# Proyecto 3: Orquestación

*Instituto Tecnológico Autónomo de México*
*Gilberto Iglesias*
*Andrea Fernández Conde*
*Lizbeth Contreras*
*Amanda Barrera*

*30/05/2015*

## Contents

# 1 Introducción

Generalmente se dice que el procesamiento a gran escala debe cumplir las tres V's: Volumen, Velocidad y Variedad. La necesidad de procesar información a gran escala de forma muy rápida ha llevado a la creación de una gran variedad de soluciones con diferentes características, lo anterior ha permitido que en la actualidad se pueda disponer de diversas herramientas para la implementación de procesos que impliquen el análisis de grandes cantidades de datos. Cada una de las herramientas desarrolladas hasta la actualidad cuentan con ventajas y desventajas que deben ser analizadas dependiendo el tipo de proyecto y análisis que se quiera desarrollar, lo anterior permitirá identificar la herramienta que mejor se adapte a las necesidades del proyecto a desarrollar. A continuación se describen las características de cada una de las herramientas utilizadas en el desarrollo de nuestro proyectos y la implementación que se realizó.

# 2 Pipeline de datos

## 2.1 Objetivo

Implementar el flujo para la base de datos de **GDELT**:

- Obtención:
  - Crawler
  - Identificar si hay nueva información
  - Bajar a zip
  - Guardar en disco
  - Descomprimir
- Limpieza:
  - Revisar columnas
  - Quitar columnas no requeridas
  - Estructurar las columnas en un orden apropiado
  - Realizar proceso de normalización de datos (e.g. fechas a UTC)
  - Enviar a un archivo de texto
- Manipulación:
  - Detectar que se escribió un archivo de texto y triggerear la subida al HDFS
  - Realizar un proceso de analítica que actualice la información
  - Tener una base de datos para shiny actualizada

## 2.2 Herramientas a utilizar

1. Sistema de carpetas / configuración / docker
2. Flume
3. Luigi
4. Spark
5. Sqoop
6. Hive/Impala

### 2.3 Implementación

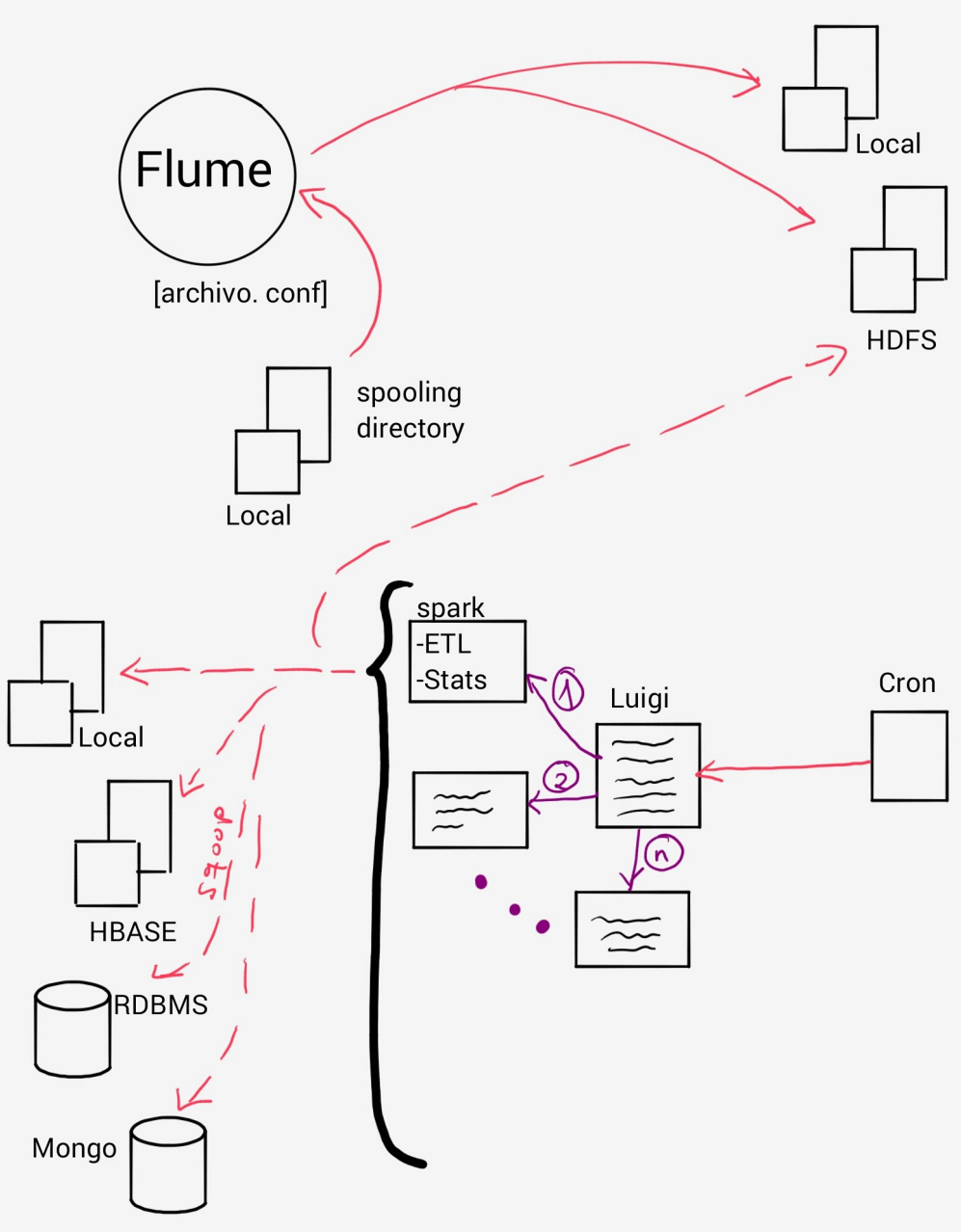



Figura 1: Flujo de datos implementado.

