Contenido

Clase 01 - Introducción BigData

Definición y Naturaleza:

- Big Data no es fácil de definir y el término fue "inventado por el marketing".
- Representa una **nueva generación de tecnologías y arquitecturas**.

- Están diseñadas para **extraer valor económicamente** de volúmenes muy grandes de una amplia variedad de datos.
- Involucra la capacidad de permitir la **captura, el descubrimiento y/o el análisis de alta velocidad**.
- No es una tecnología única, sino una **combinación de tecnologías** antiguas y nuevas que ayudan a las empresas a obtener información práctica.
- Es el resultado de la convergencia de factores tecnológicos que han transformado la manera en que se gestionan y aprovechan los datos.
- El auge de Internet, el continuo crecimiento de las redes sociales, los sitios de archivos multimediales y el e-commerce contribuyeron al surgimiento del Big Data.
- La reducción de costes en el almacenamiento y los ciclos de computación ha alcanzado un punto crítico, haciendo posible almacenar y analizar datos que antes eran inasequibles.

Dimensiones de Big Data (Las 'V'):

El Big Data se clasifica generalmente según tres características principales, aunque se mencionan otras:

Volumen: La capacidad de gestionar un **enorme volumen de datos**. En 2015, el universo digital estaba compuesto por 6 Zettabytes (ZB) de datos (1 ZB = 1000 Hexabytes).

Velocidad: La rapidez con la que se generan y procesan los datos. La proliferación de sensores en tiempo real es un buen ejemplo de esta alta velocidad. Los datos en tráfico, de vida efímera pero alto valor, crecen más deprisa que el resto del universo digital.

Variedad: Se refiere a la **amplia variedad de tipos de datos**. El contenido desestructurado está creciendo.

Valor: La capacidad de **extraer valor** de esta información (por ejemplo, mejoras en el rendimiento del negocio, segmentación de clientes, toma de decisiones y automatización de decisiones tácticas).

Veracidad: La precisión de los datos para predecir el valor empresarial y asegurar que los resultados del análisis tengan sentido.

Tipos de Datos en Big Data:

Datos estructurados: Tienen una longitud y formato definidos (ej., números, fechas, cadenas). Suelen almacenarse en bases de datos relacionales (RDBMS). Representan aproximadamente el 20% del total de datos. Ejemplos: Datos de sensores, logs de aplicaciones, operaciones bancarias.

Datos no estructurados: No siguen un formato específico. Constituyen aproximadamente el 80% de los datos disponibles. Ejemplos: Texto escrito en lenguaje natural, contenido multimedia (imágenes, fotos, audio, video), informes, reportes, redes sociales.

Datos semiestructurados: Presentan una estructura parcial (ej., archivos de texto plano, planillas de cálculo).

Requisitos de Tiempo:

Problemas de tiempo real (Online): Requieren baja latencia, es decir, un tiempo de retardo mínimo para que un servicio se ejecute. Ejemplos: Detección de fraudes, detección de fallas, publicidad web.

Problemas de no tiempo real (Batch/Off-line): Procesamiento de grandes conjuntos de datos. Ejemplos: Segmentación de clientes, tomas de decisiones (semanales, mensuales, anuales).

Desafíos y Tecnologías Fundamentales:

Desafíos clave: Almacenamiento, procesamiento (debe ser rápido y efectivo), y la diversidad de los datos.

Computación Distribuida: Es una tecnología fundamental que permite conectar computadoras en red para escalar tareas. El Big Data requiere la integración de redes rápidas y clústeres de hardware económicos que se pueden combinar en *racks*.

Virtualización y Cloud Computing: La virtualización (el uso de recursos informáticos para imitar otros recursos) proporciona la eficiencia necesaria para las plataformas de Big Data y permite la escalabilidad. El modelo de nube (Cloud Computing) es ideal por su escalabilidad elástica y su arquitectura distribuida, facilitando la implementación práctica del Big Data.

MapReduce, Hadoop y BigTable: Estas innovaciones fueron cruciales para impulsar la gestión de datos, permitiendo procesar cantidades masivas de datos de manera eficiente y rentable.

Sistemas de Gestión de Bases de Datos (DBMS): Se utilizan bases de datos relacionales (RDBMS) como MySQL y PostgreSQL, así como bases de datos NoSQL (Not only SQL).

- Las bases de datos NoSQL (ej., MongoDB, CouchDB, Riak, Hbase) son esenciales para Big Data, ya que están diseñadas para gestionar grandes volúmenes de datos no relacionales con alta escalabilidad y flexibilidad.

Aplicaciones y Casos de Uso:

- El Big Data es crucial para resolver problemas complejos y multidisciplinares.

- Casos de uso incluyen: **Segmentación de clientes** (marketing, ventas, *churn* de clientes), **optimización de procesos de negocio** (laboratorios, farmacias, hospitales), **rendimiento deportivo** (patrones de juego, análisis con imágenes y sensores), y **seguridad** (fraudes, ciberataques, perfil criminal, detección de amenazas).
- También se aplica en la **optimización de ciudades** (tráfico, suministro eléctrico).

