UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

Trabajo de Fin de Grado

Alternativas de procesado de datos para presentación en Dashboards

Guillermo Hoyo Bravo
Javier Aracil Rico
Septiembre 2021

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

Agradecimientos

Gracias a mi tutor Javier Aracil Rico por brindarme esta gran oportunidad de compartir un proyecto con él, de abrirme el proyecto personalmente pensando en mi CV y mis expectativas de futuro, por estar atento y hacerme sentir seguro, y por ayudarme con todo y de todas las maneras que ha podido. Siempre respetándose y siendo muy cordial.

Gracias a Alejandro Peña Almansa, Graduado en la Universidad Autónoma de Madrid en Ingeniería de Telecomunicación, Máster en Ingeniería de Telecomunicación, y actualmente estudiante de PhD. en la UAM, por escucharme atentamente mejorando mi estado emocional y ayudarme con la comprensión lectora de mis expresiones.

Gracias a David Nevado: Licenciado en la Universidad autónoma de Madrid en Doble grado de Matemáticas e Informática y Licenciado en el Máster de Ciberseguridad: Por brindarme su apoyo moral

Por último, gracias a Raúl Torres Fernández: Estudiante de Ingeniería Informática de la Universidad Politécnica de Madrid y a Hugo Martins Ribeiro: Estudiante de Ingeniería Informática en la Universidad Complutense de Madrid, por ayudarme cuando más lo necesitaba y creía que no podía más o cuando me frustraba por el número de horas con un error típico y complicado de identificar en Python.

Resumen

Este Trabajo de Fin de Grado se enfoca en analizar diferentes métodos de indexar datos para crear Dashboards, de manera que se tomen decisiones más eficaces a la hora de realizar seguimientos de proyectos o empresas. Se analizará principalmente el tiempo que tarda cada una, además de evaluar otras características del proceso como la complejidad de este.

Estos procesos se realizan con Elasticsearch como indexador de los datos, donde se crearán dos índices diferentes desde dos tecnologías: Python & Logstash. Los índices se crearán a partir de los datos de la combinación de dos archivos .DUMP (Volcado de Memoria). Este archivo .DUMP será convertido a dos nuevos formatos: Un archivo JSON (JavaScript Object Notation), el cual será indexado por Python, y un archivo CSV (Comma-Separated Values), que será indexado mediante Logstash.

Una vez los datos sean indexados, se crearán dos Dashboards con Grafana. Estos Dashboards tendrán como Datasource los índices creados en Elasticsearch. De esta manera se comprobará que ambos índices se han generado correctamente, visualizando que la comparación de múltiples campos de estos dos Dashboards dan resultados idénticos.

Una vez realizado el proceso completo con archivos. DUMP reducidos y de prueba, se realiza el proceso completo midiendo el tiempo que tardan los distintos procesos en completar la tarea. Este tiempo no incluye la creación del Dashboard en Grafana. De igual modo, se realizarán pruebas mientras se ejecutan para ver la cantidad de recursos que pueden llegar a utilizar estos dos diferentes programas.

Python será el lenguaje de programación ocupado de realizar las transformaciones de datos, a la vez que de la medición de tiempos de los programas .py. Python también será el responsable de indexar el archivo JSON a Elasticsearch.

Logstash será el encargado de indexar el otro archivo, el archivo CSV. Para medir el tiempo de este proceso se usa el comando de la terminal: $ time ls -l /

El entorno de Web Services utilizado es instalado completamente desde Docker y Docker-Compose. Este TFG también cuenta con una interfaz gráfica para Elasticsearch llamada Dejavu que aporta seguridad y fluidez a los procesos.

Finalmente se visualizan los resultados y se extraen conclusiones acerca de qué proceso es más eficaz para indexar los datos.

Palabras clave

Dashboard, Elasticsearch, DUMP, CSV, JSON, Docker, Datasource, Indexar, Web Service, Dejavu, Logstash.

Abstract

This Final Degree Project focuses on analyzing different methods of indexing data to create Dashboards, so that more effective decisions are made when monitoring projects or companies. The main characteristic that was taken into account for the analysis is the time that each process lasted. In addition other features are analyzed such as the complexity of each process.

These processes will be carried out with Elasticsearch as the data indexer, where two different indexes will be created from two technologies: Python & Logstash. The indexes will be created from the data of the combination of two.DUMP (Memory Dump) files. This .DUMP file will be converted to two new formats: A JSON (JavaScript Object Notation) file, which will be indexed by Python, and a CSV (Comma-Separated Values) file, which will be indexed by Logstash.

Once the data is indexed, two Dashboards will be created with Grafana. These Dashboards will have as Datasource the indexes created in Elasticsearch. In this way, it will be verified that both indexes have been generated correctly, visualizing that the comparison of multiple fields of these two Dashboards gives identical results.

After the entire process is done, with the test and reduced.DUMP files, the entire process is performed by measuring the time it takes for the various processes to complete the task. This time does not include the creation of the Dashboard in Grafana. In the same way, tests will be carried out while they are running to see the amount of resources that these two different programs can use.

Python will be the programming language used to perform data transformations, as well as time measurement of .py programs. Python will also be responsible for indexing the JSON file to Elasticsearch.

Logstash will be in charge of indexing the other file, the CSV file. For measuring the time of this process, it is used the terminal command: $ time ls -l /

The Web Services environment used is completely installed from Docker and Docker-Compose. This TFG also has a graphical interface for Elasticsearch called Dejavu that provides security and fluidity to the processes.

Finally, the results are visualized and conclusions are drawn about which process is more efficient to index the data.

Key Words

Dashboard, Elasticsearch, DUMP, CSV, JSON, Docker, Datasource, Indexing, Web Service, Dejavu, Logstash.

1 Introducción

La Memoria de este TFG está dirigida a comparar los distintos archivos, y procesos o maneras de llegar a crear Dashboards para analizar datos de netflow con Grafana. Los procesos de análisis de datos requieren de grandes ficheros de datos, por lo que suelen tomar bastante tiempo, además de grandes cantidades de recursos.

Hoy en día este tema tiene una gran importancia, la mayoría de las empresas utilizan estas tecnologías para realizar seguimientos de Proyecto o de la misma empresa, pudiendo encontrar situaciones favorable o desfavorable y así poder controlarlas y provocarlas de manera más o menos frecuente

1.1 Motivación

La motivación principal de este TFG, teniendo en cuenta la importancia que tiene el análisis de datos actualmente, es la de agilizar su proceso. El análisis de datos es utilizado mundialmente, aporta muchos beneficios a los procesos o empresas que lo utilizan. Al aprender la manera más eficiente de crear este proceso, se puede incrementar el rendimiento de una empresa o de un proyecto.

Hace miles de años el ser humano empezó a recopilar información con dibujos en la pared, permitiéndole ordenar objetos, contabilizar, o administrar una sociedad. La información no es más que el conjunto de datos (o bytes en caso de la informática). Cuanto más se comprende un tema, más profundo puede volverse. A su vez, resulta más sencillo tratarlo o lidiar con él, y de igual modo, menos tiempo requiere de nuestra atención obteniendo mejores resultados.

Por ejemplo, si ocurren un problema por primera vez, este resulta complicado de resolver, se necesita investigar, aprender nueva información para lidiar con él, y requiere de nuestro tiempo y esfuerzo para resolverlo. Sin embargo, si ya se ha hecho frente a este problema o inconveniente más de una vez, esté resultará sencillo de resolver, casi automático.

En resumen, como dijo Francis Bacon precedido por muchos otros: “la Información es poder. Esto quiere decir que cuanta más información se recopile y se entienda, más posibilidades y beneficios se pueden obtener de ella. Para esto es para lo que se ha creado este TFG, para poder incrementar en la manera de lo posible, el modo de estudio de grandes cantidades de datos, obteniendo una evolución en los procesos deseados, o incluso en ciertos sectores.

Ejemplos de sectores que han evolucionado con el estudio de ingentes cantidades de información son: El marketing, que gracias al Big Data permite tomar decisiones estratégicas. El periodismo, gracias a un estudio intensivo de los datos de la audiencia, de las situaciones de los espectadores, y de sus reacciones, pasó a personalizar sus anuncios. Y la IA, básicamente necesita enormes Data Sets para entrenarse, evolucionar, y funcionar de la forma más perfecta posible.

Básicamente, igual que se aprendió a escribir hace miles de años, consiguiendo un gran desarrollo de la sociedad. La sociedad ha evolucionado gracias a la revolución

industrial, la invención de las máquinas, de los ordenadores y las bases de datos. Hoy en día, está volviendo a pasar, existe una nueva revolución industrial que va a hacer crecer a la sociedad de maneras inimaginables gracias a: las IAs, los Robots, el IOT (Internet de las Cosas), y el Big Data.

Objetivos

El objetivo de este TFG es llegar a una conclusión sobre qué proceso es más eficiente a la hora de preparar los datos para crear un Dashboard. De manera que aporte información útil para la toma de decisiones de este proceso tan utilizado y famoso hoy en día.

Primero se va a analizar el coste temporal de la conversión de dos archivos .DUMP a uno del tipo JSON. Siguiendo por el mismo proceso, pero convirtiendo estos dos archivos iniciales a uno del tipo CSV. Esto se va a realizar mediante el lenguaje de programación de Python, y va a permitir comparar ambos archivos para ver cuál de los dos es más eficaz o ligero para almacenar grandes cantidades de datos de manera estructurada.

Después de realizar esto, se comparan dos procesos de indexar datos a Elasticsearch. Elasticsearch es el Data Source que usa Grafana para poder crear Dashboards.

1. Una vez realizada la conversión de archivos DUMP a JSON, tomar este como fuente para alimentar un índice de Elasticsearch. Para ello se genera un programa en Python que se conecta a Elasticsearch mediante su librería, realizando la creación del índice y el volcado de datos del archivo JSON en este de manera controlada.

2. De igual manera que el primer proceso, con el archivo CSV se alimentará un nuevo índice de Elasticsearch. Este nuevo índice será generado mediante Logstash con un comando de la terminal, que toma como referencia el archivo logstash.conf para poder procesar los datos.

1.2 Organización de la memoria

La memoria consta de los siguientes capítulos:

- **Capítulo 1: Introducción**. En este capítulo se presentan los siguientes apartados: la motivación, los objetivos, y la organización de este trabajo.

- **Capítulo 2: Estado del Arte**. En este capítulo se realiza un estudio de las tecnologías que se utilizan, de los diferentes usos o resultados que ya se han obtenido con ellas, y de los conocimientos que se han necesitado para la realización del trabajo.

- **Capítulo 3: Diseño.** Este capítulo trata el diseño del proceso del trabajo, los requisitos funcionales y no funcionales de este, así como los diagramas de diseño de estos.

- **Capítulo 4: Desarrollo**. En este capítulo se desarrollan formalmente los procesos de desarrollo del trabajo. Se explican detalladamente los pasos seguidos para poder realizar la subida de los diferentes archivos datos desde Python y desde Logstash a Elasticsearch, para poder generar Dashboards. También se especifica cómo se realiza la medición de tiempos para los resultados finales.

- **Capítulo 5: Integración, Pruebas y Resultados**. A lo largo de este capítulo se explica la instalación del entorno de programación, las pruebas realizadas que aportan veracidad al proyecto, y los resultados obtenidos de las pruebas y del proceso completo.

- **Capítulo 6: Conclusión y Trabajo Futuro**. Este apartado sintetiza el trabajo, recopila las principales conclusiones y resultados del trabajo. También se proponen los futuros pasos que podría seguir este como trabajo de investigación.

2 Estado del arte

En este capítulo se recopila la investigación realizada para analizar las diferentes tecnologías que se han utilizado en el TFG. También se estudian otros trabajos similares relacionados con este.

2.1 Archivos de Datos

Los archivos de datos de este TFG contienen grandes cantidades de información. Así son los archivos requeridos para analizar datos. Vamos a utilizar 3 principalmente, con posibilidad de usar otros. Estos son:

DUMP o volcado de memoria. Este archivo es una réplica de la memoria de una base de datos, actualizada hasta el mismo momento en el que se ha realizado. Es decir, es una copia de seguridad de una base de datos. No contiene estructura alguna. Es la base de la que parte este proyecto. Ver anexo A.1.

JSON o JavaScript Object Notation [1], es un archivo de texto sencillo, se puede observar un ejemplo en el Anexo A.2. Está especializado en el intercambio de datos. Es muy famoso hoy en día por su rapidez y facilidad de uso. Tanto para humanos como para máquinas, resulta un lenguaje muy cómodo para escribir, leer, generar o procesar y convertir los datos. En el proyecto servirá como fuente de datos para indexar mediante Python a Elasticsearch.

###############Se borra la fotoo000!!!!

CSV o Comma-Separated Values [2]. Este un formato de archivo representa un array de valores numéricos y de texto. Es un archivo plano, como un archivo de Excel. Este archivo delimita los datos por columnas, separando cada campo por comas. La primera fila indica los nombres de los datos y el resto, debajo de cada "nombre", el dato de cada fila. Este archivo también funcionará como fuente de datos de un índice para elasticsearch que se creará mediante Logstash. Ver Anexo A,3

JSONL o JSON Lines. Es un archivo pensado para logs. Es muy similar a JSON, evoluciona de este. Es muy flexible para compartir mensajes entre procesos que cooperan. Guarda datos de manera estructurada para procesarlos uno a uno. Este archivo se puede indexar en Elasticsearch directamente desde terminal con un comando.

2.2 Archivos de Aplicaciones

YML (YAML -Ain't marking language): Es un archivo de texto plano, es comúnmente usado para archivos de configuración y aplicaciones donde los datos son guardados o transmitidos. Aporta gran legibilidad y estructura, mantiene un buen soporte para otros lenguajes como Python o JavaScript y para diferentes tipos de datos como pueden ser los diccionarios o arrays.

CONF (Configuration): Los archivos de configuración usados para la construcción o ejecución de varios procesos como: aplicaciones, sistemas operativos y de

infraestructuras de dispositivos. Estos pueden definir y personalizar sus parámetros, ajustes, opciones, y preferencias. Suelen mantener un formato de texto simple.

