MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

TRABAJO FIN DE MÁSTER

**DESARROLLO DE UNA SOLUCIÓN BDaaS (BIG DATA AS A SERVICE) PARA EL ANÁLISIS DE GRAFOS TEMPORALES A GRAN ESCALA**

DAVID PISONERO FUENTES

2022

MÁSTER uNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

trabajo fin de MÁSTER

**Título:** Desarrollo de una solución BDaaS (Big Data as a Service) para el análisis de grafos temporales a gran escala

**Autor:** D. David Pisonero Fuentes

**Tutor:** D. Félix Cuadrado Latasa

**Ponente:** D. Juan Carlos Dueñas López

**Departamento:** Departamento de Ingeniería se Sistemas Telemáticos (DIT)

Miembros del Tribunal

**Presidente:** D. ……………

**Vocal:** D. …………..

**Secretario:** D. …………..

**Suplente:** D. ……………..

Los miembros del tribunal arriba nombrados acuerdan otorgar la calificación de: ………

Madrid, a de de 20…

**UNIVERSIDAD POLITÉCNICA DE MADRID**

**ESCUELA TÉCNICA SUPERIOR**

**DE INGENIEROS DE TELECOMUNICACIÓN**

****

MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

TRABAJO FIN DE MÁSTER

**DESARROLLO DE UNA SOLUCIÓN BDaaS (BIG DATA AS A SERVICE) PARA EL ANÁLISIS DE GRAFOS TEMPORALES A GRAN ESCALA**

DAVID PISONERO FUENTES

2022

RESUMEN

Históricamente, los grafos han sido una herramienta de gran utilidad en diferentes disciplinas, desde las Matemáticas hasta las Telecomunicaciones, pasando por las Ciencias Sociales.

En aras de aprovechar la información que estos nos ofrecen, especialmente en el mundo actual, donde convivimos con la realidad del Big Data, surge la necesidad de procesar dichos grafos de la forma más rápida y eficiente posible, hecho que supone un reto de considerable dificultad, debido a la naturaleza interactiva de los vértices y aristas que componen estas estructuras.

Así surge Raphtory, una herramienta capaz de realizar el procesado **distribuido** de grafos mediante la implementación del paradigma teórico “***Thinking like a vertex***”, desarrollado por Google, que sí permite la distribución de operaciones de procesado de grafos entre diferentes actores. De hecho, Raphtory va más allá, pues también incluye la posibilidad de llevar a cabo análisis temporales, es decir, estudiar la evolución de un grafo a lo largo del tiempo, mediante el uso de ventanas.

Así pues, este Trabajo Fin de Máster realiza un estudio teórico sobre el procesado distribuido de grafos a través del paradigma antes mencionado, analiza el funcionamiento de la herramienta Raphtory, y experimenta con sus diferentes formas de despliegue, desde la opción monolítica, hasta la distribuida basada en servicios alojados en contenedores y pods.

A continuación, se implementa una solución **BDaaS** (Big Data as a Service), que permite el despliegue distribuido de Raphtory en Google Cloud haciendo uso de Kubernetes, aplicando cierto nivel de automatización, de manera que un potencial usuario pueda utilizar esta herramienta eliminando las costosas barreras de entrada que supone el procesado distribuido de grafos masivos, gracias a las ventajas que ofrece la **Nube**.

Se finaliza el trabajo con las conclusiones extraídas sobre el funcionamiento de Raphtory, y los despliegues desarrollados, y se plantean unas líneas futuras para la evolución de este servicio BDaaS.

summary

Historically, graphs have been a very useful tool among different disciplines, ranging from Maths to Telecommunications, including Social Sciences.

In order to take advantage of the information they offer, specially nowadays, when we have to coexist with the reality of Big Data, the necessity of processing said graphs as fast and efficiently possible arises, which creates a considerably difficult challenge, due to the interactive nature of the vertices and edges composing these structures.

This is why Raphtory appears, a tool able to carry out the **distributed** processing of graphs thanks to the implementation of the theoretical paradigm “**Thinking like a vertex**”, developed by Google, which does enable the distribution of the graph processing operations between different actors. In fact, Raphtory goes beyond that, including the possibility of conducting temporal analysis, that is, studying the graph evolution through time, making use of windows.

That being the case, this Master’s Thesis carries out a theoretical study about the distributed graph processing by means of said paradigm, it analyses Raphtory’s functioning, and it experiments with its different ways of deployment, from the monolithic option, to the distributed one based on services hosted inside containers and pods.

Following that, a **BDaaS** (Big Data as a Service) solution is implemented, which offers a distributed deployment for Raphtory on Google Cloud using Kubernetes, applying a certain level of automatization, so a potential user could use this tool without the expensive barrier entries brought up by the massive graphs distributed processing, thanks to the possibilities offered by the **Cloud**.

This work ends with the conclusions obtained about Raphtory’s functioning, and the implemented deployments, as well as some future proposals relating to the evolution of this BDaaS service.

PALABRAS CLAVE

Raphtory, procesado de grafos, análisis temporal, sistemas distribuidos, Big Data, Nube, Kubernetes, Docker, Docker Compose, GKE, Cloud Storing, BdaaS.

KEYWORDS

Raphtory, graph processing, temporal analysis, distributed systems, Big Data, Cloud, Kubernetes, Docker, Docker Compose, GKE, Cloud Storing, BDaaS.

índice del contenido

[1. Introducción y objetivos 1](#_Toc106387190)

[1.1. Introducción 1](#_Toc106387191)

[1.2. Objetivos 2](#_Toc106387192)

[2. Desarrollo 3](#_Toc106387193)

[2.1. Estado del arte 3](#_Toc106387194)

[2.1.1. Pregel y Thinking like a vertex 3](#_Toc106387195)

[2.1.2. Análisis temporal de grafos 6](#_Toc106387196)

[2.1.3. Programación en Scala 6](#_Toc106387197)

[2.1.4. Computación en la nube y herramientas DevOps 7](#_Toc106387198)

[2.1.5. almacenamiento basado en objetos 7](#_Toc106387199)

[2.1.6. Amazon EMR (Elastic Map-Reduce) 8](#_Toc106387200)

[2.2. Análisis detallado de Raphtory 8](#_Toc106387201)

[2.2.1. Componentes de Raphtory y funcionamiento 9](#_Toc106387202)

[2.2.2. Análisis temporal de grafos en raphtory 13](#_Toc106387203)

[2.3. Despliegues básicos de raphtory: Single y PD 15](#_Toc106387204)

[2.3.1. Despliegue monolítico de raphtory 15](#_Toc106387205)

[2.3.2. Despliegue pseudo distribuido de Raphtory 18](#_Toc106387206)

[2.4. Despliegue de Raphtory con Docker y Docker Compose 20](#_Toc106387207)

[2.5. Despliegue de Raphtory como BDaaS en la Nube 21](#_Toc106387208)

[2.5.1. Estudio y evaluación de tecnologías para el despliegue en la nube 21](#_Toc106387209)

[2.5.2. Despliegue de Raphtory en Google Cloud 25](#_Toc106387210)

[3. Resultados 27](#_Toc106387211)

[3.1. Resultados del despliegue de Raphtory en modo Single 27](#_Toc106387212)

[3.2. Resultados del despliegue de Raphtory en modo pseudo distribuido 30](#_Toc106387213)

[3.3. Resultado del despliegue de Raphtory con Docker y Docker Compose 37](#_Toc106387214)

[3.4. Resultados del despliegue de Raphtory como un BDaaS en la Nube 41](#_Toc106387215)

[3.4.1. Construcción y subida de las imágenes 42](#_Toc106387216)

[3.4.2. Gestiones en Google Cloud 43](#_Toc106387217)

[3.4.3. Configuración de Google Cloud SDK 45](#_Toc106387218)

[3.4.4. Despliegue de Apache Pulsar 46](#_Toc106387219)

[3.4.5. Despliegue de Raphtory en el clúster de GKE 47](#_Toc106387220)

[3.4.6. Automatización del despliegue con GitHub Actions 54](#_Toc106387221)

[4. Conclusiones y líneas futuras 55](#_Toc106387222)

[4.1. Conclusiones 55](#_Toc106387223)

[4.2. Líneas futuras 55](#_Toc106387224)

[5. Bibliografía 56](#_Toc106387225)

[Anexo A: aspectos éticos, económicos, sociales y ambientales 59](#_Toc106387226)

[A.1 Introducción 59](#_Toc106387227)

[A.2 Descripción de impactos relevantes relacionados con el proyecto 59](#_Toc106387228)

[A.3 Análisis detallado de algunos de los principales impactos 60](#_Toc106387229)

[A.4 Conclusiones 60](#_Toc106387230)

[Anexo B: presupuesto económico 62](#_Toc106387231)

Índice de ilustraciones

[Ilustración 1. Tiempo de ejecución frente a número de nodos trabajadores en experimento con Pregel. 5](#_Toc106387232)

[Ilustración 2. Tiempo de ejecución frente a número de vértices en experimento con Pregel. 5](#_Toc106387233)

[Ilustración 3. Diagrama que representa los componentes de Raphtory y sus interacciones. 9](#_Toc106387234)

[Ilustración 4. Diagrama de flujo de la adición y borrado de entidades. 11](#_Toc106387235)

[Ilustración 5. Código de un análisis temporal que realiza un filtrado. 14](#_Toc106387236)

[Ilustración 6. Código de un análisis temporal con perspectivas y ventanas. 14](#_Toc106387237)

[Ilustración 7. Posibilidades del análisis temporal en Raphtory. 15](#_Toc106387238)

[Ilustración 8. Ejemplo de despliegue monolítico. 16](#_Toc106387239)

[Ilustración 9. Fragmento de contenido de los datos de entrada. 16](#_Toc106387240)

[Ilustración 10. Builder utilizado en el ejemplo. 17](#_Toc106387241)

[Ilustración 11. Código fundamental de la clase RaphtoryService. 18](#_Toc106387242)

[Ilustración 12. Código del objeto Component. 19](#_Toc106387243)

[Ilustración 13. Diagrama del despliegue PD y sus comunicaciones. 19](#_Toc106387244)

[Ilustración 14. Despliegue en Docker y Docker Compose. 21](#_Toc106387245)

[Ilustración 15. Diagrama de la estructura de un clúster de Apache Pulsar. 23](#_Toc106387246)

[Ilustración 16. Despliegue de Raphtory como BDaaS en Google Cloud 26](#_Toc106387247)

[Ilustración 17. Terminal con las versiones instaladas de las herramientas necesarias para Raphtory. 27](#_Toc106387248)

[Ilustración 18. Comando y salida de la compilación del JAR de Raphtory. 28](#_Toc106387249)

[Ilustración 19. Éxito de la compilación del JAR de Raphtory. 28](#_Toc106387250)

[Ilustración 20. Compilación en SBT del ejemplo de LOTR. 28](#_Toc106387251)

[Ilustración 21. Posibilidades de ejecución del ejemplo monolítico. 29](#_Toc106387252)

[Ilustración 22. Ejecución exitosa de FileOutputRunner en el ejemplo monolítico. 29](#_Toc106387253)

[Ilustración 23. Directorio con la salida de FileOutputRunner. 30](#_Toc106387254)

[Ilustración 24. Salida de la ejecución de DegreeSeparation en el despliegue monolítico. 30](#_Toc106387255)

[Ilustración 25. Ejemplo de configuración de las variables de entorno en un despliegue PD. 31](#_Toc106387256)

[Ilustración 26. Logs de despliegue del líder en PD. 32](#_Toc106387257)

[Ilustración 27. Un nuevo componente se une al clúster, siendo registrado por el líder. 32](#_Toc106387258)

[Ilustración 28. Despliegue del Spout, y unión al clúster del Líder. 32](#_Toc106387259)

[Ilustración 29. Despliegue del Builder, y unión al clúster del Líder. 33](#_Toc106387260)

[Ilustración 30. Log del terminal en el despliegue PD que muestra la aparición de Dead Letters. 33](#_Toc106387261)

[Ilustración 31. Despliegue del gestor de particiones en PD. 34](#_Toc106387262)

[Ilustración 32. Despliegue del Query Manager en la modalidad PD. 34](#_Toc106387263)

[Ilustración 33. Log del líder que muestra que el clúster de Raphtroy está preparado. 34](#_Toc106387264)

[Ilustración 34. Clase Client para probar Raphtory PD. 35](#_Toc106387265)

[Ilustración 35. Ejecución de un cliente que se conecta con el clúster. 35](#_Toc106387266)

[Ilustración 36. El líder reconoce al cliente en el clúster. 35](#_Toc106387267)

[Ilustración 37. Salidas de la ejecución de Raphtory en modo PD con dos particiones. 36](#_Toc106387268)

[Ilustración 38. Salida de la ejecución de GraphState en Raphtory PD 36](#_Toc106387269)

[Ilustración 39. El líder elimina los componentes de Raphtory conforme estos se van apagando. 36](#_Toc106387270)

[Ilustración 40. Dockerfile empleado en el despliegue Dockerizado. 37](#_Toc106387271)

[Ilustración 41. Fichero docker-compose.yml con la configuración del despliegue dockerizado. 38](#_Toc106387272)

[Ilustración 42. Configuración del contenedor del cliente. 39](#_Toc106387273)

[Ilustración 43. Parte de los logs de la construcción de las imágenes. 40](#_Toc106387274)

[Ilustración 44. Parte de los logs del despliegue en Docker. 40](#_Toc106387275)

[Ilustración 45. Parte del log de "docker-compose up", con el clúster preparado. 40](#_Toc106387276)

[Ilustración 46. Despliegue del clúster operativo en Docker Desktop. 41](#_Toc106387277)

[Ilustración 47. Entrada en el cliente, y comprobación de los resultados. 41](#_Toc106387278)

[Ilustración 48. Dockerfile para el despliegue en la Nube. 42](#_Toc106387279)

[Ilustración 49. Captura de mi repositorio en Docker Hub. 43](#_Toc106387280)

[Ilustración 50. Proyecto creado en Google Cloud. 43](#_Toc106387281)

[Ilustración 51. Creación de la cuenta de servicio. 44](#_Toc106387282)

[Ilustración 52. Claves de la cuenta de servicio. 44](#_Toc106387283)

[Ilustración 53. Clúster de tres nodos creado en Google Kubernetes Engine. 45](#_Toc106387284)

[Ilustración 54. Cloud Storage con el bucket creado. 45](#_Toc106387285)

[Ilustración 55. Configuración de Google Cloud SDK. 45](#_Toc106387286)

[Ilustración 56. Credenciales y conexión con GKE. 46](#_Toc106387287)

[Ilustración 57. Script de lanzamiento de Apache Pulsar. 46](#_Toc106387288)

[Ilustración 58. Contexto del clúster de GKE. 47](#_Toc106387289)

[Ilustración 59. Despliegue de Apache Pulsar en GKE. 47](#_Toc106387290)

[Ilustración 60. Redirección de puertos en Apache Pulsar. 47](#_Toc106387291)

[Ilustración 61. Variables de entorno del despliegue en Raphtory en la Nube. 48](#_Toc106387292)

[Ilustración 62. Código relevante de CloudSpoutç 49](#_Toc106387293)

[Ilustración 63. Despliegue en GKE de Raphtory. 49](#_Toc106387294)

[Ilustración 64. Código relevante de la clase CloudBucketOutputFormat. 50](#_Toc106387295)

[Ilustración 65. Comando para entrar en un Pod. 50](#_Toc106387296)

[Ilustración 66. Interfaz gráfica de Lens. 50](#_Toc106387297)

[Ilustración 67. Ejecución del cliente en la Nube. 51](#_Toc106387298)

[Ilustración 68. Comienzo del análisis en la Nube. 51](#_Toc106387299)

[Ilustración 69. Finalización del análisis. 51](#_Toc106387300)

[Ilustración 70. Fichero de salida generado en Cloud Storage. 51](#_Toc106387301)

[Ilustración 71. Salida del análisis en la Nube. 52](#_Toc106387302)

[Ilustración 72. Panel de GKE que muestra el despliegue con dos Partition Manager. 52](#_Toc106387303)

[Ilustración 73. Panel de Lens que muestra los dos Partition Manager. 53](#_Toc106387304)

[Ilustración 74. Arranque de la Partición 0. 53](#_Toc106387305)

[Ilustración 75. Arranque de la Partición 1. 53](#_Toc106387306)

[Ilustración 76. Finalización del análisis en la Partición 0. 53](#_Toc106387307)

[Ilustración 77. Finalización del análisis en la Partición 1. 53](#_Toc106387308)

[Ilustración 78. Los dos ficheros de salida del análisis ejecutado con dos particiones. 54](#_Toc106387309)

[Ilustración 79. Los dos ficheros de salida descargados. 54](#_Toc106387310)

Glosario

|  |  |
| --- | --- |
| Término | Significado |
| BDaaS | Big Data as a Service, Big Data como Servicio |
| CLI | Command Line Interface |
| CI/CD | Continuous Integration / Continuous Delivery |
| EMR | Elastic Map-Reduce |
| GCS | Google Cloud Storing |
| GKE | Google Kubernetes Engine |
| IAM | Identity and Access Management, Gestion de la Identidad y el Acceso |
| JAR | Java ARchive |
| JDK | Java Development Kit |
| JVM | Java Virtual Machine |
| K8s | Kubernetes |
| LOTR | Lord Of The Rings, El Señor de los Anillos |
| NIST | National Institute of Standards and Technology, Instituto Nacional de Estándares y Tecnología |
| OOP | Object Oriented Programming |
| PD | Pseudo Distribuido |
| SDK | Software Development Kit |
| SQL | Structured Query Language |
| TFM | Trabajo Fin de Máster |
| TIC | Tecnologías de la Información y la Comunicación |

# Introducción y objetivos

## Introducción

La teoría de grafos empieza a desarrollarse a partir del año 1736, cuando Leonhard Euler publica su *Solutio problematis ad geometriam situs pertinentis* para resolver el problema de los puentes de Königsberg, consistente en encontrar un camino que permitiera recorrer los siete puentes sobre el río Pregel, en la actual Kaliningrado, sin tener que repetir el paso por ninguno de ellos [1].

De este modo, podemos definir un grafo con el siguiente formalismo: G = <V, E>, es decir, una combinación formada por un conjunto de vértices relacionados entre sí mediante un conjunto de aristas (*Edges*), entendiendo por arista como un par no ordenado de vértices, representado por una línea a priori no orientada [2].

Así, los grafos, y la teoría matemática que los estudia, se han ido desarrollando desde el siglo XVIII hasta la actualidad, convirtiéndose en una herramienta muy potente para representar y analizar información de diversa naturaleza.

Es por ello que, históricamente, la teoría de grafos se ha aplicado en multitud de disciplinas. Por supuesto, las matemáticas son la más obvia, pero también podemos encontrar ejemplos en la física, con las Leyes de Kirchhoff para el análisis de circuitos eléctricos, o en la química, con la representación de estructuras moleculares mediante grafos, donde los átomos son vértices, y sus enlaces, aristas.

Más recientemente, la teoría de grafos también se ha expandido hacia la sociología, posicionándose como una herramienta de utilidad en el estudio de las interacciones entre individuos, por ejemplo, analizando cómo se expanden ciertos discursos ideológicos a través de los usuarios de una red social [3]. También podemos encontrar casos de uso en la economía, que, en ocasiones, utiliza grafos para representar las relaciones entre sectores productivos. Incluso en un tema de actualidad como la epidemiología, se emplean grafos para poder estudiar las transmisiones de enfermedades infecciosas entre personas.

Por supuesto, las telecomunicaciones son un campo donde los grafos poseen un impacto muy notable. Más allá del ejemplo ya mencionado de las Leyes de Kirchhoff para el análisis de circuitos, la teoría de grafos es la base del diseño de las redes de comunicaciones, dando lugar al algoritmo de Dijkstra, y siendo la piedra angular de los protocolos de encaminamiento, empleados para organizar el tráfico en redes de diverso calado, incluyendo la universalidad de Internet.

De este modo, podemos concluir que los grafos son una parte fundamental de la sociedad y de nuestras propias vidas, constituyendo una de las herramientas más útiles y efectivas desarrolladas en toda la Historia de la humanidad.

Así, no es de extrañar que, en los últimos tiempos, el procesado de grafos se haya convertido en un área que ha atraído la atención de cada vez más expertos, en aras de aprovechar la información que nos ofrecen de la manera más eficiente posible, especialmente teniendo en cuenta la realidad del Big Data, que implica trabajar con enormes cantidades de datos, lo que se traduce en grafos de carácter masivo.

Y es que el procesado de grafos conforma un reto computacional de elevada dificultad, ya que suscita importante problemas derivados de la naturaleza interactiva de sus componentes, pues, en un grafo de gran tamaño, podemos tener miles de millones de vértices intercambiando información mediante las respectivas aristas.

Si, además, con la intención de mejorar dicho procesado, decidimos aplicar la computación distribuida, teniendo que dividir el grafo en varias particiones, que, por lo explicado anteriormente, no son completamente independientes, la complejidad del problema crece exponencialmente. De este modo, recientemente se han ido desarrollando diferentes paradigmas y tecnologías que tratan de lidiar con este problema, y que comentamos en la siguiente sección, como, por ejemplo, el objeto de este TFM, Raphtory.

## Objetivos

El objetivo fundamental de este Trabajo Fin de Máster es el de implementar un servicio de tipo BDaaS, es decir, Big Data as a Service, desarrollado mediante el despliegue en Google Cloud, haciendo uso de Kubernetes y del almacenamiento basado en objetos, de la herramienta de procesado de grafos conocida como Raphtory.

Para ello, en primer lugar, es necesario llevar a cabo un estudio sobre el procesamiento distribuido de grafos, entrando en detalle en la solución implementada por Raphtory, el paradigma “*Thinking like a vertex”*, y analizando alguna tecnología similar, como Amazon Elastic Map-Reduce, evaluando las ventajas proporcionadas por la Nube y las herramientas DevOps en cuanto a virtualización, computación, y almacenamiento, que habilitan este tipo de servicios.

A continuación, se desarrolla un análisis centrado en Raphtory, especialmente en las partes que lo componen, y en sus diferentes posibilidades de despliegue, poniendo el foco en la opción monolítica y pseudo distribuida.

Con esta información, el siguiente objetivo es la implementación de un despliegue verdaderamente distribuido de Raphtory, haciendo uso de contenedores Docker coordinados por Docker Compose.

Así, llegamos al objetivo final, que es el despliegue de Raphtory completamente distribuido en la nube de Google, utilizando Kubernetes y almacenamiento basado en objetos mediante GKE (Google Kubernetes Engine) y GCS (Google Cloud Storing), proporcionando a un potencial usuario una oferta de analítica como servicio que le permita llevar a cabo este tipo de actividades de procesado de grafos en la Nube con cierto grado de automatización, sorteando así las costosas barreras de entrada que existen en cuanto a complejidad y carga de procesamiento.

Por último, también se realiza una comparativa, a nivel cualitativo, entre las diferentes opciones exploradas, y se ofrecen unas conclusiones acerca de Raphtory, los servicios BdaaS, y el trabajo realizado, extrayendo unas posibles líneas futuras para esta tecnología.

# Desarrollo

En esta sección, se va a llevar a cabo un estado del arte en el que se comentan los retos que supone el procesado de grafos en la actualidad, con grandes cantidades de datos, y haciendo uso de la computación distribuida. Se analizará una solución reciente a este problema, el paradigma teórico *Thinking like a vertex*, y se abordará la potencia del análisis temporal de grafos.

A continuación, se comentarán algunas tecnologías que permiten llevar a cabo este tipo de procesado, poniendo especial énfasis en Raphtory, que es la solución que se aborda en este TFM. Se analizará la herramienta, sus partes, y su funcionamiento, y se desarrollarán diferentes formas de despliegue, desde la opción monolítica, hasta la basada en servicios alojados en contenedores Docker orquestados con Docker Compose.

Finalmente, se implementará una solución BDaaS en la nube, con cierto grado de automatización orientado a DevOps, utilizando para ello Pods de Kubernetes desplegados en Google Cloud, sobre GKE, y haciendo uso del almacenamiento basado en objetos proporcionado por GCS.

De este modo, se plantea a un potencial usuario una oferta de analítica como servicio que le permita sortear las barreras de entrada para este tipo de actividades, tanto a nivel de dificultad, como de carga de procesamiento.

## Estado del arte

Como ya se ha comentado en la Introducción, el procesado de grafos, especialmente cuando estos son masivos, supone un reto computacional de elevada complejidad. Es por ello que, en los últimos tiempos, se han desarrollado nuevos paradigmas teóricos, y tecnologías punteras, orientadas a la resolución de este problema, y que discutimos a continuación.

### Pregel y Thinking like a vertex

En el año 2010, un grupo de expertos de Google lanzó un sistema de procesado de grafos a gran escala [4] al que bautizaron como Pregel, en honor al río de los puentes de Königsberg ya mencionado.

Dichos expertos enuncian la dificultad que supone el procesado de grafos, debido a la pobre localidad de los accesos a memoria, el escaso trabajo que se realiza por vértice, y la variación del grado de paralelismo en el curso de la ejecución, problemas que, además, se acentúan cuando se intenta llevar a cabo un procesado distribuido. Así, a la hora de implementar un algoritmo para esta actividad, se consideran las siguientes posibilidades, todas ellas con importantes desventajas [4]:

* Crear una infraestructura distribuida ad hoc, lo que implica un esfuerzo considerable que habría que repetir con cada algoritmo y grafo.
* Utilizar una plataforma ya existente, como Map-Reduce, que, por no estar optimizada para el procesado de grafos, proporciona un rendimiento relativamente escaso.
* Usar algoritmos que se ejecutan en un solo equipo, lo que deriva en problemas de escalabilidad.
* Utilizar sistemas que sí están orientados al procesado en paralelo de grafos, pero que, por lo menos hasta el momento, presentan complicaciones con las características de los sistemas distribuidos, como la tolerancia a fallos.

