PUCE Virtual

Sistemas de Big Data

Ecosistema Spark

Clase 5

MAESTRÍA EN
SISTEMAS DE INFORMACIÓN
Mención Data Science

La excelencia no se improvisa

INTRODUCCIÓN

El ecosistema ***Apache Spark*** se ha consolidado como una de las plataformas más poderosas y versátiles para el procesamiento de grandes volúmenes de datos, ofreciendo un entorno de trabajo de alto rendimiento en aplicaciones tanto en batch como en tiempo real. Una de sus herramientas clave es ***Spark SQL***, que permite realizar consultas estructuradas utilizando ***SQL*** tradicional; al mismo tiempo que integra capacidades de procesamiento distribuido sobre grandes conjuntos de datos (Zaharia et al., 2016). Este componente no solo soporta **consultas *SQL***, sino también operaciones sobre **Dataframes** y **Datasets**, lo que mejora la flexibilidad al trabajar con **datos estructurados** y **semiestructurados**. Por otro lado, ***Spark Streaming*** permite la manipulación de flujos de datos en tiempo real, utilizando la **arquitectura** de ***microbatching*** para procesar grandes volúmenes de datos a medida que llegan, lo que es ideal para aplicaciones en tiempo real, como análisis de redes sociales o monitoreo de sistemas (Karau & Kannan, 2017). El ecosistema también incluye ***Spark MLlib***, una librería dedicada al aprendizaje automático, que proporciona una amplia gama de algoritmos de ***machine learning*** optimizados para su ejecución en un entorno distribuido (Meng et al., 2016).

En cuanto a la parte de análisis de grafos, ***GraphX*** es otro componente esencial del **ecosistema Spark**, que facilita el procesamiento y análisis de grafos a gran escala. Permite a los usuarios realizar operaciones complejas sobre estructuras de grafos, como búsquedas, recorridos y cálculos de métricas de centralidad, todo de manera distribuida (Xin et al., 2013). Gracias a su integración con otros componentes del ecosistema, como ***Spark SQL*** y ***Spark MLlib***, ***GraphX*** es capaz de manejar tareas complejas de análisis y ***machine learning*** sobre datos estructurados y no estructurados en paralelo, proporcionando así una plataforma robusta y escalable, para resolver problemas de ***big data*** en diversas industrias. A través de la integración de estos componentes, el ecosistema ***Spark*** ofrece una solución completa para el procesamiento, análisis y modelado de datos a gran escala en tiempo real y batch (Zaharia et al., 2016).

RDA 2: Aplicar Técnicas de Big Data en grandes volúmenes de datos

Clase 5. Ecosistema *Spark*

La Figura 1 muestra información sobre el ecosistema de ***Spark.***

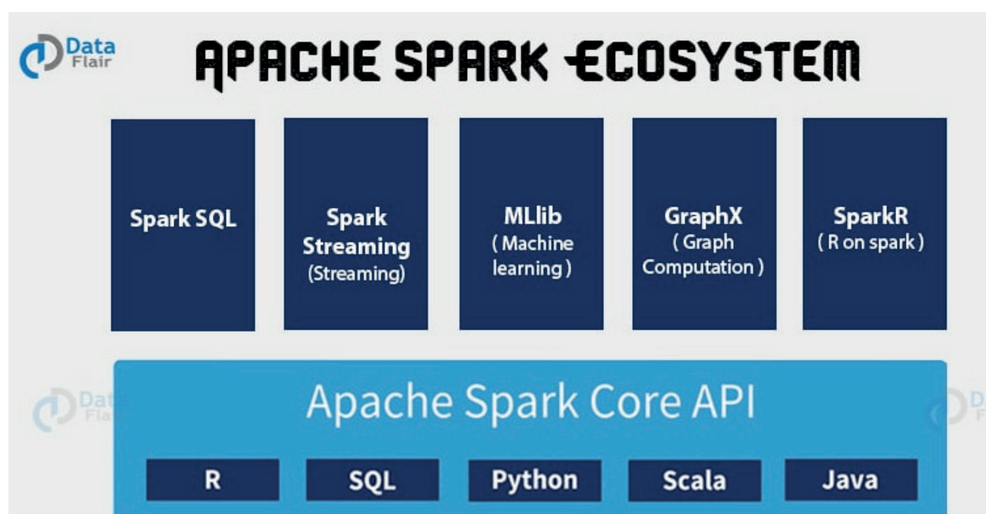

Figura 1. Ecosistema Spark.
Obtenida de: https://data-flair.training/blogs/apache-spark-ecosystem-components/