Conversores de Archivos:

Los conversores de archivos se crearán mediante Python. Son programas sencillos, que se han realizado infinidad de veces con muchos tipos de datos. Esto resulta en un proceso más simple en su programación. Por ejemplo, para transformar los archivos originales a JSON, dentro de la librería de JSON [3] existe una función para generar un volcado de datos específico. Otro ejemplo de la evolución de este proceso es la librería Panda [4] que tiene funciones para archivos XLSX y CSV [5].

Estos programas siempre tienen una entrada de datos, por ejemplo: el propio fichero DUMP. Sobre los que se operan hasta obtener el resultado deseado. En nuestro caso, la entrada es un array diccionario, que guarda los nombres de los campos como llaves y como valores del diccionario los datos que contienen estos dos archivos DUMP. De este modo, podemos generar un bucle que según las necesidades que se tengan genere de salida un tipo de archivo específico. En nuestro caso se generar en diferentes programas un JSON y un CSV como se ha mencionado anteriormente.

Además, estos conversores de archivos miden el tiempo que tardan en realizar el proceso mediante las funciones time de la librería: time [6]. Otra librería que ha sido necesaria ha sido la mencionada anteriormente: json, para poder utilizar la función mencionada de volcado de datos. Podemos encontrar los códigos de los programas en los apartados del Anexo B. Conversores de archivos. Anexo B.1. JSON.py , Anexo B.2: CSV.py.

Estos procesos completos, cuentan con varias pruebas que aseguran la ejecución correcta del programa y su correcto funcionamiento. No altera los datos ni su orden en ningún momento.

Python es el lenguaje de programación utilizado para desarrollar las funcionalidades necesarias para hacer funcionar este TFG. Más concretamente se ha utilizado su versión 3.8. Se ha utilizado para automatizar tareas, como la generación de ficheros convirtiendo los datos, o la indexación de uno de estos ficheros a la tecnología de Elasticsearch.

En este caso se ha programado con Python desde el entorno: Pycharm. Pycharm es un entorno de desarrollo diseñado específicamente para Python. Cuenta con múltiples opciones muy útiles, como una conexión directa con git a github u otra serie de plugin personalizados que amplían sus funcionalidades.

Para poder realizar el programa para indexar el archivo JSON desde Python, ha sido necesario la descarga de las dependencias de Elasticsearch [7]. Sin ellas, Python no puede conectarse al servicio web, además que cuenta con una función de volcado (bulk) que agiliza el proceso.

Este programa de indexación desde Python se basa en: medir el tiempo completo del principio a fin del proceso para futuro análisis. Conectarse al Servicio Web Elasticsearch. Cargar los datos del fichero JSON con lista. Finalmente, se utiliza la función que crea el índice y vuelca los datos de esta lista. Fue necesaria la investigación para encontrar la manera de conectar Python con Elasticsearch para crear un índice de manera sencilla. Para saber más ver el anexo C.1. Elastic.py

2.3 ELK (Elasticsearch, Logstash, Kibana) & Grafana

Elasticsearch [8] es un Servicio Web para la indexación de datos. Es principalmente reconocido por su velocidad y por su modo de trabajar con los datos. Elasticsearch puede crear un solo índice al cual se le ingresan datos simultáneamente desde diferentes fuentes. Estos datos pueden ser tratados antes de entrar al índice, obteniendo un servicio de gran calidad con todo tipo de datos, y visualizados posteriormente. Normalmente este proceso completo se realiza con la Pila ELK [9]. El proceso de esta pila está muy bien explicado en la figura 2.2.

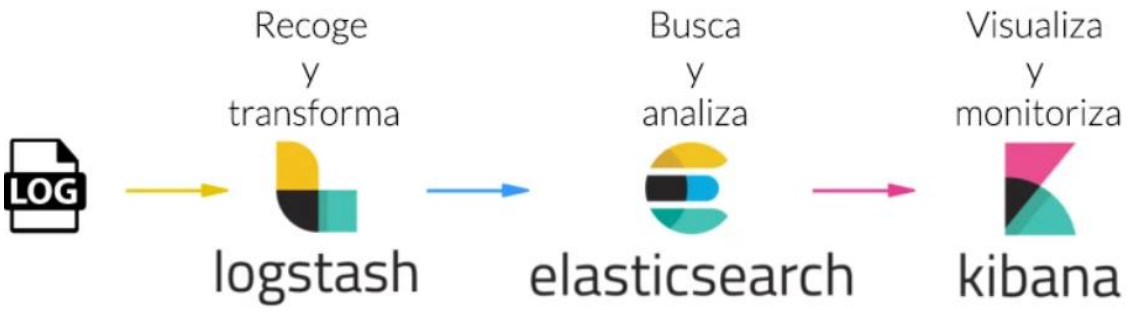

Figura 2.2 Explicación pila ELK visualmente

Elasticsearch es mayoritariamente utilizado como motor de búsqueda, aunque tiene otros usos como el de monitoreo de Rendimiento, o el de Analíticas de Logs.

Logstash [10] es un producto creado directamente por y para Elasticsearch. Logstash es el encargado de agregar, procesar y enviar los datos a los índices de Elasticsearch, permitiendo el tipo de ingreso de datos simultáneo mencionado con anterioridad.

Logstash hace uso de un archivo de configuración, donde se conecta a la URL de Elasticsearch, y se le indican las fuentes de los datos a indexar y el destino de estos, así como el proceso de transformación que van a sufrir. De este modo, cuando ejecutemos Logstash, los datos van de forma fluida al índice indicado. Este archivo en el TFG se llama **logstash.conf**, se puede ver en el Anexo C.2 Logstash.conf.

Kibana es una tecnología perteneciente a Elasticsearch de visualización de datos. Sirve para crear Dashboards e Interpretarlos de manera gráfica. Como Elasticsearch, Kibana también es Open-Source. Se suele conocer este trío de tecnologías como la pila ELK (Elasticsearch, Logstash, Kibana). Es muy famoso ya que tanto Logstash como Kibana

están preparados para trabajar con Elasticsearch. Su conexión suele ser complicada, pero gracias a Docker, esta pila se ha vuelto tan famosa y sencilla de configurar. Este tema se tratará más adelante.

Grafana [11] es otra tecnología Open-Source de visualización de datos. Como Kibana, sirve para crear Dashboards e Interpretarlos. Puedes añadir alertas para comprender mejor la aparición de situaciones favorables o desfavorables y tomar decisiones estratégicas.

Existe información acerca de cómo usar Elasticsearch como un Datasource para crear el Dashboard [12]. Igual que cómo personalizarlos, aunque cada Dashboard es único, así como la información que contienen. Se han usado tutoriales visuales de la plataforma YouTube para aprender a manejarnos de manera básica.

2.4 Control de Datos con Dejavu

Para aportar mayor veracidad a este proyecto, vamos a utilizar el Servicio Web de Dejavu como otra interfaz gráfica de Elasticsearch. Esta es más sencilla que las mencionadas anteriormente, nos sirve para importar y buscar los datos antes de pasarlos a un Dashboard con Grafana. Así conseguimos un mayor control de calidad, así como una gran fluidez de trabajo, consiguiendo una Interfaz Gráfica sencilla de los índices que creamos, algo muy importante de lo que carece Elasticsearch.

2.5 Docker para instalar los Web Services

Docker [13] es una plataforma Open Source de contenerización, donde se desarrollan y trasladan de aplicaciones muy fácilmente con contenedores. Permite a un nuevo desarrollador descargarse el entorno con las librerías y dependencias ya instaladas y configuradas en el mismo sistema operativo. Esta tecnología permite construir, ejecutar, parar y actualizar contenedores de imágenes muy rápidamente con comandos sencillos.

Docker se ha vuelto muy famoso por la facilidad de generar contenedores. Cuenta con una gran comunidad que sube a Docker-hub [14] múltiples contenedores conectados a un mismo host con la arquitectura completa de aplicaciones.

Docker-Compose [15] es la herramienta de Docker que permite este proceso. Docker-Compose, junta, personaliza y configura diferentes contenedores e imágenes en un simple archivo .yml, el cual es visible en el anexo D.1 Docker-compose.yml. Se puede ejecutar con un único comando, instalando el entorno completo. Entre las opciones de personalización de imágenes que nos permite este archivo, utilizaremos la de modificar el puerto de la imagen, y la de conectarse a otras imágenes o contenedores.

3 Diseño

En este capítulo, se cuenta cómo y porqué se han tomado las diferentes decisiones del diseño del trabajo. Por ello, cuenta con los requisitos funcionales y no funcionales que se han requerido, así como el diseño de diagramas de flujo de los procesos.

3.1 Requisitos

Primero vamos a comentar los requisitos que se han tomado en cuenta para desarrollar los diferentes programas del trabajo.

Requisitos Funcionales

- **RF1**: Se han de medir el tiempo que tarda en realizarse el proceso de transformación de datos de manera precisa. Se usará la librería time para ello en el grupo de programas de transformación de archivos, y el comando time para el grupo de programas de indexación de datos. Estos tiempos empiezan la medición desde el inicio de cada programa, hasta su finalización. Son procesos individuales.
- **RF2**: Se han de medir los recursos utilizados durante cada proceso. Este requisito pide el pico de uso de cada uno en cada proceso, para su futura comparación. Para ello se utilizará un script de bash desde el que se pueden observar estas mediciones actualizadas cada dos segundos. Este requisito medirá, como el anterior, los recursos de ambos procesos de datos explicados en los diagramas de las figuras 3.1 y 3.2.
- **RF3:** Se requiere la transformación de archivos de datos. Más concretamente se pide la unión de dos archivos DUMP transformados a un único archivo JSON y a un único archivo CSV. Ambos ficheros JSON y CSV deben consistir de los mismos datos.
- **RF4:** Es necesario la indexación correcta de los ficheros de datos (JSON y CSV). Este proceso ha de realizar la conexión a Elasticsearch, el parseo de los datos si es necesario, y la creación de un índice donde volcar los datos.
- **RF4.1:** La indexación de datos del fichero JSON se ha de llevar a cabo mediante un programa de Python. Los datos que contenga el fichero JSON han de ser los mismos que contenga el índice de Elasticsearch.
- **RF4.2:** La indexación de datos del fichero CSV se ha de llevar a cabo mediante Logstash creando un archivo de configuración. Los datos que contenga el fichero CSV han de ser los mismos que contenga el índice de Elasticsearch.
- **RF5:** Los Web Services se montarán con contenedores de Docker. Permitiendo su conexión localmente. Igualmente, Logstash sirve como entrada de datos para un índice de Elasticsearch para el fichero CSV, de igual manera que Python es la entrada de datos para otro índice de Elasticsearch con el fichero JSON. Grafana se conecta a Elasticsearch permitiendo así usar los índices creados como Datasource para la creación de Dashboards. Dejavu también se conectará a

Elasticsearch permitiendo la visualización de índices aportando al proyecto fluidez, seguridad y profesionalidad.

Requisitos no Funcionales:

- **RNF1:** Se dispone de un ordenador portátil de 4GB de RAM con un procesador de 7ª generación con 2,6 GHz y un disco (Partición) SSD de 65 GB.
- **RNF2:** Se requiere de la creación de Dashboards en Grafana para comprobar que los datos de ambos índices son iguales. De manera indirecta se prueba que los ficheros JSON y CSV también contienen los mismos datos. Esto aporta veracidad a los procesos de indexación y de Transformación de archivos simultáneamente
- **RNF3:** Se crean Dashboards en Grafana donde se pueda medir el proceso completo. Estos Dashboards están esparcidos en el tiempo pudiendo ver la cantidad de datos que se han indexado cada minuto o segundo. De manera que se pueda ver el rendimiento con los componentes y recursos del ordenador. Este gráfico aportará viabilidad al proyecto y veracidad a las conclusiones obtenidas.
- **RNF4:** Se realizará el proceso con varios grupos de ficheros idénticos para probar la capacidad del ordenador.
- **RNF5:** Se recopilan las mediciones obtenidas en tablas de manera que se pueda estudiar esta información con facilidad. Más concretamente, se requieren tablas de mediciones de tiempo y recursos (consultar índice de tablas).
- **RNF5.1:** Las tablas 5.1 y 5.3 para comparar el tiempo de que tardan los procesos (Tanto con archivos de prueba como los reales), y el tamaño de estos ficheros.
- **RNF5.2:** Las tablas 5.2 y 5.4 para comparar el tiempo de Indexación de los archivos desde Python y desde Logstash (Tanto con archivos de prueba como los reales) y el máximo de los recursos utilizados para el proceso (únicamente para los archivos reales).
- **RNF5.3:** La tabla 6.1 de las conclusiones obtenidas de comparar los procesos completos. Esta tabla muestra los resultados finales de los procesos con los archivos originales. Se habla del tiempo que tarda cada uno, de los recursos utilizados por cada uno, de la complejidad que ha supuesto cada proceso, y de otros factores que se han considerado relevantes. De este modo, se sintetizan los conocimientos obtenidos consiguiendo un acceso sencillo para consultar la información.

Diagramas de Flujo de los Procesos.

Se pasa a explicar el diseño de los tres procesos principales realizados para poder llegar a los resultados y conclusiones del proyecto. Para ello, se explican tres diagramas, y ciertas decisiones que se han tomado por lo que son así.

Proceso de transformación de archivos.

Como se puede ver, en el diagrama de la Figura 3.1, Se explica el proceso de transformación de Archivos, el código se encuentra en el anexo B. En este se ha tomado la decisión de realizar dos programas casi idénticos para cada tipo de archivo que queremos obtener.

Esto se ha decidido así para poder hacer pruebas sin necesidad de entradas manuales, obteniendo mejores resultados tanto en mediciones de tiempo como de rendimiento.

Los procesos se basan en juntar ambos archivos DUMP de entrada en un diccionario, para luego volcar los datos en el nuevo tipo de archivo seleccionado. Se comprueba que la longitud del diccionario después de añadir un archivo completo, tiene el mismo número de líneas de datos que el este, o que ambos si es el segundo archivo. Así se comprueba que tenga todos los datos.

De igual modo, se realiza una prueba con archivos DUMP muy reducidos, visualizando los resultados manualmente. Otra prueba que se realiza es comparar el último elemento del diccionario con el último del fichero que se acaba de terminar de añadir.