De este modo, los expertos llegan a la conclusión de que los sistemas actuales no cumplen con los requisitos de procesado de grafos masivos, por lo que inventan el sistema Pregel, el cual, a grandes rasgos, consiste en realizar una serie de computaciones llamadas “súper pasos”.

Durante un súper paso, se invoca a una función de procesado definida por el usuario para CADA VÉRTICE, conceptualmente, en paralelo. Dicha función de procesado especifica el comportamiento de un único vértice en un único súper paso, lee los mensajes enviados al vértice en el súper paso previo, y puede enviar mensajes que serán recibidos por otros vértices en el súper paso inmediatamente posterior.

Debido a que en este mecanismo todo el procesado se lleva a cabo para los vértices, y se realiza de manera independiente para cada uno de ellos, se habla de que sigue un paradigma llamado “*Thinking like a vertex*”. Además, es idóneo para la computación distribuida, ya que no es necesario especificar ningún mecanismo para detectar el orden de ejecución dentro de un súper-paso, y todas las comunicaciones se realizan de un súper paso S al súper paso S+1.

Así, un flujo de ejecución típico de un sistema Pregel consiste en una fase de entrada, en la que se inicializa el grafo, seguido por una secuencia de súper pasos, y, finalmente, la generación de una salida.

En el súper paso inicial, cada vértice empieza en el estado “activo”, se realiza su procesado, y se envían los mensajes correspondientes a otros vértices. Una vez llega el final del súper paso, el vértice vota qué es lo que va a hacer a continuación: si tiene más operaciones que hacer, votará por mantenerse activo, y seguirá ejecutando en el siguiente súper paso. Sin embargo, si ya ha terminado su procesado, vota por parase, y pasa a estar en el estado “inactivo”, del cual solo saldrá si más adelante recibe un mensaje de otro vértice.

Se considera que el algoritmo ha convergido cuando todos los vértices se encuentran en el estado inactivo, momento en el cual el sistema pasa a generar la salida (aunque es posible definir un número máximo de súper pasos, de manera que la ejecución no se alargue excesivamente).

En Pregel, los mensajes intercambiados por los vértices constan de un valor, y un destinatario. Se garantiza su entrega, y que no van a llegar duplicados, pero no el orden, ya que, como se ha comentado, no es necesario. Además, prevé la posibilidad de definir Combinadores que aúnen varios mensajes remitidos a un mismo destinatario en uno solo, Agregadores que recopilan información de procesado y son capaces de generar estadísticos, e incluso la posibilidad de que, debido a dichos mensajes, mute la topología del grafo.

Este modelo de computación es en el que basa Raphtory, objeto de este TFM, pues, si consultamos su código fuente [5], podemos observar cómo sigue el paradigma de “*Thinking like a vertex*”, implementando el concepto de súper paso, la máquina de estados de los vértices, o las condiciones de convergencia para los algoritmos.

En cuanto a la implementación de Pregel en un sistema distribuido [4], que es el problema que nos ocupa, se define una arquitectura básica en la que el grafo en cuestión se divide en varias particiones, cada una de las cuales está constituida por un conjunto de vértices y sus aristas salientes, que se distribuyen entre las máquinas de un clúster en configuración de maestro (ocupado de la coordinación) y esclavos (que se encargan del procesado). En cuanto a las operaciones dentro del sistema distribuido, se definen las siguientes fases [4]:

1. Se lanza el maestro y los esclavos, los cuales se registran enviando mensajes al maestro a través del sistema de gestión del clúster.
2. El maestro determina la cantidad de particiones del grafo, y las reparte equitativamente entre los esclavos.
3. El maestro recibe los datos de entrada, y se reparte entre los esclavos mediante mensajes. Una vez se ha cargado toda la entrada, los vértices de todas las particiones conmutan al estado activo.
4. Se ejecutan los súper pasos, llevándose a cabo el procesado, y el envío de mensajes, de forma asíncrona. Además, al término de estos, cada esclavo le comunica al maestro cuántos de sus vértices siguen activos. Mientras haya uno o más, el proceso se repite.
5. Cuando finaliza la computación de todo el algoritmo, cada esclavo guarda su porción de los datos procesados, a partir de los cuales, posteriormente, se generará la salida.

Además, siguiendo con la teoría de sistemas distribuidos, Pregel también define mecanismos de tolerancia a fallos, mediante un sistema de “checkpointing”, que le permite mantener el estado del grafo en el tiempo, y mensajes de ping para detectar si ha caído algún esclavo, y es necesario repartir su procesado entre los demás.

Cuando se proceda al análisis en detalle de Raphtory, se observará que, en líneas generales, esta es la arquitectura seguida, con un maestro o líder encargado de la coordinación, dividiendo el grafo en particiones, y con una serie de esclavos o trabajadores que se ocupan de las diferentes tareas de procesado.

Finalmente, es destacable notar que este sistema ha sido probado en diferentes problemas de grafos, como, por ejemplo, la paginación en el buscador de Google (el algoritmo Page Rank, que relaciona webs entre sí en función del número de enlaces entre ellas), o la búsqueda del camino óptimo en una red, obteniéndose resultados satisfactorios [4]:

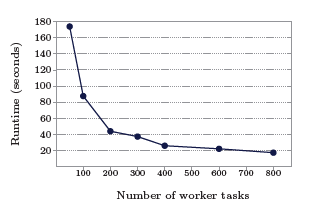


Ilustración 1. Tiempo de ejecución frente a número de nodos trabajadores en experimento con Pregel.

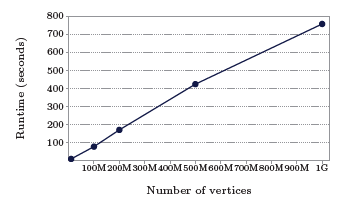


Ilustración 2. Tiempo de ejecución frente a número de vértices en experimento con Pregel.

Dichos resultados experimentales muestran un muy buen comportamiento de los tiempos de procesado al incrementar el número de nodos trabajadores, o el número de vértices del grafo, al menos similar al que podría obtenerse con otras soluciones más consolidadas, pero con una implementación más sencilla, y, sobre todo, una mayor escalabilidad.

### Análisis temporal de grafos

Como se ha comentado anteriormente, los grafos constituyen una herramienta clave en el mundo actual. Así, otra tecnología innovadora que se ha desarrollado en el ámbito del procesado de grafos es el análisis temporal, que permite estudiar cómo estos evolucionan a lo largo del tiempo, añadiendo una nueva dimensión respecto a la información que puede extraerse con su procesado.

De este modo, Raphtory se ha diseñado con la capacidad de no solo conseguir el procesado distribuido de grafos, sino también llevar a cabo análisis temporales sobre los mismos [6], tal y como se mostrará más adelante.

El análisis temporal de grafos también introduce una complejidad adicional a los problemas ya comentados, pues se añade la necesidad de mantener en memoria los estados por los que va pasando el grafo a lo largo del tiempo. Raphtory implementa una solución a este problema, especialmente enfocada a la escalabilidad, y que se desarrollará con mayor detalle en secciones posteriores.

Otras tecnologías que también abordan la dificultad de los análisis temporales son Kineograph [7], ImmortalGraph [8], y Chronograph [9]. La primera de ellas mantiene el grafo en memoria, y va ingiriendo actualizaciones que se graban en una tabla, lo que permite generar snapshots del grafo, aunque es extensivo en memoria [7]. Por su parte, ImmortalGraph trata de realizar un almacenado eficiente del grafo, y mantener un log de cambios que permite hacer consultas sobre cualquier instante de tiempo, aunque está más bien planteado para repositorios estáticos de datos [8]. Finalmente, Chronograph trata de combinar ambos sistemas, pero exige regenerar los snapshots a partir de los logs de actualizaciones para cualquier consulta [9].

En todo caso, el modelo general para el análisis temporal de grafos se basa en la comparación de snapshots, aunque, en ocasiones, dichos modelos se complican bastante cuando se tienen en cuenta eventos como el borrado y la adición de aristas. Como ya se ha comentado anteriormente, en el análisis detallado de la herramienta Raphtory se desarrollará con más detalle el modelo de análisis temporal que implementa.

### Programación en Scala

Actualmente, existe un gran número de lenguajes de programación, que siguen diversos paradigmas, como, por ejemplo, la programación orientada a objetos, o la procedimental.

De todos ellos, Raphtory ha optado por Scala [10], un lenguaje de propósito general, basado en Java, que combina características tanto de la OOP (Object Oriented Programming), como de la programación funcional.

Así, Scala presenta numerosas ventajas, como, por ejemplo, la relativa sencillez de su aprendizaje, la seguridad que otorga el tipado estático fuerte, la verbosidad reducida gracias a la inferencia de tipos, la transparencia referencial, o un modelo de concurrencia basado en actores que ofrece múltiples posibilidades [11]. Además, Scala se ejecuta sobre la JVM (Java Virtual Machine), por lo que es totalmente compatible con Java, pudiendo así aprovechar un gran número de librerías ya desarrolladas, y ejecutando su código en una plataforma altamente optimizada.

Por todo ello, Scala supone una alternativa interesante para llevar a cabo un proyecto de esta envergadura, donde la orientación a objetos encaja especialmente bien con el procesado distribuido de Raphtory dividido en diferentes componentes. Asimismo, no solo el proyecto de Raphtory, sino también el código desarrollado específicamente en este TFM, está escrito en Scala.

### Computación en la nube y herramientas DevOps

El procesado distribuido de grafos presenta una barrera de entrada especialmente importante en cuanto a los recursos necesarios, sobre todo cuando se trata de grafos masivos. Como se ha comentado, puede darse el caso de tener cierta cantidad de máquinas trabajando en paralelo, cada una de ellas procesando millones de vértices, con algún tipo de mecanismo de coordinación. También hay que tener en cuenta el almacenamiento necesario para estas cantidades masivas de datos.

De este modo, para cumplir el objetivo fundamental del TFM, es decir, el despliegue de una solución BDaaS para que un usuario cualquiera pueda acceder a este tipo de servicios, despreocupándose de estos problemas, se hace necesario utilizar tecnologías de virtualización, computación, y almacenamiento en la Nube.

El NIST (National Institute of Standards and Technology) define la computación en la Nube [12] como aquella que habilita un acceso por red ubicuo, conveniente, y bajo demanda, a un conjunto de recursos de computación compartidos y configurables (redes, servidores, almacenamiento, aplicaciones, y servicios) que pueden ser rápidamente aprovisionados y liberados con un esfuerzo de gestión mínimo por parte del proveedor del servicio.

Así, la Nube se caracteriza por proporcionar servicios bajo demanda sin interacción humana con el proveedor, acceso a través de la red mediante múltiples dispositivos, *multi-tenancy* (ofrece recursos comunes que pueden dar servicio a numerosos usuarios simultáneamente), elasticidad (rápida adaptación a la demanda), y un servicio medido, con un uso de recursos monitorizado, controlado e informado, permitiendo incluso el pago por uso [13].

De este modo, la computación en la nube proporciona numerosas ventajas a los usuarios, como el ahorro económico, la eficiencia en la utilización de los recursos, y la rapidez y simplicidad en su acceso a los mismos [13].

Por tanto, mediante las tecnologías de la Nube, se puede implementar un servicio BDaaS para el procesado de grafos que sea desplegable de forma rápida y ligera, accesible a través de Internet para cualquier persona, y a un coste mucho más reducido que en local, tal y como se hace en este TFM a través de Google Cloud, permitiendo superar las barreras de entrada que se han comentado al principio de la sección.

Asimismo, en aras de agilizar y facilitar el despliegue de este servicio BDaaS, también se ha hecho uso de las modernas tecnologías DevOps, que permiten automatizar todos los pasos de la construcción del software, desde la integración, hasta la administración [14].

Es por ello por lo que en este TFM se ha utilizado Docker, Docker Compose, y Kubernetes, con sus correspondientes scripts para desplegar y ejecutar el SW ya empaquetado mediante Pods; Helm para automatizar parte de la configuración y los despliegues; y GitHub Actions para integrar el proceso, y lanzar el servicio BDaaS de forma rápida y cómoda.

Al final, gracias a estas tecnologías, se ofrece una analítica BDaaS al usuario que le permite centrarse en su información y sus algoritmos de procesado, pudiendo despreocuparse tanto de los recursos necesarios y su coste, como de la complejidad de la configuración del despliegue, lo que lo convierte en un servicio sencillo, rápido, y útil.

### almacenamiento basado en objetos

En el procesado de grafos, donde pueden estar involucradas grandes cantidades de información, el modelo de almacenamiento puede marcar una diferencia notable.

Tradicionalmente, lo más utilizado ha sido el almacenamiento basado en archivos, en el que los datos se guardan como un elemento único de información en un directorio, siguiendo una jerarquía.

Con el auge de la virtualización, otro modelo que también ha cobrado importancia es el almacenamiento basado en bloques, en el que un archivo se divide en bloques de datos, que se guardan de forma independiente, con un mayor nivel de desacoplo.

Sin embargo, con la aparición de las tecnologías de la Nube, el modelo que se está imponiendo, especialmente en este contexto, es el almacenamiento basado en objetos, en el cual los datos a almacenar se consideran como objetos totalmente independientes, con sus propios metadatos, y con un identificador único, con lo que se prescinde de cualquier agrupación jerárquica.

Así, gracias a estas características, el almacenamiento basado en objetos presenta una mayor escalabilidad, eficiencia (pues no se forman cuellos de botella por la complejidad de los sistemas con directorios), y disponibilidad, con métodos que garantizan la coherencia y la réplicación de los datos [15].

Como se comentará más adelante, debido a que deseamos desplegar nuestro servicio en la Nube, se hará uso de esta tecnología de almacenamiento basado en objetos, en concreto, de la solución proporcionada por Google.

### Amazon EMR (Elastic Map-Reduce)

Para finalizar este estado del arte, se va a comentar brevemente una solución similar a Raphtory, pero implementada por Amazon.

Se trata de Amazon Elastic Map-Reduce, consistente en una plataforma de Big Data en la Nube que permite ejecutar tareas de procesamiento de datos distribuidos a gran escala, incluyendo también consultas SQL (Structured Query Language), y aplicaciones de Machine Learning con frameworks de código abierto, como Apache Spark [16].

Así, de forma similar a Raphtory, Amazon EMR presenta el siguiente flujo de trabajo:

1. Desarrollo de un algoritmo o aplicación propia de procesamiento de datos, utilizando diferentes lenguajes, desde Java a Python.
2. Carga de los datos a procesar, y del algoritmo, en el almacenamiento basado en objetos de Amazon, Amazon S3.
3. Configuración y lanzamiento de un clúster en Amazon EC2, configurando la cantidad de instancias y su tipo, así como la ubicación deseada para la salida.
4. Monitorización del funcionamiento del clúster, en Amazon CloudWatch.
5. Obtención del resultado desde Amazon S3.

De este modo, Amazon provee también de un servicio de tipo BDaaS en la nube, similar a lo que se ha pretendido implementar con Raphtory en este Trabajo Fin de Máster, pero, obviamente, mucho más automatizado y depurado, aunque sin análisis temporal de grafos.

En cualquier caso, la existencia de soluciones como Amazon EMR muestra que el ámbito del procesamiento distribuido de grafos ofertado como un BDaaS es un campo de interés, en el que se pueden esperar más investigaciones y progresos en los próximos años.

## Análisis detallado de Raphtory

Como se ha comentado en secciones anteriores, el objetivo principal de este TFM es el despliegue de una solución de analítica de grafos como servicio. Así, el núcleo de dicho despliegue es Raphtory, una herramienta de procesado de grafos y análisis temporal, capaz de mantener grafos temporales en un conjunto de particiones distribuidas, ingiriendo y procesando actualizaciones en paralelo en casi tiempo real, y desarrollada en Scala por la empresa inglesa Pometry [6] [17].

Por tanto, en este apartado llevamos a cabo un análisis detallado de la herramienta, en aras de comprender su funcionamiento, especialmente en lo que se refiere a las partes que lo componen, y al elemento diferenciador de los análisis temporales, de manera que podamos diseñar e implementar las diferentes modalidades de despliegue que se desarrollarán más adelante.

### Componentes de Raphtory y funcionamiento

La arquitectura y funcionamiento de Raphtory se basan en un modelo de actores, que carecen de estado compartido, y se comunican entre sí mediante mensajes que indican la siguiente función a realizar [6].

En la Ilustración 3 podemos ver a estos actores, y como interactúan entre sí:

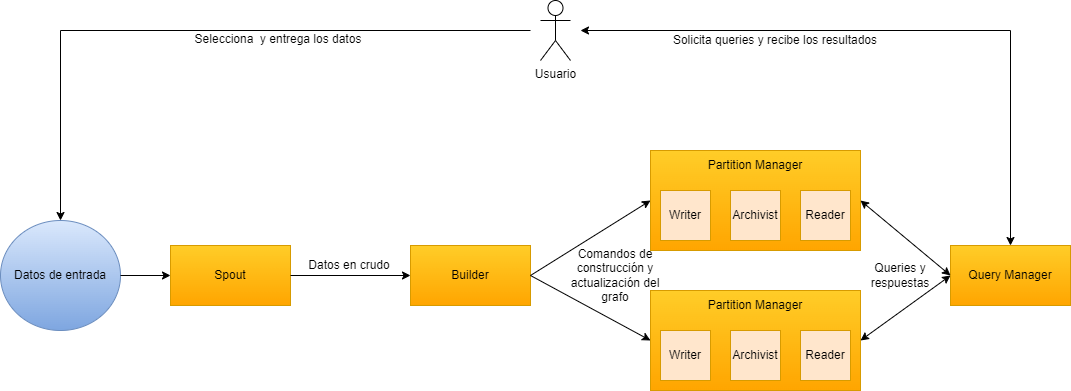


Ilustración . Diagrama que representa los componentes de Raphtory y sus interacciones.

A grandes rasgos, se observa que el usuario ha de seleccionar y preparar un conjunto de datos, que compondrán el grafo a procesar. Dichos datos son ingeridos por Raphtory mediante el Spout (caño), y se envían al Builder o constructor. Este componente convierte los datos en crudo en comandos de actualización que permiten a los gestores de particiones, o Partition Managers, almacenar la partición correspondiente, valga la redundancia, y mutarla en caso de que variaran los datos ingeridos por el Spout. Finalmente, el gestor de consultas, es decir, el Query Manager, recibe las solicitudes de análisis por parte del usuario, que serán ejecutadas en el gestor de particiones, y le enviará los resultados obtenidos.

A continuación, detallamos con algo más de profundidad cada uno de estos componentes.

#### Spout

El Spout simplemente es el componente que se encarga de ingerir los datos de entrada; es el origen de los eventos en Raphtory. Dichos datos pueden ser de diferentes tipos, como, por ejemplo, bases de datos, repositorios de ficheros, o colas de mensajes.

Así, el Spout realiza la conexión inicial requerida para acceder a los datos de estas fuentes, consumiendo los eventos en forma de tuplas, y distribuyéndolos hacia el Builder (en caso de que existan varios, se envían con un balanceo de carga de tipo Round-Robin). También puede haber más de un Spout, en cuyo caso actúan de forma totalmente desacoplada, ingiriendo los datos en paralelo de varias fuentes.

#### Builder

El constructor se encarga de recibir los datos en crudo procedentes del Spout, y convertirlos en operaciones de actualización del grafo, que son enviadas al gestor de particiones, de manera que este pueda crear o actualizar las entidades (vértices, aristas) que componen la partición asignada.

Dichas operaciones tienen una marca temporal (procedente de los datos, o del propio reloj interno) que permite al gestor de particiones conocer el orden de las entidades afectadas, sin necesidad de una sincronización temporal, lo que será útil posteriormente para los análisis temporales. Es decir, aunque los mensajes con operaciones lleguen desordenados, y puedan sugerirse situaciones contradictorias, gracias a las marcas temporales, las actualizaciones se ordenan correctamente, y el grafo mantiene la coherencia en todo momento.

En cuanto al envío de mensajes, Raphtory utiliza un algoritmo de particionamiento global para decidir dónde se almacena cada vértice. En concreto, es un tabla hash, que no requiere estado, y posee buena escalabilidad y propiedades de balanceo. Además, es una solución de compromiso, puesto que, en la mayoría de ocasiones, la estrategia de reparto óptima depende del algoritmo que se vaya a utilizar en el procesado.

Asimismo, los Builders utilizan un protocolo de “fire and forget” [6] para las actualizaciones que envían, de manera que puedan carecer de estado. Básicamente, este consiste en la definición de una ventana temporal, dentro de la cual todos los vértices y aristas del grafo han de haberse creado o actualizado. Si no es el caso, se planifica la eliminación de la entidad. Esta característica puede ser muy útil en el análisis de redes sociales, ya que, seleccionando una ventana temporal reducida, se puede extraer rápidamente a los usuarios “pequeños” que más crecen en popularidad en un breve intervalo de tiempo.

#### Partition manager

Los gestores de particiones son el núcleo de Raphtory, pues cada uno de ellos se encarga de almacenar una partición del grafo completo. Cada una de las particiones, de forma similar al sistema Pregel [4], contiene un conjunto único de vértices y sus aristas salientes, cada uno de ellos con su propia historia estructural.

Así, el gestor de particiones es responsable de mantener al día dichas historias, ejecutar las queries de análisis que le llegan, y realizar backups incrementales de las entidades. Para ello, se sirve de sus tres subcomponentes, es decir, el escritor (maneja las actualizaciones del estado del grafo, y las inserta en el historial de entidades afectadas), el lector (gestiona las peticiones de análisis, las ejecuta con el algoritmo indicado por el usuario, y devuelve los resultados), y el archivista (persiste los nuevos datos en almacenamiento permanente, y archiva las entidades “antiguas” para aliviar la carga de memoria).

Hay que tener en cuenta que los vértices y aristas contenidos en el gestor de particiones se almacenan siguiendo un patrón singleton, lo que garantiza un entrelazado seguro de las operaciones, pues los tres subcomponentes llevan a cabo accesos concurrentes.

En cuanto a la gestión de las entidades que componen un grafo, el Partition Manager almacena los vértices de forma completa, y maneja las aristas a partir de su origen, como plantea Pregel. Si el vértice destino también se encuentra en la partición, se dice que la arista es “local”; si, por el contrario, está localizado en una partición distinta, hablamos de aristas “divididas”, en cuyo caso el gestor de particiones que posee el vértice destino crea una arista “fantasma”, para que ambos puedan acceder a su estado. Estas aristas divididas se almacenan del mismo modo que las locales, pero añadiendo la ubicación de la copia fantasma, de manera que si ocurriera un cambio en ella, se podría enviar también a la copia para que se mantengan sincronizadas.

A continuación, detallamos el funcionamiento de los subcomponentes del gestor de particiones:

##### Writer o Escritor

El escritor, como ya hemos comentado, se ocupa de todas las actualizaciones que se producen en el estado de la partición almacenada, tanto de las operaciones recibidas por parte del Builder, como de los mensajes de sincronización de otras particiones (el problema ya descrito de las aristas fantasma). Para mitigar los picos que puedan producirse en el flujo entrante de actualizaciones, estas se van guardando en una cola ordenada hasta que sean correctamente procesadas. Dichas actualizaciones pueden ser de dos tipos:

* Adición/Modificación de entidades: cuando se añade un vértice, si no existe ningún objeto con el identificador con el que se está instanciando, se crea uno nuevo, comenzando así su historial, y su mapa de propiedades. Si, por el contrario, el objeto ya existía, se inserta un estado “Creado” en el historial, para evitar confusión si llegara con retraso un comando de borrado. El caso de las aristas es similar, con la diferencia de que, cuando se crea una, también se generan órdenes de creación de sus vértices origen y destino, evitando en cualquier caso que la arista quede colgando. Si fuera una arista dividida, el Partition Manager ordena al gestor que contiene el vértice destino que cree dicho vértice, junto con una arista fantasma.
* Eliminar entidades: de forma similar al caso anterior, no se comprueba si la entidad ya había sido borrada, pues podría llegar con retraso un comando de creación entre las dos eliminaciones. Si quisiéramos deshacernos de una arista dividida, también se envían las órdenes al otro gestor de particiones. Por supuesto, el borrado de un vértice implica la eliminación de todas sus aristas asociadas.

En la Ilustración 4, extraída directamente de [6], se puede observar un flujo de estas operaciones de adición y borrado, donde verde significa presente, y rojo, eliminado. También se han omitido las propiedades de los vértices tres y cuatro por simplicidad:

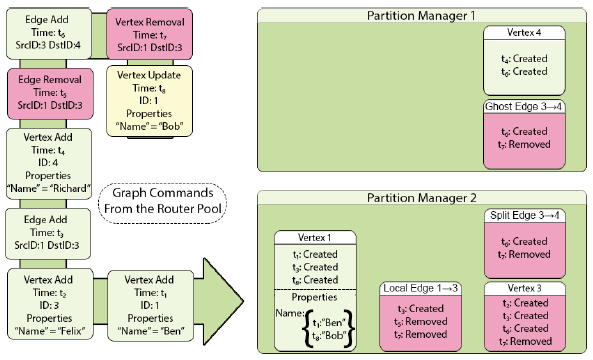


Ilustración . Diagrama de flujo de la adición y borrado de entidades.

##### Archivist o Archivista

El archivista se encarga de reducir la carga de memoria en el sistema mediante la ejecución de comprobaciones en las entidades almacenadas, comprimiendo los historiales, y descargando entidades inactivas en almacenamiento secundario, lo que permite tanto insertar nuevas actualizaciones en el grafo, como recuperar el historial para lanzar análisis temporales.

