PUCE Virtual

Sistemas de Big Data

Procesamiento de datos a gran escala

Clase 3

MAESTRÍA EN
SISTEMAS DE INFORMACIÓN
Mención Data Science

La excelencia no se improvisa

1. INTRODUCCIÓN DE LA CLASE

RDA 1: Utilizar tecnologías y herramientas de Big Data para grandes volúmenes de datos.

El **crecimiento exponencial** de los datos ha impulsado el desarrollo de tecnologías capaces de procesar información a gran escala de manera eficiente. **MapReduce** es un paradigma de programación **distribuida** propuesto por **Google,** que permite el procesamiento de **grandes volúmenes** de datos a través de operaciones de mapeo y reducción, dividiendo tareas en múltiples **nodos** de **cómputo** y combinando resultados de forma paralela (Dean & Ghemawat, 2008). Para almacenar estos datos **distribuidos** de manera eficiente, **Hadoop Distributed File System** (HDFS) proporciona un sistema de archivos escalable y tolerante a fallos, optimizado para el almacenamiento de grandes conjuntos de datos en clústeres de hardware convencional (Shvachko et al., 2010). La combinación de MapReduce y HDFS ha permitido la gestión eficiente de datos masivos, estableciendo las bases del procesamiento distribuido moderno.

Por otro lado, **Apache Spark** ha revolucionado el procesamiento de datos al introducir un modelo basado en memoria, que mejora **significativamente** el rendimiento en comparación con **Hadoop MapReduce**. Spark permite la construcción de ***pipelines*** de datos mediante el uso de **Resilient Distributed Datasets** (RDDs) y **dataframes**, facilitando tanto el **procesamiento batch** como en tiempo real (Zaharia et al., 2016). Su compatibilidad con diversas fuentes de datos y su integración con herramientas como **MLlib,** para aprendizaje automático, lo han convertido en una solución poderosa para la analítica de datos a gran escala. La implementación de *pipelines* de datos con **Spark** optimiza los flujos de **procesamiento** al reducir la latencia y mejorar la eficiencia computacional en entornos empresariales y científicos.

Clase 3. Procesamiento de datos a gran escala

3.1. Copiar datos a HDFS

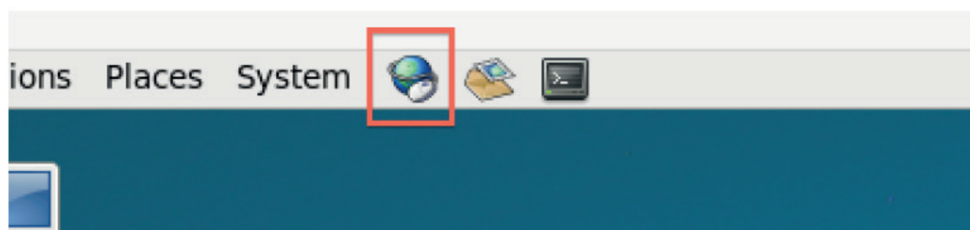
En la siguiente sección se muestra una **guía** y la explicación de los pasos que se deben seguir para **mover archivos** desde el sistema operativo (en este caso es ***Linux***, usando la máquina virtual descargada de Cloudera) hacia **Hadoop Distributed File System.**

Los **objetivos** de esta actividad son los siguientes:

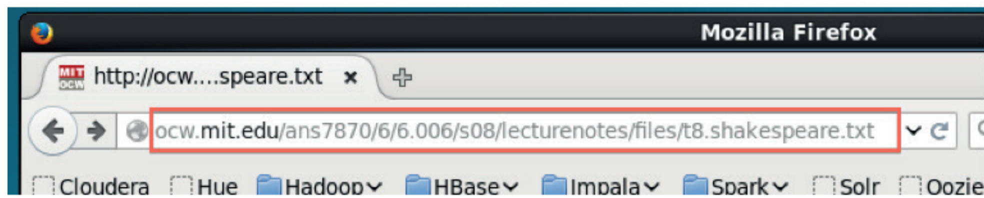
- Interactuar con Hadoop usando **línea de comandos.**
- Copiar archivos desde y hacia **Hadoop Distributed File System.**

Instrucciones:

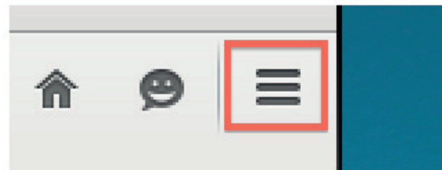
1. **Abrir el *browser*** haciendo clic en el í**cono** de *browser*, en la parte superior **izquierda** de la pantalla.



