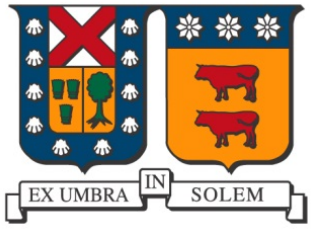
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE



“SUGERENCIAS PARA LA CONSTRUCCIÓN DE CONSULTAS SPARQL, UNA ALTERNATIVA OPEN SOURCE”

CARLOS PONCE

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Carlos Buil
Profesora Correferente: Cecilia Reyes

Enero - 2022

DEDICATORIA

Dedicado a mi familia, amigos y todos quienes me rodean día a día.
Sin ustedes, no hubiera logrado superar este increíble desafío.

AGRADECIMIENTOS

Antes que todo quiero agradecer a mi familia, por el apoyo incondicional durante estos años de estudio, sé que siempre estarán ahí para mí.

Agradecer también a mis amigos, en especial al “Grupito de Atrás’’, por estar siempre a mi lado en las buenas y en las malas durante los 6 años que hemos estado en la Universidad, si bien no todos seguimos aquí, es imposible olvidar todo lo que hemos pasado juntos, todos esos momentos sacando la vuelta en clases, aquellas incontables tardes e incluso noches estudiando en el Laboratorio de Computación y esos certámenes reprobados para luego volver a encontrarnos en la misma asignatura el semestre siguiente y continuar con nuestras estupideces. Cuando ingresas a la carrera lo primero que te recomiendan es conseguirte un grupo de estudio, te dicen que va a ser difícil solo, que la vas a pasar mal sin amigos y creo que después de todo este tiempo no me queda más que reafirmar esto. Definitivamente no lo hubiera logrado sin ustedes. Gracias por todo.

No quiero dejar de lado al increíble equipo del Laboratorio de Computación, si bien hoy en día solo quedamos el Profesor Horst y yo como miembros del Laboratorio de aquella generación que me recibió en mi segundo año, el 2017, se que quienes pertenecen hoy y quienes vendrán harán un excelente trabajo como miembros que apoyan a la comunidad del Departamento. La mayoría de los que me recibieron y acompañaron ya se titularon, otros están haciendo postgrados en otros países y algunos, al igual que yo, terminando su proceso de titulación. Fue en este lugar y junto a las increíbles personas que se reunían aquí donde pase la mayor parte del tiempo fuera de las clases, donde estudiábamos, descansábamos, nos desahogábamos, dormíamos y jugábamos, fue nuestro espacio en nuestro segundo hogar. Además de estas experiencias, en este lugar logre conocer en primera persona dos de las especialidades de la informática: el mundo del desarrollo de software y la administración de sistemas. Todo lo que he logrado aprender aquí es invaluable.

Finalmente, quiero agradecer el cuerpo docente del Departamento, en especial a nuestra Subdirectora de Pregrado, Cecilia Reyes, por estar siempre disponible y motivada para apoyar a los alumnos durante toda su estadía en la Universidad y al Profesor Horst von Brand, por construir, apoyar y liderar el increíble grupo humano llamado Laboratorio de Computación. Gracias por apoyar a esta increíble comunidad.

RESUMEN

Resumen—En este trabajo, se presenta el proceso de optimización del rendimiento de una aplicación monolítica, comenzando por un análisis de sus funcionalidades, diseño e implementación, continuando con un rediseño y reimplementación de sus procesos, para finalizar con un conjunto de mejoras y pruebas en múltiples ambientes de ejecución en una nube pública, todo esto, realizando un conjunto de iteraciones enfocadas a la mejora continua de nuestros resultados, utilizando la metodología ágil *PDCA*. Como resultado, hemos logrado diseñar e implementar una solución orientada a microservicios que permite reducir en un $97\,\%$ el tiempo de ejecución necesario para obtener resultados desde la aplicación y, que además, logra actualizar en tiempo real su base de datos con nueva información publicada directamente en nuestra fuente de datos, *Wikidata*.

Palabras Clave—SPARQL; RDF; Web Semántica; Autocompletado; Rendimiento.

ABSTRACT

***Abstract*—**In this work, we present our process to optimize the performance of an existing monolithic application, starting with an analysis of its functionalities, design and implementation, following with a redesign and reimplementation of its processes, and ending with a set of improvements and tests in multiple execution environments in a public cloud, throught the performance of a set of iterations focused on the continuous improvement of our results using the agile PDCA methodology. As a result, we have managed to design and implement a microservices oriented solution that allows us to reduce by $97\,\%$ the execution time required to obtain results from the application and can update its database in real time with new information published directly to our data source, Wikidata.

Keywords*—SPARQL*; *RDF*; *Semantic Web*; *Autocompletion*; *Performance*.

GLOSARIO

API: *Application Programming Interface*. Interfaz de Programación de Aplicaciones.
AWS: *Amazon Web Services*. Nube pública de Amazon.
CPU: *Central Processing Unit*. Unidad Central de Procesamiento.
EBS: *Elastic Block Store*. Almacenamiento Elástico de Bloques, Servicio de almacenamiento en *AWS*.
EC2: *Elastic Compute Cloud*. Nube de Computo Elástica, Servicio de computo en *AWS*.
FIFO: *First In, First Out*. Primero en Entrar, Primero en Salir.
GIL: *Global Interpreter Lock*. Seguro Global del Interprete.
HDD: *Hard Drive Disk*. Unidad de Disco Duro.
HTTP: *HyperText Transport Protocol*. Protocolo para la Transferencia de Hipertexto.
HTTPS: *HyperText Transport Protocol Secure*. Protocolo Seguro para la Transferencia de Hipertexto.
IRI: *Internationalized Resource Identifier*. Identificador Internacionalizado de Recursos.
JDK: *Java Development Kit*. Kit de Desarrollo en *Java*.
JFR: *JDK Flight Recorder*. Grabador de Eventos para la *JVM*.
JSON: *JavaScript Object Notation*. Notación de Objeto de JavaScript.
JVM: *Java Virtual Machine*. Maquina Virtual *Java*.
MITM: *Man-in-the-middle*. Ataque de Intermediario.
MOM: *Message Oriented Middleware*. Intermediario Orientado a Mensajes.
NVME: *Non-Volatile Memory Host Controller Interface Specification*. Especificación de Interfaz de Controlador de Host de Memoria No Volátil.
OPM: *The Open Provenance Model*. Modelo de Procedencia Libre.
OWL: *Web Ontology Language*. Lenguaje de Ontologías para la *Web*.
PDCA. *Plan, Do, Check, Act*. Planear, Hacer, Verificar, Actuar.
RAM: *Random Access Memory*. Memoria de Acceso Aleatorio.
RDF: *Resource Description Framework*. Marco de Descripción de Recursos.
RDFS: *Resource Description Framework Schema*. Esquema para el Marco de Descripción de Recursos.
RIF: *Rule Interchange Format*. Formato de Intercambio de Reglas.
SWRL: *Semantic Web Rule Language*. Lenguaje de Reglas para la *Web* Semántica
SATA: *Serial Advanced Technology Attachment*. Tecnologia de Conexión Serial Avanzada.
SFH: *SPARQL For Humans*.
SPARQL: *SPARQL Protocol And Query Language*. Lenguaje de Consultas y Protocolo *SPARQL*.
SPOF: *Single Point Of Failure*. Punto Único de Falla.
SSD: *Solid State Drive*. Disco de Estado Solido.
URI: *Uniform Resource Identifier*. Identificador de Recursos Uniforme.
W3C: *World Wide Web Consortium*. Consorcio de la Red Informática Mundial.
WQS: *Wikidata Query Service*. Servicio de Consultas de *Wikidata*.
WWW: *World Wide Web*. Consorcio de la Red Informática Mundial.
XML: *Extensible Markup Language*. Lenguaje de Marcado Extensible.

ÍNDICE DE CONTENIDOS

RESUMEN IV

ABSTRACT IV

GLOSARIO V

ÍNDICE DE FIGURAS IX

ÍNDICE DE TABLAS XI

INTRODUCCIÓN 1

CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA 2
1.1 La *Web* Semántica, RDF y SPARQL 2
1.2 Dificultades al utilizar SPARQL 3
1.3 Generación de sugerencias 5
1.4 Objetivos 6
1.4.1 Objetivo general 6
1.4.2 Objetivos específicos 6

CAPÍTULO 2: MARCO CONCEPTUAL 7
2.1 *Web* Semántica 7
2.2 Arquitectura de la *web* semántica 7
2.2.1 Referencias, transporte y principios de los datos enlazados 8
2.2.2 Intercambio de datos 9
2.2.3 Consultas y actualizaciones 10
2.2.4 Ontologías y razonamiento 13
2.2.5 Reglas 14
2.2.6 Seguridad y encriptación 14
2.2.7 Unificación e integración 14
2.2.8 Confianza 15
2.2.9 Aplicaciones 16
2.3 Generación de sugerencias 18
2.3.1 Indexación de documentos 18
2.3.2 *Ranking* de resultados 19
2.4 Visualizaciones 20
2.4.1 *RDFExplorer* 20
2.4.2 *SPARQLforHumans* 20

CAPÍTULO 3: PROPUESTA DE SOLUCIÓN 22
3.1 Metodología de trabajo 22
3.2 Plan de trabajo 23

3.3 Primera iteración: Análisis inicial de la solución existente . . . 24
3.4 Segunda iteración: Mejoras al rendimiento general . . . 25
3.5 Tercera iteración: Múltiples componentes, *FlatBuffers* y *Message Brokers* . . . 35
3.5.1 Procesamiento de texto . . . 35
3.5.2 Serialización binaria . . . 36
3.5.3 Generación de grupos en formato *FlatBuffers* . . . 42
3.5.4 PageRank . . . 42
3.5.5 *Message Brokers* . . . 51
3.5.6 Actualización del valor *PageRank* . . . 53
3.5.7 Construcción de índices . . . 53
3.6 Cuarta iteración: Microservicios, paralelización y actualizaciones en tiempo real 57
3.6.1 *RabbitMQ* . . . 58
3.6.2 Arquitectura orientada a eventos . . . 59
3.6.3 Actualización en tiempo real . . . 62
3.6.4 Rendimiento de instancias *AWS* . . . 64
3.6.5 Tiempos obtenidos . . . 67
3.6.6 Actualizaciones . . . 67

CAPÍTULO 4: VALIDACIÓN DE LA SOLUCIÓN . . . 69
4.1 Diseño e implementación final de la solución . . . 69
4.2 Despliegue de la solución . . . 70
4.3 Pruebas unitarias . . . 71
4.4 Comparación . . . 73
4.5 Costos de la solución . . . 74

CAPÍTULO 5: CONCLUSIONES . . . 76
5.1 Discusión . . . 76
5.1.1 Optimizaciones . . . 77
5.1.2 *Time-space complexity trade-off* . . . 77
5.1.3 *Lucene* . . . 77
5.2 Trabajo futuro . . . 78
5.2.1 Nuevas tecnologías . . . 78
5.2.2 Mejoras al rendimiento . . . 78
5.2.3 Nuevas funcionalidades . . . 79
5.2.4 Despliegue de la aplicación . . . 79
5.2.5 Interacción con el usuario . . . 80

ANEXOS . . . 81
5.1 Reporte de *Syscalls* para estrategias de construcción de grafos. . . . 81
5.1.1 Estrategia SingleThread . . . 81
5.1.2 Estrategia ThreadedReader . . . 82
5.1.3 Estrategia MultiProcess . . . 84
5.1.4 Estrategia MultiProcessZMQ . . . 86

REFERENCIAS BIBLIOGRÁFICAS 89

ÍNDICE DE FIGURAS

1 Un grafo *RDF* básico. 2

2 Un grafo que representa un patrón complejo. 3

3 Consulta *SPARQL* generada en base al grafo incompleto de la figura 2. 4

4 RDFExplorer, construcción de una consulta *SPARQL*. 5

5 Arquitectura de la *web* semántica. 8

6 Descripción de un documento *RDF/XML*. 10

7 Descripción de un documento *RDF/Turtle*. 11

8 Patrón de un grafo *RDF* básico. 11

9 Una consulta *SPARQL* realizada al servicio Wikidata. 12

10 Ejemplo de un proceso descrito utilizando *OPM*. 16

11 Consulta *SPARQL* realizada en el *Wikidata Query Service*. 17

12 Visualización de una consulta *SPARQL* en el *Wikidata Query Service*. 17

13 Ejemplos de un *forward index* y un *inverse index*. 19

14 Arquitectura del sistema *SPARQLforHumans* 21

15 Ciclos *PDCA*. 22

16 Proceso de arranque de la aplicación *SPARQLforHumans*. 26

17 Diagrama de componentes para la aplicación *SPARQLforHumans*. 27

18 Diagrama de arquitectura para la aplicación *SPARQLforHumans*. 28

19 Reporte generado por *JFR*. 32

20 Reporte generado por *Async-Profiler*. 32

21 Generación de objetos `CustomRDFTriple`. 33

22 Diagrama de arquitectura para la solución. 34

23 Ejemplo de serialización en formato *JSON*. 37

24 Operaciones en el proceso de serialización *JSON* vs *Protobuf* vs *FlatBuffers*. . . 39

25 Operaciones en el proceso de deserialización *JSON* vs *Protobuf* vs *FlatBuffers*. . 39

26 Memoria utilizada en el proceso de serialización *Protobuf* vs *FlatBuffers*. 40

27 Memoria utilizada en el proceso de deserialización *Protobuf* vs *FlatBuffers*. . . 40

28 Velocidad y tamaño de mensajes generados *JSON* vs *Protobuf* vs *FlatBuffers*. . . 40

29 Esquema utilizado para la serialización en formato binario a través de *FlatBuffers* 41

30 Diagrama de arquitectura para la solución. 42

31 Estrategia de múltiples hilos. 45

32 Estrategia de múltiples procesos. 46

33 Tiempo de *CPU* por *syscall* para la estrategia *SingleThread* 48

34 Total de llamadas por *syscall* para la estrategia *SingleThread* 48

35 Tiempo de *CPU* por *syscall* para la estrategia *ThreadedReader* 49

36 Total de llamadas por *syscall* para la estrategia *ThreadedReader* 49

37 Tiempo de *CPU* por *syscall* para la estrategia *MultiProcess* 50

38 Total de llamadas por *syscall* para la estrategia *MultiProcess* 50

39 Estrategia de múltiples procesos y ZMQ. 54

40 Tiempo de *CPU* por *syscall* para la estrategia *MultiProcessZMQ* 55

41 Total de llamadas por *syscall* para la estrategia *MultiProcessZMQ* 56

42 Diagrama de arquitectura para la solución. 56

43 Diagrama de arquitectura orientado a microservicios para la solución. 57

44 Interacción entre componentes a través de la ejecución de la aplicación 61

45 Evento obtenido desde *EventStreams* en formato *JSON* 63

46 Reducción inesperada del rendimiento en *broker* 64

47 Utilización de *CPU* y créditos de *CPU* en *AWS* para la instancia `t3.2xlarge` . . 65

48 Rendimiento en *broker* utilizando una instancia *m5.4xlarge* 66
49 Rendimiento de la instancia *m5.4xlarge* 66
50 Tiempo de ejecución en distintas instancias 72
51 Costo total de ejecución en distintas instancias 74
52 Costo total y tiempo de ejecución en distintas instancias 75

ÍNDICE DE TABLAS

1 Utilización de recursos en el ambiente de desarrollo. 30
2 Rendimiento de herramientas para la manipulación de grafos en *Python*. 43
3 Tiempos en *CPU* para distintas estrategias de construcción de grafos. 47
4 Tiempos en *CPU* para distintas estrategias de construcción de grafos. 53
5 Tiempos obtenidos en ambiente productivo. 67
6 Cantidad de notificaciones procesadas desde *Kafka* 68
7 Tiempos necesarios para procesar la información de *Wikidata* 73
8 Tiempos obtenidos en múltiples tipos y tamaños de instancias 73
9 Costo por hora según tipo de instancia en *AWS* 75
10 Reporte generado por `strace` para la estrategia SingleThread. 82
11 Reporte generado por `strace` para la estrategia ThreadedReader. 84
12 Reporte generado por `strace` para la estrategia MultiProcess. 86
13 Reporte generado por `strace` para la estrategia MultiProcessZMQ. 88

INTRODUCCIÓN

SPARQL (del inglés *SPARQL Protocol And RDF Query Language*) es un lenguaje de consultas para bases de datos orientadas a grafos en formato *RDF* (del inglés *Resource Description Framework*), el cual, es complejo de utilizar para aquellos usuarios que no tienen experiencia con conceptos de la *Web* semántica y sus tecnologías. Para facilitar la adopción de *SPARQL* se han desarrollado múltiples herramientas orientadas a la experiencia de sus usuarios, una de ellas es *RDFExplorer*, la cual, en conjunto con el proyecto *SPARQLforHumans* busca facilitar aún más esta adopción a través de una interfaz *web* que permite explorar los conjuntos de datos disponibles en el repositorio *RDF* sobre el cual se está trabajando, presentando al usuario recomendaciones de consultas que puede realizar.

Dicho esto, nuestro trabajo se enfoca en mejorar el rendimiento de *SPARQLforHumans*, puesto que hemos logrado identificar dos oportunidades importantes que permitirán mejorar la experiencia de los usuarios al utilizar esta aplicación.

