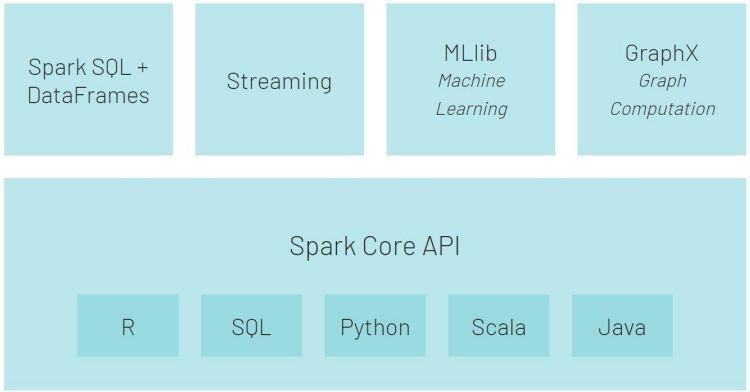
Taller introductorio a Spark

¿Que es Spark?

Apache Spark Ecosystem

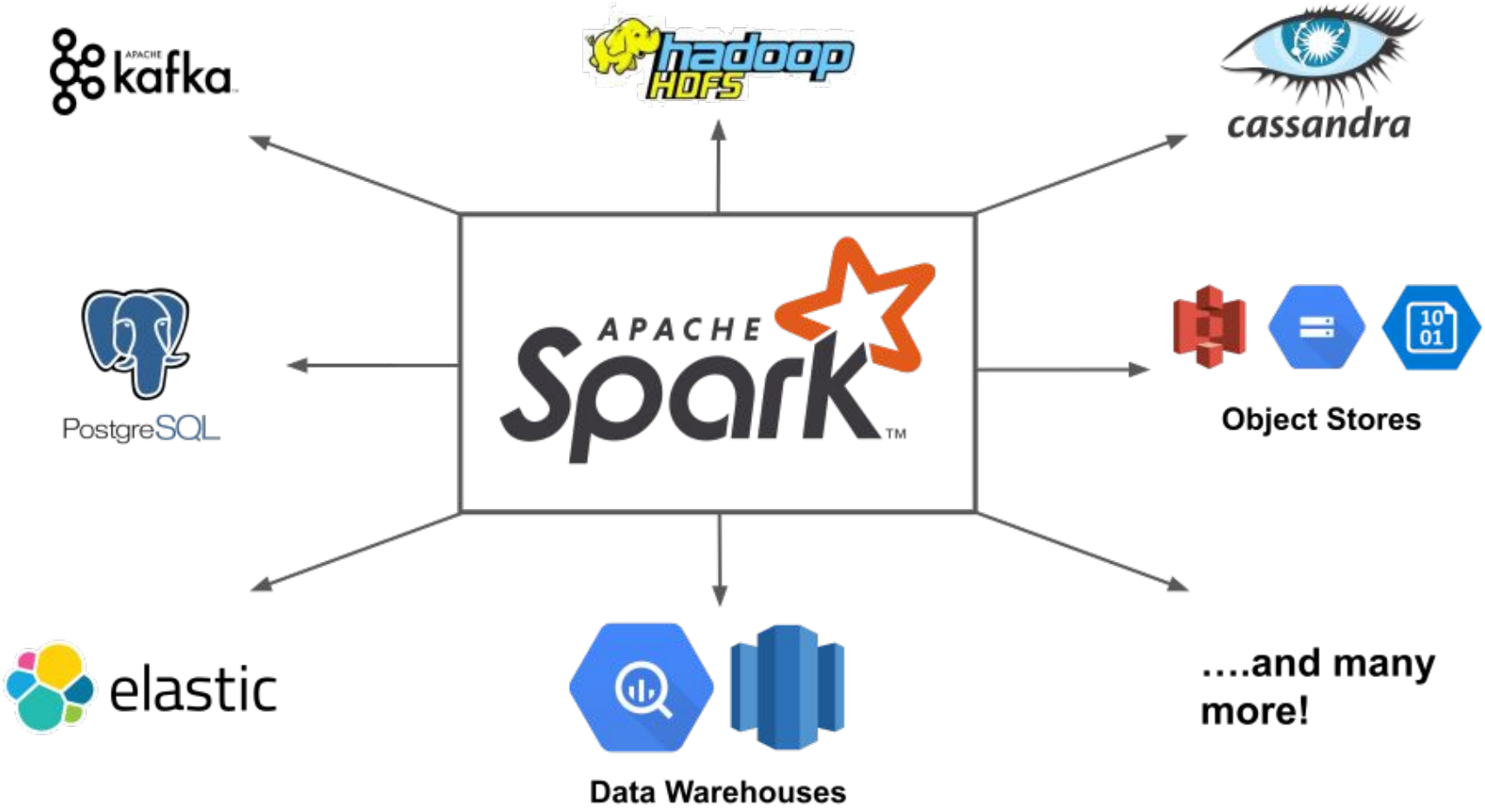
El núcleo Spark es un sistema de propósito general que proporciona programación, distribución y monitoreo de las aplicaciones en un clúster. Los componentes encima del núcleo están diseñados para interoperar estrechamente, permitiendo a los usuarios combinarlos.