Mientras se ejecutan estos procesos, la función time() de la librería time realiza una medición del tiempo que tarda cada proceso, para su futuro estudio,

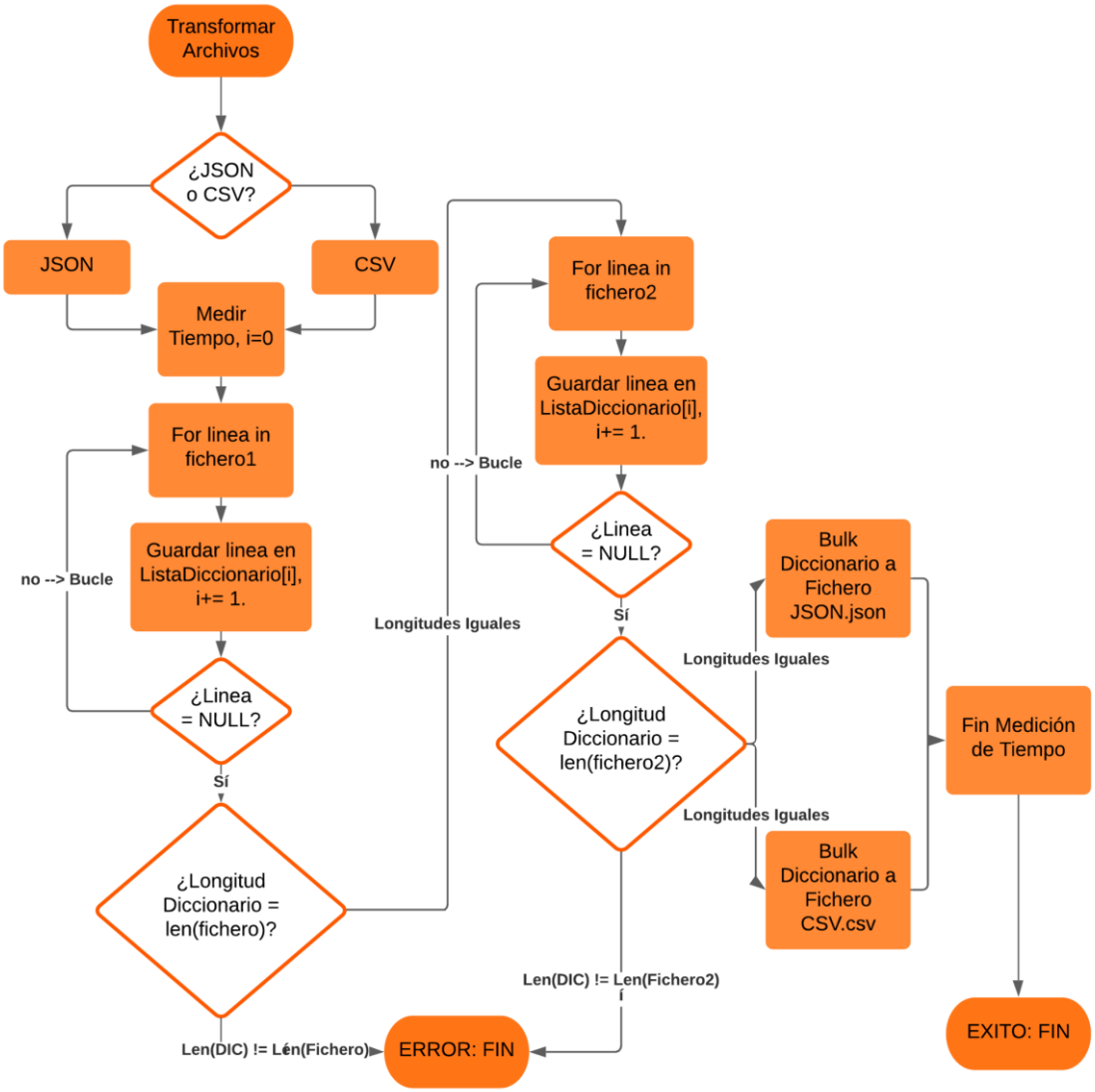

Figura 3.1 Diagrama de Flujo del proceso de transformación de archivos.

Proceso de Indexación de Archivos

Este diagrama de flujo contemplado en la Figura 3.2 explica ambos procesos de indexación de datos en Elasticsearch [16]. Estos procesos se basan en la conexión al servicio de Elasticsearch, en la lectura de los ficheros de datos y en la creación de índice con estos. Además de realizar las mediciones correspondientes.

Para realizarlo, se ha decidido que la mejor medición de tiempo posible se realiza mediante el comando time de la terminal de Ubuntu. En el proceso anterior se utiliza la función time() de la librería time de Python. En este caso, para que el sistema de medición encaje en ambos procesos, se ha optado por este comando de terminal. Se puede encontrar el código de ambos programas en el anexo B completo.

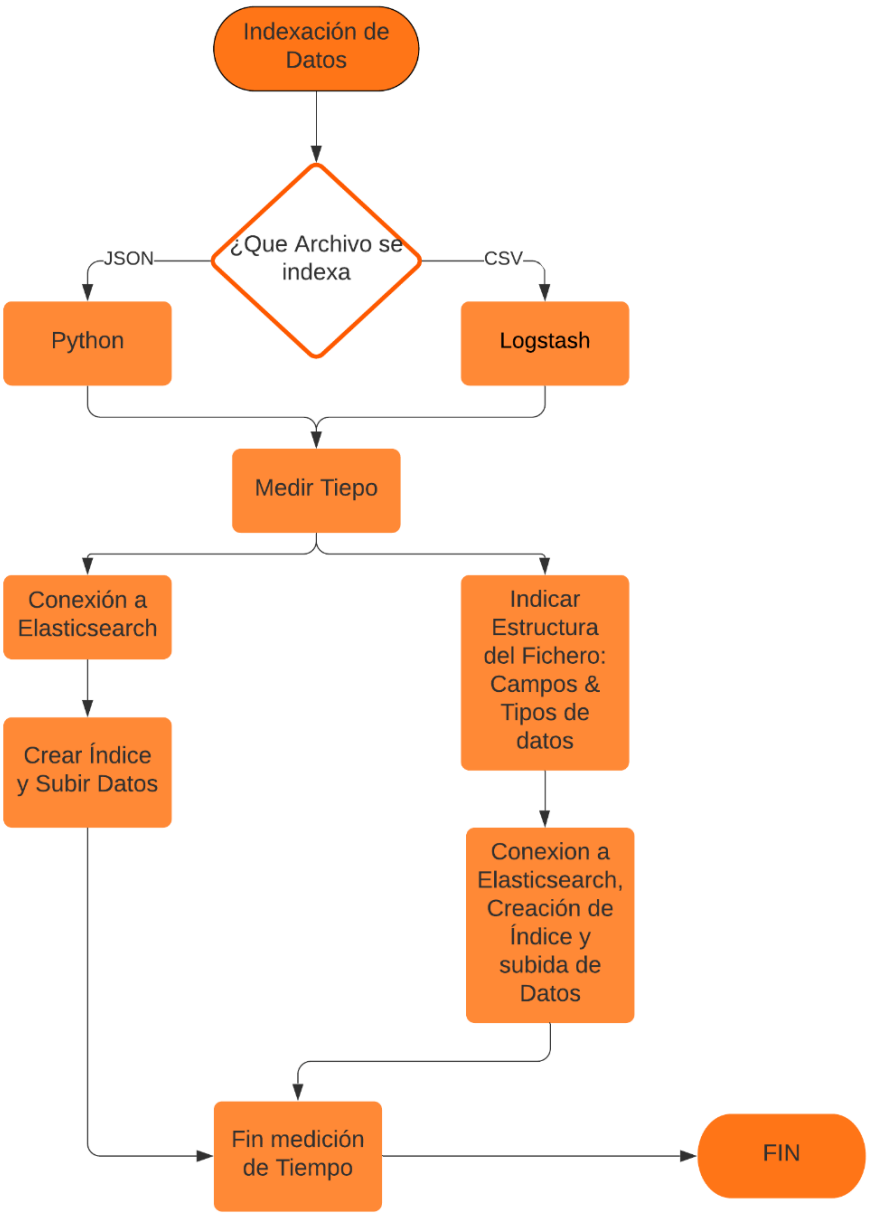

Figura 3.2 Diagrama de Flujo de los programas indexadores de datos.

Proceso completo de Medición de Datos para futuras conclusiones

Este diagrama de Flujo visible en la Figura 3.3 explica el proceso completo que asegura la calidad del trabajo con diversas pruebas. Se dejan fuera del diagrama las pruebas más pequeñas realizadas entre procesos, algunas visibles en las Figuras 3.1 y 3.2.

El proceso completo se basa en medir los recursos que utilizan los programas mediante un script que se auto recarga cada 1 segundos, obteniendo los picos de estos para poder compararlo en su máximo punto de estrés. El script se encuentra en el anexo D.2. Se

valora la opción de comparar los procesos con el programa system monitor que incluye Ubuntu. Se han realizado pruebas donde se ve que es menos exacto, por ello se utiliza el script.

Ahora, se realiza la transformación de archivos, comprobando que el número de líneas de datos en ambos es igual. En caso de que no sean iguales, significa que un proceso no funciona correctamente, por lo que habría que parar el proceso de medición para arreglar los posibles fallos.

Se continúa indexando datos. Una vez terminadas las indexaciones, la medición del uso de recursos se para. Ahora comprobamos visualmente la correcta conexión a los índices y de sus datos desde la Interfaz Gráfica de Dejavu. Esto sirve para detectar fallos grandes u obvios que se hayan podido generar en este proceso.

Viendo que el proceso es correcto, pasamos a realizar la comprobación de la veracidad de los procesos con Grafana. Para ello, se crean en Grafana dos Datasource desde cada índice utilizado, con el campo de referencia el más estándar del fichero.

Ahora, se generan los mismos Datasource, con las mismas opciones para los mismos campos. Si ambos reaccionan del mismo modo, mostrando resultados idénticos, significa que el proceso está completado y es completamente correcto.

Esto significa que se pasa al proceso de comparación de los resultados de las mediciones obtenidas. Se comparan los tiempos de ambos sets de programas, los recursos utilizados por ambos, y también, los tamaños de los ficheros en el apartado de transformación.

Finalizando con las conclusiones, apoyándose en estos resultados estudiados, así como en unos nuevos Dashboards generados desde ambos índices como Datasource, pero con el campo de referencia el campo @timestamp que se refiere al tiempo real en el que se ha subido el dato al índice.

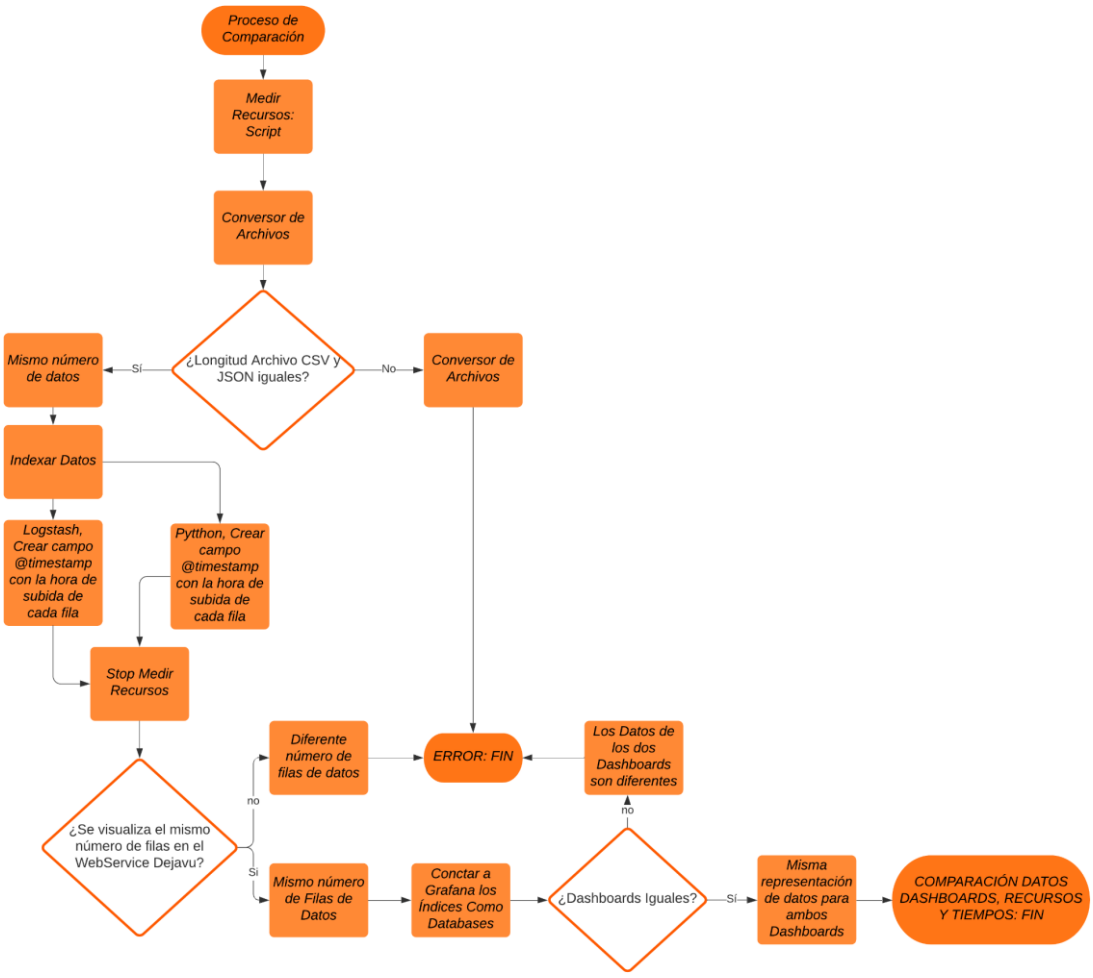

Figura 3.3 Diagrama de Flujo del Proceso práctico completo

4 Desarrollo

En este capítulo se explica paso a paso cómo se han llegado a los resultados de este trabajo. Para ello se habla sobre la generación del entorno de trabajo y su porqué, las herramientas utilizadas y cómo se han usado. Siguiendo por las mediciones hechas para comparar. Finalizando los procesos de los programas generados.

4.1 Entorno

A continuación, se va a describir los detalles del entorno, las decisiones tomadas para seleccionarlo y sus detalles.

- **SO: Ubuntu**

  Este TFG ha sido desarrollado en el Sistema Operativo Linux, más concretamente en Ubuntu 20.04 – LTS. Este proceso se realiza con el programa de Rufus realizando un dual boot. Linux es mucho más manejable para realizar mediciones del estado del ordenador y para programar y configurar entornos.