La primera de ellas, corresponde a acelerar el proceso inicial de generación de resultados en base a una entrada de datos desde el repositorio *RDF Wikidata*, el cual actualmente demora bastante. La segunda oportunidad contempla actualizar los datos procesados de forma constante una vez que el proceso inicial ha finalizado, considerando que el repositorio de *Wikidata* puede llegar a ser actualizado miles de veces en un solo día [Wikipedia, 2022] creemos que es importante mantener actualizada nuestra propia base de datos para que el usuario siempre pueda interactuar con los datos más cercanos a los del repositorio fuente.

Si bien la integración entre *SPARQLforHumans* y *RDFExplorer* existe hoy en día, no es parte de nuestro alcance mejorar, modificar o hacernos cargo de esta integración. Hemos definido que solo nos centraremos en las mejoras aplicables a *SPARQLforHumans*.

Debido a la incertidumbre que existe sobre cuáles son las mejoras a realizar, el impacto de estas y cuál será nuestro resultado final, hemos decidido utilizar alguna de las metodologías ágiles actualmente existentes.

Por último, comentar sobre la estructura de este documento. En el primer capítulo presentamos una definición formal de nuestro problema a resolver, junto con nuestros objetivos específicos y el objetivo general, luego, en el capítulo dos exhibimos el marco conceptual sobre el cual se encuentran las aplicaciones que buscamos intervenir.

Más adelante, en el capítulo tres exponemos nuestra metodología de trabajo, plan de trabajo y nuestra propuesta de solución, después, mostramos nuestro proceso de validación de la solución seleccionada en el capítulo cuatro para finalmente, en el capítulo cinco, presentar nuestras conclusiones, junto con una discusión del trabajo realizado y un conjunto de propuestas y temas para un trabajo futuro.

CAPÍTULO 1
DEFINICIÓN DEL PROBLEMA

1.1. La *Web* Semántica, RDF y SPARQL

La *Web* semántica es un conjunto de extensiones para la *World Wide Web*, desarrolladas por el *World Wide Web Consortium (W3C)*, las cuales buscan permitir que la información disponible en *internet* pueda ser procesada de forma eficiente y automática por máquinas [Berners-Lee et al., 2001]. Para lograr esto, la *web* semántica propone agregar a la información ya existente metadatos semánticos para que estos puedan ser procesados por agentes inteligentes, los cuales, obtendrán esta información sin operadores humanos. Estos metadatos pueden ser descritos utilizando múltiples estándares, uno de ellos es el *Resource Description Framework (RDF)* [W3C et al., 2014].

RDF nos permite describir relaciones a través de *triples* del tipo "*Sujeto - Predicado - Objeto*", los cuales, pueden ser representados de manera visual a través de grafos. Un ejemplo de esto se puede apreciar en la figura 1. En el grafo, el sujeto y el objeto son representados por vértices y el predicado, la relación entre ellos, se representa con una arista.

Cada uno de los elementos del *triple* corresponde a un objeto en la *web*, los cuales, son identificados por Identificadores de Recursos Uniforme[1]. Un ejemplo de estas *URIs* es http://www.wikidata.org/entity/Q1 la cual identifica a "El Universo" en el repositorio de datos *Wikidata*. Sin embargo, existen ocasiones en las que no buscamos definir todo el grafo, sino que también podemos dejar determinados vértices o aristas en blanco, incógnitas, de forma tal que el grafo que estamos construyendo sea uno incompleto. Esto introduce el concepto de una variable en un grafo incompleto.

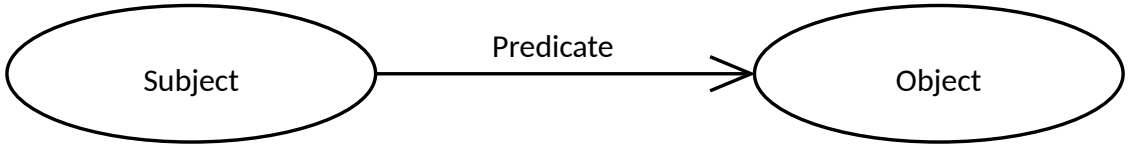


Figura 1: Un grafo *RDF* básico.
Los nodos *Subject* y *Object* están conectados a través de la relación *Predicate*. Fuente: *RDF 1.1 Concepts and Abstract Syntax*. *World Wide Web Consortium*.

Un grupo de múltiples *triples* nos permite construir grafos más complejos, los cuales, en conjunto con vértices variables nos permiten definir patrones de grafos, los cuales, al asignar valores concretos en sus variables generan una solución al patrón representado. Un ejemplo de estos patrones se puede apreciar en la figura 2.

[1]Del ingles *Uniform Resource Identifiers (URIs)*.

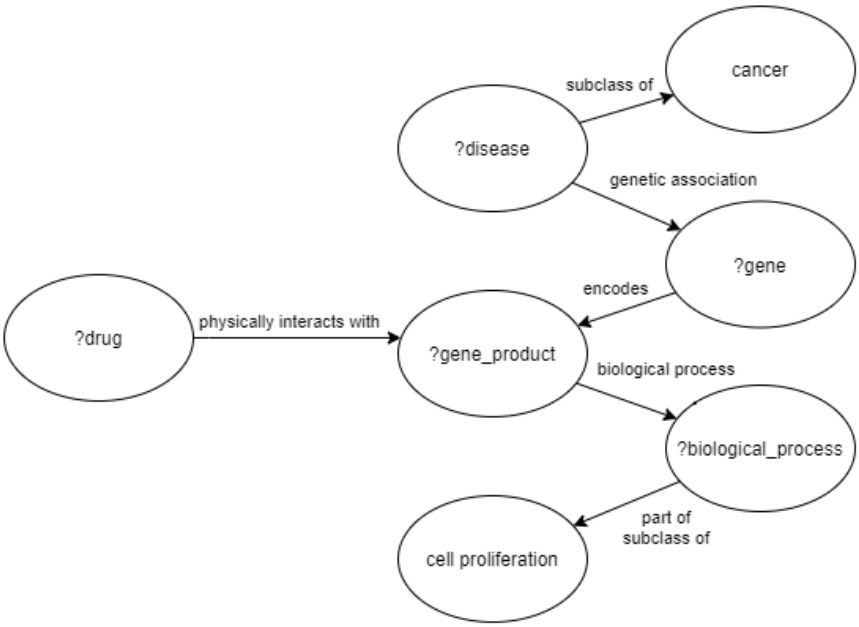


Figura 2: Un grafo que representa un patrón complejo.
Aquellos grafos que son solución de este patrón corresponden a aquellos medicamentos para el cáncer que afecta a los genes relacionados con la proliferación celular. Fuente: Elaboración propia.

Estos grafos incompletos pueden ser presentados a motores de bases de datos orientadas a grafos para ser completados, a través de consultas, con información que este repositorio contiene, obteniendo así, soluciones que cumplen con el patrón presentado. Existen múltiples motores de bases de datos orientadas a grafos, en este trabajo nos centraremos en el repositorio de datos *Wikidata* [Erxleben et al., 2014] y su motor de búsqueda *Wikidata Query Service*[2].

En un ejemplo concreto, podemos obtener soluciones para el grafo de la figura 2 en *Wikidata* a través de la consulta *SPARQL* que se puede apreciar en la figura 3. *SPARQL (SPARQL Protocol And RDF Query Language)* corresponde al lenguaje estandarizado para describir consultas a repositorios de datos RDF.

1.2. Dificultades al utilizar SPARQL

En base a todo lo anterior podemos notar lo siguiente: Para utilizar *SPARQL* y lograr obtener información útil y relevante desde un repositorio *RDF*, necesitamos dos cosas.

1. Conocer la sintaxis del lenguaje para consultas *SPARQL*.

[2]https://query.wikidata.org/

```
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wd: <http://www.wikidata.org/entity/>

SELECT DISTINCT ?biological_process ?drug ?gene ?disease WHERE {
    ?gene_product wdt:P682 ?biological_process .
    ?biological_process (wdt:P361|wdt:P279)* wd:Q14818032 .
    ?drug wdt:P129 ?gene_product .
    ?gene wdt:P688 ?gene_product .
    ?disease wdt:P2293 ?gene .
    ?disease wdt:P279* wd:Q12078 .
}
```

Figura 3: Consulta *SPARQL* generada en base al grafo incompleto de la figura 2.
Fuente: Elaboración propia.

2. Conocer la estructura de los elementos disponibles en el repositorio *RDF*.

Es bastante difícil cumplir estos prerrequisitos si nuestros usuarios no tienen experiencia previa con estas tecnologías. Por lo tanto, para apoyar a nuestros usuarios en su búsqueda de información se han desarrollado herramientas y plataformas que facilitan la construcción de consultas y la exploración de las entidades disponibles en los repositorios *RDF* como por ejemplo, *RDFExplorer* [Vargas et al., 2019], *Tabulator* [Berners-Lee et al., 2006] , *Explorator* [Araújo et al., 2009], *DBpedia Atlas* [Valsecchi et al., 2015], *RDF Visualizer* [Sayers, 2004], *SPARQL Assist* [McCarthy et al., 2012] o *YASGUI* [Rietveld and Hoekstra, 2017].

En este trabajo hemos decidido utilizar *RDFExplorer* en base a las siguientes características:

- Facilidad de uso en su interfaz.
- Construcción interactiva de consultas *SPARQL*.
- Navegación de las entidades disponibles en la base de datos *RDF*.
- Generación de resultados parciales.
- Código fuente disponible y abierto [Vargas, 2019].
- Disponible a través de una interfaz Web.

Una captura de la interfaz de *RDFExplorer* se puede apreciar en la figura 4.

Sin embargo, aun cuando utilizamos herramientas como *RDFExplorer* debemos tener algún grado de conocimiento sobre las propiedades de las entidades disponibles en nuestro repositorio *RDF*. Para facilitar esta tarea, podemos generar recomendaciones o sugerencias sobre

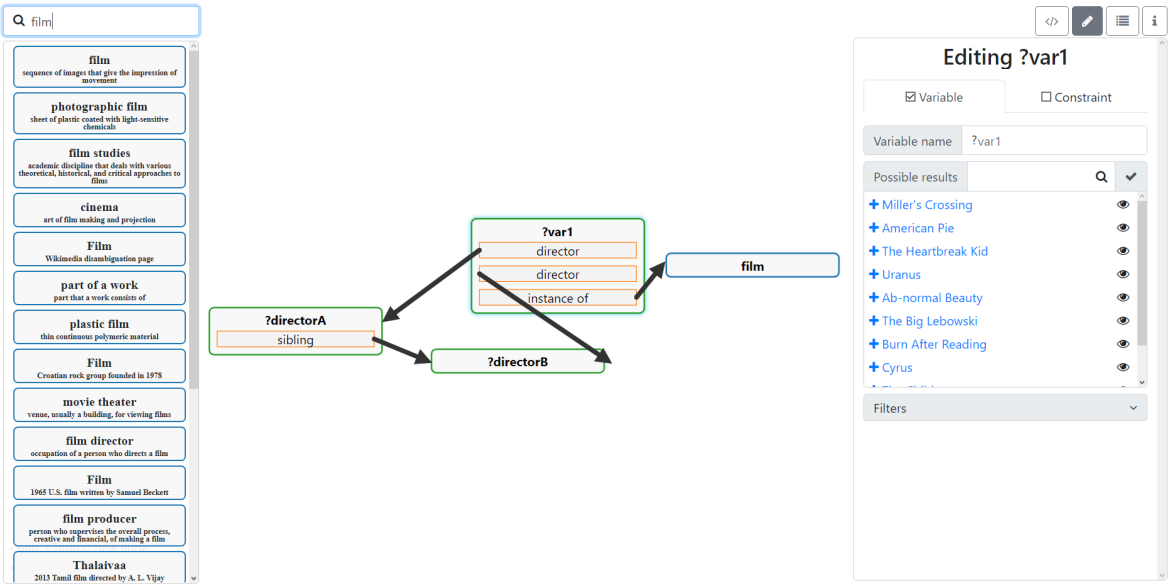


Figura 4: RDFExplorer, construcción de una consulta *SPARQL*.
Obtiene el conjunto de aquellas películas que han sido dirigidas por hermanos. A la derecha, un conjunto de posibles resultados. Fuente: Elaboración propia.

los posibles resultados o soluciones de un grafo incompleto determinado. Esto es similar a las herramientas y funcionalidades para el autocompletado en procesadores de texto o entornos de desarrollo integrados orientados al desarrollo de software [Bruch et al., 2009].

1.3. Generación de sugerencias

En la actualidad, existen soluciones y herramientas que nos permiten obtener recomendaciones en base a consultas *SPARQL* incompletas. Ejemplos de estas soluciones son *Gosparqled* [Campinas, 2014], *LinkedWiki editor* [Rafes et al., 2018], *SPARKLIS* [Ferré, 2017], *SPACE* [Kramer et al., 2013] y *SPARQLforHumans* [De la Parra, 2020]. De las alternativas mencionadas, una tiene una importante ventaja; *SPARQLforHumans* se encuentra integrado con *RDFExplorer*, sin embargo, presenta ciertos problemas descritos por sus autores.

- El tiempo necesario para iniciar el software puede tomar hasta 109 horas.
- Una vez iniciado el software, sus resultados son inmediatamente obsoletos. Esto, debido a que en *Wikidata* pueden llegar a ocurrir $200{,}000$ cambios por día [Wikipedia, 2022].
- El hecho de estar diseñado para soportar distintos tipos de repositorios *RDF* no permite realizar optimizaciones importantes a su rendimiento.

Por nuestra parte, vemos estos problemas como oportunidades de mejora para las aplicaciones *SPARQLforHumans* y *RDFExplorer*

1.4. Objetivos

En base a todo lo expuesto anteriormente, en este trabajo buscamos replicar y lograr las mismas funcionalidades logradas por el proyecto *SPARQLforHumans* implementando mejoras orientadas a la reducción del tiempo de procesamiento y a la actualización en tiempo real de sus datos provenientes desde *Wikidata* utilizando solamente herramientas de código abierto.

1.4.1. Objetivo general

Analizar, diseñar e implementar una nueva arquitectura para el proyecto *SPARQLforHumans*, utilizando exclusivamente componentes de código abierto, logrando reducir el tiempo necesario para obtener resultados en la aplicación.

1.4.2. Objetivos específicos

- Revisar y evaluar la arquitectura actual del proyecto *SPARQLforHumans*.
- Diseñar la arquitectura del sistema utilizando solamente componentes de código abierto.
- Implementar la arquitectura propuesta.
- Evaluar el rendimiento de la solución implementada.

CAPÍTULO 2
MARCO CONCEPTUAL

Para abordar nuestro problema, primero debemos definir algunos conceptos que han sido mencionados en las secciones anteriores de forma más formal, tales como, la *web* semántica, *SPARQL*, *RDF* y el contexto en el que estos se utilizan. Para esto, nos apoyaremos en las siguientes definiciones.

2.1. *Web* Semántica

La red informática mundial, *World Wide Web* o simplemente *web* es el sistema de información público más importante del mundo desarrollado en los últimos 30 años, el cual permite la transmisión de documentos electrónicos identificados por *URIs (Uniform Resource Identifiers)* y que pueden estar enlazados a otros documentos a través de enlaces de hipertexto.

La *web* fue diseñada como un espacio para la información con el objetivo de no solo ser útil para las comunicaciones entre humanos, sino que también, un lugar donde las máquinas podrían ayudar y participar. Sin embargo, uno de los principales problemas de la *web* es que la mayor parte de su contenido ha sido diseñado para ser consumido por humanos, lo que implica que para las máquinas y el *software* no es fácil acceder e interpretar el contenido disponible, incluso si este proviene de una base de datos estructurada a través de columnas claras y tipificadas. La *web* semántica busca desarrollar herramientas, lenguajes, protocolos y estándares que permitan, tanto a maquinas como humanos, procesar toda la información disponible en la *web*. En base a esto, podemos definir a la *web* semántica como la idea de generar una red de datos en la *web* que, hasta cierto punto, se puede interpretar como una base de datos global. [Berners-Lee et al., 1998]

2.2. Arquitectura de la *web* semántica

La *web* semántica está construida en base a múltiples bloques, los cuales representan estándares y lenguajes utilizados para lograr determinadas funcionalidades descritas en su arquitectura [Harth et al., 2011], una representación gráfica de esta se puede observar en la figura 5, la cual, podemos resumir en las siguientes capas, ordenadas desde los niveles inferiores a superiores:

1. Referencias, transporte y principios de los datos enlazados.
2. Intercambio de datos.

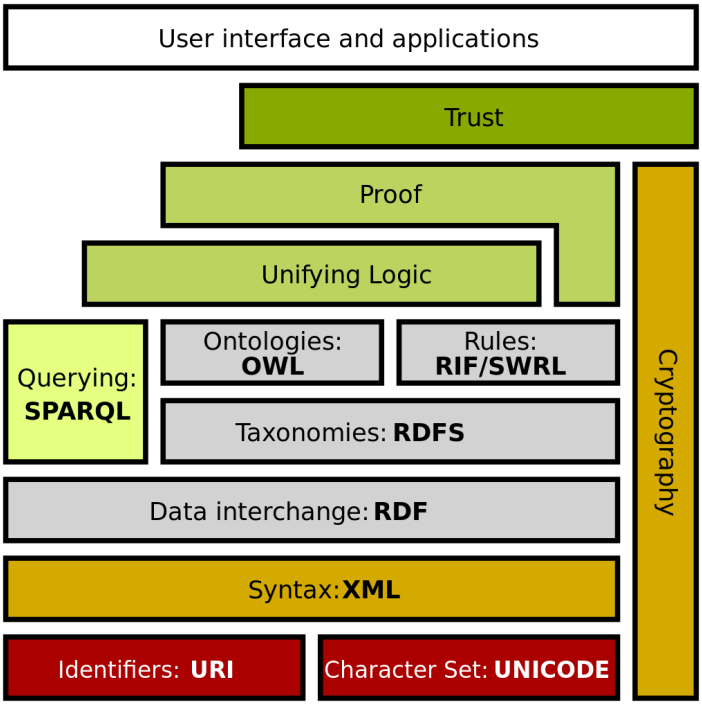


Figura 5: Arquitectura de la *web* semántica.
Fuente: *Semantic Web Stack*. Wikipedia.