El núcleo Spark está diseñado para escalar desde uno hasta miles de nodos. Puede ejecutarse en una variedad de administradores de clústers.



Big Data and Spark

- Data is increasing in volume, velocity, variety.
- The need to have faster results from analytics becomes increasingly important.
- Apache Spark is a computing platform designed to be *fast* and *general-purpose*, and *easy to use*
  - Speed
    - In-memory computations
    - Faster than MapReduce for complex applications on disk
  - Generality
    - Covers a wide range of workloads on one system
    - Batch applications (e.g. MapReduce)
    - Iterative algorithms
    - Interactive queries and streaming
  - Ease of use
    - APIs for Scala, Python, Java
    - Libraries for SQL, machine learning, streaming, and graph processing
    - Runs on Hadoop clusters or as a standalone


kafka
hadoop HDFS
cassandra
PostgreSQL
APACHE Spark™
Object Stores
elastic
Data Warehouses
….and many more!

Who uses Spark and why?

- Parallel distributed processing, fault tolerance on commodity hardware, scalability, in-memory computing, high level APIs, etc.
- Saves time and money
- Data scientist
  - Analyze and model the data to obtain insight using ad-hoc analysis
  - Transforming the data into a useable format
  - Statistics, machine learning, SQL
- Engineers
  - Develop a data processing system or application
  - Inspect and tune their applications
  - Programming with the Spark's API
- Everyone else
  - Ease of use
  - Wide variety of functionality
  - Mature and reliable

Spark Streaming

Muchas aplicaciones necesitan la capacidad de procesar y analizar no solo datos por lotes, sino también flujos de datos nuevos en tiempo real. Spark Streaming permite potentes aplicaciones analíticas interactivas tanto en stream como en datos históricos. Se integra fácilmente con una amplia variedad de fuentes de datos populares, incluyendo HDFS, Flume, Kafka y Twitter.

Spark Machine Learning

MLlib es la biblioteca de aprendizaje automático (ML) de Spark. Su objetivo es hacer que el aprendizaje automático práctico sea escalable y fácil. En un nivel alto, proporciona herramientas como:

- ML Algorithms: algoritmos de aprendizaje comunes como clasificación, regresión, agrupamiento y filtrado colaborativo
- Featurization: extracción de características, transformación, reducción de dimensionalidad y selección
- Pipes: herramientas para construir, evaluar y ajustar pipes de ML
- Persistence: guardar y cargar algoritmos, modelos y pipes
- Utilities: álgebra lineal, estadísticas, manejo de datos, etc.