5.1. *Spark SQL*

En esta sección nos concentraremos en conseguir los siguientes objetivos:

- Procesar datos estructurados usando el módulo de ***Spark SQL.***
- Explicar los numerosos beneficios de trabajar con ***Spark SQL.***

Spark SQL es el componente de Spark que permite consultar datos estructurados y datos no estructurados usando un lenguaje de consulta común. Se puede conectar con múltiples fuentes de datos y brinda **APIs** para convertir el resultado de los querys a RDDs en programas de **Python**, **Scala** y **Java**.

Spark SQL brinda el mecanismo para que los usuarios de SQL puedan desarrollar querys en Spark. Adicionalmente, Spark SQL habilita herramientas de **Business Intelligence** para conectarse a Spark usando protocolos estándar, como JDBC y ODBC. Spark SQL también brinda APIs para convertir los query data en **dataframes,** para mantener datos distribuidos. Los dataframes son estructuras que están organizadas mediante columnas y lucen básicamente como si fueran tablas.

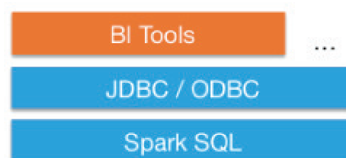

Figura 2. Standard Conectivity. Obtenida de: https://spark.apache.org/sql/

El primer paso para correr cualquier sentencia de **SQL** en **Spark** en crear el **SQLContext**. El segundo paso es crear un dataframe, con la finalidad de desarrollar operaciones complejas en el dataset. A continuación, se muestra un ejemplo del código que se debe usar. Ejemplo:

```
from pyspark.sql import SQLContext

sqlContext = SQLContext(sc)
```

Los **dataframes** pueden ser creados por:

- RDDs creados con anterioridad.
- Tablas de ***hive*** o alguna otra fuente de datos.

Un archivo puede ser leído y convertido a un *dataframe* usando un único comando. La función ***show*** mostrará el *dataframe* en el **Spark Shell**. Ejemplo:

```
# Leer

df = sqlContext.read.json("/filename.json")

# Mostrar

df.show()
```

Los **RDDs** pueden ser convertidos en **dataframes,** pero requieren un poco más de trabajo. Primero hay que convertir cada línea a una fila y una vez que los datos están en un **dataframe** se puede realizar una serie de **operaciones** de transformación en él. Tal y como se muestra en la ilustración, pueden ser de tipo *show*, *printSchema*, *select*, *filter* y *groupBy*.

```
#Leer

from pyspark.sql import SQLContext, Row sqlContext = SQLContext(sc)

# Cargar un archivo de texto y convertir cada línea en una fila.
```

```
lines = sc.textFile("filename.txt") cols = lines.map(lambda l: l.split(",")) data = cols.map(lambda p:
Row(name=p[0], zip=int(p[1])))

# Crear un dataframe

df = sqlContext.createDataFrame(data)

# Registrar el dataframe como tabla

df.registerTempTable("table")

# Ejecutar el SQL

Output = sqlContext.sql("SELECT * FROM table WHERE ...")

# Mostrar el contenido del dataframe

df.show()

# Imprimir el esquema

df.printSchema()

# Seleccionar solo la columna X

df.select("X").show()

# Seleccionar todos pero incrementar el descuento en un 5%

df.select(df["name"], df["discount"] + 5).show()

# Seleccionar la gente cuya altura sea superior a los 4.0 ft

df.filter(df["height"] > 4.0).show()

# Contar la cantidad de gente agrupada por zip

df.groupBy("zip").count().show()
```

En síntesis, *Spark SQL* permite ejecutar *querys* en *spark*, conectarse a una variedad de fuentes de datos y también *deployar* herramientas de ***business intelligence*** en *Spark.*

Documentación de Spark SQL [https://spark.apache.org/sql/]

5.2. *Spark Streaming*

En esta sección nos concentraremos en **alcanzar** los siguientes **objetivos**:

- Entender cómo funciona *Spark* datos en *streaming.*
- Listar algunas de las fuentes de datos en *streaming* que son soportadas por *Spark.*
- Describir la ventana de deslizamiento de *spark* (***sliding windows*** en Spark).