3. Consultas y actualizaciones.
4. Ontologías y razonamiento.
5. Reglas.
6. Seguridad y encriptación.
7. Unificación e integración.
8. Confianza.
9. Aplicaciones.

A continuación, definiremos estas capas de forma más detallada entre las secciones 2.2.1 hasta la 2.2.9.

2.2.1. Referencias, transporte y principios de los datos enlazados

El acceso a los datos es fundamental en la arquitectura de la *web* semántica. Podemos tomar como referencia, el modelo utilizado por los servidores *web*, en el cual, los documentos

disponibles se encuentran enlazados a otros de forma descentralizada, esto es, que aquel documento referenciado no necesariamente se encuentra en el mismo servidor que está haciendo referencia a él. Estos enlaces, son utilizados por los usuarios para navegar entre los millones de servidores disponibles en la *web*.

Las *URI/IRI* y el protocolo *HTTP* son igual de importantes para el núcleo que define tanto a la *World Wide Web* como a la *web* semántica. En un ejemplo concreto, la *URI* `https://en.wikipedia.org/wiki/Back_to_the_Future` en la *web*, representa el documento en el servidor `https://www.wikipedia.org` que contiene información sobre la serie de películas y obras de título "Volver al Futuro", en cambio, en el contexto de la *web* semántica, la *URI* `https://www.wikidata.org/wiki/Q1` representa al "universo" como una entidad real, la cual es parte del `https://www.wikidata.org/wiki/Q3327819` "multiverso" y es estudiado por la `https://www.wikidata.org/wiki/Q338` "cosmología".

Los datos del ejemplo anterior son publicados por "*The Wikipedia Fundation*" a través del servicio *Wikidata* [Vrandečić and Krötzsch, 2014], pero las relaciones descritas podrían enlazar a otros editores de contenido, como por ejemplo *DBpedia* [Valsecchi et al., 2015], el cual corresponde a un esfuerzo comunitario para extraer información estructurada desde distintas fuentes y enlazarlas a través del formato *RDF*. Para lograr esto, los editores que publican contenido en la *web* semántica aplican los siguientes principios a sus datos, los cuales son conocidos como los "principios para datos enlazados" o *"LinkedData principles"* [Bizer et al., 2011].

1. Utilizar *URIs* como nombres para entidades.
2. Utilizar *URIs HTTP* para que los usuarios puedan buscar y acceder a estas entidades.
3. Cuando un usuario consulta una *URI*, debes entregar información relevante, utilizando estándares como *RDF* y *SPARQL*.
4. Debes incluir enlaces a otras *URIs*, para que los usuarios descubran más entidades.

2.2.2. Intercambio de datos

Los datos de la *web* semántica son generados por distintas entidades alrededor del mundo, las cuales no están necesariamente coordinadas entre ellas, por lo que su arquitectura debe soportar la creación distribuida de datos junto con la integración de múltiples fuentes y la interoperabilidad entre la información creada [Bizer et al., 2011]. Este tipo de requerimientos los cumplen las estructuras de datos basadas en grafos como *RDF*. *RDF* es un formato que describe grafos dirigidos, los cuales representan información en forma de tríos "sujeto - predicado - objeto", en estos, el sujeto y el objeto corresponden a nodos del grafo y el predicado es un arco que los relaciona, como se puede observar en la figura 1. En estos tríos, cualquiera de estas entidades puede tomar el valor de una *URI*, un valor literal (cadenas de

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:ex="http://example.org/stuff/1.0/">
    <rdf:Description
        rdf:about="http://www.w3.org/TR/rdf-syntax-grammar"
        dc:title="RDF1.1 XML Syntax">
        <ex:editor>
             <rdf:Description ex:fullName="Dave Beckett">
               <ex:homePage rdf:resource="http://purl.org/net/dajobe/"/>
            </rdf:Description>
        </ex:editor>
    </rdf:Description>
</rdf:RDF>
```

Figura 6: Descripción de un documento *RDF/XML*.
Fuente: RDF 1.1 XML Syntax. World Wide Web Consortium.

texto, números o fechas) o simplemente un nodo vacío (identificadores que no pueden ser referenciados por otra entidad).

RDF corresponde a la especificación de un lenguaje abstracto para describir relaciones entre entidades, el cual, puede ser serializado en múltiples formatos de texto como *Extensible Markup Language (XML)* [Beckett and McBride, 2004] en la figura 6 o en un formato más compacto llamado *Turtle* [Beckett et al., 2014] en la figura 7.

2.2.3. Consultas y actualizaciones

Los principios de los datos enlazados explicados en la sección 2.2.2 nos entregan guías sobre cómo realizar la publicación y permitir el acceso a datos simples, sin embargo, no es posible realizar consultas complejas a aquellos datos publicados utilizando estos principios, puesto que, no es obligatorio contar con un sistema o mecanismo de consulta para publicar información. Si continuamos con el ejemplo de las películas “Volver al Futuro”, consideremos que buscamos obtener los títulos de aquellas películas en las que los miembros del elenco de “Volver al Futuro” también han actuado. Una forma relativamente sencilla de lograr esto, es acceder a la *URI* que representa la película, obtener las *URIs* de los actores y de forma iterativa acceder a estas para obtener el nombre de las películas en las que cada miembro ha participado. No es difícil darse cuenta, que este proceso no es para nada eficiente en tiempo de ejecución ni recursos de red utilizados.

Para resolver esta búsqueda, podemos utilizar el lenguaje de consultas *SPARQL*, cuyo

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix ex: <http://example.org/stuff/1.0/> .

<http://www.w3.org/TR/rdf-syntax-grammar>
    dc:title "RDF/XML Syntax Specification (Revised)" ;
    ex:editor [
        ex:fullname "Dave Beckett" ;
        ex:homePage <http://purl.org/net/dajobe/>
    ] .
```

Figura 7: Descripción de un documento *RDF/Turtle*.
Fuente: Turtle (Syntax). Wikipedia.

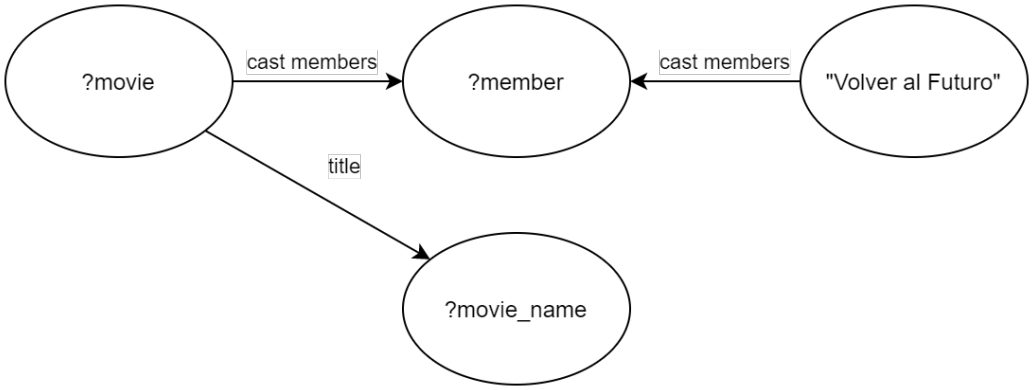


Figura 8: Patrón de un grafo *RDF* básico.
Fuente: Elaboración propia.

nombre corresponde al acrónimo recursivo "***S**PARQL **P**rotocol **A**nd **R**DF **Q**uery **L**anguage*" [W3C et al., 2013]. Este lenguaje está diseñado para evaluar consultas en repositorios de datos que están almacenados en formato *RDF*, en estos repositorios, la información no se obtiene accediendo de forma iterativa a distintas *URIs* que representan entidades, si no que enviando consultas a un *endpoint*[3] que soporta *SPARQL*. *SPARQL* permite a sus usuarios especificar *URIs* arbitrarias, las cuales podrían no ser accesibles a través de la *web*, junto con un patrón de grafo dirigido el cual debe coincidir con los datos disponibles en el repositorio y en el que pueden ser descritas determinadas restricciones para los datos obtenidos. En la figura 8 se puede apreciar el patrón del grafo utilizado para consultar por las películas en las que el elenco de "Volver al Futuro" ha participado.

Esta consulta puede ser representada utilizando *SPARQL*, como se puede apreciar en la fi-

[3] *Endpoint:* Del inglés punto final. Se refiere a un punto en la red expuesto por un sistema informático con el cual se puede interactuar o establecer un canal de comunicaciones.

```
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wd: <http://www.wikidata.org/entity/>

# Buscamos obtener solo el nombre de la película
SELECT DISTINCT ?movie_name WHERE {
    # La variable ?member es 'miembro del elenco' (P161)
    # de 'Volver al Futuro' (Q91540)
    wd:Q91540 wdt:P161 ?member .
    # La variable ?member es 'miembro del elenco' (P161)
    # de la variable ?movie
    ?movie wdt:P161 ?member .
    # La variable ?movie_name es 'título' (P1476)
    # de la variable ?movie
    ?movie wdt:P1476 ?movie_name .
}
```

Figura 9: Una consulta *SPARQL* realizada al servicio Wikidata.
Fuente: Elaboración propia.

gura 9. Una consulta *SPARQL* está compuesta de múltiples secciones, las sentencias *PREFIX* son utilizadas para abreviar *URIs* y su rol es apoyar la legibilidad de la consulta. El núcleo de la operación se encuentra en la sección *WHERE*, en la que se debe definir de forma precisa el patrón de nuestro grafo dirigido que deberá coincidir con la información semántica disponible en el repositorio consultado. Un grafo básico consiste en patrones individuales los cuales pueden ser sujetos, predicados u objetos unidos por variables, formando una plantilla, la que será completada en el proceso de evaluación de la consulta. De forma opcional, una sentencia *WHERE* puede estar acompañada de una expresión *FILTER* la cual puede acotar los resultados obtenidos a determinadas estructuras que cumplan con criterios especificados.

La especificación *SPARQL* puede ser implementada en múltiples repositorios orientados a grafos, entre los populares se encuentran Sesame [Broekstra et al., 2002], Jena [McBride, 2001], Virtuoso [Openlink, 2015], BigData [Thompson et al., 2016], OWLIM [Kiryakov et al., 2005] y RDF-3X [Neumann and Weikum, 2010]. Debido a que en muchas ocasiones los datos están almacenados en bases de datos relacionales, se necesitan *wrappers*[4] que nos permitan acceder a estos datos a través de *APIs*. Ejemplos conocidos de estos *wrappers* son D2R [Bizer and Cyganiak, 2006] y Triplify [Auer et al., 2009]. Estas herramientas permiten a fuentes de datos legadas ser expuestas y consultadas como grafos semánticos, lo que facilita la transición hacia estas nuevas tecnologías.

[4]*Wrapper*: Del inglés envoltorio. Una función o segmento de *software* que ejecuta a otras funciones ya sea por comodidad o para mejorar la compatibilidad o interoperabilidad del *software* ejecutado. En nuestro caso, se envuelve un repositorio de datos relacionales para soportar consultas *SPARQL* sin la necesidad de realizar cambios en los datos almacenados.

Además de la especificación de un lenguaje, *SPARQL* define los protocolos de acceso y formatos interoperables para los conjuntos de datos almacenados. Los datos son expuestos a través de *HTTP*, lo que permite un libre acceso y elimina la necesidad a los usuarios de descargar estos datos para consultarlos.

Actualmente el estándar *SPARQL* permite realizar consultas a una única fuente de datos a la vez. Para acceder a datos desde múltiples fuentes al mismo tiempo es posible realizar varias consultas secuenciales a los distintos repositorios, sin embargo, la responsabilidad de unir estos resultados de forma consistente es transferida al usuario que realiza la consulta. Actualmente se está diseñando una solución alternativa para realizar consultas federadas.

2.2.4. Ontologías y razonamiento

Para codificar un significado en nuestros datos debemos utilizar construcciones lógicas. Las tecnologías que habilitan el razonamiento en la *web* semántica son los esquemas *RDF*, el lenguaje de ontologías[5] *OWL (Web Ontology Language)* [Antoniou and Van Harmelen, 2004] y *RIF (Rule Interchange Format)* [Kifer, 2008].

Para explicar el razonamiento en la *web* semántica, es decir, generar conclusiones en base a hechos y verdades disponibles en nuestros repositorios de datos, podemos utilizar una ontología junto con las propiedades `rdfs:subClassOf` y `owl:sameAs`. La propiedad `rdfs:subClassOf` puede ser utilizada para definir jerarquías de clases y entidades. Consideremos el siguiente ejemplo, definimos una ontología llamada `OntologíaFormula1` en el contexto de la competencia de automovilismo internacional, esta ontología define además dos clases: `f1:Competidores`, `f1:Equipo` y contiene un axioma[6] el cual indica que `f1:Equipo` es una subclase de `f1:Competidores` a través de la propiedad `rdfs:subClassOf`. Esta relación le permite a un agente inteligente deducir que las instancias de `f1:Equipo` también son del tipo `f1:Competidores`, de esta forma si en nuestro repositorio existen los registros “Charles Leclerc” del tipo `f1:Competidores` y “Ferrari” del tipo `f1:Equipo`, cuando realizamos una consulta por todas las entidades del tipo `f1:Competidores` obtendremos como resultado “Charles Leclerc” y “Ferrari” incluso cuando las instancias no especifican esta relación de forma explícita.

Otra propiedades importante es `owl:sameAs`, la cual puede ser usada para denotar que dos recursos son idénticos, incluso cuando se encuentran en repositorios distintos. Por ejemplo, podemos indicar que el recurso `https://www.wikidata.org/wiki/Q27586` es idéntico a `http://dbpedia.org/page/Ferrari`, lo que permite a nuestro agente inteligente consolidar información más completa para una entidad determinada desde múltiples fuentes de datos.

[5] Ontología: En el campo de la informática, corresponde a una definición formal de tipos, propiedades y relaciones entre distintas entidades las cuales pertenecen a un determinado conjunto o dominio.

[6] Axioma: Una sentencia que debe ser considerada verdadera y que se utiliza como premisa para realizar futuros razonamientos o argumentos.

2.2.5. Reglas

Otro mecanismo que nos permite generar conclusiones en base a datos existentes son las reglas lógicas. Estas reglas están compuestas de dos secciones, antecedentes y consecuencias: si se cumplen las sentencias en los antecedentes, entonces las sentencias en las consecuencias son verdaderas. La *W3C* recomienda utilizar *RIF* [Kifer and Boley, 2013] para intercambiar reglas entre sistemas. Un conjunto común de reglas utilizadas en múltiples sistemas ha sido estandarizado en *RIF Core* [Boley et al., 2010].

2.2.6. Seguridad y encriptación

En un sistema global abierto como la *internet*, donde la información es transmitida a través de canales inseguros y sin autenticación utilizando infraestructura mantenida por una gran cantidad de organizaciones, es necesario definir mecanismos y protocolos para realizar intercambios de información de forma segura. Estos problemas son abordados utilizando la capa de criptografía en la *web* semántica.

Para asegurar que los datos no son alterados durante su transmisión se utiliza el protocolo *HTTPS* [Rescorla, 2000] el cual hace uso de encriptación para proteger la información de ataques de intermediarios o *MITM (man-in-the-middle)*[7].

El estándar *RDF* utiliza firmas digitales para asegurar la autenticidad del contenido entregado por los repositorios de datos. Este proceso se realiza utilizando métodos estándares y conocidos de firma electrónica [Carroll, 2003], lo que permite demostrar que el contenido proviene de una fuente confiable y que no ha sido modificado durante su transmisión a través de canales inseguros.

Si buscamos establecer la identidad de un usuario que intenta utilizar un servicio determinado podemos utilizar mecanismos de autenticación como *OpenID* [Recordon and Reed, 2006], en el cual, los usuarios son redirigidos a un portal donde se verifican sus credenciales y se obtiene su identidad digital.

2.2.7. Unificación e integración

El proceso de unificación en la *web* semántica se refiere al cómo podemos asegurar que un identificador determinado es el correcto para referenciar a una entidad determinada. Este problema se genera debido a las características de la *web* semántica, puesto que corresponde a un sistema distribuido global en el que cualquier actor puede publicar su propia

[7] *Man-in-the-middle attack:* Del inglés “Ataque de intermediario”. En criptografía y la seguridad informática, corresponde a un ataque en el cual, un intermediario puede observar y alterar el contenido de un canal de comunicaciones seguro entre dos partes, las cuales, creen estar interactuando de forma directa entre ellas.

información con sus propios identificadores. Un ejemplo de esto se puede apreciar en el desarrollo de la sección 2.2.4 en el cual los identificadores *Q27586* del *publisher*[8] *Wikidata* y *Ferrari* del *publisher DBpedia* hacen referencia a la misma entidad real: la compañía de motores y vehículos italiana de nombre "Ferrari".

