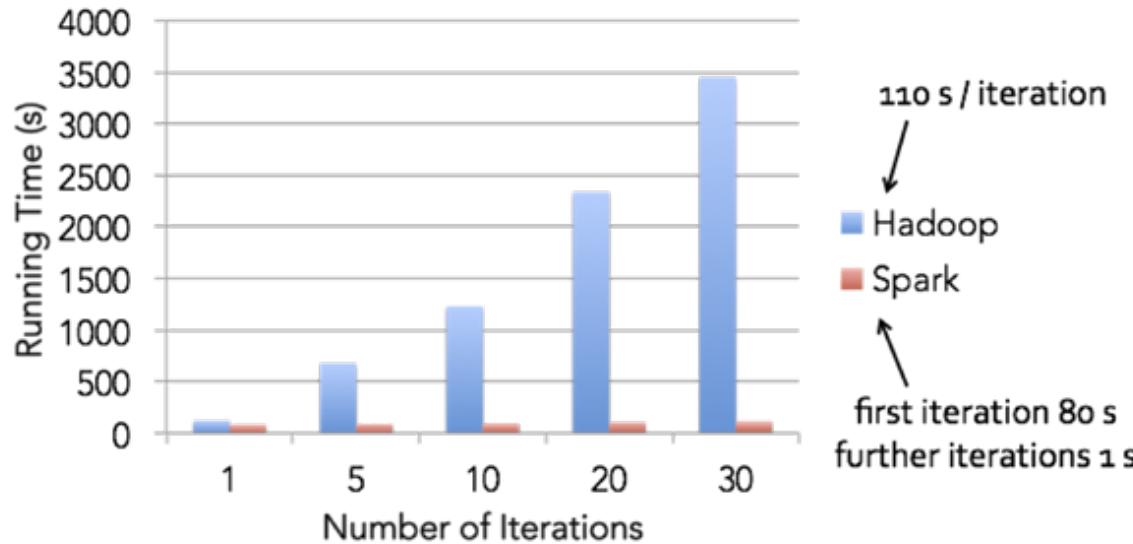
Arquitectura



- El driver envía tareas a los espacios disponibles en los ejecutores.

Arquitectura

- La ejecución es mucho más eficiente que en Hadoop.
- Comparación para la tarea WordCount usando MapReduce:



Modelo de Programación



- Estructura de una aplicación básica en Spark:
 1. Conectar al clúster mediante el *SparkContext*
 2. Cargar datos a memoria en formato *Resilient Distributed Dataset (RDDs)*, *DataFrame* o *Dataset*
 3. Aplicar transformaciones y acciones sobre los datos
 4. Persistir los datos resultantes (en base de datos, HDFS, archivos de texto plano, etc.)