Spark GraphX

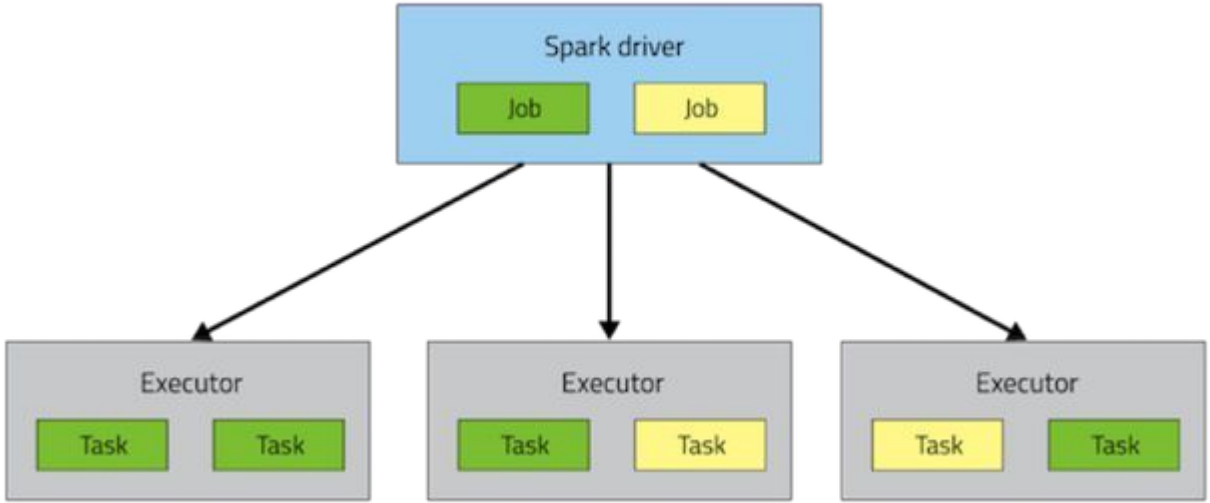
GraphX es un motor de cálculo de grafos que permite construir, transformar y analizar interactivamente sobre datos estructurados. Viene con una biblioteca completa de algoritmos comunes entre los que se encuentran:

- PageRank
- Connected components
- Label propagation
- SVD++
- Strongly connected components
- Triangle count

Modelo de ejecución

Spark execution model

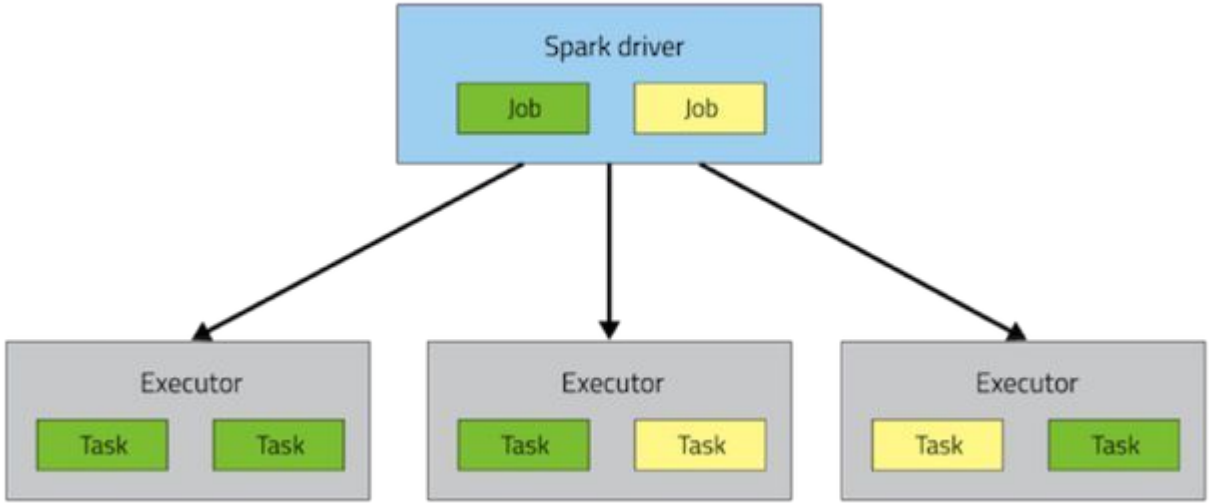
La ejecución de una aplicación Spark implica conceptos como **driver, executor, task, job,** y **stage**.