Reusar identificadores permitiría a los agentes inteligentes descubrir y navegar el universo de datos descentralizado de forma más fluida y sencilla. Para aportar a esto, los mantenedores de contenido pueden agregar enlaces a *URIs* de otros mantenedores. En el caso anterior, si *DBpedia* agrega un enlace a la *URI* `https://www.wikidata.org/wiki/Q27586` en la descripción de su documento "Ferrari", se establece una asociación explícita de estas dos representaciones para una misma entidad en distintos repositorios. Otra manera de declarar de forma explícita esta relación es a través ontologías *OWL* usando las propiedades `owl:sameAs` o `rdfs:subClassOf`.

2.2.8. Confianza

No todos los datos son creados de la misma manera en la *web* semántica por lo que para lograr determinar el valor de los datos y como pueden ser utilizados, debemos conocer sus orígenes. Los orígenes de los datos pueden ser identificados de múltiples formas, una de estas es siguiendo la cadena de los procesos de información que los generaron, como por ejemplo, consultas a repositorios de forma automatizada [Dividino et al., 2009a] [Flouris et al., 2009]. Estos procesos pueden responder preguntas como quien ha creado los datos, desde donde provienen, cómo es que fueron construidos en base a otros datos y cuáles fueron las reglas de inferencias[9] que se utilizaron para agregar propiedades implícitas.

Cuando el proceso por el cual los datos fueron generados no está especificado completamente, se necesitan herramientas más abstractas para lograr rastrear los origines de la información. Estas herramientas pueden ser obtenidas desde entornos de trabajo cómo el *Open Provenance Model (OPM)* [Moreau et al., 2008], en el cual, las fuentes o procesos de determinados datos pueden ser representados como *black boxes*[10] y la información sobre las interacciones, dependencias y restricciones del proceso es obtenida a través de metadatos. Una muestra de un proceso descrito utilizando *OPM* se puede apreciar en la figura 10.

Una vez que hemos definido la procedencia de nuestros datos, podemos definir otra información relacionada a su origen tales cómo, autor, certeza de la información y fuentes, para así, a través de un proceso matemático [Dividino et al., 2009b], lograr calcular un valor a la confianza de los datos disponibles. Este proceso, puede ser aplicado a múltiples fuentes, sentencias y grafos, permitiéndonos obtener un valor de confianza para estructuras más

[8] *Publisher:* Del inglés editor. En nuestro caso, se refiere a los mantenedores de contenido en los repositorios de datos *RDF* distribuidos a través de *internet*.

[9] Inferencia: Proceso por el cual se derivan conclusiones a partir de premisas.

[10] Black Box: Del inglés "Caja Negra". Se refiere a un proceso o subproceso del cual no conocemos su funcionamiento interno y solo podemos observar sus entradas y salidas.

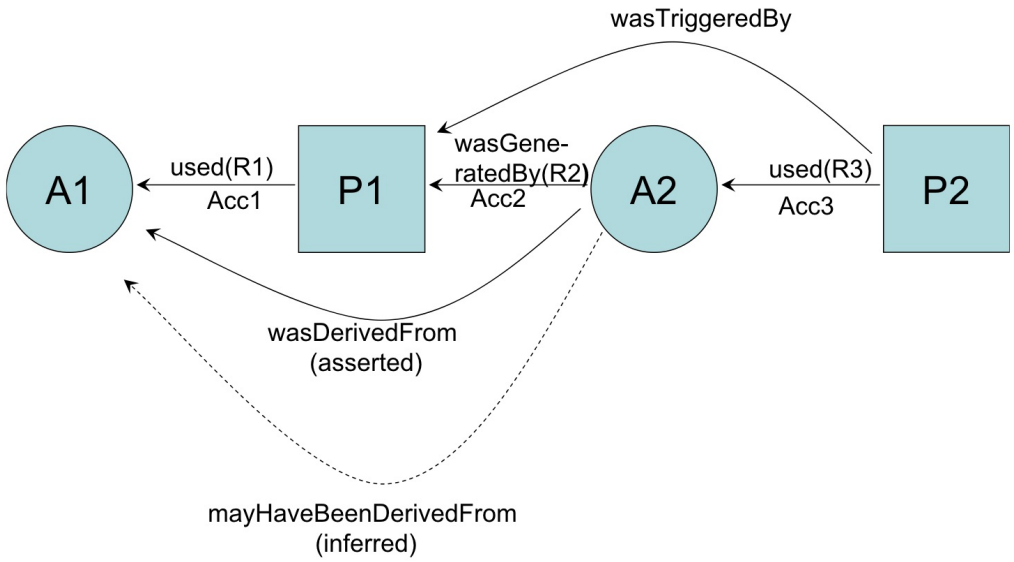


Figura 10: Ejemplo de un proceso descrito utilizando *OPM*.
Fuente: *The Open Provenance Model*.

complejas.

2.2.9. Aplicaciones

El paradigma básico para interactuar con datos es consultar y obtener una respuesta: los usuarios construyen consultas y los sistemas entregan respuestas. Por lo tanto, para que un sistema permita a los usuarios interactuar con sus datos, necesita una interfaz que acepte consultas como entrada y muestre las respuestas obtenidas desde el sistema de forma visual. En base a esto, muchas de las actuales interfaces disponibles para interactuar con repositorios *RDF* o *endpoints SPARQL* corresponden a interfaces interactivas *web*, como el *Wikidata Query Service* donde es posible realizar consultas y visualizar resultados de forma interactiva. Ejemplos de entrada y salida se pueden revisar en las figuras 11 y 12.

La *web* semántica genera nuevos desafíos para el diseño y desarrollo de interfaces que sean fácil de utilizar para usuarios sin conocimientos técnicos y que permitan responder a preguntas del estilo “¿Qué tipo de música se escucha en las estaciones de radio de Alemania?” o “¿Qué personas han participado en la realización de películas sobre viajes en el tiempo?” de forma rápida y fluida.

Con las definiciones realizadas entre las secciones 2.2.1 y 2.2.9 es que ahora podemos comprender la web semántica y resumirla como el conjunto de herramientas, lenguajes, protocolos y estándares que permiten tanto a humanos como a agentes inteligentes procesar toda la información disponible en la *web*, generando una base de datos global y distribuida.

```
# Awarded Chemistry Nobel Prizes

#defaultView:Timeline
SELECT DISTINCT ?item ?itemLabel ?when (YEAR(?when) as date) ?pic
WHERE {
    # ... ha sido premiado (P166)
    ?item p:P166 ?award .
    # ... con el Premio Nobel de Química (Q44585)
    ?award ps:P166 wd:Q44585 .
    ?award pq:P585 ?when .
    OPTIONAL {
        ?item wdt:P18 ?pic
    }
    SERVICE wikibase:label {
        bd:serviceParam wikibase:language "en" .
    }
}
```

Figura 11: Consulta *SPARQL* realizada en el *Wikidata Query Service*. Corresponde a obtener a aquellas personas que han sido premiadas con el Premio Nobel de Química. Fuente: Elaboración propia.

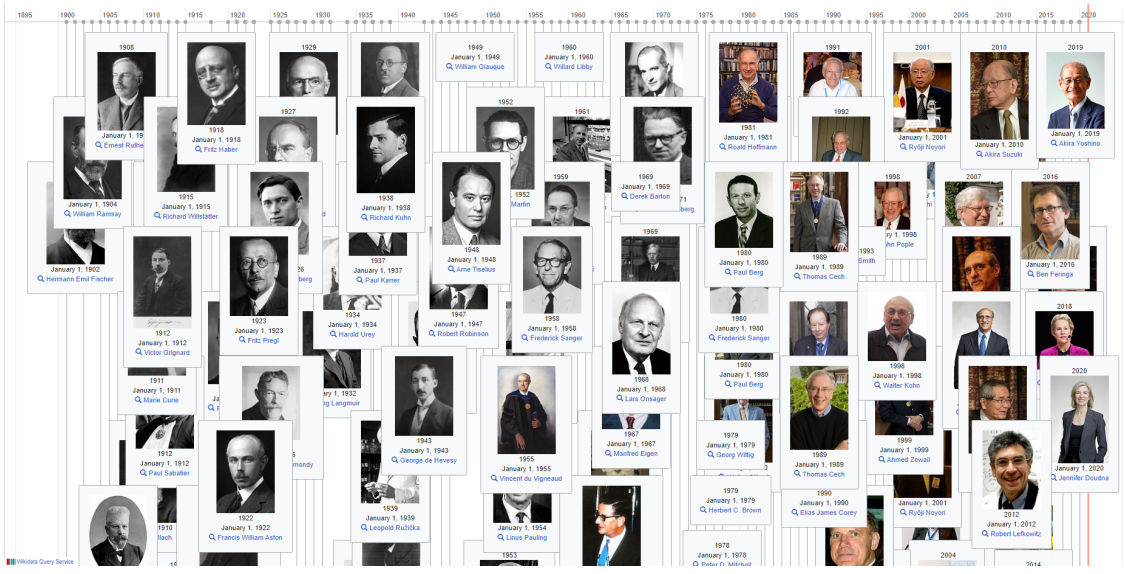


Figura 12: Visualización de una consulta *SPARQL* en el *Wikidata Query Service*. Representa los resultados de obtener a aquellas personas que han sido premiadas con el Premio Nobel de Química. Fuente: *Wikidata Query Service*.

2.3. Generación de sugerencias

El proceso de generación de sugerencias de entidades en base a consultas *SPARQL* parciales es la interfaz principal a nuestro sistema. Sin embargo, para realizar este proceso, debemos ser capaces de conocer las relaciones disponibles en nuestro conjunto de datos sin consultar directamente al repositorio fuente. Para esto, crearemos un repositorio local que contendrá las entidades, identificadores, descripciones, relaciones y otras propiedades necesarias para enviar las sugerencias a nuestros usuarios utilizando el proyecto *Apache Lucene* [Apache, 2012]. Además, debido a la gran cantidad de relaciones que pueden existir entre las entidades disponibles, debemos generar un mecanismo que nos permita ordenar nuestros resultados por relevancia. Utilizaremos el algoritmo *PageRank* [Page et al., 1999] para realizar este proceso.

2.3.1. Indexación de documentos

Al buscar información sobre un documento de texto utilizando palabras claves, estamos realizando una búsqueda sobre el contenido de este texto. Para responder a esto, necesitamos almacenar información sobre el contenido que estamos revisando junto con la relación que indica como resultado, el identificador único de la fuente. Por ejemplo, cuando buscamos un libro a través del nombre de uno de sus capítulos, las palabras claves son aquellas que se encuentran en el nombre del capítulo y el identificador es el nombre del libro. Este proceso de búsqueda se suele realizar a través de índices en bases de datos relacionales.

En el contexto de las bases de datos relacionales, los tipos de indicies más conocidos son el índice delantero (*forward index*) y en índice inverso (*inverse index*). Estrictamente hablando, no existe una diferencia técnica en estos dos tipos de índices, puesto que la implementación es la misma y es al momento de contextualizar el entorno en el que utilizaremos estos índices donde se genera una diferencia. En el mundo de la búsqueda de documentos, un *forward index* se construye describiendo la relación "El documento `id=7de44189` contiene las palabras `word1`, `word2`, `word3` y `word4`", en cambio, un *inverse index* nos presenta la información de la forma "La palabra `word2` se encuentra en los documentos `id=[7de44189, 8f1db8bb]`". Una ilustración de este ejemplo se puede observar en la figura 13.

Nuestro repositorio local consiste en un *inverse index*, en el cual, las etiquetas, descripciones, propiedades y relaciones descritas en el grafo *RDF* apuntan al identificador de la entidad descrita. De esta forma, podemos buscar entidades utilizando sus propiedades de la misma forma en la que buscamos un libro a través de su contenido.

Utilizaremos el proyecto *Apache Lucene* [Apache, 2012] para lograr esta funcionalidad, ya que soporta estos y muchos más tipos de índices aplicados a documentos de texto.

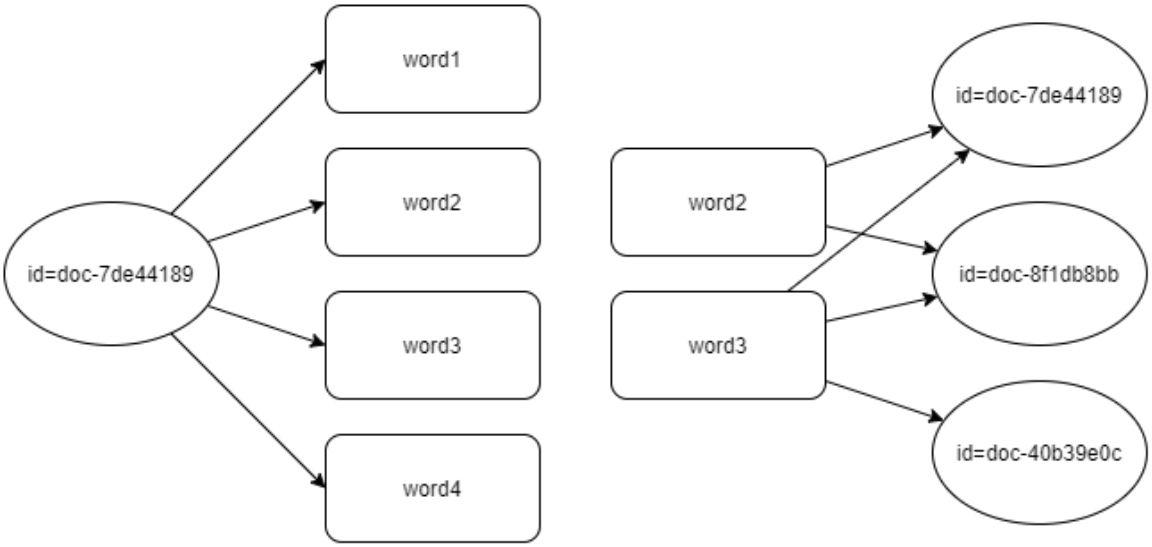


Figura 13: Ejemplos de un *forward index* y un *inverse index*.
A la izquierda *forward index* y a la derecha *inverse index*Fuente: Elaboración propia.

2.3.2. *Ranking* de resultados

Una vez que hemos indexado entidades para generar sugerencias, debemos entregar a nuestro usuario las más relevantes para la consulta parcial que estamos construyendo. Para esto, utilizaremos el algoritmo *PageRank* [Page et al., 1999], el cual nos permite asignar un valor de importancia a los resultados obtenidos en base a la cantidad de conexiones que cada elemento del grafo *RDF* tiene con otras entidades.

Si bien existen múltiples mecanismos para realizar un *ranking* de nodos en un grafo, tales como, *HITS (Hypertext Induced Topic Selection)* [Kleinberg et al., 1998] o *TextRank* [Mihalcea, 2004], hemos decidido utilizar *PageRank* debido a su popularidad, la existencia de múltiples implementaciones existentes listas para utilizar en varios lenguajes de programación y a que buscamos darle continuidad en nuestra solución basada en *SPARQLforHumans*.

PageRank fue diseñado para agregar orden e importancia a los resultados de los primeros motores de búsqueda en la *web*. En el diseño del algoritmo, los autores proponen que los sitios disponibles en la *web* pueden ser modelados utilizando un grafo dirigido, en el cual, cada sitio es un nodo del grafo y cada enlace hacia o desde estos sitios representa un arco entre los nodos. Una vez que hemos construido nuestra representación, debemos ejecutar un algoritmo recursivo para asignar una alta importancia a aquellos documentos que tienen muchos enlaces entrantes y pocos enlaces salientes.

El proceso recursivo asigna un valor inicial al *rank* de todos los documentos en la representación de manera que la suma de todos estos valores es 1. El siguiente paso es compartir el *rank* de un documento entre aquellos documentos a los que enlaza: un documento con

alto *rank* entregará bastante puntaje a sus enlazados en comparación a un documento de bajo *rank* y con muchos enlaces salientes. Este proceso recursivo está asegurado para converger, por lo que después de una cantidad determinada de iteraciones, el *rank* de todos los documentos ha sido calculado.

El valor calculado por *PageRank* es asignado a los documentos almacenados en nuestros índices *Lucene* a través de una propiedad llamada *Document Level Boosting*[11]

2.4. Visualizaciones

2.4.1. *RDFExplorer*

RDFExplorer es una interfaz *web* que le permite a sus usuarios construir consultas *SPARQL* de forma visual, agregando entidades y propiedades a estas consultas mediante un grafo. Un ejemplo de la construcción de una consulta a través de esta interfaz se puede revisar en la figura 4. En la figura, algunas de las propiedades de la entidad `?var1` tienen valores fijos, otras son incógnitas y se pueden observar sugerencias en la sección derecha. Estas recomendaciones, son generadas a través de una consulta directa a los servicios remotos de *Wikidata* la cual podría tomar unos minutos en entregarnos resultados o incluso podríamos no obtener una respuesta del servicio.

Es necesario además considerar, que el usuario puede modificar el grafo mientras esperamos una respuesta del servicio remoto. Al realizar una modificación, debemos cancelar la solicitud pendiente y enviar una nueva, la cual, podría sufrir el mismo destino al no obtener una respuesta. Este efecto resulta en que nuestro usuario no podrá obtener recomendaciones y no logrará explorar el conjunto de datos disponible.