## 3 Configuración

### 3.1 Docker

Se utiliza el docker de clase disponible en https://registry.hub.docker.com/u/nanounanue/docker-hadoop/. Una vez bajada la imagen, se levanta el contenedor:

```
## Para correr el docker
# Levantamos el contenedor
sudo docker run -ti --name hadoop-pseudo \
    -v /home/animalito/Documents/hadoop_data:/home/itam/data \
    -p 2122:2122 -p 2181:2181 -p 39534:39534 -p 9000:9000 \
    -p 50070:50070 -p 50010:50010 -p 50020:50020 -p 50075:50075 \
    -p 50090:50090 -p 8030:8030 -p 8031:8031 -p 8032:8032 \
    -p 8033:8033 -p 8088:8088 -p 8040:8040 -p 8042:8042 \
    -p 13562:13562 -p 47784:47784 -p 10020:10020 -p 19888:19888 \
    -p 8000:8000 -p 9999:9999 \
    nanounanue/docker-hadoop

# Start: prende el contenedor
# Exec: te conecta al contenedor
sudo docker exec -it hadoop-pseudo /bin/zsh
```

Después de la ejecución del contenedor, se utiliza el usuario `itam` con contraseña `itam`.

### 3.1.1 Permisos

Es muy importante dar los permisos apropiados a las carpetas. En el usuario `itam` en docker ejecutamos:

```
### En el docker, preparamos para el flume
# Permisos para crear carpetitas, en itam
hadoop fs -chown itam /user/itam/datasets/gdelt
chmod -R 777 /opt
sudo chown -R itam /opt/gdelt
```

## 3.2 Carpetas de configuración

Para que todo funcione, primero se debe configurar una estructura de carpetas. Se corre entonces, en el docker, con usuario `itam` el script de bash `command_preparation_folders.sh`.

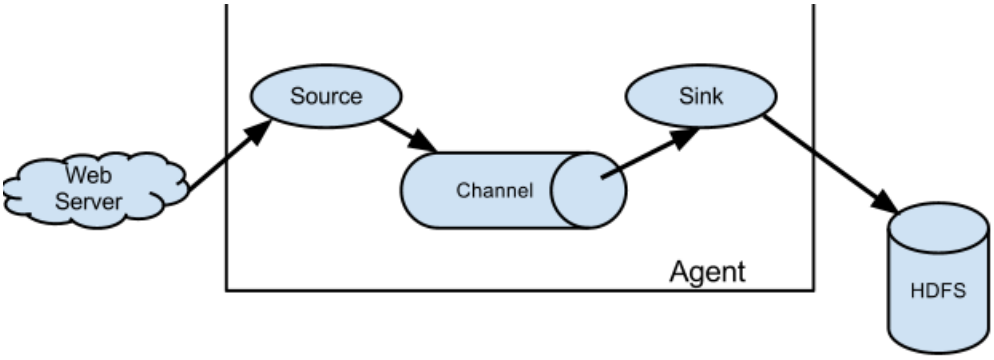
```
## En local (docker): borro todo
rm -R /home/itam/data/datasets/gdelt/flume_spooldir/*

rm -R /opt/gdelt/log/formato_avro/checkpoint/*
rm -R /opt/gdelt/log/formato_avro/data/*


rm -R /opt/gdelt/log/formato_normal/checkpoint/*
rm -R /opt/gdelt/log/formato_normal/data/*

rm -R /opt/gdelt/log/formato_normal_temp/checkpoint/*
rm -R /opt/gdelt/log/formato_normal_temp/data/*

## En local (docker): creo todos los directorios
mkdir -p /home/itam/data/datasets/gdelt/flume_spooldir
mkdir -p /opt/gdelt/log/formato_normal/checkpoint/
mkdir -p /opt/gdelt/log/formato_normal/data
mkdir -p /opt/gdelt/log/formato_normal_temp/checkpoint/
mkdir -p /opt/gdelt/log/formato_normal_temp/data
```

```
mkdir -p /opt/gdelt/log/formato_avro/checkpoint/
mkdir -p /opt/gdelt/log/formato_avro/data


## En el hadoop: borro todos los directorios
hadoop fs -rm -R /user/itam/datasets/gdelt/normal
hadoop fs -rm -R /user/itam/datasets/gdelt/avro
hadoop fs -rm -R /user/itam/datasets/gdelt/temp
hadoop fs -rm -R /user/itam/datasets/gdelt/resultados

## En el hadoop: creo todos los directorios
hadoop fs -mkdir -p /user/itam/datasets/gdelt/normal
hadoop fs -mkdir -p /user/itam/datasets/gdelt/avro
hadoop fs -mkdir -p /user/itam/datasets/gdelt/temp
hadoop fs -mkdir -p /user/itam/datasets/gdelt/resultados


## Para ver donde estan, busco la ruta en el docker para los primeros
## hadoop fs -ls directorio para ver que esta todo en el hadoop
```

# 4 Apache Flume


## 4.1 ¿Qué es?

`Flume` es un sistema distribuido y seguro para recoger, agregar y mover grandes volúmenes de datos provenientes de logs desde distintas fuentes a un almacén de datos centralizado. La arquitectura de `Flume` se puede dividir en seis partes: - Fuente externa. Se trata de la aplicación o mecanismo, como un servidor web o una consola de comandos desde la cual se generan eventos de datos que van a ser recogidos por la fuente. - Fuente. Una fuente es un componente que se encarga de recoger eventos desde la fuente externa y pasárselos transaccionalmente al canal. - Canal. Un canal es otro componente que actuará de almacén intermedio entre la fuente y el sumidero. La fuente será la encargada de escribir los datos en el canal y permanecerán en él hasta que el sumidero u otro canal los consuman. - Sumidero. Este componente será el encargado de recoger los datos desde el canal intermedio dentro de una transacción y de moverlos a un repositorio externo. - Repositorio externo. Nos sirve para almacenar en un sistema de ficheros como puede ser HDFS. - Interceptores. Serán una parte transversal de la arquitectura y podrán ser relacionados cuando ocurran distintos tipos de eventos en el flujo. Los interceptores podrán procesar los datos y añadirles la lógica que se necesite.