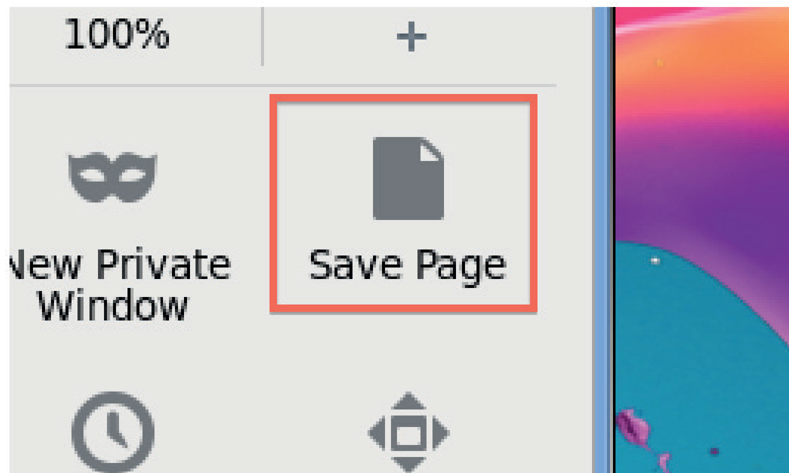
2. Descargar el archivo Shakespeare. Ahora descargaremos un archivo para posteriormente copiarlo a HDFS. Escribe el siguiente link en el *browser*. [*https://ocw.mit.edu/ans7870/6/6.006/s08/lecturenotes/files/t8.shakespeare.txt*](https://ocw.mit.edu/ans7870/6/6.006/s08/lecturenotes/files/t8.shakespeare.txt)



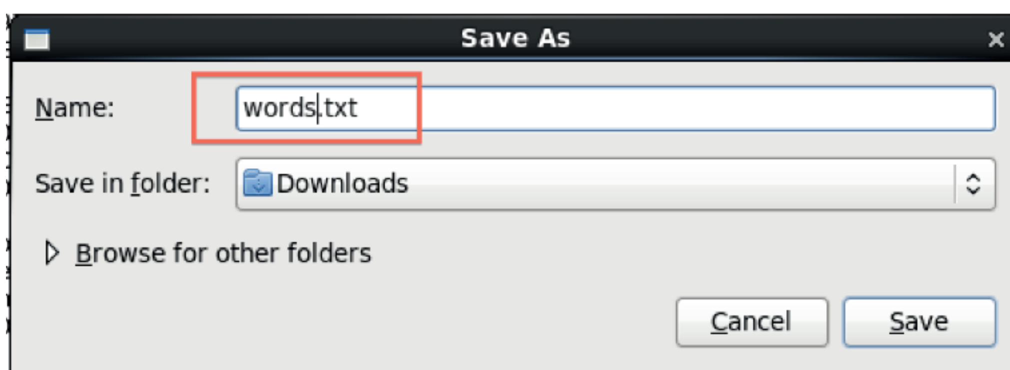
Una vez que la página este cargada, dale clic en el botón *open.*



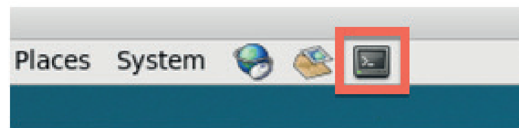
Dale clic en guardar página.



Cambia el nombre del archivo a ***words.txt*** y dale clic en guardar.



Abre una terminal de **línea de comandos.**



Ejecuta *cd Downloads* para moverte al directorio *Downloads*

```
[cloudera@quickstart ~]$ cd Downloads/
[cloudera@quickstart Downloads]$ 
```

Ejecuta *ls* para verificar que el archivo *words.txt* se haya grabado

```
[cloudera@quickstart Downloads]$ ls
words.txt
```

3. Copia el archivo a HDFS. Ejecuta *hadoop fs –copyFromLocal words.txt* para copiar el texto a HDFS.

```
[cloudera@quickstart Downloads]$ hadoop fs -copyFromLocal words.txt
[cloudera@quickstart Downloads]$ 
```

4. Verifica que el archivo se haya copiado a **HDFS.** Ejecuta *hadoop fs –ls* para verificar que el archivo se haya copiado a HDFS.

```
[cloudera@quickstart Downloads]$ hadoop fs -ls
Found 1 items
-rw-r--r--   1 cloudera cloudera    5458199 2016-02-12 15:14 words.txt
[cloudera@quickstart Downloads]$ 
```

5. Copia un archivo dentro de **HDFS.** Puedes hacer una copia de un archivo en HDFS. **Ejecuta** *hadoop fs -cp words.txt words2.txt* para hacer una copia de *words.txt* llamada *words2.txt*

```
[cloudera@quickstart Downloads]$ hadoop fs -cp words.txt words2.txt
[cloudera@quickstart Downloads]$ 
```

Puedes ver el nuevo archivo ejecutando el siguiente comando: *hadoop fs -ls*

```
[cloudera@quickstart Downloads]$ hadoop fs -ls
Found 2 items
-rw-r--r--   1 cloudera cloudera    5458199 2016-02-12 15:14 words.txt
-rw-r--r--   1 cloudera cloudera    5458199 2016-02-12 15:15 words2.txt
[cloudera@quickstart Downloads]$
```