  Dentro de Ubuntu hemos instalado directamente desde terminal los programas de, Git, Pycharm, Logstash, Docker, Docker-Compose.

- **Git**

  Git [17] es una herramienta de Open Source distribuido de control de versiones, es usado para la cooperación de proyectos. Desde el inicio del proyecto hasta el final de este, Git ayuda a guardar los avances realizados, a volver a versiones anteriores de manera sencilla por si el proyecto ha resultado dañado, erróneo o perjudicado.

  Git ha sido muy útil durante el proyecto completo. Se ha utilizado desde el inicio guardando los archivos .DUMP, hasta el final guardando los resultados finales. Ha ayudado a tener un progreso controlado y monitorizado, ya que se han realizado muchas pruebas con diferentes resultados.

  Del mismo modo, al igual que los resultados, también se han guardado y controlado los programas que nos los han generado. Tanto los conversores de datos como los programas de indexación han pasado por cambios. Ha sido útil hasta para desarrollar el entorno de programación, ya que los Web Services de este han dependido de un archivo Docker-Compose.yml.

- **Pycharm y el Lenguaje de Programación Python**

  Pycharm [18] es el protagonista del trabajo. Ha sido utilizado para todo lo posible para desarrollar el trabajo con el lenguaje de programación Python 3 [19]. Es un editor de código creado específicamente para Python, por lo que está muy bien preparado para realizar un desarrollo profesional. También tiene conexión directa a Git, facilitando el proceso de control de versiones.

  Se ha usado en el proceso de comparación de tiempos ya explicado anteriormente, siendo la indexación de Python uno de los dos procesos principales a comparar

mediante dificultad, tiempo y cantidad de recursos. También se ha realizado el proceso completo de conversión de archivos.

Python viene instalado por defecto en Ubuntu, Pycharm se puede descargar e instalar directamente desde el centro de Software de Ubuntu.

Las librerías principales usadas para este trabajo son:

- JSON: Principalmente esta es una librería para parsear los datos del JSON en un diccionario o lista, o al revés. Pudiendo así pasar los datos del diccionario creado a partir de los archivos .DUMP a un JSON directamente. Es la manera más sencilla para convertirlo.

- TIME: Este módulo o librería nos presta relojes y funciones para poder obtener fechas. Obteniendo el tiempo local al inicio del programa antes de su finalización, podemos medir con precisión y exactitud el tiempo que tarda el proceso en ejecutarse.

- ELASTICSEARCH: para poder conectarnos a Elasticsearch desde python, es necesario descargar e instalar su cliente localmente. Con ello creamos un entorno donde podemos trabajar con ambas tecnologías juntas. También cuenta con funciones específicas de Python para interactuar con el cliente de Elasticsearch y realizar el proceso de indexación.

- **Docker & Docker-Compose**

  Docker [20] es la evolución de un proyecto de almacenamiento de aplicaciones web y bases de datos. Docker fue lanzado con un increíble éxito en 2015. Hoy, Docker tiene el banco de imágenes y contenedores más grande del mercado, Docker-hub. Gracias a esto este proyecto se ha ahorrado tiempo, generando un contenedor con los Web Services de Grafana, Elasticsearch, y Dejavu. Estas imágenes se instalan y generan con el comando $ Docker-compose up -d.

  Este comando ejecuta el archivo Docker-compose.yml, al que se hace referencia en el Anexo…Este archivo YML contiene la versión en la que está escrito, a continuación, el nombre y versión de una imagen, continuando con una personalización de esta. Se pueden acceder a modificar múltiples opciones, en este caso, lo principal será la conexión a otras imágenes y los puertos en los que se despliegan los servicios.

  Podemos instalar Docker entrando es su página web y siguiendo los pasos descritos. Hay varias maneras de hacerlo, pero es un proceso muy sencillo. Lo mismo ocurre para Docker Compose.

  Se ha decidido dejar fuera del archivo YML la imagen de Logstash porque Python viene instalado por defecto. La función de este TFG es realizar una comparación realista entre tecnologías en un mismo entorno. Tener una de las dos tecnologías principales a comparar dentro de un contenedor del Docker [21], y la otra directamente instalada en

el ordenador, puede resultar contraproducente para los resultados finales, o alterarlos haciendo que el resultado sea expuesto como inverosímil.

- **Elasticsearch**

  Elasticsearch [22] es uno de los motores de búsqueda más famosos y competitivos del mercado. Utiliza el motor de búsqueda de Lucene. Utiliza también una licencia de Apache, esto lo relaciona estrechamente con Grafana, este Servicio Web también tiene una licencia Apache.

  Por su fama y competitividad en el mercado es que se ha decidido usar Elasticsearch. Además de por sus cualidades, como la posibilidad de indexar o añadir grandes volúmenes de datos y sus funcionalidades de API. Está orientado a archivos JSON, ofreciendo un gran número de librerías integradas para los lenguajes de programación, resultarnos útil la de Python.

- **Logstash**

  Logstash es desarrollada directamente por Elasticsearch. Este es un motivo principal de su uso, se compara una herramienta local de la aplicación central que se utiliza (Elasticsearch), con una tecnología como Python, básica para un programador con capacidades ilimitadas la cual ha demostrado su potencial infinidad de veces.

- **Dejavu**

  Dejavu [23] es una interfaz gráfica creada para Elasticsearch. Se ha decidido utilizarla porque es muy sencilla y rápida de utilizar para monitorizar los datos y realizar pruebas con los índices. Elasticsearch carece de una interfaz gráfica decente, y la que vamos a utilizar para realizar las pruebas finales, Grafana, es bastante más compleja que Dejavu.

  Además, está preparado para importar archivos JSON o CSV, los que vamos a utilizar.

  Dejavu será nuestro punto intermedio de seguridad para comprobar que la fase de programación y de medida de resultados ha terminado, faltando únicamente la prueba final que añade la veracidad científica de este TFG.

- **Grafana**

  Grafana fue creada en 2014, hoy en día gracias a su comunidad y su suma de plugin es una herramienta mucho más potente. Grafana está escrita con Node.js LTS, y preparada para la visualización de Datos Temporales.

  Tiene una funcionalidad muy peculiar, correr en modo TV, para que el analista pueda enseñar, directamente desde la herramienta, lo que ha aprendido de estos datos. También contiene muchos atajos con el teclado para su uso eficiente.

  Su flexibilidad, visibilidad y dinamismo diferencia Grafana de otras tecnologías similares, como de Kibana, y por la que la hemos seleccionado para este trabajo.

4.2 Programas

- **Convertidores de Archivos**

  Esta es la base del proyecto. Para realizarla se han comprendido los archivos de datos con los que se trabaja y sus estructuras.

  Para convertir los archivos utilizamos Python como ya se ha explicado. Esta parte consistirá en dos programas, uno para cada archivo de datos, de este modo se pueden realizar pruebas simultáneas con los dos archivos a la vez.

  Es un proceso simple, hay algunas excepciones de formatos que se han encontrado. Por ejemplo, el formato JSON requería de una pequeña diferencia para poder subirse a Elasticsearch desde Python.

  De igual manera, al juntar los dos diccionarios se creaba un (\n) extra entre ambos, realizando una pausa si se intentaban indexar los datos. Por eso las pruebas son tan importantes y se han realizado tan intensivamente en la base del proyecto.

- **Indexadores de Datos**

  Esta sección es la base de las comparaciones. Es el objetivo y de donde salen los resultados más notables del proyecto.

  Empezando por el programa de Python, se ha comprendido la librería de Elasticsearch, consiguiendo grandes resultados de una manera simple y compacta. Resumidamente se realiza una petición para conectarnos como clientes del Web Service Elasticsearch.

  Una vez obtenida, se abre el archivo de datos deseado, en este caso el JSON, y se ejecuta la función de la librería que permite realizar un volcado de datos. Para realizar este volcado de datos, es necesario cargar el fichero JSON en una lista. Así funciona la función helpers.bulk() utilizada.

4.3 Métodos de Mediciones

- **Tiempo**

  Cronómetro: Esta opción se valoró como método alternativo y global para cualquier proceso. Por el rango de fallo humano quedo descartada completamente.

  Comando Time: Esta opción se utiliza para medir el tiempo que tarda la indexación desde Logstash. Por ser coherentes con las mediciones, el programa de indexación creado con Python también coge los tiempos ofrecidos por este comando.

Librería time, función time(). Esta función es la principal encargada de medir los tiempos. Es útil, sencilla, precisa y visual. Se utiliza en todos los programas .py de este TFG.

- **Rendimiento**

System Monitor: Para medir el uso de la CPU, así como la RAM en porcentajes, se ha pensado usar este programa interno de python. El programa actualiza la información rápidamente, y es muy visual. Se descarta debido a que muestra los porcentajes por procesos, y es posible que Logstash, al utilizar Java, se divida en varios procesos complicados.

Comando top y htop: De manera muy similar, estos comandos [24] muestran información muy similar por terminal. Por los mismos motivos se prefirió utilizar otro método.

Bash Script: Se crea un Script que muestra por terminal cada 2 segundos la información del porcentaje de uso completo de la CPU. Para realizar las mediciones de los recursos, se leen los valores de CPU, RAM y Disco iniciales. Estos se monitorean mientras se realizan los procesos. De esta manera, se obtiene el pico de uso de los recursos. Se puede consultar el Anexo D.2 para ver sus detalles

Con las mediciones iniciales y las mediciones en los puntos máximos, se puede calcular la diferencia obteniendo el uso de los procesos.

Se descarta medir los recursos que utilizan las pruebas realizadas a partir de los archivos DUMP reducidos por la escasez de información aportada y la instantaneidad de los procesos.

5 Pruebas y Resultados

5.1 Pruebas

- **Pruebas del entorno**

Para comprobar que funcione el entorno se han utilizado diferentes comandos, que verifican su correcto funcionamiento.

- **Programas Instalados Localmente (Desde Terminal)**

  Python no necesita comprobación, porque viene instalado de manera predeterminada con Ubuntu 20.04. de igual modo, nos sirve con abrir la aplicación de pycharm para comprobar su funcionamiento.

  Para comprobar las instalaciones de **Docker** y **Docker-compose**, se han usado los siguientes comandos.

  ```
  [USER@]:~$ docker -v
  ```

Respuesta: Docker version 20.1.3, build 48d30b5

[USER@]:~$ docker-compose --version
Respuesta: docker-compose version 1.28.2, build 67630359

Para comprobar la correcta instalación de Logstash usamos los siguientes comandos de esta manera determinada:

Primero obtenemos la ubicación del programa (los comandos de Logstash son únicamente ejecutables desde su dirección)

[USER@]:~$ sudo whereis logstash.
Respuesta: logstash: /etc/logstash /usr/share/logstash

Ahora nos movemos a su carpeta bin desde la que ejecutar los comandos y utilizamos el comando de versión como en Docker.

[USER@]:~$ sudo cd /usr/share/logstash/bin
Respuesta: /usr/share/logstash/bin

[USER@]:~$ sudo ./logstash –version.

Para ejecutar el archivo Logstash.conf que indexa los datos se utiliza este comando:

[USER@]:~$ sudo ./logstash -f logstash.conf

Para medir el tiempo:

[USER@]:~$ sudo time ./logstash -f logstash.conf

EJEMPLO: time ls -l /

- **Web Services instalados con Docker (Containers)**

  Aquí se enseña las pruebas para comprobar el correcto funcionamiento de los Web Services instalados desde docker-compose:

  - **Comandos de Control:** Con ellos podemos comprobar o modificar el estado del Web Service.

    Ejemplo del comando de control de estado. Los comandos para modificarlo son iguales cambiando status por start, stop o restart.

    [USER@]:~$ sudo systemctl status <WebService>

    En este caso: <WebService> = Logstash

    Respuesta: ● logstash.service - logstash

    Loaded: loaded (/etc/systemd/system/logstash.service; disabled; vendor pre>

```
     Active: active (running) since Mon 2021-05-17 17:30:23 CEST; 5s ago
   Main PID: 18984 (java)
     Tasks: 28 (limit: 9336)
    Memory: 470.0M
    CGroup: /system.slice/logstash.service
            └─18984 /usr/share/logstash/jdk/bin/java -Xms1g -Xmx1g -
XX:+UseCon>
```

- **URLs:** Modo de acceso a los Web Services y sus funciones

  La URL general para acceder a un Web Service es:

  http://localhosr:<PortNumber>

  Vamos a acceder a la URL de cada uno, obteniendo los siguientes resultados.