Spark execution model

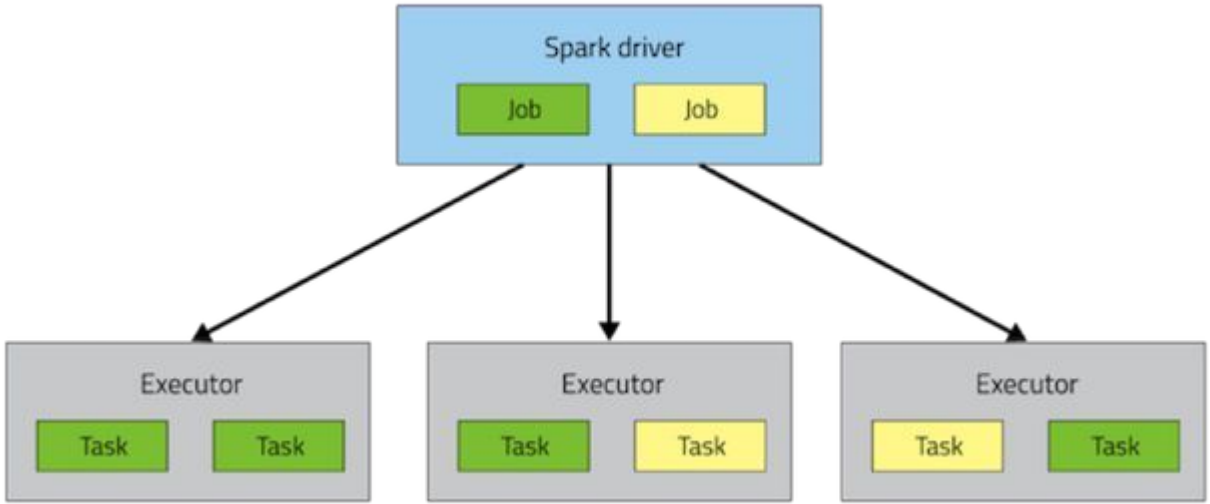
En tiempo de ejecución una aplicación Spark se mapea a un **driver** único y a un conjunto de procesos de ejecución distribuidos entre los nodos de un clúster.

El **driver** de procesos gestiona el flujo de **jobs** y **tasks**, el driver está disponible todo el tiempo que se ejecuta la aplicación.



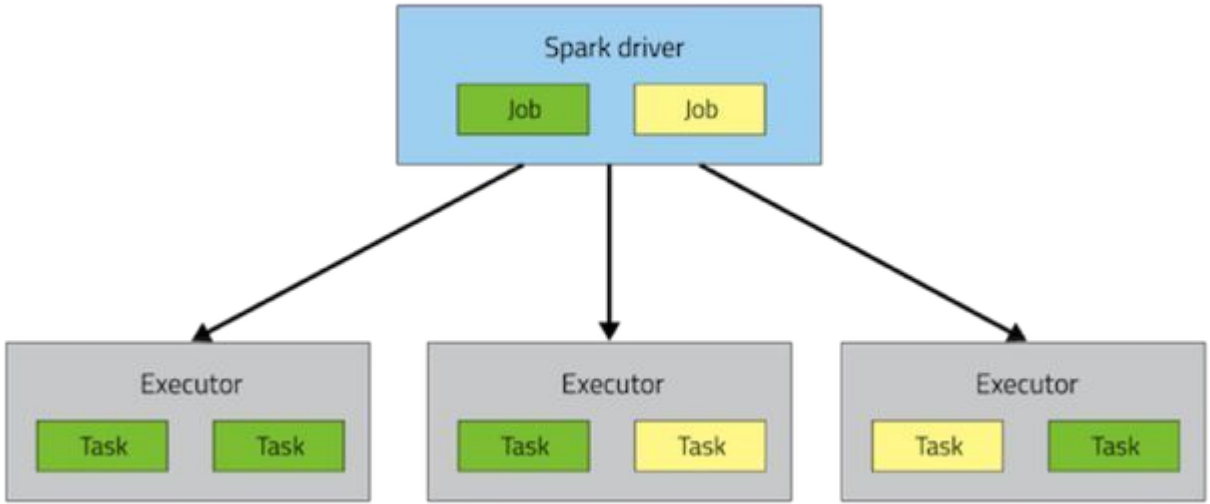
Spark execution model

Los **executors** son responsables de realizar el **job**, en forma de **tasks**, así como de guardar en caché cualquier información generada.



