Facultad de Ciencias

Análisis de rendimiento "*Delta Lake vs Parquet*" en procesos ETL sobre un clúster de Spark

"Delta Lake vs Parquet" performance analysis in ETL processes on a Spark cluster

Trabajo de Fin de Máster
para acceder al

MÁSTER EN CIENCIA DE DATOS

Autor: Ángel Gómez Fernández - Cavada

Directora: Aida Palacio Hoz

Resumen

Se realiza un análisis, y comparativa, en tiempos de lectura y escritura, de procesos de carga ETL (*batch*) sobre ficheros *Parquet* y *Delta Lake*, empleando dos tipos de almacenamiento: *Block Storage* y *Object Storage*. Todo esto se realiza sobre un clúster Hadoop + Spark desplegado sobre la plataforma de *Cloud Computing* OpenStack instalada en el Instituto de Física de Cantabria (IFCA), en el cual se han configurado las diferentes máquinas empleando un nodo auxiliar y roles de Ansible.

El objetivo de este proyecto es estudiar el rendimiento de ambos tipos de extensión de fichero para entornos Big Data y poder deducir a futuro en qué situaciones es mejor emplear uno u otro.

Abstract

An analysis and comparison, in read and write times, of ETL (batch) loading processes on Parquet and Delta Lake files is carried out, using two types of storage: Block Storage and Object Storage. All this is done on a Hadoop + Spark cluster deployed on the OpenStack Cloud Computing platform installed at the Institute of Physics of Cantabria (IFCA), in which the different machines have been configured using an auxiliary node and Ansible roles.

The objective of this project is to study the performance of both types of file extension for Big Data environments and to be able to deduce in the future in which situations it is better to use one or the other.

Capítulo 1 -Introducción

1.1 Motivación

En la actualidad el tratamiento de los datos, sobre todo de los datos masivos, se encuentra en un punto álgido en el que cualquier empresa, o institución, quiere y debe estar a la vanguardia de las novedades que surgen.

Aunque muchas veces no seamos conscientes, vivimos rodeados de datos. Las nuevas tecnologías han propiciado un aumento exponencial de la cantidad de datos que producimos y enviamos (un buen ejemplo pueden ser los millones de *WhatsApps* que se envían por minuto a nivel mundial), pero no solo basta con contabilizar esos datos, sino que es necesario almacenarlos, gestionarlos e interpretarlos para poder sacarles valor.

Cada día convivimos con más datos, y datos cada vez más relevantes o sensibles, como pueden ser datos médicos, meteorológicas, bancarios, etc., que por sí mismos requieren cada vez mejores arquitecturas, análisis, sistemas más eficientes y, en definitiva, que las tecnologías de la información avancen y evolucionen, y proporcionen soluciones óptimas a estas necesidades.

Por otro lado, esto es un arma de doble filo, ya que, en un sector con tanta expansión y evolución, las empresas crean y promocionan estas nuevas soluciones desde el mejor punto de vista posible para ellas, visibilizando el contexto en que mejor funciona dicha solución, y a veces generan falsas expectativas que es necesario de contrastar con hechos y resultados. De ahí surge la necesidad de realizar pruebas de concepto (PoCs) cuando aparecen estas nuevas alternativas, para verificar si realmente suponen un avance y son de utilidad, o en qué situaciones lo son y en qué situaciones no.

1.2 Objetivos

Los objetivos de este Trabajo de Fin de Master son académicos y de investigación. No solo se basa en el conocimiento teórico de las tecnologías, *frameworks*, infraestructuras, etc., implicadas en el mundo *Big Data* y del trabajo que llevaría a cabo un ingeniero de datos, sino también de llevar a cabo un PoC (*proof of concept*) del análisis del rendimiento de nuevas extensiones de ficheros de datos masivos con otras extensiones más

estandarizadas en este sector, además de cómo afecta su sistema de almacenamiento en éstas.

Los objetivos principales que se quieren abordar son:

- Creación de un clúster Hadoop + Spark sobre OpenStack.
- Despliegue del clúster de manera automática.
- Conocimiento de las extensiones de almacenamiento *Parquet* y *Delta Lake.*
- Conocimiento y uso del *framework* Spark.
- Uso de la interfaz Pyspark para Apache Spark en Python.
- Uso y conocimiento de diferentes tipos de almacenamiento (*Block Storage* y *Object Storage*).

1.3 Metodología

Para llegar a los objetivos que se acaban de exponer, ha sido necesario realizar las siguientes tareas a lo largo del proyecto:

1) Revisar documentación sobre fundamentos teóricos en las siguientes temáticas:
    a) Apache Hadoop
    b) Apache Spark
    c) Delta Lake
    d) OpenStack y Ansible
    e) Tipos de almacenamiento
2) Diseño de la arquitectura y de los casos de prueba.
3) Montaje del clúster y entorno de prueba.
4) Obtención del conjunto de datos a explotar.
5) Análisis de los resultados y conclusiones.

Además, este proyecto se ha estructurado según los siguientes capítulos:

- **Capítulo 2:** Se explican los fundamentos teóricos que son necesarios conocer para poner en contexto el proyecto. Además, se habla del estado del arte en este sector.
- **Capítulo 3:** Se explica la arquitectura montada, y los casos de prueba y medidas que se han tomado posteriormente sobre ella.

- **Capítulo 4:** Contiene los resultados obtenidos tras ejecutar los diferentes *jobs* de *pyspark*, permitiendo exponer la comparativa de las diferencias de rendimiento.
- **Capítulo 5:** Se exponen las conclusiones obtenidas del proyecto, así como la continuación y ampliación que se podría hacer de él.

Capítulo 2 – Estado del arte y fundamentos teóricos

En este capítulo se pretende describir, de una forma sintetizada y concisa, aquellos fundamentos y conceptos previos que se deben comprender para seguir de forma adecuada el desarrollo de esta PoC, así como el estado del arte de las tecnologías y procesos asociados a este ámbito.

Cuando se habla de *Big Data* es inevitable hablar de Apache Hadoop y Apache Spark, como los *frameworks* más estandarizados dentro de la industria, y que todas las empresas e instituciones que trabajan en esta línea conocen y utilizan. Estas tecnologías son más sencillas de desplegar y configurar que otras, como por ejemplo Kubernetes, además de que son, como se ha comentado, mucho más extendidas dentro de la comunidad, por lo que hay más información y soporte frente a los diversos errores que puedan surgir que por ejemplo con tecnologías como Apache Storm, Hydra o Google BigQuery.

Lo más común al utilizar Hadoop y Spark es emplear, para el almacenamiento de los datos, HDFS (*Hadoop Distributed File System*) como sistema de almacenamiento por defecto, y los ficheros *parquet* como la extensión predeterminada.

Aunque lo más extendido sea utilizar HDFS y *parquet* como se acaba de comentar, en la actualidad han surgido nuevas alternativas y posibilidades, como puede ser *Delta Lake*, una capa de almacenamiento *open source* que pretende dar un punto de vista diferente al almacenamiento distribuido de los datos y metadatos.

Por último, estos *frameworks*, sistemas de almacenamiento, extensiones de ficheros, etc., están pensados y diseñados para una computación distribuida y un gran volumen de información, como ya se ha comentado, y para exprimir el máximo de estos procesos se utilizan los recursos de un clúster. En esta PoC, partiendo de los recursos que nos proporciona el clúster de *Cloud Computing* del CPD (Centro de Procesamiento de Datos) del IFCA [1], se levantará un pequeño clúster con OpenStack para hacer las pruebas de rendimiento, en el cuál se desplegarán los diferentes *frameworks* mediante roles de Ansible.

2.1 Apache Hadoop

Apache Hadoop es un *framework* que permite la computación distribuída, nacido en 2004, con la finalidad de manejar grandes volúmenes de datos en grupos de computadores (clústers) utilizando modelos de programación simples. Este *framework* está diseñado para ser escalable, pudiendo estar desde un servidor individual hasta clústers de miles de computadores, y se compone de los siguientes módulos como se ve en la **Figura 1**:

- **Hadoop Common:** Bibliotecas y utilidades necesarias para otros módulos de Hadoop.
- **HDFS (*Hadoop Distributed File System*):** Sistema de archivos distribuido que proporciona acceso de alto rendimiento a los datos de la aplicación.
- **Hadoop YARN:** Software responsable de administrar los recursos en el clúster.
- **Hadoop MapReduce:** Paradigma o modelo de programación basado en YARN para el procesamiento en paralelo de grandes conjuntos de datos.

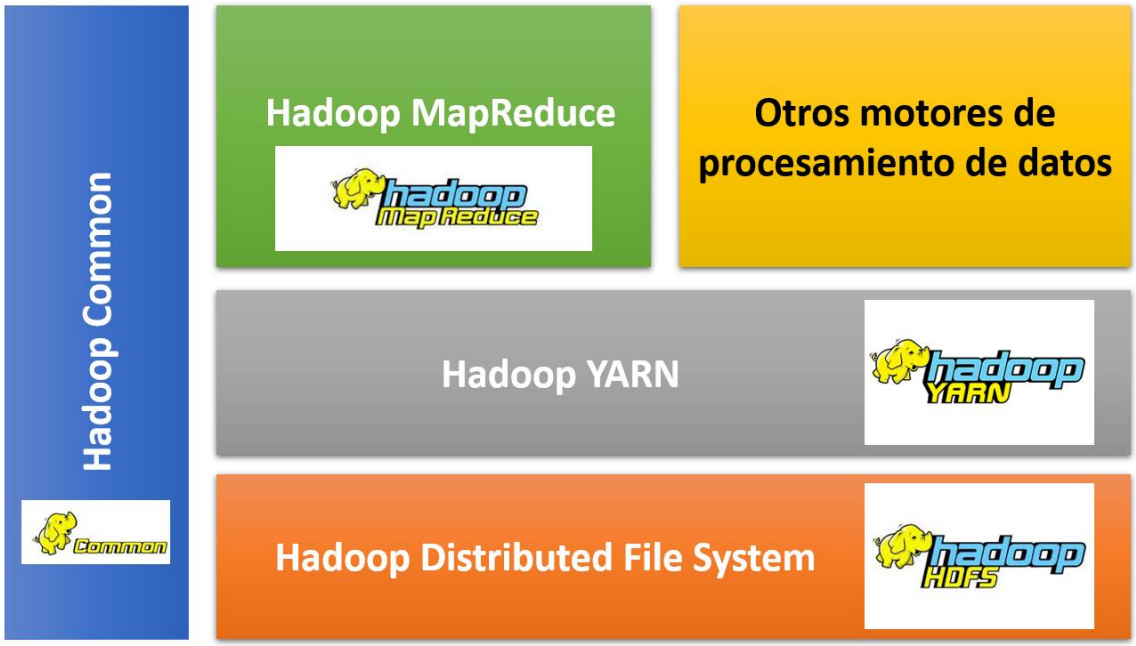

Figura 1. Gráfico resumen de los diferentes módulos de Apache Hadoop.

