Proyecto de curso - 2ª parte

Kevin Santiago Artunduaga Vivas, Juan Camilo Vargas Velez, Sebastian Diaz Noguera, Angie Patricia Manzano Melendez, Samuel Pinzón Valderrutén

Universidad Autónoma de Occidente

Cali, Colombia.

Selección del dataset objetivo

En primer lugar, para la aplicación se necesitó una base de datos que incluyera información sobre aeropuertos, aerolíneas, vuelos, horarios, retrasos, cancelaciones, entre otros aspectos relevantes. Para ello se hizo uso de kaggle.com, una plataforma web que contiene herramientas y recursos para la ciencia de datos.

El dataset objetivo, llamado "Flight Status Prediction" [1] proporciona información histórica sobre vuelos de Estados Unidos desde enero de 2018 hasta el año 2022, incluyendo detalles sobre los vuelos cancelados y retrasados por las aerolíneas, con el fin de predecir qué vuelos serán cancelados o retrasados y el tiempo de retraso. Este dataset cuenta con el número de 61 columnas en total por cada csv del año, y un peso total de 25.78 GB, como es una gran cantidad de datos se decidió enfocarse en el año 2021, para hacer un filtro de estos y poder analizar cómo la situación del COVID-19 afecto estas industrias de viajes, por lo que habían bastantes personas tratando tener un retorno a su país y no aislarse en el lugar donde se encontraban.

En él se proporciona información detallada sobre la salida y llegada de un vuelo, incluyendo información sobre el aeropuerto de origen y destino, hora prevista de salida y llegada, hora real de salida y llegada, retrasos en la salida y llegada, número de cola del avión, información sobre la aerolínea y la ruta del vuelo, así como información sobre el estado del vuelo, como si se ha cancelado o desviado.

Definición de la arquitectura completa del sistema

La arquitectura completa del sistema inicia con un balanceador de carga para de esta manera garantizar la disponibilidad de la aplicación y facilitar el número de solicitudes realizadas. Para ello se hace la réplica de la aplicación, esto implica la creación de una copia idéntica de la misma, lo que permite distribuir la carga de manera equilibrada entre ellas. En caso de que uno de los servidores que aloja la aplicación principal experimente algún problema o falle, el balanceador de cargas redirigirá automáticamente las solicitudes al servidor de respaldo, asegurando la continuidad del servicio sin interrupciones.

Para la aplicación se hizo uso de vue, un framework de javascript encargado específicamente para el diseño de páginas. En donde se realizaron las interfaces del inicio de sesión, la página de los aeropuertos y la de las aerolíneas, teniendo claro que estos hacen parte de los usuarios de BBS71.

Como se utilizaron 3 microservicios, es necesario correr cada uno por un puerto diferente, pero mediante una Api Gateway se gestionan y exponen múltiples servicios a través de un puerto por el que salen todos los microservicios. Haciendo uso de ella como un punto de entrada centralizado para todas las solicitudes de la aplicación.

El backend de la aplicación se basa en microservicios que accederán a través de API REST. Como se mencionó los microservicios son: aerolíneas, aeropuertos y usuarios. Mediante ella se envían peticiones y recibe respuestas que contienen datos en función de la acción solicitada gracias a las solicitudes HTTP y respuestas, por

medio de operaciones comunes como GET, POST, PUT y DELETE.

El microservicio de aerolíneas se encarga de gestionar la información relacionada con las aerolíneas con los datos sobre los vuelos que se han efectuado, junto con el aeropuerto de origen y destino, el horario de salida y de llegada, y el estado del vuelo. Las aeronaves que disponen y el número de vuelos que han hecho, las rutas realizadas y por último las estadísticas que tienen; número total de vuelos, número de aeronaves y el número de vuelos cancelados.

El microservicio de aeropuertos se encarga de gestionar la información relacionada con los aeropuertos, en particular, los vuelos de salida y llegada, detallando la fecha de el, la hora programada, la ciudad de origen o destino, el nombre de la aerolínea, el número de vuelo y el estado de este. Por último, proporciona estadísticas sobre el número de vuelos de salida y llegada, así como la lista de aerolíneas que realizan operaciones en cada aeropuerto.

El microservicio de usuarios se encarga de gestionar la información relacionada con los usuarios que utilizan la aplicación. Este microservicio realiza las peticiones a la base de datos de acuerdo a las credenciales que el usuario introduzca, si estas coinciden dejará ingresar al usuario.

Se hizo uso de una base de datos no relacional mediante mongoDB, debido a su estructura flexible, escalabilidad horizontal, rendimiento y velocidad, soporte para datos de gran tamaño, consultas flexibles y potentes, así como su implementación con tecnologías modernas.