Como se puede deducir de las explicaciones previas, la huella de memoria que dejan las operaciones sobre entidades es considerable, pero, por cómo funciona el escritor, la historia es también maleable. Por ejemplo, supongamos que tenemos una historia asociada a un cierto vértice, donde, en los diferentes instantes t, han llegado los siguientes eventos: H = {<t5, creado>, <t4, creado>, <t2, creado>, <t1, creado>}. Supongamos ahora que llega un evento de borrado en el instante t3. Dadas las circunstancias, el archivista podría comprimir la historia de este modo: H = {<t4, creado>, <t3, borrado>, <t1, creado>}. Sustancialmente, es exactamente la misma información, pero se ha reducido la memoria consumida en un 40 %. Este tipo de tareas en segundo plano son las que realiza el archivista de un gestor de particiones.

Como se ha comentado anteriormente, una vez que la historia ha sido comprimida, este subcomponente también se encarga de llevarla al almacenamiento secundario, lo que además protege al sistema de pérdida de información en caso de que se produzca algún error. Este proceso se realiza en base a unos umbrales internos, que intentan alcanzar un equilibrio entre la eficiencia, y la cantidad de datos accesible en almacenamiento primario.

##### Reader o Lector

Finalmente, los lectores pueden considerarse como el motor de procesado de Raphtory, ya que se ocupan de ejecutar las funciones definidas por el usuario sobre las entidades que posee su partición.

Estos lectores no mantienen un estado, y el procesado que realizan esta basado en un modelo centrado en torno a los vértices, siguiendo el sistema Pregel y el paradigma “Thinking like a vertex” [4]. Además, cuando se completa uno de los súper pasos, se informa al Query Manager de ello, y se devuelven unos resultados parciales, en aras de que puedan insertarse nuevas entidades a pesar de que se esté ejecutando un análisis.

Para poder acceder a las entidades dentro de una partición, se utiliza un Graph Retrieval Proxy, que enmascara la dificultad del proceso, y muestra el último estado en el que se encontraba la entidad deseada. También tiene en cuenta si dichas entidades se encuentran en almacenamiento primario, o si ya habían sido archivadas por el archivista.

#### Query manager

Finalmente, el último componente de Raphtory es el gestor de consultas, que, fundamentalmente, se encarga de permitir al usuario interactuar con el grafo, ejecutando análisis, u obteniendo el estado de dicho grafo en cierta ventana temporal.

Así, se ocupa de determinar las funciones que ha de ejecutar el lector, y las condiciones de finalización, estructurándose en tres fases:

* Setup: se difunde una petición de análisis a todos los lectores, y se realizan las inicializaciones pertinentes.
* Análisis: una vez todos los lectores han mandado su ACK, se van ejecutando los súper pasos con la función definida por el usuario.
* Finalización: cuando todos los lectores hayan acabado un súper paso, si el Query Manager decide que se han alcanzado las condiciones de terminación, se da por terminada la ejecución del análisis, y se devuelven los resultados. En caso contrario, se continúa con el siguiente súper paso.

Además, el gestor de consultas da soporte a los análisis perpetuos, es decir, aquellos en los que se suprime la fase de finalización, generándose unos resultados progresivos, y aproximados, ya que pueden producirse mutaciones en el grafo de forma simultánea.

### Análisis temporal de grafos en raphtory

Tal y como se ha descrito en secciones previas, el análisis temporal constituye otra dimensión de procesado que permite extraer más información de un grafo. Esta característica constituye un elemento diferenciador de Raphtory, así que vamos a describirla más detalladamente.

Raphtory propone un modelo teórico de grafo temporal [6] que se puede condensar en el formalismo G(t) = <V(t), E(t) >, es decir, seguimos teniendo un grafo formado por dos conjuntos, uno de vértices, y otro de aristas, pero, además, estos poseen un estado que varía en el tiempo, siendo t0 la inicialización del grafo, y tn el instante del estado más reciente.

De este modo, el grafo temporal contiene todos los grafos observados entre los instantes t0 y tn. Además, cada vértice y cada arista posee una historia, que ya habíamos introducido en el apartado anterior, con la forma H = {<ti, creado>, <tj, eliminado>, …}, donde cada modificación constituye una nueva dupla, formada por una marca temporal, y una operación. Además, las entidades pueden tener propiedades, cada una de ellas con su propia historia de valores por los que va pasando.

Gracias a estos historiales, el análisis temporal de Raphtory puede proporcionar el estado de un dato en cualquier instante de tiempo, siendo capaz de recrear el grafo G(t) en ese instante, en lugar de limitarse a proporcionar snapshots en instantes predeterminados.

En este modelo de análisis temporal, se establecen tres posibles eventos que pueden acontecer a las entidades:

* Adición de entidades: cuando se añade un vértice, se comprueba si existe, y, en su caso, se crea una historia con una dupla de creación. Lo mismo ocurre para las aristas, en las que, además, se verifica que los vértices que une, existan.
* Eliminación de entidades: de forma similar a la adición, se comprueba que la entidad exista, y se encola un evento de borrado en su historia. En el caso de los vértices también se planifica la eliminación de sus aristas asociadas.
* Actualización de propiedades: cuando se crea una entidad, ya sea un vértice o una arista, también se añaden sus propiedades, si las tuvieran. Durante la vida del grafo, dichas propiedades van actualizándose, cambiando su valor, e, incluso, pueden aparecer propiedades nuevas. Como se ha comentado, dichas propiedades también tienen su propia historia, con el detalle de que, cuando se desee realizar un cambio sobre ella, es necesario verificar que la entidad a la que está asociada, exista.

En efecto, si nos atenemos a la sección previa Writer o Escritor, podemos observar que este es el modelo de grafo temporal que implementa Raphtory. En cualquier caso, hay que tener en cuenta la complejidad introducida por los análisis temporales, añadiendo, por ejemplo, los problemas de sincronización en sistemas distribuidos, con las posibles inconsistencias que pueden surgir.

Para finalizar este apartado, se comentan a continuación, con algo de código extraído directamente del proyecto, algunos ejemplos [18] de análisis temporales. Básicamente, una vez se ha creado un grafo temporal en Raphtory, los análisis que podemos llevar a cabo permiten filtrar en el tiempo, crear perspectivas, o definir ventanas sobre las que llevar a cabo trabajos de análisis.

#### Filtrado temporal

El filtrado temporal permite obtener los resultados del análisis ciñéndose a un intervalo de tiempo específico. Para ello, Raphtory proporciona las funciones:

* from(start): considera toda la historia del grafo tras “start”.
* *until(end)*: considera hasta “end”.
* *to(end)*: igual que *until*, pero se puede usar en conjunción con *from*.
* slice(start, end): equivalente a from(start), to(end).

Así, si, por ejemplo, solo nos interesara la historia de un grafo entre el 1 de enero de 2020 y el 1 de enero de 2021, podríamos ejecutar el siguiente código:

Texto

Descripción generada automáticamente

Ilustración . Código de un análisis temporal que realiza un filtrado.

#### Creación de perspectivas y definición de ventanas

Raphtory también permite ejecutar una secuencia de perspectivas en diferentes instantes de tiempo, a fin de descubrir los cambios que van sucediendo en el grafo. Para ello, también nos proporciona las siguientes funciones:

* depart(time, increment): indica el instante inicial, y cada cuanto tiempo ejecuta el análisis.
* climb(time, increment): indica el instante final, y cada cuanto tiempo hacia atrás se ejecuta el análisis.
* range(start, end, increment): marca un final, un principio, y cada cuanto tiempo se ejecuta el análisis entre ellos.

También se pueden definir ventanas, que establecen los intervalos temporales en los cuales se van a tener en cuenta los cambios que se producen en el grafo (hacia el pasado, hacia el futuro, o ambos). Es decir, cualquier cambio fuera de la ventana, será completamente ignorado.

Aunando estos conceptos, con el siguiente código, podríamos extraer los cambios que se producen en el grafo, empezando el 1 de enero de 2020, con intervalos de tiempo de 1 día, y teniendo en cuenta una ventana temporal de 1 semana en el pasado:

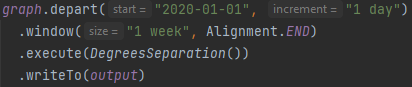


Ilustración . Código de un análisis temporal con perspectivas y ventanas.

En la Ilustración 7, extraída directamente de [18], se reflejan gráficamente las posibilidades del análisis temporal en Raphtory:

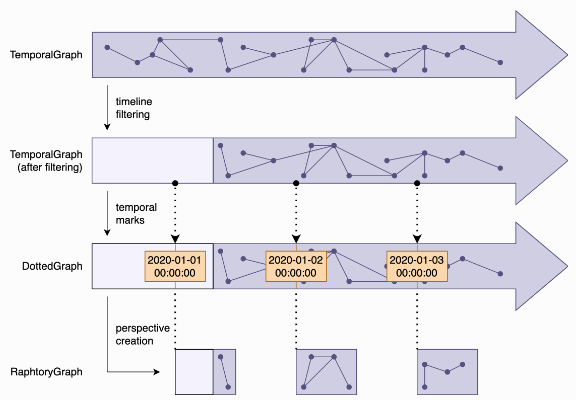


Ilustración . Posibilidades del análisis temporal en Raphtory.

## Despliegues básicos de raphtory: Single y PD

Habiendo explorado Raphtory con un mayor nivel de detalle, y estudiado las posibilidades de análisis que nos ofrece, nos podemos encaminar hacia el objetivo final del TFM, que es su despliegue en la Nube como un servicio BDaaS.

Para ello, en primer lugar, se reproducen y explican las dos formas básicas de ejecución de Raphtory que nos ofrece la empresa creadora, Pometry: la opción monolítica o “Single”, útil para observar el funcionamiento básico de la herramienta,, y la pseudo distribuida, que es la clave para los despliegues implementados después.

### Despliegue monolítico de raphtory

La forma más sencilla e inmediata de lanzar Raphtory es la opción Single. En este despliegue, solo es necesario seguir los siguientes pasos:

* Indicar un Spout de los disponibles, ya que no es lo mismo ingerir un fichero local que uno que se encuentra en la Nube, como se verá más adelante.
* Implementar un Builder: Raphtory proporciona un modelo de Builder, pero exige que lo implementemos, al menos sobrescribiendo el método parseTuple para saber cómo ha de interpretar los datos.
* Llamar al método load(spout, builder) del objeto Raphtory, para cargar los datos seleccionados, e inicializar los componentes, es decir, el Partition Manager, y el Query Manager.
* Definir una salida: de nuevo podemos elegir entre las opciones disponibles, ya que no es lo mismo volcar el resultado a un fichero en local que imprimirlo por pantalla.
* Indicar qué análisis queremos realizar sobre el grafo. Podemos utilizar alguno de los algoritmos de procesado que nos proporciona Raphtory, o uno de desarrollo propio.

Para ilustrar este despliegue, se ha preparado el siguiente ejemplo en Scala:

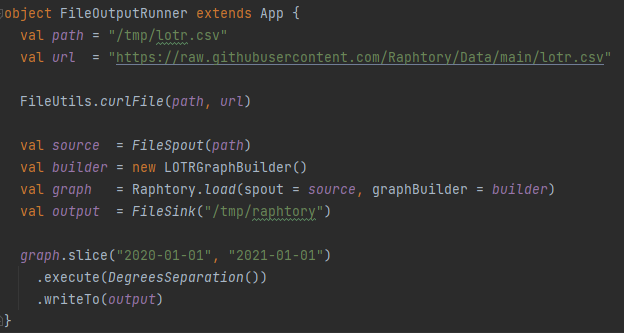


Ilustración . Ejemplo de despliegue monolítico.

Como se puede observar, este objeto ejecutable descarga un fichero de Internet, de tipo CSV, lo guarda en la carpeta tmp del equipo, selecciona un Spout de tipo FileSpout, indicando la ruta donde se encuentran los datos a procesar; un Builder, que se ha implementado con anterioridad; se llama al método load() para cargar los datos de entrada y crear el resto de componentes; se define la salida, en este caso, un volcado de resultados a un fichero también en tmp, y, finalmente, se especifica un análisis temporal, el mismo que se ha empleado de ejemplo en el apartado anterior.

En este despliegue particular, los datos de entrada, es decir, el fichero lotr.csv, consisten en tuplas de tres elementos, cada una de ellas con los nombres de dos personajes de la saga El Señor de los Anillos, y la frase del libro en la que aparecen juntos. Por ejemplo:

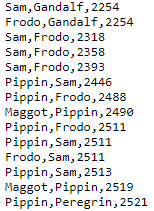


Ilustración . Fragmento de contenido de los datos de entrada.

En cuanto al Builder, en este ejemplo, se ha utilizado el siguiente código:

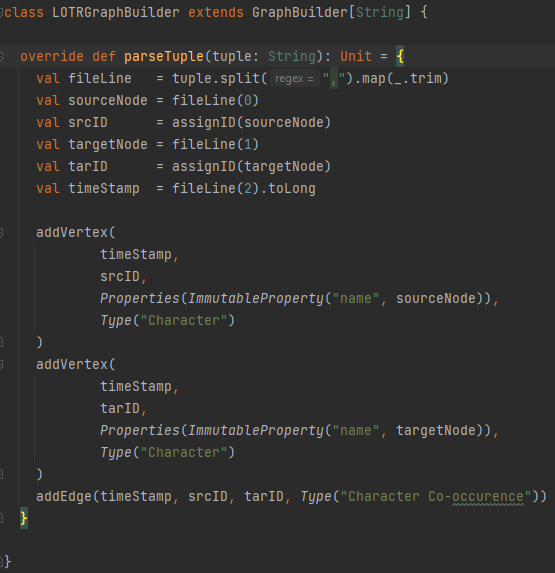


Ilustración . Builder utilizado en el ejemplo.

En este Builder, se empieza por convertir una línea de texto de lotr.csv en una tupla de tres elementos, del tipo (Personaje 1, Personaje 2, número de frase). Después, se asigna un ID a cada uno de los personajes, y se utiliza el número de frase como marca temporal. Finalmente, se crea un vértice para cada uno de los personajes, y una arista que los una.

Como se puede observar, los vértices se añaden sin comprobar si existían antes, tal y como se ha explicado en la sección Componentes de Raphtory y funcionamiento. También se asigna una propiedad a los vértices, en este caso, el propio nombre del personaje, aunque podría ser cualquier otra.

Finalmente, vemos que el algoritmo ejecutado en el análisis del grafo es “DegreesSeparation”, uno de los predefinidos por Raphtory, que simplemente calcula la distancia mínima entre el vértice de un personaje, y el resto. Se podría haber implementado un algoritmo de procesado propio, aunque este trabajo se saldría del ámbito del TFM actual. En cualquier caso, esto implicaría la implementación de al menos los siguientes métodos:

* step(): recibe una función que se aplica a cada vértice del grafo, y permite que estos cambien su estado y envíen mensajes a sus vecinos. Permite realizar inicializaciones para el algoritmo definitivo, de manera que los vértices se preparen con un estado por defecto.
* iterate(): es similar a step(), pero la función que recibe en este caso se va a ejecutar en los súper pasos hasta que se alcance alguna condición de terminación. Posee un flag llamado executeMessageOnly, que fuerza a que solo los vértices que han recibido mensajes se ejecuten, siguiendo así el sistema Pregel.
* select(): permite asociar un vértice a un objeto de tipo Row, en el que ir almacenando los resultados.
* filter(): se utiliza para reducir la cantidad de resultados almacenados.
* explode(): permite variar el formato de los resultados, por ejemplo, convirtiendo un array en objetos individuales.
* writeTo(): finalmente, se encarga de escribir los resultados en un formato de salida.

Así, Raphtory admite tres tipos de algoritmos, en función de la cantidad de súper pasos necesarios para alcanzar la convergencia:

* Algoritmos de cero pasos: aquellos que se pueden completar con el conocimiento interno de los vértices, sin enviar mensajes.
* Algoritmos de un paso: en esta clase de algoritmo, solo es necesario enviar mensajes una vez, con lo que se converge en único paso, tal y como indica el nombre.
* Algoritmos iterativos: aquellos que requieren un número indeterminado de pasos de mensajería.

Este ejemplo de despliegue monolítico es muy sencillo, pero ilustra de forma práctica el análisis de Raphtory descrito en los apartados anteriores, y permite observar el funcionamiento real de la herramienta. Los resultados de la ejecución se recogen en la sección Resultados, y todo el código interno de Raphtory, no solo las utilidades del ejemplo, está disponible en [5].

### Despliegue pseudo distribuido de Raphtory

La otra opción que nos ofrece Raphtory es el despliegue pseudo distribuido. En esta modalidad, en lugar de usar Raphtory.load(spout, builder), que se ocupa de instanciar y ejecutar todos los componentes, lo que hacemos es crear un objeto, al que he llamado Component, que hereda de la clase RaphtoryService, y recibe un argumento de tipo String con el nombre del componente que se quiera lanzar.

De este modo, si deseamos desplegar, por ejemplo, un Builder, crearemos un objeto Component pasándole el argumento “builder”, y así sucesivamente para todas las partes de Raphtory, pues, internamente, la clase antecesora, en función del String que hayamos elegido, define un componente u otro:

Texto

Descripción generada automáticamente

Ilustración . Código fundamental de la clase RaphtoryService.

Por su parte, en la Ilustración 12, tenemos el objeto Component, en el que también hay que definir un Spout y un Builder, para que Raphtory sepa cómo se han de ingerir e interpretar los datos:

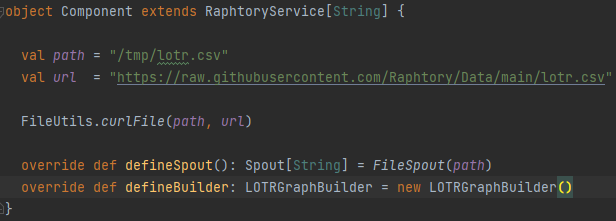


Ilustración . Código del objeto Component.

Si observamos la clase RaphtoryService, podemos apreciar que hay un nuevo componente que no había aparecido hasta ahora, el líder. Este simplemente se ocupa de gestionar las comunicaciones entre el resto de elementos, y permite que los componentes se descubran entre sí al realizar el despliegue.

Sin embargo, en versiones posteriores de Raphtory al momento en el que se desarrolló este ejemplo, el líder ha desaparecido, y sus funciones han sido asimiladas por el resto de elementos. Es por ello por lo que no se ha mencionado en la fase de análisis.

En cualquier caso, este despliegue recibe el nombre de pseudo distribuido porque, aunque es cierto que cada componente de Raphtory se lanza como un objeto que se ejecuta de manera independiente, al final, todos ellos corren en un mismo equipo, simplemente están escuchando y enviando mensajes en diferentes puertos del ordenador. La Ilustración 13 muestra esta situación:

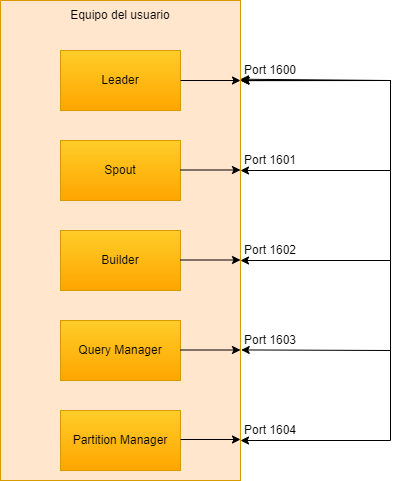


Ilustración . Diagrama del despliegue PD y sus comunicaciones.

Para las comunicaciones entre los componentes, Raphtory utiliza Akka [19], un framework basado en Scala para la construcción de aplicaciones distribuidas y concurrentes, que ejecuta sobre la JVM (Java Virtual Machine), y se establece sobre el modelo de actores de este lenguaje de programación. En el apartado de Resultados, se observarán los problemas que genera esta herramienta en las comunicaciones entre componentes, motivo por el cual, en versiones posteriores de Raphtory, fue reemplazada por Apache Pulsar, que se comentará más adelante.

De este modo, aunque no nos hallemos ante un despliegue verdaderamente distribuido, sí es cierto que esta modalidad establece la fundación para la división de Raphtory en componentes independientes, lo que será aprovechado a continuación para los despliegues verdaderamente distribuidos en Docker y GKE.

## Despliegue de Raphtory con Docker y Docker Compose

En aras de alcanzar nuestro objetivo de ofrecer Raphtory como un servicio de tipo BDaaS en la Nube, el siguiente paso consiste en la implementación de un despliegue totalmente distribuido, al menos en local, para, después, poder dar el salto hacia Google Cloud.

En el desarrollo de esta modalidad de despliegue, se ha optado por el uso de contenedores Docker [20], ya que ofrecen una alternativa rápida, ligera, y eficiente en recursos, para poder lanzar máquinas virtuales, con un gran catálogo de imágenes por defecto, que, además, son fácilmente configurables mediante un Dockerfile, de manera que reúnan los requisitos que necesitamos en la implementación de un componente de Raphtory.

Así, la idea general del despliegue en Docker consiste en la creación de un contenedor para cada uno de los elementos de Raphtory. Dichos contenedores toman como base una imagen de tipo “adoptopenjdk:11”, suministrada directamente a Docker por la comunidad AdoptOpenJDK [21], que contiene un JDK (Java Development Kit) de código abierto, que permitirá ejecutar las características de Raphtory. Por supuesto, tal y como se detallará en la sección de Resultados, donde se proporcionan los detalles del despliegue, estas imágenes se customizarán convenientemente para que puedan implementar un componente de Raphtory.

Una vez hemos desarrollado los contenedores, el siguiente paso consiste en desplegarlos y orquestarlos de forma automática, asegurando que pueden comunicarse correctamente entre ellos, a través de algún tipo de red que permita el paso de mensajes necesario para el funcionamiento de Raphtory.

Para obtener esta funcionalidad, se ha optado por Docker Compose [22], una herramienta que permite la definición y ejecución de aplicaciones Docker multi contenedor, configuradas con ficheros en formato YAML (YAML Ain’t Markup Language) [23]. Así, Docker Compose se va a encargar de organizar el despliegue de los componentes de Raphtory, exponer los puertos necesarios, y crear una red sencilla entre ellos, de manera que todos los contenedores puedan unirse a ella, y sean alcanzables y descubribles entre sí [22]. El envío de mensajes de unos componentes a otros seguirá utilizando el framework de Akka [19].

Como se trata de un despliegue en local con un número reducido de contenedores, cuyo objetivo es probar Raphtory de forma completamente distribuida, y servir de “puente” para llegar al despliegue del BDaaS en la nube, se ha preferido apostar por la inmediatez y sencillez de Docker Compose frente a otras posibilidades, como Docker Swarm [24], que también proporcionan escalabilidad, balanceo de carga, o redes más completas, a costa de una mayor complejidad.

En la Ilustración 14 se muestra como queda la estructura final del despliegue:

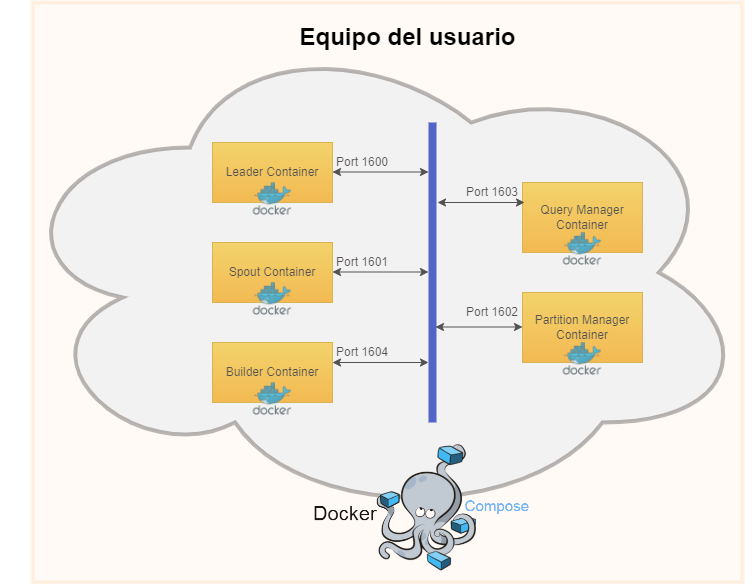


Ilustración . Despliegue en Docker y Docker Compose.

Los detalles concretos del despliegue, así como los resultados que se obtienen al ejecutarlo, se desarrollan en la sección de Resultados.

## Despliegue de Raphtory como BDaaS en la Nube

Finalmente, una vez que se ha logrado implementar un despliegue completamente distribuido de Raphtory utilizando Docker, estamos en disposición de dar el paso final, y llevar a cabo el despliegue de Raphtory en la Nube, de manera que podamos ofrecer a un usuario un servicio BDaaS que le permita sortear las barreras de entrada ya mencionadas en la obtención de la analítica como servicio.

Sin embargo, a la hora de lanzar Raphtory en la Nube, existen numerosas consideraciones a tener en cuenta, especialmente en lo relativo a las tecnologías a emplear para la implementación del despliegue.

Así, antes del desarrollo de esta parte del TFM, se decidió llevar a cabo un estudio sobre qué herramientas utilizar para ofrecer un servicio BDaaS lo más cómodo, funcional, y automático posible, y que se recoge a continuación.

### Estudio y evaluación de tecnologías para el despliegue en la nube

A la hora de realizar un despliegue de Raphtory en la Nube, se ha hecho necesario considerar qué herramientas se van a emplear en, por ejemplo, construcción y almacenamiento de imágenes, comunicaciones entre componentes, despliegue y orquestación de los mismos, automatización, o almacenamiento en la Nube.

#### Construcción y almacenamiento de imágenes: Docker y Docker Hub

En primer lugar, si deseamos implementar un despliegue automático de los componentes de Raphtory en la Nube, necesitamos una tecnología que permita construir las imágenes que estos van a emplear, y un lugar de almacenamiento para las mismas que sea accesible desde la Nube.