Dentro del ecosistema Hadoop, cabe mencionar el Apache Parquet, o más conocido simplemente como ***parquet***, el cual es un formato de almacenamientos de datos en columnas, independiente del tipo de *framework* de procesamiento de datos, modelo de datos o lenguaje de programación escogido.

2.2 Apache Spark

Apache Spark es un *framework* de computación distribuida para procesar grandes volúmenes de datos. Originalmente fue desarrollado por AMPLab en la Universidad de Berkeley en California, en 2009. Los creadores fundaron *Databricks* (empresa muy conocida mundialmente en la actualidad) para comercializar Spark. Más tarde, se convertiría en proyecto *open source* de Apache.

Al contrario que muchos *frameworks* relacionados con Hadoop, no utiliza el motor *MapReduce*, sino que utiliza su propio motor de ejecución distribuido para ejecutar trabajos en un clúster. Sin embargo, está muy integrado con el ecosistema Hadoop, ya que es muy común que Spark se ejecute sobre YARN y sobre el sistema de ficheros HDFS.

Spark es conocido por su capacidad para mantener en memoria grandes conjuntos de datos entre trabajos. Esta capacidad permite a Spark superar el rendimiento (incluso en 100 veces) respecto a *MapReduce*.

Spark proporciona APIs para escribir programas en varios lenguajes: Scala, Java, Python, R y SQL; y ha demostrado ser una buena plataforma para construir herramientas de análisis. Para ello ofrece módulos de *machine learning* (MLlib), procesamiento de grafos (GraphX), *streaming* (Spark Streaming) y SQL (Spark SQL). Además, Spark permite la integración con muchos sistemas gestores de bases de datos (relacionales y no relacionales) como se puede ver en la **Figura 2.**

Figura 2. Gráfico que muestra algunas de las integraciones que permite el *framework* Apache Spark con sistemas gestores de bases de datos. [4]

2.3 Delta Lake

Delta Lake es un proyecto *open source* que permite la construcción de arquitecturas ***Lakehouse*** sobre *Data Lakes* (lagos de datos) con motores de computación como por ejemplo Spark.

La arquitectura ***Lakehouse*** se define como un sistema de gestión de datos basado en un almacenamiento *low-cost* y de acceso directo, que, además, ofrece funciones tradicionales de los SGBD como son las transacciones ACID, versionado de datos, auditoría, indexación, almacenamiento en caché y optimización de consultas. Los *Lakehouses* buscan fusionar la facilidad de acceso y el soporte para las capacidades de análisis empresarial que se encuentran en los *Data Warehouses* con la flexibilidad y el coste relativamente bajo de los *Data Lakes*.

La arquitectura de los *Data Warehouse* se desarrolló en la década de 1980 para ayudar a las empresas en su proceso de toma de decisiones. Los datos son consumidos por herramientas de BI, donde los ejecutivos y otro personal pueden visualizar y analizar datos en el formato de informes y gráficos.

Con la llegada del *Big Data*, las arquitecturas tradicionales como los *Data Warehouse* tuvieron que repensarse. Con datos provenientes de diferentes fuentes, en diferentes formatos y, por lo general, en un volumen mayor, era necesario que surgiera un nuevo paradigma para llenar este vacío. En un lago de datos, los datos se almacenan en su formato sin procesar y solo se consultan cuando surge una pregunta, recuperando datos relevantes que luego se pueden analizar para ayudar a responder la pregunta.

Finalmente, surge la arquitectura *Lakehouse*, donde se intenta unificar y explotar los puntos fuertes de ambos predecesores, llegando hasta donde nos encontramos actualmente con este nuevo tipo de arquitectura y nuevos tipos de extensiones de fichero como *Delta*.

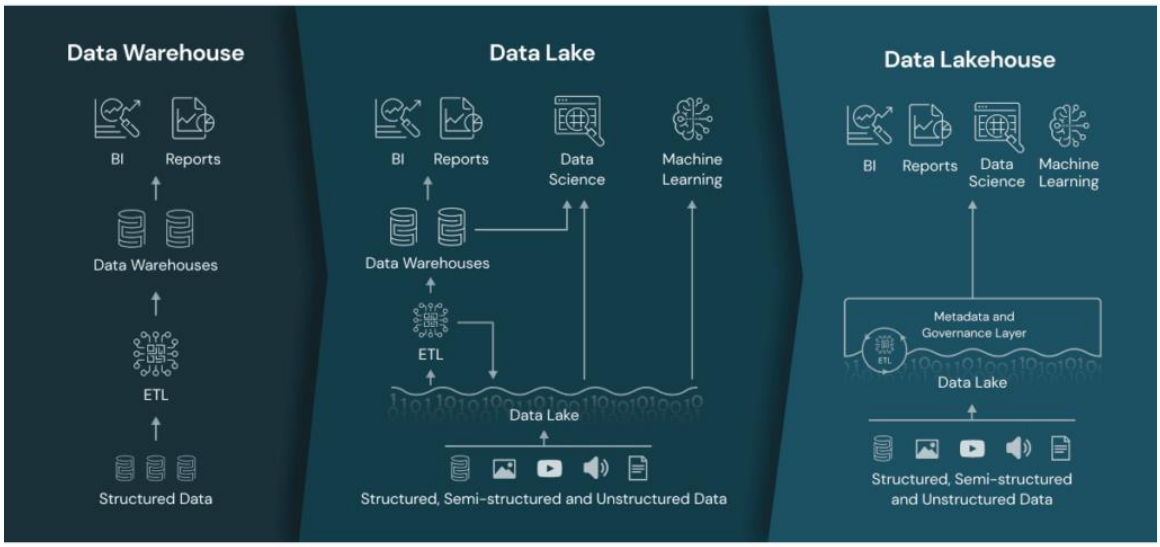

Figura 3. Evolución de las arquitecturas de plataformas de datos hasta el modelo actual *Lakehouse*. [4]

Las características principales de *Delta Lake* son:

- **Transacciones ACID:** Se asegura la atomicidad, consistencia, aislamiento y durabilidad de las transacciones procesadas.

- **Metadatos escalables:** Aprovecha la potencia de procesamiento distribuida de Spark para manejar todos los metadatos para tablas a escala de petabytes, con millones de particiones y archivos.
- ***Time travel*: *Snapshots*** de los datos, lo que permite a los desarrolladores volver a versiones anteriores por auditoría o realizar *roll-backs*.
- **Unifica *Batch* y *Streaming*:** Una tabla *Delta* es tanto una tabla *batch*, como un origen o destino de datos para *streaming*. La ingesta de datos en streaming, las cargas históricas *batch* o las consultas interactivas funcionan de forma inmediata y en cualquier momento.
- **Aplicación y evolución del esquema:** Permite definir un esquema y estructura de los datos, lo que ayuda a asegurar que los datos son correctos y no se corrompen las tablas. Además, permite modificaciones del esquema de forma simple.
- ***Upserts* y *Deletes*:** Soporta APIs Scala / Java / Python para fusionar, actualizar y eliminar conjuntos de datos. Esto permite simplificar casos de uso complejos.

Por último, una vez explicado qué es el proyecto *Delta Lake*, cuáles son sus características principales y sobre qué arquitectura se construye, es necesario explicar cómo funciona *Delta*.

Delta almacena el esquema, la información del particionado y otras propiedades del dato (metadatos) en el mismo directorio que los datos. Este esquema e información inicial del particionado se encuentra en el fichero 00000.json bajo el directorio _delta_log, y a partir de ese momento las siguientes operaciones que se realicen crearán nuevos archivos JSON adicionales. Los JSON añadidos posteriormente al 00000.json contienen los ficheros borrados, los ficheros añadidos y los identificadores de las transacciones.

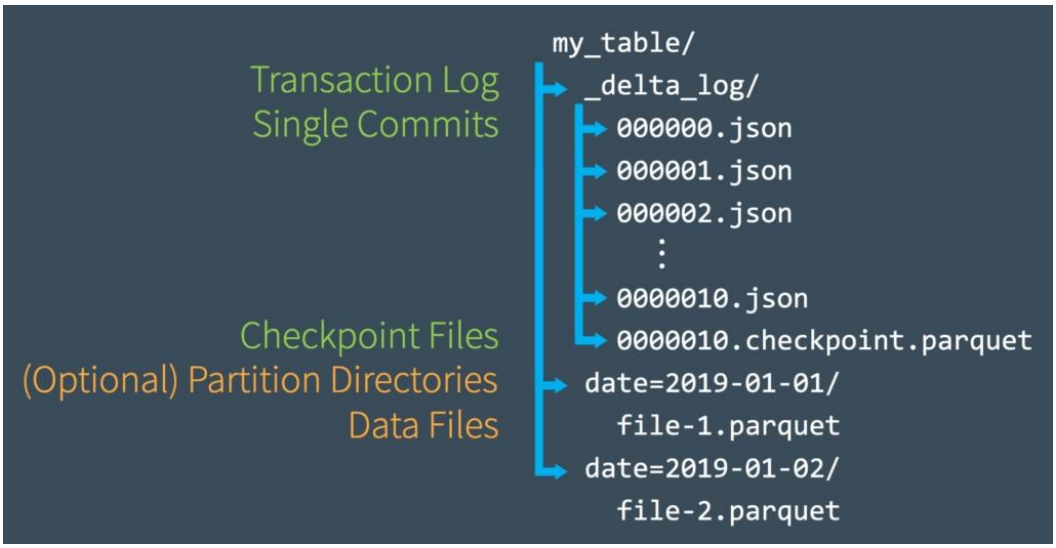

Figura 4. Estructura del directorio de una tabla Delta. [4]

Los datos en una tabla Delta son almacenados en formato *Parquet*, como se puede comprobar en la **Figura 4**. Al lanzar cualquier consulta sobre la tabla *delta*, Spark verifica los registros de transacciones para ver cuáles son las nuevas transacciones que se han publicado en la tabla. Esto asegura que la tabla siempre esté sincronizada con los últimos cambios y proporcione búsquedas más rápidas.

Delta Lake está pensado para sistemas transaccionales, donde haya un flujo constante de inserciones y actualizaciones de los datos, y un alto volumen de consultas, y no tanto para un sistema *batch* de volcado que, por ejemplo, haga solo una carga diaria de un gran volumen de datos, y que además sean datos que no tengan porqué variar en el tiempo.

2.4 Openstack y Ansible

OpenStack es un proyecto de código abierto de computación en la nube que permite gestionar grandes grupos de recursos de computación, de almacenamiento y de red de un *datacenter*. Permite proporcionar una IaaS (*Infraestructure as a service)*, todo gestionado y aprovisionado mediante APIs con mecanismos de autenticación comunes [5].