6. Copia un archivo desde HDFS. También podemos copiar n archivo desde HDFS al sistema de archivos local, Ejecuta *hadoop fs -copyToLocal words2.txt.* Para copiar *words2.txt* al directorio local.

```
[cloudera@quickstart Downloads]$ hadoop fs -copyToLocal words2.txt
[cloudera@quickstart Downloads]$
```

Ahora ejecutemos *ls* para ver que el archivo fue copiado y verificar que se encuentre en esa ubicación.

```
[cloudera@quickstart Downloads]$ ls
words2.txt  words.txt
[cloudera@quickstart Downloads]$
```

7. Eliminar el archivo de HDFS. Vamos a eliminar el archivo *words2.txt* en HDFS. Ejecutando el comando *hadoop fs -rm words2.txt*

```
[cloudera@quickstart Downloads]$ hadoop fs -rm words2.txt
16/02/12 15:17:01 INFO fs.TrashPolicyDefault: Namenode trash configuration: Dele
tion interval = 0 minutes, Emptier interval = 0 minutes.
Deleted words2.txt
[cloudera@quickstart Downloads]$
```

8. Ejecuta *hadoop fs -ls* para verificar que el archivo ya no esté.

3.2. Ejecutar el programa WordCount (MapReduce)

En esta sección se proporciona una guía y explicación para ejecutar el programa **WordCount** usado **MapReduce** en la máquina virtual de Cloudera.

Los **objetivos** de esta actividad son los siguientes:

- Ejecutar la aplicación de **WordCount.**
- Copiar los resultados de **WordCount** a **HDFS.**

1. Abrir la terminal de comandos.

2. Mira los programas de ejemplo de MapReduce. Hadoop tiene aplicaciones preinstaladas de ejemplo de **MapReduce**. Puedes ver una lista ejecutando el siguiente comando: **hadoop jar /usr/jars/hadoop-examples.jar.** Nosotros estamos interesados en correr *WordCount.*

```
[cloudera@quickstart ~]$ hadoop jar /usr/jars/hadoop-examples.jar
An example program must be given as the first argument.
Valid program names are:
  aggregatewordcount: An Aggregate based map/reduce program that counts the words in the in
put files.
  aggregatewordhist: An Aggregate based map/reduce program that computes the histogram of t
he words in the input files.
  bbp: A map/reduce program that uses Bailey-Borwein-Plouffe to compute exact digits of Pi.
  dbcount: An example job that count the pageview counts from a database.
  distbbp: A map/reduce program that uses a BBP-type formula to compute exact bits of Pi.
  grep: A map/reduce program that counts the matches of a regex in the input.
  join: A job that effects a join over sorted, equally partitioned datasets
  multifilewc: A job that counts words from several files.
  pentomino: A map/reduce tile laying program to find solutions to pentomino problems.
  pi: A map/reduce program that estimates Pi using a quasi-Monte Carlo method.
  randomtextwriter: A map/reduce program that writes 10GB of random textual data per node.
  randomwriter: A map/reduce program that writes 10GB of random data per node.
  secondarysort: An example defining a secondary sort to the reduce.
  sort: A map/reduce program that sorts the data written by the random writer.
  sudoku: A sudoku solver.
  teragen: Generate data for the terasort
  terasort: Run the terasort
  teravalidate: Checking results of terasort
  wordcount: A map/reduce program that counts the words in the input files.
  wordmean: A map/reduce program that counts the average length of the words in the input f
iles.
  wordmedian: A map/reduce program that counts the median length of the words in the input
files.
```

La salida del programa indica que **WordCount** toma el nombre de uno o más archivos de entrada el nombre del directorio de salida. Nota: Esos archivos se encuentran en **HDFS** y no en el sistema local de **archivos**.

3. Verificar la existencia de los archivos de entrada. En la sección anterior descargamos los trabajos de **Shakespeare** y los copiamos en **HDFS**, verifiquemos que este archivo siga en **HDFS** par que nosotros podamos ejecutar **WordCount** sobre este archivo. Ejecuta el comando ***hadoop fs –ls***

```
[cloudera@quickstart Downloads]$ hadoop fs -ls
Found 1 items
-rw-r--r--   1 cloudera cloudera    5458199 2016-02-12 15:14 words.txt
[cloudera@quickstart Downloads]$ 
```

4. Ver los argumentos de línea de comandos de **WordCount**. Podemos aprender como correr **WordCount** examinando los argumentos del comando. Ejecuta ***hadoop jar /usr/jars/hadoop-examples.jar wordcount.***

```
[cloudera@quickstart ~]$ hadoop jar /usr/jars/hadoop-examples.jar wordcount
Usage: wordcount <in> [<in>...] <out>
```