Con estas situaciones en mente fue creado el proyecto *SPARQLforHumans* [De la Parra, 2020], con el objetivo de mejorar la experiencia del usuario a la hora de explorar conjuntos de datos en *RDFExplorer*.

2.4.2. *SPARQLforHumans*

SPARQLforHumans es una aplicación que busca situarse entre *RDFExplorer* y un repositorio de datos *RDF* para reducir los tiempos de respuesta generados a la hora de realizar determinados tipos de consultas *SPARQL* a un servicio remoto. Esto se logra procesando un conjunto de datos desde la fuente de información remota para ser almacenada en índices que representan las entidades y propiedades del mismo. La arquitectura de *SPARQLforHumans* se puede observar en la figura 14.

[11] Aumentar la importancia de una determinada entidad a través de un mecanismo externo.

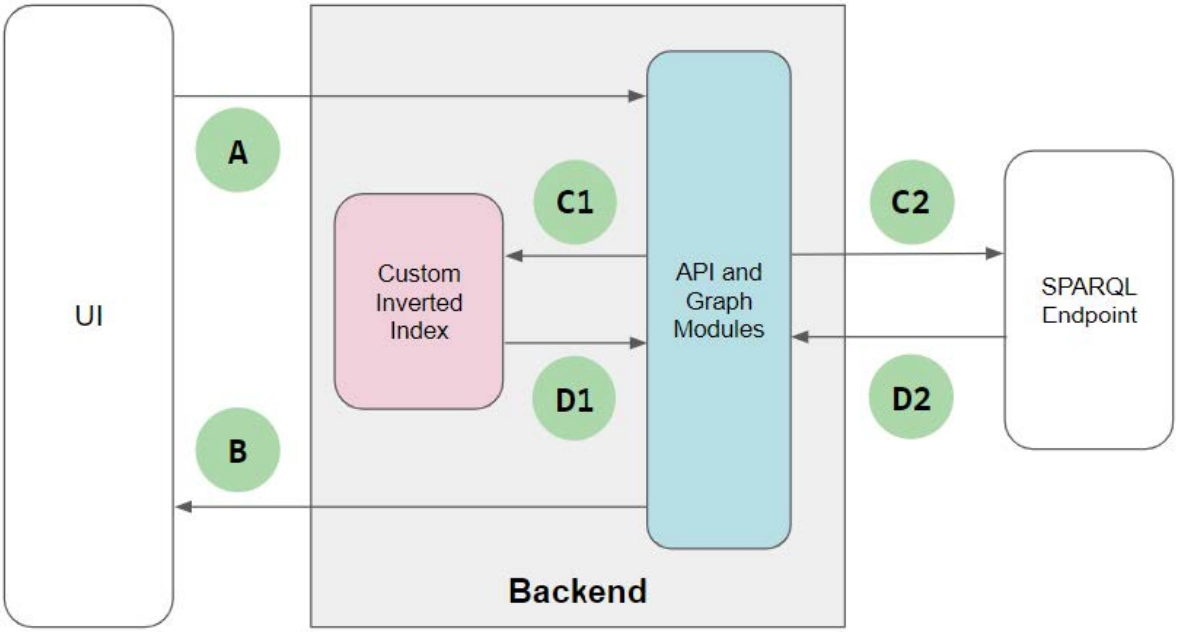


Figura 14: Arquitectura del sistema *SPARQLforHumans*
Fuente: [De la Parra, 2020]

CAPÍTULO 3
PROPUESTA DE SOLUCIÓN

3.1. Metodología de trabajo

Para nuestro proyecto, buscaremos utilizar una metodología de trabajo orientada a la agilidad. Existen múltiples metodologías de este estilo, pero hemos decidido utilizar el ciclo *PDCA*[12] (*Plan*, *Do*, *Check*, *Act*) por las siguientes razones:

- **Simple**. Nuestro mayor motivo de interés, es que *PDCA* consiste en un proceso simple, directo e intuitivo que podemos adoptar e implementar en nuestro flujo de trabajo, desarrollo e implementación de la solución.
- **Cíclico**. Debido a su naturaleza cíclica e iterativa, *PDCA* permite identificar las causas de los posibles errores durante el proceso de desarrollo del proyecto. Además, a medida que se implementan distintas soluciones es posible obtener información y experiencia para comprender el proceso que se busca mejorar.
- **Adaptable**. Esta metodología es una estrategia muy adaptable, aquellas personas que elijan implementarla pueden decidir con total libertad qué aspectos se deben considerar en cada una de las etapas del ciclo, la única condición es que las definiciones se deben mantener a lo largo de todo el proceso. Esta adaptabilidad permite que *PDCA* sea altamente escalable, ya que se puede adaptar a cualquier situación y en organizaciones de cualquier tamaño, incluso, en equipos de una sola persona.

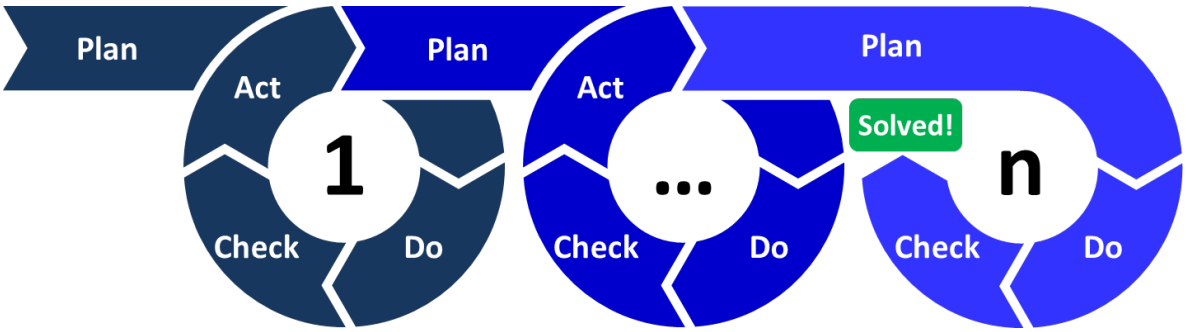


Figura 15: Ciclos *PDCA*.
Múltiples iteraciones del ciclo *PDCA* son realizadas hasta que se logra resolver el problema.
Fuente: *PDCA - Wikipedia Commons*.

En su esencia *PDCA* es una filosofía para abordar problemas. Primero, identificamos el problema y establecemos nuestros objetivos. Luego, probamos distintos enfoques para alcanzar

[12]PDCA: Del inglés Planear, Hacer, Verificar, Actuar.

dichos objetivos, analizamos nuestros resultados y adaptamos nuestro comportamiento en base a estos. Finalmente, avanzamos la iteración utilizando la solución que mejor ha funcionado. [Business, 2021]

Cada ciclo *PDCA* consiste en las siguientes etapas:

- **Planear**. Debemos comprender nuestro estado actual y el estado deseado. En pocas palabras, esta etapa busca que definamos nuestros objetivos, el cómo alcanzarlos y cómo medir nuestro progreso hacia dichos objetivos.
- **Hacer**. Una vez que hemos definido un plan de acción o una potencial solución para un problema, debemos probarla. Este paso es donde debemos poner a prueba los cambios propuestos en la etapa Planear. Sin embargo, esto se debe considerar como un experimento, no como una solución final. Por lo tanto, todas las pruebas se deben realizar en entornos controlados y a pequeña escala.
- **Verificar**. Luego de completar nuestras pruebas, debemos comprobar que los cambios o soluciones propuestas tienen el efecto deseado. En esta etapa se debe analizar la información recopilada durante la etapa Hacer y comparar lo obtenido con los objetivos y metas originales. En resumen, debemos evaluar nuestro nivel de éxito y qué cosas vamos a conservar para el siguiente paso del ciclo.
- **Actuar**. Al llegar a esta etapa, ya hemos logrado identificar una solución o propuesta de cambio para implementar en nuestro problema o proceso. Se deben aplicar estos cambios en la escala que sea necesaria por el proceso y con esto se definen las bases para una nueva iteración.

3.2. Plan de trabajo

Como hemos mencionado anteriormente, vamos a utilizar ciclos *PDCA* para encontrar la solución a nuestro problema, estos ciclos, son definidos conforme avanzamos en las iteraciones y debido a que se encuentra en el marco de las metodologías ágiles, sería irresponsable definir con antelación los objetivos de cada ciclo. Sin embargo, podemos definir determinados hitos que buscamos lograr durante todo este proceso, los cuales, detallamos a continuación:

1. Entender la arquitectura inicial de la solución existente (*SPARQLforHumans*).
2. Diseñar y definir una arquitectura acorde a nuestras restricciones.
3. Reducir significativamente el tiempo necesario para ejecutar la solución.
4. Diseñar y definir una arquitectura que permita la actualización en tiempo real de los datos utilizados por nuestra solución.

5. Actualizar los datos de nuestra solución en tiempo real.
6. Validar la solución a través de pruebas unitarias automatizadas.

Estos hitos, permitirán guiar las iteraciones de nuestro proceso *PDCA* y así no perder el rumbo durante la implementación.

Con las definiciones realizadas en las secciones 3.1 y 3.2 tenemos lo necesario para comenzar nuestras iteraciones.

3.3. Primera iteración: Análisis inicial de la solución existente

Planificación

Para nuestra primera iteración, nuestro objetivo es entender el proceso y los componentes de la aplicación *SPARQLforHumans*, para esto, revisaremos el código fuente disponible de forma pública en la plataforma *GitHub* https://github.com/gabrieldelaparra/SPARQLforHumans [De la Parra, 2020].

Implementación

Comenzaremos con entender el proceso. Al revisar el código fuente de la aplicación, logramos identificar las siguientes etapas:

1. Inicio de la aplicación.
    a) Lectura de un fichero en el formato de compresión *gzip* [Deutsch, 1996] desde *Wikidata*.
    b) Filtro y validación de líneas en formato *N-Triples*.
    c) Creación de objetos *Triple* en base a líneas válidas.
    d) Generación de propiedades inversas para determinadas entidades.
    e) Concatenación de líneas válidas y nuevas propiedades a nuevo fichero *gzip*.
    f) Ordenar líneas de nuevo fichero en función de la entidad *RDF* observada en cada línea.
2. Indexación de entidades.
    a) Lectura de nuevo fichero *gzip* generado anteriormente.
    b) Creación de objetos *Triple*.

   c) Generación de documentos en índice *Lucene* en base a propiedades de cada entidad.

3. Obtención del valor *PageRank* para entidades.
4. Indexación de propiedades.
   a) Lectura de documentos disponibles en índice *Lucene* para entidades.
   b) Cálculo de frecuencias para las propiedades existentes en los distintos documentos.
   c) Generación de documentos en nuevo índice *Lucene* en función de la frecuencia de cada propiedad identificada.
5. Sistema en vivo.
   a) Ejecución de prueba estadística y de rendimiento.
   b) Inicio de servidor Web.

Las interacciones entre cada etapa se pueden observar en el diagrama de procesos presentado en la figura 16 y los componentes identificados se pueden observar en la figura 17. Además, presentamos la arquitectura actual del sistema en la figura 18.

Resultados

Gracias al levantamiento del proceso y componentes realizado, podemos ver de forma más clara cómo interactúan cada uno de los distintos módulos de la aplicación existente. Esto permite entender su funcionamiento interno, de forma tal, que podemos comenzar a optimizar cada uno de los componentes identificados por separado, acercándonos a nuestro objetivo final.

3.4. Segunda iteración: Mejoras al rendimiento general

Planificación

Nuestra segunda iteración consiste en diseñar e implementar el conjunto de mejoras iniciales al rendimiento de la solución. Para lograr esto, comenzaremos con un nuevo diseño y una nueva implementación desde cero, en el lenguaje de programación basado en la *JVM*,[13]

[13]Del inglés. *Java Virtual Machine*.

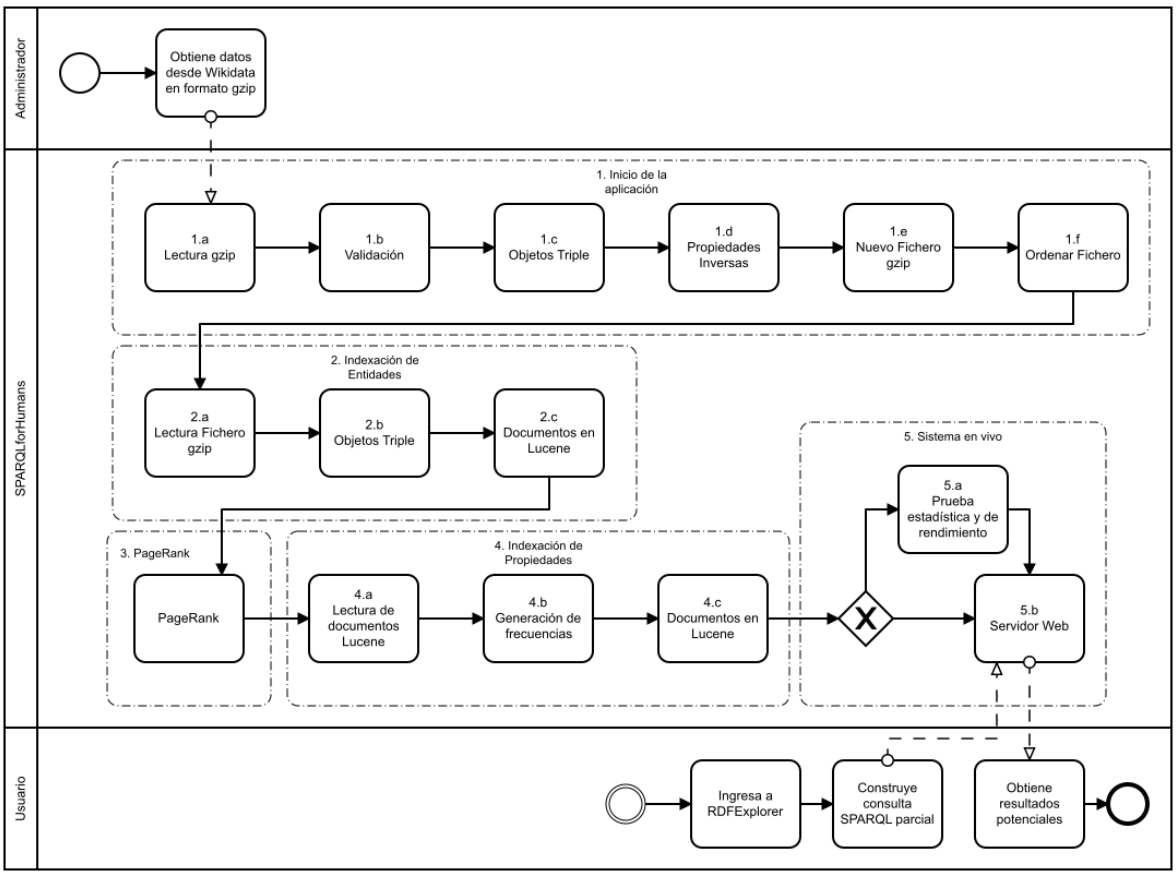


Figura 16: Proceso de arranque de la aplicación *SPARQLforHumans*.
Fuente: Elaboración Propia.

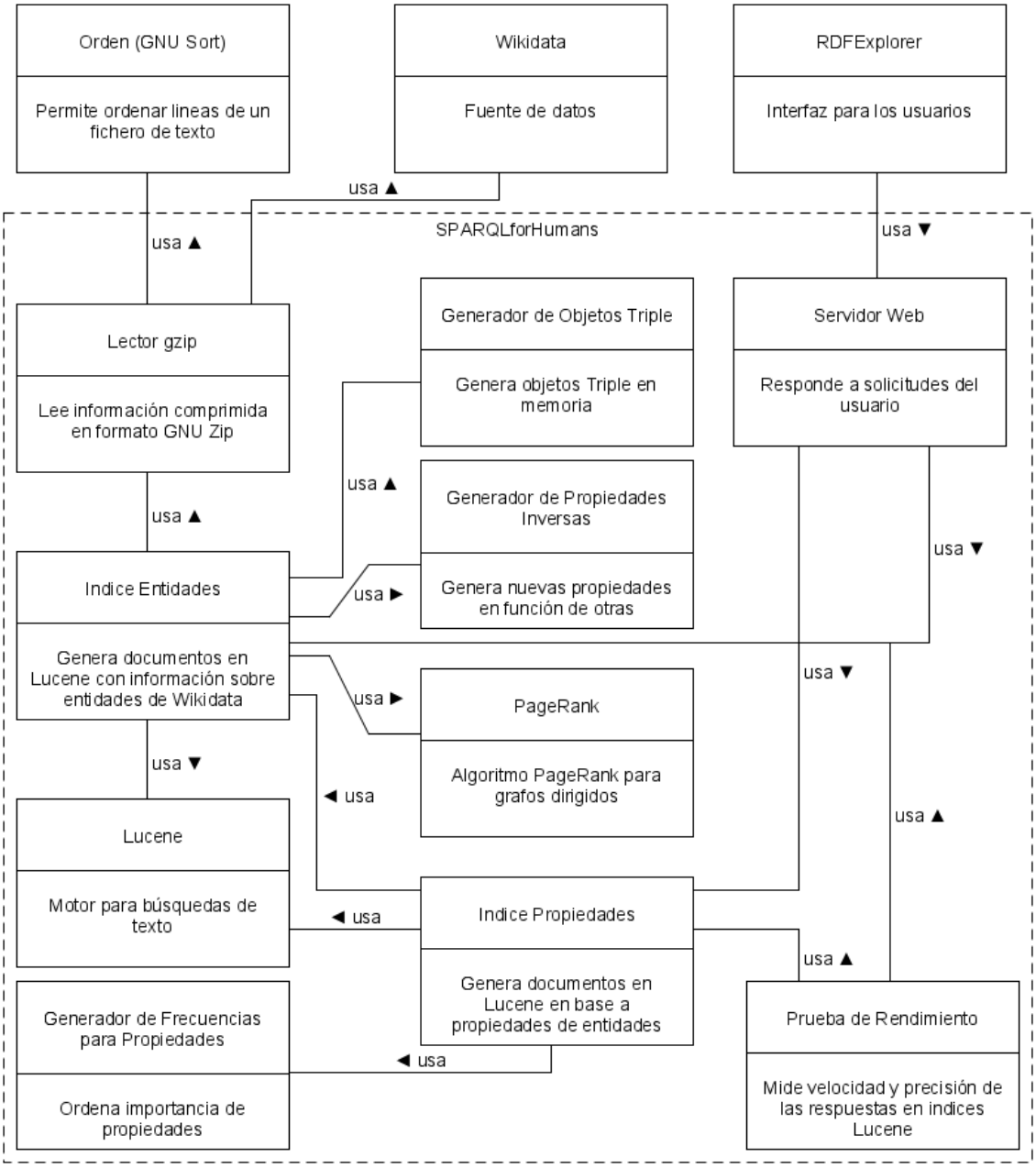


Figura 17: Diagrama de componentes para la aplicación *SPARQLforHumans*.
Fuente: Elaboración Propia.