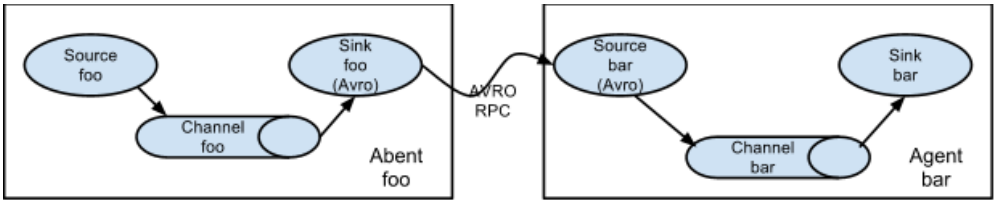

## 4.2 ¿Para qué se utiliza?

El uso de `Flume` no sólo se limita a la agregación de datos desde logs. Debido a que las fuentes de datos son configurables, `Flume` permite ser usado para recoger datos desde eventos ligados al tráfico de red, redes sociales, mensajes de correo electrónico a casi cualquier tipo de fuente de datos posibles. Flume soporta cuatro tipos de protocolos para leer datos:

- Avro
- Thrift
- Syslog
- Netcat

Los distintos componentes de un flujo de eventos en Flume deberán implementar algún tipo de cliente o servidor que sea compatible con alguno de los cuatro protocolos anteriores.

Al tratarse de una arquitectura modular podemos concatenar distintos flujos para hacer un sistema más complejo como puede verse en la siguiente imagen:




## 4.3 Ventajas y desventajas

Sus principales ventajas:

- Permite mover, de manera agregada y eficiente, grandes cantidades de datos de resgistro de muchas fuentes diferentes a Hadoop.
- Permite usar como entrada de datos un gran número de fuentes de eventos.
- Flume está pensado para procesos en streaming (intentando llegar a algo cercano a Real-time).
- Permite definir el flujo de ejecución de manera que podemos indicar los componentes por los que va pasando nuestra ejecución. Sus principales desventajas:
- cada nodo puede engrentar algunos problemas de rendimiento cuando la capacidad de procesamiento de datos del nodo se excede de forma inesperada debido a la enorme cantidad de carga de trabajo.
- Si la cantidad de datos transmitidos al nodo es demasiado pequño en comparación con su capacidad de procedamiento de datos, el nodo puede llegar a ser subtutilizado.‘