5. Ejecuta **WordCount.** Ejecuta WordCount para el archivo *words.txt. hadoop jar /usr/jars/hadoop-examples.jar wordcount words.txt out.*

```
[cloudera@quickstart Downloads]$ hadoop jar /usr/jars/hadoop-examples.jar wordcount words.txt out
16/02/12 15:27:34 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.0:8032
16/02/12 15:27:35 INFO input.FileInputFormat: Total input paths to process : 1
16/02/12 15:27:35 INFO mapreduce.JobSubmitter: number of splits:1
```

Mientras la aplicación de WordCount se ejecuta, ***Hadoop*,** va imprimiendo el progreso en términos de Map y Reduce. Cuando el programa WordCount termina ambos procesos van a indicar el 100% de completados.

```
16/02/12 15:27:46 INFO mapreduce.Job:  map 0% reduce 0%
16/02/12 15:27:54 INFO mapreduce.Job:  map 100% reduce 0%
16/02/12 15:28:02 INFO mapreduce.Job:  map 100% reduce 100%
16/02/12 15:28:02 INFO mapreduce.Job: Job job_1455318527581_0001 completed successfully
```

6. Revisar el directorio de salida de **WordCount**. Una vez que **WordCount** a terminado, verificaremos que salida se obtuvo. Primero veamos el directorio de salida, la carpeta ***out*** fue creada en **HDFS** mediante la ejecución del comando ***hadoop fs –ls***

```
[cloudera@quickstart Downloads]$ hadoop fs -ls
Found 2 items
drwxr-xr-x   - cloudera cloudera          0 2016-02-12 15:28 out
-rw-r--r--   1 cloudera cloudera    5458199 2016-02-12 15:14 words.txt
[cloudera@quickstart Downloads]$ _
```

Podemos ver ahora que hay dos ítems en HDFS: *words.txt* que es el archivo que se creó previamente y la carpeta *out* es la carpeta creada por WordCount.

7. Inspeccionar dentro del directorio de salida. El directorio creado por **WordCount** contiene varios archivos. Mira dentro del directorio ejecutando el siguiente comando ***hadoop –fs ls out***

```
[cloudera@quickstart Downloads]$ hadoop fs -ls out
Found 2 items
-rw-r--r--   1 cloudera cloudera          0 2016-02-12 15:28 out/_SUCCESS
-rw-r--r--   1 cloudera cloudera     717768 2016-02-12 15:28 out/part-r-00000
[cloudera@quickstart Downloads]$ _
```

El archivo *part-r-00000* contiene el resultado de **WordCount**. El archivo ***_SUCESSS*** significa que **WordCount** se ejecutó satisfactoriamente.

8. Copiar los resultados de **WordCount** al sistema local de archivos. Copia *part-r-00000* al sistema local de archivos ejecutando *hadoop fs –copyToLocal out/part-r-00000 local.txt*

```
[cloudera@quickstart Downloads]$ hadoop fs -copyToLocal out/part-r-00000 local.txt
[cloudera@quickstart Downloads]$
```

Cada línea del archivo de resultados muestra el número de **ocurrencias** de cada palabra en el archivo de entrada. Por ejemplo, **Accuse** aparece cuatro veces en la entrada, pero **Accussing** aparece solo una vez.

```
Accost-  1
Account  1
Accountant      1
Accounted       1
Accoutred       1
Accurs'd        2
Accurs'd,       1
Accursed        4
Accusativo,     2
Accuse   4
Accusing        1
Acheron  2
Acheron,        1
Aches    1
```

3.3. *Pipelines* para procesamiento de Big Data

DataFlow. Consideremos el programa de MapReduce **WordCount,** que lee uno o más archivos de texto y cuenta la **cantidad** de cada palabra en estos archivos. En este ejemplo, el primer paso es dividir a los archivos en los diferentes nodos del clúster de **HDFS** en **particiones** del mismo archivo o múltiples archivos.

Luego, se ejecuta una operación de tipo **Map**, en este caso una **función definida por el usuario** para contar **palabras** se ejecuta en cada uno de esos nodos. Posteriormente, todos los pares ***key-values*** que fueron la **salida** de la operación **Map** son ordenamos con base en el valor de *key* y los pares *key-values* con la misma palabra son movidos o dirigidos al mismo nodo.

Finalmente, la operación de **Reduce** es ejecutada en estos nodos para sumar los valores con el mismo par ***key-value*** con el mismo ***key.*** La siguiente figura resume estos pasos:

Figura N.º 1. Big Data Processing Pipeline.

Esta secuencia de pasos se puede aplicar para lograr **escalabilidad en paralelización** de datos para diversas aplicaciones. En general, nos referimos a este patrón como **'*split-to-merge*'.** En estas aplicaciones, los datos ***flows*** de cada una de las etapas pasan por diferentes **transformaciones,** con diferentes necesidades de **escalalibidad**, hasta llegar a un producto final.