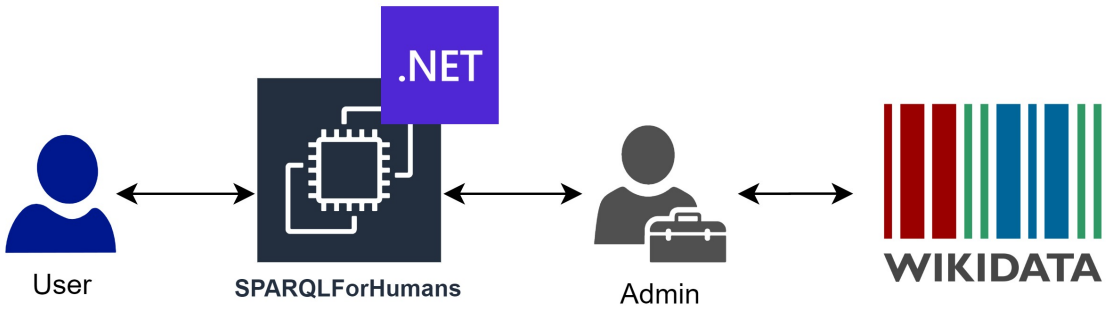


Figura 18: Diagrama de arquitectura para la aplicación *SPARQLforHumans*.
Fuente: Elaboración Propia.

[Venners, 1998] *Kotlin*[14]. Nuestra implementación se encuentra disponible en un repositorio publico de la plataforma *GitHub* https://github.com/capgadsx/SPARQLforHumans.

Para esta iteración, definiremos nuestro progreso en función de la cantidad de tiempo reducido en el proceso de arranque de nuestra aplicación. Es decir, desde el punto `(1.a)` hasta el punto `(5.b)` de nuestra definición. Cómo objetivo, hemos definido reducir en al menos un $50\,\%$ este tiempo de 109 horas.

Implementación

Comenzaremos aclarando nuestra decisión de utilizar *Kotlin* y la *JVM* para la implementación de la solución. Algo importante a notar es que *SPARQLforHumans* fue escrito utilizando el *.NET Framework* del lenguaje *C#*, sin embargo, el proyecto *Apache Lucene* [Apache, 2012] utilizado como el corazón de la solución está desarrollado en el lenguaje *Java*, por lo que se utiliza una biblioteca de compatibilidad llamada *Lucene.NET*[15]. Para nuestra solución, buscamos utilizar el proyecto original, *Apache Lucene*, por lo que deberíamos utilizar *Java* o algún otro lenguaje compatible con la *JVM*.

Kotlin es un lenguaje de programación moderno, maduro y versátil, el cual busca facilitar el desarrollo de aplicaciones sobre la *JVM*. Es conciso, seguro y completamente interoperable con *Java* y otros lenguajes que se ejecutan en la *JVM*. Por lo tanto, podemos utilizar sin problemas todas las funcionalidades de *Apache Lucene* a través de un lenguaje más moderno.

Lo siguiente es comentar sobre la implementación realizada. Nuestra solución de referencia corresponde a un sistema monolítico y para nuestras primeras mejoras vamos a seguir este modelo de arquitectura. En resumen, hemos realizado lo siguiente:

[14] *The Kotlin Programming Language*. https://kotlinlang.org/
[15] https://lucenenet.apache.org/

- Hemos eliminado el módulo de orden. En vez de ordenar nuestros resultados al final del subproceso `(1)`, vamos a utilizar una estructura de datos llamada *TreeMap*,[16] la cual permite mantener ordenados las propiedades inversas y objetos *Triple* generados en base al identificador de la entidad que actualmente estamos procesando, como por ejemplo $Q88618255$, sin la necesidad de utilizar las herramientas externas *gzip* y *sort*. Este cambio permite eliminar las etapas `(1.e)` y `(1.f)` de nuestra nueva solución.
- Hemos modernizado el componente que interactúa con *Lucene*, utilizando la última versión disponible para el formato de índices y la clase *MMapDirectory*[17] para interactuar con estos, la cual, corresponde a una interfaz de alto rendimiento que logra aprovechar el *cache* del sistema operativo donde se ejecuta.
- Hemos utilizado la biblioteca *Apache Jena* [McBride, 2001] para generar objetos *Triple*, esto permite generar nuevas propiedades y *Triples* de forma sencilla.
- El resto de los componentes, han sido implementados en *Kotlin* sin cambios significativos con la implementación de referencia.

Para realizar nuestras pruebas y obtener resultados, hemos definido tres ambientes de ejecución, desarrollo, pruebas y producción.

1. Ambiente de desarrollo.
   - Conjunto de datos: $200\mathrm{MB}$ de datos *N-Triple* comprimidos en formato *gzip*.
   - Recursos.
     - *CPU*. Intel(R) Core (TM) i5-7200U CPU @ 2.50GHz, 4 núcleos.
     - *RAM*. $7\mathrm{GB}$.
     - Almacenamiento. Samsung SSD 960 EVO 250GB. Interfaz NVME.
2. Ambiente de pruebas.
   - Conjunto de datos: *1GB* de datos *N-Triple* comprimidos en formato *gzip*.
   - Recursos.
     - *CPU*. Intel(R) Core (TM) i5-3470 CPU @ 3.20GHz, 4 núcleos.
     - *RAM*. *10GB*.
     - Almacenamiento. Seagate 320GB 7.2K SATA 3.5 HDD.
3. Ambiente de producción.
   - Conjunto de datos: *45GB* de datos *N-Triple* comprimidos en formato *gzip*.

[16] *TreeMap*: Un diccionario llave-valor, el cual, al ser recorrido de forma lineal entrega sus llaves de forma ordenada. https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html

[17] https://lucene.apache.org/core/8_9_0/core/org/apache/lucene/store/MMapDirectory.html

Recurso	Porcentaje de utilización
Procesador lógico 1	0 %
Procesador lógico 2	1.3 %
Procesador lógico 3	100 %
Procesador lógico 4	0.7 %
RAM	64 %
Memoria de intercambio (swap)	0 %

Tabla 1: Utilización de recursos en el ambiente de desarrollo.
Fuente: Elaboración Propia.

- Recursos, (Amazon AWS EC2 Instance - t3.2xlarge).
  - *CPU*. Intel(R) Xeon(R) Platinum 8259CL CPU @ 2.50GHz, 8 núcleos.
  - *RAM*. *32GB*.
  - Almacenamiento. Amazon AWS EBS, General Purpose SSD (gp2) 200GB. Interfaz NVME.

Al realizar pruebas en pequeña escala, los resultados son alentadores, pero hemos notado un problema, no estamos utilizando de forma eficiente los recursos disponibles de nuestro entorno de ejecución, especialmente nuestra memoria *RAM*.

En la tabla 1, podemos notar que solo estamos utilizando un 25 % de nuestra *CPU* total disponible y una gran parte de la memoria *RAM* del sistema, durante toda la ejecución de nuestra aplicación.

A pesar de esto, hemos decidido ejecutar el proceso en el ambiente de producción, para así obtener una métrica inicial sobre el tiempo que toma ejecutar nuestra solución. Lamentablemente, solo logramos obtener un error del tipo *java.lang.OutOfMemoryError* después de unas cuantas horas de ejecución.

La excepción *java.lang.OutOfMemoryError* es un indicador común de una fuga de memoria. Normalmente, este error se genera cuando no hay espacio suficiente para asignar un objeto en el *stack*[18] de la *JVM*. En este caso, el recolector de basura no puede hacer espacio disponible para acomodar un nuevo objeto, y el *heap*[19] no puede expandirse más. También, este error puede ser lanzado cuando no hay suficiente memoria nativa para soportar la carga de una nueva clase en la *JVM*. En un caso extremadamente raro, un error

[18] *Stack*: Corresponde a una sección de la memoria de un programa o proceso en la cual se almacenan variables locales, resultados parciales e información sobre la función actual que se esta ejecutando en un hilo determinado.

[19] *Heap*: Corresponde a una sección de la memoria de un programa o proceso en la cual se almacenan datos generados durante la ejecución de este. En el caso de la *JVM*, esta sección es compartida por todos los hilos existentes y los datos almacenados son eliminados de forma dinámica a través de un proceso llamado recolección de basura.

del tipo *java.lang.OutOfMemoryError* puede ser lanzado cuando se está gastando una cantidad excesiva de tiempo en la recolección de basura y poca memoria está siendo liberada [Oracle, 2021c] [Oracle, 2021b].

En nuestro caso, creemos que este error se debe a que no es posible acomodar nuevos objetos en nuestra memoria. Además, buscamos saber dónde estamos utilizando nuestra *CPU*, con el objetivo de implementar potenciales optimizaciones. Para lograr esto, debemos responder las siguientes preguntas:

1. ¿Dónde se está utilizando la *CPU*?
2. ¿Dónde se están creando los objetos que utilizan nuestra memoria *RAM*?
3. ¿Qué objetos son los que se encuentran en nuestra memoria *RAM*?

Esta clase de dudas pueden ser resueltas utilizando un tipo específico de herramientas llamadas *profilers*. Los *profilers* permiten examinar un programa en ejecución de forma dinámica con el objetivo de encontrar problemas relacionados al rendimiento o la utilización de otros recursos como memoria y red en la aplicación.

Para responder a nuestra primera incógnita, podemos utilizar una herramienta llamada *Java Flight Recorder*[20], la cual, permite examinar y grabar lo que está sucediendo en la *JVM*. Con esta herramienta podemos generar la figura 19, donde se puede observar una lista de métodos junto con el porcentaje del tiempo que fueron ejecutados en referencia al total del tiempo de ejecución de nuestro programa [Oracle, 2021a].

Podemos notar, que el $98{,}2\,\%$ del tiempo, la *CPU* está siendo utilizado por el método `process` de la clase `GZNTriplesPreProcessor`, el cual, utiliza a `processLine` y `filterLine`, ambos de la clase `NTriplesPreProcessor` y a `reverse` de la clase `CustomRDFTriple`. Estos métodos son los encargados de leer, validar y procesar cada línea del fichero en formato *gzip* utilizado como entrada de nuestra aplicación. Esto significa que no hemos logrado avanzar de las etapas `(1.b)`, `(1.c)` y `(1.d)` en nuestra solución antes de encontrarnos con el error de memoria.

Para las segunda y tercera preguntas, podemos utilizar el gráfico de la figura 20, generado por la herramienta *Async-Profiler*[21]. Esta figura, corresponde a una representación de aquellas funciones que han generado los objetos que se encuentran en la memoria de la *JVM* en un momento determinado. Podemos notar algunos puntos importantes en esta representación.

- La mayor parte de los objetos han sido generados desde `processLine`.

[20] *JFR*: Es una herramienta que permite recolectar datos de diagnostico y rendimiento sobre aplicaciones *Java* en tiempo real. *JFR* está fuertemente integrado en la *JVM*, lo que permite un impacto muy bajo en el rendimiento de la aplicación analizada.

[21] https://github.com/jvm-profiling-tools/async-profiler

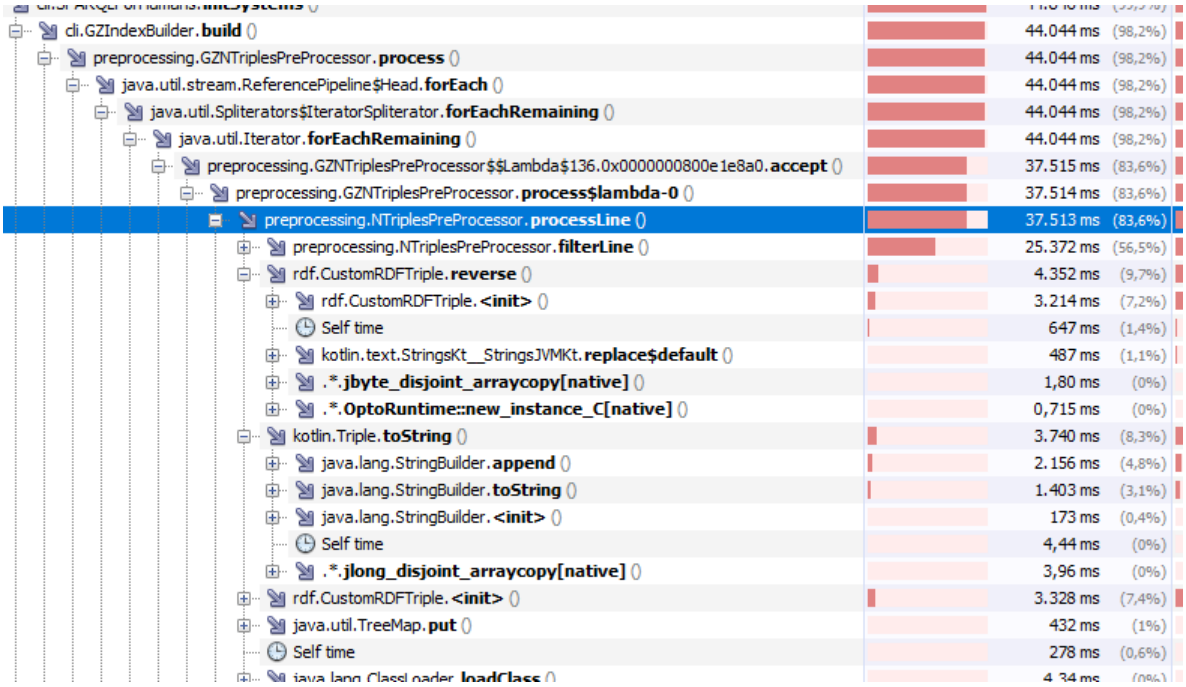


Figura 19: Reporte generado por *JFR*.
Fuente: Elaboración Propia.

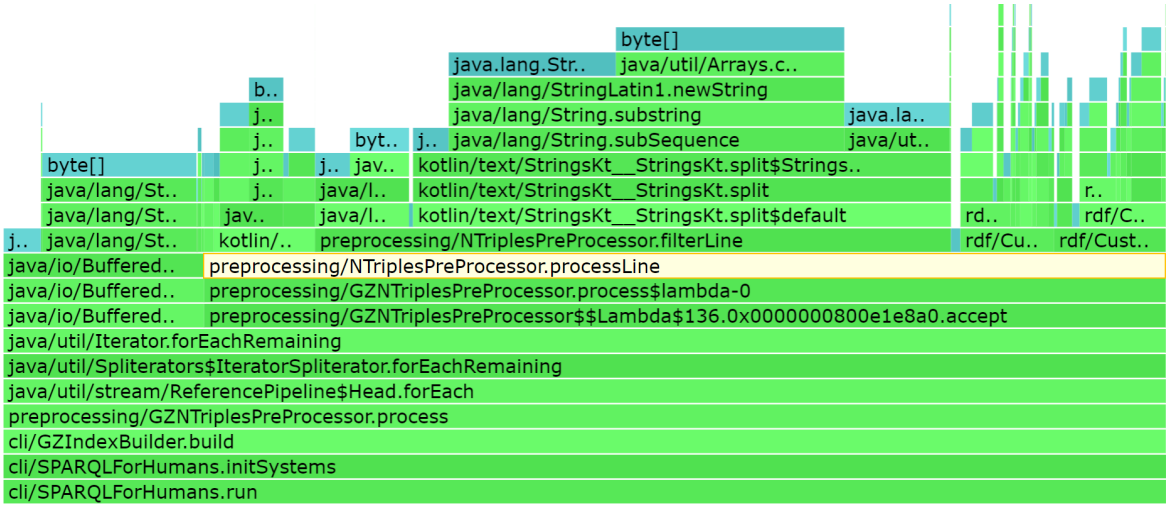


Figura 20: Reporte generado por *Async-Profiler*.
Fuente: Elaboración Propia.