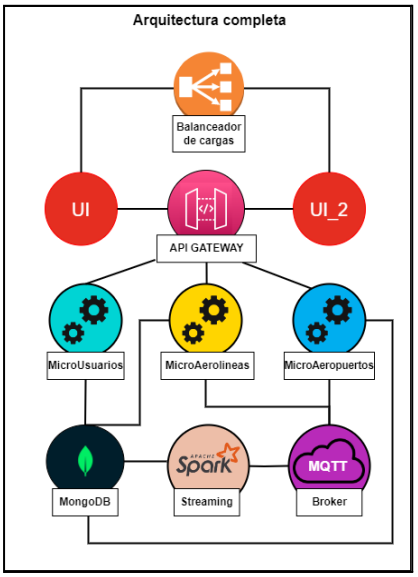
Para asegurar una comunicación entre el backend y el análisis distribuido, es necesario utilizar un intermediario o broker de mensajería. Este componente se encarga de gestionar la entrega de mensajes y garantizar que sean enviados y recibidos de manera efectiva.

Para realizar el análisis distribuido a partir del dataset obtenido, se hizo la implementación de Apache Spark ya que esta dispone de la facilidad de procesar datos en paralelo y está hecha para leer grandes volúmenes de datos. Fue utilizado para realizar las estadísticas en los microservicios de aerolíneas y aeropuertos.

La arquitectura propuesta permitirá una alta escalabilidad y flexibilidad, ya que cada microservicio puede ser desarrollado, implementado y escalado independientemente de los demás. Además, la API REST permitirá una fácil integración con otras aplicaciones y sistemas externos y por el lado del análisis distribuido, se tiene en cuenta que información es la que se quiere usar para procesar los datos de manera simultánea y paralela.

Ahora bien, para generar un avance en lo que se tiene planeado, se necesita el uso de una tecnología para empaquetar la aplicación en contenedores y otra para el despliegue en un cluster de procesamiento de datos distribuido sobre el dataset seleccionado.

Fig. 3: “Arquitectura completa”



Generación y selección de alternativas de solución:

Al evaluar diferentes alternativas de solución para el empaquetado de la aplicación en contenedores se consideraron tres tecnologías Kubernetes, Apache Mesos y Docker. Y para el despliegue en un cluster de procesamiento de datos distribuido se

evaluaron Apache Hadoop, Apache Flink y Apache Spark.

Alternativas del empaquetado en contenedores

Kubernetes es una herramienta de orquestación de contenedores ampliamente utilizada que ofrece una gran cantidad de características y funcionalidades avanzadas, como el escalado automático y la programación de trabajos. “Con Kubernetes, puede ejecutar cualquier tipo de aplicación en contenedor mediante el uso del mismo conjunto de herramientas para entornos locales y en la nube” [2]. Sin embargo, puede ser complejo el aprendizaje de esta aplicación, la configuración y administración pueden requerir más experiencia técnica que las otras opciones.

Apache Mesos, por su parte, es una herramienta de gestión de clústeres que puede ejecutar aplicaciones en contenedores. Ofrece una gran flexibilidad en la asignación de recursos y la programación de trabajos, lo que lo hace ideal para aplicaciones distribuidas y escalables. Este dispone de un “soporte para ejecutar aplicaciones heredadas y nativas de la nube en el mismo clúster [3]”. Sin embargo, al igual que Kubernetes puede ser más complejo de configurar y administrar que otras soluciones de contenedores.

Docker, en cambio, es una plataforma de contenedores de código que “crea herramientas simples y un enfoque de empaquetado universal que agrupa todas las dependencias de la aplicación dentro de un contenedor que luego se ejecuta en Docker Engine”[4]. También cuenta con una amplia gama de herramientas y complementos disponibles; entre ellas está Docker Compose, Docker Swarm, Docker Hub y Docker Machine. Docker también es compatible con una gran variedad de sistemas operativos y lenguajes de programación, lo que lo vuelve una herramienta multifuncional.

Alternativas del despliegue en un cluster de procesamiento de datos distribuido

Apache Spark es un sistema de procesamiento de datos de código abierto que ofrece un alto rendimiento y capacidades de procesamiento distribuido, en otras palabras un cluster de procesamiento de datos distribuido. “Está diseñado para ofrecer la velocidad computacional, la escalabilidad y la capacidad de programación necesarias para Big Data, específicamente para la transmisión de datos, datos de gráficos, machine learning y aplicaciones de inteligencia artificial ”[5]. Además, Spark ofrece una variedad de bibliotecas y herramientas para el procesamiento y análisis de datos, como Spark SQL, Spark Streaming, etc.