Respecto a la virtualización, tal y como se hizo en el caso anterior, y por los mismos motivos, se ha optado por Docker [20], pues su solución de contenedores es ligera, sencilla de manejar, y lo suficientemente maleable como para cumplir los requisitos de un despliegue automático en la Nube, frente a la mayor lentitud y consumo de recursos que obtendríamos si optáramos por la virtualización pesada.

En lo que concierne al almacenamiento de las imágenes, necesitamos algún tipo de repositorio que sea accesible para el despliegue en la Nube. En este caso, se plantearon dos opciones:

* Docker Hub [25]: este es el repositorio oficial de la empresa Docker para subir tus propias imágenes. Es una opción atractiva, puesto que, ya que se utiliza Docker en la construcción de las imágenes, es lógico usar su repositorio para almacenarlas. Además, está muy integrado con las herramientas de escritorio de Docker que se han utilizado, y, por lo menos en el momento de realizar este TFM, la mayoría de sus funciones siguen siendo gratuitas.
* Google Cloud Artifact Registry [26]: esta es la solución de Google para la administración de imágenes de contenedores y paquetes de lenguajes en la Nube. También supone una alternativa muy atractiva, por su integración directa en Google Cloud, y las facilidades que posee para integración y entrega continuas. Sin embargo, hay que tener en cuenta que no deja de ser un producto con un coste económico asociado.

Tras evaluar las herramientas, y probar el uso de cada una de ellas, determinamos que Docker Hub presenta un rendimiento muy inestable, pésimo en ocasiones (la subida de una imagen puede durar desde un minuto hasta, las más de las veces, diez), mientras que Artifact Registry posee un comportamiento mucho más eficiente.

Sin embargo, a la hora de desplegar Raphtory, nos dimos cuenta de que, para acceder a las imágenes de un repositorio externo, solo se soporta el uso de credenciales en forma de usuario y contraseña, que sí está integrado en Docker Hub, pero no en Artifact Registry, que utiliza un sistema basado en tokens. Por ende, a pesar de su inferior rendimiento, nos vemos obligados a utilizar Docker Hub.

#### Comunicaciones entre los componentes

Hasta el momento, el envío de los mensajes entre los componentes de Raphtory se había realizado mediante Akka [19], lo que era especialmente conveniente teniendo en cuenta que se trata de un framework de Scala, el lenguaje de programación en el que está escrito Raphtory.

Sin embargo, como ya se ha comentado, y se mostrará en la sección de Resultados, Akka origina algunos problemas con la mensajería entre componentes, que, si bien no impiden que al menos la herramienta funcione, sí que supone un empeoramiento del rendimiento.

Así, en este punto del trabajo, la empresa Pometry actualiza la herramienta Raphtory a la versión 0.5, la cual introduce numerosos cambios, entre ellos, la sustitución de Akka por Apache Pulsar [27], una plataforma distribuida en código abierto para mensajería de tipo Pub-Sub y streaming, orientada a soportar cargas de trabajo en tiempo real con millones de eventos.

Apache Pulsar presenta una estructura basada en un clúster compuesto de: brokers, que se encargan de comunicarse con Productores y Consumidores, y de realizar las transferencias entre ellos; “bookies”, los elementos para el almacenamiento y la persistencia de los datos; y Zookeeper, para la configuración y coordinación del clúster. La Ilustración 15, extraída directamente de [28], representa esta estructura:

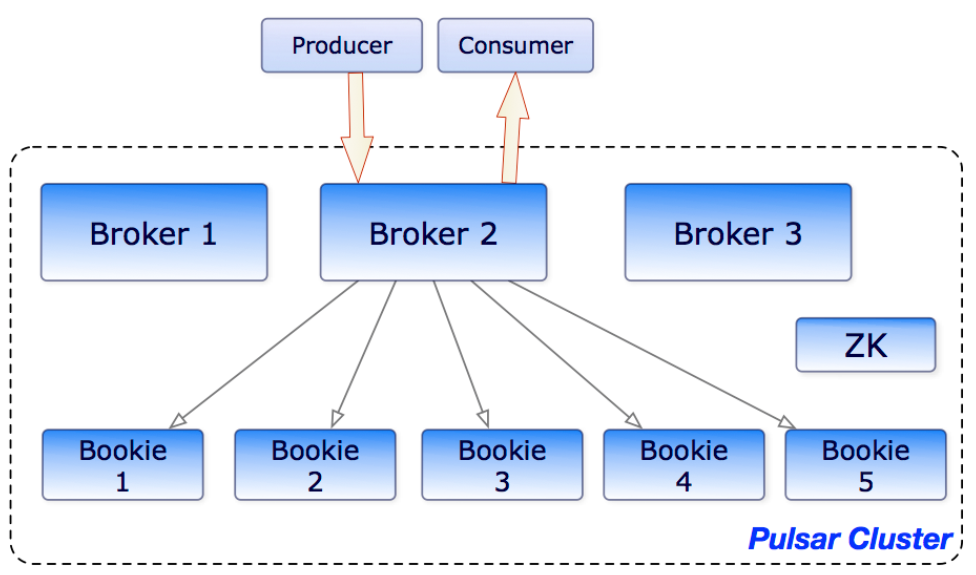


Ilustración . Diagrama de la estructura de un clúster de Apache Pulsar.

De este modo, Apache Pulsar, con su arquitectura distribuida, posee numerosas ventajas sobre Akka, como escalabilidad, consistencia, y disponibilidad, que mejoran el funcionamiento general de Raphtory, y permite eliminar los problemas ya referidos.

Así, aunque no se trate de una decisión de nuestro diseño, pues viene directamente impuesta desde Pometry, sí se ha considerado de valor estudiar mínimamente las características y ventajas de Apache Pulsar, en aras de comprender la sustitución de Akka. Además, esta herramienta va a constituir un elemento más del despliegue en la nube, como se comentará más adelante.

#### Tecnología base del despliegue en la Nube

A la hora de realizar el despliegue en la Nube, se realizó una fase de estudio para determinar qué tecnología era la más adecuada como plataforma base para realizar dicho despliegue.

Esta plataforma ha de contener cierto grado de automatización, permitir ejecutar los contenedores en un clúster de la Nube, facilitar la administración, orquestación, control y monitorización de los contenedores, y habilitar el acceso desde el exterior a los servicios desplegados.

En un principio se sopesó llevar directamente el despliegue previo a la nube, y usar Docker Compose. Sin embargo, se determinó que este despliegue podía considerarse como excesivamente básico, y con una escalabilidad demasiado reducida si se desea ofrecer un servicio BDaaS real con posibilidades de escalabilidad en el futuro.

Así, finalmente, se tomó la decisión de emplear una herramienta con propiedades de crecimiento horizontal, y un mayor grado de automatización, esto es, se optó por utilizar Kubernetes [29].

Al final, Kubernetes ofrece un entorno de administración basado en contenedores, orquestando la computación, las redes, y el almacenamiento con cierto grado de automatización, de manera que constituye una plataforma que, a pesar de su complejidad, es apropiada para el despliegue, escalado, y administración de nuestro servicio BDaaS.

Además, Kubernetes posee numerosas ventajas [29] beneficiosas para nuestro proyecto, como la creación ágil de contenedores encapsulados en Pods, integración continua, monitorización de la salud de los componentes, portabilidad entre nubes, y eficiencia en el uso de recursos.

Si a estos puntos sumamos el hecho de que Pometry introdujo en la versión 0.5 algunas facilidades en forma de clases de Scala para el despliegue con Kubernetes, y al elevado grado de integración de esta herramienta con Google Cloud (no es de extrañar, pues fue la propia Google quien la desarrolló), podemos concluir que Kubernetes es la tecnología idónea para la base del despliegue.

Finalmente, es notable resaltar que, para este estudio y evaluación de Kubernetes, se ha experimentado con Minikube [30], una distribución reducida de Kubernetes para trabajar en local, pero que permite hacer algunas pruebas sencillas y familiarizarse con la complejidad de K8s.

#### Computación y almacenamiento en la Nube

Habiendo decidido la tecnología base del despliegue, el siguiente paso natural es seleccionar una nube donde desplegar nuestro servicio BDaaS. Las opciones más inmediatas a considerar son Microsoft Azure, Amazon Web Services, y Google Cloud.

De estas tres alternativas, nos decantamos desde un primer momento por Google Cloud [31], ya que no solo proporciona un crédito gratuito para realizar pruebas, sino que también es la posibilidad que mayor grado de integración presenta con Kubernetes. Así, aunque no posea tantos servicios como las demás alternativas, es más económica, y cumple con los requisitos mínimos de seguridad, rendimiento, almacenamiento, seguridad, etc., exigibles a nuestro servicio.

Así pues, de cara a la computación, Google Cloud nos proporciona el servicio GKE (Google Kubernetes Engine) [32], que, de forma sencilla y automática, nos permite crear un clúster con un número configurable de nodos en el que lanzar nuestros despliegues de Kubernetes. Además, posee escalabilidad automática para pods y clústeres (lo que puede ser útil de cara a la evolución del servicio), redes bien protegidas, y gestión de la identidad, para poder conectar con el despliegue de forma sencilla, como se explicará más adelante.

Finalmente, en lo que respecta al almacenamiento de los datos, Google Cloud ofrece varias soluciones, como, por ejemplo, Persistent Disk [33], que proporciona almacenamiento basado en bloques, orientado a máquinas virtuales pesadas; o Filestore [34], que es almacenamiento basado en archivos.

Sin embargo, tal y como se ha comentado en el apartado Estado del arte, a fin de obtener almacenamiento en la nube, el modelo óptimo es el basado en objetos, que presenta buenas características de escalabilidad, eficiencia, y disponibilidad

Por ende, la solución por la que se ha optado de cara al almacenamiento es GCS (Google Cloud Storage) [35], que nos provee del susodicho modelo basado en objetos, almacenándolos en unas estructuras no jerarquizadas llamadas buckets, con un control de acceso con credenciales, y elevada consistencia e interoperabilidad.

De este modo, Google Cloud nos proporciona las características de computación en la Nube, integración con Kubernetes, y almacenamiento, que necesita nuestro proyecto.

#### Automatización del despliegue

En aras de ofrecer un servicio de tipo BDaaS en la Nube a un usuario potencial, otro punto a considerar, que impulsa el valor añadido del mismo, es la automatización del despliegue, de manera que se reduzca la complejidad todo lo posible a la hora de utilizar el servicio, y este sea accesible de forma cómoda y rápida.

Así, para realizar el despliegue automático de Apache Pulsar, el elemento encargado de las comunicaciones entre componentes, y que, como ya se ha comentado, constituye su propio despliegue, se va a utilizar un chart de Helm.

Helm [36] es una herramienta que permite gestionar software construido para Kubernetes, como un gestor de paquetes, mediante el concepto de chart, que es un fichero el cual, en formato YAML [23], empaqueta toda la configuración necesaria para crear una instancia del servicio en Kubernetes.

Gracias a Helm, con un sencillo script podemos instalar un chart de Apache Pulsar, configurarlo según nuestras necesidades, y lanzar un despliegue en la Nube de este servicio de manera rápida y sencilla. En apartados posteriores se explicará exactamente cómo se realiza este proceso.

Por último, también existe la posibilidad de realizar una automatización más general del servicio completo, más allá de lo que ya se está haciendo con scripts, Docker, Kubernetes, Helm, y las utilidades de Google Cloud.

En este punto se evaluaron varias posibilidades, como, por ejemplo, GitLab [37], una plataforma de DevOps que permite realizar desde la planificación de un proyecto hasta su despliegue y monitorización.

Sin embargo, la sencillez, y la falta de tiempo para investigar más a fondo esta herramienta, hizo que nos decantáramos finalmente por GitHub Actions [38], una funcionalidad de GitHub que permite, mediante ficheros YAML, indicar una secuencia de operaciones con el código de nuestro repositorio que se ejecutarán sobre una máquina aprovisionada por GitHub.

De este modo, si hemos subido a un repositorio todo el código necesario para la realización del despliegue, podemos crear una GitHub Action que vaya ejecutando las acciones necesarias para completar el lanzamiento del servicio BDaaS, de forma completamente automática, tan solo pulsando un botón en nuestro repositorio.

Además, las máquinas que nos proporciona GitHub para la ejecución de las acciones tienen un gran número de utilidades preinstaladas, como Helm, el SDK (Software Development Kit) de Google, Java, y Scala, lo que simplifica en gran medida las configuraciones necesarias.

En secciones siguientes, se desarrollará con más detalle cómo se han utilizado las GitHub Actions para automatizar parte del despliegue, creando un flujo de trabajo personalizado que integra conceptos de CI/CD (Continuous Integration / Continuous Delivery), potenciando el valor del BDaaS de cara a un futuro usuario.

### Despliegue de Raphtory en Google Cloud

Después de estudiar, evaluar, y decidir las tecnologías necesarias para poder desplegar Raphtory como un servicio de procesado distribuido de grafos en la Nube, describimos a continuación los aspectos fundamentales del mismo, aunque los detalles concretos de cómo se lleva a cabo, paso a paso, se recogen en el apartado de Resultados.

En primer lugar, para poder realizar el despliegue, necesitamos construir las imágenes que se montarán en los Pods de Kubernetes. Como se ha comentado, estas se construyen con Docker, incluyendo todo el código en Scala que queremos ejecutar, tanto de Raphtory, como del ejemplo demostrativo que vamos a utilizar. Finalmente, las imágenes quedan almacenadas en Docker Hub, donde pueden ser accedidas remotamente.

Paralelamente, se pueden configurar todas las características necesarias de la Nube de Google, utilizando para ello Google Cloud SDK [39], la herramienta oficial para, desde línea de comandos, poder acceder a todos los servicios de Google Cloud.

En concreto, es necesario crear un proyecto, un clúster de GKE, una cuenta de IAM asociada al proyecto con los permisos y credenciales necesarias, e instalar las funcionalidades de Kubernetes, para poder conectar con dicho clúster desde nuestro CLI (Command Line Interface). También se puede arrancar un bucket de GCS, y almacenar nuestros datos a procesar.

Con estas tecnologías correctamente configuradas, podemos empezar con los despliegues, en primer lugar, el de Pulsar, y, una vez que este pasa a un estado funcional, el de Raphtory.

Para probar el despliegue, se puede ejecutar algún ejemplo de procesado del grafo, en el cual se leerán los datos desde el bucket de GCS, y se escribirán los resultados también en uno de estos buckets.

Por último, también se puede comprobar la automatización introducida con GitHub Actions, la cual …

En la Ilustración 16, podemos observar un diagrama conceptual de cómo queda realizado el despliegue de Raphtory como BDaaS en Google Cloud, donde los Pods contienen los contenedores que ejecutan los componentes de Pulsar y Raphtory:

Diagrama

Descripción generada automáticamente

Ilustración . Despliegue de Raphtory como BDaaS en Google Cloud

# Resultados

En este apartado, se recogen los resultados obtenidos de la fase de desarrollo. Así, se comenta el proceso detallado de cada uno de los despliegues, y las salidas obtenidas al ejecutarlos con diversas configuraciones.

## Resultados del despliegue de Raphtory en modo Single

En primer lugar, se recoge el proceso para lanzar Raphtory en su opción monolítica, es decir, ejecutándolo en el host como un objeto único que se encarga de todas las operaciones de procesado, desde la ingesta de los datos, hasta la generación de resultados.

Este ejemplo también nos permite ilustrar las distintas herramientas de las que depende Raphtory, lo que es útil para tener en cuenta los pasos necesarios en el resto de despliegues.

En primer lugar, se requieren estas utilidades:

* Java, en su versión 11, pues necesitamos un JDK para ejecutar el proyecto.
* Scala, en, al menos, la versión 2.12, que, como ya se ha comentado, es el lenguaje de programación en el que se ha desarrollado Raphtory.
* SBT [40], como mínimo en la versión 1.5. Esta es una herramienta de compilación muy útil para Scala, que también permite empaquetar el código, gestionar sus dependencias externas, o ejecutar pruebas.
* El repositorio de Raphtory, que se puede clonar directamente desde GitHub [5].

El método de instalación varía de un Sistema Operativo a otro, aunque es notable resaltar que, en el caso de utilizar distribuciones de Linux, es extremadamente sencillo obtener estas herramientas a través de SDKMAN! [41], una utilidad que permite la instalación y gestión de múltiples SDK en diferentes versiones.

En mi caso particular, este despliegue se ha realizado en un ordenador portátil con Windows 11, en su versión 21H2:

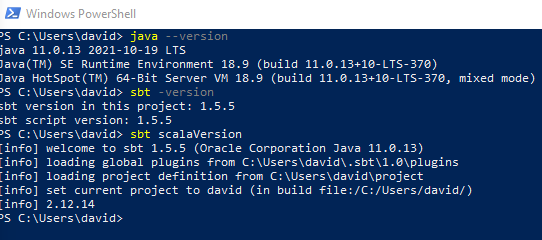


Ilustración 17. Terminal con las versiones instaladas de las herramientas necesarias para Raphtory.

Habiendo cumplido estos requisitos, el siguiente paso consiste en desplazarnos a la raíz del repositorio, y compilar el JAR (Java ARchive) de Raphtory. Para ello, es suficiente con el comando de la Ilustración 18, ejecutado directamente desde la consola que nos proporciona el IDE (Integrated Development Environment) IntelliJ [42], que es el que se ha empleado en el desarrollo y pruebas de este proyecto, ya que los lenguajes utilizados han sido Java y Scala.

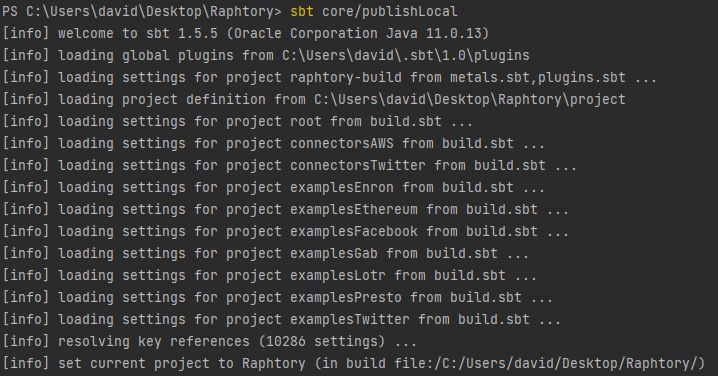


Ilustración 18. Comando y salida de la compilación del JAR de Raphtory.

Al final, SBT nos comunica el éxito de la operación:

Finalización del comando de compilación del JAR de Raphtory

Ilustración 19. Éxito de la compilación del JAR de Raphtory.

A continuación, nos desplazaremos al directorio del ejemplo, que, como se ha comentado en la sección de Desarrollo, es el de El Señor de los Anillos, y las apariciones conjuntas de los personajes en las frases del libro. Aquí, tenemos que lanzar una consola de SBT, y compilar el código de dicho ejemplo:

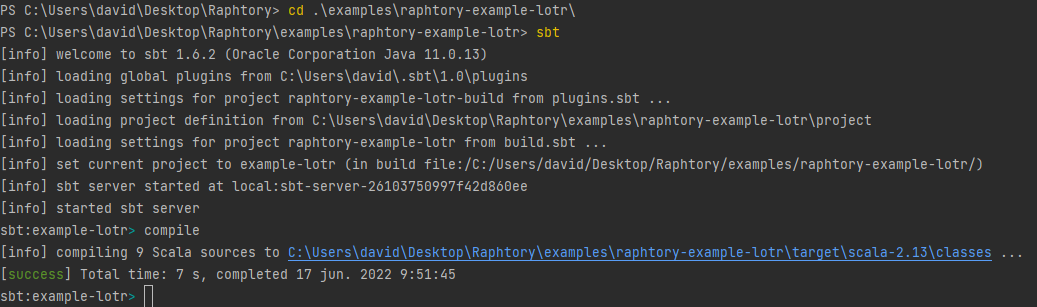


Ilustración 20. Compilación en SBT del ejemplo de LOTR.

Finalmente, solo nos queda ejecutarlo, lo que genera el mensaje de la Ilustración 21, puesto que SBT ha detectado numerosas clases que extienden de App, y que, por tanto, son ejecutables.

En este caso, se va a lanzar el ejemplo explicado en el Desarrollo, que, si recordamos, era FileOutputRunner. Este proceso termina de forma exitosa, tal y como se muestra en la Ilustración 22. Aquí podemos observar cómo se obtiene el grafo de entrada (lotr.csv), cómo se lanzan cada uno de los componentes que forman Raphtory, y, cuando ya está todo desplegado, cómo se ejecuta el análisis que se ha solicitado (DegreeSeparation para medir la distancia desde Gandalf al resto de personajes).

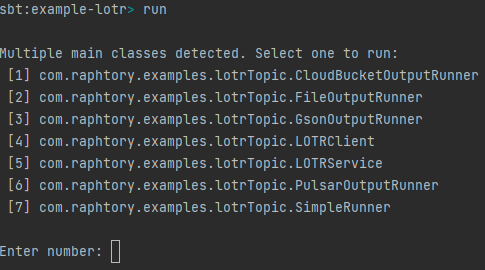


Ilustración 21. Posibilidades de ejecución del ejemplo monolítico.



Ilustración 22. Ejecución exitosa de FileOutputRunner en el ejemplo monolítico.

Por último, podemos comprobar en la Ilustración 23 y la Ilustración 24 que, en efecto, en el directorio tmp de nuestro equipo, se ha creado una carpeta con el nombre del algoritmo y el identificador de trabajo de análisis asignado, el cual, en su interior, contiene un fichero de texto, partition-0, con los resultados del análisis.

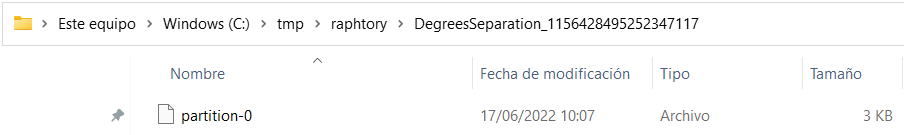


Ilustración 23. Directorio con la salida de FileOutputRunner.

Como lo estamos ejecutando con la opción monolítica en el host, simplemente se utiliza un Partition Manager, por lo que el resultado final se extrae de una sola partición, que aglutina los resultados del procesado del grafo completo.

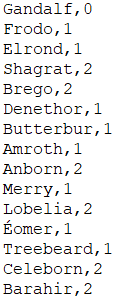


Ilustración 24. Salida de la ejecución de DegreeSeparation en el despliegue monolítico.

Podemos observar el nombre del personaje, y la distancia que lo separa de los demás, medida como saltos entre vértices. En efecto, dicha distancia será 0 consigo mismo, en el caso de Gandalf, normalmente 1 con personajes principales que aparecerán frecuentemente en el libro, como Frodo o Merry, y números superiores para personajes secundarios.

Con este ejemplo, concluimos la demostración del despliegue más básico de Raphtory, en la opción monolítica.

## Resultados del despliegue de Raphtory en modo pseudo distribuido

A continuación, realizamos una demostración del despliegue de Raphtory en modo pseudo distribuido, el cual, tal y como se explicó en la fase de Desarrollo, fundamentalmente consiste en el lanzamiento de cada componente de Raphtory por separado, coordinados por un líder, pero que, en realidad, simplemente son diferentes objetos, ejecutándose todos ellos en un solo equipo, enviando y recibiendo mensajes en puertos diferentes.

En primer lugar, para el correcto funcionamiento de este despliegue, es necesario configurar una serie de variables de entorno:

* RAPHTORY\_BUILD\_SERVERS: cantidad de servidores de Builders que se desean.
* RAPHTORY\_BUILDERS\_PER\_SERVER: cantidad de Builders en cada servidor.
* RAPHTORY\_PARTITION\_SERVERS: cantidad de servidores de Partition Managers.
* RAPHTORY\_PARTITIONS\_PER\_SERVER: cantidad de Partition Managers por servidor.
* RAPHTORY\_LEADER\_ADDRESS: dirección IP donde se despliega el líder.
* RAPHTORY\_LEADER\_PORT: puerto donde se despliega el líder.
* RAPHTORY\_BIND\_ADDRESS: dirección IP donde se va a desplegar el componente en cuestión.
* RAPHTORY\_BIND\_PORT: puerto donde se va a desplegar el componente en cuestión.

De este modo, un ejemplo de configuración será el siguiente:

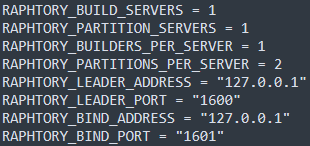


Ilustración 25. Ejemplo de configuración de las variables de entorno en un despliegue PD.

Se ha seleccionado un único servidor de particiones y constructores, para que el ejemplo sea sencillo y fácil de desplegar, pero dos particiones, en aras de demostrar que, en efecto, se puede realizar un procesado distribuido de un grafo, aunque sea dentro de un solo host, utilizando dos Partition Manager.

Además, se le ha asignado al líder la dirección local del equipo, y un puerto que no se esté usando, el 1600. El resto de componentes también se despliegan en la misma dirección IP, como ya se ha comentado, pero en un número de puerto distinto, que variará de un componente a otro.

Estas variables de entorno se pueden configurar desde un fichero interno del código de Raphtory llamado application.conf, con lo que simplemente se pueden modificar los valores, y la herramienta se encarga de manera automática de realizar los ajustes necesarios, sin necesidad de exportar todas las variables de entorno una y otra vez.

Habiendo seguido este proceso, solo nos queda desplegar cada componente, y ejecutar un análisis para comprobar el correcto funcionamiento de Raphtory. En este punto, sí que es extremadamente importante tener en cuenta que el líder ha de ser el primer componente en desplegarse, para que los demás puedan descubrirse y tener un punto de unión para sus comunicaciones. Si no seguimos este orden , la herramienta fallará.