Analogía para Solidificar la Comprensión:

Imaginar Big Data es como pasar de tener una biblioteca pequeña con libros perfectamente catalogados (datos estructurados y bases de datos tradicionales) a gestionar el contenido de todas las bibliotecas, archivos y conversaciones del mundo (la marea de información digital). Para manejar este volumen (Volumen), esta mezcla de formatos (Variedad), y la rapidez con la que se generan nuevos textos y grabaciones (Velocidad), no se puede usar el mismo método de archivo antiguo. Se necesita un **sistema de bibliotecas distribuidas** (computación distribuida y Nube) que use robots y algoritmos eficientes (MapReduce, Spark) para escanear, categorizar y encontrar valor en esa inmensa cantidad de información dispersa.

La Pila (Pila Tecnológica o Arquitectura)

El concepto de **"Pila"** hace referencia al **modelo de arquitectura en capas** o *framework* de referencia que se utiliza para visualizar y organizar todos los componentes de software, hardware, servicios y bases de datos necesarios para gestionar una solución integral de Big Data.

Necesidad Arquitectónica: Dado que los datos se han convertido en el motor del crecimiento y la innovación, es fundamental contar con una **arquitectura subyacente** que respalde las crecientes exigencias.

Estructura del Sistema: La pila tecnológica se utiliza como un marco de referencia para considerar las tecnologías de Big Data que aborden los requisitos funcionales de los proyectos. El diseño de esta arquitectura se debe concebir como una **estrategia**, no como un simple proyecto.

Componentes de la Pila (Vistos como Capas): La pila se organiza en capas que van desde el hardware hasta las aplicaciones:

- **Capa 0**: Infraestructura Física Redundante: El nivel más bajo de la pila, incluye el hardware y la red, y debe ser resiliente y redundante.
- **Capa 1**: Infraestructura de Seguridad: Protege todos los elementos del entorno de macrodatos.
  - Interfaces y Fuentes: Proporcionan acceso bidireccional a todos los componentes, desde las aplicaciones corporativas hasta los flujos de datos de Internet, siendo las APIs abiertas esenciales.

- **Capa 2**: Bases de Datos Operativas: Motores de bases de datos, rápidos, escalables y robustos, que contienen colecciones de datos relevantes para el negocio. Aquí se incluyen los SGBD relacionales (SQL) y los NoSQL, como MongoDB y PostgreSQL.
- **Capa 3**: Organización de Datos: Incluye servicios y herramientas (como MapReduce y Hadoop) que capturan, validan y ensamblan diversos elementos de macrodatos en colecciones contextualmente relevantes.
- **Capa 4**: Almacenes de Datos Analíticos: Contiene los almacenes de datos organizados para facilitar el análisis del negocio.

Integración: Sin **servicios de integración**, el Big Data no es posible, ya que se necesitan interfaces en cada nivel y entre cada capa de la pila.

Clase 02 - Map-Reduce; Hadoop; HDFS

I. Hadoop (Framework y Ecosistema)

Hadoop es un *framework* de software de código abierto desarrollado originalmente por Doug Cutting en Yahoo! (basado en el trabajo de Google) y gestionado por la Fundación Apache.

Definición y Objetivo:

- Es un *framework* que soporta el procesamiento de grandes bases de datos en un ambiente distribuido.
- Ejecuta aplicaciones para el tratamiento de grandes volúmenes de datos.
- Está diseñado para procesar enormes cantidades de datos (desde terabytes hasta petabytes) estructurados y no estructurados, implementado en *racks* de servidores estándar (hardware económico).

Componentes Centrales (El Núcleo):

Hadoop se compone de dos elementos principales:

1º. Sistema de Archivos Distribuido de Hadoop (HDFS): El clúster de almacenamiento de datos.
2º. Motor MapReduce de Hadoop: Una implementación de alto rendimiento del algoritmo MapReduce para el procesamiento distribuido.

Modelo de Operación:

- Diseñado para el **procesamiento *off-line*** de los datos (procesamiento en *batch*).

- Funciona con la idea de "escriba una sola vez y lea muchas".
- **No permite la lectura aleatoria** ni el procesamiento *on-line* (en tiempo real).
- Tolerante a fallas y diseñado para ser autorreparable (puede detectar fallos, adaptarse y seguir funcionando).
- Se ejecuta en el lugar donde se encuentran los datos, buscando minimizar la latencia moviendo el código al dato.

Otros Componentes del Ecosistema Hadoop:

La plataforma incluye otros proyectos complementarios:

- **YARN (Yet Another Resource Negotiator):** Servicio central de Hadoop para la administración de recursos y la planificación/seguimiento de trabajos, crucial para optimizar la utilización del clúster.
- **HBase:** Base de datos distribuida NoSQL orientada a columnas (inspirada en BigTable de Google) que usa HDFS para almacenamiento persistente, proporcionando acceso aleatorio de lectura/escritura en tiempo real a grandes volúmenes de datos.
- **Hive:** Capa de almacenamiento de datos orientada a procesamiento por lotes que utiliza HDFS y MapReduce, y proporciona un lenguaje similar a SQL llamado HiveQL para facilitar el análisis.
- **Pig:** Entorno basado en *scripts* (Pig Latin) que simplifica el uso de Hadoop para no programadores, liberándolos de la necesidad de escribir código MapReduce directamente.
- **Sqoop (SQL-to-Hadoop):** Herramienta para la importación y exportación de datos en bloque entre Hadoop y almacenes de datos externos (como RDBMS).
- **Zookeeper:** Servicio de coordinación esencial para la creación de aplicaciones distribuidas, gestionando la sincronización de procesos y colas.