Apache Hadoop es un ecosistema de software de código abierto que proporciona herramientas y frameworks para el almacenamiento y procesamiento distribuido de grandes conjuntos de datos. “Hadoop también incluye un sistema de almacenamiento distribuido, el Hadoop Distributed File System (HDFS), que almacena datos en discos locales de su clúster en bloques de gran tamaño”[6]. Además, Hadoop ofrece un modelo de programación llamado MapReduce para el procesamiento paralelo de datos en lotes. Hadoop también cuenta con otras herramientas como Hadoop YARN para la gestión de recursos y Apache Hive para consultas SQL en grandes conjuntos de datos.

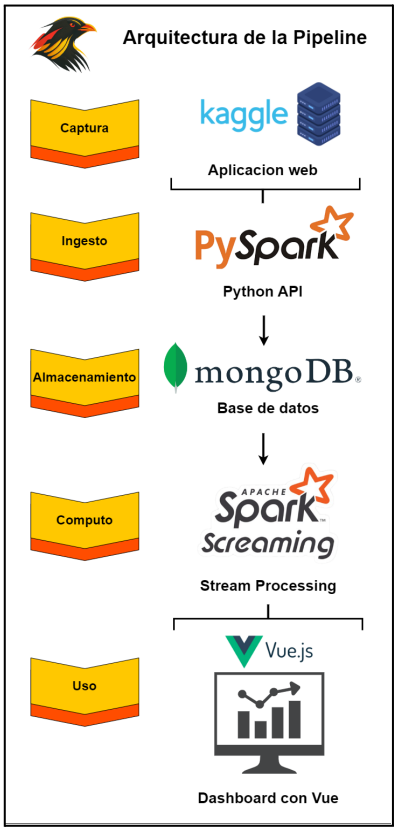
Apache Flink “es un framework de código abierto orientado al procesamiento de flujos de datos en streaming de forma distribuida y con alta disponibilidad”[7]. Se destaca por su capacidad para realizar análisis en tiempo real.Flink proporciona un modelo de programación llamado DataStream API para el procesamiento de datos de flujo continuo y un modelo de programación llamado DataSet API para el procesamiento de datos en lotes. Flink se basa en el concepto de flujos de datos acíclicos dirigidos (Directed Acyclic Graph, DAG) para representar y ejecutar tareas de procesamiento de datos.

Después de considerar cuidadosamente cada una de estas alternativas, hemos seleccionado Docker como la mejor opción para el empaquetado de nuestra aplicación en contenedores y para el

procesamiento de datos distribuido Apache Spark según nuestras necesidades, especialmente para la realización de microservicios. Además de la facilidad de uso de ambas, y la amplia gama de herramientas que se presentan, a su vez ya hemos venido aprendiendo a como hacer el uso de ellas gracias a las prácticas realizadas en clase el cual han aportado de gran manera a los conocimientos desarrollados. Por lo tanto confiamos en que esta selección se adaptará de una forma más eficiente a las necesidades de nuestra aplicación.

Propuesta del pipeline, componentes o algoritmos a utilizar

Fig. 4: “Pipeline de los componentes”



Ingesto: El camino ideal por el cual se tuvo que haber utilizado era el uso de Pyspark para poder procesar la gran cantidad de datos que disponía el dataset seleccionado, pero en realidad se hicieron uso de unos scripts de Python; El primer script, llamado "bbs71_flights_etl.py", se encarga de realizar un proceso de extracción, transformación y carga (ETL) para cambiar los formatos de tiempo de minutos a formato de 24 horas. Estos cambios se guardan en un archivo CSV llamado "flights.csv". Luego, el script "bbs71_flight_data.py" crea una colección en MongoDB a partir del archivo CSV anteriormente creado, utilizando el modelo diseñado en la primera etapa del proyecto. A continuación, el script "bbs71_user_creation.py" crea usuarios clasificados según su tipo (aerolínea o aeropuerto) utilizando la colección de vuelos. Por último, el script "bbs71_stats.py" crea una colección de estadísticas (flight_stats) clasificadas según el tipo de usuario.

Almacenamiento: Posterior a la ingesta de los datos estos fueron almacenados en mongodb mediante 3 colecciones, una diseñada al modelado de mongo para la organización de la información de los vuelos (flights) en donde se guardaron 600 mil documentos de los cuales en total eran 6 millones, que por temas de almacenamiento no se guardaron en su totalidad. Se hizo el mismo proceso con los usuarios (users), de los cuales se obtuvieron 387, para los aeropuertos se obtuvieron 366 y 21 aerolíneas. Por último, se creó la colección de estadísticas llamada "flight_stats". Para obtener estos datos, se extrajeron 387 documentos, ya que se generan a partir de cada usuario registrado en la aplicación.