Spark execution model

Invocar una acción dentro de una aplicación Spark desencadena el inicio de un **job**. Spark examina el conjunto de datos del que depende esa acción y formula un plan de ejecución. El plan de ejecución ensambla las transformaciones del conjunto de datos en **stages**. Un **stage** es una colección de tareas que ejecutan el mismo código, cada una en un subconjunto diferente de datos.



Resilient Distributed Dataset

La abstracción principal del núcleo de Spark se llama Resilient Distributed Dataset o RDD. Esencialmente, es solo una colección distribuida de elementos que se paraleliza en todo el clúster. Existen dos tipos de operaciones RDD: transformaciones y acciones.

- Las transformaciones son aquellas que no devuelven un valor. Spark simplemente crea estos gráficos acíclicos directos o DAG, que sólo se evaluarán en tiempo de ejecución (**evaluación perezosa)**
- . El aspecto de tolerancia a fallos de los RDD permite a Spark reconstruir las transformaciones utilizadas para construir el linaje para recuperar los datos perdidos.
- Las acciones son cuando las transformaciones se evalúan junto con la acción que se requiere para ese RDD. **Las acciones devuelven valores**. El primer paso es cargar el conjunto de datos desde Hadoop. Luego aplica transformaciones sucesivas en él, como filtrar, asignar o reducir. Nada sucede hasta que se llama a una acción. **El DAG se actualiza cada vez hasta que se llama a una acción**. Esto proporciona tolerancia a fallos.

Resilient Distributed Dataset

- Las acciones son cuando las transformaciones se evalúan junto con la acción que se requiere para ese RDD. **Las acciones devuelven valores**. El primer paso es cargar el conjunto de datos desde Hadoop. Luego aplica transformaciones sucesivas en él, como filtrar, asignar o reducir. Nada sucede hasta que se llama a una acción. **El DAG se actualiza cada vez hasta que se llama a una acción**. Esto proporciona tolerancia a fallos.

- Por ejemplo, digamos que un nodo se desconecta. Todo lo que necesita hacer cuando vuelva a estar en línea es volver a evaluar el gráfico hasta donde lo dejó. El almacenamiento en caché se proporciona con Spark para permitir que el procesamiento ocurra en la memoria. Si no cabe en la memoria, se derramará al disco.

RDD operations - Transformations

- A subset of the transformations available. Full set can be found on Spark's website.
- Transformations are lazy evaluations
- Returns a pointer to the transformed RDD

Transformation	Meaning
map(func)	Return a new dataset formed by passing each element of the source through a function *func*.
filter(func)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(func)	Similar to map, but each input item can be mapped to 0 or more output items. So func should return a Seq rather than a single item
join(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key.
reduceByKey(func)	When called on a dataset of (K, V) pairs, returns a dataset of (K,V) pairs where the values for each key are aggregated using the given reduce function *func*
sortByKey([ascending], [numTasks])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K,V) pairs sorted by keys in ascending or descending order.