Dicho esto, para desplegar cada componente, tal y como se explicó en la sección de Desarrollo, podemos arrancar una consola de SBT en la ruta correspondiente, compilar el proyecto, y ejecutar el objeto Component pasando el String apropiado. Así, en el caso del líder, podemos observar la Ilustración 26.

Vemos que se le ha pasado el String “leader”, que se ha desplegado en la dirección IP y puerto seleccionados, y que se ha lanzado un clúster de Akka, en el cual, él, que es el único nodo, ha sido asignado como líder.

Del mismo modo, a continuación, podemos ver que se han desplegado el resto de componentes de Raphtory, utilizando el String correspondiente en cada caso. Asimismo, en la Ilustración 27, podemos ver cómo el líder registra a los nuevos nodos que se van uniendo al clúster.

También es muy interesante destacar cómo, en la Ilustración 30, se empieza a observar el problema que presenta Akka en este proyecto, que ya se había mencionado con anterioridad. En efecto, podemos apreciar la aparición de Dead Letters [43].

Las Dead Letters son mensajes que no se pueden entregar, ya que Akka funciona con una calidad Best-Effort, y, por ende, es posible que, si las comunicaciones entre componentes comienzan a saturarse, algunos mensajes vayan a ser descartados.

Al final, las retransmisiones solucionan el problema, pero, desde luego, se obtiene un rendimiento inferior comparado con el que se podría obtener sin las Dead Letters.

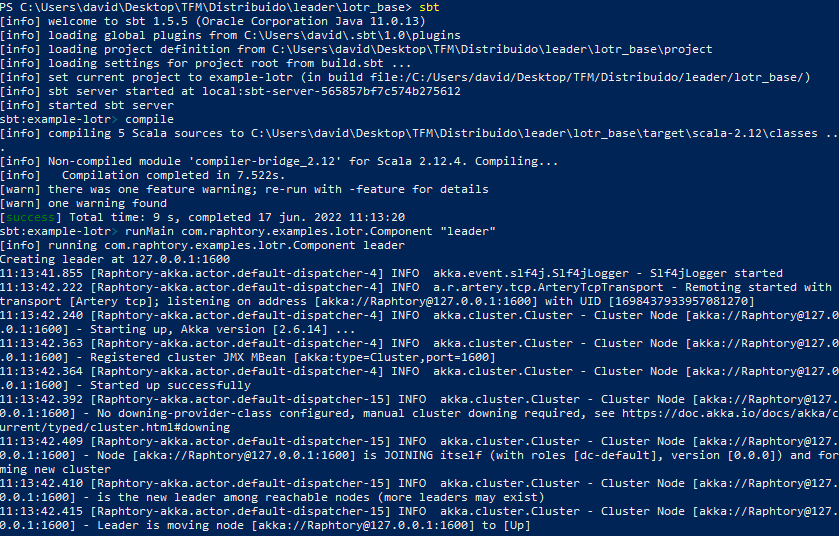


Ilustración 26. Logs de despliegue del líder en PD.

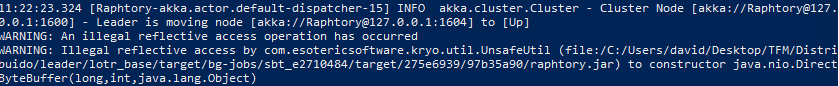


Ilustración 27. Un nuevo componente se une al clúster, siendo registrado por el líder.

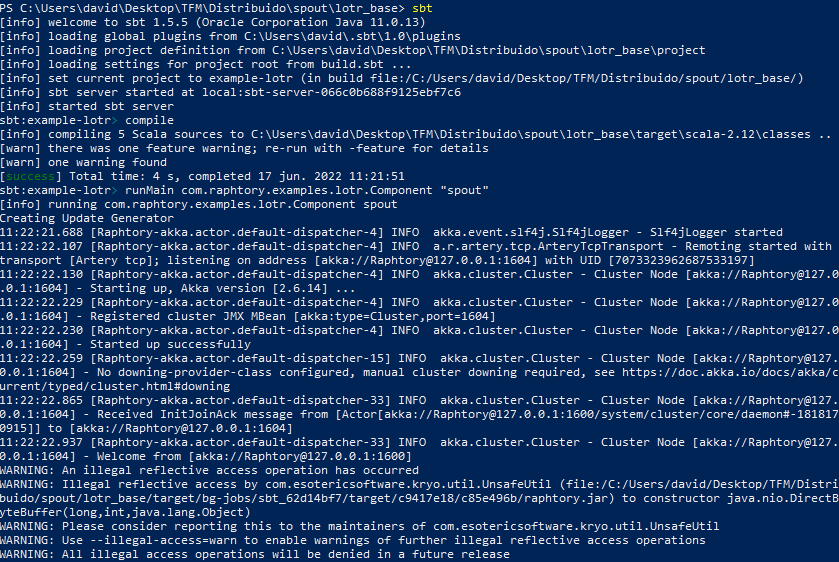


Ilustración 28. Despliegue del Spout, y unión al clúster del Líder.

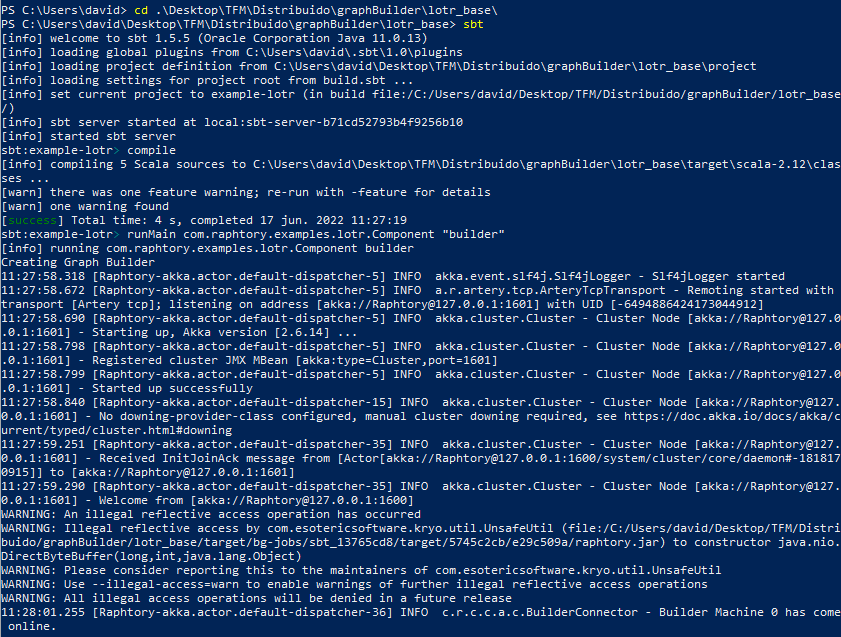


Ilustración 29. Despliegue del Builder, y unión al clúster del Líder.

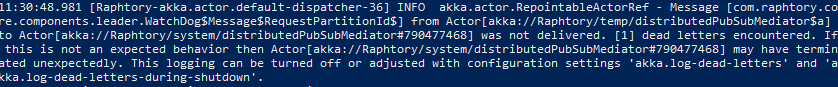


Ilustración 30. Log del terminal en el despliegue PD que muestra la aparición de Dead Letters.

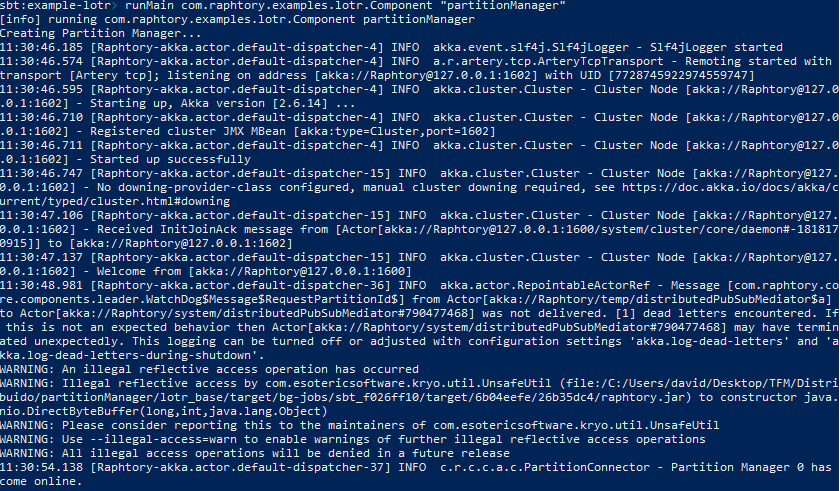


Ilustración 31. Despliegue del gestor de particiones en PD.

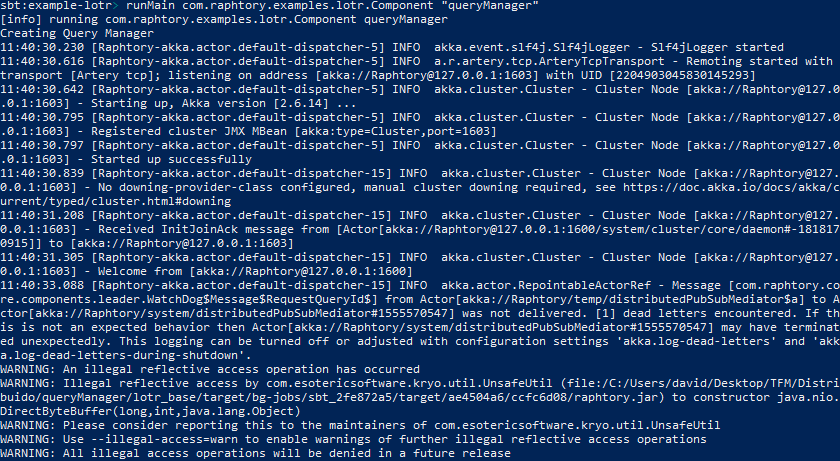


Ilustración 32. Despliegue del Query Manager en la modalidad PD.

Una vez hemos desplegado todos los componentes de Raphtory, y todos se han conectado con el líder, este nos informa de que ya está toda la estructura creada, y lista para empezar a procesar, tal y como observamos en la Ilustración 33. Además, vemos que el clúster se ha configurado exactamente como habíamos indicado con las variables de entorno, con dos particiones.

Terminal que nos muestra que el clúster está listo.

Ilustración 33. Log del líder que muestra que el clúster de Raphtroy está preparado.

Finalmente, para comprobar que funciona, se ejecuta también una clase cliente, mostrada en la Ilustración 34.

Texto

Descripción generada automáticamente

Ilustración 34. Clase Client para probar Raphtory PD.

Así, el objeto Client se conecta con el clúster, a través de la dirección IP y puerto del líder, y ejecuta una consulta, en este caso, con el algoritmo GraphState, que genera una serie de metadatos para cada vértice del grafo, y cuya descripción completa y formato de salida pueden consultarse en [44], ya que la algoritmia escapa al ámbito de este TFM.

En cualquier caso, en la Ilustración 35 podemos ver la ejecución del cliente, intensificándose el problema de las Dead Letters, y, en la Ilustración 36, cómo el Líder lo reconoce correctamente.

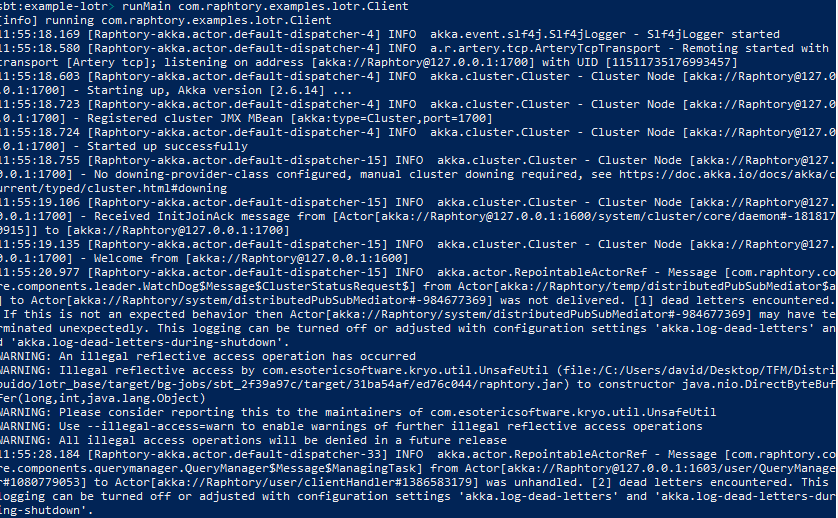


Ilustración 35. Ejecución de un cliente que se conecta con el clúster.

Terminal que muestra al líder reconociendo al cliente.

Ilustración 36. El líder reconoce al cliente en el clúster.

Finalmente, si nos vamos al directorio de resultados, tal y como observamos en la Ilustración 37, vemos que ahora tenemos dos ficheros de salida, cada uno de ellos proveniente de una de las dos particiones con las que se ha configurado el despliegue.



Ilustración 37. Salidas de la ejecución de Raphtory en modo PD con dos particiones.

Estos dos ficheros contienen el resultado de procesar con GraphState cada una de las dos particiones del grafo, con el formato que se puede observar en la Ilustración 38, y en el que no se va a entrar, ya que, como se ha comentado, la algoritmia de procesado de grafos no es el objeto de este Trabajo Fin de Máster.

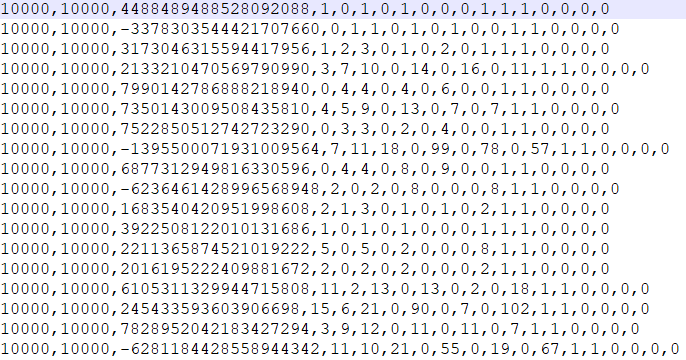


Ilustración 38. Salida de la ejecución de GraphState en Raphtory PD

De igual modo, una vez ejecutado el procesado, si vamos apagando cada componente, podemos ver como el líder lo notifica, y los va eliminando el clúster.

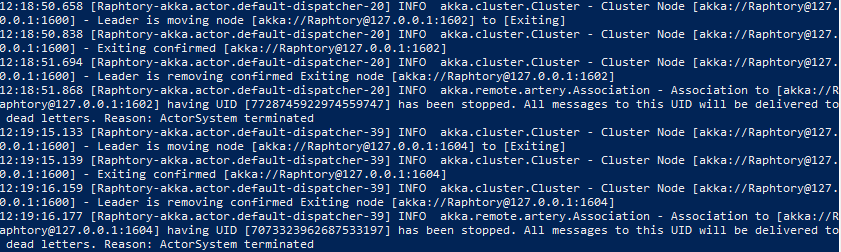


Ilustración 39. El líder elimina los componentes de Raphtory conforme estos se van apagando.

Por tanto, a través de esta demostración, se ha comprobado que Raphtory puede desplegarse como un sistema distribuido, cada componente ejecutándose de forma independiente, y en el que es viable realizar el procesado también distribuido de un grafo, pues lo hemos dividido en dos particiones, y se ha obtenido una salida correspondiente al procesado de cada una de ellas.

De este modo, se muestra cómo Raphtory, y esta modalidad de despliegue, son capaces de vencer las dificultades del procesado distribuido de grafos, recogidas en el Estado del arte, aprovechando sus ventajas, y que puede comportarse como un sistema distribuido funcional.

## Resultado del despliegue de Raphtory con Docker y Docker Compose

Habiendo experimentado y probado el comportamiento de Raphtory en sus modalidades monolítica y pseudo distribuida, a través de los ejemplos desarrollados en los apartados anteriores, a continuación vamos a implementar un despliegue propio verdaderamente distribuido, mediante el uso de contenedores.

Como se comentado en la fase de Desarrollo, el objetivo fundamental consiste en que cada componente de Raphtory se ejecute como un objeto distinto, pero en un contenedor propio, de manera que nuestra herramienta constituiría un sistema distribuido real, que habilita un verdadero procesado distribuido de un grafo.

Así, para implementar los contenedores necesarios, y orquestar el despliegue, vamos a emplear Docker y Docker Compose.

El primer paso consiste en la creación de un Dockerfile que sirva como base para la imagen de los contenedores. Esto es relativamente sencillo, porque, realmente, la distinción a la hora de ejecutar un componente u otro de Raphtory son las variables de entorno, y el String que se pasa como argumento a la clase RaphtoryService, ambos fácilmente configurables con Docker Compose.

De este modo, se ha desarrollado el siguiente Dockerfile común para todos los contenedores:

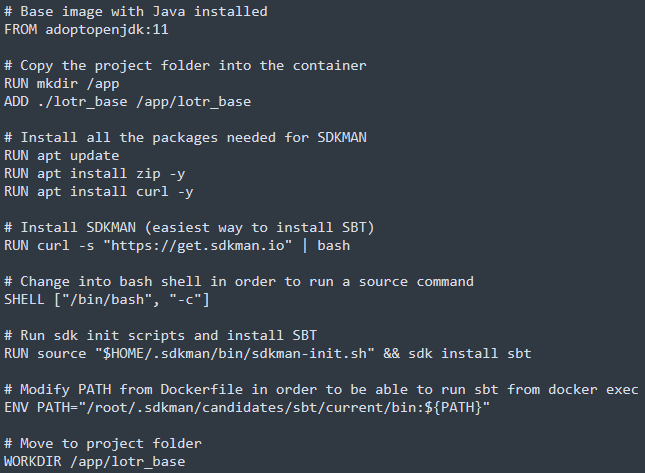


Ilustración 40. Dockerfile empleado en el despliegue Dockerizado.

Así, tomamos como base la imagen adoptopenjdk:11, referenciada en [21], y descrita en la sección “Despliegue de Raphtory con Docker y Docker Compose”. Dicha imagen contiene un JDK de código libre sobre una distribución mínima de Linux, llamada Alpine [45], gracias a lo cual podemos realizar un despliegue ligero y eficiente, pero correcto.

A continuación, en la imagen del contenedor, creamos un directorio app, y copiamos el proyecto “lotr\_base”, que contiene los JAR de Raphtory y el ejemplo de LOTR correctamente empaquetados.

El siguiente paso es la instalación de SBT, para lo cual, tal y como se ha comentado anteriormente, al tratarse de una distribución de Linux, lo más cómodo es utilizar SDKMAN!. Es por ello por lo que, previamente, se instalan los paquetes zip y curl, y se ejecuta el comando para descargar y ejecutar la instalación.

A continuación, es necesario indicar a la imagen que utilice el Shell de POSIX /bin/bash, más completo que la opción por defecto /bin/sh, en aras de poder ejecutar SDKMAN! e instalar SBT. Con estos pasos realizados, modificamos el PATH para poder lanzar SBT desde el contenedor, y situamos el directorio de trabajo en la carpeta con el proyecto.

De este modo, tenemos una imagen con todas las necesidades cubiertas para ejecutar un componente de Raphtory. Lo único que nos queda es crear un despliegue en Docker Compose que lance cada parte en un contenedor, y los conecte entre sí. En un principio, se ha preparado la siguiente configuración, en un fichero docker-compose.yml:

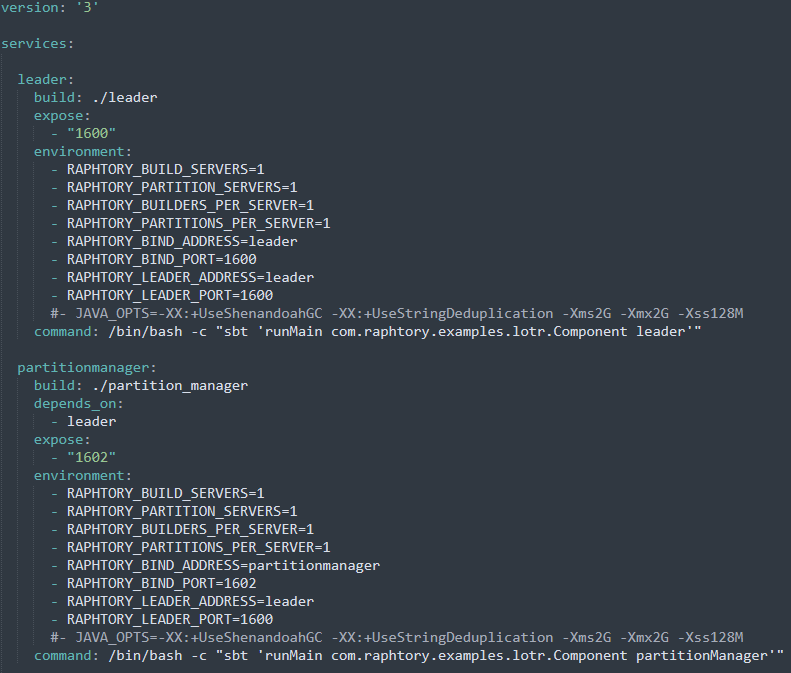


Ilustración 41. Fichero docker-compose.yml con la configuración del despliegue dockerizado.

Como se puede observar, simplemente se lanza cada componente, indicando la ruta donde se encuentra el Dockerfile que va a utilizar, que es el descrito anteriormente. A continuación, todos los componentes, excepto el líder, llevan la directiva “depends\_on”, para que no se desplieguen hasta que lo haya hecho el propio líder, ya que, como se ha comentado en el apartado previo, es obligatorio seguir este orden para el correcto funcionamiento de Raphtory.

Después, se expone el puerto correspondiente, para que se puedan recibir y enviar mensajes a través de él, mediante la red por defecto que despliega Docker Compose.

El siguiente paso es la configuración de las variables de entorno, exactamente las mismas que en el caso pseudo distribuido, solo que, ahora, utilizamos el DNS de Docker Compose para indicar las direcciones IP en las que se encuentra cada componente. También se llevó a cabo algo de experimentación con las opciones de Java, tal y como se muestra en las líneas comentadas de la Ilustración 41, pero, finalmente, no se apreciaba un cambio significativo en el rendimiento, o en la aparición de Dead Letters, así que se prefirió dejar la configuración por defecto.

Finalmente, se ejecuta el comando de ejecución del componente (el mismo que en PD), pasando el String correspondiente para que se despliegue un líder, un gestor de particiones, un gestor de consultas, un constructor, y un Spout. En la Ilustración 41 solo se muestran los dos primeros elementos de Raphtory, pero el fichero contiene todos los demás, con las configuraciones adecuadas.

Además, para facilitar las pruebas, también se despliega un contenedor cliente, que espera a la ejecución de todos los demás, y lanza la misma clase Client que en la modalidad distribuida.

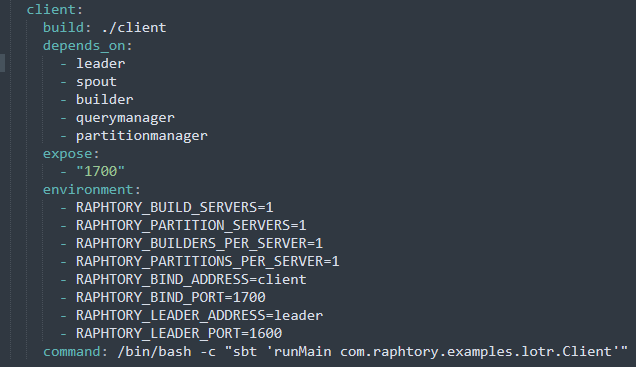


Ilustración 42. Configuración del contenedor del cliente.

Así, lo único que queda es construir las imágenes, como en la Ilustración 43, y lanzar el despliegue, reflejado en la Ilustración 44, usando los comandos básicos de Docker Compose (docker-compose build y docker-compose up)

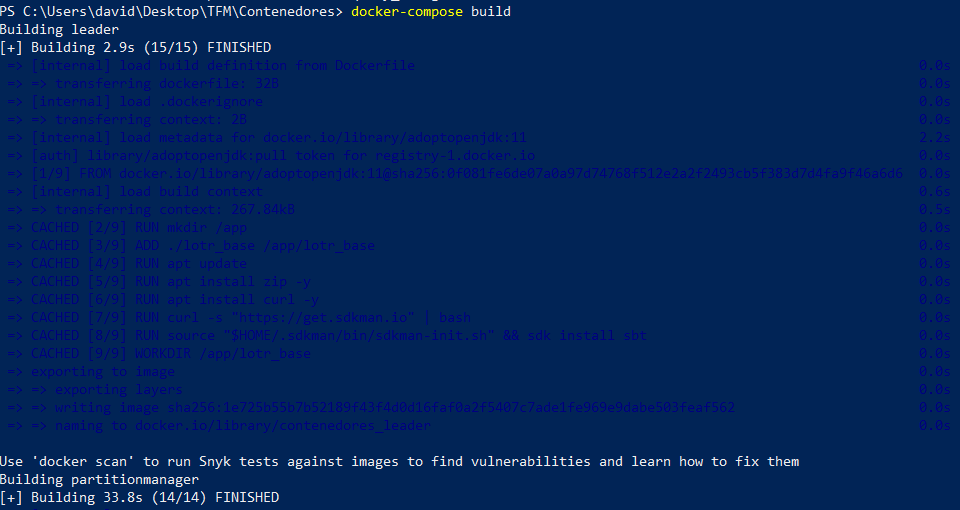


Ilustración 43. Parte de los logs de la construcción de las imágenes.