Este proyecto tiene una arquitectura modular, como se puede ver en la **Figura 5**, en la cual los principales módulos que lo forman son:

- **Compute (Nova):** Proporciona y gestiona instancias (máquinas virtuales).
- **Networking (Neutron):** Sistema para la gestión de redes y direcciones IP.

- **Dashboard (Horizon):** Proporciona una interfaz gráfica para el uso de la tecnología.
- **Servicio de identidad (Keystone):** Sistema que proporciona API de autenticación.
- **Servicio de imagen (Glance):** Servicio de imágenes que permite a los usuarios añadir, recuperar y registrar imágenes de VM (máquina virtual) e imágenes de contenedores.
- **Servicio de almacenamiento Block Storage (Cinder):** Es un servicio de almacenamiento de bloques para OpenStack. Está diseñado para proporcionar recursos de almacenamiento a los usuarios finales que Nova puede consumir.
- **Servicio de almacenamiento Object Storage (Swift):** Es un servicio de almacenamiento de objetos para OpenStack. Está diseñado para escalar y optimizado para durabilidad, disponibilidad y concurrencia en todo el conjunto de datos. Swift es ideal para almacenar datos no estructurados que pueden crecer sin límites.

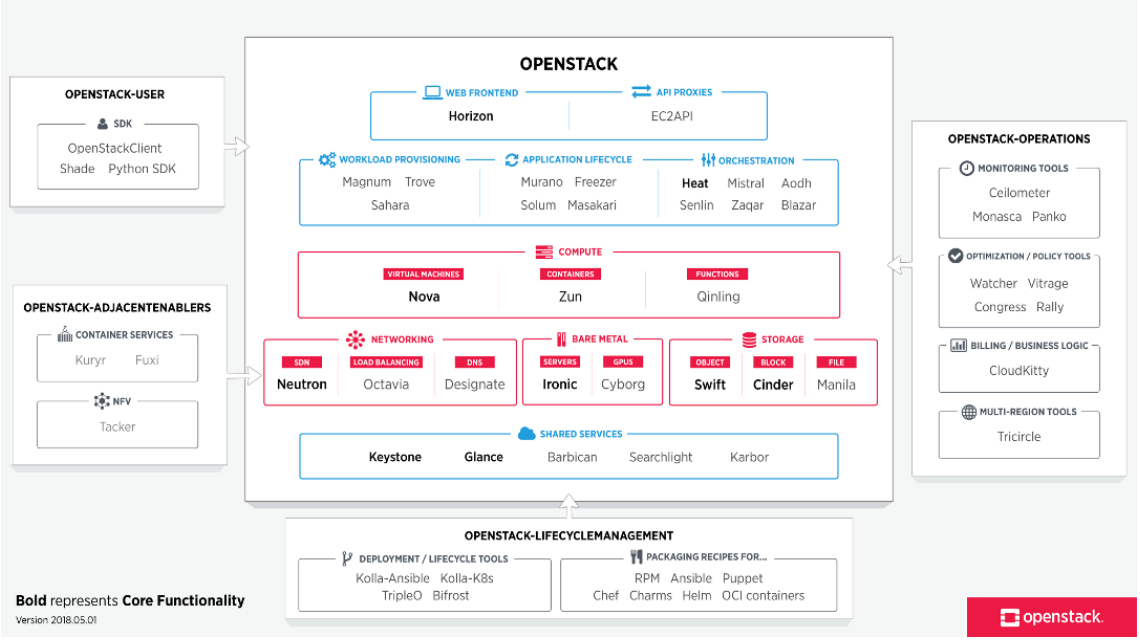

Figura 5. Estructura de los módulos que conforman el proyecto OpenStack. [5]

Por otro lado, **Ansible** es un software de gestión de la configuración automático y remoto. Permite, por ejemplo, desplegar un clúster sobre OpenStack automáticamente,

centralizando la configuración de numerosos servidores de una forma sencilla y automatizada.

Una parte importante y destacable de Ansible son los **roles**. Son formas de cargar automáticamente una estructura de archivos/directorios, instalación de software, etc., sobre una o varias instancias.

Gracias a roles predefinidos se puede instalar, por ejemplo, Spark y Hadoop sobre un clúster completo, de una forma automatizada y sencilla. Además, estos roles se pueden distribuir de forma sencilla a través de repositorios como GitHub.

2.5 Tipos de almacenamiento

2.5.1 Block Storage

Con el *Block Storage* (almacenamiento en bloques), los archivos se dividen en bloques de datos de tamaño uniforme, cada uno con su propia dirección, pero sin información adicional (metadatos) que proporcione más contexto de lo que es ese bloque de datos. Se puede encontrar almacenamiento en bloque en la mayoría de las cargas de trabajo empresariales. El *Block Storage* tiene una amplia variedad de usos, como, por ejemplo, sistemas tradicionales transaccionales.

2.5.2 Object Storage

El *Object Storage* o almacenamiento basado en objetos consiste en tratar los datos de las unidades de almacenamiento como objetos.

Esta forma de almacenamiento se ha vuelto muy popular con los servicios en la nube como *Amazon S3* o *Azure Blob Storage*. Cualquiera que haya almacenado una imagen en *Facebook* o una canción en *Spotify* ha utilizado el almacenamiento de objetos incluso si no lo sabe.

Cada objeto se compone de ciertos elementos y características:

- **Datos:** Estos datos pueden ser cualquier conjunto de información. No se tienen por qué corresponder con un fichero.
- **Metadatos:** Los metadatos asociados a cada objeto pueden crecer y extenderse con nueva información relativa al dato. Son definidos por el creador del objeto y

contienen toda la información que se considere de importancia para su almacenamiento. Comúnmente, entre estos metadatos se encuentra información de uso, fechas, permisos, etc.

- **Identificador del objeto:** Este identificador debe ser único, ya que se usa para localizar el objeto en nuestro sistema.

Es muy frecuente su uso para almacenamiento de datos que necesitan tener una alta disponibilidad, ser muy durables y no variar en el tiempo. Estas propiedades se consiguen habitualmente con técnicas de replicación (almacenando varias copias del dato en un sistema distribuido), de esta forma, aunque existan problemas en algunas partes del sistema, se puede seguir dando servicio sin interrupción. La replicación del dato también es útil para determinar automáticamente datos corruptos comparando el valor o el hash de cada objeto (identificador único) con el mismo valor calculado en sus copias.

Capítulo 3 – Arquitectura, diseño e infraestructura

3.1 Arquitectura y configuración de nodos

Como se ha comentado previamente, el clúster, basado en OpenStack, se ha montado sobre el *Cloud Computing* del CPD (Centro de Procesamiento de Datos) del IFCA [1]. Este clúster, muy sencillo para llevar a cabo este trabajo, está compuesto por 3 nodos: 2 nodos empleados para la computación y 1 nodo auxiliar para la administración y configuración de los otros dos.

La configuración de los diferentes nodos es la siguiente:

Instance Name	Image	IP Address	Flavor	VCPU	RAM	Disk
nodoMasterTFM	**IFCA Ubuntu 20.04**	172.16.35.93	cm4.xlarge	4	7.3 GB	30 GB
nodoSlaveTFM	IFCA Ubuntu 20.04	172.16.35.4	cm4.xlarge	4	7.3 GB	30 GB
nodoAnisbleTFM	**IFCA Ubuntu 20.04**	172.16.35.47	m1.small	1	2 GB	10 GB

Tabla 1. Tabla que contiene las configuraciones y recursos asignados a cada uno de los nodos del clúster.

Los nodos “nodoMasterTFM” y “nodoSlaveTFM”, como se puede ver en la **Tabla 1** porque son los nodos que más recursos tienen asignados, son los que van a conformar el clúster de computación distribuída, siendo Apache Hadoop y Apache Spark los *frameworks* utilizados para la gestión, almacenamiento y computación de los datos, mientras que el “nodoAnsibleTFM” va a ser el nodo auxiliar, que no necesita tantos recursos, a través del cuál vamos a configurar e instalar Hadoop y Spark en los nodos de computación de una forma automatizada y sencilla usando Ansible. El esquema del clúster desplegado se puede ver en la **Figura 6.**

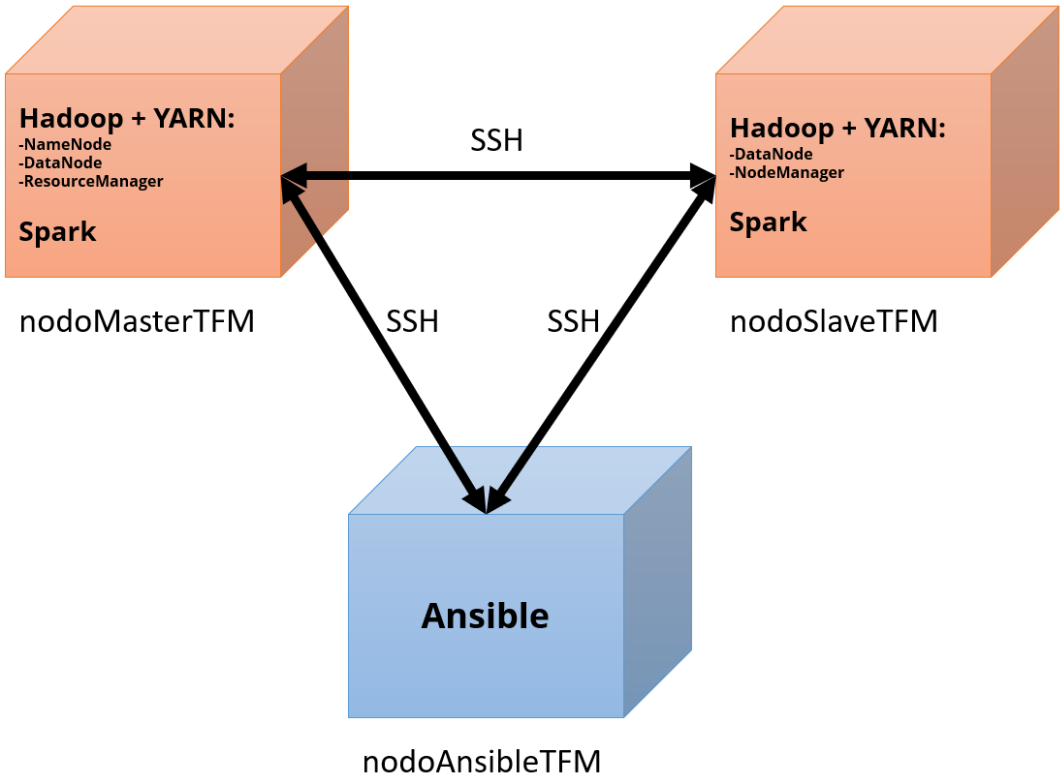

Figura 6. Esquema del clúster desplegado para el proyecto sobre la plataforma OpenStack.