Spark Streaming brinda procesamiento escalable para datos en tiempo real, que corren en la parte superior de ***Spark Core.***

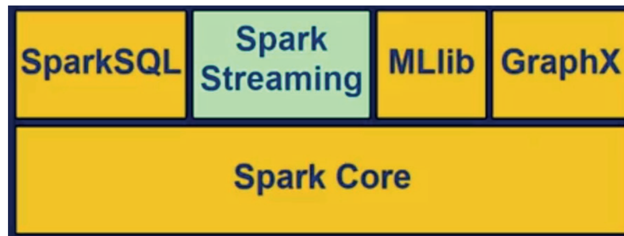

Figura 3. Spark Streaming. Obtenido de: https://spark.apache.org/docs/latest/streaming-programming-guide.html

Los datos continuos en ***streaming*** son convertidos o agrupados en ***RDDs*** discretos, que luego pueden ser procesados en paralelo. Spark streaming brinda ***APIs*** para Scala, Java y ***Python,*** así como otros productos de *Spark*. ***Spark Streaming*** puede leer datos desde varios tipos de fuentes, incluyendo ***Kafka*** y ***Flume***. *Apache Kafka* es una plataforma de transmisión de datos en tiempo real, que permite recopilar, procesar y almacenar grandes volúmenes de información; mientras que *Flume* recolecta y aplica funciones de agregación sobre *logs* de datos.

Figura 4. Spark Streaming Data Sources. Obtenido de: https://spark.apache.org/docs/latest/streaming-programming-guide.html

Spark Streaming también puede leer información desde fuentes de datos batch, tales como HDF, S3 y muchas otras bases de datos No SQL. Además, *Spark Streming* puede leer directamente de Twitter (X) y muchas más fuentes que proveen datos en tiempo real.

Creación y procesamiento en *Spark Streaming*:

Spark Streming lee datos en *streaming* y los convierte en ***microbatches,*** que se los conoce como ***DStreams,*** que es el nombre corto para *discretized stream* o en español ***stream*** discreto. En el siguiente ejemplo, se puede ver un flujo de datos en *streaming* de 10 segundos, que es convertido en cinco ***RDDs*** usando una longitud de *batch* de dos segundos.

De manera similar que otros ***RDDs***, transformaciones tales como *map*, *reduce* y *filter* pueden ser aplicadas a los ***DStreams***. *DStreams* pueden ser agregados en ventanas que te permiten aplicar procesamiento y ***sliding windows*** de datos. En este ejemplo, el tamaño de la ventana es de 4 segundos y el ***sliding interval*** es de 2 segundos.