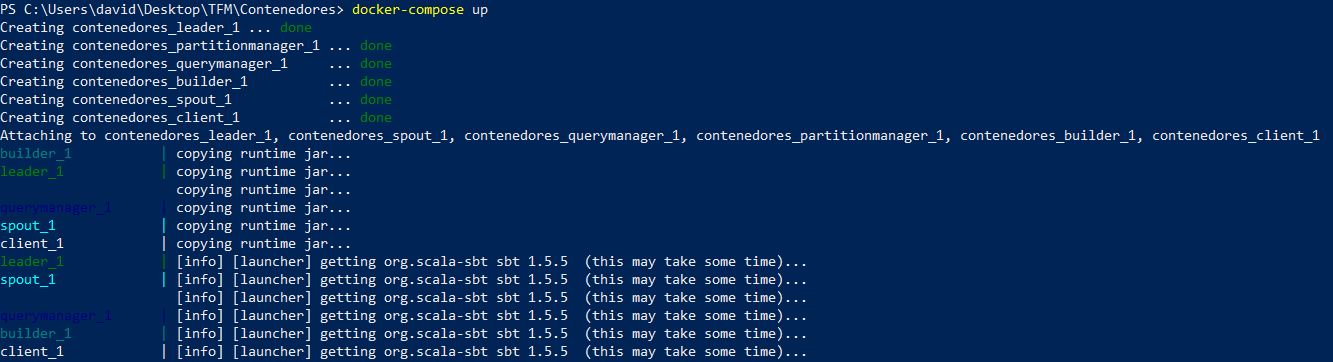


Ilustración 44. Parte de los logs del despliegue en Docker.

Como se observa en la Ilustración 45, cuando todos los componentes de Raphtory se han desplegado, cada uno en su contenedor, el líder, en un mensaje similar al del caso PD, nos muestra que el clúster ya está listo para empezar a recibir solicitudes de análisis. Además, Docker Desktop también nos muestra en la Ilustración 46 que el clúster se ha desplegado correctamente.

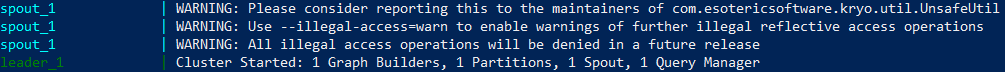


Ilustración 45. Parte del log de "docker-compose up", con el clúster preparado.

De este modo, podemos realizar el procedimiento mostrado en la Ilustración 47, en la que entramos en el contenedor del cliente, nos situamos en el directorio /tmp, y vemos que, del mismo modo que en los despliegues anteriores, se ha creado un directorio con el nombre del algoritmo ejecutado, junto con el identificador del trabajo de análisis. Dentro, se encuentra el fichero partition-0 con el resultado, igual que en los casos anteriores.

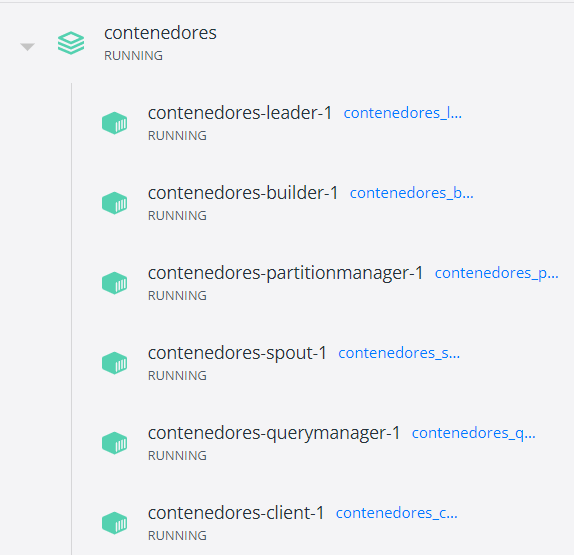


Ilustración 46. Despliegue del clúster operativo en Docker Desktop.

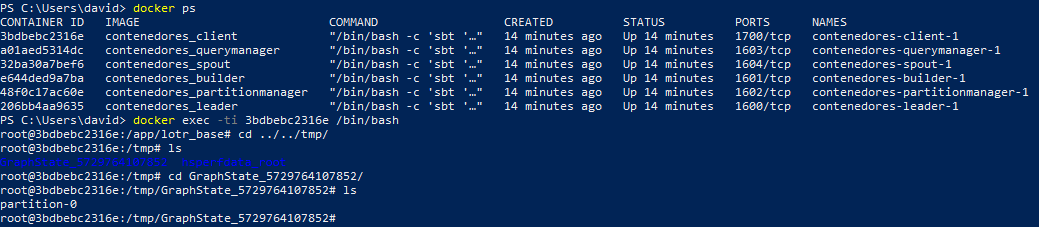


Ilustración 47. Entrada en el cliente, y comprobación de los resultados.

Así, hemos completado la implementación de un despliegue de Raphtory verdaderamente distribuido, en el que cada componente se ejecuta en un contenedor distinto, todo ello orquestado por Docker Compose.

## Resultados del despliegue de Raphtory como un BDaaS en la Nube

Tras llevar a cabo los anteriores despliegues, y la evaluación de tecnologías necesarias, tal y como se recoge en la sección de Desarrollo, estamos en disposición de cumplir el objetivo final del TFM, es decir, desarrollar un despliegue de Raphtory en la nube, como un BDaaS, para el procesado distribuido de grafos.

Como ya se ha explicado, los pasos generales son, fundamentalmente:

* Generación de las imágenes con Docker, y subida a Docker Hub.
* Creación de un proyecto en Google Cloud, con un clúster de GKE, y un bucket de almacenamiento de GCS.
* Configuración del SDK de Google Cloud para conectar con nuestro clúster en la nube.
* Configuración y despliegue de Apache Pulsar.
* Configuración y despliegue de Raphtory.

En este punto, es importante volver a resaltar que Pometry actualizó Raphtory a la versión 0.5, la cual introdujo cambios muy significativos, de los cuales, los que más afectan a este trabajo son la sustitución de Akka por Apache Pulsar, que permite deshacernos del problema de las Dead Letters, y la desaparición del líder como componente de los despliegues distribuidos, descentralizando aun más esta modalidad de la herramienta.

Así, el despliegue de Raphtory como un BdaaS en la Nube se realiza del siguiente modo:

### Construcción y subida de las imágenes

En primer lugar, es necesario construir las imágenes de Docker que se van a utilizar para ejecutar los componentes en la nube dentro de Kubernetes.

Para ello, Pometry nos proporciona directamente un Dockerfile, que ha sido ligeramente modificado, con las siguientes características:

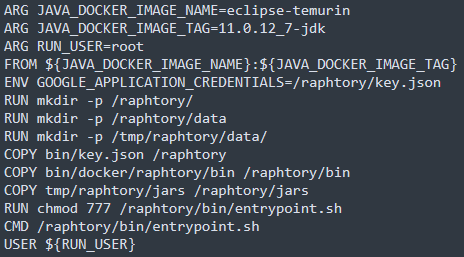


Ilustración 48. Dockerfile para el despliegue en la Nube.

Como vemos, el Dockerfile parte de una imagen base eclipse-temurin, que es la versión más actual de la ya referenciada en [21] con AdoptOpenJDK. Sobre esta imagen, se copian los JAR que van a ser utilizados, tanto el general de Raphtory, como el de nuestro ejemplo, y se ejecutan algunas configuraciones del entorno del contenedor con el script entrypoint.sh. Simplemente, tenemos que copiar ambos JAR en la ruta que indica el Dockerfile, y ejecutar docker build.

Al hallarnos en la versión 0.5, la compilación con SBT ha cambiado. Ahora, en la raíz del repositorio ejecutamos dos comandos:

* sbt core/assembly, para generar el JAR completo de Raphtory.
* sbt examplesLotr/package, que construye el JAR de nuestro ejemplo, que sigue siendo el de El Señor de los Anillos.

En este punto, también es destacable el hecho de que se está introduciendo el fichero key.json en la imagen, que son las credenciales para que, a la hora de ejecutar un análisis, Raphtory pueda conectarse con GCS para leer y escribir los resultados. Asimismo, mediante la variable de entorno GOOGLE\_APPLICATION\_CREDENTIALS, podemos indicar donde encontrar dichas credenciales.

En cualquier caso, construidas las imágenes, hay que hacer un docker push image:tag a Docker Hub. Así, por ejemplo, obtendríamos el siguiente resultado, que muestra que las imágenes están correctamente almacenadas:

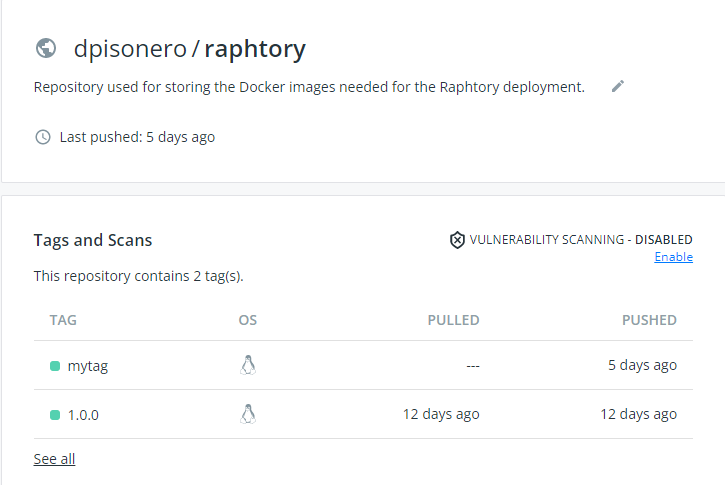


Ilustración 49. Captura de mi repositorio en Docker Hub.

### Gestiones en Google Cloud

Antes de continuar con el despliegue, deberíamos asegurarnos de que somos propietarios de un proyecto en Google Cloud, con todos los requisitos necesarios.

En mi caso, como se aprecia en la Ilustración 50, he creado un proyecto llamado RaphtoryTest, con el ID correspondiente.

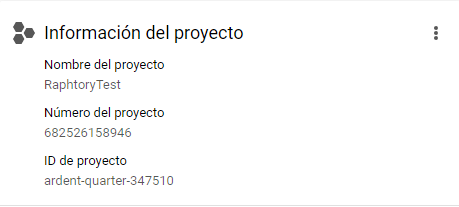


Ilustración 50. Proyecto creado en Google Cloud.

Asimismo, en el apartado IAM y administración > Cuentas de servicio, me he creado una nueva cuenta, llamada TestUpDown (Ilustración 51), con permisos de propietario. En el apartado claves (Ilustración 52), he creado un juego, y lo he descargado, que es el fichero key.json que se mencionaba en el apartado anterior. Este paso es necesario porque, sin estas claves, no podríamos conectar con GCS para la lectura de los datos de entrada, y la escritura de los resultados.

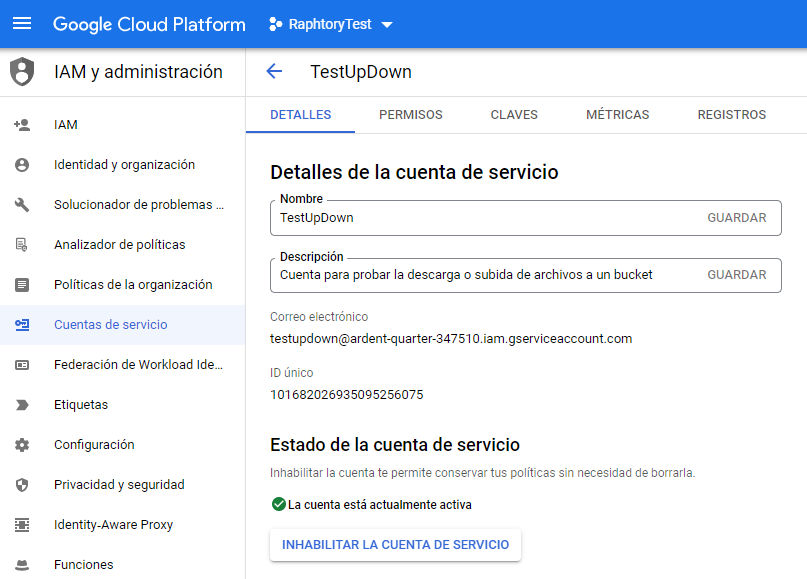


Ilustración 51. Creación de la cuenta de servicio.

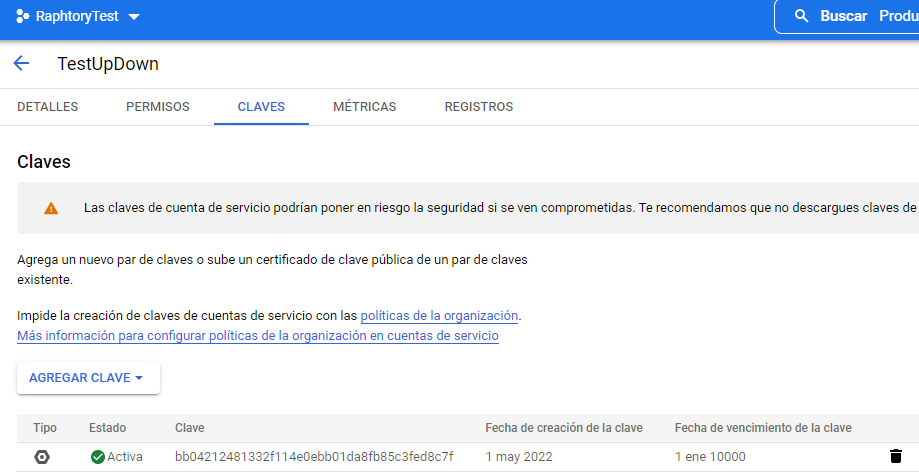


Ilustración 52. Claves de la cuenta de servicio.

Creado nuestro proyecto, el siguiente paso es el lanzamiento de un clúster donde alojar nuestros despliegues de Kubernetes. En mi caso, tal y como representé en el diagrama, lo he montado con tres nodos, en la zona de computación Europa – Sudoeste -1, para que esté lo más cerca posible de Madrid.

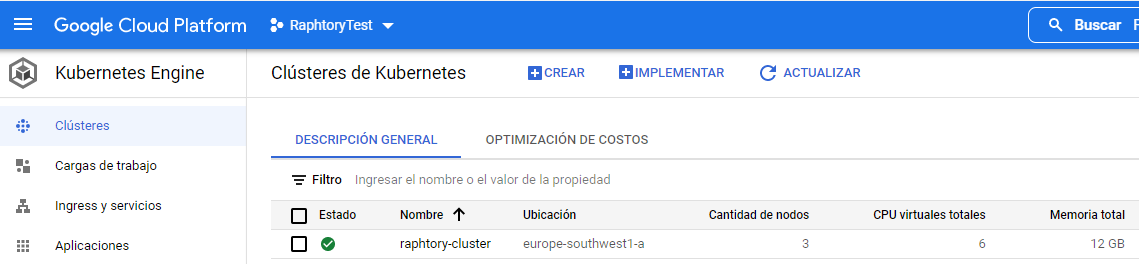


Ilustración 53. Clúster de tres nodos creado en Google Kubernetes Engine.

Finalmente, he creado un bucket en Cloud Storage para guardar tanto los datos de entrada, como los resultados. Curiosamente, Google exige que el nombre de dicho bucket sea universalmente único, así que hay que tenerlo en cuenta.

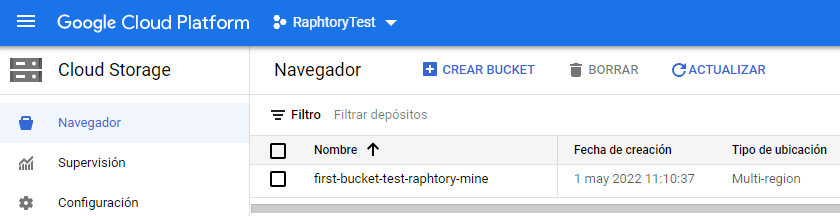


Ilustración 54. Cloud Storage con el bucket creado.

Todos estos elementos se pueden crear utilizando la interfaz gráfica de la consola de Google Cloud, por lo que es muy sencillo y cómodo.

### Configuración de Google Cloud SDK

Con todo el proceso previo, el siguiente paso es la configuración de Google Cloud SDK para poder conectar con los elementos que acabamos de crear.

Suponiendo que tengamos esta herramienta descargada e instalada, simplemente, en un terminal, ejecutamos el comando gcloud init (Ilustración 55). En mi caso, ya lo tenía configurado todo, pero, si no, simplemente seleccionamos nuestra cuenta de correo, y el mismo identificador de proyecto y zona de computación que empleamos en los pasos anteriores.

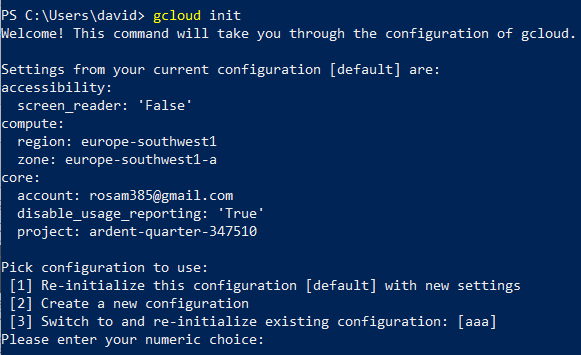


Ilustración 55. Configuración de Google Cloud SDK.

La última configuración es la conexión con nuestro clúster de GKE. Para ello, solo es necesario instalar el cliente de Kubernetes en gcloud, ejecutando gcloud components install kubectl, y configurarlo con las credenciales de nuestro clúster (Ilustración 56). Como se muestra en dicha ilustración, si ahora ejecutamos comandos de K8s, estos nos devuelven la información del clúster desplegado en la nube.

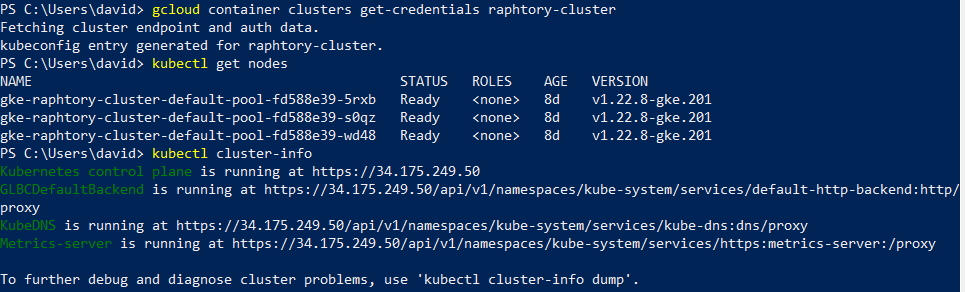


Ilustración 56. Credenciales y conexión con GKE.

### Despliegue de Apache Pulsar

Después de realizar estas configuraciones, estamos en disposición de lanzar el despliegue de Apache Pulsar, que, como ya se ha comentado, sustituye a Akka para las comunicaciones entre componentes.

Para llevar esto a cabo, podemos usar el siguiente script:

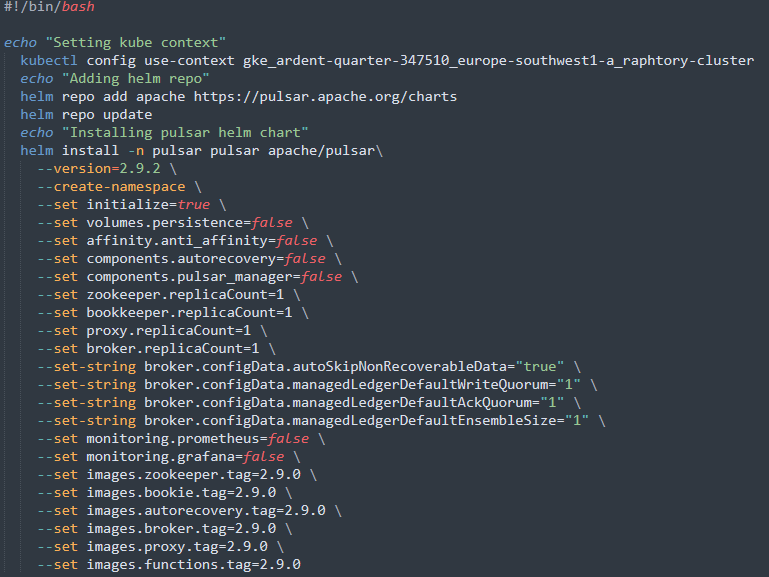


Ilustración 57. Script de lanzamiento de Apache Pulsar.

Como vemos, en primer lugar, configuramos el cliente de Kubernetes para que quede asociado al contexto de nuestro clúster en la Nube. Este lo podemos obtener ejecutando el comando de la Ilustración 58 en un terminal con gcloud configurado.

Terminal con el contexto del clúster de Kubernetes.

Ilustración 58. Contexto del clúster de GKE.

En el resto del script, simplemente se añade el chart oficial de Apache Pulsar, y se configuran algunos metadatos que se adecuen a nuestro despliegue, que son los recomendados por defecto, a excepción de algunas versiones, que se han seleccionado las que garantizan que el despliegue termina funcionando.

Tras esperar un tiempo prudencial, aproximadamente unos 5 minutos, podemos consultar nuestro panel de GKE, en la pestaña de Ingress y servicios, y encontraremos lo mostrado en la Ilustración 59.

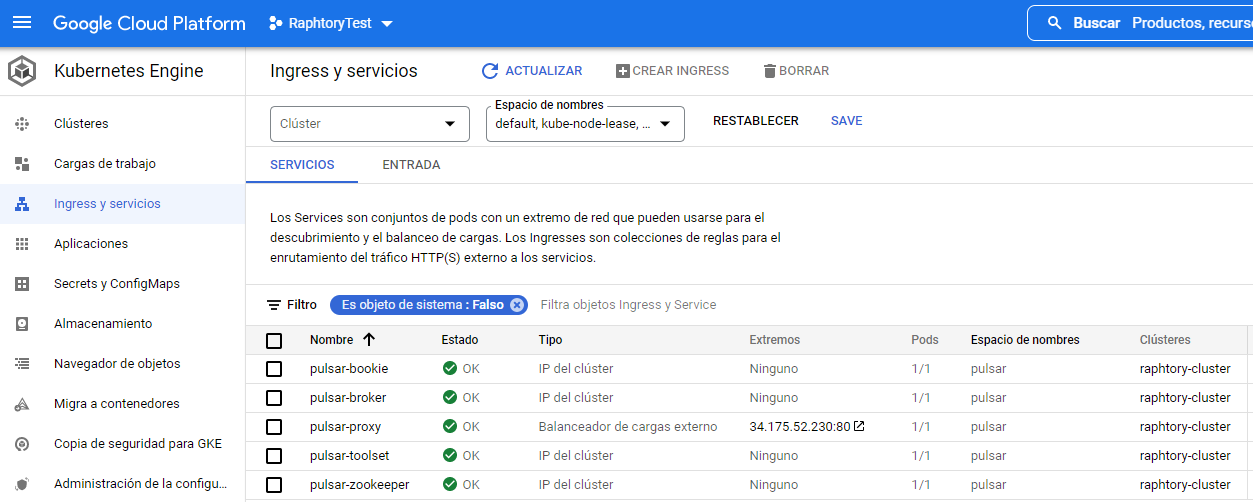


Ilustración 59. Despliegue de Apache Pulsar en GKE.

Como vemos, los Pods de Pulsar han quedado correctamente desplegados, y se corresponden con la estructura comentada en la sección de Desarrollo, junto con un Proxy adicional.

Una vez que comprobamos que el despliegue es correcto, para que las comunicaciones entre los componentes de Raphtory se encaminen correctamente a través de Pulsar, es necesario ejecutar los tres comandos de la Ilustración 60.

Texto de los comandos de encaminamiento en Pulsar.

Ilustración 60. Redirección de puertos en Apache Pulsar.

### Despliegue de Raphtory en el clúster de GKE

En este punto, ya tenemos todas las piezas necesarias para, por fin, desplegar Raphtory en la Nube, y ofrecer nuestro servicio BDaaS.

Así, lo primero que tenemos que hacer, de forma similar a los anteriores despliegues, es la configuración de las variables de entorno. Como se observa en la Ilustración 61, estas han cambiado considerablemente respecto a los despliegues anteriores, al hallarnos en la versión 0.5.

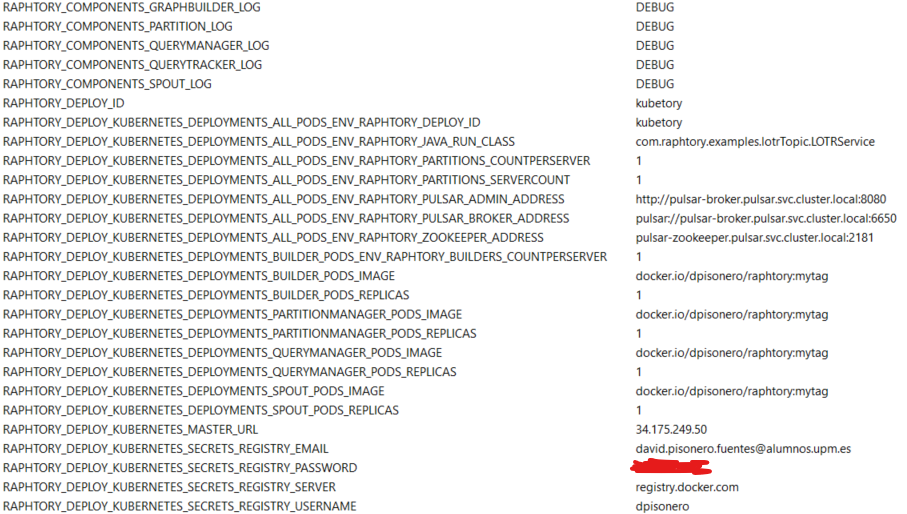


Ilustración 61. Variables de entorno del despliegue en Raphtory en la Nube.