La comunicación entre los nodos se va a realizar mediante el protocolo SSH. SSH o *Secure Shell*, es un protocolo de administración remota que le permite a los usuarios controlar y modificar sus servidores remotos a través de Internet a través de un mecanismo de autenticación.

Concretamente se ha aplicado un cifrado asimétrico, el cual utiliza dos claves separadas para el cifrado y descifrado. Estas dos claves se conocen como la **clave pública** (*public key*) y la **clave privada** (*private key*). Juntas, estas claves forman el par de **claves pública-privada** (*public-private key pair*).

La clave privada es secreta, mientras que la clave pública es pública, como su nombre indica. Ambas claves están relacionadas matemáticamente y una información cifrada con una clave pública solo puede ser descifrada con su clave privada, y viceversa (lo que sería una firma).

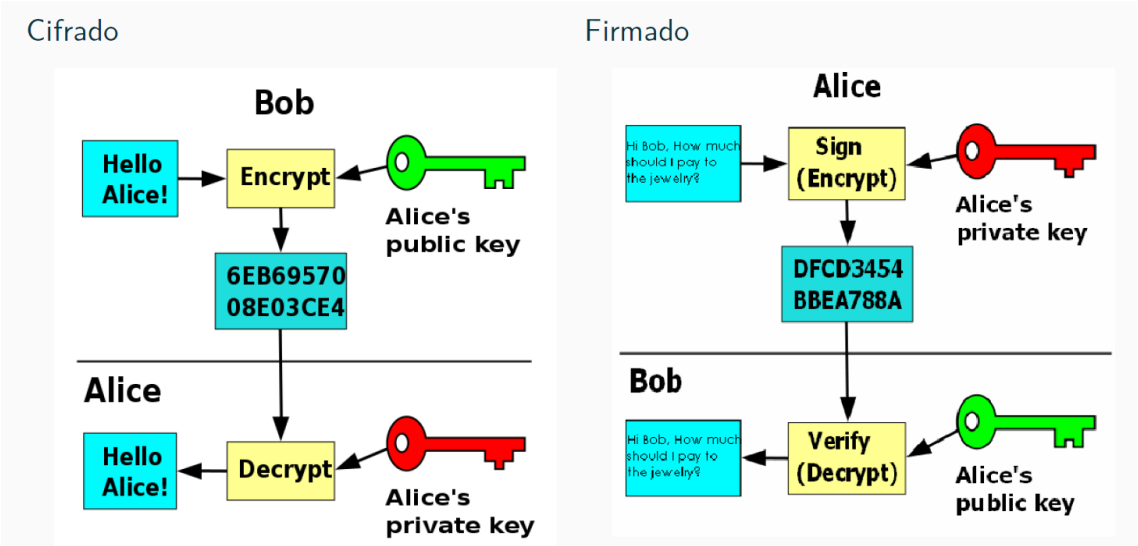

Figura 7. Ejemplo de cifrado y firma utilizando el par de claves pública – privada. [7]

3.2 Instalación de entorno Hadoop y Spark con roles Ansible

Como se ha comentado previamente, la arquitectura del clúster está formada por dos nodos de computación y un nodo auxiliar para administrarlos a ambos. El primer paso para la instalación del entorno Hadoop y Spark en los nodos del clúster es la instalación de Ansible en el nodo auxiliar.

Ansible nos va a permitir poder ejecutar roles. Los roles de Ansible posibilitan realizar tareas de administración y configuración sobre varios *hosts* simultáneamente de una manera sencilla y reproducible en el tiempo. Una vez tienes un rol de Ansible configurado correctamente, servirá para, en el futuro, volver a realizar las mismas tareas de administración sin tener que repetir los pasos uno a uno, máquina a máquina, etc.

Este nodo auxiliar con Ansible nos va a permitir, en resumen, desplegar de una forma automatizada y escalable en el tiempo. Una vez configurados los roles y *playbooks* será posible, y sobre todo en un clúster distribuido como este, ampliar el número de nodos *slave* para aumentar los recursos de una forma sencilla en caso de que fuera necesario.

La forma de ejecutar estos roles es primero ejecutar el *playbook*. En el *playbook* se indica los *hosts* sobre los que se va a actuar, y, en el orden deseado, los roles que se quieren ejecutar. Los roles son directorios compuestos por otros directorios como *vars*, *files*, *template*, *tasks*, *defaults*, etc., y dentro del directorio *tasks* se encuentra el fichero

main.yml que es donde realmente se encuentran todos los pasos necesarios para, en este caso, instalar Hadoop o Spark.

Después de instalar ansible, utilizando el usuario administrador *root* en el nodo auxiliar, se llevan a cabo los siguientes pasos para la configuración e instalación correcta del clúster de computación distribuída:

1. Se modifica el fichero /etc/ansible/ansible.cfg añadiendo las siguientes variables:
   inventory = /etc/ansible/hosts
   host_key_checking = False
   remote_user = ubuntu

- El fichero hosts se usará como inventario
- Deshabilitar la opción para chequear las keys de los nodos. Por ejemplo, si un nodo se reinstala tendrá una key diferente en el fichero known_hosts y aparecerá un mensaje de error hasta que sea corregido
- El usuario remoto que usaremos será ubuntu

2. Se modifica el fichero /etc/ansible/hosts añadiendo los hosts de los 2 nodos de computación, para posteriormente poder ejecutar los roles sobre ellos:
   [nodosGroupTFM]
   ubuntu@172.16.35.31
   ubuntu@172.16.35.47

3. Con el usuario *root*, dentro del nodo de Ansible, se generan un par de claves. Esto es importante para que el nodo auxiliar se pueda, más adelante, conectar a las máquinas que conforman el clúster de computación distribuída:
   # ssh-keygen

4. Se comprueba la clave creada, que posteriormente será copiada a los otros nodos para que esté habilitada la conexión a estos desde el "nodoAnsibleTFM":
   # cat /root/.ssh/id_rsa.pub

5. Se realizan los pasos de conexión iniciales al "nodoMasterTFM" y, empleando el usuario "ubuntu", se crea en este nodo el fichero *id_rsa.pub*, en el cual se ha

metido la clave pública del "nodoAnsibleTFM" expuesta en el paso 8. Una vez en el fichero *id_rsa.pub* se copia dicha clave a *authorized_keys*:
 # cat id_rsa.pub > authorized_keys

6. Se ejecutan los siguientes comandos *chmod* para dar los permisos necesarios a diferentes directorios:
   # chmod 755 /home
   # chmod 700 /home/ubuntu /home/ubuntu/.ssh
   # chmod 600 /home/ubuntu/.ssh/authorized_keys

7. Se realizan los pasos 9 y 10 sobre el nodo "nodoSlaveTFM".

8. Teniendo ya la clave pública en los nodos de computación, habilitando así la conexión a dichos nodos desde el nodo auxiliar, se procede a comprobar que se puede hacer *ping* a ambos *hosts* con un comando de Ansible. Este paso se realiza, nuevamente, desde el nodo auxiliar "nodoAnsibleTFM" con el usuario *root*:
   # ansible all -m ping

9. Ahora se procede a utilizar el playbook y los roles de ansible [8]. En este caso la ruta donde se van a crear los ficheros es "/root/git/ids2DeltaLake/ansible".
   El *playbook* y los roles son reutilizados y ya están previamente configurados para el despliegue de Hadoop y Spark. Este es el punto fuerte del uso de Ansible, el cual permite partir de una base y no tener que hacer todo el proceso de configuración del clúster de forma manual. El *playbook* es *site_mods.yml* y los roles son *mods-hadoop* y *mods-spark*.

   Pese a que las configuraciones del *playbook* y los roles son reutilzados, se han tenido que adaptar a nuestro proyecto. Entre esos cambios, se ha modificado el número de nodos de replicación de HDFS a dos, se ha cambiado la versión de Spark a la versión 3.3.0, se han tenido que modificar algunos *paths* dentro de la configuración de los roles, etc.

   El código de los diferentes YAML se encuentra en el "**ANEXO:** *Playbook* y roles de Ansible" a este documento.

10. Finalmente, con el usuario *root*, en el "nodoAnsibleTFM", se ejecuta el *playbook* para instalar los entornos Hadoop y Spark en los dos nodos de computación. Como se puede ver en la **Figura 8**, se llevan a cabo satisfactoriamente todos los *tasks* de los roles de Ansible, lo que lleva a la correcta instalación de Hadoop y Spark en ambos nodos:
    # anisble-playbook site_mods.yml –become

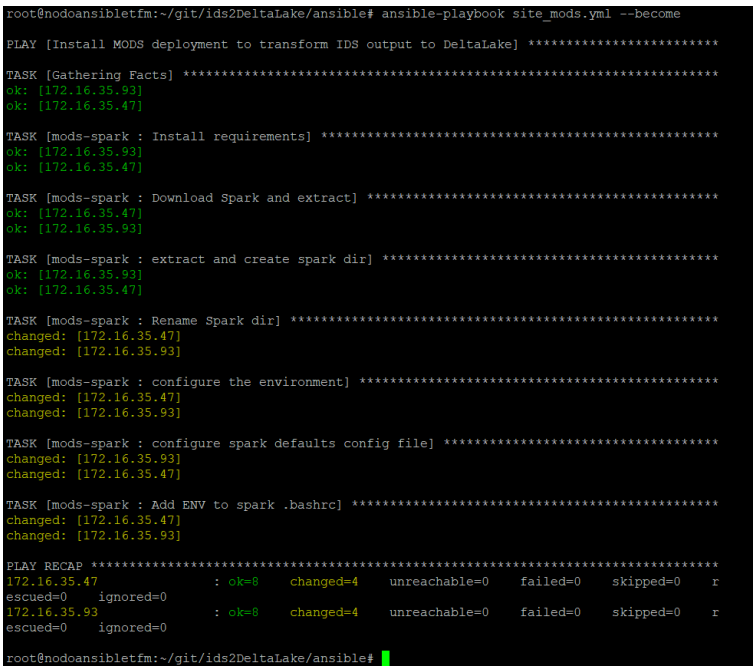

Figura 8. Ejemplo de la salida estándar (*STDOUT*) tras la ejecución correcta del rol *mods-spark.*

Tras el correcto despliegue, se han tenido que realizar muchos ajustes sobre el clúster. Entre ellos, revisar los *paths* de configuración de Hadoop, ya que había alguno que no era correcto y provocaba, por ejemplo, que no se llevase a cabo la replicación de los datos de HDFS entre los dos *datanodes*.

Por otro lado, han surgido muchos problemas que han llevado bastante tiempo solventar con las comunicaciones entre los nodos del clúster, debido a las reglas de *firewall* de la plataforma OpenStack, y los permisos sobre los puertos. No se conseguían resolver las conexiones entre las IPs privadas, lo que provocaba, por ejemplo, que el nodo *master* no fuera capaz de derivar tareas al nodo *slave*, y por ello, no podíamos ejecutar de forma distribuida, que era uno de los objetivos de montar este clúster.