  Url Elasticsearch: En la Figura 5.1 podemos observar el contenido que aparece al acceder a Elasticsearch, si este está activo y funcionando: la URL es: http://localhost:9200/


```
{
  "name" : "5743ccb234d5",
  "cluster_name" : "docker-cluster",
  "cluster_uuid" : "G07X6_r3TSy3ufvRL7oQTA",
  "version" : {
    "number" : "7.0.1",
    "build_flavor" : "oss",
    "build_type" : "docker",
    "build_hash" : "e4efcb5",
    "build_date" : "2019-04-29T12:56:03.145736Z",
    "build_snapshot" : false,
    "lucene_version" : "8.0.0",
    "minimum_wire_compatibility_version" : "6.7.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```


Figura 5.1. Funcionamiento correcto del web Service Elasticsearch. Donde se encuentra la información básica de este.

- **Url Dejavu**: En la Figura 5.2 podemos observar el contenido que aparece al acceder a Dejavu, si este está activo y funcionando. La URL es: http://localhost:1358/

Figura 5.2. Funcionamiento correcto del Web Service Dejavu. Se muestra la página principal con los tips de conexión y algunas opciones.

- **Url Grafana**: En la Figura 5.3 podemos observar el contenido que aparece al acceder a Grafana, si este está activo y funcionando: la URL es: http://localhost:3000/

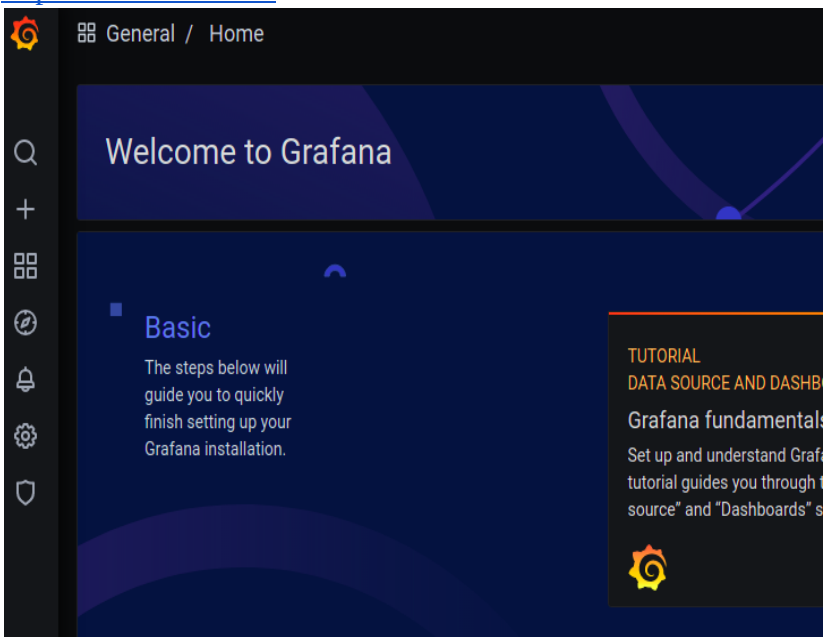

Figura 5.3 Funcionamiento correcto del web Service Grafana. Se muestra la página principal una vez iniciada sesión, con el menú de opciones a la izquierda. Inicio de sesión con usuario y contraseña: admin, admin.

- **Pruebas Procesos**

  Lo primero que se ha hecho para probar el proceso diseñado, ha sido crear dos archivos de datos .DUMP, con información de los ficheros originales y que nos generarán los resultados finales. De este modo ahorraremos mucho tiempo implementando los programas, probándolos, y comparándolos con pequeñas alteraciones para encontrar el mejor resultado.

  Pruebas Conversión de Archivos

  Se han hecho pruebas generando los Conversores.py. Al guardar un archivo completo en el diccionario, se comparó este al diccionario desde cero, de modo que si se encuentra alguna alteración en el diccionario añadiendo el primer archivo. DUMP o el segundo se pueda sanear rápidamente.

  También se comprobó por terminal cada línea que se añadía al realizar el bulk al nuevo archivo de datos. De este modo se fue comprobando que los datos mantenían el formato correcto, sin añadir algún (\n) adicional que estropeara el proceso.

  Una vez generado el nuevo archivo, se comprobaba una a una las líneas de este archivo con las respectivas filas de los archivos de prueba DUMP. Asegurando el éxito con los archivos de datos masivos que son los originales.

  Además, ambos archivos imprimen el número total de elementos transformados. Al ejecutar ambos podemos comprobar este número viendo que ambos tienen el mismo número de filas de datos.

  Pruebas Indexadores de Archivos

  Una vez se han indexado los datos, es momento de comprobar si el proceso se ha realizado correctamente. Para ello, se puede mirar la salida de cada fila de datos subido por terminal. Esto es bastante caótico, además de que ralentiza el proceso.

  Otro modo de hacerlo es usar los comandos de Elasticsearch desde la terminal. Primero se buscan los índices creados con el comando, como se muestra en la Figura 5.4.