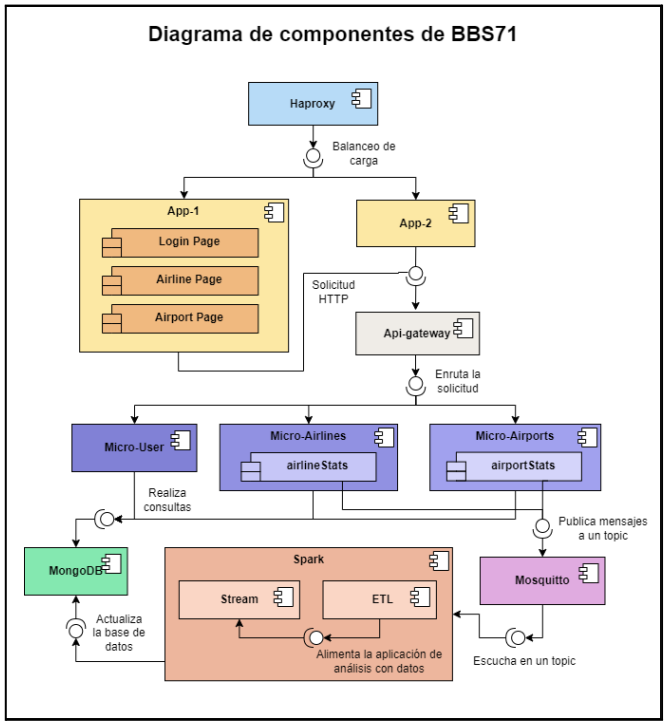
Cómputo: En este caso, la idea principal era utilizar Spark Streaming para realizar el análisis distribuido de los datos. Sin embargo, debido al volumen de datos disponible, no se consideró necesario utilizar un broker tan robusto como Apache Kafka. En este caso particular, los datos disponibles eran estáticos y no requerían ser compartidos en tiempo real. Sin embargo, en un contexto real, es común que los datos de los vuelos sean dinámicos y se necesite compartir información en tiempo real para el análisis distribuido. Así que se utilizó MQTT como intermediario para la aplicación de Apache Spark, esto porque gracias a él es posible el procesado de datos en paralelo y está hecha para leer grandes volúmenes de datos.

Uso: Por último para la interfaz de este análisis, se empleó Vue de modo que las consultas de las estadísticas se visualizan en un dashboard

variando estas según al usuario que se ingrese, mostrando información relevante a los aeropuertos o aerolíneas.

Diagrama de los componentes a utilizar

Fig. 5: “Diagrama de los componentes”



Relación y flujo de trabajo entre los componentes

Se inicia con una petición al balanceador de carga HAProxy. Este balanceador distribuirá las solicitudes entre dos instancias de la aplicación web: la App 1 y su réplica, la App 2. Ahora bien, lo primero que el usuario verá al ingresar a la app página será el login.

Una vez autenticado, se utilizan microservicios como intermediarios entre el Frontend y la base de datos. El Api Gateway dirige las solicitudes a los microservicios correspondientes, los cuales consultan la base de datos para obtener los datos necesarios. Para obtener estadísticas relevantes, los microservicios envían mensajes a un broker de mensajes MQTT. Spark procesa estos mensajes y actualiza la base de datos en función de su contenido. Los microservicios también se encargan de la autenticación, verificando las credenciales en la base de datos. Si la autenticación es exitosa, los usuarios pueden acceder a un menú de visualización con diversas opciones de navegación y herramientas, incluyendo un dashboard específico para cada tipo de usuario.

Para el análisis distribuido Spark toma dos componentes el de ETL y Stream, el de ETL hace el respectivo cambio de formatos que alimentan a la aplicación de Stream, este realiza las consultas para las estadísticas de aeropuertos y aerolíneas.

Descripción de los componentes

En el balanceador de carga HAProxy, se utiliza el algoritmo de balanceo Round Robin para distribuir de manera equilibrada las solicitudes entre las aplicaciones disponibles (App 1 y App 2) . Esto ayuda a evitar la sobrecarga de un solo servidor y mejora el rendimiento y la disponibilidad de la aplicación en general.

La aplicación web consta de tres páginas principales: la página de inicio de sesión, la página para aeropuertos y la página para aerolíneas.

En la página de inicio de sesión, los usuarios deben proporcionar sus credenciales para autenticarse. Dependiendo del tipo de usuario, ya sea aeropuerto o aerolínea, se utilizarán diferentes métodos de autenticación.

Los aeropuertos inician sesión utilizando la abreviación de la ciudad y una contraseña basada en el código del aeropuerto de destino. Esta información se valida para el acceso. Después del inicio de sesión, se muestra una tabla con los vuelos programados para ese aeropuerto. La tabla incluye la fecha, hora, ciudad de origen, aerolíneas involucradas, número de vuelo y estado. También se pueden aplicar filtros para mostrar solo los vuelos de salida o llegada según se requiera.