II. HDFS (Hadoop Distributed FileSystem)

HDFS es el sistema de archivos distribuido de Hadoop, diseñado para ser altamente escalable y tolerante a fallos, formando los cimientos del ecosistema Hadoop.

Definición y Características:

- Es una solución versátil y resiliente basada en *clusters* para la gestión de archivos en entornos de macrodatos.
- Ofrece transparencia al usuario, permitiendo operar con todos los archivos del *cluster* como si estuvieran en un solo sistema de archivos.
- Está optimizado para el caso donde los datos **se escriben una sola vez y se leen repetidamente**.

Manejo de Archivos (Bloques):

- Todos los archivos se dividen en fragmentos más pequeños llamados **bloques**.
- El tamaño por defecto de los bloques es 64 MB, aunque es configurable.
- Los bloques pueden estar físicamente en cualquier computadora (DataNode).

Arquitectura (Componentes de Procesos):

HDFS opera con una arquitectura maestro-esclavo:

NameNode (Maestro):

- Maneja el árbol del *filesystem* y los metadatos de cada archivo y carpeta (datos sobre datos, como permisos, fechas de modificación, etc.).
- Conoce qué *DataNode* maneja cada bloque del *filesystem*.
- Gestiona todo el acceso a los archivos, incluyendo lecturas, escrituras, creaciones, eliminaciones y replicación de bloques.
- Para un rendimiento óptimo, los metadatos se cargan en la memoria física del servidor mientras el *cluster* está funcionando.

DataNode (Esclavos/Servidores de Bloques):

- Son los encargados de llevar a cabo la lectura y escritura de los bloques en el *filesystem* del sistema operativo (SO).
- Llevan a cabo la creación, el borrado y la replicación de los bloques.
- Envían informes periódicos y mensajes de latido (*heartbeat*) al *NameNode*.

Secondary namenode: realiza tareas auxiliares al namenode.

Tolerancia a Fallos y Replicación:

- Permite la réplica de bloques para la recuperación de fallas.
- Los bloques de datos se replican en múltiples nodos para garantizar la resistencia (por ejemplo, tres réplicas: una local, una remota en otro *rack*, y una tercera en un servidor diferente del *rack* remoto).

Operaciones:

- Permite crear, borrar y renombrar archivos y carpetas dentro del FS distribuido.
- Permite copiar archivos desde el FS local al HDFS y viceversa.

III. MapReduce (Paradigma de Programación y Motor de Procesamiento)

MapReduce es tanto un algoritmo como un paradigma de programación y un *framework* de ejecución diseñado para manejar tareas complejas en grandes volúmenes de datos distribuidos.

Origen y Propósito:

- Fue diseñado por ingenieros de Google a principios de la década de 2000.
- Fue crucial para resolver problemas prácticos como el rastreo web y la frecuencia de consultas, que requerían distribución del trabajo a gran escala.
- Permite a los desarrolladores escribir programas capaces de procesar enormes cantidades de datos no estructurados en paralelo en un grupo distribuido de procesadores.
- Es una de las innovaciones que impulsó una nueva generación de gestión de datos, al permitir procesar cantidades masivas de datos de manera eficiente y rentable.

Modelo de Programación y Principios:

- Es un *framework* diseñado para distribuir tareas en múltiples nodos.
- Se basa en el modelo de **programación funcional**, donde los operadores no modifican la estructura de los datos originales, sino que crean nuevas estructuras de datos como resultado (programación no destructiva).
- La filosofía principal es "**escriba una sola vez y lea muchas**".
- Se requiere pensar en cómo resolver un problema sin tener acceso a todos los datos simultáneamente.
- La función de *map* es **conmutativa**, lo que significa que el orden de ejecución no importa, facilitando la paralelización.

Fases Fundamentales del Paradigma (Map y Reduce):

Toda tarea MapReduce se divide en estas dos fases principales:

1. **Fase Map (Mapeo):** Los datos de entrada (generalmente pares clave-valor) son procesados, uno a uno, y transformados en un conjunto intermedio de datos. La función `map` aplica una función a cada elemento de una lista de entrada y genera una nueva lista. El componente map distribuye el problema o las tareas de programación entre un gran número de subprocesos.

2. **Fase Reduce (Reducción):** Se reúnen los resultados intermedios y se reducen a un conjunto de datos resumidos, que es el resultado final de la tarea. La función *reduce* agrega todos los elementos resultantes del cálculo distribuido para obtener un único resultado.

Etapas de Ejecución en Hadoop MapReduce:

Un *job* (unidad de trabajo) de MapReduce se divide en cuatro fases:

1º. **Map:** Tarea programable que procesa los datos de entrada.
2º. **Shuffle:** Fase interna donde se recopilan los resultados intermedios de los *mappers* y se envían a los *reducers*.
3º. **Sort:** Fase interna donde se ordena la salida de la etapa de *shuffle* para optimizar el procesamiento por parte de la tarea de *reduce*.
4º. **Reduce:** Tarea programable que calcula el conjunto de resultados final a partir de los datos ordenados.