3.3 Configuración y conexión con el *Object Storage* (SWIFT)

La conexión con HDFS es sencilla, ya que está montado sobre los propios nodos del clúster, y además se ha configurado durante la ejecución de los roles de Ansible, pero la configuración y conexión con el *Object Storage* no es tan trivial.

Para *Swift* (*Object Storage*), en vez de utilizar el servicio de OpenSatck, nos beneficiamos de que tenemos el *storage* de OpenStack montado a través de ceph (sistema de archivos distribuido libre, diseñado para el uso con gran cantidad de datos). Así, utilizamos *ceph rados gw* [9] para montar *swift*.

La escritura/lectura en *Swift* no es sencilla, se ha necesitado utilizar la API de la librería **io.github.nanhu-lab:hadoop-cephrgw** que se descarga automáticamente en la configuración de Spark, pero se encuentra en el repositorio de Maven (desarrollada por Apache, es una herramienta de software para la gestión y construcción de proyectos Java) [10] para poder conectarte con el *Object Storage* que se ha creado en OpenStack.

La configuración empleada en lo *scripts* es la siguiente:

```
--packages "io.delta:delta-core_2.12:1.2.1,io.github.nanhu-lab:hadoop-
cephrgw:1.0.2,org.javaswift:joss:0.10.4"
--conf "spark.hadoop.fs.ceph.username=gomezan"
--conf "spark.hadoop.fs.ceph.password=lFs%8KFpfrGP"
--conf "spark.hadoop.fs.auth.method=keystone"
--conf "spark.hadoop.fs.tenant.name=ifca.es:tfm_AngelGomez"
--conf "spark.hadoop.fs.tenant.id=d0aa906b6e3d4201bebc7dd7a55ffcfd"
--conf "spark.hadoop.fs.domain.name=default"
--conf "spark.hadoop.fs.auth.uri=https://api.cloud.ifca.es:5000/v3/auth/tokens"
--conf "spark.hadoop.fs.s3a.connection.ssl.enabled=false"
--conf "spark.delta.logStore.class=org.apache.spark.sql.delta.storage.S3SingleDriverLogStore"
--conf "spark.hadoop.fs.ceph.impl=org.apache.hadoop.fs.ceph.rgw.CephStoreSystem"
```

3.4 Data-Set

Se ha escogido un juego de datos en *Kaggle* [11] de dominio público, ya que la finalidad es poder comparar el rendimiento de diversos procesos ETL, y la información de los datos en sí no es relevante, sino la volumetría y el formato de éstos.

Lo que sí se ha tenido en cuenta ha sido la naturaleza de los datos, ya que *Delta* es un formato que, como se ha explicado en sus características principales, está enfocado a transacciones, aunque la transaccionalidad de los datos sería relevante si se ejecutaran procesos *streaming* (posible ampliación de este proyecto que se explicará más adelante), mientras que *parquet* no está diseñado para una naturaleza de datos específica.

En este caso los procesos de carga masiva (*batch*) que vamos a realizar van a ser agnósticos del tipo de dato que se esté volcando, pero si en un futuro se quisiera ampliar este proyecto con ejecuciones *streaming*, como se acaba de comentar, lo interesante sería que en ambas situaciones el juego de datos sea el mismo y de naturaleza transaccional, para sacar unas conclusiones más ambiciosas.

El fichero es un CSV con 8 columnas y aproximadamente un millón de registros, aunque para poder hacer pruebas más exhaustivas, y poder obtener mejores conclusiones sobre los rendimientos, se ha duplicado la información del fichero hasta obtener, aproximadamente, unos 10 millones de registros (como se ha comentado, lo importante para el rendimiento es una volumetría mínima, y no que luego los *parquet* tengan información duplicada o no).

Finalmente, el *Data-Set* a emplear tiene un tamaño de 1.2 GB, con sus 10 millones, aproximadamente, de registros. No es un tamaño del orden de las decenas de Gb como se suelen emplear en estos contextos de *Big Data*, pero a la hora de ejecutar y sacar rendimientos y comparativas hemos podido comprobar que con esta volumetría las diferencias en tiempos son evidentes y analizables, lo que nos proporciona un buen equilibrio entre buenos resultados y tiempos cortos de ejecución de los *scripts* (lo cual se traduce en poder ejecutar más veces y obtener resultados más consistentes y contrastados).

La descripción de las columnas del fichero es la siguiente:

- **UserId:** Identificador único para cada usuario

- **TransactionId:** Identificador único de la transacción
- **TransactionTime:** Hora de la transacción
- **ItemCode**: Código identificador del artículo comprado
- **ItemDescription:** Descripción del artículo
- **NumberOfItemPurchased**: Número de unidades compradas del artículo
- **CostPerItem**: Precio de cada unidad
- **Country**: País donde se ha realizado la transacción

Donde la distribución de los valores por cada millón de registros es la siguiente:

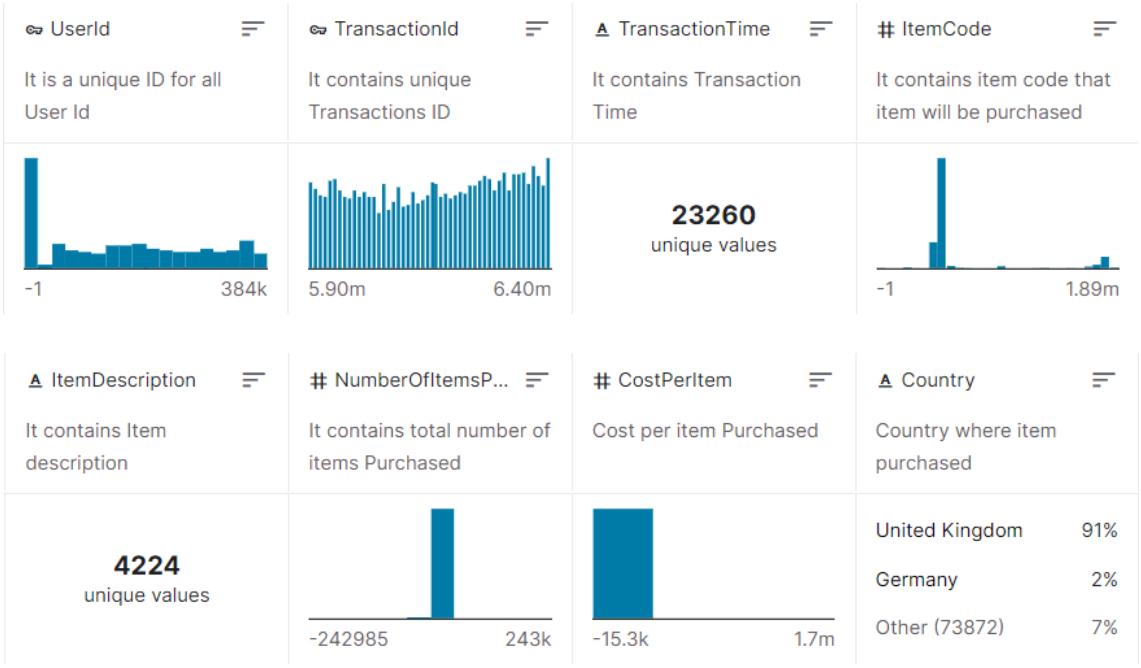

Figura 9 Distribución del juego de datos por cada millón de registros [11].

UserId	TransactionId	TransactionTime	ItemCode	ItemDescription	NumberOfItemsPurchased	CostPerItem	Country
278166	6355745	Sat Feb 02 12:50:...	465549	FAMILY ALBUM WHIT...	6	11.73	United Kingdom
337701	6283376	Wed Dec 26 09:06:...	482370	LONDON BUS COFFEE...	3	3.52	United Kingdom
267099	6385599	Fri Feb 15 09:45:...	490728	SET 12 COLOUR PEN...	72	0.9	France
380478	6044973	Fri Jun 22 07:14:...	459186	UNION JACK FLAG L...	3	1.73	United Kingdom
-1	6143225	Mon Sep 10 11:58:...	1733592	WASHROOM METAL SIGN	3	3.4	United Kingdom

Figura 10. Ejemplo de 5 registros del CSV.

3.5 Casos de uso

Los casos de uso que se van a describir a continuación nos van a permitir comparar los rendimientos de los ficheros *parquet* y *delta* desde varios puntos de vista, siendo nosotros los que favorezcamos, y perjudiquemos, el rendimiento de estos tipos de fichero con tal de evaluar cómo se comportan en las condiciones teóricas óptimas para ellos, y en las del otro.

Por poner un ejemplo, un fichero *parquet* sabemos que tiene un buen rendimiento tanto en escritura, como en lectura, en un proceso masivo *batch* guardando los datos sobre un HDFS (*block storage*), ya que es un uso muy habitual que se le da, pero, ¿cómo se comportará un fichero *delta* en esas mismas condiciones? ¿será más rápido escribiendo o leyendo?

Por ello, los casos de uso se van a tener en cuenta según los siguientes puntos:

- **Tipo de flujo:** *batch*
- **Extensión del output:** *parquet* y *Delta*
- **Tipo de almacenamiento:** *Block Storage* (HDFS) y *Object Storage* (SWIFT - OpenStack)
- **Tipo de medida:** Lectura y escritura

Esto nos lleva a, en total, tener 8 resultados, es decir, 4 comparativas (2 de rendimiento en escritura y 2 de rendimiento en lectura). Además, para obtener cada uno de los resultados (por ejemplo, el tiempo de escritura de un *parquet* en un proceso *batch* donde el *parquet* se encuentra alojado en un *Block Storage*) se va a realizar una media a partir de 5 ejecuciones diferentes del mismo flujo.

Capítulo 4 – Resultados

En este capítulo se van a mostrar los diferentes gráficos comparativos, tanto de rendimiento del clúster en función de los recursos asignados para nuestro *Data Set*, como el rendimiento de *Parquet* frente a *Delta* en cada uno de los casos de uso. Se han desarrollado diferentes scripts para la obtención de dichos resultados que se podrán consultar en el "**ANEXO:** *Scripts Python***"** al final de este documento.

4.1 Rendimiento del clúster según recursos asignados

En este primer punto lo que se ha querido es, mediante diferentes pruebas, buscar cuál es la mejor asignación de recursos al proceso ETL (ver **Tabla 2**) para el tratamiento de nuestro *Data Set*, el cuál, como ya se ha comentado previamente, tiene alrededor de los 10 millones de registros, y ocupa un volumen de aproximadamente 1.2 Gb.