### 4.4 Implementación

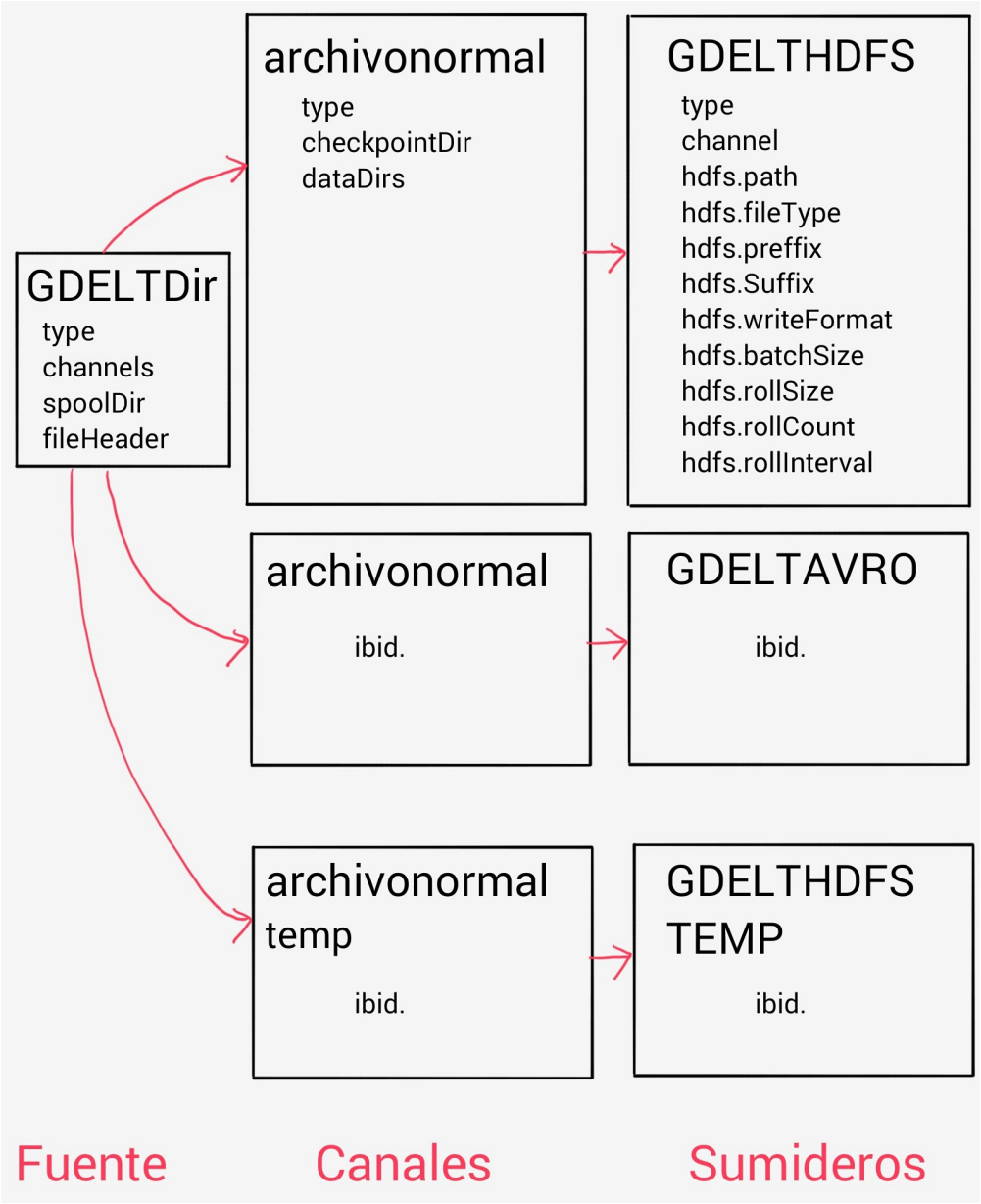



Figura 2: Fuente, canales y sumideros definidos en *gdelt_flume_agent.conf*.

#### 4.4.1 Configuración

Como se muestra en la figura 2, la configuración del agente de flume `gdelt` necesita que se especifiquen fuente, canales y sumideros. Asimismo, es necesario que, para cada uno, se definan mínimamente los atributos en el diagrama. Existen muchos atributos adicionales que pueden consultarse en la documentación.

El archivo de configuración utlizado en este ejemplo, se encuentra en `conf_files\/gdelt\_flume\_agent.conf` y se ve como sigue:

```
## Los definimos: cuales son (fuente, canales, sumideros)
```

```
GDELTAgent.sources = GDELTDir
GDELTAgent.channels = archivonormal archivoavro archivonormaltemp
GDELTAgent.sinks = GDELTHDFS GDELTAVRO GDELTHDFSTEMP

# Fuente e Interceptores
GDELTAgent.sources.GDELTDir.type = spooldir
GDELTAgent.sources.GDELTDir.channels = archivonormal archivoavro archivonormaltemp
GDELTAgent.sources.GDELTDir.spoolDir = /home/itam/data/datasets/gdelt/flume_spooldir
GDELTAgent.sources.GDELTDir.fileHeader = true

# Canales
GDELTAgent.channels.archivonormal.type = file
GDELTAgent.channels.archivonormal.checkpointDir = /opt/gdelt/log/formato_normal/checkpoint/
GDELTAgent.channels.archivonormal.dataDirs = /opt/gdelt/log/formato_normal/data/

GDELTAgent.channels.archivoavro.type = file
GDELTAgent.channels.archivoavro.checkpointDir = /opt/gdelt/log/formato_avro/checkpoint/
GDELTAgent.channels.archivoavro.dataDirs = /opt/gdelt/log/formato_avro/data/

GDELTAgent.channels.archivonormaltemp.type = file
GDELTAgent.channels.archivonormaltemp.checkpointDir = /opt/gdelt/log/formato_normal_temp/checkpoint/
GDELTAgent.channels.archivonormaltemp.dataDirs = /opt/gdelt/log/formato_normal_temp/data/

# Sumideros
GDELTAgent.sinks.GDELTHDFSTEMP.type=hdfs
GDELTAgent.sinks.GDELTHDFSTEMP.channel=archivonormaltemp
GDELTAgent.sinks.GDELTHDFSTEMP.hdfs.path = /user/itam/datasets/gdelt/temp
GDELTAgent.sinks.GDELTHDFSTEMP.hdfs.fileType = DataStream
GDELTAgent.sinks.GDELTHDFSTEMP.hdfs.filePrefix = GDELT-Normal-Data
GDELTAgent.sinks.GDELTHDFSTEMP.hdfs.fileSuffix = .temp
GDELTAgent.sinks.GDELTHDFSTEMP.hdfs.writeFormat = Text
GDELTAgent.sinks.GDELTHDFSTEMP.hdfs.batchSize = 10000
GDELTAgent.sinks.GDELTHDFSTEMP.hdfs.rollSize =  0
GDELTAgent.sinks.GDELTHDFSTEMP.hdfs.rollCount = 20000000
GDELTAgent.sinks.GDELTHDFSTEMP.hdfs.rollInterval = 180

GDELTAgent.sinks.GDELTHDFS.type=hdfs
GDELTAgent.sinks.GDELTHDFS.channel=archivonormal
GDELTAgent.sinks.GDELTHDFS.hdfs.path = /user/itam/datasets/gdelt/normal
GDELTAgent.sinks.GDELTHDFS.hdfs.fileType = DataStream
GDELTAgent.sinks.GDELTHDFS.hdfs.filePrefix = GDELT-Normal-Data
GDELTAgent.sinks.GDELTHDFS.hdfs.writeFormat = Text
GDELTAgent.sinks.GDELTHDFS.hdfs.batchSize = 10000
GDELTAgent.sinks.GDELTHDFS.hdfs.rollSize =  0
GDELTAgent.sinks.GDELTHDFS.hdfs.rollCount = 20000000
GDELTAgent.sinks.GDELTHDFS.hdfs.rollInterval = 180

GDELTAgent.sinks.GDELTAVRO.type=hdfs
GDELTAgent.sinks.GDELTAVRO.channel=archivoavro
GDELTAgent.sinks.GDELTAVRO.hdfs.path = /user/itam/datasets/gdelt/avro
GDELTAgent.sinks.GDELTAVRO.hdfs.fileType = DataStream
GDELTAgent.sinks.GDELTAVRO.hdfs.filePrefix = GDELT-Avro-Data
GDELTAgent.sinks.GDELTAVRO.hdfs.fileSuffix = .avro
GDELTAgent.sinks.GDELTAVRO.serializer = avro_event
```

```
GDELTAgent.sinks.GDELTAVRO.hdfs.batchSize = 10000
GDELTAgent.sinks.GDELTAVRO.hdfs.rollSize  = 0
GDELTAgent.sinks.GDELTAVRO.hdfs.rollCount = 20000000
GDELTAgent.sinks.GDELTAVRO.hdfs.rollInterval = 180
```