Ventajas de MapReduce:

- Paralelización y distribución de tareas automática.
- Tolerante a fallos: puede reasignar tareas incompletas a otros nodos si se produce un fallo.
- Escalable.
- Flexibilidad de programación (soporta Java, Python, C#, Ruby, C++).
- Abstracción: El desarrollador no tiene que preocuparse por la ubicación real de los datos o los detalles de la infraestructura.

Analogía para HDFS, Hadoop y MapReduce:

Podríamos ver a **Hadoop** como una gran **Planta de Procesamiento de Ladrillos Digitales**.

El **HDFS** es el **Patio de Almacenamiento (Distributed FileSystem)** de esta planta. Los archivos grandes (las montañas de arcilla) se rompen en pedazos pequeños, estandarizados (los **Bloques** de 64 MB o más). Para que el almacén sea a prueba de fallos, cada bloque se copia varias veces (**Replicación**) y se guarda en diferentes estanterías o DataNodes. El **NameNode** es el bibliotecario maestro que sabe exactamente dónde está cada bloque, sus copias y cómo se unen para formar el archivo original, aunque nunca toca la materia prima, solo administra el inventario (los metadatos).

MapReduce es el **Proceso de Fabricación Estandarizado**. Es la maquinaria de ensamblaje distribuida que toma la materia prima (los bloques) y realiza el trabajo pesado en paralelo. La fase **Map** distribuye las instrucciones a los trabajadores (TaskTrackers) para que clasifiquen y transformen los ladrillos a pie de obra. La fase **Reduce** recolecta los resultados parciales de todos los trabajadores, los combina y produce el producto final o informe deseado.

El **Framework Hadoop** es el **Sistema Operativo** de la planta, que contiene tanto el patio de almacenamiento (HDFS) como la maquinaria (MapReduce) y se encarga de que todo funcione de manera eficiente y distribuida.

Etapas de un trabajo en el paradigma MapReduce

Son un proceso secuencial y distribuido diseñado para manipular grandes volúmenes de datos. Un *Job* de MapReduce se divide en **cuatro fases**, siendo *Map* y *Reduce* las tareas que deben ser programadas por el usuario, mientras que *Shuffle* y *Sort* son fases internas de la ejecución.

A continuación, se detallan las cuatro etapas clave de un trabajo MapReduce:

Map (Mapeo):

- **Definición:** Es la primera tarea programable que se debe realizar para una aplicación de MapReduce.
- **Función:** Los datos de entrada son procesados, uno a uno, y transformados en un **conjunto intermedio de datos**.
- **Proceso:** Se llama a una instancia distinta de la función `map` para procesar los datos por cada par de entrada.
- **Resultado:** La función `map` aplica una función a cada elemento de una lista de entrada (definido como un par clave-valor) y genera una nueva lista de salida. El componente *map* distribuye el problema o las tareas de programación entre un gran número de subprocesos.
- **Continuidad:** Una vez que las tareas de mapeo finalizan, envían sus resultados a una partición específica como entradas para las tareas de reducción.

Shuffle (Mezcla):

- **Naturaleza:** Es una fase interna en la ejecución del *job*.
- **Función:** Se encarga de la **recopilación y copia** de los resultados intermedios generados por las tareas de *Map*.
- **Mecanismo:** El mecanismo que realiza esta acción se conoce como **«mezcla y ordenación»** (*shuffle* y *sort* en conjunto).
- **Transferencia:** Los datos intermedios se copian a través de la red a los nodos reductores tan pronto como se generan. Todos los valores de la misma clave se envían al mismo reductor, lo que ayuda a garantizar un mayor rendimiento y eficiencia.

Sort (Ordenación):

- **Naturaleza:** Es una fase interna en la ejecución del *job*.

- **Función:** El proceso de «mezcla y ordenación» recopila y **prepara todos los datos mapeados** para su reducción.
- **Proceso:** Una vez que los resultados intermedios se recopilan en la partición, se realiza una reorganización para **ordenar la salida** y optimizar su procesamiento por parte de la tarea de reducción.

Reduce (Reducción):

- **Definición:** Es la segunda tarea programable que debe ejecutarse.
- **Requisito:** La función *reduce* no puede comenzar hasta que se complete la fase de asignación (mapeo).
- **Función:** Una vez que se reúnen los resultados intermedios (ya ordenados), la función *reduce* se aplica para **reducirlos a un conjunto de datos resumidos**, que constituye el resultado final de la tarea. La función *reduce* agrega todos los elementos resultantes del cálculo distribuido para obtener un resultado.
- **Salida:** La salida de la reducción también es un par clave y valor y la última tarea es **escribir los datos en HDFS** (Hadoop Distributed FileSystem).

Clase 03 - Spark; RDD y Dataframes; SQL

I. Apache Spark

Origen y Evolución

- Spark nació en 2009 en los laboratorios de la Universidad de California.
- Desde 2013, pertenece a la Fundación Apache.