Las aerolíneas inician sesión utilizando su nombre y una contraseña basada en su código IATA. Esta información se utiliza para autenticarse. Después del inicio de sesión, se muestra una tabla que contiene los vuelos específicos de la aerolínea. La tabla incluye detalles como el número de vuelo, la matrícula del avión, el origen, el destino, el estado del vuelo y las horas de salida y llegada. Además,

se ofrecen filtros para mostrar solo los vuelos con determinados orígenes y destinos, según las necesidades de la aerolínea.

Para gestionar y exponer los múltiples microservicios, es necesario correr cada uno por un puerto diferente, pero mediante una Api Gateway se gestionan y exponen múltiples servicios a través de un puerto por el que salen todos los microservicios, en este caso lo asignamos en el 3000. Mientras que los otros salen por el 3001 (Usuarios), 3002 (aerolíneas) y 3003 (aeropuertos).

El microservicio de aerolíneas dispone de 4 rutas, la primera “/:iata” dedicada a traer los datos de los vuelos dependiendo de la aerolínea que se identifique a partir de código IATA, en segundo lugar está "/stats/:iata" destinada a arrojar las estadísticas a partir de la aerolínea, filtrando solo los datos que tiene que ver con ella. Después está "/aircraft/:iata" que hace la labor de seleccionar únicamente las aeronaves que tiene la aerolínea. Para terminar esta “/routes/:iata" que depura las rutas por el código IATA de la aerolínea.

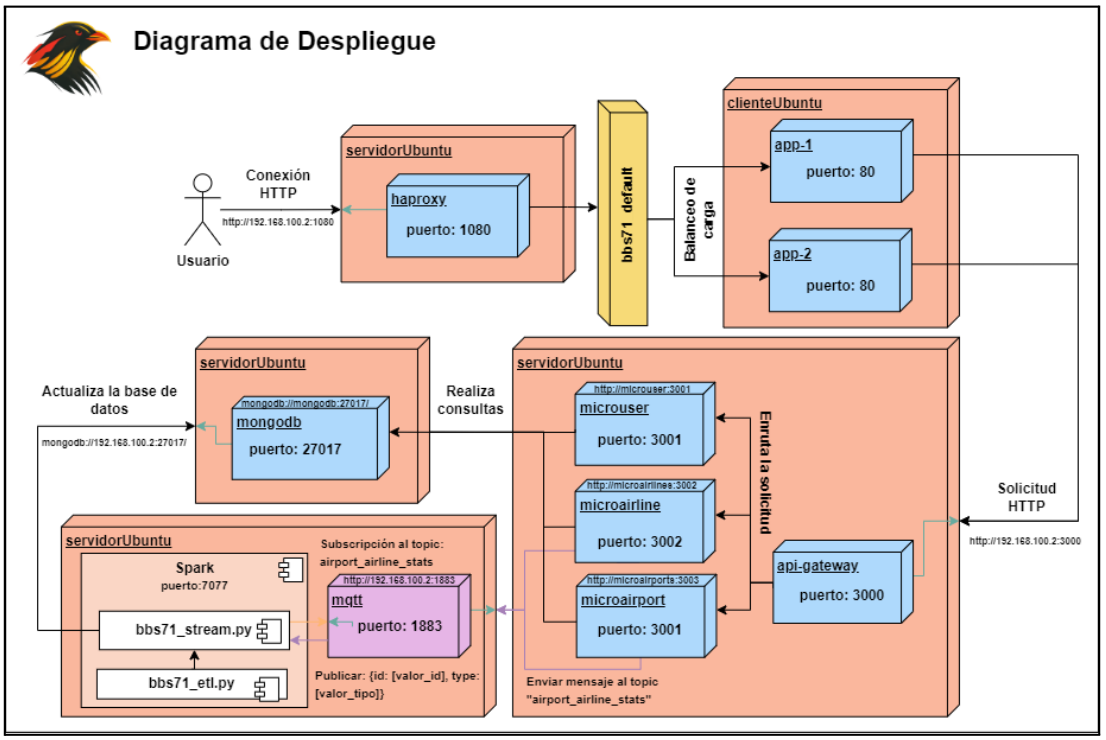
Por otro lado el microservicio de aeropuertos cuenta con 5 rutas, donde la primera “/departure/:originAirportID” busca obtener información sobre los vuelos de salida de un aeropuerto, basándose en el ID del aeropuerto de origen. En esta segunda ruta “/arrival/:destAirportID” se trae la información sobre los vuelos de llegada a un aeropuerto específico. Por otro lado “/stats/:airportID” está encargada de extraer las estadísticas de un aeropuerto. A su vez “/airlines/:airportID” recopila información sobre las aerolíneas asociadas a un aeropuerto específico. Por último está "/routes/:airportID" que obtiene información sobre las rutas asociadas a un aeropuerto en particular.

