Procesamiento de datos

Práctica Obligatoria

Curso 2023-2024

INSO & MAIS

UTAD

Sumario

1 Introducción

Nuestra empresa *Procesamiento Big Data* ha sido contratada por un cliente para un proyecto para el análisis del tráfico aéreo en un contexto Big Data. Los datos de este proyecto se basan en 4 conjunto de datos:

- Países
- Aerolíneas
- Aeropuertos
- Rutas

Por otro lado, el cliente tiene un lago de datos compuesto por las siguientes tecnologías:

- Apache Spark
- HDFS
- Postgres
- Cassandra

Para la primera parte del proyecto, el cliente nos pide realizar una PoC (Proof of Concept) sobre una máquina virtual para comprobar la viabilidad del proyecto y realizar gran parte del desarrollo funcional.

Para el desarrollo de esta primera parte, se han planificado 3 fases:

1. Preparación del entorno de la máquina virtual
2. Ingesta de los datos en el lago de datos
3. Análisis de datos ingestados en el lago de datos

Para esta parte del proyecto, el cliente nos pide que presentemos un informe técnico en formato PDF en el que se especifiquen TODOS y cada uno de los **pasos** realizados para llevar a cabo estas 3 fases. Además, para aportar claridad y mejorar la calidad del documento, se nos pide incluir **capturas de pantalla** con todos los pasos llevados a cabo.

Para el desarrollo del proyecto, el líder técnico de nuestro proyecto ha generado una serie de tareas para cada fase y nos ha encomendado algunas de estas tareas técnicas:

2 Preparación del entorno de la máquina virtual

Partiremos de la máquina virtual utilizada en clase para realizar las siguientes tareas:

Preparación de Cassandra

Apache Cassandra no está instalado en nuestra máquina virtual. En este paso deberemos realizar su instalación.

Documentar y explicar la instalación de Apache Cassandra en nuestra máquina virtual.

Pasos a seguir:

bigdata> sudo apt-get update
bigdata> sudo apt install python2
bigdata> wget https://dlcdn.apache.org/cassandra/3.11.16/apache-cassandra-3.11.16-bin.tar.gz
bigdata> tar -zxvf apache-cassandra-3.11.16-bin.tar.gz
bigdata> rm apache-cassandra-3.11.16-bin.tar.gz

Desplegar HDFS

En este paso, vamos a utilizar el HDFS ya instalado en la máquina virtual.

Documentar y explicar cómo arrancar HDFS en la máquina virtual. Realiza alguna comprobación para asegurar que el servicio se ha levantado correctamente.

Desplegar postgres

En este paso, vamos a utilizar el Postgres ya instalado en la máquina virtual. Postgres se arranca por defecto al iniciar sesión en Ubuntu.

Documentar y explicar esta circunstancia. Arranca la shell de Postgres para realizar alguna comprobación.

Desplegar Cassandra

En este paso, vamos a arrancar Cassandra Service y a utilizar su shell.

Documentar y explicar cómo arrancar Cassandra Service y cómo utilizar la Cassandra Shell (cqlsh) para generar un Keyspace, ya que es un requisito para tareas posteriores.

Pasos a seguir:

- Arrancar Cassandra service

bigdata> cd apache-cassandra-3.11.16/
bigdata> bin/cassandra

- Iniciar Cassandra shell

bigdata> bin/cqlsh

- Crear Keyspace

cqlsh> CREATE KEYSPACE practica WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 1};

- Salir de la Cassandra shell

cqlsh> exit

- (SOLO A NIVEL INFORMATIVO para parar el servicio de forma ordenada al terminar la práctica) Parar Cassandra service

bigdata> ps auwx | grep cassandra | head -n1 | awk '{print $2}' | xargs sudo kill

Desplegar clúster de Spark Standalone

En este paso, se debe configurar y realizar el despliegue de un gestor de recursos para, en pasos posteriores, arrancar un clúster de Spark utilizando los recursos de este gestor de recursos.

Documentar y explicar los pasos para configurar y arrancar un despliegue de 1 Master y 2 Workers de Spark Standalone.
NOTA: Los recursos (cores y memoria) dependerán de los recursos que tenga la máquina virtual.

Accede a la WebUI de Spark Standalone para comprobar que el clúster se ha desplegado correctamente y explica en el informe la información que se muestra en esta WebUI.

Arrancar una shell de Spark

En este paso, se debe arrancar una shell de Spark formando un clúster de 2 executors a través del uso del gestor de recursos desplegado en el paso anterior.

Documentar y explicar los pasos para arrancar una shell de Spark indicando el master del gestor de recursos del paso anterior y con la configuración necesaria para tener 2 executors de tal forma que sus recursos (cores y memoria) coincidan con los de los 2 workers del gestor de recursos.