Capacidades Clave y Velocidad

- Soporta los dos modos de trabajo principales en Big Data: consultas en grandes volúmenes de datos (*batch processing*) y procesamiento de *streaming*.
- Está optimizado para trabajar en RAM.
- Según los *benchmarks*, Spark es 100 veces más rápido que Hadoop MapReduce.
- El sistema opera bajo el modelo *master-slave*, donde el módulo principal, denominado driver, se ejecuta en el *master* y es el encargado de enviar las operaciones a realizar sobre los RDDs a cada nodo del *cluster*.
- Se conecta al *cluster* mediante un objeto denominado SparkContext.

SparkContext y Conexión al Cluster

- El SparkContext es el objeto que permite la conexión con el clúster y es fundamental para el funcionamiento de Spark.
- El SparkContext crea y maneja los RDDs.
- Es el punto de entrada para cualquier aplicación Spark.

Spark Shell y Entornos de Trabajo

- De manera nativa, las aplicaciones en Spark se programan en Java.
- Spark incluye shells interactivos para Python (PySpark) y Scala (spark-shell):
- PySpark: El shell crea automáticamente un SparkContext y lo deja disponible en una variable llamada `sc`.
- Scala: Utiliza el shell `spark-shell`.
- Al ejecutar scripts fuera del shell, se debe crear el SparkContext de manera explícita:

```
from pyspark import SparkContext
sc = SparkContext("local", "MyProgram")
```

Los scripts se ejecutan con el comando `spark-submit` de forma secuencial, aunque las operaciones internas se ejecutan en paralelo de manera transparente para el usuario.

Integración y Almacenamiento

- Spark fue desarrollado para trabajar con el HDFS de Hadoop.
- Permite la integración con otros medios de almacenamiento, incluyendo HBase, Cassandra, MongoDB y Amazon's S3.

Módulos de Arquitectura (*Librerías*)

Una aplicación Spark solo requiere Spark Core (manejo de funciones y tareas de *scheduling*) más una única librería, aunque es posible crear aplicaciones más potentes usando más de una.

Otros módulos clave incluyen:

- Spark SQL: Módulo para trabajar con datos estructurados.
- Spark Streaming: Optimizado para el trabajo con flujos de datos y diseñado para alimentarse de fuentes como Apache Kafka y Apache Flume.

- MLlib: Librería de *machine learning* que contiene algoritmos de clasificación, regresión y *clustering*. Sus algoritmos están implementados para ejecutarse de manera eficiente en un ambiente distribuido.
- GraphX: Librería para el análisis sobre grafos de datos, con algoritmos diseñados para ejecutarse de manera eficiente en un ambiente distribuido.

II. RDDs (Resilient Distributed Datasets) y DataFrames

A. Resilient Distributed Datasets (RDDs)

Definición y Propósito:

- Las RDDs son la base o **núcleo de Spark**.
- Son bases de datos distribuidas y elásticas.
- Su diseño está optimizado para el **almacenamiento de los datos en la RAM** de forma distribuida en un *cluster*.

Características de Resiliencia:

- Son **tolerantes a fallas**, ya que el sistema realiza el *tracking* de las diferentes operaciones realizadas.
- Son **inmutables**.
- Se dividen en **particiones** que se distribuyen entre los nodos de un *cluster*.

Creación y Operaciones:

- Se construyen a partir de la lectura del contenido de un archivo o base de datos, o mediante la paralelización de colecciones de datos.
- Una vez cargados los datos en una RDD, se pueden realizar dos tipos de operaciones básicas, que son parte de la API de Spark:
    1. **Transformaciones:** Cambian el RDD original a través de procesos como mapeo o filtrado.
    2. **Acciones:** Cálculos que se realizan sobre una RDD sin modificarla, como el conteo o las sumas.

B. DataFrames

Relación con RDDs

- Los DataFrames son una **abstracción de los RDDs**.
- Un DataFrame es esencialmente un **RDD con esquema**.

Estructura

- Consisten en **tuplas de datos con nombres y tipo de datos** para cada campo, lo que implica que tienen un esquema definido.
- Estos DataFrames son **distribuidos**.
- Se crean mediante el `sqlContext`.

III. Spark SQL y el Lenguaje SQL

Spark SQL

- Es el módulo de Spark diseñado para trabajar específicamente con **datos estructurados**, es decir, aquellos que poseen un esquema.
- Utiliza la abstracción de **DataFrames** (RDDs con esquema) para su procesamiento.
- Permite el uso del **lenguaje SQL** para realizar tareas de selección, filtrado, agrupación y *joins*.
- Permite cargar datos de formatos variados, incluyendo JSON, Hive, Parquet, ODBC y JDBC.
- Puede integrarse con HiveQL.
- Facilita la integración entre el lenguaje SQL y lenguajes de programación como Python, Scala o Java.

SQL (Structured Query Language):

- Es un lenguaje de consulta estructurado.
- Originalmente, el modelo relacional añadió un nivel de abstracción con SQL, facilitando a los programadores extraer valor de los datos.
- SQL es el mecanismo más utilizado para crear, consultar, mantener y operar bases de datos relacionales (RDBMS).
- Se utiliza para las operaciones CRUD (crear, recuperar, actualizar y eliminar) en bases de datos relacionales.
- A lo largo de los años, su popularidad ha llevado a que se extienda su uso incluso a algunas bases de datos no relacionales.
- En el ecosistema Hadoop, existe un lenguaje similar a SQL llamado HiveQL, utilizado por Hive, el cual soporta operaciones como SELECT, JOIN, y AGREGATED.
- En un entorno tradicional de base de datos relacional, las tablas se pueden consultar mediante una clave común, como `CustomerID`.