Para ello, se ha llevado a cabo el caso de uso de volcado de los datos del CSV a un *parquet*, con un sistema de almacenamiento *Block Storage* (HDFS):

Number of Executors	Executor memory	Executor Cores	Write Time/s
1	1G	1	77,41
1	1G	2	45,92
1	1G	3	31,29
1	1G	4	31,77
1	2G	2	37,15
1	2G	3	28,79
1	2G	4	27,05
1	3G	2	52,04
1	3G	4	29,51
1	4G	4	29,023
2	1G	2	31,76
2	1G	3	23,55
2	1G	4	20,81
2	2G	2	27,58
2	2G	3	21,78
2	2G	4	22,95
2	3G	2	30,26
2	3G	3	20,93
2	3G	4	20,21
2	4G	3	20,51
2	4G	4	23,48

Tabla 2. Tiempos de escritura del *Parquet* en HDFS en función del número de ejecutores, memoria y *cores* usados en la ejecución.

Tras obtener esta tabla de resultados se ha realizado un gráfico de dispersión para, de una manera más visual, ver el rendimiento en función de los ya mencionados parámetros:

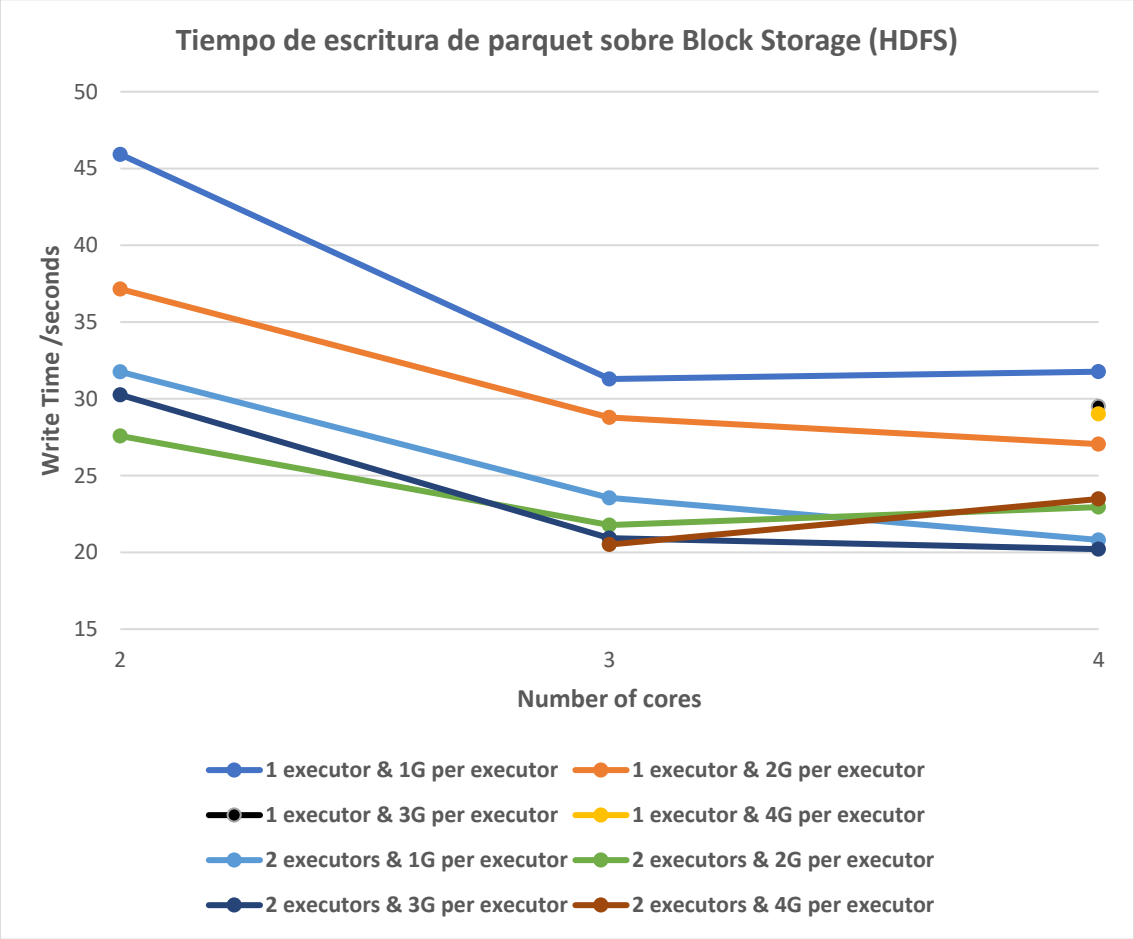

Figura 11. Gráfico de dispersión obtenido para cada una de las series de datos de la Figura 11.

Mediante la **Tabla 2**, y el gráfico de la **Figura 11**, se puede observar que la mejor asignación de los recursos del clúster a la ETL, y, por lo tanto, la que se va a usar para obtener las comparativas *Parquet vs Delta* es:

- **Número de ejecutores:** 2
- **Memoria asignada por ejecutor:** 3 Gb
- **Número de *cores* o núcleos por ejecutor:** 4

Como se puede observar el máximo de *cores* disponibles por nodo es 4, y el emplearlos todos favorece la computación, al igual que emplear los dos nodos, pero un exceso de memoria asignada a cada nodo puede producir una penalización de rendimiento, debido a que no necesita tanta memoria para el procesamiento de los datos, y gestionar el exceso ralentiza la computación.

4.2 *Parquet vs Delta*

En segundo paso, ya una vez identificada la mejor configuración de recursos para el clúster, se procede a obtener los diferentes resultados, en tiempos de lectura y escritura, en función del tipo de almacenamiento y de extensión de fichero. En todos los casos se ejecutarán procesos *batch*, es decir, vamos a realizar volcados completos del fichero CSV de una sola vez.

4.2.1 Parquet – Block Storage (HDFS)

Write/s	Read/s
19,08	0,35
14,95	0,23
16,27	0,23
14,52	0,44
15,26	0,27

Tabla 3. Tabla con los datos obtenidos para los tiempos de lectura y escritura del *Data Set* sobre el *Block Storage* (HDFS) en formato *Parquet*.

El valor de la media, junto con su desviación estándar, para ambas medidas, obtenido a partir de los datos de la **Tabla 3** es:

Write Time: 16.0 ± 1.8 segundos

Read Time: 0.30 ± 0.09 segundos

4.2.2 Delta – Block Storage (HDFS)

Write/s	Read/s
26,81	0,02
28,11	0,02
24,68	0,03
27,53	0,03
26,67	0,02

Tabla 4. Tabla con los datos obtenidos para los tiempos de lectura y escritura del *Data Set* sobre el *Block Storage* (HDFS) en formato *Delta*.

El valor de la media, junto con su desviación estándar, para ambas medidas, obtenido a partir de los datos de la **Tabla 4** es:

Write Time: 26.8 ± 1.3 segundos

Read Time: 0.02 ± 0.01 segundos

4.2.3 Parquet – Object Storage (SWIFT)

Write/s	Read/s
50,17	6,09
50,04	5,66
50,29	5,65
49,87	5,64
50,69	6,03

Tabla 5. Tabla con los datos obtenidos para los tiempos de lectura y escritura del *Data Set* sobre el *Object Storage* (SWIFT) en formato *Parquet*.

El valor de la media, junto con su desviación estándar, para ambas medidas, obtenido a partir de los datos de la **Tabla 5** es:

Write Time: 50.2 ± 0.3 segundos

Read Time: 5.8 ± 0.2 segundos

4.2.4 Delta – Object Storage (SWIFT)

Write/s	Read/s
36,86	1,07
35,84	0,98
34,42	1,01
34,43	0,97
34,08	1,07

Tabla 6. Tabla con los datos obtenidos para los tiempos de lectura y escritura del *Data Set* sobre el *Object Storage* (SWIFT) en formato *Delta*.

El valor de la media, junto con su desviación estándar, para ambas medidas, obtenido a partir de los datos de la **Tabla 6** es:

Write Time: 35.1 ± 1.2segundos

Read Time: 1.02 ± 0.05 segundos

4.2.5 Comparativas de rendimiento

Tras exponer los diferentes resultados, y las medias y sus errores, se puede adelantar que hay, en tiempo de escritura, un mejor rendimiento de *parquet* sobre HDFS y un mejor rendimiento de *delta* sobre *Swift*, mientras que en tiempos de lectura siempre es mejor *delta*, debido a sus metadatos.

A continuación, se procede a mostrar los gráficos comparativos, para una mejor percepción a nivel visual, y cuantitativa, de las diferencias de rendimiento de ambos tipos de extensión de fichero en función del sistema de almacenamiento empleado:

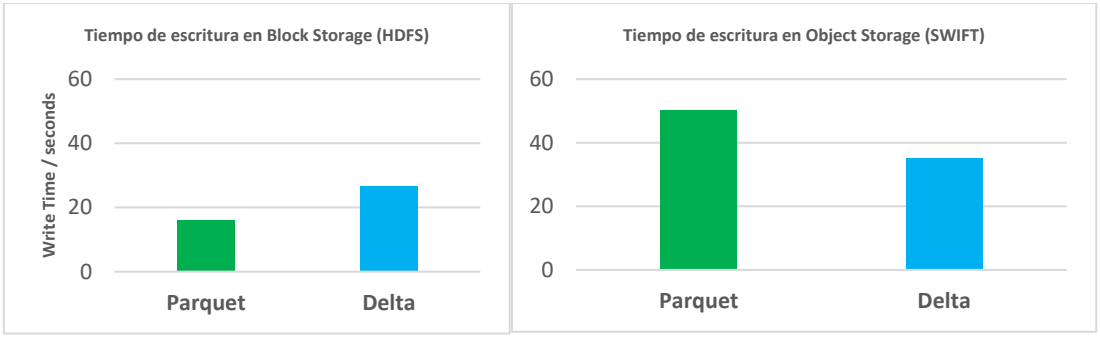

Figura 12. Gráficos de barras donde se pueden comparar el rendimiento, en segundos, de la velocidad de escritura de *Parquet vs Delta* para el *Data Set* de 1.2 Gb sobre un *Block Storage* (HDFS) y sobre el *Object Storage* (SWIFT).

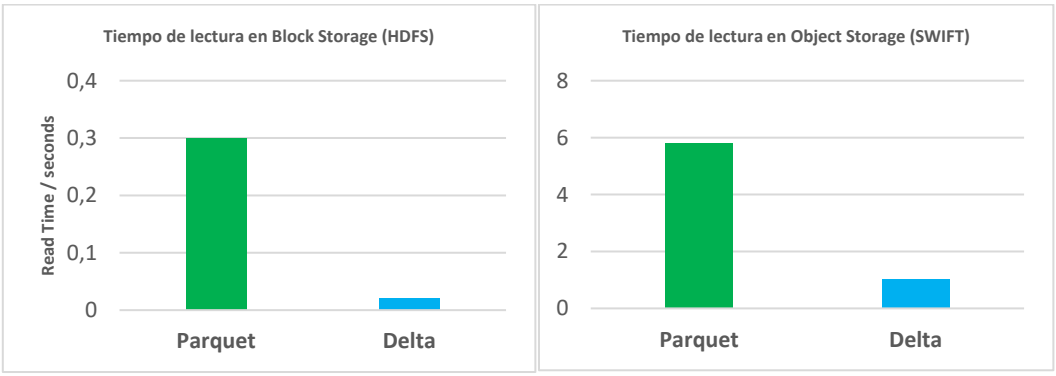

Figura 13. Gráficos de barras* donde se pueden comparar el rendimiento, en segundos, de la velocidad de lectura de *Parquet vs Delta* para el *Data Set* de 1.2 Gb sobre un *Block Storage* (HDFS) y sobre el *Object Storage* (SWIFT).