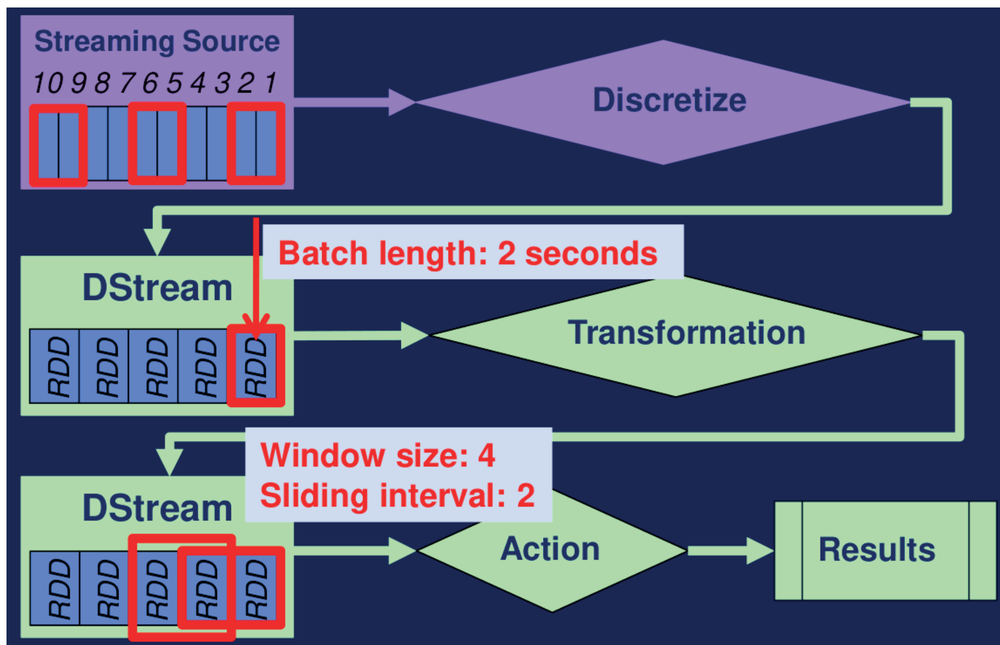

Figura 5. Procesamiento de datos con Spark Streaming:
Coursera Big Data Integration and Processing

Para concluir:

- ***Spark Streaming*** es una librería de *Spark* que permite trabajar con datos en *streaming* en un *near real time*.
- ***DStreams*** pueden ser usados como cualquier otro *RDD;* y pueden pasar por cualquier otra transformación, como un *dataset batch*.
- ***DStreams*** pueden crear una *sliding window,* para realizar cálculos en una ventana de tiempo.

Documentación de Spark Streaming [https://spark.apache.org/docs/latest/streaming-programming-guide.html]

5.3. *Spark MLlib*

El objetivo de esta sección es:

- Describir qué es y en qué consiste ***Mlib.***
- Listar las categorías principales de las técnicas disponibles en Mlib.
- Explicar parte del código contenido en los algoritmos de *Mlib*.

Mlib es una librería escalable de *machine learning* que corre en la parte superior de ***Spark Core.*** **Mlib** brinda una implementación distribuida de los algoritmos de *machine learning* más comunes, así como algunas funciones utilitarias. Al igual que *Spark Core*, *Mlib* tiene *APIs* para *Scala*, *Java*, *Python* y R.

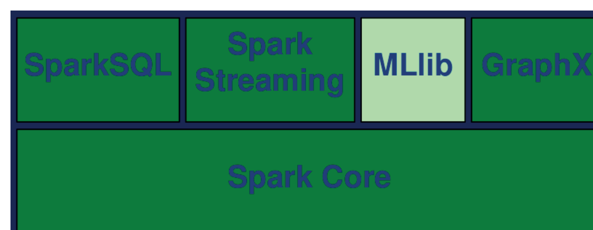

Figura 6. Spark Mlib. Obtenido de: https://spark.apache.org/mllib/

Mlib ofrece algunas técnicas y algoritmos comúnmente usados en un proceso de *machine learning*. Las principales categorías son: ***Machine Learning***, **Estadística** y algunas herramientas utilitarias para técnicas comunes. Tal como el nombre lo sugiere, varios de los algoritmos de ***machine learning*** están disponibles en ***Mlib***. Estos son algoritmos que se usan para construir modelos de clasificación, regresión y ***clustering***.

También hay técnicas para evaluar el resultado de los **modelos**. Por ejemplo, se pueden computar los valores, para la curva **ROC** (*Receiver of Creating Characteristics*), que es una técnica estadística muy común para graficar el ***performance*** de un modelo de clasificación binaria.

Mlib también provee funciones estadísticas; por **ejemplo,** podemos nombrar **estadísticas *summary***, promedios, desviación estándar, etc.; correlaciones y métodos para crear muestras sobre un *dataset*. *Mlib* también tiene técnicas comúnmente utilizadas en el proceso de ***machine learning,*** tales como reducción de dimensionalidad y métodos de **transformación** de ***features*** para preprocesamiento de datos. En pocas palabras, Spark Mlib ofrece algunas técnicas usadas frecuentemente en los pipelines de ***machine learning*.**

A continuación, se muestra el código en ***pyspark*** para relizar un ***summary statistics*** de columnas de un ***dataset*** de números:

```
# Importamos la librería Statistics desde el módulo stat de mlib

from pyspark.mllib.stat import Statistics

# Creamos los datos en un RDD

dataMatrix = sc.parallelize([ [1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12] ])

# Computamos las estadísticas de la columna

summary = Statistics.colStats(dataMatrix)

print(summary.mean())

print(summary.variance())
```