Antes de arrancar la shell, ten en cuenta que ni el conector JDBC de Postgres (https://jdbc.postgresql.org/download/postgresql-42.7.3.jar) ni el conector Spark de Cassandra (https://repo1.maven.org/maven2/com/datastax/spark/spark-cassandra-connector_2.12/3.3.0/spark-cassandra-connector_2.12-3.3.0.jar) se incluyen por defecto en la distribución de Apache Spark, por lo que deberás incluir sus artefactos JAR en el despliegue.

Accede a la WebUI de la Spark Application (no confundir con la WebUI de Spark Standalone) para comprobar que la shell ha levantado un aplicación de Spark con 2 executors y explica en el informe la información que se muestra en esta WebUI.

3 Ingesta de los datos en el lago de datos

En este apartado, se hace referencia a las tareas necesarias para volcar datos del proyecto en las diferentes fuentes de datos a través del uso de la aplicación de Spark (Shell).

Los datos utilizados en este proyecto están basados en el portal *Openflights*, por lo que se deberá visitar su web para obtener información adicional sobre ellos: https://openflights.org/data.php (si la web está caída, que ocurre con frecuencia, visitar https://web.archive.org/web/20230930101821/https://openflights.org/data.html). También se puede consultar su repositorio de GitHub para más información: https://github.com/jpatokal/openflights.

Los datos para este proyecto están comprimidos en el fichero *datos_practica.tar.gz*. Incluir este fichero en la máquina virtual y descomprimirlos en la carpeta *Descargas*.

A continuación se detallan las tareas que se deben realizar para la ingesta de estos datos en el lago de datos:

Countries

Los datos sobre países (fichero *countries.txt*) están en un formato no estándar (***name::<name> ## iso_code::<iso_code> ## dafif_code::<dafif_code>***), por lo que se va a tener que tratar como datos desestructurados (RDDs) y realizar una serie de transformaciones con el objetivo de realizar una limpieza de los datos y darle una estructura (***name: string, iso_code: string, dafif_code: string***) para obtener un Dataframe. Una vez obtenido el Dataframe, se debe utilizar el conector **csv** para guardar los datos en el path ***/practica/countries/*** de ***HDFS***.

NOTA: Ejemplo de transformación de una fila:

- Tipo:
  - ***String → Array[String]***
- Valores:
  - ***"name::Aruba ## iso_code::AW ## dafif_code::AA" → Array("Aruba", "AW", "AA")***

Documentar y explicar:

- La carga de los datos de *countries.txt* en un RDD de Spark (***sc.textFile...***)
- Las transformaciones necesarias para darle una estructura compatibles con una esquema de Dataframe
- La escritura de los datos de este Dataframe en la ruta **/practica/countries/** de **HDFS** en formato **csv** (***dataframe.write...***)
- Muestra 5 filas de estos datos una vez ya almacenados en **HDFS** (***spark.read...***)

Airlines

Los datos sobre aerolíneas (fichero *airlines.dat*) están en formato **csv**. Por tanto, estos datos se pueden cargar en un Dataframe con el conector correspondiente y las opciones que se requieran. Sin embargo, hay una de las columnas que requiere una transformación ya que, aunque la semántica es de tipo boolean, sus valores no son tienen la sintaxis para ello (*true/false*). Se trata de la columna *active*, la cual se deberá leer inicialmente como un *String*, pero a la cual se le deberá aplicar una serie de transformaciones para cambiar sus valores (*n, Y, N*) a un valor booleano (*true o false*) y así actualizar el esquema para que la columna *active* sea de tipo *Boolean*.

Además, se requiere enriquecer los datos de este Dataframe, añadiendo una nueva columa *country_iso* que corresponde al *iso_code* (datos **csv** en la ruta **/practica/countries/** de **HDFS**) correspondiente al país indicado en la columna *country*. Una vez obtenido este Dataframe, se debe utilizar el conector **parquet** para guardar los datos **particionándolos** por la columna *country* en el path **/practica/airlines/** de **HDFS**.