RDD operations - Actions

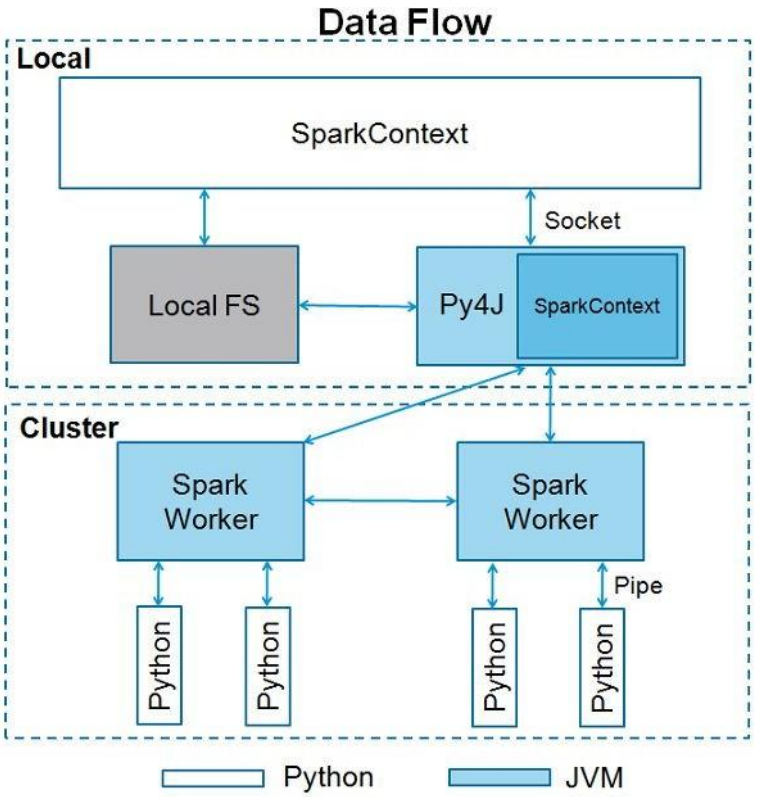
- Actions returns values

Action	Meaning
collect()	Return all the elements of the dataset as an array of the driver program. This is usually useful after a filter or another operation that returns a sufficiently small subset of data.
count()	Return the number of elements in a dataset.
first()	Return the first element of the dataset
take(n)	Return an array with the first n elements of the dataset.
foreach(func)	Run a function func on each element of the dataset.

Spark Context

SparkContext

Es el punto de entrada a cualquier funcionalidad de Spark. Cuando ejecutamos cualquier aplicación, se inicia un programa **driver**, que contiene la función main y el **SparkContext** se inicia aquí.



SparkContext Class

```
class pyspark.SparkContext (
   master = None,
   appName = None,
   sparkHome = None,
   pyFiles = None,
   environment = None,
   batchSize = 0,
   serializer = PickleSerializer(),
   conf = None,
   gateway = None,
   jsc = None,
   profiler_cls = <class 'pyspark.profiler.BasicProfiler'> )
```

SparkContext Class

Algunos de los parámetros más relevantes:

- ***master***: Es la URL del clúster al que se conecta (remota o localmente).
- **appName**: nombre de la aplicación con que se identifica en el cluster
- **conf**: lista de opciones de spark

SparkConf Class

Un objeto Lista {SparkConf} para establecer propiedades de Spark, puede ver la lista completa en https://spark.apache.org/docs/latest/configuration.html

- **spark.executor.cores**: El número de núcleos a usar en cada ejecutor
- **spark.cores.max**: Cantidad máxima de núcleos de CPU que puede solicitar la aplicación al cluster
- **spark.executor.memory**: Cantidad de memoria a utilizar por proceso ejecutor, soporta sufijos de unidad de tamaño ("k", "m", "g" o "t", por ejemplo, 512m, 2g).

SparkConf Class

Ejemplo, se solicitan 4 núcleos y memoria de 4 gigabytes:

```
conf = pyspark.SparkConf()
conf.set('spark.executor.cores', '4')
conf.set('spark.cores.max', '4')
conf.set('spark.executor.memory', '4g')
```

Iniciando un contexto

```
# crear el contexto de spark
conf = pyspark.SparkConf()

conf.set('spark.executor.cores', '4')
conf.set('spark.cores.max', '4')
conf.set('spark.executor.memory', '4g')
# local
sc = pyspark.SparkContext(master="local",appName="MyApp",conf=conf)
# remoto
sc = pyspark.SparkContext(master="spark://10.10.22.162:7077",appName="MyApp",
                          conf=conf)
```

Spark SQL

Spark SQL, DataFrames and Datasets

Spark SQL es un módulo de para el procesamiento de datos estructurados. Trabaja sobre **Datasets**, que son un conjunto de datos distribuidos.