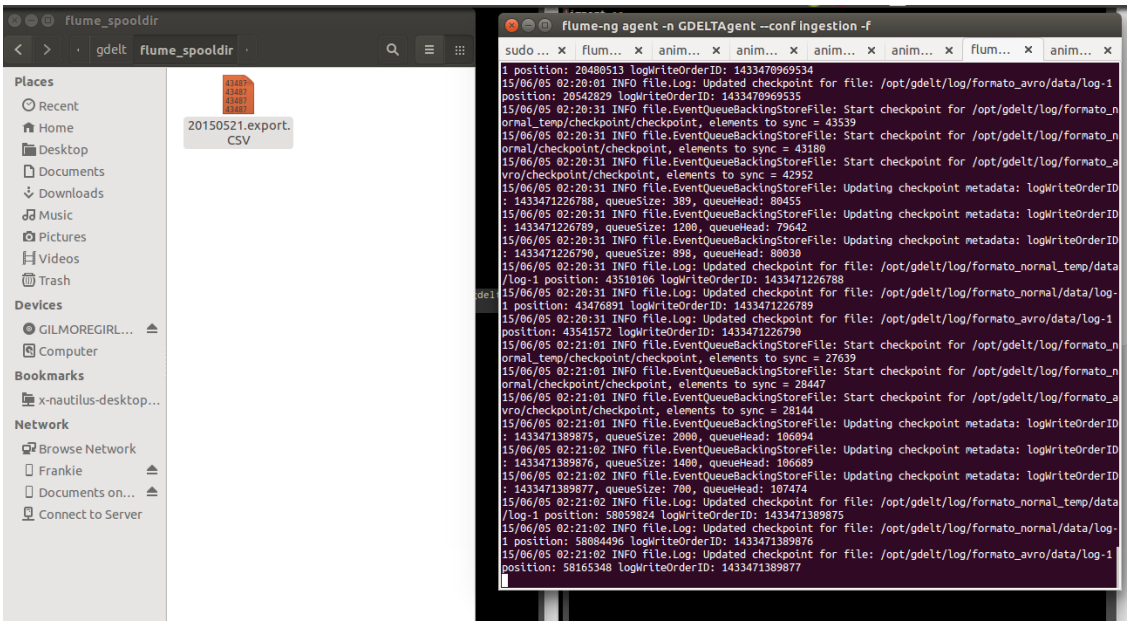
### 4.4.2 Ejecución

Estos archivos deben estar montados en una carpeta en local para el docker. De esta manera, con el usuario `itam`, ejecutamos el agente de flume de la siguiente manera:

```
### Para correr el flume:
## Paso 1
## Corremos flume desde docker, su itam
flume-ng agent -n GDELTAgent --conf ingestion -f /home/itam/data/conf_files/gdelt_flume_agent.conf
## Paso 2
## Copias el archivo que quieres trepar a la carpeta `flume_spooldir`
```

Con el agente de flume prendido, solo es necesario copiar los archivos en el `spoolDir` definido en la configuraciön para que el agente los procese e inserte en el HDFS y avro.

Si todo esta apropiadamente configurado, entonces al copiar un archivo de `gdelt` en el `spoolDir` local (sobre el cual se monta el `docker`) inmediatamente se activan los procesos del agente gdelt.



# 5 Spark

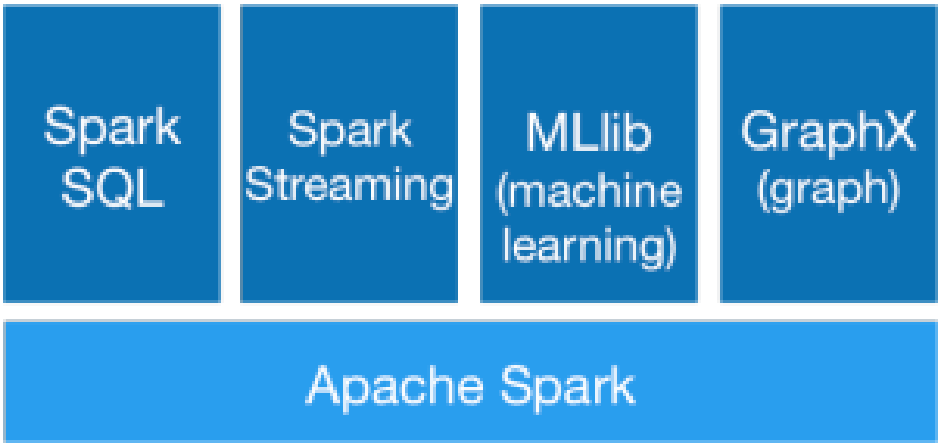

## 5.1 ¿Qué es?

`Spark` es una plataforma de computación de código abierto para análisis y procesos avanzados. Desde su principio `Spark` fue diseñado para soportar en memoria algoritmos iterativos que se pudieran desarrollar sin escribir un conjunto de resultados cada vez que procesaba un dato. Esta habilidad para mantener todo en memoria es una técnica de computación de alto rendimiento aplicado al análisis avanzado.

## 5.2 ¿Para qué se utiliza?

`Spark` es utilizado para implementar análisis avanzados, cuenta con un `framework` que incluye:

- La librería Mlib para implementar funciones para Machine Learning.
- El motor de gráficos GraphX.
- `Spark Streaming` para procesar en tiempo real grandes cantidades de datos.
- `Sparck SQL` para procesar consultas en `SQL`.