En este caso, se utiliza el protocolo MQTT para la comunicación entre los microservicios de aerolíneas y aeropuertos, y la aplicación de Spark. Los microservicios publican mensajes en el topic "airport_airline_stats" en formato {id:, type} con información relevante para el análisis de estadísticas. La aplicación de Spark se suscribe a este topic y recibe los mensajes. Según el tipo de mensaje (aerolínea o aeropuerto), Spark realiza análisis específicos para obtener las estadísticas requeridas. Una vez que Spark ha realizado el análisis, actualiza la colección de estadísticas ("flight_stats") en la base de datos. Los valores obtenidos se utilizan para mantener actualizadas las estadísticas y facilitar su acceso para consultas posteriores.

Las estadísticas incluyen información como el número total de vuelos ("total_flights"), vuelos cancelados ("flights_cancelled"), cantidad de aviones ("tail_numbers"), vuelos retrasados ("flights_delay"), vuelos durante la temporada de vacaciones ("flights_holidaySeason") y el mes con mayor cantidad de vuelos ("flights_max_month"). La aplicación de Spark se ejecuta utilizando un maestro en la máquina correspondiente, y es necesario contar con un esclavo que esté escuchando las peticiones. El esclavo puede ser utilizado en la misma IP que el maestro para simplificar el despliegue y la configuración.

Diagrama de despliegue

Fig. 6: “Diagrama de despliegue”



Implementación y pruebas

Implementación solución

La solución implementada consta de una aplicación con diferentes funciones. Se utilizó Docker para el despliegue de la aplicación y Apache Spark para el análisis distribuido del dataset.

Los componentes necesarios para la aplicación se organizaron en una carpeta, donde se despliegan como servicios. Se crearon imágenes de Docker para la mayoría de los servicios, incluyendo Api Gateway, microbrowser, microairlines, microairport, haproxy, app1 y app2. Los servicios restantes, mongodb y mqtt, se trabajaron mediante volúmenes. Se utilizó un archivo docker-compose.yml para llamar a todos los servicios necesarios y desplegar la aplicación.

Para el análisis distribuido con Spark, se creó una carpeta separada. Se trató el dataset en una aplicación Spark, generando nuevos archivos CSV. Otra aplicación Spark se encargó del análisis distribuido, cargando los archivos CSV resultantes y realizando las consultas necesarias. Esta aplicación estaba suscrita al topic del broker mqtt y actualizaba la base de datos con los resultados obtenidos.

Para utilizar la aplicación, era necesario tener todos los servicios en funcionamiento, incluyendo la aplicación de análisis de Spark. Se utilizó Docker Swarm para ejecutar los contenedores en nodos diferentes y permitir el escalado de servicios. La aplicación se ejecutó en dos nodos, uno para la aplicación web y otro para los demás servicios. Se usó el comando 'docker stack deploy' para levantar los servicios y se ejecutó la aplicación de Spark.

Prueba de funcionamiento de componentes implementado

La aplicación implementada se puede probar ingresando la dirección IP del servidor seguida del puerto 1080 en un navegador (http://192.168.100.2:1080/). El balanceador de carga haproxy distribuye la carga entre dos servicios de aplicación. Al acceder a la dirección, se muestra un formulario de inicio de sesión donde se solicita el usuario y la contraseña. Dependiendo del tipo de usuario (aeropuerto o aerolínea), se muestra una vista diferente.

Si el usuario es una aerolínea, se muestra un menú con cuatro opciones: 'flights' muestra todos los

vuelos de la aerolínea, 'airplanes' muestra los aviones y el número de vuelos realizados, 'flight routes' muestra los posibles orígenes y destinos de los vuelos, y 'dashboard' muestra información interesante sobre los vuelos en forma de tarjetas. Para las tres primeras opciones se utiliza el microservicio de aerolínea, mientras que para la última opción se utiliza el análisis distribuido a través del broker.

Si el usuario es un aeropuerto, también se muestra un menú con cuatro opciones: Arrivals/Departures' muestra los vuelos que llegan o salen de ese aeropuerto, 'Airlines' muestra las aerolíneas que han realizado vuelos en el aeropuerto y cuántos vuelos han realizado, 'Flight routes' cumple la misma función que en el caso de las aerolíneas, y 'Dashboard' muestra datos interesantes sobre los vuelos. Para las tres primeras opciones se utiliza el microservicio de aeropuerto, mientras que para el dashboard se utiliza el análisis distribuido para obtener y mostrar los resultados en la aplicación.

