-Resumen de los Videos de Spark: <https://www.youtube.com/watch?v=_oVFCVg-b-s&list=PLeo_qKwGPZYfBG3cN79gwBHiucKuVbPQw>

**Índice:**

**1-Map-Reduce.**

**2-Apache Spark.**

**-Del 3 al 9 verlos directamente del archivo de Spark:**

**3-Lectura de Datos.**

**4-Acciones.**

**5-Transformaciones.**

**6-Wordcount.**

**7-Joins.**

**8-Transformaciones sobre particiones.**

**9-Persistencia y cache.**

**1-Map-Reduce.**

-Vamos a ver **cómo hacer procesamiento distribuido de datos utilizando Map-Reduce**.

-Cuál es el problema que intentamos resolver?: Cuando tenemos pocos datos/datos acotados y necesitamos procesarlos... tenemos diversas herramientas disponibles para realizar esta tarea:

* Utilizar la librería de Python pandas y procesar los datos con esto, como vimos anteriormente.
* También podríamos usar una BD relacional (como Mysql o MariaDb) y hacer nuestra consulta con sql.
* O también podríamos cargar los datos en un Excel y procesarlos ahí.

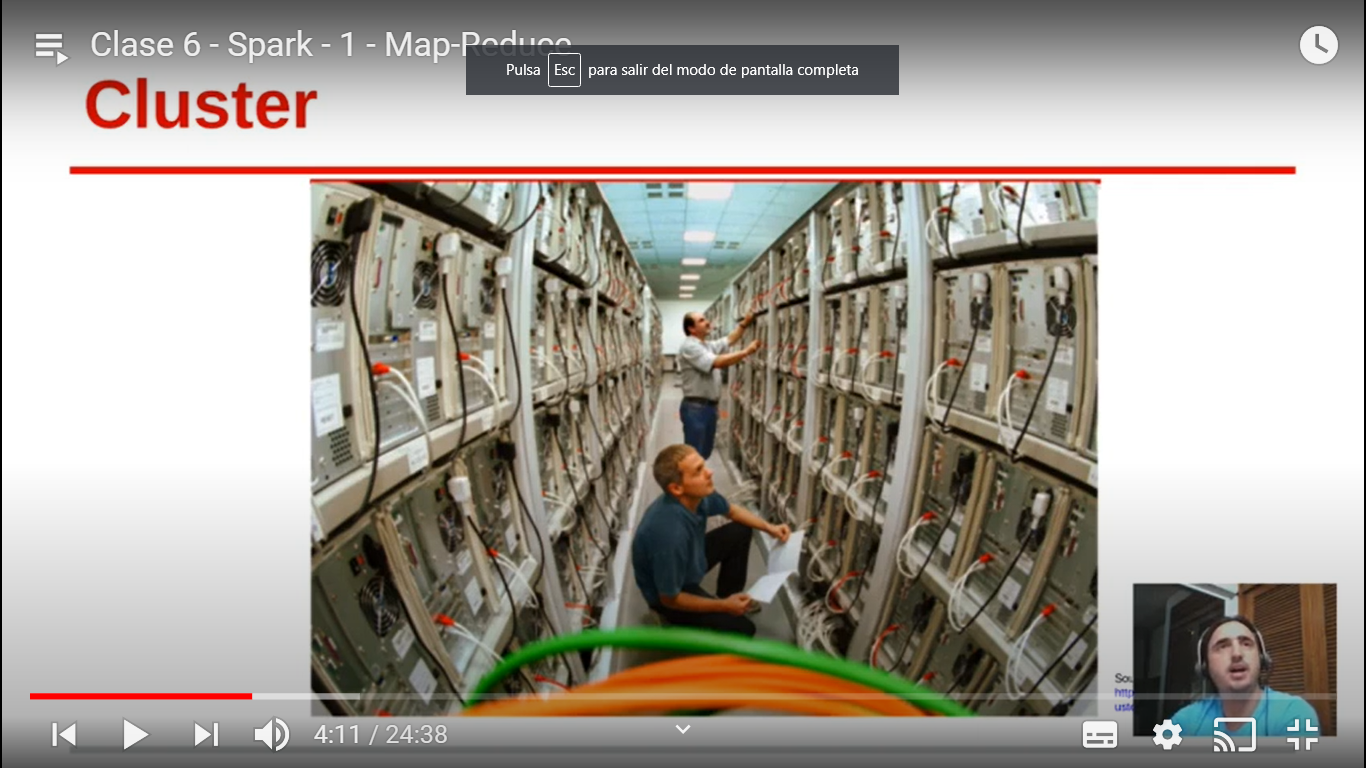
-Los problemas surgen cuando tenemos muchísimos más datos. Tenemos que preocuparnos en cómo procesar estos datos eficientemente de modo que se tarde un tiempo razonable en este proceso.

-Acá vamos a ver cómo procesar los datos cuando los mismos son TAN grandes que no entran en un solo servidor y tienen que estar distrbuidos en varios servidores. Una posible solución a esto es el procesamiento distribuido utilizando map-reduce.

-¿Qué es **Map-Reduce**? 🡪 Es el procesamiento distribuido de datos utilizando un **CLUSTER** de computadoras; de modo que cada nodo del **Cluster** (cada PC) procesará los datos que tiene ese nodo y luego se obtendrá el resultado completo.