  ```
  $ curl -XGET 'localhost:9200/_cat/indices?v'
  ```

  ```
  health status index    uuid                   pri rep docs.count docs.deleted store.size pri.store.size
  yellow open   mockjson QrTKE9uuQFupxXwx4u_jUw   1   1       8000            0      1.7mb          1.7mb
  yellow open   csv      KvOFy4SSTki5qbU_9jkSPA   1   1       8000            0      3.8mb          3.8mb
  ```

  Figura 5.4 Índices de Elasticsearch

  Aquí se pueden observar los índices CSV y JSON indexados correctamente, respectivamente desde Logstash y desde python. Aquí podemos observar diferente información de estos, como el tamaño del índice: ambos muestran 8000 líneas de datos, y el tamaño que ocupa el índice.

Otro modo de comprobar lo mismo es desde la interfaz gráfica del Web Service Dejavu, ya que la de Elasticsearch es bastante mala. Una vez dentro del Servicio, nos ofrece una conexión rápida a los índices, y una representación de calidad. Abrimos los dos índices desde diferentes pestañas para comprobar los datos uno a uno, o el propio tamaño de estos.

Al ver que los índices concuerdan, verificando el proceso. Se usa **Grafana**, donde se puede validar que los campos de ambos índices son exactamente iguales.

Para esto se han creado dos Dashboards, cada uno con un índice como Datasource. Una vez han sido creado, se puede seleccionar el campo para representar, así como otras opciones o filtros. Si se seleccionan los campos uno por uno, se puede comprobar que son iguales en ambos, incluso realizar alguna prueba.

En las Figuras 5.5a, y 5.5b se puede observar los siguientes campos de los dos índices de prueba creados, con el campo timestamp-ts1 como referente, debajo de cada uno encontramos las Figuras 5.6a, y 5.6b. Indicando el Datasource de cada una:

Figura 5.5a Representaciones de diferentes campos y valores del índice JSON de prueba con el campo timestamp-ts1 como referente

Figura 5.5a Datasource del índice JSON de prueba

Figura 5.5b Representaciones de diferentes campos y valores del índice CSV de prueba con el campo timestamp-ts1 como referente

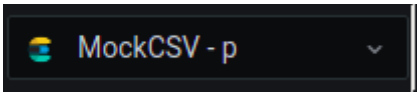

Figura 5.6b Datasource del índice de CSV de prueba

Figura 5.7 Dashboard de prueba para la comprobación de datos iguales en los índices

En la Figura 5.7, se muestra en un mismo Dashboard, diferentes representaciones de campos de diferentes índices. En concreto, el color rojo indica que pertenece al Datasource de JSON y el verde a CSV.

Las diferentes gráficas nos muestran primero el conteo de filas de datos totales, siendo este total 8000, dividido en dos partes de 4000 (una por fichero DUMP). Abajo a la izquierda la suma de valores del campo bytes_dir-ibyt, siendo la primera columna de 38,7 Mil y la segunda de 74,9 Mil. Finalmente, se observa la media del valor fl, llegando los puntos respectivamente a 1,54 y 1,59.

Como los paneles de ambos Datasource son exactamente iguales, se confirma la correcta transformación de los ficheros y la correcta creación de los índices. Se puede realizar con muchos más campos y con otros tipos de visualizaciones como vemos en la Figura 5.8.

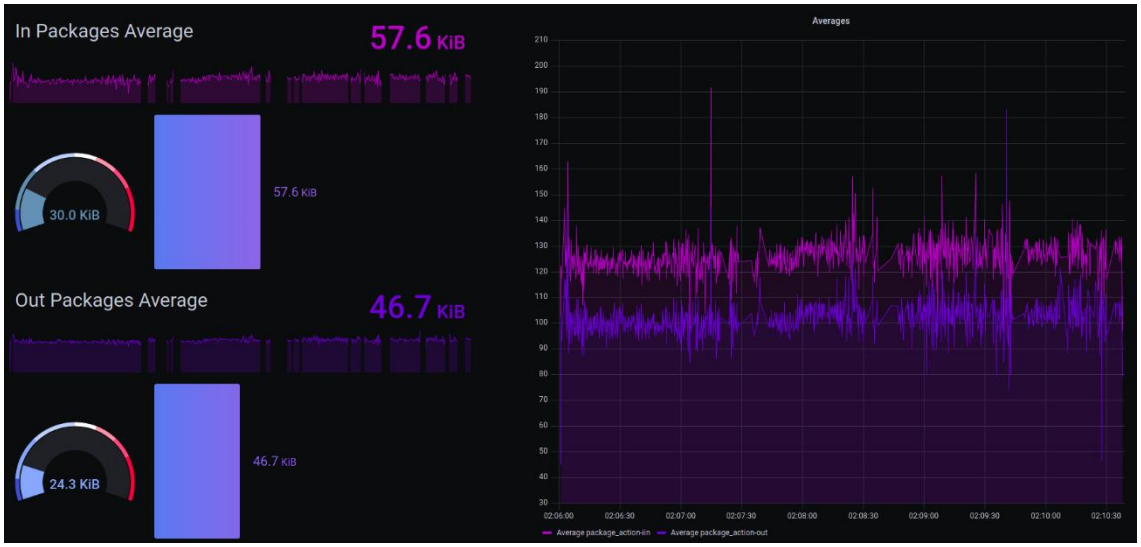

Figura 5.8 Comparación de campos del índice real CSV

Más concretamente, la Figura 5.8 nos muestra una comparación entre los campos In-pakages y Out-pakages. Podemos observar que entran más que los que salen. Primero se ve la media de estos de dos maneras diferentes, y luego una tabla donde se ve la cantidad de cada campo en el tiempo.En ello podemos leer que ambos, los paquetes de entrada y los de salida se comportan igual. Se generar casi el mismo número a la vez, con los mismos picos

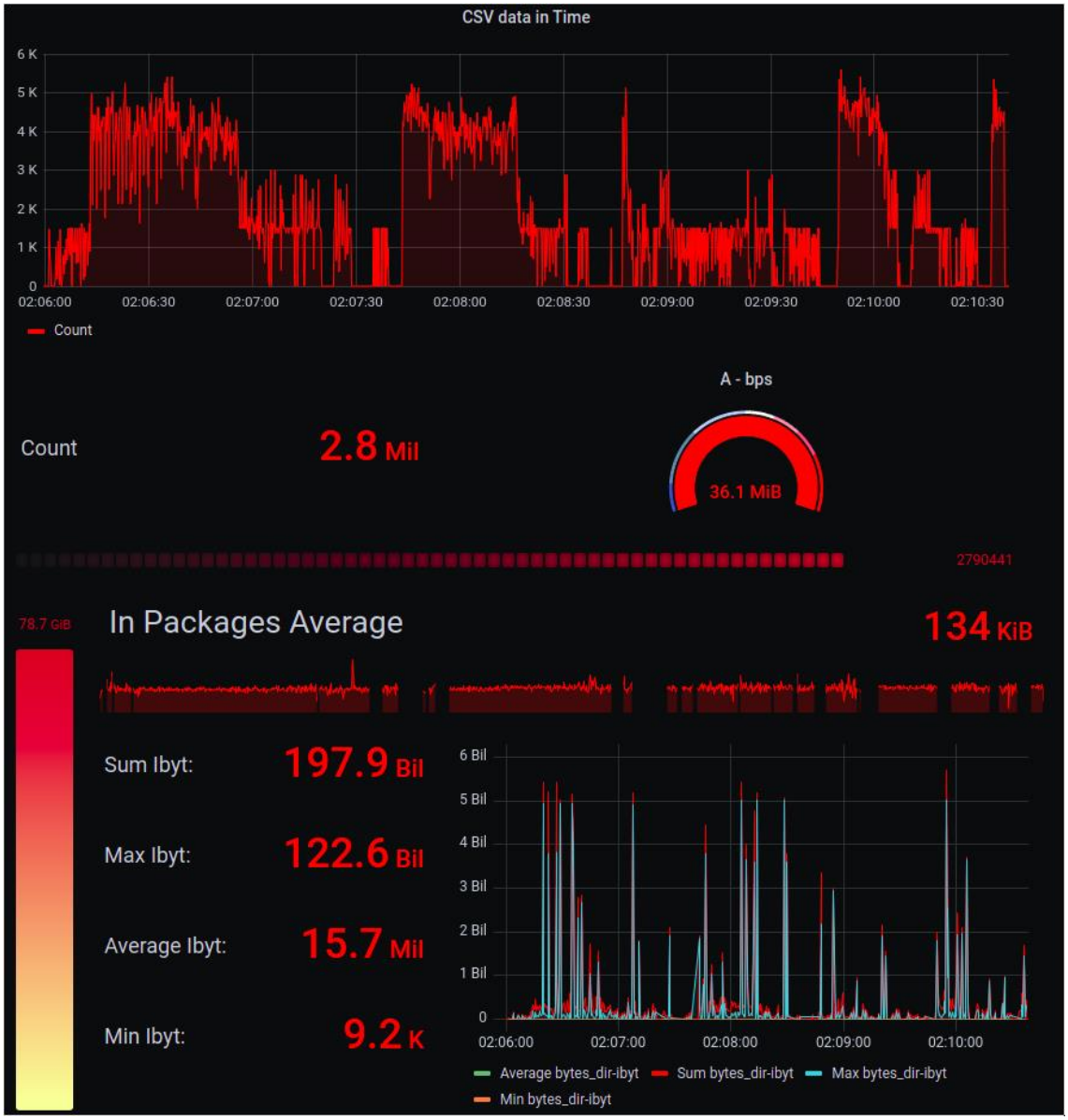

Figura 5.9 Resultados de campos y datos del índice real CSV

En la Figura 5.9 encontrada justo encima, se observan los distintos campos, valores y resultados del índice CSV Real. Este formato de figura es muy importante, va a ser la base, junto con las tablas, para entender el comportamiento de las dos tecnologías mientras indexan, en especial la primera gráfica de la imagen. Por motivos técnicos y de recursos se va a necesitar usar ejemplos más pequeños que este.

Empezando con la gráfica de indexación en el tiempo, continuando con la cuenta total de datos representado de 2 maneras, de con número y con una barra horizontal. Justo a la derecha de la cuenta de datos se encuentra la media de la cuenta total de datos.

Pasada la línea horizontal se encuentra una silueta de la gráfica de la media de los paquetes de entrada y su total numéricamente. Después están los diferentes resultados del campo Ibyt representados también numéricamente, y a su derecha: una gráfica con estos valores mostrados en el tiempo de indexación. A la izquierda una barra vertical mostrando el máximo del campo Ibyt.

5.2 Resultados

Se van a mostrar los diferentes resultados de mediciones obtenidos para poder comparar los procesos. Las líneas de datos totales de los archivos de prueba 0 es de 8000. Los archivos de la prueba 1 son de tamaño 1000000. Los ficheros originales contienen 2790441 líneas de datos.

Mediciones obtenidas de los Programas

	Conversión de Ficheros			**Tamaño Ficheros**		
	Prueba 0	*Prueba 1*	*Real*	*Prueba 0*	*Prueba 1*	*Real*
JSON	0.239s	31.987	71.242s	3,8 MB	470.4 MB	1,3 GB
CSV	0.080s	11.897	23.800s	1,3 MB	158,4 MB	441.9 MB

Tabla 5.1 Medición de tiempo de los conversores de ficheros.

	Conversión de Ficheros			
	JSON		**CSV**	
	Prueba 1	*Real*	*Prueba*	*Real*
CPU	8.8%	25,7%	8.2%	9,6%
RAM	16,3%	42%	11%	28%
Disco	1%	2%	1%	1%

Table 5.2 Medición de los recursos utilizados por los conversores de ficheros

	Indexadores de Datos			**Tamaños índices**		
	Prueba 0	*Prueba 1*	*Real*	*Prueba 0*	*Prueba 1*	*Real*
Python	0,787s	1 min, 44s	-Kill-	1,6 MB	170.9MB	---
Logstash	1,042s	1 min, 47s	4 min, 38s	3.5 MB	470.2MB	1GB

Tabla 5.3 Medición de tiempo de los indexadores de datos

En Grafana, la gráfica indica que los datos generados por python en el tiempo se completan en escasos 9 segundos, mientras que para logstash es 1 minuto 32 segundos, esto se debe a cómo se ha generado el campo @timestamp. Logstash lo genera a la vez que sube una fila de datos. Python genera todos y después indexa las filas.

	Indexación de Datos			
	Python		**Logstash**	
	Prueba 1	*Real*	*Prueba 1*	*Real*
CPU	32,7%	---	100%	100%
RAM	40%	+100%	11%	14%
Disco	1%	1%	1%	3%

Tabla 5.4 Medición de los recursos utilizados por los indexadores de ficheros

En las Tablas de indexación, Tabla 5.3 y Tabla 5.4, se ve que faltan datos. Esto se debe a que no ha sido posible medir el uso de recursos con el fichero original y python, debido a que el ordenador se congela, dejan de funcionar los procesos, incluido el script, haciendo imposible la medición. En los datos que consigue guardar el Script, se ve como sube la RAM rápidamente hasta usarse al completo, y después baja drásticamente a como estaba antes de empezar proceso (esto representa el kill).

Gracias a las Tablas 5.1 y 5.2 entendemos que CSV pesa menos, lo que está directamente relacionado a su uso de recursos, el cual también es menor.

6 Conclusiones y Trabajo Futuro

Este capítulo trata las conclusiones de resumir el trabajo completo. Recordando al lector que se ha realizado y más por encima como, para conseguir una lectura provechosa. Además de los dos procesos estudiados, este capítulo también informará acerca de otros trabajos de investigación posibles de realizar desde este punto del TFG.

6.1 Conclusiones

Con la información medida, y una vez comprobada la veracidad de los datos confirmada por las pruebas con Dashboards (los resultados de las consultas que muestran los gráficos deben ser iguales). Se ha llegado a una conclusión bastante acertada, que es que: ningún proceso es mejor que el otro al cien por cien, cada uno funciona mejor según los recursos disponibles. Vamos a desarrollar esto a continuación.

Las conclusiones y resultados de tiempos obtenidos en este TFG son únicamente replicables o muy similares con el mismo ordenador con el que se ha utilizado para ejecutar los programas y procesos. El tiempo va ligado a la potencia de este, a mayor potencia, menos tardarán los procesos en completarse. Lo mismo pasa con el uso de los recursos.

En la Figura 6.1 se puede observar cómo se han indexado los datos durante el tiempo en Grafana. Esto permite ver el rendimiento de cada tecnología y entender su comportamiento.

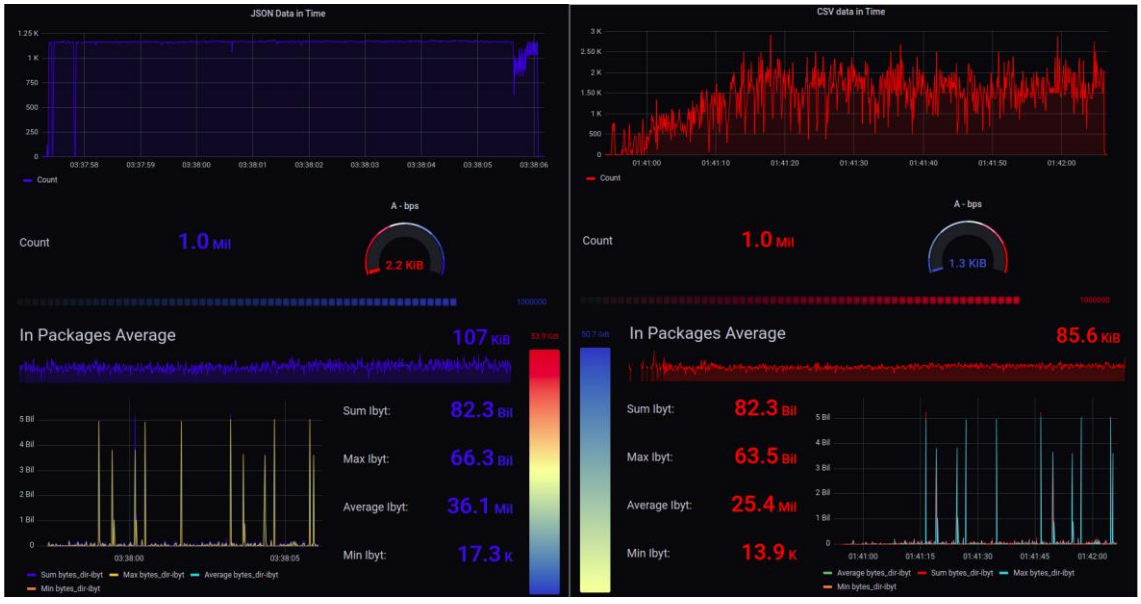

Figura 6.1. Dashboard comparativo de los métodos de indexación a Elasticsearch

Además podemos observar diferentes resultados de los datos, en la mayoría de comparaciones se puede apreciar que el método por Python con JSON (azul), es más estable. También podemos comprobar que ambos ficheros son iguales, con resultados como la suma del campo ibyt.

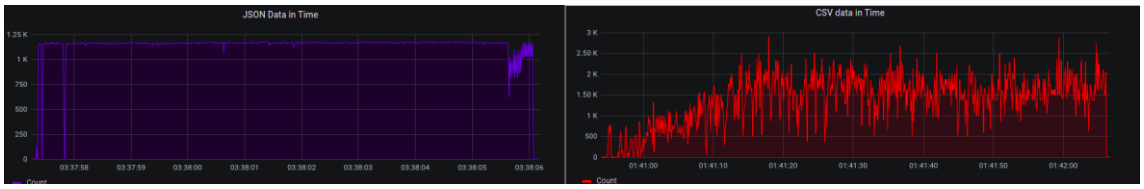

Figura 6.2 Zoom en tiempo de indexación de datos.

En la Figura 6.2 se ven ampliados los gráficos de indexación en el tiempo (Azul - JSON, Rojo - CSV), donde se puede estudiar el supuesto funcionamiento de las tecnologías. Se aprecia la estabilidad de python, y el flujo y variación generado con logstash.

Esto mismo es apreciable también en un proceso más básico con archivos reducidos en la Figura 6.3

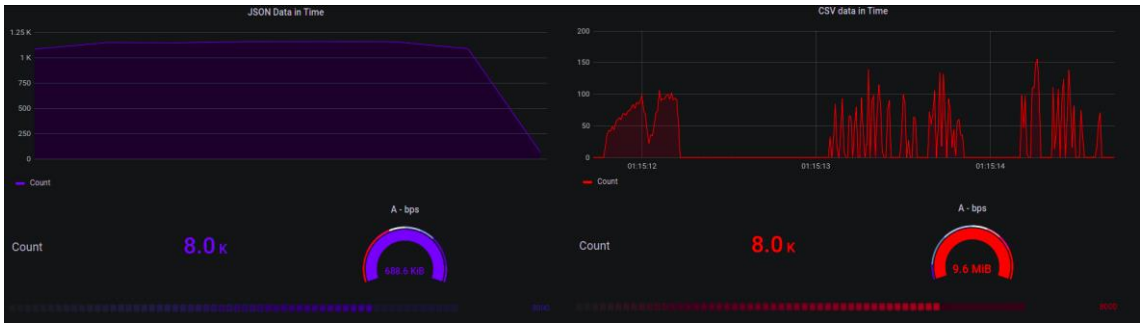

Figura 6.3 Zoom en tiempo indexación de datos de prueba.

Gracias a la Tabla 5.3, se entiende que el proceso de Python es algo más rápido pero consume mucha memoria RAM, poniendo en peligro el proceso, y haciendo que sea poco accesible para un usuario estándar. Una vez se llega al 100% de RAM, se realiza un kill del proceso porque no puede continuar, en este caso, el ordenador utilizado tiene 8GB de RAM. Si se dispone de los recursos necesarios, o una RAM mucho más potente que la CPU podría usarse fiablemente obteniendo resultados más rápidos.

Logstash es algo más lento, utiliza la CPU como recurso principal para indexar los datos. A diferencia de con la RAM, se puede utilizar la CPU al 100%. Por esto se debe que existen picos de subida y de bajada, o donde directamente no hay indexación de datos como se observa en las gráficas de las Figuras 6.2 y 6.3. La CPU se acaba de liberar o saturar.

Por otro lado, el proceso de transformación de datos es casi el triple de grande en tiempo para convertir a JSON. Esto tiene una menor relevancia, porque son tiempos bastante más pequeños, y se sigue usando una enorme cantidad de datos de golpe. Esto se debe a que el archivo JSON ocupa el triple.También mencionar que ocurre justo lo contrario con el tamaño de los índices, los índices generados desde logstash del archivo CSV [25] pesan el triple que los generados desde python.

Ambos procesos se realizan sin controles de errores o prints por pantalla para que sean más fluidos, gasten el menor número de recursos posibles y tarden el menor tiempo posible. La curva de aprendizaje de Logstash resultó muy sencilla, muy amigable para

cualquiera que quiera empezar. Además, es una gran ventaja utilizarlo ya que está especialmente preparado para funcionar con Elasticsearch, a diferencia de Python.

Gracias a Docker, se ha agilizado el proceso enormemente, esta herramienta es algo fundamental hoy en día y debería de utilizarse mucho más. Esta herramienta se retroalimenta con el número de personas que la usan, es decir, cuanta más gente la utilice más potente será, por lo que ha sido un completo acierto trabajar con ella para realizar este TFG.

La Tabla 6.5 resume completamente los dos procesos y sus mejores y peores cualidades.

	Tiempo	**Recursos**	**Dificultad**	**Otros**
Python	Corto	Excesivo uso de RAM (puede generar Kill)	Complejo	Se crearon problemas realizando el bulk de datos por actualizaciones y escasa documentación. Además del kill.
Logstash	Corto	Usa la CPU al máximo, cualquiera puede usarlo	Sencillo	Da sensación de flujo de trabajo y de profesionalidad trabajando directamente con Elasticsearch ya que están hechos aposta para trabajar juntos.

Tabla 6.5 Tabla Resumen de los Procesos, donde podemos compararlos rápidamente y apreciar los puntos fuertes y flojos de cada uno.

Gracias a la Tabla 6.5 se concluye que: Logstash es razonablemente mejor. No tienes problemas de recursos y por lo general parece que el tiempo de generar el índice apenas tardará unos segundos más, tal vez un minuto, pero parece prescindible comparado con el riesgo de no obtener los datos, algo presentado en la opción de Python. Además, logstash está especialmente preparado para trabajar con elasticsearch, lo que índica su gran forma de funcionar, y la curva de aprendizaje es bastante sencilla. Únicamente se recomienda el uso de Python si se tiene una memoria suficientemente grande, y que sea mucho más potente comparada con la CPU.

6.2 Trabajo futuro

Como trabajo Futuro se propone un estudio de mercado de archivos y tecnologías competentes.

Para este estudio, sería necesario realizar las mismas mediciones utilizadas para este trabajo. Consistiría de dos partes.

1. Estudio de conversión de archivos de datos.
   Lo primero sería comparar, como hemos hecho, cuanto tardan diferentes archivos de datos en transformarse desde un .DUMP con Python. Un ejemplo sería coger los dos usados en este trabajo: JSON, CSV, y añadir otros como XLSX, TXT o LOG o JSONL.

Una vez analizados los mejores archivos y descartados los peores, sería conveniente realizar las siguientes pruebas con los que sean validados para el proceso, pero también se podría realizar únicamente con los que mostrasen mejores resultados.

2. El siguiente proceso consistiría en analizar el tiempo y recursos que tarda cada uno en indexar a Elasticsearch, para poder compararlos entre sí como en este trabajo. Pero, además, comprobar también el viceversa, es decir, JSON con logstash y CSV con Python. Añadiendo nuevos métodos: se ha encontrado una manera de indexar archivos JSON y JSONL directamente desde la terminal de Linux, incluso sin uno de elasticsearch, Grafana puede usar JSON como Datasource directamente.

   Finalmente, se podría comparar Grafana con otros servicios como Power BI (que puede indexar directamente un excel), o Kibana. De todos modos, Grafana está especializado en la comparación de uso de recursos y en el tiempo, por lo que nos es tan útil y lo hemos usado.

Si además de tener los datos reales de consumo de recursos y tiempo de cada programa con cada archivo, entendemos en que se especializa cada uno: Elasticsearch es líder en Motor de Búsqueda, como se ha mencionado antes, pero: por ejemplo, también se utiliza para realizar analíticas de log. Es posible que haya una tecnología que lidere esta función específica, o tal vez merezca la pena subir los archivos JSON directamente como Datasource de Grafana para generar los Dashboards.

Así, se consigue que sea posible crear una tabla de ventajas y desventajas de cada una de ellas, incluyendo los pros y los contras. Tal vez los resultados entre dos de 4 comparadas sean similares, llegando a la conclusión de que una es mejor para la tarea de motor de búsqueda con JSON y la otra para Analíticas de log con CSV.

Esta tabla aportaría mucho conocimiento al mundo. Como se ha dicho desde el principio del TFG, la información es poder. Con algo así, la gente podría utilizar eficientemente los servicios proporcionados por las tecnologías estudiadas, sacando el máximo provecho a ellas, y al tiempo de trabajo.

Se estaría sabiendo sacar beneficio y rendimiento a cada situación de forma eficaz y versátil, teniendo el abanico de posibilidades de la tabla, y pudiendo decidir según el caso particular.

BIBLIOGRAFÍA

JSON

[1]

@incollection{friesen2019introducing,

title={Introducing json},

author={Friesen, Jeff},

booktitle={Java XML and JSON},

pages={187--203},

year={2019},

publisher={Springer}

}

CSV

[2]

@article{shafranovich2005common,

title={Common format and MIME type for comma-separated values (CSV) files},

author={Shafranovich, Yakov},

year={2005},

publisher={RFC 4180, October}

}

[3] DocsPython. JSON encoder and decoder. 2021. URL: https://docs.python.org/3/library/json.html (visitado 10-09-2021)

[4] Pandas. Pandas. 2021. URL: https://pandas.pydata.org/ (visitado 10-09-2021).

[5] Datahub. CSV - Comma Separated Values. 2021. URL: https://datahub.io/docs/data-packages/csv (visitado 10-09-2021)

[6] DocsPython. Tiempo de acceso y conversiones. 2021. URL: https://docs.python.org/es/3/library/time.html (visitado 10-09-2021)

[7] Elasticsearch-py documentation. Helpers. 2021. URL: https://elasticsearch-py.readthedocs.io/en/master/helpers.html (visitado 10-09-2021)

ELASTICSEARCH

[8]

@book{gormley2015elasticsearch,

title={Elasticsearch: the definitive guide: a distributed real-time search and analytics engine},

author={Gormley, Clinton and Tong, Zachary},

year={2015},

publisher={" O'Reilly Media, Inc."}

}

e

ELK [9]

@inproceedings{bajer2017building,

title={Building an IoT data hub with Elasticsearch, Logstash and Kibana},

author={Bajer, Marcin},

booktitle={2017 5th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)},

pages={63--68},

year={2017},

organization={IEEE}

}

[10] Elasticsearch Documentation. Logstash. 2021. URL: https://www.elastic.co/es/logstash/ (visitado 10-09-2021)

[11] Grafana. Grafana. 2021. URL: https://grafana.com/grafana/ (visitado 10-09-2021)

GRAFANA [12]

@inproceedings{betke2017real,

title={Real-time I/O-monitoring of HPC applications with SIOX, elasticsearch, Grafana and FUSE},

author={Betke, Eugen and Kunkel, Julian},

booktitle={International Conference on High Performance Computing},

pages={174--186},

year={2017},

organization={Springer}

}

DOCKER [13]

@article{anderson2015docker,

title={Docker [software engineering]},

author={Anderson, Charles},

journal={Ieee Software},

volume={32},

number={3},

pages={102--c3},

year={2015},

publisher={IEEE}

}

[14] Docker-hub. Docker-hub. 2021. URL: https://hub.docker.com/ (visitado 10-09-2021)

Docker Compose [15]

@incollection{jangla2018docker,

title={Docker Compose},

author={Jangla, Kinnary},