Pruebas de escalabilidad y desempeño

El objetivo de las pruebas es evaluar el rendimiento de los diferentes componentes del sistema, como el balanceador, el Api Gateway, los microservicios y la aplicación de stream. Se han diseñado los siguientes escenarios de carga:

Prueba de carga normal: Se simuló una carga típica en la aplicación utilizando 100 usuarios. Cada usuario realizó acciones durante 30 segundos y repitió el proceso 3 veces (3 loops). Esta prueba permitió evaluar el rendimiento de la aplicación bajo una carga típica.

Prueba de carga alta: Se incrementó la carga en la aplicación para simular una situación de mayor demanda. Se utilizaron 500 usuarios que realizaron acciones durante 30 segundos y repitieron el proceso 5 veces (5 loops). Esta prueba permitió evaluar cómo respondió la aplicación cuando se sometió a una carga más intensa.

Prueba de estrés: Se simuló una carga extrema en la aplicación. Se utilizaron 1000 usuarios que realizaron acciones durante 30 segundos y repitieron el proceso 10 veces (10 loops). El objetivo de esta prueba fue evaluar la capacidad de la aplicación para manejar una carga máxima y determinar si existían puntos de saturación o cuellos de botella en el sistema**.**

Métricas a recolectar:

1. Tiempo de respuesta promedio de las acciones realizadas por los usuarios.
2. Desviación estándar del tiempo de respuesta.
3. Porcentaje de error en las acciones realizadas.

Estas pruebas se realizan utilizando JMeter, una herramienta ampliamente utilizada para pruebas de carga y rendimiento. Con los resultados obtenidos, se podrán identificar posibles áreas de mejora y tomar medidas para optimizar el rendimiento del sistema, basándose en las métricas recolectadas.

Se identificó que la estructura actual del sistema presentaba un posible cuello de botella en la aplicación de stream. Se observó que las estadísticas no se calculaban para los usuarios hasta que no se completaba el cálculo de las estadísticas de un usuario anterior. Esto generaba una demora considerable en el procesamiento de las solicitudes de los usuarios, lo que afectaba el rendimiento general del sistema.

Para abordar el problema del cuello de botella en el componente de stream, se implementó una solución utilizando el enfoque de procesamiento paralelo. En lugar de esperar a que se complete el cálculo de las estadísticas de un usuario antes de pasar al siguiente, se diseñó un escenario en el que cada solicitud de métricas se manejaba como un hilo independiente, permitiendo un procesamiento simultáneo y paralelo de las solicitudes de los usuarios.

En la prueba específica del componente de stream, se simuló una carga concurrente con 1 o 5 usuarios. Cada usuario realizaba solicitudes independientes para obtener métricas y se procesaban en hilos dedicados. Esto permitió

ejecutar múltiples solicitudes simultáneamente, optimizando el tiempo de respuesta y reduciendo el cuello de botella. Se evaluaron las solicitudes a las rutas "/airline/stats" y "/airport/stats", registrando los tiempos de respuesta y considerando el uso de hilos.

Ruta HTTP	Usuarios simultáneos	Tiempo de respuesta	Hilos
/airline/stats	1	36740	Sí
/airline/stats	5	180979,4	Sí
/airline/stats	1	38837	No
/airline/stats	5	201952,4	No
/airport/stats	1	42228	Sí
/airport/stats	5	242405,8	Sí
/airport/stats	1	41116	No
/airport/stats	5	2624292	No

En resumen, el procesamiento paralelo con hilos mejoró la capacidad de respuesta, el rendimiento del componente de stream durante la carga concurrente de usuarios y reducir el impacto del cuello de botella en el componente de stream.

Resultados de las pruebas de carga normal:

Componente	Réplicas	Ruta HTTP	Tiempo de respuesta	Desviación estándar	Porcentaje de error
Haproxy	1	Na	10	27,72	0,00%
Api Gateway	1	/user/login	33633	16013,21	0,00%
Api Gateway	1	/airline/QX	93	20,06	0,00%
Api Gateway	1	/airline/aircraft/QX	178794	70230,68	47,67%
Api Gateway	3	/airline/aircraft/QX	205384	61193,14	0,00%
Api Gateway	3	/airport/departure/10620	75	7,01	0,00%
Api Gateway	3	/airport/airlines/10620	13559	7499,86	0,00%

En general, los resultados de las pruebas de carga normal muestran que el sistema tiene un rendimiento aceptable bajo una carga típica. Los tiempos de respuesta y las desviaciones estándar son relativamente bajos en la mayoría de las rutas evaluadas, lo que indica una buena capacidad de respuesta del sistema.