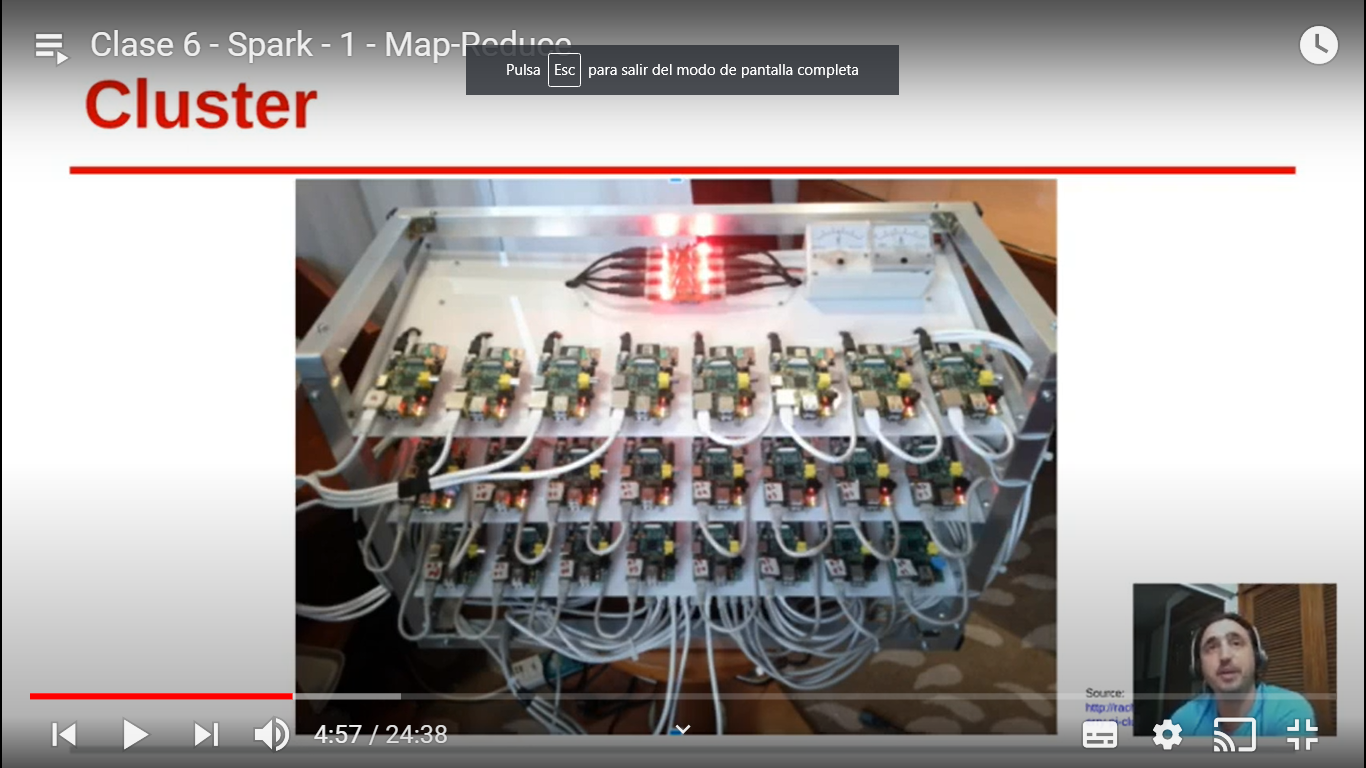
-¿Qué es un **Cluster**? 🡪 Es un conjunto de computadoras que trabajan juntas y pueden ser vistas como un sistema único. Cada computadora del Cluster se la llama **NODO**. Y cada Nodo puede tener una configuración específica o todos los Nodos pueden tener la misma configuración. Cada Nodo son PC’s “estándar”, NO son supercomputadoras. Hay clusters que sirven para distintas tareas. Los nodos se tienen que comunicar entre sí, por esto se utilizan generalmente redes Ethernet de alta velocidad.

-Ejemplo de un Cluster en una Universidad de Alemania:



🡪 Vemos que cada nodo es un gabinete de PC estándar... y en la foto parece que hay cientos de estos nodos. Las empresas más grandes (Google, Amazon, etc.) tienen clusters con miles de nodos.

-Acá por ej. vemos un Cluster formado por nodos que son RaspberryPi:



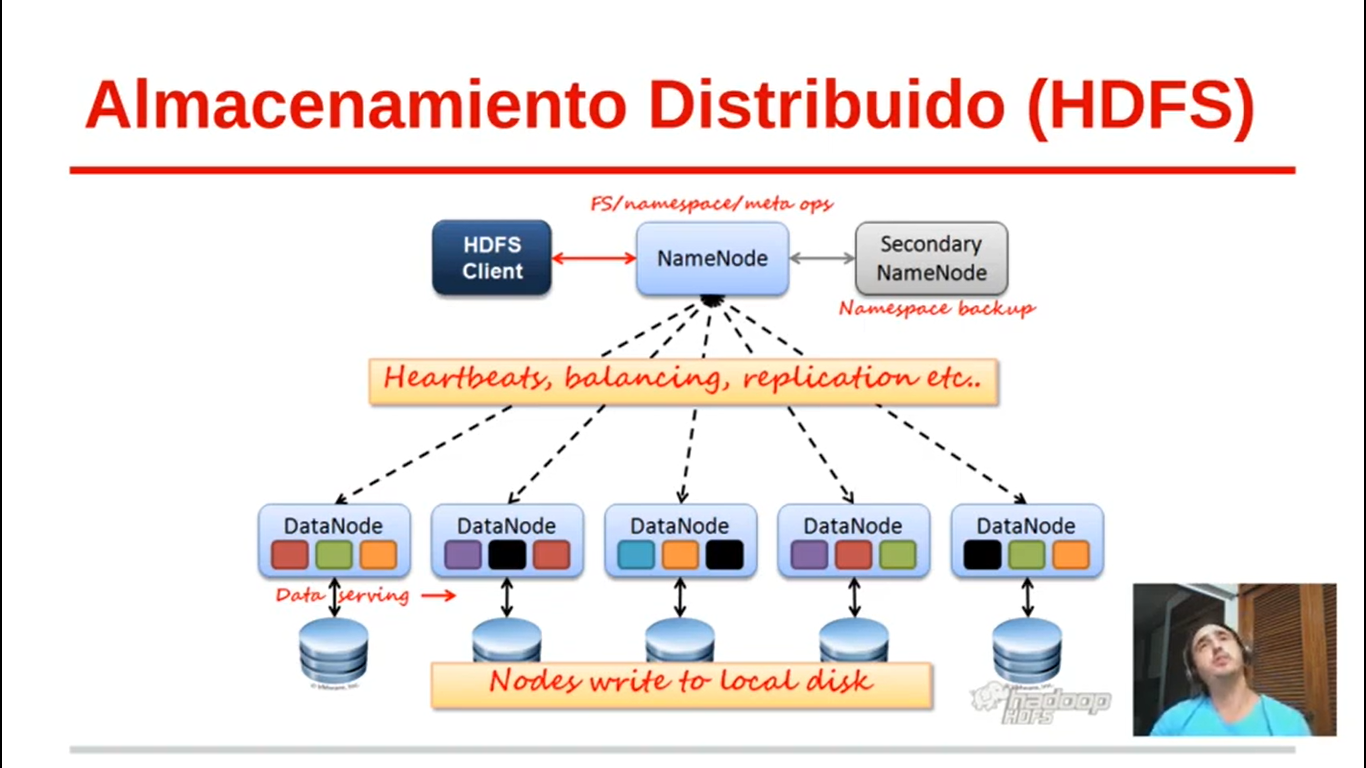
-Cuando tenemos los datos distribuidos en distintos nodos, tenemos que tener un sistema de archivos que pueda manejar estos datos distribuidos... por esto el almacenamiento distribuido debe ser un **FileSystem Distribuido**. Funcionalidades que debe tener el mismo:

* Es el encargado de gestionar cómo y dónde guardar la información en una PC y cómo poder consultarla. Los bloques que componen el FileSystem pueden estar en distintos nodos. Otras funcionalidades.
* Debe almacenar grandes volúmenes de datos en múltiples equipos.
* Replicación de datos. Si un nodo se cae queremos recuperar el archivo, entonces lo que hace el FileSystem Distribuido es replicar los bloques de los archivos en cierta cantidad de nodos.
* Tolerancia a fallos. Por ejemplo debe ser tolerante a problemas en los nodos o en las redes.
* Alta disponibilidad. Debe seguir funcionando aunque algún nodo se caiga.
* Relativamente de bajo costo.

-Distintas implementaciones de **File System** Distribuido:

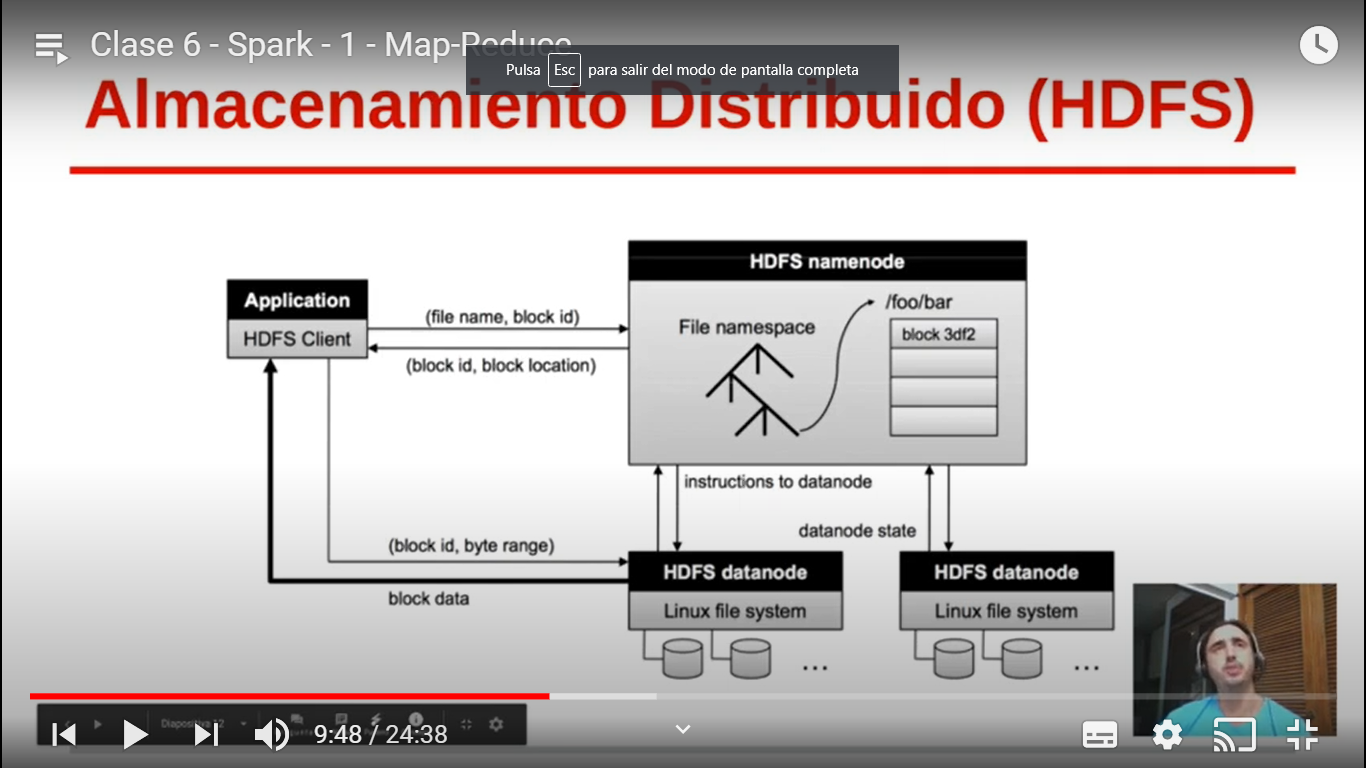
* GFS (Google).
* HDFS (Apache Hadoop).
* CEPH (la integran varias empresas de Internet).
* S3 (Amazon).