Primero que nada, los datos son particionados para luego pasar por diferentes funciones definidas por el usuario para hacer algo, desde operaciones estadísticas hasta *joineos* de datos y funciones de ***machine learning.*** Al final, los resultados pueden ser combinados usando algoritmos de ***merging*** u otras operaciones. La secuencia de estos pasos se conoce como ***Big Data Pipelines***.

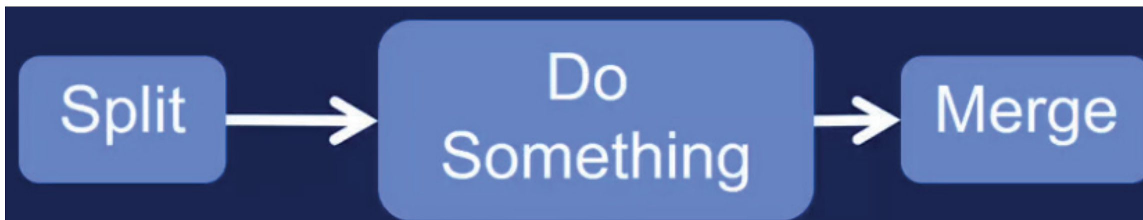


Figura N.º 2. Big Data Pipelines.

En el contexto del procesamiento de **Big Data,** el mecanismo de paralelización de cada *step* en el *pipeline* es principalmente **data parallelism.** El **data parallelism** se puede definir simplemente como ejecutar las mismas funciones, de forma simultánea, para los elementos o particiones de un **dataset** en **múltiples cores.** En el ejemplo del **WordCount** se puede evidenciar cada una de las etapas y sus mecanismos de paralelización de datos como se indica en la siguiente figura.

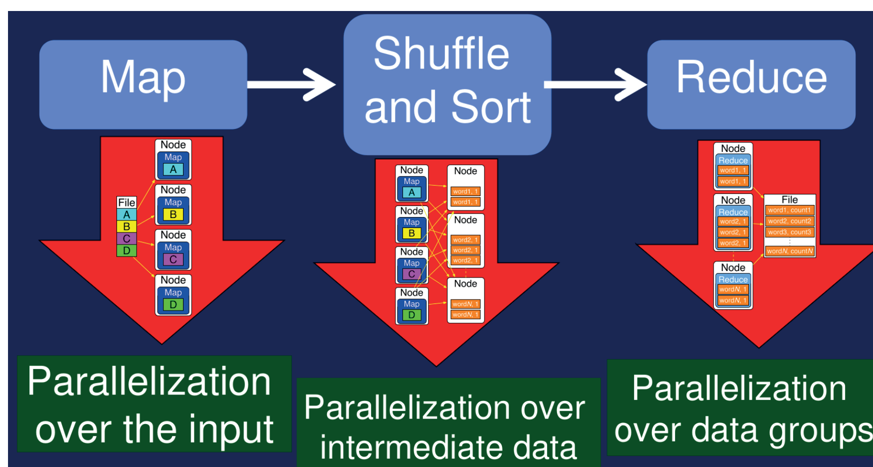


Figura N.º 3. *Data Parallelism.*

Transformaciones en los Data Pipelines

En una definición **simple**, las transformaciones de datos son **funciones** o **herramientas** que se usan para convertir datos de una forma A a una forma B. Como por ejemplo, transformar madera en muebles. **Map,** seguramente, es una de las transformaciones más comunes que podemos encontrar en el mundo del **Big Data.**

La **transformación Map** aplica la misma operación a cada miembro de una **colección de datos.** La operación **Reduce** recolecta las **ítems** que tienen el mismo **key, por ejemplo para el caso WordCount** el **key** son las palabras; por lo tanto, suma la frecuencia de cada una de las palabras. Map y Reduce son un tipo de transformaciones que trabajan en una lista de **keys** y pares de datos.

Figura N.º *4. Data Transformation.*

Otra transformación muy usada es el producto cartesiano (Cross), que es en esencia una multiplicación entre dos conjuntos de datos **key-value,** sin importar su **key**. En otras palabras, consiste en realizar algún proceso a cada par de los dos conjuntos.

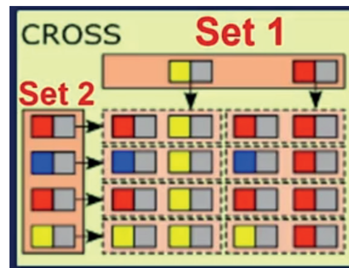


Figura N.º 5. Cross/Cartesiano.

La transformación **Match/Join** es en esencia una **multiplicación selectiva**. Al igual que en el caso anterior la idea es realizar algún proceso a cada par de datos de los dos datasets **que tengan la misma key.**

Otra operación es **Co-Group**, que consiste en agrupar ítems en común. En otras palabras, primero colectar cosas similares y luego aplicar un proceso a cada colección.