Un **DataFrame** es un **Dataset** organizado en columnas con nombre. Es conceptualmente equivalente a una tabla en una base de datos relacional. Los **DataFrames** se pueden construir a partir de una amplia variedad de fuentes, como archivos de datos estructurados (csv, json, xml, etc), tablas en Hive, bases de datos externas o **RDD** existentes. En este tipo de datos se pueden realizar consultas **SQL**.

Los **RDD (Resilient Distributed Datasets)** son un conjunto de objetos Java o Scala que representan una colección de elementos particionados en los nodos del clúster que se pueden operar en paralelo.

Características de los DataFrame/RDD

- **De naturaleza inmutable**: podemos crear DataFrames/RDD pero no podemos cambiarlos. Se les pueden aplicar transformaciones.
- **Evaluaciones perezosas**: lo que significa que una tarea no se ejecuta hasta que se realiza una acción.
- **Distribuido**: RDD y DataFrame son distribuidos por naturaleza.

Ventajas de DataFrame

- Están diseñados para procesar una gran colección de datos estructurados o semiestructurados.
- Los datos se organizan en columnas con nombre, lo que ayuda a Spark a comprender el esquema de un DataFrame para optimizar el plan de ejecución en sus consultas.
- Tienen la capacidad de manejar petabytes de datos.
- Tiene soporte para una amplia variedad de fuentes y formatos de datos.
- Tiene soporte para APIs de diferentes lenguajes como Python, R, Scala, Java.

Historia de las APIs

Distribute collection of JVM objects

Functional Operators (map, filter, etc.)

Distribute collection of Row objects

Expression-based operations and UDFs

Logical plans and optimizer

Fast/efficient internal representations

Internally rows, externally JVM objects

Almost the “Best of both worlds”: **type safe + fast**

But slower than DF
Not as good for interactive analysis, especially Python

Cargando archivos

sql.SparkSession.read

Interfaz utilizada para cargar un DataFrame desde sistemas de almacenamiento externo (archivos locales o remotos).

Este ejemplo muestra cómo cargar un archivo csv, se pueden exportar archivos de varios orígenes:

```
# archivo local
input_data = sqlContext.read.csv("my_file.csv")

# desde un servidor hadoop de archivos

input_data = sqlContext.read.csv("hdfs://10.10.22.162:9000/user/hadoop/my_file.csv")

# desde Amazon Simple Storage Service S3 (requiere configuración de llaves)

input_data = sqlContext.read.csv("s3n://my_file.csv")
```

sql.SparkSession.read

Interfaz utilizada para cargar un DataFrame desde sistemas de almacenamiento externo (archivos locales o remotos).

Este ejemplo muestra cómo cargar un archivo json, se pueden exportar archivos de varios orígenes:

```
# archivo local
input_data = sqlContext.read.json("my_file.json")

# desde un servidor hadoop de archivos

input_data = sqlContext.read.json("hdfs://10.10.22.162:9000/user/hadoop/my_file.json")

# desde Amazon Simple Storage Service S3 (requiere configuración de llaves)

input_data = sqlContext.read.json("s3n://my_file.json")
```

DataFrame

Detalles del dataframe cargado

```
print( type(input_data) )
input_data.printSchema()

input_data.show()
```

Crear una vista para manejar los datos:

```
input_data.createOrReplaceTempView("zonas")
```

DataFrame SQL

Se pueden hacer consultas con sentencias SQL sobre los DataFrames, la función sql ejecuta la sentencia y regresa un DataFrame con los datos de la consulta:

```
# seleccionar las columnas CVE_ZM, NOM_ZM y POB_2015, en caso de que el nombre de la columna
# Tenga caracteres conflictivos (como espacios o caracteres especiales), se pueden usar
# comillas invertidas ` `
result = sqlContext.sql("SELECT CVE_ZM, NOM_ZM, POB_2015 from zonas")
result.show()
```

El resultado siempre es un DataFrame.

Funciones DataFrame

Algunas funciones relevantes del DataFrame:

- **select**: genera un DataFrame con las columnas requeridas
- **rdd**: convierte el DataFrame a RDD
- **limit**: genera un DataFrame con los primeros “N” renglones especificados.
- **show**: Imprime en pantalla los primeros 20 renglones especificados.
- **filter**: genera un DataFrame con los renglones donde una condición es verdadera.
- **where**: es un alias de **filter**.
- **collect**: Copia al host todo el DataFrame en forma de lista de renglones.
- **withColumn**: Modifica los valores de una columna o agrega una nueva columna.