Documentar y explicar:

- La carga de los datos de *airlines.dat* en un Dataframe a través del conector de **csv** y las propiedades necesarias para ello (***spark.read...***)
- Las transformaciones necesarias para convertir los datos y el tipo de la columna *active* de *String* a *Boolean*
- Añadir al Dataframe una columna *country_iso* que corresponda al *iso_code* de cada país de cada fila. Dicho valor debe ser inferido de la columna *iso_code* de los datos almacenados en la ruta **/practica/countries/** de **HDFS** en formato **csv** (***dataframe.join...***)
- La escritura de los datos de este Dataframe en la ruta **/practica/airlines/** de **HDFS** en formato **parquet** y **particionando** por el campo *country* (***dataframe.write...***)
- Muestra 5 filas de estos datos una vez ya almacenados en **HDFS** (***spark.read...***)

Airports

Los datos sobre aeropuertos (fichero *airports.dat*) están en formato **csv**. Por tanto, estos datos se pueden cargar en un Dataframe con el conector correspondiente y las opciones que se requieran. Una de las columnas está en una unidad de medida que no es compatible con los sistemas del cliente. Esa columna es *altitude* y sus valores están en pies. Para realizar la transformación a metros, se debe desarrollar una **UDF** y ser aplicada en dicha columna. Una vez obtenido este Dataframe, se debe utilizar el conector **JDBC** para guardar los datos en la tabla **airports** de **Postgres**.

Además, el líder técnico de nuestro proyecto indica que, al hacer la lectura de estos datos, se deben añadir las propiedades *partitionColumn*, *lowerBound*, *upperBound* y *numPartitions* propias del conector **JDBC**.

Documentar y explicar:

- Desarrollo y registro de una **UDF** para convertir números enteros de pies a metros
- La carga de los datos de *airports.dat* en un Dataframe a través del conector de **csv** y las propiedades necesarias para ello (***spark.read...***)
- La utilización de la **UDF** anterior para convertir los valores de la columna *altitude* de pies a metros
- La escritura de los datos de este Dataframe en la tabla **airports** de **Postgres** a través del conector **JDBC** de Spark(***dataframe.write...***)
- Muestra 5 filas de estos datos una vez ya almacenados en **Postgres**, añadiendo los parámetros *partitionColumn*, *lowerBound*, *upperBound* y *numPartitions*. Justifica la elección de los valores elegidos para estas 4 propiedades y explica la utilidad de añadir estos 4 parámetros a una lectura de datos **JDBC** de Spark (***spark.read…***)

Routes

Los datos sobre rutas (fichero *routes.dat*) están en formato **csv**. Por tanto, estos datos se pueden cargar en un Dataframe con el conector correspondiente y las opciones que se requieran. Una vez obtenido este Dataframe, se debe utilizar el conector de **cassandra** para guardar los datos en la tabla **routes** del keyspace **practica**.

Documentar y explicar:

- La carga de los datos de *routes.dat* en un Dataframe a través del conector de **csv** y las propiedades necesarias para ello (***spark.read...***)
- La escritura de los datos de este Dataframe en la tabla **routes** del keyspace **practica** de **Cassandra** (***dataframe.write...***)
- Muestra 5 filas de estos datos una vez ya almacenados en **Cassandra** (***spark.read...***)

4 Análisis de datos ingestados en el lago de datos

En este apartado, se busca realizar una serie de consultas analíticas para demostrar al cliente cómo poder extraer información relevante para su caso de uso. Además, también se busca demostrar cómo persistir datos agregados en una nueva tabla para poder ser consultados de manera más rápida por herramientas de BI o de Dashboarding

Consultas

A continuación, se plantean una serie de preguntas con los datos que acabamos de volcar al lago de datos y que deben ser resueltas a través de operaciones analíticas con Apache Spark.

Además, para demostrar la versatilidad de Spark gracias a sus diferentes APIs, se pide responder a las preguntas utilizando tanto con la API de Dataframe (***spark.read...***) como con sentencias SQL (***spark.sql("CREATE TABLE ... ")*** + ***spark.sql ("SELECT ... FROM ...")***)

- ¿Qué aeropuerto está a mayor altitud (columna *altitude*)?
- ¿Cuántos aeropuertos hay en España (Spain)?:
- ¿Qué países tienen aeropuertos cuyo horario de verano (columna *dst*) sea el de Europa (E)?
- ¿Cuántas aerolíneas tiene en total EEUU (United States)?
- ¿Cuales son los 10 países con más aerolíneas inactivas (*active=false*)?
- ¿Qué países tienen aerolíneas en activo (*active=true*) y aeropuertos con una latitud (*latitude*) mayor a 80?

Persistir datos agregados

Por último, también queremos mostrar al cliente la capacidad que tiene nuestra plataforma para guardar datos agregados en tablas nuevas para poder realizar informes o consultar datos específicos con menor procesamiento y menor latencia.

Para ello, se necesita crear una tabla llamada *aggregations* que guarde sus datos en formato **parquet** en la ruta **/practica/aggregations/** de **HDFS** y que responda a la siguiente pregunta:

- ¿Cuántas rutas sin paradas (*stops*) a destinos con una altitud (*altitude*) mayor a 200 metros se hicieron por país (*country*) con aerolíneas que ya no están activas (*active*)?