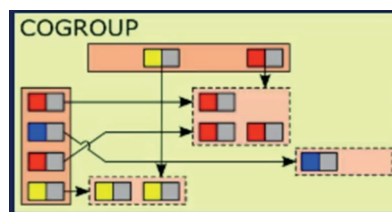


Figura N.º 6. CoGroup.

Finalmente, está la función **filter,** que consiste en aplicar una condición para quedarnos únicamente con los elementos que cumplen con ese criterio. Por ejemplo, si tienes número del 1 al 8 un filtro podría ser quedarnos con los **múltiplos de 2.**

Agregaciones

Las **agregaciones** son la aplicación de una **transformación** sobre todos los elementos de un **dataset** de datos **específico.** Uno de los ejemplos más simples que hay respecto de las agregaciones es la suma. En la figura siguiente se aplica la **transformación** suma sobre todo el **dataset** se estrellas.

Figura N.º 7. Agregaciones. Sumatoria.

Otra agregación que se puede realizar es sumar con base en algún criterio. En este caso, podría ser la **suma agrupada** por color de **estrellas;** así, tendríamos 3 amarillas, 5 verdes y 6 rosadas. También podemos aplicar otras transformaciones como *average, max, min, standard desviation.*

3.4. Introducción a Spark

Apache Spark es un ***framework*** de computación distribuida diseñado para el procesamiento eficiente de grandes volúmenes de datos. Su arquitectura permite el procesamiento en memoria, lo que mejora significativamente el rendimiento en comparación con sistemas basados en disco, como **Hadoop MapReduce** (Zaharia et al., 2016). **Spark** proporciona múltiples **API** para lenguajes como Python, Scala, Java y R, lo que facilita su adopción en diversos entornos de desarrollo y análisis de datos (Armbrust et al., 2015). Además, su compatibilidad con herramientas como Apache Hadoop, **Apache Hive y Apache Kafka** lo convierte en una solución versátil para aplicaciones de Big Data y aprendizaje automático (Karau et al., 2017).

Antes de entrar en detalle respecto de las capacidades de **Apache Spark,** es importante hablar de algunas de las deficiencias de **Hadoop MapReduce.**

- Solo funciona para operaciones de tipo Map y Reduce.
- Depende de leer datos de **HDFS.**
- Soporte nativo para el lenguaje **Java** únicamente.
- No tiene soporte para procesamiento en ***streaming.***

Hadoop Spark usa un modelo de programación expresivo, que permite hacer muchas cosas con **pocas líneas de código**, el procesamiento lo hace en **memoria** y no en disco; eso aumenta significativamente la velocidad de ejecución de las aplicaciones. Adicionalmente, **Spark** soporta diversidad en ***workloads*** (*batch* o *streming*). Spark brinda ***API*** simples para Python, Scala, Java y SQL, con interfaces interactivas y permite usar librerías tanto nativas como internas.

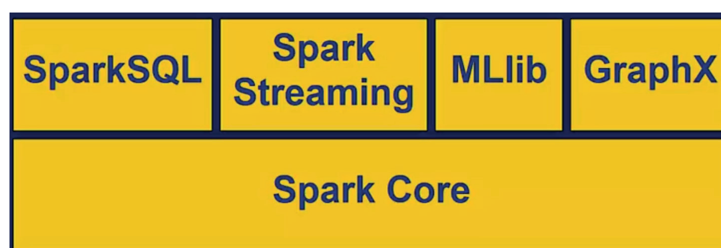


Figura N.º 8. *The Spark Stack.*

El **Spark Stack** es un conjunto de componentes que se comunican e interactúan con el **Spark Core** para realizar diferentes actividades.

Spark Core es en donde las capacidades principales del *framework* de *Spark* son implementadas, esto incluye soporte para calendarizaciones **distribuidas**, manejo de **memoria** y **tolerancia a fallos**. Otra parte importante del ***Spark Core*** es la **API** que permite definir los **RDD** (**Resilient Distributed Datasets**) los **RDD** son la principal abstracción de programación en ***Spark,*** que se encarga de llevar la data a través de múltiples nodos conectados en paralelo y transformarlos.

Spark SQL es el componente de *Spark* que posibilita consultar datos estructurados y no estructurados usando SQL; además, se puede conectar con múltiples fuentes de datos y brinda **API** que permiten convertir los resultados en los querys en ***RDD,*** en programas de Python, Scala, Java.

Spark Streaming es en donde la manipulación de datos ocurre dentro del *framework* de *Spark;* adicionalmente, habilita la creación de pequeñas agregaciones de datos que viene de sistemas de ingesta de datos en ***streaming***, Estos *datasets* agregados se llaman **microbatches** y pueden ser convertidos en **RDD** en ***Spark Streaming,*** para **procesarlos**.