Resultados de las pruebas de carga alta:

Componente	Réplicas	Ruta HTTP	Tiempo de respueta	Desviación estándar	Porcentaje de error
Haproxy	1	Na	4	3,31	0,00%
Api Gateway	1	/user/login	77591	34461,35	0,00%
Api Gateway	1	/airline/QX	10765	4502,01	0,00%
Api Gateway	3	/airline/aircraft/QX	26154	48073,09	99,56%
Api Gateway	5	/airline/aircraft/QX	46364	67566,53	91,72%
Api Gateway	3	/airport/departure/10620	12334	5364,77	0,00%
Api Gateway	3	/airport/airlines/10620	71895	30990,61	16,16%

En las pruebas de carga alta, se observa que el sistema experimentó un aumento significativo en los tiempos de respuesta en comparación con las pruebas de carga normal. Esto indica que bajo una carga más intensa, el sistema puede enfrentar dificultades para mantener un rendimiento óptimo.

Resultados de las pruebas de estrés:

Componente	Réplicas	Ruta HTTP	Tiempo de respueta	Desviación estándar	Porcentaje de error
Haproxy	1	Na	4	357,29	41,00%
Api Gateway	3	/user/login	108163	26708,88	90,09%
Api Gateway	3	/airline/QX	64048	21761,82	0,00%
Api Gateway	3	/airline/aircraft/QX	9999	5798,39	100,00%
Api Gateway	5	/airline/aircraft/QX	3900	2262,31	100,00%
Api Gateway	3	/airport/departure/10620	39058	13390,2	0,00%
Api Gateway	5	/airport/airlines/10620	10294	7317,3	100,00%

En las pruebas de carga de estrés, se observa un incremento significativo en el tiempo de respuesta y la desviación estándar en comparación con las pruebas de carga normal. Esto indica que el sistema está experimentando dificultades para manejar la carga extrema y se están generando errores en algunas de las solicitudes.

Según los resultados de las pruebas, se recomienda utilizar 3 réplicas de Api Gateway para obtener un mejor rendimiento. En el caso de Haproxy, una única réplica fue suficiente para cumplir con los requisitos de las pruebas de carga normal, carga alta y de estrés. Por lo tanto, para Haproxy no es necesario agregar más réplicas en este momento.

Conclusiones

Del anterior trabajo realizado, se reconoce que el planteamiento de una arquitectura de microservicios con conexión a una Api Rest ofrece una mayor capacidad de dividir una aplicación en módulos independientes - autónomos y flexibilidad en el desarrollo de aplicaciones. Esto permite una gestión más eficiente de los recursos,

una mejor escalabilidad y la capacidad de adaptarse rápidamente a los cambios en los requisitos, lo que conduce a un desarrollo más ágil y una mayor capacidad de respuesta en el entorno empresarial actual.

Lo anterior, lleva a la implementación de un balanceador de carga, que se convierte en una gran ayuda para no saturar una aplicación o página con un gran número de peticiones a un solo servidor, generando así una solución la cual puede distribuir sus cargas de trabajo.

La inclusión de Apache Spark en el análisis distribuido del proyecto habilita la capacidad de procesar eficientemente grandes conjuntos de datos, lo que conduce a la generación de estadísticas significativas. Además, permite la integración sinérgica de los microservicios con el dashboard correspondiente.

Gracias al empaquetado de un clúster utilizando Docker, se logró una notable facilidad en términos de escalabilidad y orquestación de los contenedores que alojan los microservicios. Esto permitió una gestión eficiente de los recursos y una respuesta ágil a las demandas cambiantes de la aplicación.

Por último, una arquitectura adecuada es fundamental para garantizar un sistema de alto rendimiento y evitar problemas como los cuellos de botella. Es importante considerar aspectos como la escalabilidad, la eficiencia en el procesamiento y la optimización de recursos durante el diseño y la implementación del sistema. Sin embargo, es necesario realizar un seguimiento continuo y análisis para identificar posibles limitaciones y aplicar mejoras adicionales en caso de persistir los cuellos de botella.

Referencias

[1] (2022) kaggle Website.[Online].Disponible en: https://www.kaggle.com/datasets/robikscube/flight-delay-dataset-20182022?select=Airlines.csv

[2] (2006) AWS amazon Kubernetes.[Online] Disponible en: https://aws.amazon.com/es/kubernetes/

[3] (2012) Apache Mesos [Online]. Disponible en: https://mesos.apache.org/

[4] (2013) Docker Website. [Online]. Disponible en: https://www.docker.com/products/container-runtime/

[5] (2014) IBM Website. [Online]. Disponible en: https://www.ibm.com/mx-es/topics/apache-spark

[6] (2013) AWS amazon Hadoop. [Online]. Disponible en: https://aws.amazon.com/es/elasticmapreduce/details/hadoop/

[7] (2023) Aprender BIG DATA Apache Flink.[Online] Disponible en: https://aprenderbigdata.com/introduccion-apache-flink/