Distributed matrix

Distributed matrix

En algún momento, los datos se requieren en otra forma para realizar operaciones, un tipo especial son Matrices Distribuidas. Una matriz distribuida tiene índices de fila y columna de tipo long y valores de tipo doble, almacenados distributivamente en uno o más RDD. Spark implementa cuatro tipos de matrices distribuidas:

- RowMatrix
- IndexedRowMatrix
- CoordinateMatrix
- BlockMatrix

RowMatrix

Resenta una matriz distribuida orientada a renglones sin índices de renglones significativos.

```
from pyspark.mllib.linalg.distributed import RowMatrix
from pyspark.mllib.linalg import Vectors

data = result.select("idf").rdd.map(lambda row:
                        Vectors.sparse(row.idf.size, row.idf.indices, row.idf.values
                        )
                        )
mat = RowMatrix(data)
```

RowMatrix Functions

Algunas de las operaciones que se pueden hacer en Matrices.

- **multiply**: Multiplica esta matriz por una matriz local densa a la derecha.
- **computeCovariance**: Calcula la matriz de covarianza, interpretando cada fila como una observación.
- **computeGramianMatrix**: Calcula la matriz de Gram
- **computeSVD**: Calcula los factores de descomposición de valores singulares (SVD)
- **computePrincipalComponents**: Calcula los ***K*** componentes principales de la matriz dada

computeSVD

En nuestro ejemplo, se aplicad SVD a nuestra matriz, reduciendo la dimensión (columnas de características) a solo 100 elementos.

```
# Aquí solo trabajamos con la matriz U

svd = mat.computeSVD(100, computeU=True)
U = svd.U

s = svd.s

V = svd.V
```

Convertir de Matrix a DataFrame

Todos los manejadores de Datos en Spark tienen la función **map**, **filter** y **reduce**. Se pueden utilizar para realizar operacione sobre los items.

En este ejemplo convertimos la matriz a un Dataframe

```
u = U.rows.map(lambda x: (x, )).toDF(['features'])
```

Buscar dato

Ahora buscaré los registros que tengan la palabra “matrimonio”, se imprimen los primeros 5 y elegiré el id del registro que me interese:

```
sqlContext.sql("select `registro`,index as id, rubro from documentos").where(
 F.col("texto").contains('matrimonio') ).show(5,False)
```

```
id_target = 253
```

map, filter, reduce

Los **DataFrame** y **RDD** poseen funciones tales como ***map***, ***filter*** y ***reduce***. Que son útiles para manipular los datos, en este ejemplo extraemos el vector que corresponde al id del registro que nos interesa, el vector es de tipo **VectorSparse**, por lo que lo convertimos al numpy array con la función **toArray()**.

Finalmente se “bajan” los datos con **collect**:

```
target = vectors.filter(lambda x:x[1]==id_target).map(lambda x:x[0].toArray()).collect()
# notar que collect regresa una lista, en este caso de 1 elemento

target
```

map, filter, reduce

Finalmente se calcula la distancia entre el vector elegido contra el resto de los vectores, para esto se calcula la distancia coseno con **cosine**, que es parte de scipy.

```
from scipy.spatial.distance import cosine
import numpy as np

distances = vectors.map(lambda x:  cosine(target[0], x[0].toArray()) ).collect()
# se ordenan las distancias de menor a mayor

distances = np.array(distances)
# se usan los índices que son relativos a los ids de los registros
idx = np.argsort(distances)
```

map, filter, reduce

Finalmente se extraen los top 5 registros, basados en las distancias. En esta sentencia tambien se agrega el valor de distancia:

```
rows = sqlContext.sql(
        "select `registro`,index as id, texto from documentos"
    ).rdd.filter(
        lambda x: x.id in idx[:5]
    ).map(
        lambda x: (distances[x.id], x.registro, x.texto)
    ).collect()
```