MLlib es una librería nativa de **Spark** para algoritmos de *machine learning* y evaluación de modelos. **GraphX** es la librería para *Graph Analytics* de *Spark* y habilita la conversión de los vértices de modelos de datos basados en grafos a RDD. Adicionalmente, provee implementaciones escalables para algoritmos de procesamiento de grafos.

Documentación de Spark [https://spark.apache.org/downloads.html]

3.5. Conceptos y arquitectura de Spark

Spark usa los RDD para procesar los datos en memoria. **En algunos casos, operaciones en memoria pueden ser 100 000 veces más rápidas que operaciones en disco**. Spark toma los resultados de las transformaciones de cada una de las etapas del *pipeline* en memoria usando los RDD. La siguiente ilustración explica este procedimiento:

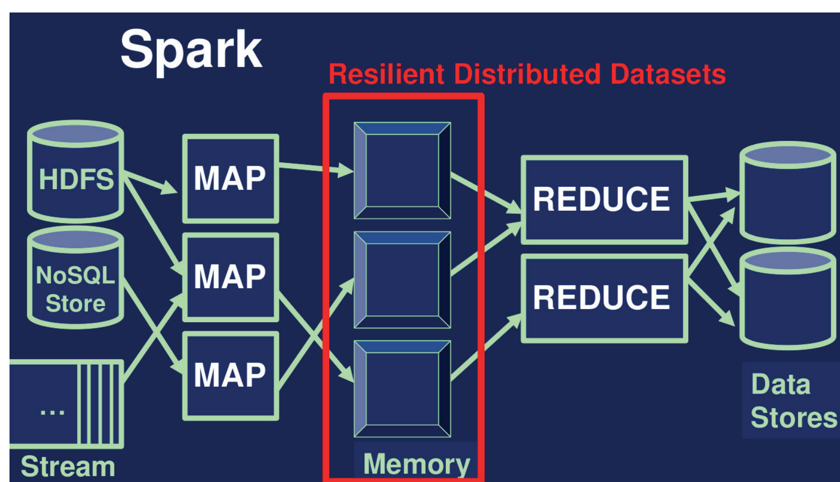


Figura N.º 9. *Spark RDD.*

Resilient Distributed Datasets:

Los ***datasets*** que crean los *RDD* pueden venir de procesos *batch* almancenados, por ejemplo en HDFS, bases de datos *no SQL*, archivos de texto, archivos *json*, etc. Cuando *Spark* lee datos desde estas fuentes, genera *RDD*. Operaciones de *Spark* pueden crear *RDD* a partir de otros *b*. Es importante comentar que los *RDD* son inmutables; es decir, no se los puede cambiar una vez que son creados. Sin embargo, se pueden crear nuevos *RDD* mediante una o varias transformaciones.

El término ***distributed*** se refiere a distribuir los datos a través de las máquinas de los clúster. La par-

tición de datos puede cambiar dinámicamente para optimizar el performance de Spark.

El último término es ***resilient;*** se refiere a que *Spark* tiene un **tranqueo** de la historia de cada partición, manteniendo una línea de tiempo sobre todo el ciclo de vida del **RDD**. Por lo tanto, **Spark** sabe dónde está la partición de datos en cada punto de cálculo. Mediante este conocimiento, **Spark** puede manejar la tolerancia a fallos y no **perder** información cuando algún **nodo falla**.

Arquitectura Spark

La arquitectura de Apache Spark se compone por medio de conceptos como **Spark Stack** y **Spark Core**, dentro del que se encuentran los *frameworks* de *Spark Shell*, RDD (*Resilient Distributed Datasets* o Conjuntos distribuidos y flexibles de datos) o Core API (Interfaz de aplicaciones).

SparkContext: independientemente del *backend* que use arquitectura Spark, **su coordinación se realiza con el *SparkContext.***

Cluster Manager: esta es la comunicación del *driver* con el *backend,* para **adquirir recursos físicos y poder ejecutar los *executors*.**

Driver: este es el proceso principal, **controla toda la aplicación y ejecuta *Spark Context*.**

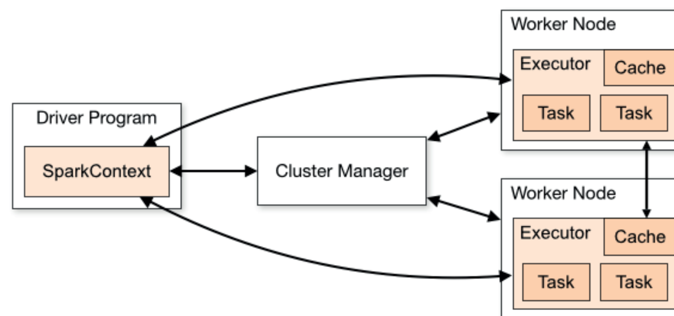


Figura N.º 10. Arquitectura de Apache Spark

Worker Node: hace referencia a las máquinas que dependen del *backend* y que se encargan de **ejecutar los procesos de los *executors*.**