-Diagrama de un File System HDFS:

-

🡪 Vemos que el cliente HDFS se comunica con el “NameNode” (el encargado de, a partir del nombre de un archivo, devolverle al cliente los ID de bloque y el nodo en que se encuentran estos ID). El Secondary NameNode es un NameNode de backup por si se cae el principal. Y los “DataNode” son los nodos, donde vemos que también tienen ciertas funcionalidades (replicación y balanceo de carga).

-Acá hay otro diagrama de almacenamiento distribuido mediante HDFS:



🡪 Acá podemos ver el detalle del flujo de datos: el HDFS Cliente le pide al HDFS NameNode un nombre de archivo (osea “quiero obtener este archivo”) y el NameNode le devuelve block ID y su localización (el nodo que tiene ese bloque). Y luego el cliente HDFS le pide a cada uno de los nodos los bloques correspondientes.

-Ahora que ya tenemos nuestro Cluster con nuestros datos distribuidos podemos resolver las consultas sobre este Cluster utilizando la estrategia Map-Reduce:

-Propuesto en el año 2004 por la gente de Google. Surge de la necesidad de Google para procesar muchísima cantidad de información de las páginas web. Estos datos estaban almacenados de una forma distribuida y con Map-Reduce encontraron una forma eficiente de procesarlos.

-Map-Reduce es un modelo de programación para procesar grandes volúmenes de datos. Y como dijimos, surge de la necesidad de procesar grandes volúmenes de datos **de forma escalable**.

-En Map-Reduce el usuario especifica una función **MAP** que procesa los registros que se deseen y nos devuelve / **genera un conjunto intermedio de pares clave/valor**. De esta forma transformamos nuestros datos originales en datos “intermedios” que nos acerca más a nuestra forma final deseada. También podemos filtrar los registros que NO nos interesan (Ver como).

-Se debe especificar también una función **REDUCE** que combina todos los valores asociados a la misma clave (de esta manera si tenemos registros que no nos interesan simplemente los filtramos con reduce). De esta manera el resultado de MAP, que nos daba un resultado en clave-valor, ahora con REDUCE combinamos los resultos y obtenemos una nueva salida. Y esta salida podría ser utilizada para un nuevo proceso map-reduce y de esta forma resolver consultas más complejas.

**-1ra etapa - MAP:**

-Como comentamos previamente, **MAP transforma nuestros dato**s.

-**Debe ser aplicada/evaluada a cada dato de nuestro set**. Si tenemos 100 millones de registros, la función MAP va a ser evaluada en estos 100 millones de registros.

-Ya que MAP es aplicada registro por registro, entonces **puede ser paralelizada** y distribuirse entre las distintas máquinas de un Cluster. **De esta forma la consulta es resuelta en forma paralela entre todos los nodos que tengamos en nuestro Cluster y aprovechamos todo nuestro poder de procesamiento.**

-Dependiendo de la implementación pueden haber algunas diferencias para usar MAP:

* En **HADOOP** un MAP convierte un registro (clave, valor) en uno o más registros con otra clave y otro valor 🡪 **Map(k,v) -> [k2,v2]**
* En cambio, en **SPARK** el MAP NO diferencia entre clave y valor, sino que toma todo el registro como uno 🡪 **Map(R) -> [R’]**

**-2da etapa - REDUCE:**

-Consiste en **combinar los resultados del MAP anterior**.

-Para esto e**s necesario procesar TODOS los datos devueltos por el MAP de TODAS las máquinas de nuestro Cluster**. Lo que tenemos que tener en cuenta acá es que los datos que queremos combinar algunos estarán en un nodo y otros en otros nodos.

-Entonces, para poder combinarlos tenemos que pasar a una etapa de “shuffle & sort” (barajar y ordenar) donde los datos se envían de un nodo a otro para tener todos los datos en el mismo nodo. De esta manera, con esta etapa se REDUCE locales en paralelo y se REDUCE entre máquinas.

-Tambien hay algunas diferencias funcionales de REDUCE de acuerdo a su implementación:

* En **HADOOP:**
  + La función REDUCE recibe un registro (clave, valor) y devuelve otro registro también con (clave, valor) 🡪 **ReduceByKey((k,v),f) -> [(k,v)]**
  + El sistema agrupa todos los registros con la misma clave y aplica la función de REDUCE.
  + Requiere que todos los registros de igual clave, a los cuales les aplicaremos REDUCE, **estén en el mismo equipo / nodo.** Acá necesitamos la etapa “Shuffle & Sort” para hacer que los datos que tengan la misma clave sean enviados a un mismo nodo. Esta etapa es muy crucial, ya que mover datos de un nodo a otro es muy costoso (Ya que la velocidad de la red al querer mover datos de un nodo a otro es más lenta que la velocidad interna al mover datos internamente en un mismo nodo).
* En **SPARK**:
  + Podemos encontrar 2 variantes de REDUCE:
    - **ReduceByKey**: similar al ReduceByKey de Hadoop, donde se combinan los elementos para una misma clave.
    - **Reduce**: dá un único resultado para TODO el set de datos.
  + En AMBAS variantes del REDUCE:
    - La función Reduce toma 2 valores para dar como resultado la combicación de ambos.
    - Y el resultado de un Reduce entre dos registros es el input del siguiente Reduce. Osea se van calculando el Reduce de a 2 valores. De esta forma el Reduce se va ejecutando de forma paralela... en cada nodo se va calculando el reduce y luego se debe calcular el reduce entre el resultado de todos los nodos para obtener el resultado final.
  + Ejemplo:
    - Si tenemos información de ventas (fecha, cliente, importe). Y queremos obtener el importe total de todas las ventas. Entonces lo que podemos hacer es un MAP sencillo que reciba un registro A (una venta) y devuelva A[importe] (osea el importe de la venta):
      * **MAP(A) -> A[importe]** 🡪 Esta es nuestra función de MAP que se va a ejecutar para todos los registros de forma distribuida en todos los nodos y vamos a obtener un nuevo registro donde únicamente vamos a tener el importe de cada una de las ventas.
    - Luego de esto tenemos que definir el REDUCE (en SPARK como dijimos se definía con 2 parámetros: A y B... osea que toma 2 registros). Y REDUCE tiene que indicar cómo combinar estos registros... en este caso como queremos calcular el importe total de ventas hacemos simplemente A + B (donde A es el importe de una venta y B es el importe de otra venta). Entonces para obtener el total sumamos A + B:
      * **REDUCE(A,B) -> A+B** 🡪 Acá tenemos definido el REDUCE.
    - 1ro se va a ejecutar el MAP en cada nodo y en cada uno de los registros de los nodos y al resultado de esto se ejecuta el REDUCE... entonces cada nodo va a poder hacer la suma de los importes que tenga ese nodo. Y luego se tienen que combinar los resultados parciales de ese nodo para obtener así el resultado final de todo el procesamiento (el importe total de todas las ventas, lo que buscábamos).
    - Hay que tener en cuenta acá que, como es necesario que cada nodo le envíe al resto su resultado parcial, acá está la estapa comentada previamente de Shuffle & Sort que es cuando se tiene que mover la información de un nodo a otro.

-Como REDUCE es ejecuta de a 2 datos a la vez de forma distribuida, entonces esta función debe cumplir con ciertos requisitos para que el resultado tenga una coherencia. Osea que las operaciones de REDUCE deben ser **CONMUTATIVAS** y **ASOCIATIVAS** de modo de ejecutarse distribuidamente y que el resultado tenga sentido. Por ejemplo si ejecutamos REDUCE con una función que no es conmutativa y asociativa como la RESTA... (por ejemplo si en el ejemplo anterior usamos a reduce como “**A – B**”... entonces el resultado que obtengamos de esto no va a tener ningún sentido, ya que va a depender del órden en que se ejecuten las operaciones REDUCE... y si las ejecutamos nuevamente nos va a dar otro resultado diferente).

**-Etapa de Shuffle & Sort que ya comentamos:**

* + Es la etapa donde se mueven datos de un nodo a otro: mueve la salida de un proceso MAP a un cierto equipo de tal forma que un REDUCER pueda procesar sus registros.
  + Como dijimos es la fase más costosa del proceso Map-Reduce.
  + Optimización para reducir la cantidad de datos a transmitir por la red.

-Lo bueno de todo esto es que hoy en día hay **Frameworks de Map-Reduce**:

* + Nos dan un importante nivel de abstracción para el procesamiento distribuido utilizando Map-Reduce.
  + Estos frameworks tienen APIs de distintos niveles para ejecutar estas tareas de Map-Reduce. Hay también APIS de muy alto nivel donde ni siquiera tenemos que definir una función de map y reduce... únicamente usamos una función que nos dá la API e internamente el framework lo convierte en Map-Reduce.
  + Además, el framework nos permite manejar/administrar todos los trabajos a ejecutar en cada máquina/nodo del cluster: osea qué nodo hace cada trabajo, dónde están los datos, manejo de errores en los trabajos (qué pasa si un nodo se cae o está lento, qué pasa si un trabajo falla), etc. Todas estas funcionalidades y también los problemas, el framework nos va a ayudar para solucionarlos y nos va a proveer distintas configuraciones o parámetros para poder manejar estas funcionalidades. Y con el framework podemos abstraernos de todo esto.

**2-Apache Spark.**

-¿Qué es Apache Spark?: Es un sistema de procesamiento distribuido que posee un framwork de abstracción para ejecutar trabajos de Map-Reduce en forma distribuida y paralela en un Cluster (formado por varios nodos como vimos anteriormente). Nos soluciona todos los detalles de bajo nivel (como la división de trabajos en cada uno de los nodos del cluster, el manejo de los errores, administración de recursos, etc.), de modo que nosotros nos tendramos que preocupar solo en cómo procesar los datos y qué tareas definir para obtener la información que queremos de nuestros datos.

-Spark posee APIS en: **JAVA**, **Scala**, **Python** y **R** con distintos niveles de abstracción y para diferentes fines.

-Distintas APIS de alto nivel:

* **RDD API** 🡪 es la API principal. Es el que provee todas las funcionalidades para ejecutar trabajos de Map-Reduce para poder ejecutar trabajos distribuidos en todos los nodos de nuestro Cluster.
* **DataFrame API** 🡪 API de mayor nivel de abstracción que nos permite hacer consultas de nuestros datos SIN tener conocimientos previos de la dinámica Map-Reduce.
* **Spark SQL** 🡪 Tambien podemos consultar nuestros datos mediante consultas SQL.
* **MLib** 🡪 Sirve para realizar trabajos de Machine Learning sobre Spark.
* **GraphX** y **GraphFrames** 🡪 2 APIs que nos permiten procesar información de GRAFOS en Spark.
* **Spark** **Streaming** 🡪 Nos permite procesar Streaming de datos.