A continuación, se muestra un ejemplo para crear un **modelo** de **clasificación** binaria usando un **árbol** de **decisión.**

```
# Importamos las librerías

from pyspark.mllib.tree import DecisionTree, DecisionTreeModel

from pyspark.mllib.util import MLUtils



# Leemos y parseamos los datos

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



# Creamos el árbol de decisión para clasificación

model = DecisionTree.trainClassifier

(parsedData, numClasses=2)
```

```
print(model.toDebugString())

model.save(sc, "decisionTreeModel")
```

En este último ejemplo, se muestra el código para un modelo de *clustering* de tipo k medias.

```
# Cargamos las librerías

from pyspark.mllib.clustering import KMeans, KMeansModel

from numpy import array

# Leemos y parseamos los datos

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

parsedData = data.map(lambda line:

array([float(x) for x in line.split(' ')]))

# Definimos el modelo k means para clustering

clusters = Kmeans.train (parsedData, k=3)

print(clusters.centers)
```

En las próximas clases se hará mayor énfasis en algunos de los modelos de *machine learning;* por lo pronto, se brindan unos ejemplos de su implementación usando ***pyspark***.

5.4. *Spark Graph X*

Los objetivos de esta sección son:

- Describir en qué consiste Graph X.
- Explicar cómo los vértices y las aristas son almacenados.
- Describir a alto nivel cómo funciona Pregel.

Graph X es una **API** de Apache Spark procesamiento en paralelo de grafos. **Graph X** usa su propio modelo de grafos. Esto significa que tanto los nodos como los vértices en un grafo pueden tener atributos y valores. En **Graph X,** las propiedades de los nodos son almacenados en una taba de vértices y las propiedades de los ***edges*** (aristas) son almacenados en una tabla de ***edges***. La información de la conectividad, es decir, la información referente a la conexión entre **aristas** y **nodos,** se almacena de forma separada de las tablas que guardan las propiedades de las aristas y los nodos.

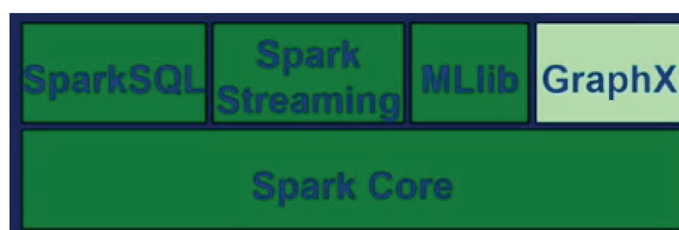

Figura 7. Spark GraphX. Obtenido de: https://spark.apache.org/graphx/

Graph X está creada a base de ***RDDs*** especiales para vértices y edges. ***VertexRDD*** representa un conjunto de vértices, en donde todos ellos tienen un atributo llamado A. El **EdgeRDD** que se muestra en la siguiente ilustración extiende el almacenamiento básico de un ***edge*** para los ***edges*** que tienen un formato columnar en cada partición, para hacer más eficiente el performance. Nótese que ***VertexID*** solo puede tener un valor único por definición. La clase ***edge*** es un objeto con un vértice como fuente y un atributo ***edge*** como vértice destino.

EdgeRDD[ED, VD] extends RDD[Edge[ED]]

Figura 8. EdgeRDD.

Adicionalmente a la perspectiva de vértices y edges de las propiedades del grafo, ***Graph* X** también tiene ***triple view***. La triple ***view*** joinea, lógicamente, las propiedades de los vértices y los *edges*. ***Graph X*** tiene un operador que puede ejecutar operaciones desde la librería *Pregel* para hacer analítica de grafos. Este operador ***Pregel*** ejecuta una serie de superpasos, que define un protocolo de mensajería para los vértices.

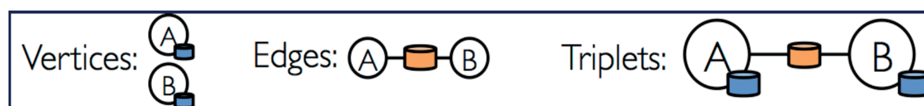

Figura 9. TripleView. Obtenida de: https://spark.apache.org/docs/latest/img/triplet.png

Para concluir, Spark puede ser usado también para procesamiento de grafos en paralelo mediante el uso e implementación de ***Graph X***. El mismo que permite almacenar información relacionada con vértices y ***edges*** mediante el uso de ***RDDs*** especiales. Adicionalmente, el operador ***Pregel*** trabaja en una serie de superpasos.