Básicamente, yendo en orden, realizan las siguientes funciones:

* Activan unos flags de depuración.
* Asignan un identificador cualquiera al despliegue, en este caso, “kubetory” (K8s + Raphtory).
* Indican la clase que ha de ejecutar cada componente, en este caso LOTRService, de la que se hablará a continuación.
* Se indican cuantos servidores de particiones se van a desplegar, y cuantas particiones va a haber en cada uno. Hay variables equivalentes para los demás componentes.
* Se indican las direcciones IP de los elementos ya desplegados de Apache Pulsar, para que las diferentes partes de Raphtory puedan comunicarse entre sí. Como se ve, se utilizan los nombres de los servicios de Kubernetes, no direcciones IP, que se resuelven internamente vía DNS.
* Se indica la dirección de Docker Hub donde están alojada la imagen base que se emplea para construir cada uno de los componentes.
* La dirección IP del clúster de GKE donde se ha de realizar el despliegue, que coincide con la enseñada en la Ilustración 56.
* Las credenciales de para poder acceder a la imagen a Docker Hub (URL, correo, y nombre de usuario). Por motivos obvios, se ha censurado la contraseña.

En lo que respecta a la clase LOTRService, esta es equivalente a la clase Component de los anteriores despliegues, pero actualizada a la versión 0.5. Es decir, contiene un objeto ejecutable que extiende de RaphtoryService, y espera un String que le indique qué componente se va a desplegar.

La única diferencia remarcable es que el Spout definido, en lugar de ser el tradicional File Spout, se ha sustituido por el Cloud Spout que se desarrollado, el cual es muy parecido, pero implementa un bloque de código que permite conectar con el bucket de Cloud Storage creado previamente para obtener los datos de entrada.

Por supuesto, se han tenido que modificar las dependencias en SBT para poder utilizar la API de Google. La parte relevante de este código se muestra a continuación.

Código relevante de la clase Cloud Spout.

Ilustración 62. Código relevante de CloudSpoutç

Básicamente, permite conectarse con el servicio de Cloud Storage a partir del identificador de proyecto, y obtener el objeto a procesar, que será el fichero lotr.csv previamente subido a la Nube por el usuario.

Dicho esto, el último paso para ejecutar el despliegue consiste en ejecutar la clase Deploy, perteneciente al proyecto de Scala de Raphtory. Esta clase se encarga, de manera completamente automática, de realizar el despliegue sobre Kubernetes, en este caso, en nuestro clúster que se encuentra en la Nube, de todas las partes de Raphtory, teniendo en cuenta toda la configuración realizada previamente. Para ejecutar dicha clase, desde la raíz del repositorio de Raphtory, hacemos:

java -cp examples/raphtory-example-lotr/target/scala-2.13/example-lotr\_2.13-0.5.jar:core/target/scala-2.13/core-assembly-0.5.jar com.raphtory.deployment.kubernetes.Deploy

De este modo, si lanzamos el Deploy, tras una espera prudencial (no más de 5 minutos), podemos consultar nuestro panel de GKE.

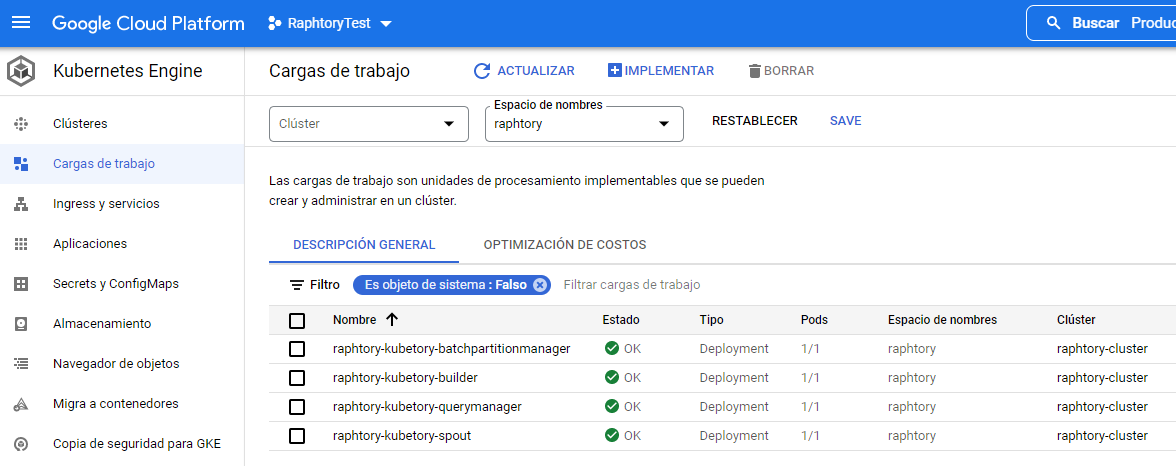


Ilustración 63. Despliegue en GKE de Raphtory.

Como podemos observar, acabamos de desplegar Raphtory, como un sistema distribuido, en la Nube, concretamente en un clúster de GKE. Si nos fijamos, el Partition Manager, en este caso, se llama Batch Partition Manager. Ello se debe a unos cambios de última hora que se realizaron en el código fuente de Raphtory, los cuales variaron el nombre de este componente, debido a que asumió algunas funcionalidades adicionales.

Para comprobar que el despliegue funciona, podemos entrar en uno de los contenedores, y ejecutar la clase cliente LOTRClient. Como ocurre con el caso de LOTRService, esta clase es equivalente a nuestro Client antiguo, con la salvedad de que se ha modificado el objeto que almacena la salida.

Concretamente, se ha implementado un CloudBucketOutputFormat, que es similar al almacenamiento de un fichero en local, pero que incluye un código adicional para que escriba la salida en la Nube.

Dicho código, observable en la Ilustración 64, accede a Cloud Storage, comprueba si el bucket que el usuario ha pasado como argumento ya existía, y, en caso contrario, lo crea. A continuación, guarda en dicho bucket el fichero de salida, con un nombre del tipo Algoritmo\_Id de trabajo de análisis\_partition-X, en función de las particiones que se hayan seleccionado.

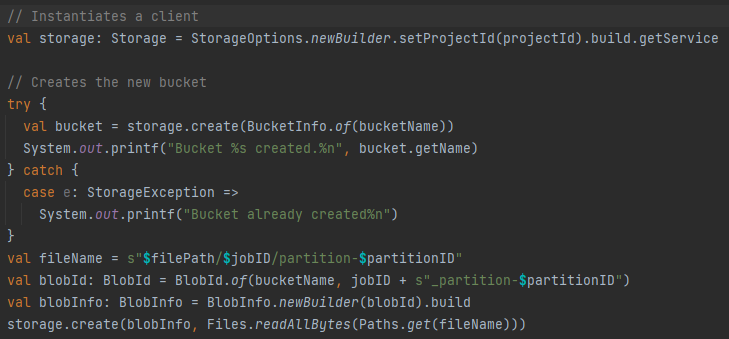


Ilustración 64. Código relevante de la clase CloudBucketOutputFormat.

Para entrar en uno de los Pods, y ejecutar el cliente, podemos utilizar el comando de la Ilustración 65.

Texto con el comando para entrar en un Pod.

Ilustración 65. Comando para entrar en un Pod.

Sin embargo, en nuestro caso, vamos a emplear las facilidades que nos proporciona Lens [46], un entorno que permite conectar con clústeres de Kubernetes, ya sea en local, o en GKE, tal y como se muestra en la Ilustración 66, pues es el mismo clúster que el que aparecía en el panel de Google Cloud, con los mismos Pods de Raphtory.

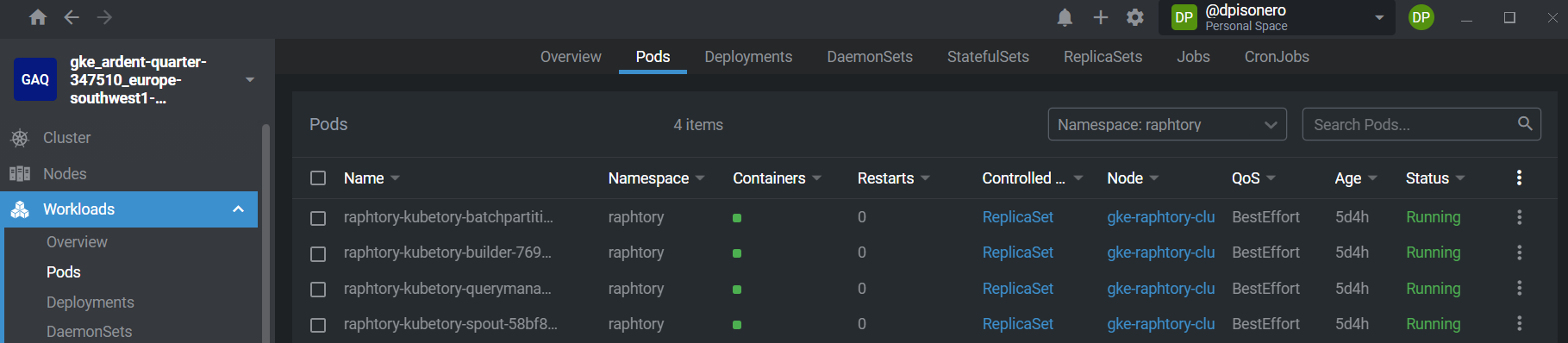


Ilustración 66. Interfaz gráfica de Lens.

De este modo, podemos entrar en un Pod cualquiera, por ejemplo, el Batch Partition Manager, y ejecutar la clase LOTRClient, la cual, en este caso, emplea el algoritmo Connected Components (identifica a qué Connected Component pertenece cada vértice o personaje, siendo un Connected Component un conjunto de vértices directamente unidos entre sí). Los detalles del algoritmo, que no son de interés para este proyecto, pueden consultarse en [47].

Al ejecutar la clase, que requiere funcionalidades de los dos JAR empaquetados en la imagen, observamos, que, en efecto, comienza el análisis:

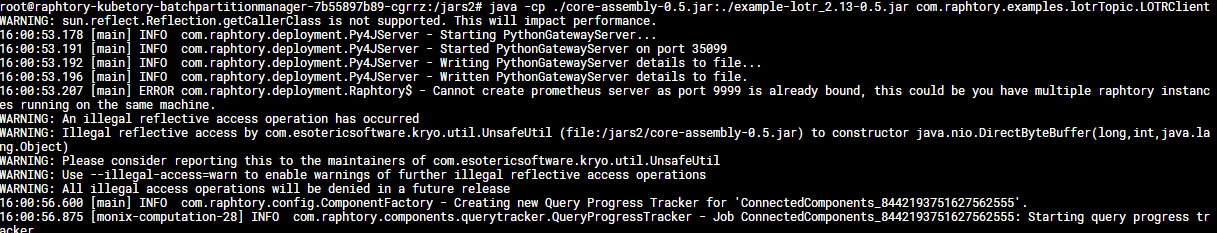


Ilustración 67. Ejecución del cliente en la Nube.

Al arrancar el análisis, si accedemos a los logs que se están generando, vemos lo indicado en la Ilustración 68.

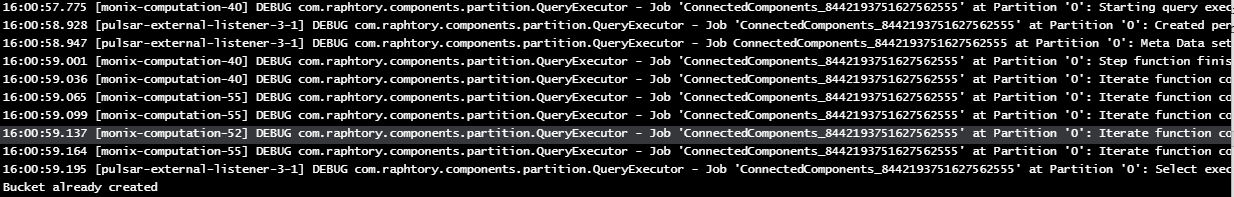


Ilustración 68. Comienzo del análisis en la Nube.

Cuando el análisis finaliza, en efecto, los logs del Pod nos avisan:

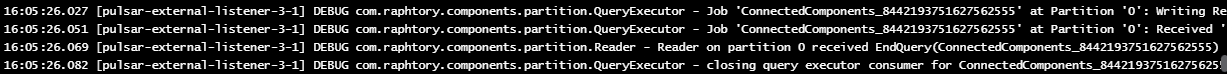


Ilustración 69. Finalización del análisis.

Finalmente, si consultamos Cloud Storage en el panel de Google Cloud, vemos que ha aparecido un fichero con el nombre especificado anteriormente (solo uno, porque hemos configurado una única partición).

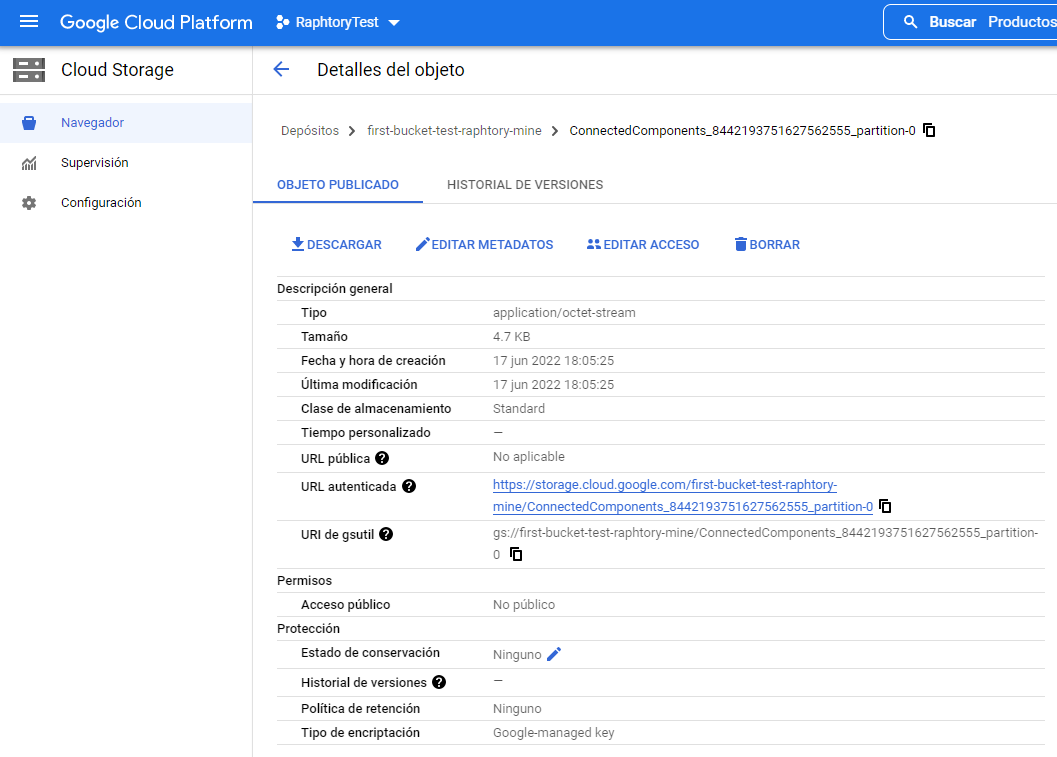


Ilustración 70. Fichero de salida generado en Cloud Storage.

Si lo descargamos, tenemos el resultado del análisis:

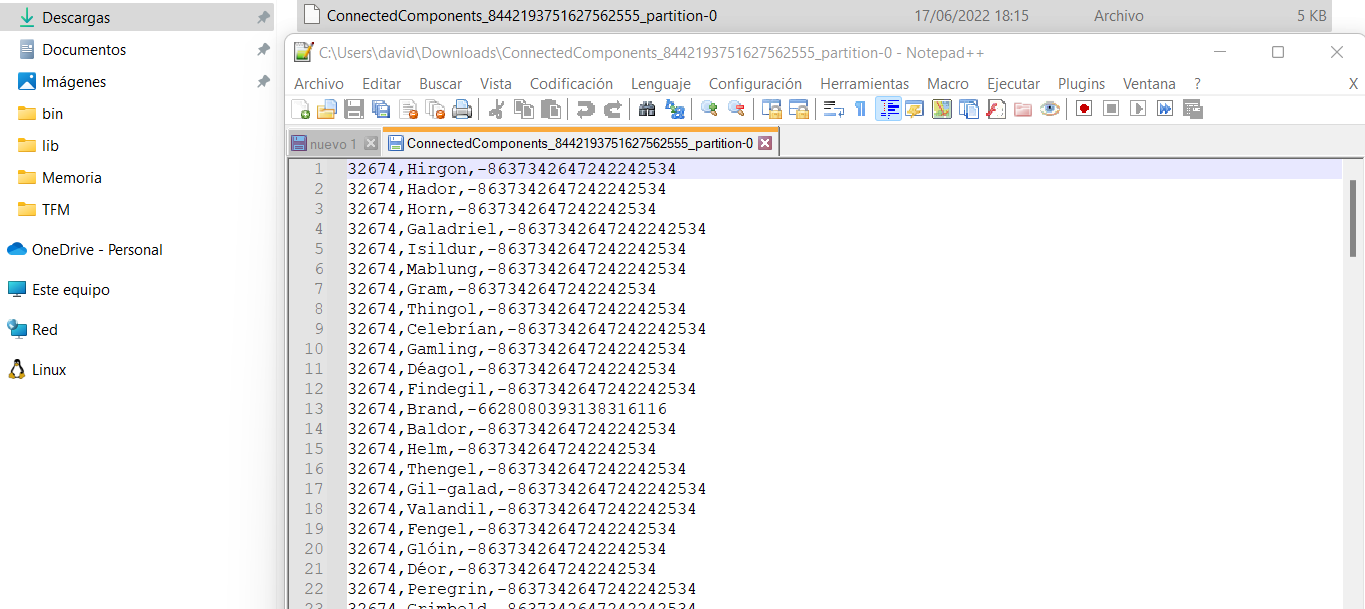


Ilustración 71. Salida del análisis en la Nube.

Por tanto, hemos cumplido el objetivo principal de este Trabajo Fin de Máster, ya que hemos desplegado en Google Cloud un servicio de procesamiento de grafos en la Nube, de forma completamente distribuida, y con un cierto nivel de automatización.

Incluimos a continuación los resultados de una prueba adicional, en la que lanzamos dos particiones, modificando las variables de entorno anteriores apropiadamente (RAPHTORY\_DEPLOY\_KUBERNETES\_DEPLOYMENTS\_ALL\_PODS\_ENV\_RAPHTORY\_PARTITIONS\_SERVERCOUNT = 2; RAPHTORY\_DEPLOY\_KUBERNETES\_DEPLOYMENTS\_PARTITIONMANAGER\_PODS\_REPLICAS = 2).

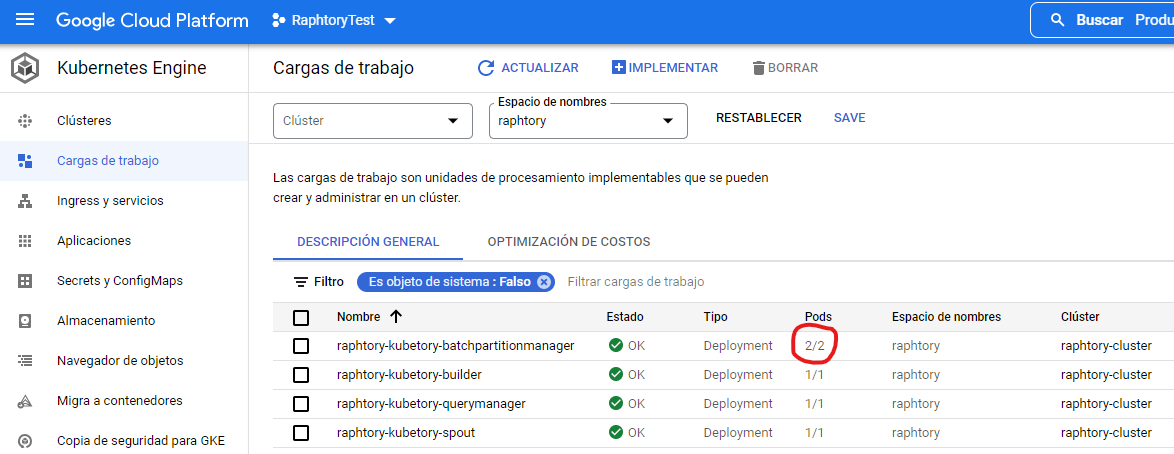


Ilustración 72. Panel de GKE que muestra el despliegue con dos Partition Manager.

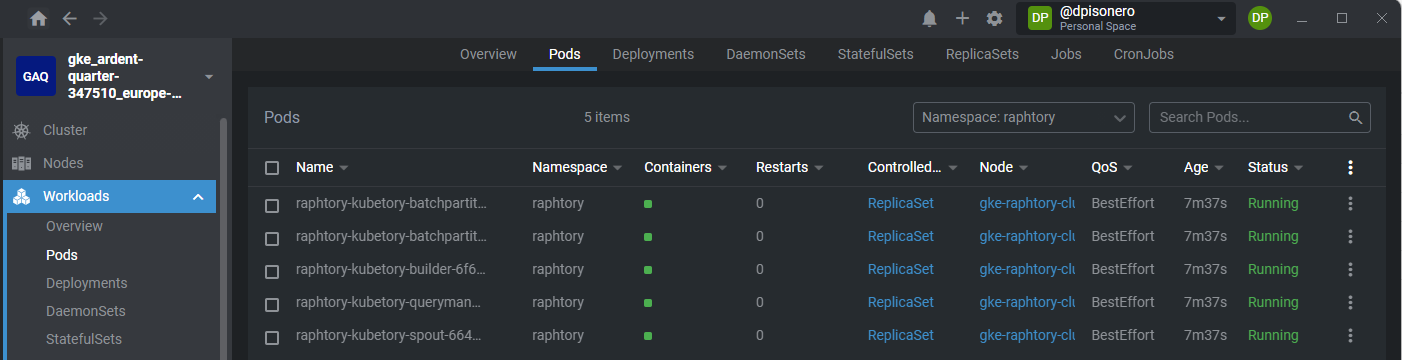


Ilustración 73. Panel de Lens que muestra los dos Partition Manager.

También vemos en logs de Lens, que, en efecto, tenemos las particiones 0 y 1:

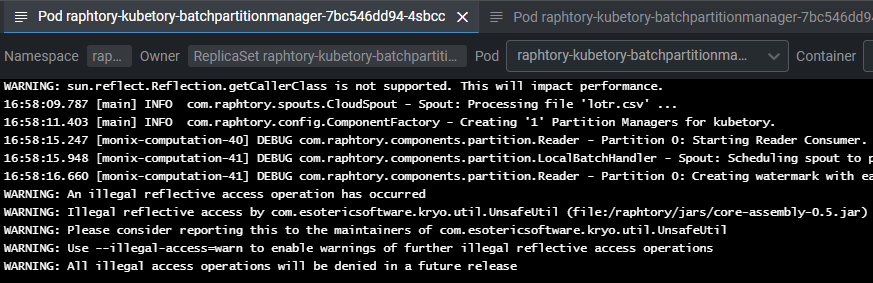


Ilustración 74. Arranque de la Partición 0.

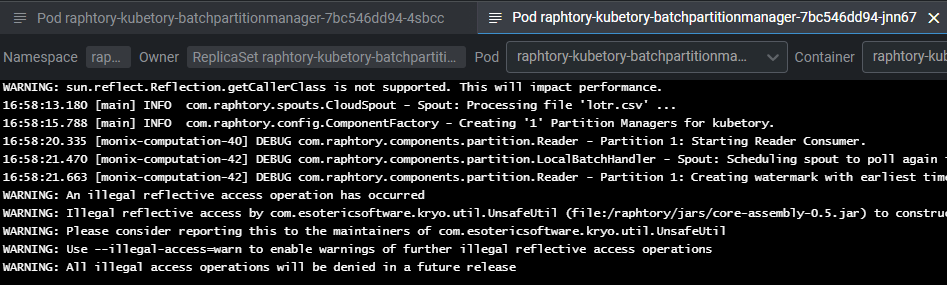


Ilustración 75. Arranque de la Partición 1.

Una vez que ejecutamos el análisis, exactamente igual que antes, vemos que se está trabajando sobre ambas particiones:

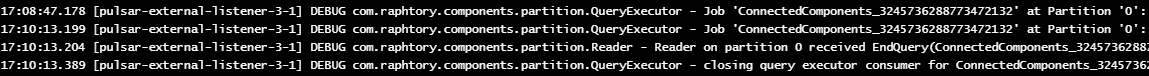


Ilustración 76. Finalización del análisis en la Partición 0.

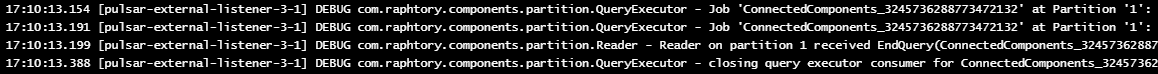


Ilustración 77. Finalización del análisis en la Partición 1.

Al terminar la ejecución de la Query, si consultamos en GCS, vemos que en nuestro bucket han aparecido, en este caso, dos ficheros, uno para cada partición (los dos primeros ficheros de la Ilustración 78, que tienen el mismo identificador de trabajo de análisis, pero uno usa el sufijo “partition-0”, y, el otro “partition-1”).