```
  booktitle={Accelerating Development Velocity Using Docker},
  pages={77--98},
  year={2018},
  publisher={Springer}
}
```

Indexar Datos [16]

```
@mastersthesis{civantos2019uso,
  title={Uso y ventajas de Elasticsearch en bases de datos no relacionales},
  author={Civantos Martos, Carla and others},
  type={{B.S.} thesis},
  year={2019}
}
```

GIT [17]

@book{chacon2014pro,

title={Pro git},

author={Chacon, Scott and Straub, Ben},

year={2014},

publisher={Springer Nature}

}

PYCHARM [18] -

@book{islam2015mastering,

title={Mastering PyCharm},

author={Islam, Quazi Nafiul},

year={2015},

publisher={Packt Publishing Ltd}

}

[19] Python Documentation. Python. 2021. URL: https://www.python.org/ (visitado 10-09-2021)

[20] Docker Documentation. Docker. 2021. URL: https://www.docker.com/ (visitado 10-09-2021)

[21] Metrics Fire. Grafana with Elastic. 2020. URL: https://www.metricfire.com/blog/using-grafana-with-elasticsearch-tutorial/ (visitado 10-09-2021)

[22] Elasticsearch Documentation. Elasticsearch. 2021. URL: https://www.elastic.co/es/ (visitado 10-09-2021)

[23] Dejavu. The missing UI for Elasticsearch. 2021. URL: https://opensource.appbase.io/dejavu/ (visitado 10-09-2021)

[24] Tecmint. Command lines tools to measure linux performance. 2021. URL: https://www.tecmint.com/command-line-tools-to-monitor-linux-performance/ (visitado 10-09-2021)

FINN - - - - - - - - - - ELIMINAR 25, era el 5 xd

[25] Datahub. CSV - Comma Separated Values. 2021. URL: https://datahub.io/docs/data-packages/csv (visitado 10-09-2021)

ANEXOS

ANEXO A

Archivos de Datos

A.1 Test .DUMP

2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,6,192.168.29.3,201.116.65.2,201.116.168.245,30758,443,0,0,9,0,2855,0,1,7633,3,317

2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,17,192.168.29.3,62.2.21.168,201.116.168.140,12193,53,0,0,1,0,83,0,1,0,0,83

2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,17,192.168.29.3,201.116.168.140,205.251.197.157,18367,53,0,0,1,0,71,0,1,0,0,71

2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,17,172.22.201.34,172.27.15.191,172.22.201.80,52108,53,219,77,1,0,85,0,1,0,0,85

2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,17,192.168.29.3,66.111.49.12,201.116.168.141,53,16361,0,0,1,0,196,0,1,0,0,196

2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,17,192.168.29.3,172.17.211.39,192.58.128.30,56482,53,0,0,1,0,72,0,1,0,0,72

2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,6,192.168.29.3,189.151.159.203,201.116.168.229,51811,80,0,0,8,0,1591,0,1,631,0,198

2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,6,172.16.201.217,172.16.215.232,172.16.215.230,46328,4,657,620,2,0,120,0,2,0,0,60

...

A.2 Test .JSON

{"timestamp-ts1": "2020-09-08T16:15:00-05:00", "timestamp-ts2": "2020-09-08T16:15:00-05:00", "timestamp-ts3": "2020-09-08T16:15:00-05:00", "pr": 6, "ips-ra": "192.168.29.3", "ips-sa": "201.116.65.2", "ips-da": "201.116.168.245", "ports-sp": "30758", "ports-dp": "443", "package_action-iin": 0, "package_action-out": 0,

"package_dir-ipkt": 9, "package_dir-opkt": 0, "bytes_dir-ibyt": 2855, "bytes_dir-obyt": 0, "fl": 1, "rates-bps": 7633, "rates-pps": 3, "rates-bpp": 317}

{"timestamp-ts1": "2020-09-08T16:15:00-05:00", "timestamp-ts2": "2020-09-08T16:15:00-05:00", "timestamp-ts3": "2020-09-08T16:15:00-05:00", "pr": 17, "ips-ra": "192.168.29.3", "ips-sa": "62.2.21.168", "ips-da": "201.116.168.140", "ports-sp": "12193", "ports-dp": "53", "package_action-iin": 0, "package_action-out": 0, "package_dir-ipkt": 1, "package_dir-opkt": 0, "bytes_dir-ibyt": 83, "bytes_dir-obyt": 0, "fl": 1, "rates-bps": 0, "rates-pps": 0, "rates-bpp": 83}

{"timestamp-ts1": "2020-09-08T16:15:00-05:00", "timestamp-ts2": "2020-09-08T16:15:00-05:00", "timestamp-ts3": "2020-09-08T16:15:00-05:00", "pr": 17, "ips-ra": "192.168.29.3", "ips-sa": "201.116.168.140", "ips-da": "205.251.197.157", "ports-sp": "18367", "ports-dp": "53", "package_action-iin": 0, "package_action-out": 0, "package_dir-ipkt": 1, "package_dir-opkt": 0, "bytes_dir-ibyt": 71, "bytes_dir-obyt": 0, "fl": 1, "rates-bps": 0, "rates-pps": 0, "rates-bpp": 71}

{"timestamp-ts1": "2020-09-08T16:15:00-05:00", "timestamp-ts2": "2020-09-08T16:15:00-05:00", "timestamp-ts3": "2020-09-08T16:15:00-05:00", "pr": 17, "ips-ra": "172.22.201.34", "ips-sa": "172.27.15.191", "ips-da": "172.22.201.80", "ports-sp": "52108", "ports-dp": "53", "package_action-iin": 219, "package_action-out": 77, "package_dir-ipkt": 1, "package_dir-opkt": 0, "bytes_dir-ibyt": 85, "bytes_dir-obyt": 0, "fl": 1, "rates-bps": 0, "rates-pps": 0, "rates-bpp": 85}

{"timestamp-ts1": "2020-09-08T16:15:00-05:00", "timestamp-ts2": "2020-09-08T16:15:00-05:00", "timestamp-ts3": "2020-09-08T16:15:00-05:00", "pr": 17, "ips-ra": "192.168.29.3", "ips-sa": "66.111.49.12", "ips-da": "201.116.168.141", "ports-sp": "53", "ports-dp": "16361", "package_action-iin": 0, "package_action-out": 0, "package_dir-ipkt": 1, "package_dir-opkt": 0, "bytes_dir-ibyt": 196, "bytes_dir-obyt": 0, "fl": 1, "rates-bps": 0, "rates-pps": 0, "rates-bpp": 196}

{"timestamp-ts1": "2020-09-08T16:15:00-05:00", "timestamp-ts2": "2020-09-08T16:15:00-05:00", "timestamp-ts3": "2020-09-08T16:15:00-05:00", "pr": 17, "ips-ra": "192.168.29.3", "ips-sa": "172.17.211.39", "ips-da": "192.58.128.30", "ports-sp": "56482", "ports-dp": "53", "package_action-iin": 0, "package_action-out": 0, "package_dir-ipkt": 1, "package_dir-opkt": 0, "bytes_dir-ibyt": 72, "bytes_dir-obyt": 0, "fl": 1, "rates-bps": 0, "rates-pps": 0, "rates-bpp": 72}

{"timestamp-ts1": "2020-09-08T16:15:00-05:00", "timestamp-ts2": "2020-09-08T16:15:00-05:00", "timestamp-ts3": "2020-09-08T16:15:00-05:00", "pr": 6, "ips-ra": "192.168.29.3", "ips-sa": "189.151.159.203", "ips-da": "201.116.168.229", "ports-sp": "51811", "ports-dp": "80", "package_action-iin": 0, "package_action-out": 0, "package_dir-ipkt": 8, "package_dir-opkt": 0, "bytes_dir-ibyt": 1591, "bytes_dir-obyt": 0, "fl": 1, "rates-bps": 631, "rates-pps": 0, "rates-bpp": 198}

{"timestamp-ts1": "2020-09-08T16:15:00-05:00", "timestamp-ts2": "2020-09-08T16:15:00-05:00", "timestamp-ts3": "2020-09-08T16:15:00-05:00", "pr": 6, "ips-ra": "172.16.201.217", "ips-sa": "172.16.215.232", "ips-da": "172.16.215.230", "ports-sp": "46328", "ports-dp": "4", "package_action-iin": 657, "package_action-out": 620,

"package_dir-ipkt": 2, "package_dir-opkt": 0, "bytes_dir-ibyt": 120, "bytes_dir-obyt": 0, "fl": 2, "rates-bps": 0, "rates-pps": 0, "rates-bpp": 60}

A.3 Test .CSV

timestamp-ts1,timestamp-ts2,timestamp-ts3,pr,ips-ra,ips-sa,ips-da,ports-sp,ports-dp,package_action-iin,package_action-out,package_dir-ipkt,package_dir-opkt,bytes_dir-ibyt,bytes_dir-obyt,fl,rates-bps,rates-pps,rates-bpp

2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,6,192.168.29.3,201.116.65.2,201.116.168.245,30758,443,0,0,9,0,2855,0,1,7633,3,317

2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,17,192.168.29.3,62.2.21.168,201.116.168.140,12193,53,0,0,1,0,83,0,1,0,0,83

2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,17,192.168.29.3,201.116.168.140,205.251.197.157,18367,53,0,0,1,0,71,0,1,0,0,71

2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,17,172.22.201.34,172.27.15.191,172.22.201.80,52108,53,219,77,1,0,85,0,1,0,0,85

2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,17,192.168.29.3,66.111.49.12,201.116.168.141,53,16361,0,0,1,0,196,0,1,0,0,196

2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,17,192.168.29.3,172.17.211.39,192.58.128.30,56482,53,0,0,1,0,72,0,1,0,0,72

2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,2020-09-08T16:15:00-05:00,6,192.168.29.3,189.151.159.203,201.116.168.229,51811,80,0,0,8,0,1591,0,1,631,0,198

…

...

ANEXO B

Conversores de Archivos

B.1 JSON.py

```
# Transform .dump DATA into .JSON
"""
@author: Hoyo Bravo, Guillermo
"""
import json
import time

LOG_FILENAME = 'LoggingDebug'

def main():
    # Start timer
    inicio = (time.time())

    # Counters for tests
    counter = 0
    counter2 = 0

    # List of Dictionaries (Join DUMP Files)
    LDictionary = []

    # Open Dump Files
```

```
f = open('nfcapd.202009081610.dump', 'r')
fi = open('nfcapd.202009081615.dump', 'r')

# Test 1 Open Million Dump Files
#f = open('UnMillon.dump', 'r')
#fi = open('unDosMillones.dump', 'r')

# MOCK Open Dump Test Files
#f = open('testA.dump', 'r')
#fi = open('testB.dump', 'r')

# Splitting the first file in lines for the JSON dictionary
for linea in f:
    campos = linea.split(',')

    # TEST dictionary fields length
    if len(campos) != 19:
        print('ERROR splitting file information')
        return -1

    # Stablishing a Dictionary Structure and inserting the first line of the file on it
    Dictionary = {
        'timestamp-ts1': campos[0],
        'timestamp-ts2': campos[1],
        'timestamp-ts3': campos[2],
        'pr': int(campos[3]),
        'ips-ra': campos[4],
```

```
        'ips-sa': campos[5],
        'ips-da': campos[6],
        'ports-sp': campos[7],
        'ports-dp': campos[8],
        'package_action-iin': int(campos[9]),
        'package_action-out': int(campos[10]),
        'package_dir-ipkt': int(campos[11]),
        'package_dir-opkt': int(campos[12]),
        'bytes_dir-ibyt': int(campos[13]),
        'bytes_dir-obyt': int(campos[14]),
        'fl': int(campos[15]),
        'rates-bps': int(campos[16]),
        'rates-pps': int(campos[17]),
        'rates-bpp': int(campos[18])
    }