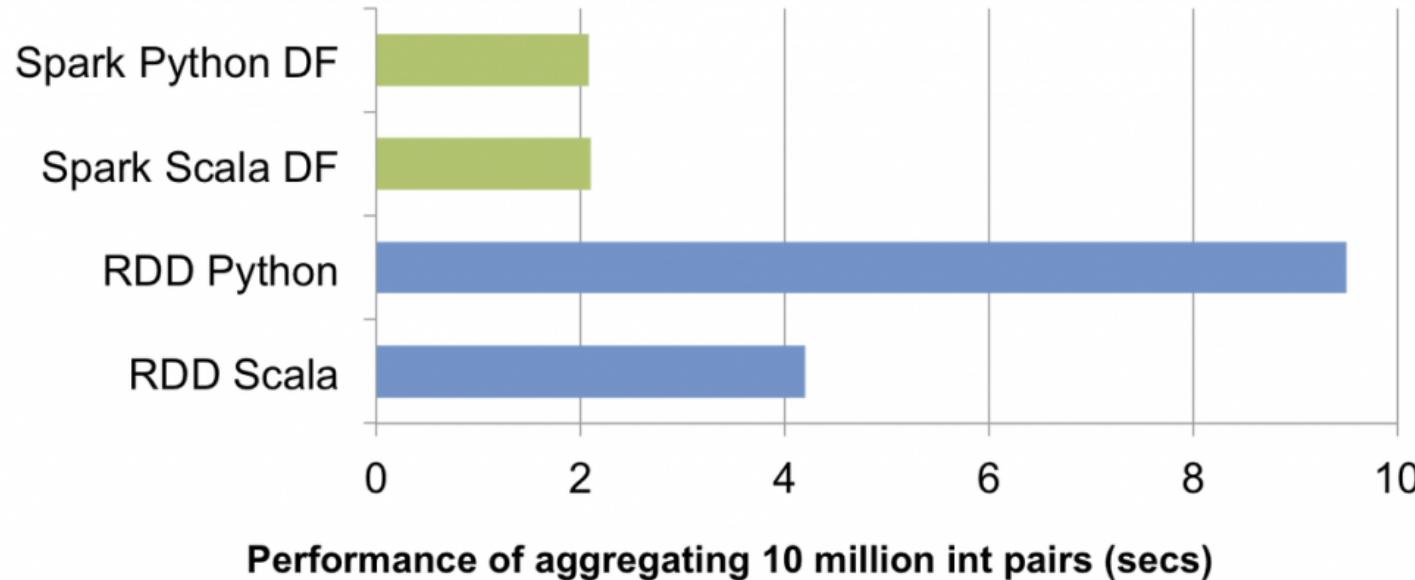
Interface de Datos



- Spark utiliza tres interfaces clave para la manipulación de datos:
 - *Resilient Distributed Data Set (RDD)*
 - *DataFrame*
 - *Dataset*

Interface de Datos

- RDD vs DataFrame



Resilient Distributed Dataset



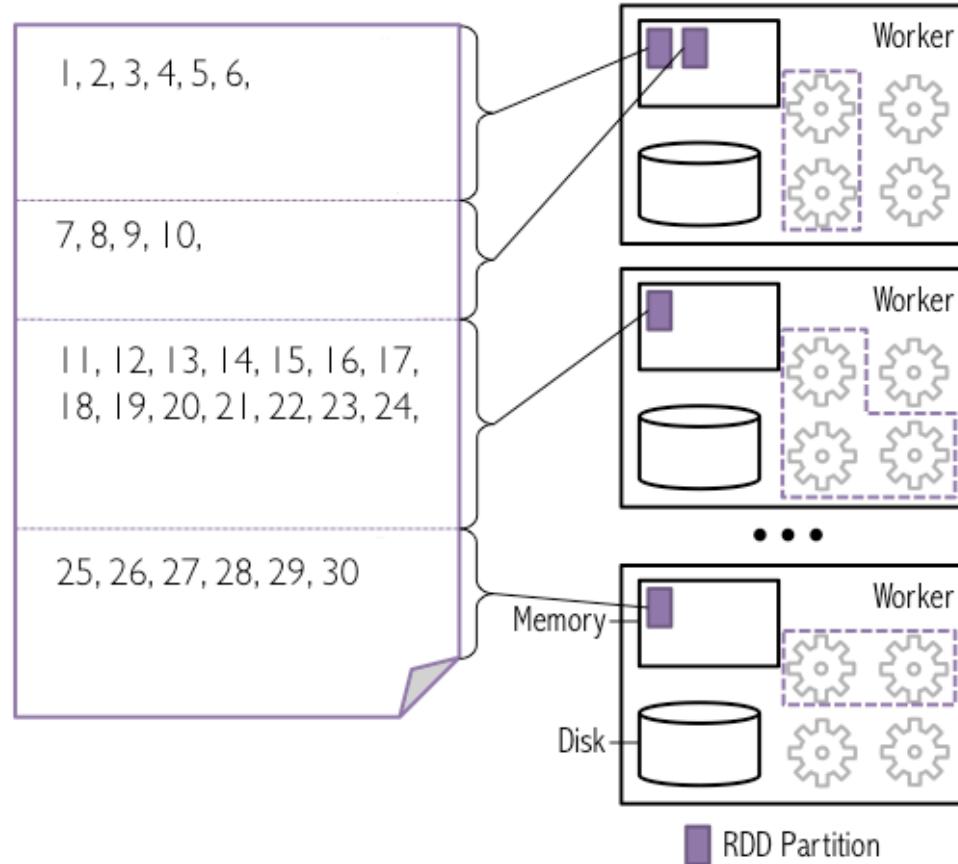
- Colección de elementos tolerante a fallos que puede ser operada en paralelo
- Proporcionan la API de bajo nivel para la manipulación de datos
- Un RDD es inmutable una vez creado
- Existen tres maneras de crear RDDs:
 - Paralelizando colecciones en el *driver program*
 - Cargando datos desde HFDS, HBase, Cassandra, S3 o Sistemas de archivos locales
 - Obtención de RDDs mediante transformaciones a los RDDs existentes

Resilient Distributed Dataset

- Un RDD crea particiones de los datos que son almacenadas en la memoria de los *workers*

Dataset is broken into partitions

Partitions are each stored in a worker's memory



Resilient Distributed Dataset



- Creación de una RDD que contiene los números del 1 al 10000:

```
data = xrange(1, 10001)
# Parallelize data using 8 partitions
# This operation is a transformation of data into an RDD
# Spark uses lazy evaluation, so no Spark jobs are run at this point
xrangeRDD = sc.parallelize(data, 8)
```

Resilient Distributed Dataset



- Creación de un RDD a partir de una colección Python

```
data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)
```