** En la comparativa de tiempos de lectura no están ambos gráficos de barras en la misma escala, debido a que en HDFS los tiempos son mucho menores, y si igualásemos las escalas no se apreciarían las barras de los tiempos sobre HDFS en la comparativa.*

Capítulo 5 – Conclusiones

5.1 Conclusiones de la PoC

En este proyecto nos hemos centrado finalmente en la parte *batch*, y no se han llevado a cabo ejecuciones *streaming* (mencionadas durante la memoria), puesto que la adaptación del *playbook* y los roles de Ansible, la resolución de los problemas de *firewall* y puertos de OpenSatck, las conexiones entre nodos, los *paths* de configuración, etc., han requerido un gran esfuerzo y tiempo y han imposibilitado realizar esas ejecuciones *streaming*. Además, de que hubiera requerido la instalación de Apache Kafka, *scripts* más complejos, etc., y consideramos que se podría exceder en el propósito, y extensión, de este proyecto.

En resumen, mencionar que se ha hecho una comparativa de las tecnologías, se ha llevado a cabo la búsqueda de la información, se ha desplegado el clúster de manera automática con una adaptación de los roles de Ansible, pero que ha permitido un ahorro de tiempo si se hubiesen instalado los nodos de manera manual, se han solucionado distintos problemas de conexiones entre los nodos del clúster que imposibilitaban llevar a cabo el análisis. En definitiva, se ha conseguido desplegar un clúster estable y escalable (como vemos también con los resultados, sale mejor con 2 ejecutores que con 1, y no sabemos si añadiendo otro podría mejorar incluso más) para hacer un análisis ETL.

Tras los datos mostrados en el **Capítulo 4** podemos obtener varias conclusiones, siendo la primera de ellas el comportamiento del tiempo de escritura en función de los recursos asignados a la ejecución.

Como se puede comprobar, los resultados siempre son mejores entre 2 ejecutores que solo escogiendo uno, dentro del clúster que se ha desplegado, ya que puede distribuir la computación entre el nodo *master* y el nodo *slave*, pero no siempre más es mejor, ya que como también se puede apreciar si le damos más memoria de la que necesita, o hilos/núcleos de más, lo que sucede es que el tiempo que invierte el clúster en gestionar dichos recursos y sus conexiones repercute negativamente en el tiempo final de la acción que queremos llevar a cabo.

Seguramente, las mismas configuraciones que hemos indicado en la **Tabla 2**, si las aplicamos a un *Data Set* de 50Gb los tiempos variarían mucho, y al ser un gran volumen

de datos el que habría que tratar, configuraciones con más recursos nos proporcionarían mejores tiempos.

Por otro lado, en la comparación entre *parquet* y *delta* podemos ver que el resultado es el esperado según la teoría. En tiempos de escritura es mejor *parquet* sobre un *Block Storage*, mientras que *delta* lo es sobre el *Object Storag,e* ya que ambas extensiones de fichero están preparadas para esos tipos de almacenamiento, y vemos como *parquet* penaliza sobre *Swift*, y *delta* penaliza sobre *HDFS.*

En cuanto a los tiempos de lectura vemos que siempre es más rápido *delta* y esto es correcto, ya que como hemos visto en el **Capítulo 3** está diseñado con unos metadatos (que la escritura de éstos también penaliza los tiempos de escritura del fichero *delta* en sí) que hacen que, aprovechando la potencia de computación que ofrece *Spark*, la consulta de la información sea mucho más rápida.

Concluyendo, *Delta Lake* puede ser un buen enfoque de cara a optimizar el rendimiento de sistemas críticos transaccionales (sobre todo operaciones *streaming*), dado que demuestra ser capaz de mejorar tiempos de reprocesado, manejar un histórico y versionado de los datos, lo que hace que en un sistema donde esté constantemente recibiendo registros va a ser más rápido escribiendo y leyendo que *parquet*. Por otro lado, *parquet* sigue siendo mejor para otros casos de uso más clásicos, como pueden ser flujos *batch* de volcado masivo de datos sobre HDFS, y de datos que, por su naturaleza, no van a ser tan reiteradamente consultados como sucede en contextos transaccionales.

5.2 Posibilidades de ampliación

En este proyecto se ha tratado el tipo de flujo *batch* debido a que, por tiempos (el tiempo estimado de instalación y estabilización del clúster fue mayor al esperado), y extensión de la propia memoria en sí, no era viable abarcar el doble de resultados y comparativas llevando a cabo ejecuciones de tipo *streaming*. Por lo tanto, las posibles vías de ampliación de esta PoC podrían seguir las siguientes:

- Realizar los mismos casos de uso, es decir, *parquet vs delta* en *Block Storage* y *Object Storage*, en tiempos de lectura y escritura, pero esta vez comparando con flujos *streaming.*

- Añadir algún otro tipo de extensión de fichero a la comparativa, como por ejemplo puede ser AVRO, XML, JSON…
- Ampliar la volumetría de los datos y el número de nodos *slave* del clúster para hacer comparativas más cercanas a la realidad comercial, y poder buscar mejores configuraciones de los recursos.
- Ejecutar otros tipos de mediciones como *overwrites*, *appends*, *deletes* o *updates*.

Referencias

[1] Computing Services – IFCA Advanced Computing. (s. f.). Recuperado 13 de septiembre de 2022, de https://computing.ifca.es/services/

[2] BELOV, V.; TATARINSTEV, A.; NIKULCHEV, E. *Choosing a Data Storage Format in the Apache Hadoop System Based on Experimental Evaluation Using Apache Spark.* Symmetry, 2021, 13, 195. https://doi.org/10.3390/sym13020195

[3] XUNTA DE GALICIA. *Estado del arte de Big Data. Oportunidades Industria 4.0 en Galicia.* Noviembre 2017.

[4] *Data Lakehouse Architecture and AI Company*. (2022, 12 septiembre). Databricks. Recuperado 13 de septiembre de 2022, de https://www.databricks.com/

[5] *Open Source Cloud Computing Platform Software*. OpenStack. Recuperado 13 de septiembre de 2022, de https://www.openstack.org/software

[6] ARMBRUST ET AL. *Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores*. PVLDB, 13(12): 3411-3424, 2020. DOI: https://doi.org/10.14778/3415478.3415560

[7] LÓPEZ GARCÍA, A. *Apuntes Master, asignatura "Seguridad, privacidad y aspectos* legales". Instituto de Física de Cantabria (IFCA) - CSIC-UC. Curso 2020-2021.

[8] Palacio, A. (s. f.). ids2DeltaLake/ansible at master · aidaph/ids2DeltaLake. GitHub. Recuperado 13 de septiembre de 2022, de https://github.com/aidaph/ids2DeltaLake/tree/master/ansible

[9] Ceph Object Gateway Swift API — Ceph Documentation. (s. f.). Recuperado 15 de septiembre de 2022, de https://docs.ceph.com/en/latest/radosgw/swift/

[10] Porter, B. (s. f.). Maven – Welcome to Apache Maven. Recuperado 15 de septiembre de 2022, de https://search.maven.org/artifact/io.github.nanhu-lab/hadoop-cephrgw/1.0.2/jar

[11] Kaggle: Your Machine Learning and Data Science Community. (s. f.). Recuperado 13 de septiembre de 2022, de https://www.kaggle.com/datasets/vipin20/transaction-data?select=transaction_data.csv

ANEXO: *Playbook* y roles de Ansible

site_mods.yml

```
- name: Install MODS deployment to transform IDS output to DeltaLake
  hosts: nodosGroupTFM
  roles:
    - {role: 'mods-spark'}
    - {role: 'mods-hadoop', hadoop_type_of_node: 'namenode'}
```

Defaults – mods-hadoop

```
---
# defaults file for mods-hadoop
mods_namenode: { name: "{{ ansible_hostname }}", ip: "{{
ansible_default_ipv4.address }}" }
hadoop_version: 3.3.1
hadoop_home: /opt/hadoop

# The type of the node: datanode / namenode
hadoop_type_of_node: namenode
java_home: /usr/lib/jvm/java-8-openjdk-amd64/
path_hdfs: /hadoop_data/hdfs
role_path: /etc/ansible/roles/mods-hadoop
```