5.5. Explorar datos con Spark SQL y Spark DataFrames

Los objetivos de esta actividad son:

- Acceder a datos de archivos *.csv* con *Spark SQL.*
- Filtrar filas y columnas de los dataframe de *Spark.*
- Agrupar y aplicar funciones de agregación en columnas de un *DataFrame* de *Spark.*
- *Joinear* dos *dataframes* de Spark con base en una columna.

Paso 1: **Conectar** con los archivos.

Importamos el módulo de ***SparkSesion*** dado que es el *entry point* para empezar a trabajar con datos y dataframes.

```
from pyspark.sql import SparkSession
```

La segunda línea crea la sesión de *Spark*:

```
spark = SparkSession.builder.appName("SQL").getOrCreate()
```

Esta tercera línea sube el archivo *csv* ***gamecliks*** a *hdfs* desde un directorio local:

```
hdfs dfs -copyFromLocal game-clicks.csv /users/bigdata/game-clicks.csv
```

Esta línea de código crea un nuevo *DataFrame* de *Spark en la variable df* :

```
df = spark.read.csv("hdfs:/users/bigdata/game-clicks.csv", header = True, inferSchema=True)
```

Paso 2: Ver el **esquema** del **DataFrame** y la cantidad de filas. Podemos llamar al método printSchema() para ver el esquema del DataFrame:

*df.**printSchema()***

```
root
 |-- timestamp: timestamp (nullable = true)
 |-- clickId: integer (nullable = true)
 |-- userId: integer (nullable = true)
 |-- userSessionId: integer (nullable = true)
 |-- isHit: integer (nullable = true)
 |-- teamId: integer (nullable = true)
 |-- teamLevel: integer (nullable = true)
```

Lo que muestra este método es una lista con los nombres y los tipos de datos de cada una de las columnas del *dataframe*.

Llamamos al método count(), que cuenta la cantidad de filas del *dataframe*.

*df.**count()***

755806

Paso 3. Vemos el **contenido** del ***dataframe***. Podemos llamar al **método** show(). Para ver el contenido del *dataframe*. El argumento especifica la cantidad de filas a mostrar.

*df.**show**(5)*

```
+-------------------+-------+------+-------------+-----+------+---------+
|          timestamp|clickId|userId|userSessionId|isHit|teamId|teamLevel|
+-------------------+-------+------+-------------+-----+------+---------+
|2016-05-26 15:06:55|    105|  1038|         5916|    0|    25|        1|
|2016-05-26 15:07:09|    154|  1099|         5898|    0|    44|        1|
|2016-05-26 15:07:14|    229|   899|         5757|    0|    71|        1|
|2016-05-26 15:07:14|    322|  2197|         5854|    0|    99|        1|
|2016-05-26 15:07:20|     22|  1362|         5739|    0|    13|        1|
+-------------------+-------+------+-------------+-----+------+---------+
only showing top 5 rows
```

Paso 4. **Filtrar** columnas en un ***dataframe***. Podemos filtrar por una o más columnas llamando al método select().

*df.**select**("userid", "teamlevel").**show**(5)*

```
+------+---------+
|userid|teamlevel|
+------+---------+
|  1038|        1|
|  1099|        1|
|   899|        1|
|  2197|        1|
|  1362|        1|
+------+---------+
only showing top 5 rows
```

Paso 5. **Filtrar** filas con base en un **criterio**. También podemos filtrar en base a filas que coincidan con algún criterio específico usando el **método** filter()

```
df.filter(df['teamlevel']>1).select('userid', 'teamlevel').show(5)
```

```
+------+---------+
|userid|teamlevel|
+------+---------+
|  1513|        2|
|   868|        2|
|  1453|        2|
|  1282|        2|
|  1473|        2|
+------+---------+
only showing top 5 rows
```

Los argumentos del método filter() son una columna, en este caso especificado como *df['teamlevel']* y la condición, que en este caso es mayor a 1. El segundo método que se ejecuta es el método *select(),* que solo muestra las columnas *userid* y *teamlevel* y mediante el comando *show* mostramos únicamente los primeros cinco registros.