Esta plataforma asegura a los usuarios la consistencia en los resultados a través de distintos tipos de análisis.

## 5.3 Ventajas y desventajas

Sus principales ventajas:

- Capacidad de procesamiento en memoria, `Spark` tiene velocidades de procesamiento hasta 100 veces más rápidas que las conseguidas utilizando `MapReduce`.
- Esquema de computación más flexible que `MapReduce`.
- Se puede descargar y ejecutar desde un ordenador personal.
- Actualmente `Spark` es apoyado comercialmente por `Cloudera`, `Hortonworks` y `DataBricks`.
- Se pueden desarrollar aplicaciones en `Spark` utilizando `Java`, `Scala`, `Phyton` y `R`.
- Unificación del streaming en tiempo real.


- Permite integrarse con una gran cantidad de fuentes y repositorios de datos.


Sus principales desventajas:

- Consume mucha memoria
- Solo soporta los sistemas de archivo a través de `HDFS`

## 5.4 Implementación

En el archivo `workflows/pyspark_script.py` se especifican las tareas de limpieza que debe realizar el spark y que, después, serán ejecutadas por el orquestador.

```
import os
import csv
from io import StringIO
import sys
from pyspark import SparkContext, SparkConf


def load_tsv(archivo):
    return csv.reader(StringIO(archivo[1]), delimiter="\t")
```

```
def limpia_informacion(line):

   return  [line[0], line[1], line[3], line[2].replace(line[3],""),line[1].replace(line[2],"")] + \
        line[5:]

def main_limpia_info(args):
    try:
        conf = (SparkConf().setMaster("local").setAppName("GdeltImportingInfo"))
        sc = SparkContext(conf = conf)
        obs = sc.textFile("hdfs://localhost/user/itam/datasets/gdelt/temp/*.temp").\
            map(lambda x: x.split("\t") ).map(limpia_informacion)

     obs_mexico = obs.filter(lambda line: line[5] == "MEX" or line[15] == "MEX").map(lambda x: "\t".join(x)
        obs = obs.map(lambda x: "\t".join(x))
     obs_mexico.saveAsTextFile("hdfs://localhost/user/itam/datasets/gdelt/resultados/mexico")
       obs.saveAsTextFile("hdfs://localhost/user/itam/datasets/gdelt/resultados/general")
        #print obs_mexico.take(5)
        print "Ejecucion exitosa de limpieza\n"
    except:
        print "Ejecucion fallida de limpieza\n"

if __name__ == '__main__':
    #print "Ejecucion de PySpark"
    if sys.argv[1] == "limpieza_datos":
        main_limpia_info(sys.argv)
    #print "Fin de ejecucion"
```

# 6 Sqoop

## 6.1 ¿Qué es?

`Sqoop` es una librería que permite importar datos desde un almacenamiento de datos estructurado, como una base de datos relacional, a `Hadoop`. `Sqoop` también permite importar datos a otras bases de datos como `Hive` o `HBase`.

## 6.2 ¿Para qué se utiliza?

`Sqoop` suministra una herramienta desde línea de comando a través de la cual se puede realizar todo el proceso de importación y exportación de datos desde una base de datos relacional a un sistema de ficheros distribuidos y viceversa.

## 6.3 Ventajas y desventajas

Sus principales ventajas:

- Agiliza y facilita el movimiento de datos dentro y fuera de Hadoop.
- Sqoop está enfocado a bases de datos relacionales.
- Sqoop utiliza durante su ejecucción el paradigma MapReduce. Lo que permite procesar la información de manera paralela en procesos batch.
- Lanza directamente una o varias tareas MapReduce para procesar los datos.

Sus principales desventajas:

- Solamente realiza la validación a los datos copiados de una sola tabla en `HDFS`.

## 6.4 Implementación

En este caso, no se llamó a sqoop en el orquestador. Sin embargo, se extrae la base de datos a `hive` directo y se crean las tablas para realizar los queries. Los archivos de hive pueden ser llamados por el orquestador de manera directa y que el flujo esté completo.

Los ejemplos de hive y los scripts de creación de las tablas tomandolas del HDFS son `load_etled.q` para la base que ya pasó por el orquestador. Además, se puede cargar de manera directa la base normal o temporal del HDFS con `load_original.q`. El primero, se ve como sigue:

```
create table gdeltetled
(GLOBALEVENTID BIGINT,
SQLDATE INT,
Year INT,
Month INT,
Day INT,
FractionDate DECIMAL,
Actor1Code STRING,
Actor1Name STRING,
Actor1CountryCode STRING,
Actor1KnownGroupCode STRING,
Actor1EthnicCode STRING,
Actor1Religion1Code STRING,
Actor1Religion2Code STRING,
Actor1Type1Code STRING,
Actor1Type2Code STRING,
Actor1Type3Code STRING,
Actor2Code STRING,
Actor2Name STRING,
Actor2CountryCode STRING,
Actor2KnownGroupCode STRING,
```

```
Actor2EthnicCode STRING,
Actor2Religion1Code STRING,
Actor2Religion2Code STRING,
Actor2Type1Code STRING,
Actor2Type2Code STRING,
Actor2Type3Code STRING,
IsRootEvent BOOLEAN,
EventCode STRING,
EventBaseCode STRING,
EventRootCode STRING,
QuadClass INT,
GoldsteinScale DECIMAL,
NumMentions BIGINT,
NumSources BIGINT,
NumArticles BIGINT,
AvgTone DECIMAL,
Actor1Geo_Type BIGINT,
Actor1Geo_FullName STRING,
Actor1Geo_CountryCode STRING,
Actor1Geo_ADM1Code STRING,
Actor1Geo_Lat DECIMAL,
Actor1Geo_Long DECIMAL,
Actor1Geo_FeatureID DECIMAL,
Actor2Geo_Type BIGINT,
Actor2Geo_FullName STRING,
Actor2Geo_CountryCode STRING,
Actor2Geo_ADM1Code STRING,
Actor2Geo_Lat DECIMAL,
Actor2Geo_Long DECIMAL,
Actor2Geo_FeatureID DECIMAL,
ActionGeo_Type BIGINT,
ActionGeo_FullName STRING,
ActionGeo_CountryCode STRING,
ActionGeo_ADM1Code STRING,
ActionGeo_Lat DECIMAL,
ActionGeo_Long DECIMAL,
ActionGeo_FeatureID DECIMAL,
DATEADDED BIGINT,
SOURCEURL STRING)
row format delimited fields terminated by '\t' stored as textfile;

LOAD DATA INPATH '/user/itam/datasets/gdelt/resultados/mexico/part-00000' OVERWRITE INTO TABLE gdeltetled;
```