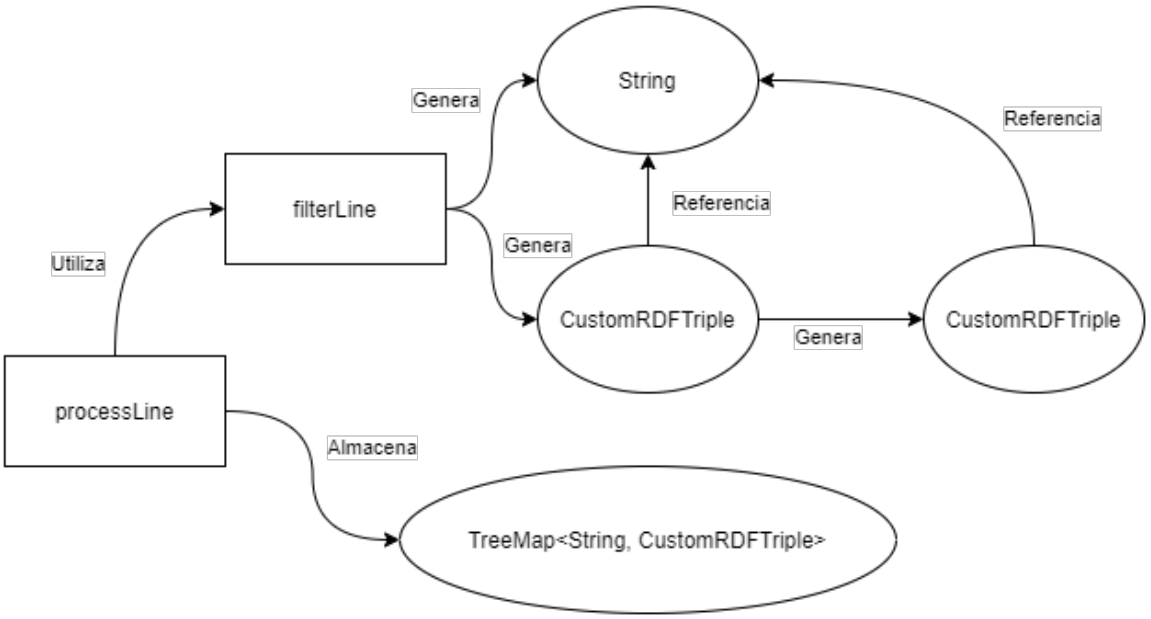


Figura 21: Generación de objetos `CustomRDFTriple`.
Fuente: Elaboración Propia.

- La mayor parte de los objetos corresponden a `Strings` o `byte[]`.
- Los objetos generados están fuertemente relacionados a operaciones de separación, subsecuencias y concatenación de cadenas de texto (`String`).

Ahora, con la definición del proceso, el código fuente de nuestra implementación y esta nueva información podemos concluir que la causa de nuestro error *java.lang.OutOfMemoryError* es la existencia de múltiples objetos del tipo `String` que no han sido liberados de la memoria durante la ejecución de nuestra solución. Esto se puede observar en más detalle a través de la figura 21.

En la figura se puede observar que la función `processLine` utiliza a `filterLine` para generar objetos del tipo `CustomRDFTriple`, el cual, a su vez, puede generar otro objeto `CustomRDFTriple` en función de sus propiedades inversas. Estos, utilizan objetos `String` para interactuar con la biblioteca *Apache Jena* y obtener determinada información sobre los *Triples* que representan. Aquellos `Strings` son almacenados como propiedades de los `CustomRDFTriples`. Finalmente, ambos `CustomRDFTriples` son almacenados de forma ordenada en una instancia `TreeMap`.

Toda esta nueva información sobre las necesidades funcionales de la aplicación nos hace cuestionar la decisión de utilizar *Kotlin* y la *JVM* para procesar grandes cantidades de texto.

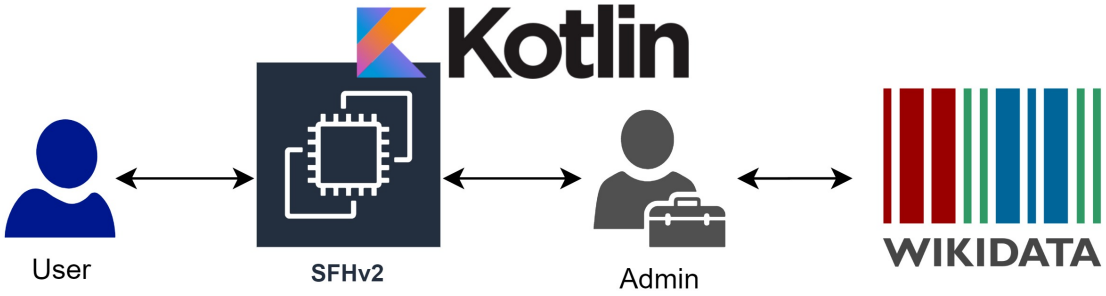


Figura 22: Diagrama de arquitectura para la solución.
Fuente: Elaboración Propia.

Resultados

No hemos logrado nuestro objetivo en esta iteración, debido a que no logramos ejecutar completamente nuestra solución. Sin embargo, no son malas noticias, hemos obtenido información valiosa para continuar en el desarrollo de la solución. En resumidas cuentas, lo siguiente:

- Debemos realizar una implementación que utilice una menor cantidad de memoria *RAM* del sistema.
- Debemos paralelizar nuestra solución, de esta forma, utilizaremos de forma más eficiente nuestra *CPU* y lograremos acercarnos aún más a nuestro objetivo.
- Existe una optimización importante en la etapa `(3 - PageRank)`, puesto que, se está utilizando una implementación iterativa del algoritmo. Funciona en pequeña escala, pero al utilizar una cantidad de 90 millones de nodos, definitivamente nos va a tomar tiempo procesar todo.

Con toda esta nueva información y nuevas necesidades, estamos listos para volver a intentar alcanzar nuestro objetivo de reducir en un $50\,\%$ nuestro tiempo de ejecución en una nueva iteración. Para terminar, un diagrama de la arquitectura actual de nuestra solución se puede apreciar en la figura 22.

3.5. Tercera iteración: Múltiples componentes, *FlatBuffers* y *Message Brokers*

Planificación

Para la tercera iteración, vamos a revisar y desacoplar los componentes de nuestro diseño anterior en base a los nuevos conocimientos e información obtenida durante la implementación de la iteración anterior. Con esto, buscamos obtener importantes mejoras de rendimiento y eficiencia en la utilización de recursos.

Nuestro objetivo es el mismo que el de la segunda iteración, reducir el tiempo de ejecución de la solución en al menos un $50\,\%$. Para lograr esto, planeamos realizar los siguientes cambios:

- Debemos mejorar el rendimiento de las etapas `(1.a)`, `(1.b)`, `(1.c)`, `(1.d)`. Para lograr esto, buscamos utilizar un lenguaje de programación que nos permita utilizar de forma más eficiente los recursos disponibles.
- La implementación actual de la etapa `(3)` utiliza un algoritmo secuencial para obtener los valores que se deben asignar a los documentos en el índice *Lucene*. En vez de utilizar un algoritmo propio, buscaremos usar una herramienta existente que nos permita obtener resultados de forma más rápida y eficiente.
- Para comunicar estos nuevos componentes, vamos a utilizar un formato binario más eficiente que el actual formato de texto *N-Triples* comprimido en *gzip*.

Implementación

3.5.1. Procesamiento de texto

Comenzaremos con una nueva implementación de las etapas `(1.a)` hasta la `(1.d)` utilizando el lenguaje de programación *GoLang*,[22]. *GoLang* es conocido por su gran eficiencia, similar a la de *C/C++*, su gran capacidad para manejar procesos altamente concurrentes y la simplicidad de su escritura. Estas características lo vuelven una gran alternativa para realizar nuestra tarea de procesar $45\texttt{GB}$ de texto comprimido en formato *gzip*.

Además, aprovecharemos la funcionalidad de la biblioteca *pgzip*[23] la cual logra descomprimir

[22] *The Go Programming Language*.https://golang.org/
[23] https://github.com/klauspost/pgzip

buffers[24] de bloques *gzip* de forma paralela. Esto nos permite procesar múltiples secciones de nuestro fichero *N-Triples* de forma paralela.

El nuevo flujo es el siguiente, una vez que hemos validado, procesado y generado nuevas propiedades para una cantidad determinada de líneas *N-Triple* que describen a una entidad, podemos combinar los objetos generados en una lista, llamaremos a esta lista "grupo". Luego, este grupo es enviado a otra rutina del programa encargada de almacenarlo en un nuevo fichero, el cual, contiene toda la información necesaria para construir los índices a excepción del valor *PageRank* de la entidad descrita por el grupo. Es importante destacar, que la rutina encargada de almacenar los grupos también se ejecuta de forma paralela al procesamiento y descompresión del fichero original. Este nuevo fichero es una representación compacta y enriquecida de la información incluida en el archivo *N-Triple* original.

3.5.2. Serialización binaria

Para entender de forma más clara nuestra representación compacta, vamos a presentar los procesos de serialización. Los procesos de serialización buscan transformar el estado actual en memoria de determinadas estructuras de datos u objetos en un formato que puede ser almacenado (como un fichero o *buffer* de datos) o transmitido (por ejemplo, a través de una red) para luego ser reconstruido en un ambiente diferente. Un formato de serialización ampliamente utilizado es el formato *JSON*[25]. Un ejemplo de este formato se puede observar en la figura 23.

Existen múltiples formatos de serialización disponibles, pero nos centraremos en algunas alternativas bastante utilizadas y en recomendaciones de la literatura existente [Proos and Carlsson, 2020] [Khare, 2018]. Revisaremos las siguientes alternativas:

- *JSON* es un formato ligero para el intercambio de datos, fácil de leer y escribir para los humanos y fácil de generar e interpretar para las maquinas. Está basado en un subconjunto del lenguaje de programación "*JavaScript Programming Language Standard ECMA-262 3rd Edition - December 1999*". *JSON* es un formato de texto completamente independiente del lenguaje de programación utilizado para generarlo e interpretarlo, pero utiliza convenciones que son familiares para aquellos programadores que han utilizado lenguajes como *C/C++*, *Java*, *Python* entre otros. Estas características hacen de *JSON* un formato ideal para intercambiar datos entre distintos equipos y ambientes.
- *ProtocolBuffers*[26] es un formato de serialización binario desarrollado y utilizado por *Google*. Desde su versión 2, el protocolo ha sido publicado de forma abierta para su

[24] Espacio de memoria en el cual se almacenan datos de forma temporal para ser utilizados por un proceso determinado. Por lo general, los datos almacenados provienen de dispositivos de entrada y salida, como por ejemplo, discos de almacenamiento o interfaces de red.

[25] *JavaScript Object Notation* https://www.json.org/

[26] https://google.github.io/flatbuffers/

```
{
    "firstName": "John",
    "lastName": "Smith",
    "address": [
        {
            "streetAddress": "21 2nd Street",
            "city": "New York",
            "state": "NY",
            "postalCode": 10021
        },
        {
            "streetAddress": "577 Airport Blvd",
            "city": "Burlingame",
            "state": "CA",
            "postalCode": 94010
        }
    ],
    "phoneNumbers": [
        "212-732-1234",
        "646-123-4567"
    ]
}
```

Figura 23: Ejemplo de serialización en formato *JSON*.
Fuente: Elaboración propia.

utilización. El proyecto se describe a sí mismo como “un formato extensible para la serialización de datos estructurados independiente de la plataforma y lenguaje donde se utiliza”. Esto se logra a través de la definición de un esquema para determinar la estructura de los datos que son almacenados al momento de serializar la información.

- *FlatBuffers*[27], al igual que *ProtocolBuffers*, es desarrollado y utilizado por *Google* y utiliza un esquema para definir la estructura de los datos serializados. El formato definido por *FlatBuffers* hace un énfasis importante en una metodología orientada a estructurar los datos de la misma forma en la que estos se encuentran en la memoria del sistema que los está procesando. En la implementación, se usa una técnica orientada a punteros en memoria, en la cual, cada propiedad de datos se encuentra en una determinada posición de un *buffer* de datos. Esta técnica, nos permite eliminar aquellos procesos de codificación y decodificación para interactuar con los datos, permitiendo realizar lecturas directas a valores serializados.

Para decidir qué formato vamos a utilizar, vamos a considerar las siguientes dimensiones:

- Velocidad.

  Para esta dimensión, podemos analizar los resultados obtenidos por [Khare, 2018] presentados en las figuras 24 y 25 para los procesos de serialización y deserialización respectivamente. En estos resultados, se puede observar la gran diferencia en velocidad de operación entre los formatos binarios y *JSON*. Los autores [Proos and Carlsson, 2020] realizan un análisis más detallado en la comparación de *FlatBuffers* y *Protobuf* en la figura 28. Los resultados son claros, los formatos binarios logran una mejor velocidad a la hora de realizar procesos de serialización y deserialización. *FlatBuffers* logra una pequeña ventaja contra *Protobuf* debido a su técnica de ordenar la información en base a la representación en memoria del objeto serializado mencionado anteriormente.

- Memoria utilizada.

  En términos de la memoria utilizada por los formatos binarios, *FlatBuffers* lleva la delantera, esto se puede observar en los resultados obtenidos por [Proos and Carlsson, 2020] que se pueden apreciar en las figuras 26 y 27.

- Tamaño del mensaje generado.

  Cuando revisamos el tamaño de los mensajes generados en la figura 28, podemos notar que esta vez *Protbuf* es el formato ganador.

Para nuestra aplicación, el orden de prioridad en las dimensiones mencionadas es: memoria utilizada, seguida de velocidad y finalmente tamaño de los mensajes. En base a esto, hemos decido utilizar *FlatBuffers* para nuestro proceso de serialización. El esquema ha utilizar para definir un grupo se puede observar en la figura 29.

[27] https://developers.google.com/protocol-buffers

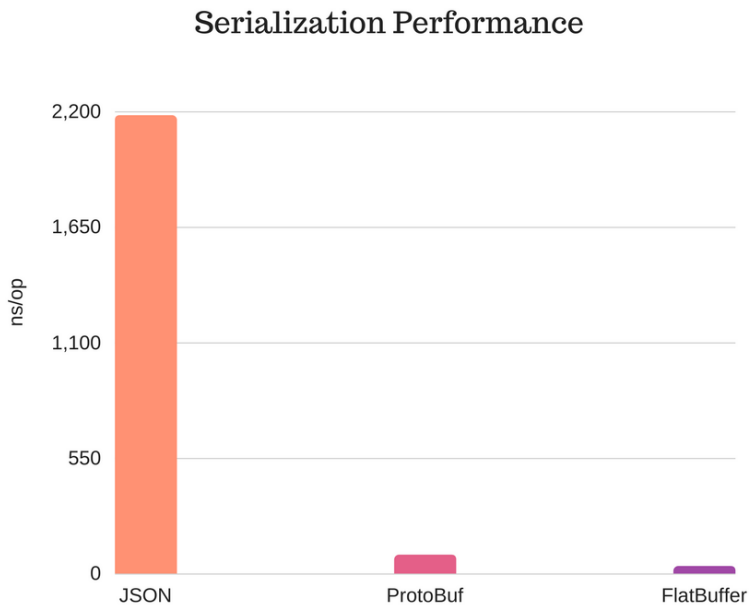


Figura 24: Operaciones en el proceso de serialización *JSON* vs *Protobuf* vs *FlatBuffers*. Menos es mejor. Fuente: [Khare, 2018].

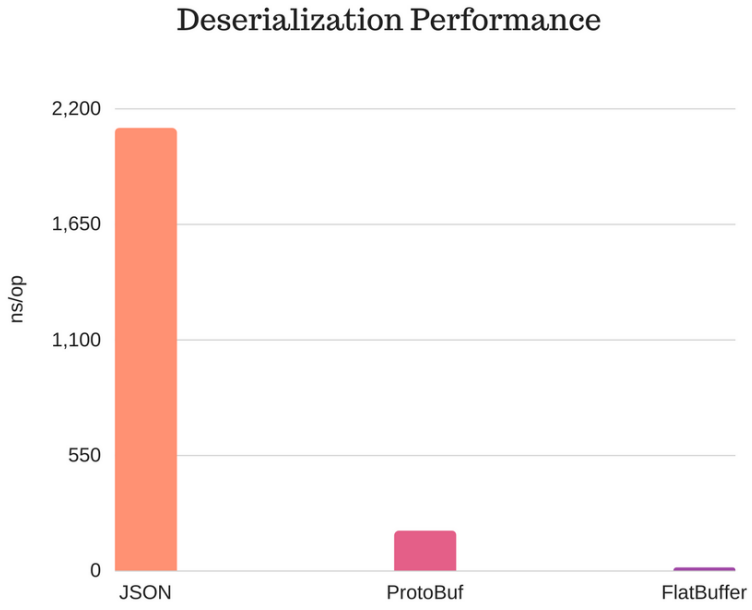
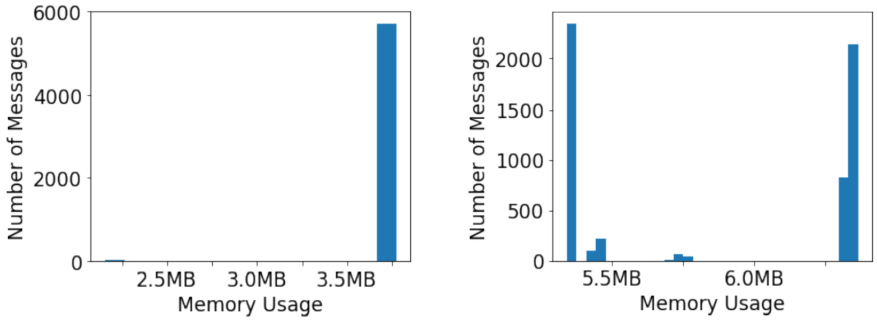
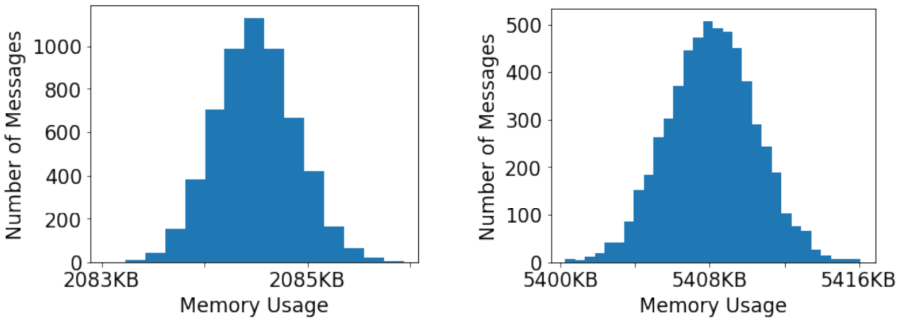


Figura 25: Operaciones en el proceso de deserialización *JSON* vs *Protobuf* vs *FlatBuffers*. Menos es mejor. Fuente: [Khare, 2018].