Analogía para la Pila Spark

Si el Big Data es un gigantesco almacén lleno de cajas (datos), **Spark** es un **robot de almacén de alta velocidad** diseñado para clasificar, mover y analizar esas cajas mucho más rápido que los sistemas antiguos (Hadoop

MapReduce). Los **RDDs** son como los **contenedores temporales en la memoria** del robot, inmutables y distribuidos, que aseguran que si una pieza del robot falla, el trabajo no se pierde. Un **DataFrame** es ese mismo contenedor, pero con **etiquetas y un manifiesto (esquema)**, lo que permite al robot y a los analistas hablar el lenguaje estándar de inventario (**SQL**) para encontrar y procesar exactamente lo que necesitan sin tener que revisar cada caja manualmente.

Clase 04 - Spark: Streaming; MLlib

Spark Streaming

Es una extensión del *framework* Spark Core diseñada para el procesamiento de flujos de datos.

Modo de Trabajo

- Spark soporta el **procesamiento de *streaming***, además de las consultas en grandes volúmenes de datos (*batch processing*).
- Spark Streaming está optimizado para trabajar con flujos de datos.
- Técnicamente, **Spark streaming no es 100% *streaming***.
- Por motivos de eficiencia y compatibilidad, **Spark streaming guarda el flujo (*stream*) en pequeños "trozos" (*chunks*)**.
- Estos trozos se ejecutan como **pequeños procesos *batch* (micro-batch)**.
- En Spark, un *stream* se representa como un **flujo discreto (DStream)**.
- Un **DStream es una secuencia de RDDs** (Conjuntos de Datos Distribuidos y Elásticos).
- Cada RDD es una **instantánea (*snapshot*) de todos los datos recopilados durante un período de tiempo**, el cual luego se procesa como un *batch*.

Características del Flujo de Datos

- El flujo de datos es **continuo**.
- La frecuencia de la llegada de los datos depende del problema.
- Los datos se recolectan **en tiempo real**.
- Fuentes:
    - Redes Sociales: Twitter, Facebook, Instagram.
    - Flujos de transacciones: Bancarias o criptomonedas (Bitcoin).
    - Monitoreo de redes: Detección de intrusiones en la red, logs de servidores.
    - Monitoreo en tiempo real de sensores + Internet de las Cosas (IoT).
    - Análisis climático
    - Análisis de información generada por dispositivos wearable.

- El modelo de datos puede basarse en recibir, utilizar y **descartar el dato**, o en usar una **ventana temporal para guardar los últimos *n* datos recibidos**.
- Ventanas de tiempo: Landmark Window - Sliding Window - Fading Window (Damped Window) - Tilted Time Window.
- Los algoritmos de *streaming* se utilizan generalmente como **clasificadores**.
- Un modelo de *streaming* puede estar **entrenado de antemano** y usarse sobre el flujo, o puede **entrenarse con el propio *streaming***, permitiéndole seguir entrenando y actuar como predictor al mismo tiempo.

Fuentes y Requisitos de Rendimiento

- Spark Streaming está diseñado para alimentarse de varias fuentes de datos, incluyendo **Apache Kafka** (mensajería distribuida), **Apache Flume** (gestor de *logs*), Amazon Kinesis, Twitter, y **sensores u otros dispositivos vía *sockets* TCP**.
- Un algoritmo de *streaming* debe cuidar tres aspectos fundamentales: **Velocidad** (debe poder operar un nuevo dato en el menor tiempo posible), **Memoria** (debe ocupar la menor cantidad de memoria RAM), y **Eficacia** (debe clasificar nuevos datos con la mayor eficacia posible).

MLlib (Machine Learning Library)

MLlib es el módulo de Spark dedicado a la implementación de algoritmos de *machine learning* (aprendizaje automático).

Definición y Propósito

- **MLlib es la librería de algoritmos de *machine learning* para Spark**.
- Los algoritmos de MLlib están diseñados e implementados para **ejecutarse de manera eficiente en un ambiente distribuido**.
- Los algoritmos que implementan los sistemas inteligentes son **algoritmos iterativos**.
- Estos algoritmos iterativos requieren dar varias "pasadas" a los datos para llevar a cabo su tarea.
- Deben estar **optimizados para un óptimo rendimiento**.

Algoritmos Incluidos

- MLlib contiene algoritmos de **clasificación, regresión, y *clustering***.
- La librería incluye una amplia variedad de algoritmos, tales como:
    - **Regresión:** Regresión logística (*Logistic regression*), Regresión lineal generalizada (*Generalized linear regression*), Regresión de supervivencia (*Survival regression*).