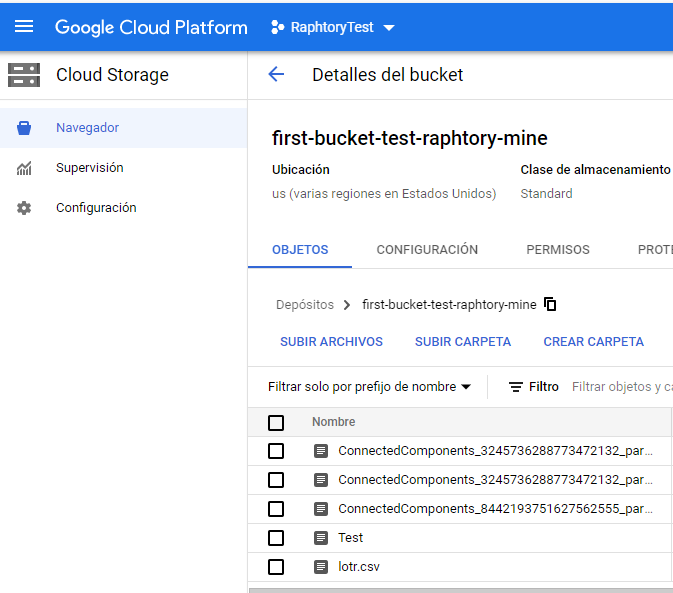


Ilustración 78. Los dos ficheros de salida del análisis ejecutado con dos particiones.

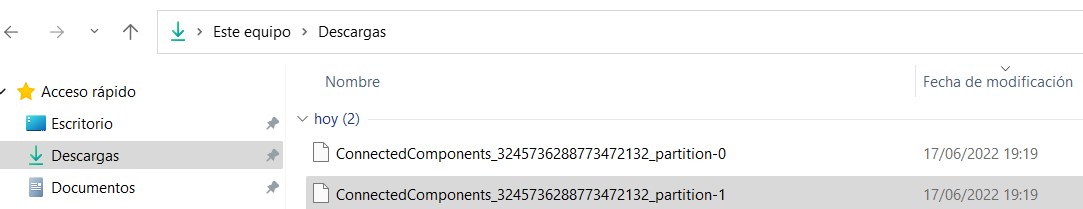


Ilustración 79. Los dos ficheros de salida descargados.

Por tanto, no solo hemos realizado un despliegue de Raphtory como un sistema distribuido en la Nube, sino que, además, hemos comprobado el correcto funcionamiento del procesamiento también distribuido de un grafo en varias particiones.

### Automatización del despliegue con GitHub Actions

El despliegue del BDaaS en la Nube para procesado distribuido de grafos que acabamos de realizar, ya posee un nivel de automatización considerable, gracias al uso de Docker, Helm, Kubernetes, Google Cloud, y diferentes scripts.

Sin embargo, en aras de aumentar el valor añadido del servicio, se ha preparado una automatización global de, al menos, parte del proceso, utilizando para ello GitHub Actions [38].

Así, …

Para finalizar esta sección de Resultados, se indica que todo el código desarrollado de ejemplos, automatización, configuraciones, ficheros de Docker, etc., puede encontrarse en mi GitHub personal, en el repositorio referenciado en [48].

# Conclusiones y líneas futuras

## Conclusiones

En primer lugar, a modo de conclusión, me gustaría resaltar que se han cumplido los objetivos planteados para este Trabajo Fin de Máster.

En efecto, se ha llevado a cabo un estudio teórico sobre el procesado distribuido de grafos y sus paradigmas y técnicas actuales, así como del concepto de análisis temporal, y de diversas tecnologías útiles en este ámbito, desde la computación en la Nube, hasta las herramientas DevOps.

Asimismo, se ha profundizado en el elemento central del TFM, Raphtory, investigando sus partes y funcionamiento, junto con sus capacidades para el procesado distribuido de grafos, y los análisis temporales.

Toda esta información nos ha ido encaminando hacia los siguientes objetivos, ya relacionados con el despliegue de la herramienta. Así, se han desarrollado y estudiado ejemplos con las modalidades básicas de ejecución de Raphtory (la solución Single, y la pseudo distribuida), para, después, ser capaz de implementar un despliegue propio totalmente distribuido y automático con Docker, y Docker Compose.

Finalmente, gracias a este trabajo previo, se ha planteado y desarrollado un despliegue en la Nube, también distribuido, de Raphtory, realizando un estudio extensivo de las diferentes tecnologías y herramientas que han permitido llevarlo a cabo. Este despliegue, además de ser funcional, ha demostrado poseer la capacidad de ejecutar el procesado distribuido de grafos, con cierto nivel de automatización.

También como conclusión, este Trabajo Fin de Máster permite observar las diferencias, al menos a nivel cualitativo, entre las diferentes modalidades de ejecución. Así, la solución Single es la más sencilla de lanzar y comprender, pero, al carecer de las propiedades de los sistemas distribuidos, presenta una pobre escalabilidad.

Por su parte, los despliegues distribuidos sí que tienen esa capacidad para ser más escalables, o conseguir una mayor tolerancia a fallos, pero, como se ha visto, son más complejos, y pueden generar problemas con las comunicaciones.

En mi opinión, la última solución que se presenta, el despliegue BDaaS en la Nube, es la que mayor potencial posee, ya que, aunque es la más compleja, se beneficia de todas las ventajas del Cloud Computing, y permite a un potencial usuario acceder a servicios de procesado de grandes volúmenes de información de manera cómoda, rápida, y a un coste reducido.

Por ende, a modo de resumen del trabajo realizado, se puede concluir que, mediante una fase de estudio y análisis, tanto teórico como práctico, acerca del procesado de grafos y de tecnologías habilitadoras, se ha podido desarrollar y probar un despliegue distribuido en la Nube de un servicio BDaaS de procesado distribuido de grafos, funcional, y con cierto grado de automatización. Además, se puede considerar que es un servicio de gran utilidad y valor añadido, ya que, gracias al trabajo realizado en la Nube, un usuario, sin necesidad de ser un experto, podría acceder al procesado distribuido de grandes cantidades de información, superando barreras de entrada como la necesidad de recursos de computación, el coste, o la complejidad.

Finalmente, querría resaltar que todo este trabajo se ha llevado a cabo en un entorno muy dinámico, conduciéndose prácticamente en paralelo con el desarrollo de la propia herramienta de Raphtory, que, durante estos meses, ha sufrido importantes cambios, como se ha reflejado en la Memoria.

Esto, por supuesto, he generado un reto considerable, ya que, en ocasiones, actualizarte a una nueva versión implica una variación sustancial de los desarrollos realizados, pero también me ha permitido manejarme en un entorno cambiante, y conocer un mayor abanico de conceptos y tecnologías, aptitudes muy útiles de cara al futuro.

## Líneas futuras

Tras realizar este trabajo, a título personal, identifico algunas áreas para la progresión de Raphtory, y del servicio BDaaS desarrollado, de cara al futuro.

En primer lugar, actualmente, cuando se realiza un procesado de grafos en diferentes particiones, cada una de las mismas genera su propio resultado independiente. Creo que merecería la pena llevar a cabo un trabajo de fusión de los resultados, de manera que se aglutinen las salidas de todas las particiones, y en todos los casos el resultado final fuera único.

Este aspecto plantea bastantes retos, ya que debería haber un componente, probablemente el Query Manager, que se ocupara de recibir las salidas de las distintas particiones, y unirlas, lo que puede ser complejo, y empobrecer el rendimiento general de la herramienta.

Otro aspecto importante de evolución en el futuro es la seguridad del despliegue del servicio BDaaS. Que solo se permita el acceso a repositorios remotos mediante una combinación de usuario y contraseña me parece que es algo frágil en términos de privacidad, y te obliga a escribir dicha contraseña en claro, al menos en las variables de entorno.

Su sustitución, o al menos coexistencia, con un sistema de tokens, como el planteado en Artifact Registry, puede proporcionar mayor valor al servicio, garantizando un nivel superior de seguridad para los usuarios.

Finalmente, otra dimensión relevante para evolucionar este desarrollo es la automatización. Aunque gran parte del despliegue se realiza mediante scripts, y herramientas que simplifican parte del trabajo, como Helm y Kubernetes, una automatización global presentaría un potencial muy elevado.

En este caso, se ha apostado por GitHub Actions para, al menos, explorar esta posibilidad, pero queda un gran margen para conseguir un despliegue automático completo, que podría incrementar el valor del servicio drásticamente.

# Bibliografía

|  |  |
| --- | --- |
| [1] | L. Euler, «Solutio problematis ad geometriam situs pertinentis,» *Commentarii Academiae Scientiarum Imperialis Petropolitanae,* nº 8, pp. 128-140, 1736. |
| [2] | C. Godsil y G. Royle, Algebraic Graph Theory, Springer, 2001. |
| [3] | N. Arnold, B. Steer, I. Hafnaoui, H. Parada, R. Mondragón, F. Cuadrado y R. Clegg, «Moving with the Times: Investigating the Alt-Right Network Gab with TemporalInteraction Graphs,» 17 Septiembre 2020. [En línea]. Available: https://www.researchgate.net/publication/344294385\_Moving\_with\_the\_Times\_Investigating\_the\_Alt-Right\_Network\_Gab\_with\_Temporal\_Interaction\_Graphs. [Último acceso: 14 Junio 2022]. |
| [4] | G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser y G. Czajkowski, «Pregel: a system for large-scale graph processing,» 11 Junio 2010. [En línea]. Available: https://research.google/pubs/pub37252/. [Último acceso: 14 Junio 2022]. |
| [5] | Pometry, «Raphtory GitHub,» 2020. [En línea]. Available: https://github.com/Raphtory. [Último acceso: 14 Junio 2022]. |
| [6] | B. Steer, F. Cuadrado y R. Clegg, «Raphtory: Streaming Analysis Of Distributed Temporal Graphs,» *Future Generation Computer Systems,* 2019. |
| [7] | R. Cheng, «Kineograph: taking the pulse of a fast-changing and connected world,» *ACM EuroSys,* pp. 85-98, 2012. |
| [8] | Y. Miao, «Immortalgraph: A system for storage and analysis of temporal graphs,» *ACM TOS,* vol. 11, 2015. |
| [9] | B. Erb, «A distributed processing platform for online and batch computations on event-sourced graphs,» *ACM DEBS,* pp. 77-78, 2017. |
| [10] | Scala, «The Scala Programming Language,» 20 Enero 2004. [En línea]. Available: https://www.scala-lang.org/. [Último acceso: 16 Junio 2022]. |
| [11] | B. Venners, M. Odersky y L. Spoon, «First Steps to Scala,» 9 Mayo 2007. [En línea]. Available: https://www.artima.com/articles/first-steps-to-scala. [Último acceso: 16 Junio 2022]. |
| [12] | NIST, «Computer Security Resource Center,» Septiembre 2011. [En línea]. Available: https://csrc.nist.gov/publications/detail/sp/800-145/final. [Último acceso: 14 Junio 2022]. |
| [13] | DIT, Introducción a los centros de datos, 2018. |
| [14] | M. Loukides, «What is DevOps?,» O'REILLY, 7 Junio 2012. [En línea]. Available: http://radar.oreilly.com/2012/06/what-is-devops.html. [Último acceso: 14 Junio 2022]. |
| [15] | IBM, «¿Qué es el almacenamiento basado en objetos?,» [En línea]. Available: https://www.ibm.com/es-es/cloud/learn/what-is-object-storage. [Último acceso: 16 Junio 2022]. |
| [16] | Amazon Web Services, «Amazon EMR,» 9 Diciembre 2020. [En línea]. Available: https://aws.amazon.com/es/emr/. [Último acceso: 14 Junio 2022]. |
| [17] | Pometry, «Pometry: Realtime graph analytics at scale,» 2021. [En línea]. Available: https://www.pometry.com/. [Último acceso: 15 Junio 2022]. |
| [18] | Pometry, «Running queries across time,» 2022. [En línea]. Available: https://raphtory.readthedocs.io/en/development/Analysis/queries.html. [Último acceso: 15 Junio 2022]. |
| [19] | Akka Team, «Akka: Build powerful reactive, concurrent, and distributed applications more easily,» 2021. [En línea]. Available: https://akka.io/. [Último acceso: 15 Junio 2022]. |
| [20] | Docker, «Docker: Home,» 20 Marzo 2013. [En línea]. Available: https://www.docker.com/. [Último acceso: 16 Junio 2022]. |
| [21] | AdoptOpenJDK Community, «The Community and code behind the Build Farm which produces high quality, FREE OpenJDK (Java) binaries.,» 2017. [En línea]. Available: https://adoptopenjdk.net/. [Último acceso: 16 Junio 2022]. |
| [22] | Docker, «Overview of Docker Compose,» 21 Diciembre 2013. [En línea]. Available: https://docs.docker.com/compose/. [Último acceso: 16 Junio 2022]. |
| [23] | YAML, «The Official YAML Web Site,» 11 Mayo 2001. [En línea]. Available: https://yaml.org/. [Último acceso: 16 Junio 2022]. |
| [24] | Docker, «Swarm mode overview,» 16 Octubre 2014. [En línea]. Available: https://docs.docker.com/engine/swarm/. [Último acceso: 16 Junio 2022]. |
| [25] | Docker, «Docker Hub Container Image Library | App Containerization,» 9 Junio 2014. [En línea]. Available: https://hub.docker.com. [Último acceso: Junio 16 2022]. |
| [26] | Google, «Artifact Registry,» 16 Noviembre 2020. [En línea]. Available: https://cloud.google.com/artifact-registry?hl=es. [Último acceso: 16 Junio 2022]. |
| [27] | Apache Foundation, «Hello from Apache Pulsar,» 2010. [En línea]. Available: https://pulsar.apache.org/. [Último acceso: 16 Junio 2022]. |
| [28] | Apache Foundation, «Apache Pulsar: Conepts and Architecture,» 2018. [En línea]. Available: https://pulsar.apache.org/docs/v2.0.1-incubating/getting-started/ConceptsAndArchitecture/. [Último acceso: 16 Junio 2022]. |
| [29] | Kubernetes, «¿Qué es Kubernetes?,» 21 Julio 2015. [En línea]. Available: https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/. [Último acceso: 16 Junio 2022]. |
| [30] | Minikube, «minikube start,» 2018. [En línea]. Available: https://minikube.sigs.k8s.io/docs/start/. [Último acceso: 16 Junio 2022]. |
| [31] | Google, «Google Cloud: Servicios de Cloud Computing,» 7 Abril 2008. [En línea]. Available: https://cloud.google.com/?hl=es. [Último acceso: 16 Junio 2022]. |
| [32] | Google, «Google Kubernetes Engine,» 2018. [En línea]. Available: https://cloud.google.com/kubernetes-engine?hl=es. [Último acceso: 16 Junio 2022]. |
| [33] | Google, «Persisten Disk,» 2 Diciembre 2013. [En línea]. Available: https://cloud.google.com/persistent-disk?hl=es-419. [Último acceso: 16 Junio 2022]. |
| [34] | Google, «Filestore,» 2008. [En línea]. Available: https://cloud.google.com/filestore?hl=es. [Último acceso: 16 Junio 2022]. |
| [35] | Google, «Cloud Storage,» 19 Mayo 2010. [En línea]. Available: https://cloud.google.com/storage?hl=es. [Último acceso: 16 Junio 2022]. |
| [36] | Cloud Native Computing Foundation, «Helm: The package manager for Kubernetes,» Noviembre 2015. [En línea]. Available: https://helm.sh/. [Último acceso: 16 Junio 2022]. |
| [37] | GitLab Inc., «GitLab, The One DevOps Platform,» 2014. [En línea]. Available: https://about.gitlab.com/. [Último acceso: 16 Junio 2022]. |
| [38] | GitHub, «GitHub Actions,» 13 Noviembre 2019. [En línea]. Available: https://docs.github.com/es/actions. [Último acceso: 16 Junio 2022]. |
| [39] | Google, «Cloud SDK - Libraries and Command Line Tools,» 8 Abril 2008. [En línea]. Available: https://cloud.google.com/sdk?hl=es. [Último acceso: 16 Junio 2022]. |
| [40] | Lightbend, Inc, «sbt - The interactive build tool,» 18 Diciembre 2008. [En línea]. Available: https://www.scala-sbt.org/. [Último acceso: 17 Junio 2022]. |
| [41] | SDKMAN!, «SDKMAN! - The Software Development Kit Manager,» 2016. [En línea]. Available: https://sdkman.io/. [Último acceso: 17 Junio 2022]. |
| [42] | IDEA, «IntelliJ - IDE para JVM eficaz y ergonómico,» Enero 2001. [En línea]. Available: https://www.jetbrains.com/es-es/idea/. [Último acceso: 17 Junio 2022]. |
| [43] | Akka, «Message Delivery Reliability,» [En línea]. Available: https://doc.akka.io/docs/akka/current/general/message-delivery-reliability.html. [Último acceso: 17 Junio 2022]. |
| [44] | Pometry, «Raphtory Algorithms,» 2022. [En línea]. Available: https://raphtory.readthedocs.io/en/development/\_autodoc/com/raphtory/algorithms/generic/index.html. [Último acceso: 17 Junio 2022]. |
| [45] | Alpine Linux Development Team, «Alpine Linux - Small, Simple, Secure,» Agosto 2005. [En línea]. Available: https://www.alpinelinux.org/. [Último acceso: 17 Junio 2022]. |
| [46] | Mirantis, Inc, «Lens - The Kubernetes IDE,» 2021. [En línea]. Available: https://k8slens.dev/. [Último acceso: 17 Junio 2022]. |
| [47] | Pometry, «Connected Components,» 2022. [En línea]. Available: https://raphtory.readthedocs.io/en/development/\_autodoc/com/raphtory/algorithms/generic/ConnectedComponents.html. [Último acceso: 17 Junio 2022]. |
| [48] | D. Pisonero Fuentes, «GitHub Raphtory Cloud,» Junio 2022. [En línea]. Available: https://github.com/dpisonero/raphtory\_cloud. [Último acceso: 17 Junio 2022]. |

# Anexo A: aspectos éticos, económicos, sociales y ambientales

## A.1 Introducción

Este Trabajo Fin de Máster tiene como objetivo ser capaz de desarrollar una solución de analítica como servicio en la Nube, para que un usuario potencial pueda acceder al procesado distribuido de grafos sorteando diferentes barreras de entrada, como la necesidad de potentes y costosos recursos de computación, o la complejidad de esta actividad.

Por tanto, es un TFM que se encuadra en el marco del sector TIC (Tecnologías de la Información y la Comunicación), fundamentalmente, en el ámbito de los sistemas telemáticos. Sin embargo, debido a la finalidad del mismo, puede tener un efecto notable en ámbitos organizativos mucho más amplios, pues el servicio que se ha comenzado a desarrollar aquí, con la conveniente evolución, podría ser utilizado por usuarios y organizaciones de toda índole, con fines muy diversos.

Así, surgen numerosos grupos de interés que podrían beneficiarse de este trabajo, ya que facilita el acceso al procesado de grandes cantidades de información, en forma de grafo, reduciendo el impacto socio-económico, puesto que ofrece el servicio a un coste reducido gracias a la Nube, y con un grado de automatización que lo simplifica considerablemente, permitiendo que personas no expertas puedan acceder a él.

Por ejemplo, el procesado distribuido de grafos puede ser útil en estudios sociométricos, para analizar el comportamiento de las personas en una red social [3], a nivel humanitario, en la investigación de la propagación de enfermedades infecciosas, o a nivel ambiental, analizando las interacciones de los seres vivos en un ecosistema.

## A.2 Descripción de impactos relevantes relacionados con el proyecto

Tras analizar el objetivo de este TFM, se pueden identificar numerosos impactos de gran relevancia en diferentes ámbitos de la sociedad y la Naturaleza.

Al final, el desarrollo de un servicio que permita acceder a cualquier persona o grupo al procesado de grandes volúmenes de información en forma de grafo, puede representar un impacto notable en sus actividades.

Así, por ejemplo, a nivel social, este trabajo podría contribuir a realizar estudios muy amplios sobre el comportamiento de las personas, o sobre sus desplazamientos por un territorio, habilitando un mejor conocimiento de nuestra sociedad.

A nivel de Ciencias de la Salud, como se ha comentado anteriormente, el servicio desarrollado podría facilitar el análisis de enfermedades infecciosas, contribuyendo, en el caso de los expertos en este tipo de dolencias, a simplificar y agilizar sus investigaciones, y, ya a nivel general, mejorar la salud y la protección de todas las personas.

Asimismo, en el ámbito comercial y económico, el acceso en la Nube a un servicio BDaaS puede acelerar muchos tipos de análisis econométricos, lo que puede contribuir a una mejora del ecosistema empresarial, y a una mayor comprensión de nuestra economía, lo que redundaría en un incremento del nivel de vida de los ciudadanos.

Incluso, en este ámbito, también se podría considerar la eficiencia del uso de recursos en la Nube, lo que genera un beneficio económico, pero también ecológico, al consumir menos electricidad, comparado con el despliegue de numerosos servicios en local con recursos no compartidos.

Por ende, podemos concluir que el desarrollo de una herramienta de este tipo puede tener efectos beneficiosos en diferentes ámbitos, como el social, el humanitario, y redundar en el progreso de diferentes grupos de interés afectados, como las empresas, las personas enfermas, o el grueso social de un territorio.

## A.3 Análisis detallado de algunos de los principales impactos

Como se ha comentado anteriormente, por la naturaleza de este Trabajo Fin de Máster, existe multitud de impactos, en diferentes ámbitos, que se pueden tener en cuenta. A continuación, se recoge alguno de los más destacados.

#### Impacto en las Ciencias de la Salud

Actualmente, con la pandemia de COVID-19, se ha puesto de manifiesto la importancia que tiene el control de enfermedades infecciosas que pueden propagarse entre los ciudadanos de un territorio.

Poseer una herramienta que pueda permitir el seguimiento de las infecciones, las interacciones que extienden la enfermedad, o los desplazamientos de las personas, con los contagios que ello acarrea, puede ser de vital utilidad.

El servicio BDaaS desarrollado podría tener un impacto especialmente positivo en la realización de estas actividades, ya que permitiría a cualquier organización médica que posea los datos necesarios, y tenga acceso a un servicio de computación en la nube, procesar de forma rápida, económica, y eficiente, toda esa información, extrayendo nuevos datos y conclusiones que redundarían en la toma de decisiones acertadas en el control de este tipo de enfermedades.

En definitiva, se podrían alcanzar numerosos beneficios en la salud de las personas, previniendo y mitigando la propagación de infecciones.

#### Impacto en términos sociales

Otra área en la que se ha identificado un impacto notable que podría provocar este tipo de servicios BDaaS accesibles en la Nube es la sociología.

En los últimos tiempos, el estudio del comportamiento y las interacciones entre personas ha cobrado una gran importancia en la sociedad, especialmente con todas las tecnologías de interconexión que existen, como, por ejemplo, las redes sociales.

De este modo, una herramienta de procesado distribuido de grafos (que pueden representar interacciones) como Raphtory, podría ser de enorme utilidad para este tipo de análisis, especialmente si es accesible para cualquiera, desde la Nube, sorteando así las barreras económicas, y de complejidad, que surgen de manera natural.

Finalmente, aunque es cierto que este clase de estudios puede redundar en un mayor conocimiento de nuestra sociedad, y, por ende, funcionar como un impulsor del progreso, también surge cierto dilema ético, ya que estas tecnologías que permiten el procesado de grandes volúmenes de información que, en este ámbito, pueden habilitar el diseño de medidas de control social y poblacional, que dañarían enormemente la libertad de los ciudadanos, produciéndose así un impacto negativo.

## A.4 Conclusiones

Por último, a modo de conclusión, es notable resaltar que este tipo de herramientas pueden generar un impacto destacable en muchos ámbitos, en general positivo.

Además, el uso de criterios de sostenibilidad, que tan presentes están hoy en día en el uso de grandes volúmenes de recursos de computación en la Nube, podría aportar un valor añadido al proyecto, reforzando los beneficios ambientales y ecológicos derivados de la reducción en el consumo de materiales y electricidad.

En definitiva, el desarrollo de un servicio BdaaS en la Nube, que es el objetivo de este TFM, puede influir en ámbitos como la sociología, con el análisis de las relaciones entre personas, en la Salud, con el estudio de la propagación de enfermedades infecciosas (como el contexto de pandemia actual), en la economía, con la reducción de costes y las investigaciones econométricas, e incluso en el Medio Ambiente, gracias a la eficiencia ecológica de los recursos compartidos.

Por tanto, es un proyecto que facilita la eliminación de las barreras tradicionales que surgen para este tipo de actividades, que implican el procesado de grafos masivos, y que puede generar numerosos beneficios de diversa índole en la sociedad.

# Anexo B: presupuesto económico

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **COSTE DE MANO DE OBRA (coste directo)** | | | | | **Horas** | | **Precio/hora** | **Total** |
|  | | | | | 750 | | 12 € | **9.000 €** |
|  | | | |  |  | |  |  |
| **COSTE DE RECURSOS MATERIALES (coste directo)** | | | | **Precio de compra** | **Uso en meses** | | **Amortización (en años)** | **Total** |
| Ordenador personal (Software incluido) | | | | 1.350,00 € | 10 | | 5 | 225,00 € |
| Monitor | | | | 400,00 € | 10 | | 5 | 66,67 € |
| Crédito Google Cloud | | | | 288,73 € | 2 | | 5 | 9,62 € |
|  | | | |  |  | |  |  |
| **COSTE TOTAL DE RECURSOS MATERIALES** | | | | | | | | **286,45 €** |
|  | |  | |  | |  | |  |
| **GASTOS GENERALES (costes indirectos)** | | 15% | | sobre CD | | | | **1392,97 €** |
| **BENEFICIO INDUSTRIAL** | | 6% | | sobre CD+CI | | | | **640,77 €** |
|  | |  | |  | |  | |  |
|  |  | |  | | |  | |  |
| **SUBTOTAL PRESUPUESTO** | | | | | | | | **11.320,19 €** |
| **IVA APLICABLE** | | | | | | | 21% | **2.377,24 €** |
|  |  | |  | | | |  |  |
| **TOTAL PRESUPUESTO** | | | | | | | | **13.697,43 €** |