Paso 6. **Agrupar** por una **columna** y *count*. El método *groupBy*() agrupa los valores de columna(s). La columna *ishit* solo tiene dos valores 0 o 1. Podemos calcular la cantidad de ocurrencias agrupando la columna *ishit* y contando los resultados:

```
df.groupBy('ishit').count().show()
```

```
+-----+------+
|ishit| count|
+-----+------+
|    1| 83383|
|    0|672423|
+-----+------+
```

Paso 7. **Calcular** el **promedio** y la **suma**. Las funciones de agregación pueden ser ejecutadas sobre columnas de un *dataframe*. Primero, vamos a importa la librería que tiene las funciones de agregación. Luego vamos a calcular el promedio y el total de valores llamando a los métodos *mean*() y sum*()* respectivamente.

```
from pyspark.sql.functions import mean, sum
```

df.**select**(*mean*(‘*ishit*’) , *sum*(‘*ishit*’)).**show**()

```
+------------------+----------+
|        avg(ishit)|sum(ishit)|
+------------------+----------+
|0.1103232840173272|     83383|
+------------------+----------+
```

Paso 8. **Joinear** dos **dataframes**. Podemos realizar un *merge* o join de dos dataframes en base a una columna. Primero, vamos a crear el dataframe para los *adclicks* y vamos a guardar el resultado en la variable *df2*:

df2 = **spark**.**read**.**csv**(*“hdfs:/users/bigdata/ad-clicks.csv”, header = True, inferSchema=True)*

Vamos a ver las columnas del *df2* llamando al método *printSchema*()

df2.**printSchema**()

```
root
 |-- timestamp: timestamp (nullable = true)
 |-- txId: integer (nullable = true)
 |-- userSessionId: integer (nullable = true)
 |-- teamId: integer (nullable = true)
 |-- userId: integer (nullable = true)
 |-- adId: integer (nullable = true)
 |-- adCategory: string (nullable = true)
```

Podemos ver que el *df2* **también** tiene columna *userid*. Luego, vamos a combinar el dataframe *gameclicks* y el *addclicks* llamando al método *join*() y guardamos el resultado en una variable llamada merge:

merge = *df*.**join**(*df2*, ‘*userid*’)

En la línea de código anterior estamos llamando al método *join*() sobre el dataframe gameclicks; el primer argumento es el dataframe a joinear; en este caso addClicks, y el segundo argumento es el nombre de la columna que está presente en ambos dataframes.

Ahora veremos el **esquema** del dataframe resultante:

merge.**printSchema**()

Podemos ver que el dataframe resultante tiene todas las columnas tanto de *gameclicks* como de *adclicks*.

Finalmente, vemos el contenido del dataframe merge:

merge.**show(5)**

```
+------+-------------------+-------+-------------+-----+------+---------+-------------------+-----+-------------+------+----+-----------+
|userId|          timestamp|clickId|userSessionId|isHit|teamId|teamLevel|          timestamp| txId|userSessionId|teamId|adId| adCategory|
+------+-------------------+-------+-------------+-----+------+---------+-------------------+-----+-------------+------+----+-----------+
|  1362|2016-05-26 15:07:20|     22|         5739|    0|    13|        1|2016-06-16 10:21:01|39733|        34223|    13|   1|     sports|
|  1362|2016-05-26 15:07:20|     22|         5739|    0|    13|        1|2016-06-15 23:52:15|38854|        34223|    13|   3|electronics|
|  1362|2016-05-26 15:07:20|     22|         5739|    0|    13|        1|2016-06-15 12:23:31|37940|        34223|    13|  15|     sports|
|  1362|2016-05-26 15:07:20|     22|         5739|    0|    13|        1|2016-06-13 00:12:01|32627|        26427|    13|  14|    fashion|
|  1362|2016-05-26 15:07:20|     22|         5739|    0|    13|        1|2016-06-12 13:02:36|31729|        26427|    13|   4|      games|
+------+-------------------+-------+-------------+-----+------+---------+-------------------+-----+-------------+------+----+-----------+
only showing top 5 rows
```

Referencias citadas en la Clase 5

- Karau, H., & Kannan, R. (2017). *Learning Spark: Lightning-fast Big Data Analysis*. O'Reilly Media.
- Meng, X., Bradley, J., Da, S., & Zaharia, M. (2016). *MLlib: Machine Learning in Apache Spark*. Journal of Machine Learning Research, 17, 1-7.
- Xin, R., Zaharia, M., Franklin, M., & Shen, C. (2013). *GraphX: A resilient distributed graph system on Spark*. Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, 1223–1226.
- Zaharia, M., Chowdhury, M., Das, T., & Ma, J. (2016). *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*. 9th Usenix Symposium on Networked Systems Design and Implementation (NSDI '12).
- Apache Software Foundation. (n.d.). *Apache Spark*. Apache. https://spark.apache.org/
- Databricks. (s.f.). *Functions — PySpark master documentation*. Recuperado de https://api-docs.databricks.com/python/pyspark/latest/pyspark.sql/functions.htm

Definición de los términos citados en la Clase 5

Hive.