    # Adding Dictionaries created to the DictionaryList
    LDictionary.append(Dictionary)
    # Counter of List Elements
    counter += 1

# TEST of the List Length = First File Length
if len(LDictionary) != counter:
    print('ERROR of the list Length')
    return -2

# Open Second File to Transform
```

```
for linea in fi:
    componentes = linea.split(',')

    # TEST dictionary fields length
    if len(componentes) != 19:
        print('ERROR splitting file information')
        return -3

    # Stablishing a Dictionary Structure and inserting the first line of the file on it
    Dictionary2 = {
        'timestamp-ts1': componentes[0],
        'timestamp-ts2': componentes[1],
        'timestamp-ts3': componentes[2],
        'pr': int(componentes[3]),
        'ips-ra': componentes[4],
        'ips-sa': componentes[5],
        'ips-da': componentes[6],
        'ports-sp': componentes[7],
        'ports-dp': componentes[8],
        'package_action-iin': int(componentes[9]),
        'package_action-out': int(componentes[10]),
        'package_dir-ipkt': int(componentes[11]),
        'package_dir-opkt': int(componentes[12]),
        'bytes_dir-ibyt': int(componentes[13]),
        'bytes_dir-obyt': int(componentes[14]),
        'fl': int(componentes[15]),
        'rates-bps': int(componentes[16]),
```

```
        'rates-pps': int(componentes[17]),
        'rates-bpp': int(componentes[18])
    }

    # Adding 2nd Dictionaries created to the DictionaryList
    LDictionary.append(Dictionary2)
    # Counter of List Elements
    counter2 += 1

# TEST of the list length = Both Files length
if len(LDictionary) != counter2 + counter:
    print('ERROR of the list Length')
    return -4

# TEST Both Files Parsers have same files length
print(" TEST: File Parsers Length::::")
print(len(LDictionary))

# TEST Last Element of the File
print(LDictionary[counter+counter2-1])

# Parse List to JSON File
with open('JSON.json', 'w') as fil:
    for element in LDictionary:
        json.dump(element, fil)
        fil.write("\n")
```

```
    #print('The files have been Exported)
'''

# MOCK JSON File
with open('MockJSON.json', 'w') as fil:
    for element in LDictionary:
        json.dump(element, fil)
        fil.write("\n")

    #print('The files have been Exported')

# MILLON File
with open('Millon.json', 'w') as fil:
    for element in LDictionary:
        json.dump(element, fil)
        fil.write("\n")

    #print('The files have been Exported)
'''
f.close()
fi.close()

fin = (time.time())

print("Tiempo Transformación de DUMP a JSON: ")
print(fin-inicio)
```

```
    return


if __name__ == '__main__':
    main()
```

B.2 CSV.py

```
# Transform .dump DATA into .JSON
"""
@author: Hoyo Bravo, Guillermo
"""
import time


LOG_FILENAME = 'LoggingDebug'


def main():
    # Start timer
    inicio = (time.time())


    # Counters for tests
    counter = 0
    counter2 = 0


    # List of Dictionaries (Join DUMP Files)
    LDictionary = []


    # Open Dump Files
```

```
f = open('nfcapd.202009081610.dump', 'r')
fi = open('nfcapd.202009081615.dump', 'r')

# Test 1 Open Million Dump Files
#f = open('UnMillon.dump', 'r')
#fi = open('unDosMillones.dump', 'r')

# MOCK Open Dump Test Files
#f = open('testA.dump', 'r')
#fi = open('testB.dump', 'r')

# Splitting the first file in lines for the JSON dictionary
for linea in f:
    campos = linea.split(',')

    # TEST dictionary fields length
    if len(campos) != 19:
        print('ERROR splitting file information')
        return -1

    # Stablishing a Dictionary Structure and inserting the first line of the file on it
    Dictionary = [
        campos[0],
        campos[1],
        campos[2],
        int(campos[3]),
        campos[4],
```

```
            campos[5],
            campos[6],
            campos[7],
            campos[8],
            int(campos[9]),
            int(campos[10]),
            int(campos[11]),
            int(campos[12]),
            int(campos[13]),
            int(campos[14]),
            int(campos[15]),
            int(campos[16]),
            int(campos[17]),
            int(campos[18])
        ]

        #Adding each Dictionary to the List
        LDictionary.append(Dictionary)
        counter += 1

    # TEST of the List Length = First File Length
    if len(LDictionary) != counter:
        print('ERROR of the list Length')
        return -2

    # Open Second File to Transform
    for linea in fi:
```

```
componentes = linea.split(',')

# TEST dictionary fields length
if len(componentes) != 19:
    print('ERROR splitting file information')
    return -3

# Stablishing a Dictionary Structure and inserting the first line of the file on it
Dictionary2 = [
    componentes[0],
    componentes[1],
    componentes[2],
    int(componentes[3]),
    componentes[4],
    componentes[5],
    componentes[6],
    componentes[7],
    componentes[8],
    int(componentes[9]),
    int(componentes[10]),
    int(componentes[11]),
    int(componentes[12]),
    int(componentes[13]),
    int(componentes[14]),
    int(componentes[15]),
    int(componentes[16]),
    int(componentes[17]),
```

```
            int(componentes[18])
        ]

        # Adding each Dictionary to the List
        LDictionary.append(Dictionary2)
        # Counter of List Lines
        counter2 += 1


    # TEST of the list length = Both Files length
    if len(LDictionary) != counter2 + counter:
        print('ERROR of the list Length')
        return -4

    # TEST Both Files Parsers have same files length
    print(" TEST: File Parsers Length::::")
    print(len(LDictionary))

    # TEST Last Element of the File
    print(LDictionary[counter + counter2 - 1])

    # Parse List to CSV File
    # REAL
    with open('CSV.csv', 'w') as fil:
        fil.write("timestamp-ts1,timestamp-ts2,timestamp-ts3,pr,ips-ra,ips-sa,ips-da,ports-
sp,ports-dp,package_action-iin,package_action-out,package_dir-ipkt,package_dir-
opkt,bytes_dir-ibyt,bytes_dir-obyt,fl,rates-bps,rates-pps,rates-bpp\n")
        for element in LDictionary:
```

```
        n = 0
        for x in element:
            fil.write(str(x))
            if n < 18:
                fil.write(",")
                n = n+1
                # print(n)
        fil.write("\n")

    #print('The files have been Exportted')

''' # MOCK
with open('MockCSV.csv', 'w') as fil:
    fil.write("timestamp-ts1,timestamp-ts2,timestamp-ts3,pr,ips-ra,ips-sa,ips-da,ports-sp,ports-dp,package_action-iin,package_action-out,package_dir-ipkt,package_dir-opkt,bytes_dir-ibyt,bytes_dir-obyt,fl,rates-bps,rates-pps,rates-bpp\n")
    for element in LDictionary:
        n = 0
        for x in element:
            fil.write(str(x))
            if n < 18:
                fil.write(",")
                n = n+1
                # print(n)
        fil.write("\n")

    #print('The files have been Exportted')
```

```
    # MILLON
    with open('Millon.csv', 'w') as fil:
        fil.write("timestamp-ts1,timestamp-ts2,timestamp-ts3,pr,ips-ra,ips-sa,ips-da,ports-sp,ports-dp,package_action-iin,package_action-out,package_dir-ipkt,package_dir-opkt,bytes_dir-ibyt,bytes_dir-obyt,fl,rates-bps,rates-pps,rates-bpp\n")
        for element in LDictionary:
            n = 0
            for x in element:
                fil.write(str(x))
                if n < 18:
                    fil.write(",")
                    n = n+1
                    # print(n)
            fil.write("\n")

        #print('The files have been Exportted')
    '''
    f.close()
    fi.close()
    fin = (time.time())
    print("Tiempo de Dump a CSV")
    print(fin-inicio)
    return

if __name__ == '__main__':
    main()
```

ANEXO C

Indexadores de Datos

C.1 Elastic.py

```
#Importamos la librería JSON
import json
#Importamos Elasticsearch para poder realizar el bulk de datos
from elasticsearch import Elasticsearch, helpers
#Importamos el Datetime
from datetime import datetime
#Importamos Time para medir el tiempo que tarda en subirse
import time

# Start Timer
inicio = (time.time())

# Declare Elasticsearch Instace for Conexion
cliente = Elasticsearch("localhost:9200")

# Build JSON File as LIST
def get_data_de_fichero(self):
    # Devuelve lista de documentos
    return [l.strip() for l in open(str(self), encoding="utf8", errors='ignore')]

# Get Data
```

```
#docs = get_data_de_fichero("JSON.json")
#docs = get_data_de_fichero("MockJSON.json")
docs = get_data_de_fichero("Millon.json")

# TEST Print docs Length
print ("Longitud de documento:", len(docs))

# Define empty list for the Elasticsearch docs
lista_docs = []

# Iterate over each string of the list docs
for num, doc in enumerate(docs):

    # catch any JSON loads() errors
    try:
        # Tranform the JSON list to a Dictionary
        dict_doc = json.loads(doc)
        # Add @Timestamp field for future Grafana Dashboards
        dict_doc["@timestamp"] = datetime.now()

        # append the dict object to the list []
        lista_docs += [dict_doc]

    # print the errors
    except Exception as err:
        print("Error in line %d" % num)
```

```
# TEST List Length

print("Longitud lista de documentos:", len(lista_docs))


try:

    print ("\nIndexación de lista documentos")


    # Use Library Elasticsearch Helpers to Bulk on a new index the list with all the information

    #respuesta = helpers.bulk(cliente, lista_docs, index = "realjson", doc_type = "JSON.json")

    #respuesta = helpers.bulk(cliente, lista_docs, index = "mockjson", doc_type = "mockJSON.json")

    respuesta = helpers.bulk(cliente, lista_docs, index = "millonjson2", doc_type = "_doc")


    # print the response returned by Elasticsearch

    # print ("Respuesta helpers.bulk():", respuesta)

    # print ("Respuesta helpers.bulk():", json.dumps(respuesta, indent=4))


    fin = (time.time())

    print("Tiempo para subir de Json a Elastic")

    print(fin-inicio)
```

C.2 Logstash.conf

```
input{

    file{

        path => "/home/lilg8b/Downloads/TFGnew/TFG/FicherosDump/TestLogstashCSV.csv"

        start_position => "beginning"
```

```
        sincedb_path => "/dev/null"
    }
}
filter{
    csv{
        separator => ","
        columns => ["timestamp-ts1", "timestamp-ts2", "timestamp-ts3", "pr", "ips-ra", "ips-sa", "ips-da", "ports-sp", "ports-dp",
        "package_action-iin", "package_action-out", "package_dir-ipkt", "package_dir-opkt", "bytes_dir-ibyt", "bytes_dir-obyt", "fl", "rates-bps", "rates-pps", "rates-bpp" ]
    }
    mutate {convert => ["pr", "integer"]}
    mutate {convert => ["ports-sp", "integer"]}
    mutate {convert => ["ports-dp", "integer"]}
    mutate {convert => ["package_action-iin", "integer"]}
    mutate {convert => ["package_action-out", "integer"]}
    mutate {convert => ["package_dir-ipkt", "integer"]}
    mutate {convert => ["package_dir-opkt", "integer"]}
    mutate {convert => ["bytes_dir-ibyt", "integer"]}
    mutate {convert => ["bytes_dir-obyt", "integer"]}
    mutate {convert => ["fl", "integer"]}
    mutate {convert => ["rates-bps", "integer"]}
    mutate {convert => ["rates-pps", "integer"]}
    mutate {convert => ["rates-bpp", "integer"]}

}
output{
    elasticsearch{
```

```
        hosts => ["localhost:9200"]
        index => "tfg-csv-logstash2"
        document_type => "cvs_tfg_data"
    }
}
```

ANEXO D

Archivos del Entorno.

D.1 docker-compose.yml

```
version: '3'


services:
        grafana:
          image: grafana/grafana
          ports:
            - 3000:3000
          links:
            - elasticsearch
        elasticsearch:
          image: docker.elastic.co/elasticsearch/elasticsearch-oss:7.0.1
          container_name: elasticsearch
          environment:
            - discovery.type=single-node
            - http.port=9200
            - http.cors.enabled=true
            - http.cors.allow-origin=http://localhost:1358,http://127.0.0.1:1358
            - http.cors.allow-headers=X-Requested-With,X-Auth-Token,Content-Type,Content-Length,Authorization
            - http.cors.allow-credentials=true
            - bootstrap.memory_lock=true
```

```
    - 'ES_JAVA_OPTS=-Xms512m -Xmx512m'
  ports:
    - '9200:9200'
    - '9300:9300'
 dejavu:
  image: appbaseio/dejavu:3.3.0
  container_name: dejavu
  ports:
    - '1358:1358'
  links:
    - elasticsearch
```

D2. script.sh

```
#!/bin/sh
while true
do
        echo CPU: `echo CPU: `top -b -n1 | grep "Cpu(s)" | awk '{print $2 + $4}'`
        FREE_DATA=`free -m | grep Mem`
        CURRENT=`echo $FREE_DATA | cut -f3 -d' '`
        TOTAL=`echo $FREE_DATA | cut -f2 -d' '`
        echo RAM: $(echo "scale = 2; $CURRENT/$TOTAL*100" | bc)
        echo HDD: `df -lh | awk '{if ($6 == "/") { print $5 }}' | head -1 | cut -d'%' -f1`

        sleep 1
        clear
```

done

ANEXO E

Manual del programador

Logstash

$ time sudo ./logstash -f logstash.conf

Elasticsearch

Ver Índices Creados

$ curl -XGET 'localhost:9200/_cat/indices?v'

Eliminar Índices

$ curl -X DELETE 'http://localhost:9200/<IndexName>'

Docker

Montar Servicios

$ docker-compose up -d

Ver Imagenes Funcionando

$ docker ps

Ver Todas las Imágenes

$ docker ps -a

Parar todos los containers

$ docker-compose down

Script Medir Recursos

Ejecutar Script Guardando los datos

$ ./script.sh > file.log

Web Services

Comandos de Control

$ sudo systemctl status/start/stop/restart <webservice>