Ejemplos de los queries de analítica que se pueden realizar son:

```
CREATE TABLE actiontypes AS
select gdeltetled.actiongeo_type, count (DISTINCT gdeltetled.globaleventid) from gdeltetled group by gdelt
```

# 7 Orquestación vía Luigi

## 7.1 ¿Qué es?

`Luigi` es una herramienta de generación de `workflows` y `pipelines` de trabajo. Permite definir distintos tipos de tareas, así como las dependencias de ejecución entre ellas, además de disponer de una interfaz de visualización para comprobar el estado de la ejecución y de la finalización del `workflow` completo. Está escrito en `Python`, y tiene plantillas predefinidas para varios tipos de tareas.

## 7.2 ¿Para qué se utiliza?

Permite gestionar la dependencia entre tareas mediante una serie de funciones que se deben reescribir. Las más importantes son las siguiente:

- Run. Esta función contendrá el código principal de la tarea que se desea ejecutar.
- Requires. La función requires establece la relación de dependencia entre dos tareas.
- Output. La función output especifica la referencia al contenido que la tarea genera. Si se lanza varias veces el mismo `workflow`, antes de ejecutar cada una de las tareas, `Luigi` comprobará que no exista la referencia marcada en el `output`. En caso de que exista, la tarea se marcará como completada y pasará a la siguiente.

### 7.3 Ventajas y desventajas

### 7.4 Implementación

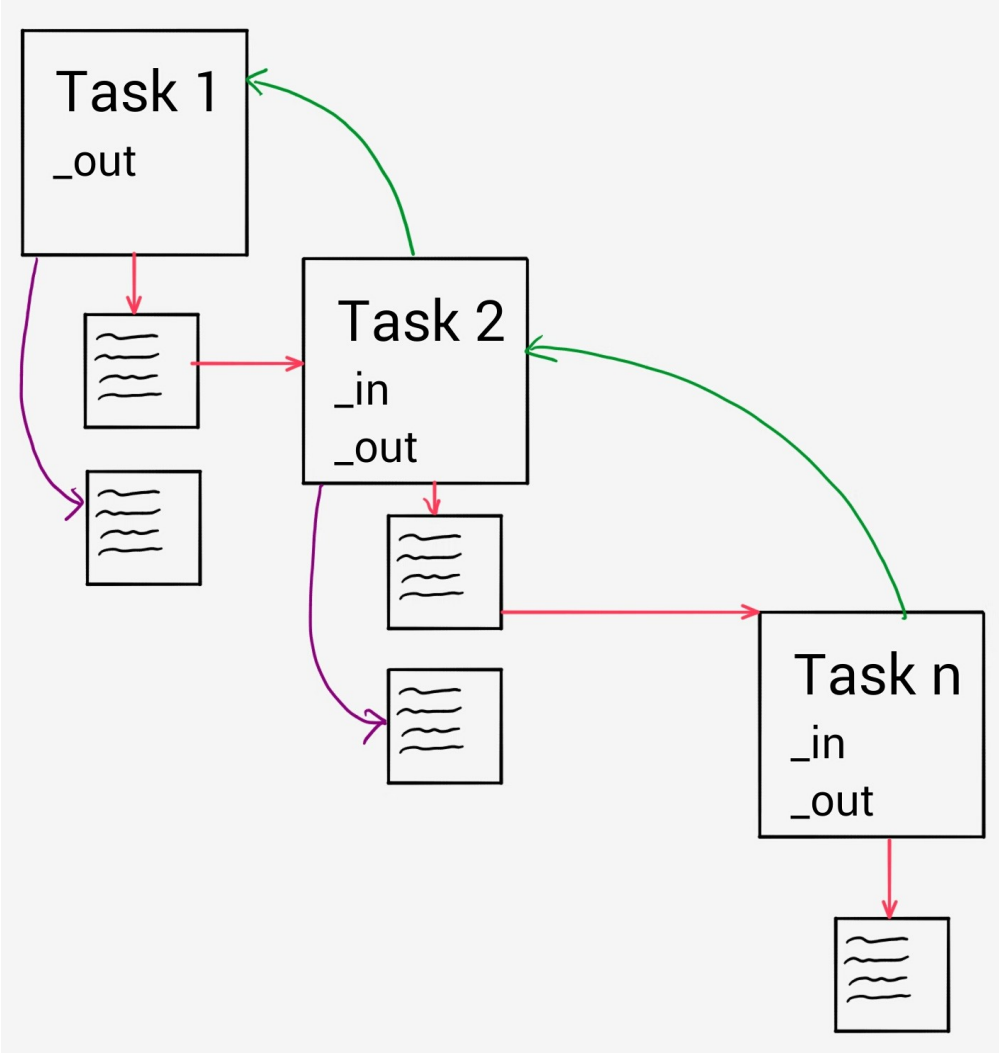



Figura 3: Esquema de las tareas implementadas en *orquestador_gdelt.py*.