Apache Hive es un sistema de **almacenamiento** de datos y procesamiento **distribuido,** diseñado para trabajar sobre **Hadoop;** permite consultas estructuradas mediante un lenguaje similar a **SQL** llamado **HiveQL**. Facilita el análisis de grandes volúmenes de datos almacenados en **HDFS** al abstraer la complejidad de **MapReduce**, proporcionando una interfaz familiar para usuarios que trabajan con bases de datos relacionales. **Hive** es ideal para consultas analíticas en big data; aunque no está optimizado para transacciones en tiempo real, debido a su latencia en la ejecución de consultas.

Sliding Window.

Sliding Window (ventana deslizante) es una técnica utilizada en **informática** y **análisis** de datos para procesar flujos continuos de información, mediante una ventana de tamaño fijo que se desplaza sobre una **secuencia de datos**. Se usa, comúnmente, en algoritmos de **optimización**, procesamiento de señales y análisis de series temporales; permite calcular métricas sobre subconjuntos de datos sin necesidad de recomputar desde cero en cada paso. En redes, se emplea en protocolos como **TCP,** para controlar el flujo de datos asegurando una transmisión eficiente y evitando congestiones.

Profundización Clase 5

La **biblioteca** Spark **Mlib** tiene algunos algoritmos de *machine learning*, entre los más importantes **po-**

1. Clasificación y Regresión

- **Regresión logística**
- **Máquinas de soporte vectorial (SVMs)**
- **Árboles de decisión**
- **Random Forest**
- **Gradient-Boosted Trees (GBT)**
- **Regresión lineal y polinómica**

2. Clustering

- **K-Means**
- **Latent Dirichlet Allocation (LDA)**
- **Gaussian Mixture Model (GMM)**
- **Bisecting K-Means**

3. Reducción de Dimensionalidad

- **PCA (Análisis de Componentes Principales)**
- **SVD (Descomposición en Valores Singulares)**

demos citar:

Fuente: Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2016). Apache Spark. Recuperado de https://en.wikipedia.org/wiki/Apache_Spark

La librería ***pyspark.sql.functions*** tiene múltiples **funciones**, entre las más importantes podemos **citar**:

1. Funciones Matemáticas

- `abs(col)` : Devuelve el valor absoluto de la columna especificada.
- `sqrt(col)` : Calcula la raíz cuadrada del valor en la columna dada.
- `exp(col)` : Calcula el exponencial del valor en la columna.
- `log(col)` : Devuelve el logaritmo natural del valor de la columna.
- `round(col, scale)` : Redondea el valor de la columna al número de decimales especificado por `scale` .

2. Funciones de Cadenas

- `concat(*cols)` : Concatena múltiples columnas en una sola.
- `substring(col, pos, len)` : Extrae una subcadena de la columna, comenzando en la posición `pos` y con una longitud de `len` .
- `upper(col)` : Convierte todos los caracteres de la columna a mayúsculas.
- `lower(col)` : Convierte todos los caracteres de la columna a minúsculas.
- `trim(col)` : Elimina los espacios en blanco al inicio y al final de la cadena en la columna.

3. Funciones de Fecha y Hora

- `current_date()` : Devuelve la fecha actual.
- `current_timestamp()` : Devuelve la marca de tiempo actual.
- `datediff(end, start)` : Calcula la diferencia en días entre dos fechas.
- `date_add(start, days)` : Suma un número específico de días a una fecha.
- `date_sub(start, days)` : Resta un número específico de días a una fecha.

Fuente: Apache Spark. (s.f.). Spark SQL, Built-in Functions. Recuperado de https://spark.apache.org/docs/latest/api/sql/index.html

La excelencia no se improvisa

síguenos

CV_ICYS_M