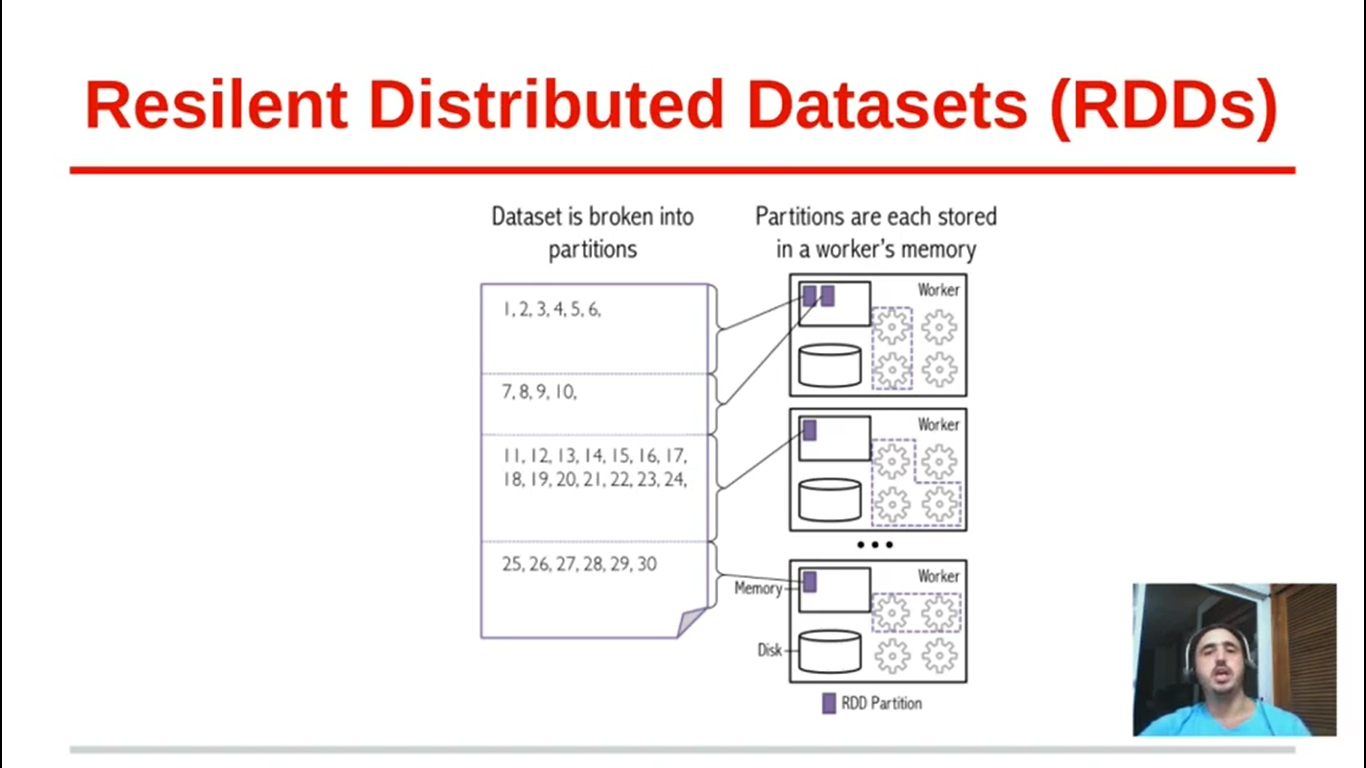
-Dinámica y **arquitectura** de Spark:

* La “receta” de procesamiento se define en el Driver: hay una comunicación entre un driver y una serie de **ejecutores** (**Executors**). De esta manera, **el usuario, genera y le envía al Driver un Script con la definición de las Tareas, le indica cómo quiere procesar a los datos.**
* Estas **Tareas** (Jobs) del Driver son enviadas a cada uno de los Executors que se encuentran en cada uno de los Nodos del Cluster.
* Estos Executors procesan los datos en base a las tareas definidas y devuelve los resultados de esas tareas al Driver.
* Resumen: Una aplicación Spark en bajo nivel consiste en un programa Driver que lanza varias operaciones paralelas en múltiples Executors que corren en un Cluster.

-**Resilent** **Distributed** **Datasets** (**RDDs**):

* **Los RDD son las estructuras de datos que maneja Spark**, **son colecciones de datos que son particionadas en los distintos nodos de nuestro Cluster.**
* Estas colecciones de datos son **guardados en memoria** (o en disco en su defecto). Spark trata de trabajar lo más posible en **memoria**, ya que es más rápido los tiempos de ejecución que utilizando el disco en el cual se tarda más tiempo en acceder.
* Además, los RDDs **son reconstruidos automáticamente** frente a fallos de máquinas o demoras en un Job.
* Tambien los RDDs **pueden ser creados a partir de datos externos** (como un archivo que esté en un HDFS o en Amazon F3 o en un FileSystem local o datos generados en el driver, etc.).

-Esquema para ver cómo trabaja un RDD:



-->Los datos en este caso son números y están divididos en particiones. Estas particiones son distribuidas en los nodos (Workers) de nuestro Cluster para ser procesadas distribuidamente. Cada nodo procesa los datos de las particiones que se encuentran en su memoria.

-Operaciones en Spark que podemos ejecutar sobre un RDD:

**1-Transformaciones.**

**2-Acciones.**

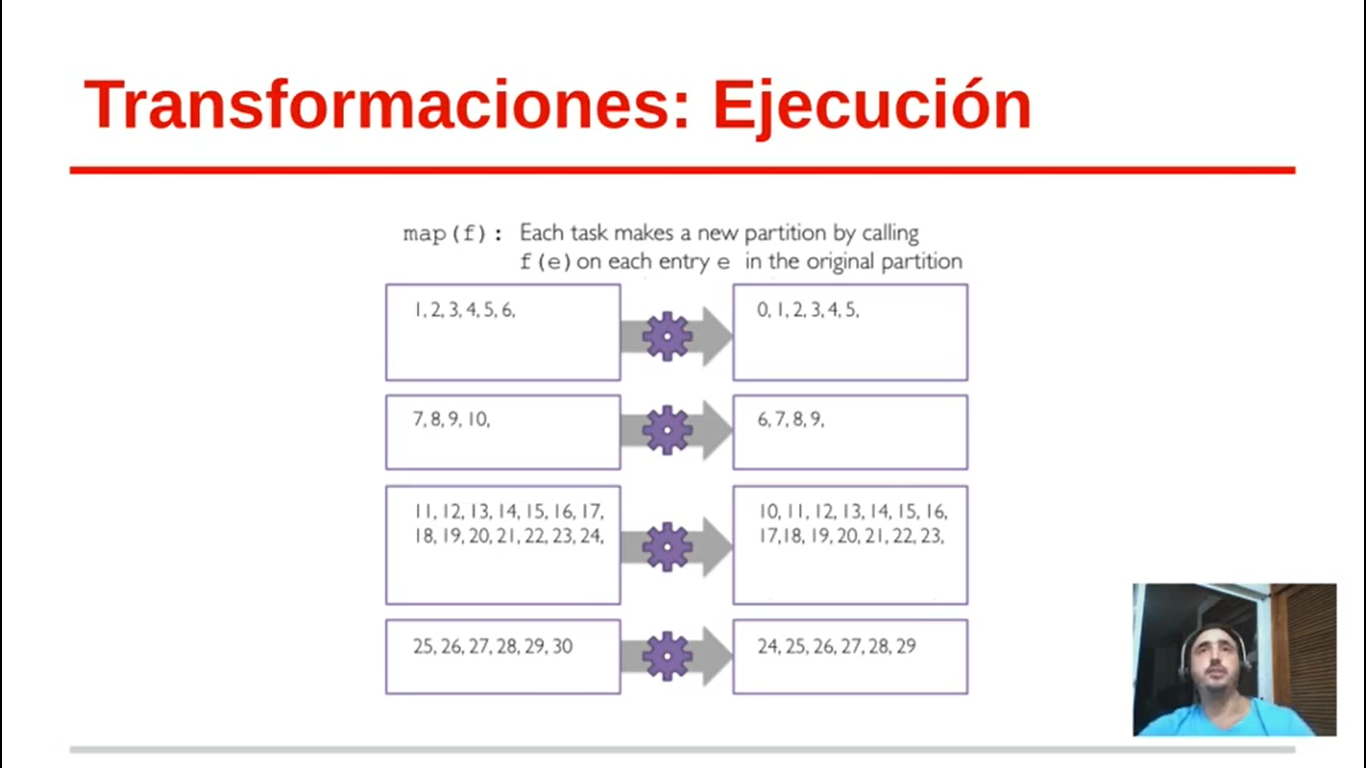
**1-Transformaciones:**

-**Crean un nuevo RDD a partir de uno o más RDDs existente/s**. Las transformaciones se encadenan una detrás de otra de modo de ir transformando nuestros datos y llegando así a la información que necesitamos.

-Las transformaciones **son Lazy** (perezosas), ya que **solo se van a ejecutar en el momento en que se ejecute una acción**. Nosotros podemos ir definiendo transformaciones encadenadas una detrás de otra pero en el momento de ejecutarlas no van a tardar casi nada. Spark lo único que va a hacer es definir el plan de trabajo.... pero hasta que no le definamos una acción no va a ejecutar nada.

-Los RDDs, asi como los RDDs resultantes de las transformaciones **se pueden cachear/persistir** (usando la Caché obviamente; ya sea en la Memoria RAM o en Disco). Al cachear un RDD todas las particiones en todos los nodos se van a cachear y de esta forma se evita el recalculo en todos los nodos. Esto es MUY útil cuando un RDD luego va a ser utilizado por varias transformaciones... entonces de esta forma evitamos tener que recalcular este RDD que cacheamos para cada una de estas transformaciones que lo utilizan.

-Acá vemos como se ejecuta una Transformación… en este caso la Transformacion MAP:



🡪 Se transforma un registro en otro registro en base a una función MAP. Esta función es ejecutada en cada una de las particiones (en cada uno de los nodos) y devuelve una nueva partición con los datos modificados. En este caso se ve que MAP solo resta un 1 a cada número. **Una transformación lo que va a hacer es crear un “STAGE” (un grupo de tareas que van a realizar el mismo cómputo pero con distintos datos de entrada: osea con una partición distinta que se va a ejecutar en un nodo distinto). Necesitamos lanzar una tarea para cada una de las particiones en que tengamos dividido nuestro RDD... y de esta forma vamos a transformar cada una de las particiones originales en una partición con los datos modificados.**

Algunas de las Transformaciones disponibles en Spark:

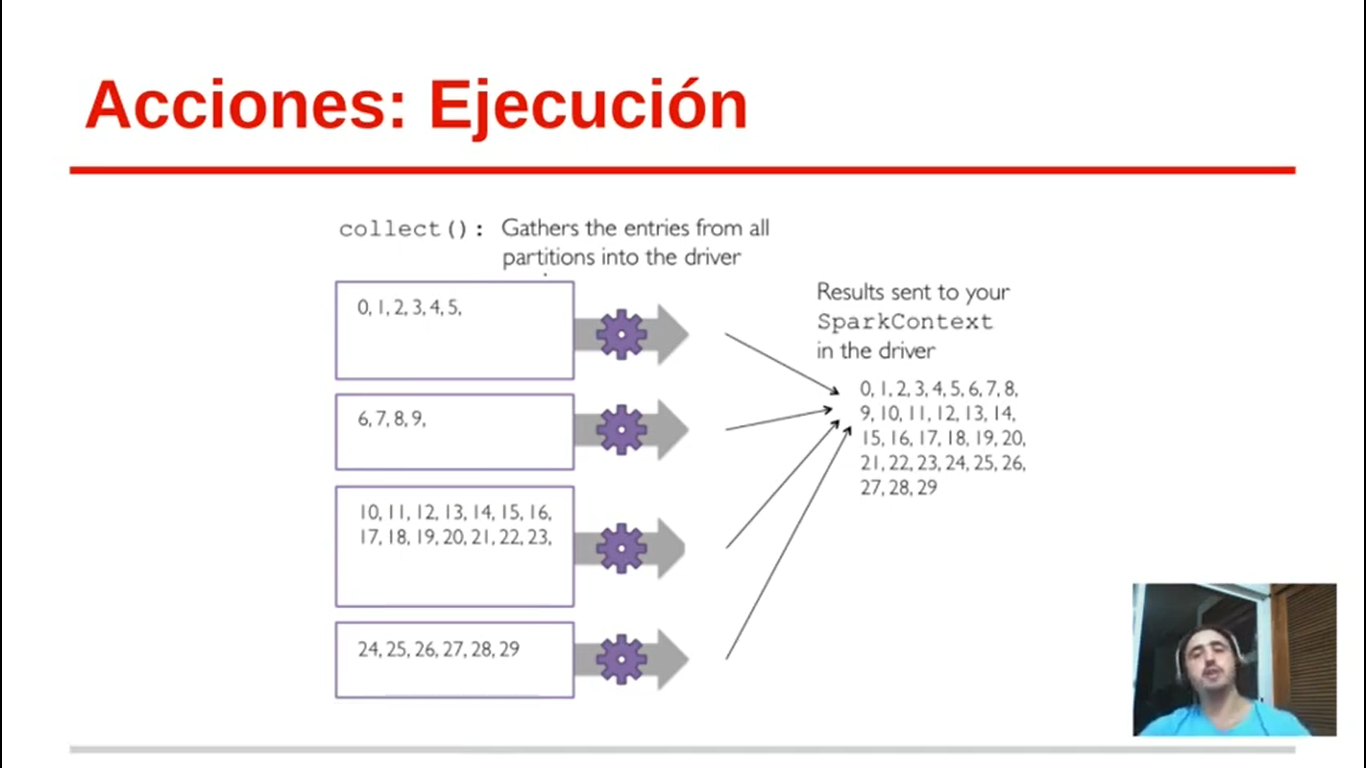
* **Map.**
* **Filter.**
* **FlatMap.**
* **ReduceByKey.**
* **GroupByKey.**
* **Join.**

🡪 Todas estas las vamos a ver en los siguientes videos.

**2-Acciones:**

-Es el otro tipo de operación que podemos aplicar sobre un RDD. Las acciones **devuelven un valor al Driver luego de procesar los datos.** Son las que provocan que varias transformaciones encadenadas finalmente sean ejecutadas... es esto que hablábamos de que las Transformaciones son Lazy. Así, una vez que ejecutamos una acción vamos a estar ejecutando tooodas las transformaciones que hayamos definido antes. De esta forma con la acción vamos a procesar las transformaciones y a devolver el resultado de la acción al Driver.

-Acá vemos como se ejecuta una Acción… en este caso la Acción COLLECT:



🡪 Collect lo que hace es simplemente traer los datos de todas las particiones resultantes de las transformaciones; y toda esta data que se compone de las particiones de los cluster es devuelta al Driver. Como comentamos antes tenemos que tener cuidado en no colapsar al Driver colocándole demasiada información.

-Algunas de las Acciones disponibles en Spark:

* Reduce.
* Collect.
* Count.
* Take.
* TakeOrdered.
* First.

🡪 Todas estas las vamos a ver en los siguientes videos.

-Guía de programación del API RDD donde podemos encontrar info generar de la API y el detalle de las Transformaciones y Acciones disponibles:

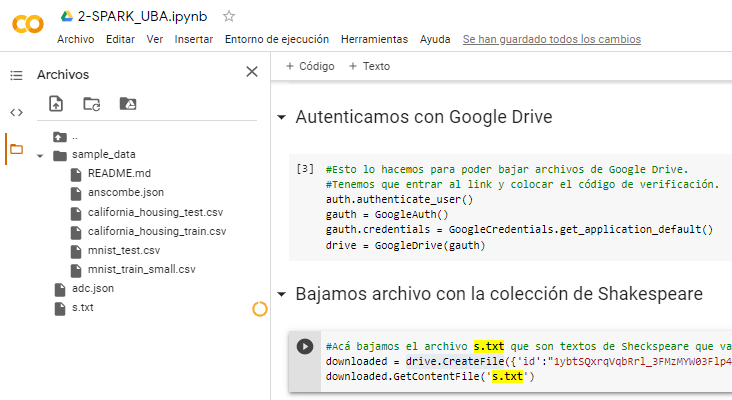
-<https://spark.apache.org/docs/latest/rdd-programming-guide.html>

**-Las siguientes secciones del índice (Del 3 al 9) verlos directamente de Spark (archivo “SPARK\_UBA.IPYNB”):**

**3-Lectura de Datos.** 🡪 Parte “1-Inicio SPARK y Lectura de datos”.

**-PARA SABER:**

🡪El archivo que descargamos de Google Drive está acá:



**4-Acciones.** 🡪 Parte “2-Acciones”.

**5-Transformaciones.** 🡪 Parte “3-Transformaciones“.

**6-Wordcount.** 🡪 Parte “4-Más ejemplos de transformaciones y acciones con los textos de Shakespeare (con uso de wordsCount)”.

**7-Joins.** 🡪 Parte “5-Transformaciones entre dos RDD (con uso de Joins)”.

**8-Transformaciones sobre particiones.** 🡪 Parte “6-Transformaciones sobre las particiones“.

**9-Persistencia y cache.** 🡪 Parte “7-Persistiendo RDD (Persistencia y CACHE)”.