# 8 Hive


## 8.1 ¿Qué es?

`Hive` es un sistema de almacén de datos que facilita el manejo sencillo de datos, consultas ad-hoc, y el análisis de grandes conjuntos de datos almacenados en sistemas de ficheros compatibles con `Hadoop`.

Algunas de las principales características de `Hive` y de su lenguaje `HiveQL` son las siguientes:

- `HiveQL` es un lenguaje tipo `SQL` que permite realizar consultas de grandes volúmenes de datos almacenados en un sistema de ficheros compatible con `Hadoop`.
- Las consultas realizadas desde `HiveQL` se ejecutan siguiendo el modelo `MapReduce`.
- `Hive` necesita almacenar metadatos y los esquemas de datos mediante un servicio metastore.
- El programador no necesita `HiveQL` ningún maper o reducer lo que agiliza el desarrollo. `Hive` se encarga de traducir la consulta escrita con `HiveQL` en tareas `MapReduce`.
- `Hive` permite que el programador pudiera escribir sus propios mares y reduces si fuera necesario
- `HiveQL` no permite inserción, actualización o borrado de datos a nivel de registro. Tampoco dota de transaccionalidad a sus consultas.
- Permite crear tablas y insertar datos que están almacenados en el sistema de ficheros de `Hadoop`.
- La latencia de las consultas suele ser mayor que las realizadas en las bases de datos relacionales debido a la inicialización de `MapReduce`.
- Schema on read vs schema on write. A diferencia de las bases de datos relacionales que garantizan que el `Schema` se cumple cuando se inserta un registro, `Hive` no garantiza esto aunque intenta garantizar el esquema en las lecturas.

## 8.2 ¿Para qué se utiliza?

Hive provee un mecanismo para dotar de estructura en los datos y realizar consultas sobre los mismos con el lenguaje tipo SQL llamado HiveQL. Al mismo tiempo este lenguaje también permite a los programadores de `MapReduce` incluir sus propios mappers y reducers cuando no sea conveniente o eficiente expresar esta lógica con HiveQL.

## 8.3 Ventajas y desventajas

Se recomienda utilizar `Hive` para el procesamiento secuencial de grandes archivos de datos multi-estructurados. Las principales ventajas de `Hive` son:

- Su capacidad de mejorar la simplicidad y la rapidez del desarrollo de `MapReduce`.

- Hace más sencillo el procesamiento de archivos relacionados entre sí.
- Utiliza queries similares a SQL.

Las principales desventajas de Hive son:

- No está completamente aislado del sistema de archivos subyacente, esto implica que con frecuencia el usuario requiere ayuda del optimizador con construcciones del lenguaje para procesar consultas más complejas.
- No puede sustituir a la funcionalidad, facilidad de uso, el rendimiento y la madurez de un DBMS.

## 8.4 Implementación

Desde python, utilizando luigi, es posible que, después de la ejecución del agente de flume, los procesos de limpieza, transformación y carga de datos (a hive, pig o postgres) sean realizados de manera automática. Cada tarea se ejecuta si (y solo si) la tarea previa fue ejecutada de manera exitosa. Debido a que la tarea posterior debe leer un archivo generada por la tarea previa, se decidió que cada tarea regresara por un lado los flujos de datos y, por otro, un archivo con el log de la tarea previa. En caso de que ésta es exitosa, entonces se realiza la siguiente. El código es como sigue:

```
import luigi
from luigi.hdfs import HdfsTarget
import os
from io import StringIO
import sys

class LimpiaInformacionTask(luigi.Task):

    def output(self):
        return HdfsTarget("/user/itam/datasets/gdelt/resultados/limpieza_info.txt")

    def run(self):
        _out = self.output().open("w")
        try:
        f = os.popen('nohup spark-submit /home/itam/workflows/pyspark_script.py limpieza_datos &')
            res_exec = f.read()
            _out.write("Ejecucion exitosa de LimpiarInformacionTask \n")
            _out.write(str(res_exec) + "\n")
            #print res_exec
        except:
            _out.write("Ejecucion fallida")
        _out.close()

class AnalizaInformacionTask(luigi.Task):

    def requires(self):
        return LimpiaInformacionTask()

    def output(self):
        return HdfsTarget("/user/itam/datasets/gdelt/resultados/analisis_info.txt")

    def run(self):
        #print "\n\n\nInicia ejecucion de analisis de informacion\n\n\n"
```

```
        try:
            _in = self.input().open("r")
            _out = self.output().open("w")
            for line in _in:
                _out.write(line)
                #print line + "\n"
            _in.close()
            _out.close()
        except:
            print "\nERROR  EN ANALISIS DE INFORMACION\n"

class ExportaInformacionTask(luigi.Task):

    def output(self):
        return HdfsTarget("/user/itam/datasets/gdelt/resultados/exportacion_info.txt")

    def requires(self):
        return AnalizaInformacionTask()

    def run(self):
        _in = self.input().open("r")
        _out = self.output().open("w")
        for line in _in:
            _out.write(line)

        _out.close()
        _in.close()


if __name__ == '__main__':
    luigi.run(main_task_cls=ExportaInformacionTask)
```

Este archivo está en `workflows/orquestador_gdelt.py`, se llama desde el docker con usuario `itam` y ejecuta también las tareas especificadas en pyspark.

# 9 Conclusiones

El flujo implementado es muy simple y se realizó de manera local. Sin embargo, la configuración es fácilmente extendible. Además, las tareas realizadas a través del orquestador pueden ser tan complicadas como un necesite.