- Creación de un RDD desde un archivo de texto

```
distFile = sc.textFile("data.txt")
```

DataFrame



- Colección de datos distribuidos organizados en columnas nombradas
- Equivalentes a una tabla en una base de datos relacional o *DataFrames* en Python
- Provee una API de manipulación de alto nivel
- Existen diferentes maneras de crear *DataFrames*:
 - Paralelizando colecciones en el *driver program*
 - Desde archivos de texto con datos estructurados (CSV, JSON)
 - Tablas en *Hive* o bases de datos externas
 - RDDs existentes

DataFrame

- Creación de un *DataFrame* que contiene los números en el rango [0, 100000]:

```
firstDataFrame = sqlContext.range(1000000)
```

- Creación de un *DataFrame* con un RDD existente:

```
data = sc.parallelize([('a', 1), ('b', 5), ('c', 3)])
df = sqlContext.createDataFrame(data)
```

- El *data frame* representará una tabla con columnas ‘a’, ‘b’, ‘c’

DataFrame



- Creación de un *DataFrame* desde un archivo CSV:

```
dataPath = "/databricks-datasets/Rdatasets/Rdatasets/data-001/csv/ggplot2/diamonds.csv"
diamonds = sqlContext.read.format("com.databricks.spark.csv")\
    .option("header", "true")\
    .option("inferSchema", "true")\
    .load(dataPath)

# inferSchema means we will automatically figure out column types
# at a cost of reading the data more than once
```

Transformaciones



- Operaciones que se aplican sobre conjuntos de datos RDDs, DataFrames, Datasets
- Algunas transformaciones comunes son *map*, *filter*, *select*, *join*, *union*, *distinct*, *groupBy*, *orderBy*
- Las transformaciones se ejecutan de manera *Lazy*. Spark recuerda las transformaciones y se ejecutan una vez se llama una acción sobre el conjunto de datos transformado

Acciones



- Comandos que son ejecutados sobre los conjuntos de datos. Generalmente ejecutan tareas de agregación, análisis, conteos, etc.
- Algunas acciones comunes son *show*, *count*, *collect*, *reduce*, *save*, *avg*
- Las acciones son ejecutadas inmediatamente y ejecutan en primer momento cualquier transformación que esté pendiente en el conjunto de datos

Acciones y Transformaciones

- Aplicación de la transformación *Map* y la acción *Reduce* en un RDD:
 - Calcular la longitud total del texto en el archivo

```
lines = sc.textFile("data.txt")
lineLengths = lines.map(lambda s: len(s))
totalLength = lineLengths.reduce(lambda a, b: a + b)
```

- Contar el número de veces que aparece una línea en el archivo

```
lines = sc.textFile("data.txt")
pairs = lines.map(lambda s: (s, 1))
counts = pairs.reduceByKey(lambda a, b: a + b)
```

Acciones y Transformaciones



- Aplicación de la transformación *groupBy* y la acción *avg* en un *DataFrame*
 - Calcular el precio promedio de los diamantes para cada par “cut”-“color”

```
df1 = diamonds.groupBy("cut", "color").avg("price") # a simple grouping
```

Ejemplos RDD

- WordCount

```
text_file = sc.textFile("hdfs://...")  
counts = text_file.flatMap(lambda line: line.split(" ")) \  
         .map(lambda word: (word, 1)) \  
         .reduceByKey(lambda a, b: a + b)
```

- Estimación de PI

```
def sample(p):  
    x, y = random(), random()  
    return 1 if x*x + y*y < 1 else 0  
  
count = sc.parallelize(xrange(0, NUM_SAMPLES)).map(sample) \  
       .reduce(lambda a, b: a + b)  
print "Pi is roughly %f" % (4.0 * count / NUM_SAMPLES)
```

Ejemplo DataFrame



```
# Constructs a DataFrame from the users table in Hive.  
users = context.table("users")  
# from JSON files in S3  
logs = context.load("s3n://path/to/data.json", "json")  
  
# Create a new DataFrame that contains "young users" only  
young = users.filter(users.age < 21)  
# Alternatively, using Pandas-like syntax  
young = users[users.age < 21]  
# Increment everybody's age by 1  
young.select(young.name, young.age + 1)  
# Count the number of young users by gender  
young.groupBy("gender").count()  
# Join young users with another DataFrame called logs  
young.join(logs, logs.userId == users.userId, "left_outer")
```

Referencias



- Documentación Apache Spark,
<http://spark.apache.org/docs>
- Documentación DataBricks,
<https://docs.cloud.databricks.com/docs/>
- Zaharia, M. et al, Learning Spark, O'reilly, 2015

¿Preguntas?

jecamargom@unal.edu.co

<http://www.mindlaboratory.org>