- **Clasificación y Bosques:** Árboles de decisión (*Decision trees*), Bosques aleatorios (*Random forests*), Árboles impulsados por gradiente (*Gradient-boosted trees*), Naive Bayes.
- **Clustering:** K-means, Mezclas gaussianas (*Gaussian mixtures*), Asignación de Dirichlet latente (*Latent Dirichlet allocation - LDA*).
- **Otros:** Mínimos cuadrados alternos (*Alternating least squares - ALS*), Conjuntos de elementos frecuentes (*Frequent itemsets*), Reglas de asociación (*Association rules*), Minería de patrones secuenciales (*Sequential pattern mining*).

III. GraphX (Graph Processing Library)

Definición y Propósito

- GraphX es la librería de Spark dedicada al análisis y procesamiento de grafos de datos.
- Los algoritmos de GraphX están diseñados e implementados para ejecutarse de manera eficiente en un ambiente distribuido.
- Es uno de los módulos clave de la arquitectura de Spark, junto con Spark Core, Spark SQL, Spark Streaming y MLlib.

Algoritmos Incluidos

GraphX contiene una variedad de algoritmos especializados para el análisis de grafos:

- **PageRank**: Algoritmo para medir la importancia de los nodos en un grafo basándose en sus conexiones.
- **Connected components**: Identifica componentes conectados dentro de un grafo.
- **Label propagation**: Algoritmo de detección de comunidades que propaga etiquetas a través de la red.
- **SVD++**: Versión mejorada de la descomposición en valores singulares para sistemas de recomendación.
- **Strongly connected components**: Identifica componentes fuertemente conectados en grafos dirigidos.
- **Triangle count**: Cuenta el número de triángulos en el grafo, útil para análisis de redes sociales y detección de comunidades.

Aplicaciones

- Análisis de redes sociales
- Sistemas de recomendación
- Detección de fraudes

- Análisis de relaciones y patrones en datos conectados

Clase 05 - Containers y Docker

La Solución al Problema de Entorno

La motivación para usar *Containers* radica en el problema clásico de incompatibilidad de entornos, resumido en la frase: **"En mi máquina funciona..."**. Este problema ocurre cuando el código de una aplicación pasa de un entorno de desarrollo a otro de ejecución, como producción, y falla debido a las diferencias entre los contextos. Un entorno es el contexto y lugar donde se ejecuta el código programado, pudiendo ser una máquina virtual en la nube (AWS, GCloud, Azure, etc.) o el dispositivo de cómputo del desarrollador.

Los contenedores surgen como la respuesta a si es posible empaquetar la aplicación con **todas sus dependencias**.

¿Qué es un Container (Contenedor)?

Un contenedor es una **unidad de software estandarizada** que incluye todas las dependencias necesarias para su ejecución en **cualquier entorno**. Es un **proceso en ejecución** y, por lo tanto, tiene un estado (ejecución, parado, etc.).

Características Principales:

1. **Ligeros:** Solo incluyen los archivos que son necesarios para su funcionamiento. Su tamaño suele medirse en **megabytes (MB)**, a diferencia de las máquinas virtuales que se miden en gigabytes (GB). Son ligeros porque comparten muchos componentes del **Kernel de Linux**, y solo adicionan los archivos de la aplicación en ejecución; existen contenedores de apenas 5 MB (como *busybox*).

2. **Portables:** Pueden empaquetarse en un archivo y distribuirse de forma rápida y sencilla a cualquier entorno, sin necesidad de dependencias adicionales para su ejecución en destino.

3. **Procesos Aislados:** Los procesos dentro de un contenedor están aislados del resto de los procesos de la máquina que los aloja. Utilizan los recursos del Kernel de Linux para funcionar de forma aislada.

Contenedores vs. Imágenes

Existe una relación estrecha entre las imágenes y los contenedores.

• **Imágenes:** Los contenedores se crean a partir de ellas. Una imagen es una unidad de software estandarizada compuesta por múltiples archivos. La diferencia clave con un contenedor es la **ausencia de un estado**; las

imágenes no son procesos, sino solo archivos, por lo que no pueden estar iniciadas, detenidas o pausadas. Las imágenes pueden ser movidas de una máquina a otra o transferidas a un almacenamiento en la nube.

• **Contenedores:** Se inician muchos contenedores a partir de una única imagen.

Contenedores vs. Máquinas Virtuales (VMs)

Es importante notar que los contenedores **no son máquinas virtuales ligeras**. Las diferencias radican en el aislamiento, el espacio en disco, la construcción y la velocidad de inicio:

Característica	Contenedores	Máquinas Virtuales (VMs)
Aislamiento	Procesos ligeros aislados que utilizan recursos del **Kernel de Linux**. No dependen de herramientas de virtualización.	Sistemas operativos funcionando **completamente aislados** dentro de la máquina local.
Espacio en Disco	Tamaño menor, medido en **MB**. Solo contienen los archivos necesarios para la aplicación.	Estructuras pesadas, medidas en **GB**, ya que necesitan un sistema operativo completo para su funcionamiento.
Construcción	Se construyen con los archivos necesarios para ejecutar **solo la aplicación** (a menudo resolviendo una única necesidad, como los microservicios).	Se crean a partir de la instalación de un sistema operativo para **múltiples propósitos**, pudiendo ejecutar muchos servicios al mismo tiempo.
Velocidad	Mucho más rápido de iniciar, detener o reiniciar.	Más lento, ya que debe cargar un sistema operativo completo.