Tasks – mods-hadoop

```
---
# tasks file for mods-hadoop
- name: "Create user for hadoop"
  user:
    name: hadoopuser
    group: hadoopgroup

- name: "Install requirements"
  apt:
    name: openjdk-8-jdk, openssh-server, openssh-client
    update_cache: yes

- name: Create hadoopgroup group
  group:
    name: hadoopgroup
    state: present

- name: create the /hadoop_data directory
  file:
    path: /hadoop_data
    state: directory
    owner: hadoopuser
```

```
    group: hadoopgroup
  become: true

- name: create the hadoop_data/hdfs directory
  file:
    path: "{{ path_hdfs }}"
    state: directory
    owner: hadoopuser
    group: hadoopgroup
  become: true

- name: "Create path for HDFS Datanode"
  file:
    path: "{{ path_hdfs }}/datanode"
    state: directory
    owner: hadoopuser
    group: hadoopgroup
  become: true
  become_user: hadoopuser

- name: "Include hadoop_type"
  include_tasks: "{{ role_path }}/tasks/{{ hadoop_type_of_node }}.yml"

- name: "Copy ssh key to authorized_keys"
  authorized_key:
    user: hadoopuser
    state: present
    key: "{{ lookup('file', '/tmp/id_rsa.pub') }}"


- name: "Set mode for authorized_keys"
  file:
    path: /home/hadoopuser/.ssh/authorized_keys
    owner: hadoopuser
    group: hadoopgroup
    mode: '0600'

- name: Checking if a file exists
  stat:
    path: "{{ hadoop_home }}"
  register: file_data

- name: Report if a file exists
  debug:
    msg: "The file or directory exists"
  when: file_data.stat.exists

- name: "Download {{ hadoop_version }}"
  get_url:
```

```
    url: "https://ftp.cixug.es/apache/hadoop/common/hadoop-{{ hadoop_version
}}/hadoop-{{ hadoop_version }}.tar.gz"
    dest: "/opt/hadoop-{{ hadoop_version }}.tar.gz"
  when: not file_data.stat.exists

- name: extract and create hadoop directory
  become: yes
  unarchive:
    remote_src: true
    src: "/opt/hadoop-{{ hadoop_version }}.tar.gz"
    dest: /opt/
    list_files: yes
    owner: root
    group: hadoopgroup
    mode: "0755"
  when: not file_data.stat.exists

- name: Rename hadoop folder
  command: "mv /opt/hadoop-{{ hadoop_version }} {{ hadoop_home }}"
  when: not file_data.stat.exists

- name: delete the older extracted file
  file:
    path: "/opt/hadoop-{{ hadoop_version }}"
    state: absent

- name: "configure the environment"
  template:
    src: "{{ item }}"
    dest: '/opt/hadoop/etc/hadoop/{{ item }}'
    owner: root
    group: hadoopgroup
  with_items:
    - core-site.xml
    - hdfs-site.xml
    - mapred-site.xml
    - yarn-site.xml
    - workers
  notify:
    - restart hadoop
    - restart yarn

- name: Change permissions for container executor
  file:
    path: "{{ hadoop_home }}/bin/container-executor"
    mode: '6050'
    owner: root

- name: "Add JAVA_HOME and other environment variable"
  blockinfile:
```

```
    name: /opt/hadoop/etc/hadoop/hadoop-env.sh
    block: |
      export JAVA_HOME={{ java_home }}
      export HADOOP_HOME={{ hadoop_home }}
      export HADOOP_LOG_DIR={{ hadoop_home }}/logs
      export HDFS_NAMENODE_USER=hadoopuser
      export HDFS_DATANODE_USER=hadoopuser
      export HDFS_SECONDARYNAMENODE_USER=hadoopuser
      export YARN_NODEMANAGER_USER=hadoopuser
      export YARN_RESOURCEMANAGER_USER=hadoopuser
  notify:
    - restart hadoop
    - restart yarn

- name: "Add ENV to hadoopuser .bashrc"
  blockinfile:
    name: /home/hadoopuser/.bashrc
    block: |
      export HADOOP_HOME={{ hadoop_home }}
      export
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:$HADOOP_HOME/b
in:$HADOOP_HOME/sbin
      export HADOOP_CONF_DIR={{ hadoop_home }}/etc/hadoop
      export HDFS_NAMENODE_USER=hadoopuser
      export HDFS_DATANODE_USER=hadoopuser
      export HDFS_SECONDARYNAMENODE_USER=hadoopuser
      export JAVA_HOME={{ java_home }}
      export HADOOP_MAPRED_HOME={{ hadoop_home }}
      export HADOOP_COMMON_HOME={{ hadoop_home }}
      export HADOOP_HDFS_HOME={{ hadoop_home }}
      export YARN_HOME={{ hadoop_home }}
      export YARN_NODEMANAGER_USER=hadoopuser
      export YARN_RESOURCEMANAGER_USER=hadoopuser
  notify:
    - restart hadoop
    - restart yarn

- name: source bashrc hadoopuser
  shell: source /home/hadoopuser/.bashrc
  args:
     executable: /bin/bash

- name: "Create path for HDFS Datanode"
  file:
    path: "/opt/hadoop/logs"
    state: directory
    owner: hadoopuser
    group: hadoopgroup
  become: yes
```

```
- name: "Format Namenode"
  expect:
    command: /opt/hadoop/bin/hdfs namenode -format
    responses:
      (/.*)Re-format filesystem(.*): "Y"
  when: hadoop_type_of_node == 'namenode'
  become: yes
  notify:
    - restart hadoop
    - restart yarn
```

Defaults – mods-spark

```
---
# defaults file for mods-spark
spark_version: 3.3
mods_namenode: { name: test-ansible, ip: 172.16.64.6 }
spark_env: {"PYSPARK_PYTHON":"/usr/bin/python3"}
spark_defaults: {"spark.master": "yarn",
                 "spark.eventLog.enabled": "true",
                 "spark.eventLog.dir": "hdfs://{{ mods_namenode.name
}}:9820/user/hadoop/spark-logs"}
spark_home: /opt/hadoop/spark
```

Tasks – mods-spark

```
---
# tasks file for mods-spark
- name: Install requirements
  apt:
    name: openjdk-8-jdk
    update_cache: yes

- name: Download Spark and extract
  get_url:
    url: "https://dlcdn.apache.org/spark/spark-{{ spark_version }}.0/spark-{{
spark_version }}.0-bin-hadoop3.tgz"
    dest: "/opt/hadoop/spark-{{ spark_version }}.0.tgz"

- name: extract and create spark dir
  become: yes
  unarchive:
    remote_src: true
    src: "/opt/hadoop/spark-{{ spark_version }}.0.tgz"
    dest: /opt/hadoop/
    list_files: yes
    mode: "0755"

- name: Rename Spark dir
```

```
  command: "mv /opt/hadoop/spark-{{ spark_version }}.0-bin-hadoop3 {{ spark_home
}}"

- name: configure the environment
  template:
    src: spark-env.sh.j2
    dest: "{{ spark_home }}/conf/spark-env.sh"

- name: configure spark defaults config file
  template:
    src: spark-defaults.conf.j2
    dest: "{{ spark_home }}/conf/spark-defaults.conf"

- name: Add ENV to spark .bashrc
  blockinfile:
    name: /home/hadoopuser/.bashrc
    block: |
      export SPARK_HOME=/opt/hadoop/spark
      export PATH=$PATH:$SPARK_HOME/bin
```

ANEXO: *Scripts Python*

batchParquetHDFS.py

```
import findspark

from pyspark.sql import SparkSession

import time

import pandas as pd

spark = SparkSession.builder.getOrCreate()

df = spark.read.option("header", "true").csv("/user/data/merged.csv")

#Write in Parquet
tic = time.time()
df.write.parquet("/tmp/parquet/transaction_data.parquet")
parquet_write = time.time() - tic

print("################################### TIEMPO DE ESCRITURA DEL
PARQUET:",parquet_write)

#Read Parquet
tic2 = time.time()
dataParquet = spark.read.parquet("/tmp/parquet/transaction_data.parquet")
parquet_read = time.time() - tic2

print("################################### TIEMPO DE LECTURA DEL
PARQUET:",parquet_read)
```

batchParquetSWIFT.py

```
import pyspark
import findspark
from pyspark.sql import SparkSession
from delta import *
import time

spark = SparkSession.builder.master("l") \
    .config("spark.hadoop.fs.ceph.username","gomezan") \
    .config("spark.hadoop.fs.ceph.password","*******") \
    .config("spark.hadoop.fs.auth.method","keystone") \
    .config("spark.hadoop.fs.tenant.name","ifca.es:tfm_AngelGomez") \
    .config("spark.hadoop.fs.tenant.id","d0aa906b6e3d4201bebc7dd7a55ffcfd") \
    .config("spark.hadoop.fs.domain.name","IFCA") \
    .config("spark.hadoop.fs.auth.uri","https://api.cloud.ifca.es:5000/v3/auth/tokens") \
```

```
   .config("spark.hadoop.fs.s3a.connection.ssl.enabled","false") \

.config("spark.hadoop.fs.ceph.impl","org.apache.hadoop.fs.ceph.rgw.CephStoreSystem
") \
   .config("spark.jars.packages", "io.github.nanhu-lab:hadoop-
cephrgw:1.0.2,org.javaswift:joss:0.10.4")
   .appName("Parquet Block Storage").getOrCreate()


df = spark.read.option("header", "true").csv("/user/data/merged.csv")

#Write in Parquet
tic = time.time()
df.write.parquet("ceph://objectStorageTFM/transaction_data_parquet")
parquet_write = time.time() - tic

print("################################### TIEMPO DE ESCRITURA DEL
PARQUET:",parquet_write)

#Read Parquet
tic2 = time.time()
dataParquet =
spark.read.parquet("ceph://objectStorageTFM/transaction_data_parquet")
parquet_read = time.time() - tic2

print("################################### TIEMPO DE LECTURA DEL
PARQUET:",parquet_read)
```

batchDeltaHDFS.py

```
import pyspark
from delta import *
import time

builder = pyspark.sql.SparkSession.builder.appName("MyApp") \
        .config("spark.jars.packages", "io.delta:delta-core_2.12:1.2.1") \
   .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension") \
   .config("spark.sql.catalog.spark_catalog",
"org.apache.spark.sql.delta.catalog.DeltaCatalog")

spark = configure_spark_with_delta_pip(builder).getOrCreate()

data = spark.read.option("header", "true").csv("/user/data/merged.csv")

# Write in Delta
tic = time.time()
data.write.format("delta").save("/tmp/delta/transaction_data")
delta_write = time.time() - tic
```

```
print("################################### TIEMPO DE ESCRITURA DEL DELTA:",delta_write)

#Read Delta
tic2 = time.time()
df = spark.read.format("delta").load("/tmp/delta/transaction_data")
delta_read = time.time() - tic2

print("################################### TIEMPO DE LECTURA DEL DELTA:",delta_read)
```

batchDeltaSWIFT.py

```
import pyspark
import findspark
from pyspark.sql import SparkSession
from delta import *
import time

spark = SparkSession.builder.master("l") \
    .config("spark.hadoop.fs.ceph.username","gomezan") \
    .config("spark.hadoop.fs.ceph.password","*********") \
    .config("spark.hadoop.fs.auth.method","keystone") \
    .config("spark.hadoop.fs.tenant.name","ifca.es:tfm_AngelGomez") \
    .config("spark.hadoop.fs.tenant.id","d0aa906b6e3d4201bebc7dd7a55ffcfd") \
    .config("spark.hadoop.fs.domain.name","IFCA") \
    .config("spark.hadoop.fs.auth.uri","https://api.cloud.ifca.es:5000/v3/auth/tokens") \
    .config("spark.hadoop.fs.s3a.connection.ssl.enabled","false") \

.config("spark.delta.logStore.class","org.apache.spark.sql.delta.storage.S3SingleDriver
LogStore") \

.config("spark.hadoop.fs.ceph.impl","org.apache.hadoop.fs.ceph.rgw.CephStoreSystem
") \
    .config("spark.jars.packages", "io.delta:delta-core_2.12:1.2.1,io.github.nanhu-
lab:hadoop-cephrgw:1.0.2,org.javaswift:joss:0.10.4")
    .appName("Delta Block Storage").getOrCreate()


data = spark.read.option("header", "true").csv("/user/data/merged.csv")

# Write in Delta
tic = time.time()
data.write.format("delta").save("ceph://objectStorageTFM/transaction_data_delta")
delta_write = time.time() - tic

print("################################### TIEMPO DE ESCRITURA DEL DELTA:",delta_write)
```

```
#Read Delta
tic2 = time.time()
df =
spark.read.format("delta").load("ceph://objectStorageTFM/transaction_data_delta")
delta_read = time.time() - tic2

print("################################### TIEMPO DE LECTURA DEL
DELTA:",delta_read)
```