Executors: proceso en el que se realiza la carga de trabajo, de manera que se obtienen las tareas desde el *driver* para **cargar, transformar y almacenar los datos.**

Arquitectura Spark vs. MapReduce

Característica	*Apache Spark*	*MapReduce*
Modelo de programación	*Funcional, basado en RDD (Resilient Distributed Datasets) y DataFrames.*	*Basado en un modelo de pasos Map y Reduce.*
Velocidad	*Mucho más rápido debido al uso de memoria RAM (in-memory).*	*Más lento, ya que realiza operaciones en disco.*
Manejo de estado intermedio	*Mantiene los datos en memoria durante el procesamiento (por defecto).*	*Los datos se escriben y leen del disco entre cada fase (Map y Reduce).*

Facilidad de uso	*Más fácil de usar, API de alto nivel (DataFrames, SQL, MLlib).*	*Menos intuitivo, requiere más código para operaciones complejas.*
Lenguajes soportados	*Scala, Python, Java, R, SQL.*	*Java (principalmente, aunque hay bibliotecas para otros lenguajes).*

Si el rendimiento y la flexibilidad son factores claves para tu caso de uso, **Apache Spark** suele ser la opción preferida. Sin embargo, si estás trabajando en un entorno donde ya se utilizan tecnologías de Hadoop y solo necesitas procesamiento *batch* básico, **MapReduce** puede seguir siendo una opción viable. Para *pipelines* de *machine learning* y *deep learning* es mucho más usado **Apache Spark** que **MapReduce.**

Referencias citadas en la Clase 3

- Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J., ... & Zaharia, M. (2015). *Spark SQL: Relational data processing in Spark*. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1383-1394.
- Dean, J., & Ghemawat, S. (2008). *MapReduce: Simplified Data Processing on Large Clusters*. *Communications of the ACM, 51*(1), pp. 107-113. https://doi.org/10.1145/1327452.1327492
- Karau, H., Konwinski, A., Wendell, P., & Zaharia, M. (2017). *Learning Spark: Lightning-fast big data analysis*. O'Reilly Media, Inc.
- Navarro, S. (17 de mayo de 2024). *¿Cómo es la arquitectura de Apache Spark?* KeepCoding. Recuperado de https://keepcoding.io/blog/arquitectura-apache-spark/
- Shvachko, K., Kuang, H., Radia, S., & Chansler, R. (2010). The Hadoop Distributed File System. *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1-10. https://doi.org/10.1109/MSST.2010.5496972
- Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2016). *Apache Spark: A Unified Engine for Big Data Processing*. *Communications of the ACM, 59*(11), pp. 56-65. https://doi.org/10.1145/2934664

Definición de los términos citados en la Clase 3

RDD:

Resilient Distributed Datasets (RDD). Estructura de datos fundamental de Apache Spark, diseñados para el procesamiento distribuido eficiente y tolerante a fallos. Un RDD es una colección inmutable de objetos que se pueden distribuir en múltiples nodos de un clúster y procesar en paralelo, permitiendo operaciones como transformaciones (**map**, **filter**) y acciones (**count**, **collect**). Su resiliencia proviene de la capacidad de recomputar automáticamente los datos en caso de fallos utilizando su *lineage*, en lugar de depender de replicaciones como en **Hadoop** (Zaharia et al., 2012). Los **RDD** ofrecen un equilibrio entre flexibilidad y eficiencia, facilitando el procesamiento de grandes volúmenes de datos en memoria o desde almacenamiento distribuido.

Pipeline. Conjunto de procesos interconectados que permiten la extracción, transformación y carga (**ETL**) de datos desde múltiples fuentes hasta su destino final, como un almacén de datos o un modelo de análisis. Su propósito es automatizar el flujo de datos de manera eficiente, asegurando su limpieza, integración y disponibilidad para su análisis o uso en aplicaciones de **inteligencia artificial** y **aprendizaje automático**. Los *pipelines* pueden operar en tiempo real o en lotes y suelen incluir herramientas como **Apache Spark, Apache Airflow o AWS Data Pipeline** para su orquestación y ejecución.

Profundización Clase 3

Resumen de la comparación entre **Apache Spark** y **MapReduce**

Apache Spark es más rápido, más fácil de usar y tiene la capacidad de procesar tanto datos *batch* como en tiempo real. También proporciona integraciones avanzadas para machine learning, procesamiento SQL y análisis de datos más complejos.

MapReduce, por otro lado, es más lento, ya que depende de la lectura y escritura en disco entre cada fase de procesamiento, pero sigue siendo una tecnología muy robusta y ampliamente utilizada, especialmente cuando se trata de trabajos *batch*.

La excelencia no se improvisa

síguenos