¿Qué es Docker?

Docker es una tecnología de contenerización **open source** utilizada para construir aplicaciones. **Docker Engine** funciona bajo un **esquema cliente-servidor** y consta de:

• **Servidor:** Conocido como dockerd (daemon). Es el proceso que realiza las operaciones en la máquina y tiene la responsabilidad de gestionar contenedores.

• **Cliente:** Proporciona la interfaz de comandos (docker) y controla e interactúa con el daemon dockerd mediante una API.

• **API's:** Actúan como la interfaz entre el cliente y el servidor. La comunicación se realiza mediante **API REST**, utilizando sockets UNIX o interfaces de red.

El conjunto de cliente y servidor se denomina **Docker Engine**. Docker se instala como cualquier otro programa, y la cantidad de contenedores que se pueden gestionar es proporcional a los recursos de CPU y RAM de la máquina host. Ambos elementos (cliente y servidor) pueden estar instalados en la misma máquina o conectarse a un servidor dockerd remoto.

Metáfora para Diferenciar Contenedores y Máquinas Virtuales:

Si la computación distribuida es como administrar una flota de camiones de reparto, una **Máquina Virtual (VM)** es un camión completamente equipado que transporta su propia cabina, motor, combustible y conductor (un sistema operativo completo y sus recursos). Un **Contenedor**, en cambio, es solo el remolque de carga estandarizado: es ligero, se acopla a la perfección a la cabina y motor ya existentes (el Kernel del host), y solo contiene la mercancía necesaria (la aplicación y sus dependencias), permitiendo un despliegue y una gestión mucho más rápidos y eficientes.

Clase 06 - Kafka

Es una plataforma de *streaming* distribuida, escalable y tolerante a fallas, utilizada para **almacenar y procesar flujos de datos**. Se define específicamente como un sistema de **mensajería distribuida**. Es una herramienta esencial en la pila tecnológica de Big Data, especialmente optimizada para el **procesamiento de *streaming*** (flujos de datos continuos en tiempo real).

Componentes Clave

1. Kafka Clúster

- Es la parte central que **almacena los flujos de datos**.
- Provee un **servicio de mensajes suscriptor-publicador**.
- Se compone de servidores llamados **Brokers**, donde se almacenan los mensajes.
- Los *Brokers* tienen la capacidad de almacenar mensajes de forma **confiable en disco** y no los eliminan inmediatamente después de la entrega, a diferencia de otros sistemas de mensajes. Esto requiere configurar la **política de retención** (por tiempo o por tamaño en bytes) de los tópicos.
- **No existe un nodo maestro** central, lo que permite un escalamiento lineal y evita cuellos de botella.
- Utiliza **Apache ZooKeeper** como servicio de coordinación para almacenar metadatos (como información sobre tópicos y particiones) y asegurar la coordinación entre *Brokers*.

2. Connect API (Framework de Conexión)

- Se utiliza para **consumir datos en el contexto de Kafka**.

- También permite la **exportación de flujos de datos** a sistemas externos, como bases de datos o sistemas de archivos distribuidos/paralelos.

3. Streams API (Librería de Programación para Streams)

- Facilita el **procesamiento de flujos de datos**.
- Permite la creación de ***pipelines* de procesamiento complejos** que reciben y escriben *streams* desde y hacia el clúster Kafka.

Estructura y Funcionamiento del Clúster

Productores (Producers)

- Son las aplicaciones que **crean, escriben y publican mensajes** en los *Brokers* del clúster.
- Deben especificar la **partición** a la que se envía el mensaje.
- Generalmente, el particionamiento por defecto utiliza un **algoritmo de *hashing* por clave**, lo que asegura que los mensajes con la misma clave compartan la misma partición.

Consumidores (Consumers)

- Son las aplicaciones que **leen los mensajes** del clúster.
- Los consumidores son **responsables de realizar el seguimiento de los mensajes leídos/procesados**, lo cual se logra almacenando los ***offsets*** (posiciones) de los mensajes.

Tópicos (Topics)

- Los mensajes se organizan en tópicos, los cuales son formalmente una **secuencia de mensajes (log)**.
- Los mensajes se añaden secuencialmente **al final del tópico**.

Particiones (Partitions)

- Los tópicos se dividen en particiones.
- **Kafka garantiza el orden de los mensajes ÚNICAMENTE dentro de una partición**. No existe garantía de orden entre mensajes de diferentes particiones o tópicos.
- El **Offset** es la posición de un mensaje dentro de su partición, determinada implícitamente a medida que se añaden mensajes.
- Para garantizar la **tolerancia a fallos y alta disponibilidad**, las particiones se replican en diferentes *Brokers*.

Mensajes

- Se definen bajo el esquema **clave-valor**, siendo las claves y los valores arreglos de bytes de tamaño variable. Cada mensaje incluye también un *timestamp* (marca de tiempo) de 64 bits.

Ventajas de la Retención de Mensajes

La capacidad de los *Brokers* para no eliminar mensajes después de la entrega permite que los consumidores lean mensajes la cantidad de veces que sea necesario y un **mayor rendimiento en operaciones de lectura**.