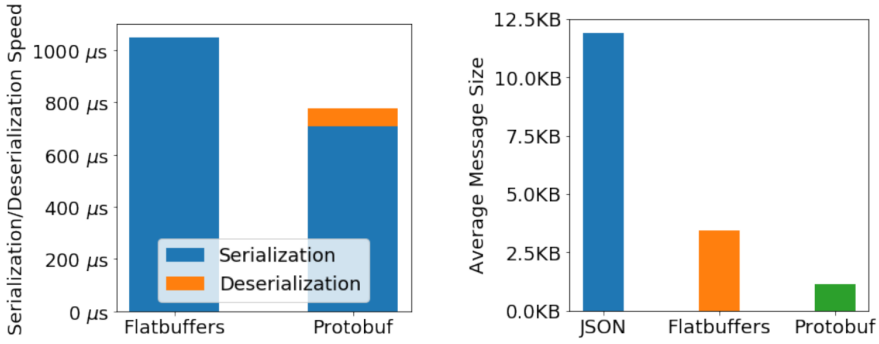
(a) Flatbuffers. (b) Protobuf.

Figura 26: Memoria utilizada en el proceso de serialización *Protobuf* vs *FlatBuffers*. Fuente: [Proos and Carlsson, 2020].



(a) Flatbuffers. (b) Protobuf.

Figura 27: Memoria utilizada en el proceso de deserialización *Protobuf* vs *FlatBuffers*. Fuente: [Proos and Carlsson, 2020].



(a) Serialization speed. (b) Serialized message size.

Figura 28: Velocidad y tamaño de mensajes generados *JSON* vs *Protobuf* vs *FlatBuffers*. Fuente: [Proos and Carlsson, 2020].

```
namespace sparqlforhumans;

enum AttrType: byte {
    DirectProp = 0,
    AltLabel,
    Label,
    Description,
    ReverseProp,
    QObject,
    Literal,
    InstanceOfProp,
    InverseInstanceOfProp,
    SubClassOfProp,
    PObject,
    Unknown,
}

table Relation {
    predid: uint;
    objid: uint;
    literal: string;
    predtype: AttrType;
    predsubtype: AttrType;
    objtype: AttrType;
}

table Group {
    sid: uint;
    stype: AttrType;
    rank: float;
    triples: [Relation];
}

struct PageRankItem {
    id: uint;
    rank: float;
}

table PageRankResponse {
    values: [PageRankItem];
}

root_type Group;
```

Figura 29: Esquema utilizado para la serialización en formato binario a través de *FlatBuffers*
Fuente: Elaboración propia.

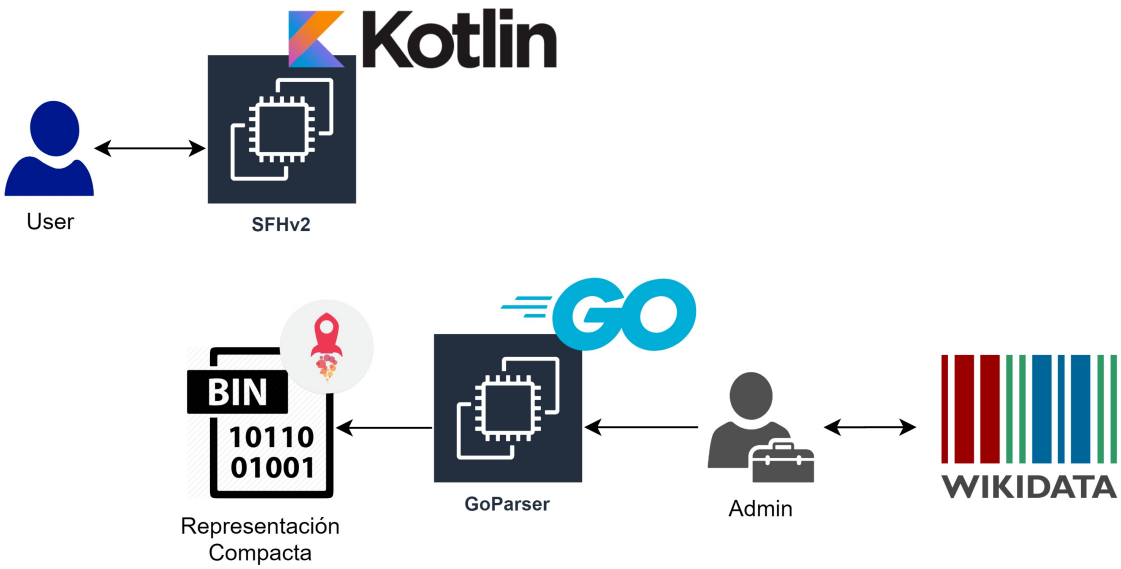


Figura 30: Diagrama de arquitectura para la solución.
Fuente: Elaboración Propia.

3.5.3. Generación de grupos en formato *FlatBuffers*

Con la definición anterior y en conjunto con el nuevo componente encargado de procesar el fichero *gzip* hemos obtenido la habilidad de procesar el total de $45GB$ que componen nuestro fichero de entrada, con aproximadamente $5{,}8$ mil millones de líneas representando 92 millones de grupos. En nuestro ambiente productivo hemos generado la representación en un tiempo aproximado de $50m$. El fichero generado tiene un tamaño total de $43GB$ y contiene toda la información necesaria para generar ambos índices *Lucene* sin considerar los valores del algoritmo *PageRank*. El estado actual de nuestra arquitectura se puede observar en la figura 30.

3.5.4. PageRank

La última propiedad que necesitamos para generar los índices *Lucene* es el valor que el algoritmo *PageRank* asigna a cada grupo en base a su importancia en el conjunto de datos. Actualmente, este valor es calculado de forma secuencial, nodo por nodo. El grafo generado en la representación compacta contiene un total de 92 millones de nodos, por lo necesitamos una nueva estrategia para calcular este algoritmo de forma más eficiente.

Para esto, vamos a utilizar la biblioteca *graph-tool*[28]. *graph-tool* es una herramienta eficiente para la manipulación y el análisis estadístico de grafos en *Python*. Sus características princi-

[28]https://graph-tool.skewed.de/

Algorithm	graph-tool (16 threads)	graph-tool (1 thread)	igraph	NetworkX
Single-source shortest path	0.0023 s	0.0022 s	0.0092 s	0.25 s
Global clustering	0.011 s	0.025 s	0.027 s	7.94 s
PageRank	0.0052 s	0.022 s	0.072 s	1.54 s
K-core	0.0033 s	0.0036 s	0.0098 s	0.72 s
Minimum spanning tree	0.0073 s	0.0072 s	0.026 s	0.64 s
Betweenness	102 s (∼1.7 mins)	331 s (∼5.5 mins)	∼10.6 mins	∼6.7 hours

Tabla 2: Rendimiento de herramientas para la manipulación de grafos en *Python*. Fuente: Graph-tool performance comparison. https://graph-tool.skewed.de/performance

pales son las siguientes:

- Velocidad. El núcleo de las estructuras de datos y algoritmos que provee la biblioteca están implementados en *C++*, en la mayoría de los casos, el rendimiento obtenido al utilizar la biblioteca en *Python* es igual que si se utilizara la versión nativa de *C++*.
- Paralelismo. Muchos de los algoritmos disponibles en la biblioteca han sido implementados de forma paralelizada utilizando *OpenMP*[29], lo que permite obtener un excelente rendimiento en ambientes con múltiples procesadores.

Una comparación del rendimiento de *graph-tool* y dos otras bibliotecas ampliamente utilizadas en *Python* se puede observar en la tabla 2. Para la prueba, se ha utilizado un grafo dirigido con $39{,}796$ nodos y $301{,}498$ arcos.

Con estas características en mente, hemos implementado el siguiente proceso utilizando *Python* y *graph-tool*:

1. Leemos el fichero que contiene la representación compacta grupo por grupo, generando un nodo y sus correspondientes arcos por cada identificador nuevo encontrado en el grupo o en alguna de sus relaciones.
2. Una vez que hemos leído todo el fichero tenemos un grafo dirigido sobre el cual ejecutamos el algoritmo *PageRank* a través de *graph-tool*.
3. Lo siguiente es obtener los resultados del algoritmo y serializarlos en formato *JSON* para facilitar su utilización en otros componentes.

Los resultados preliminares en el ambiente de desarrollo no son muy alentadores. Al procesar un fichero de $200\texttt{MB}$ con aproximadamente $350k$ nodos necesitamos $4m40s$ para solo construir el grafo dirigido, nos estamos demorando demasiado tiempo.

[29] *OpenMP:* Es una interfaz de programación de aplicaciones *(API)*, conjunto de directivas para compiladores, bibliotecas y rutinas para facilitar la programación multiproceso con un modelo de memoria compartida en múltiples plataformas. https://www.openmp.org/

Sabemos que *graph-tool* es una biblioteca altamente optimizada, por lo que probablemente el problema se encuentra en la deserialización de nuestra representación compacta. Para atacar este problema, hemos decidido darle una oportunidad a la idea de paralelizar la lectura de grupos desde el fichero binario. Inicialmente utilizaremos dos hilos[30], uno de ellos será el encargado de deserializar la información compacta y enviar las relaciones obtenidas a una cola *FIFO*[31], el segundo hilo es el encargado de obtener estos mensajes desde la cola *FIFO*, construir el grafo y ejecutar el algoritmo *PageRank*. Una representación gráfica de este componente se puede observar en la figura 31.

Los nuevos resultados no son muy alentadores. Hemos aumentado en un $14\,\%$ el tiempo necesario para construir el grafo a $5m20s$, por lo que no es suficiente. Con este resultado, como programadores, podríamos caer en la tentación de agregar múltiples hilos a nuestro programa, sin embargo, debemos recordar la existencia del *Global Interpreter Lock (GIL)* [Beazley, 2010] [Eggen and Eggen, 2019] en *Python*. El *GIL* es un seguro que le permite solo a un hilo a la vez acceder al estado de los objetos declarados en un programa y debido a que el proceso de serialización requiere utilizar tanto recursos de entrada y salida (a través de un fichero) como también poder de procesador, no podremos aprovechar el potencial de múltiples núcleos utilizando solamente hilos.

Para este tipo de tareas, en las cuales necesitamos tanto poder de procesamiento como también velocidad de lectura y escritura, debemos utilizar múltiples procesos en vez de múltiples hilos. Esto nos trae otra serie de complicaciones, como por ejemplo, compartir el estado actual de determinadas estructuras de datos u objetos junto con la sincronización, comunicación y coordinación entre estos procesos.

En nuestro caso y para mantener relativamente simple la interacción entre los procesos, hemos definido el siguiente esquema. Utilizaremos procesos a los cuales se les asignará una determinada sección del fichero binario, cada proceso debe leer y enviar la información sobre las relaciones encontradas a una cola única. Esta cola es de tipo *FIFO* y aquellos elementos enviados a ella serán procesados por el proceso principal, el cual, es el encargado de construir el grafo y ejecutar el algoritmo *PageRank*. Una representación gráfica de este componente se puede observar en la figura 32.

Los resultados de esta nueva versión son sorprendentes, al utilizar nuestro fichero de $200MB$, 3 procesos para la lectura binaria y un proceso para la generación del grafo, nos hemos demorado $25m$, un $530\,\%$ más que la versión inicial, en procesar y construir todo el grafo, lo cual es terrible.

Creemos que la gran diferencia en tiempo se debe a los mecanismos internos de sincronización utilizados por *Python* para enviar los objetos generados por los distintos lectores hacia

[30] Un hilo o subproceso corresponde a un conjunto de instrucciones que describen un flujo dentro de un proceso. Un proceso puede contener múltiples hilos y estos hilos pueden ser ejecutados de forma paralela por la *CPU*.

[31] *FIFO*. Del inglés *First In First Out*, corresponde a un tipo de cola en la cual los mensajes con mayor antigüedad dentro de la cola son aquellos que se entregan primero a su destino.

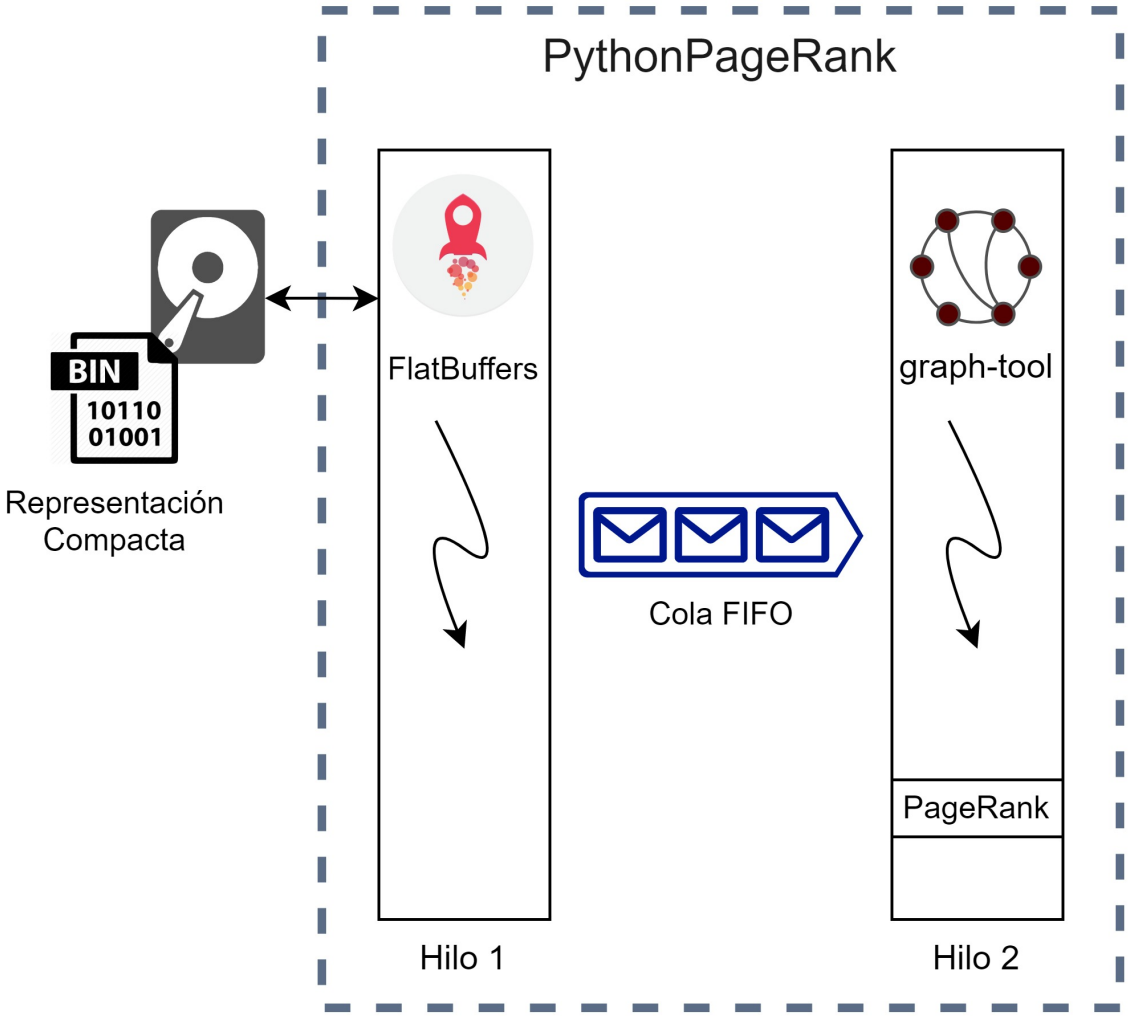


Figura 31: Estrategia de múltiples hilos.
Fuente: Elaboración Propia.

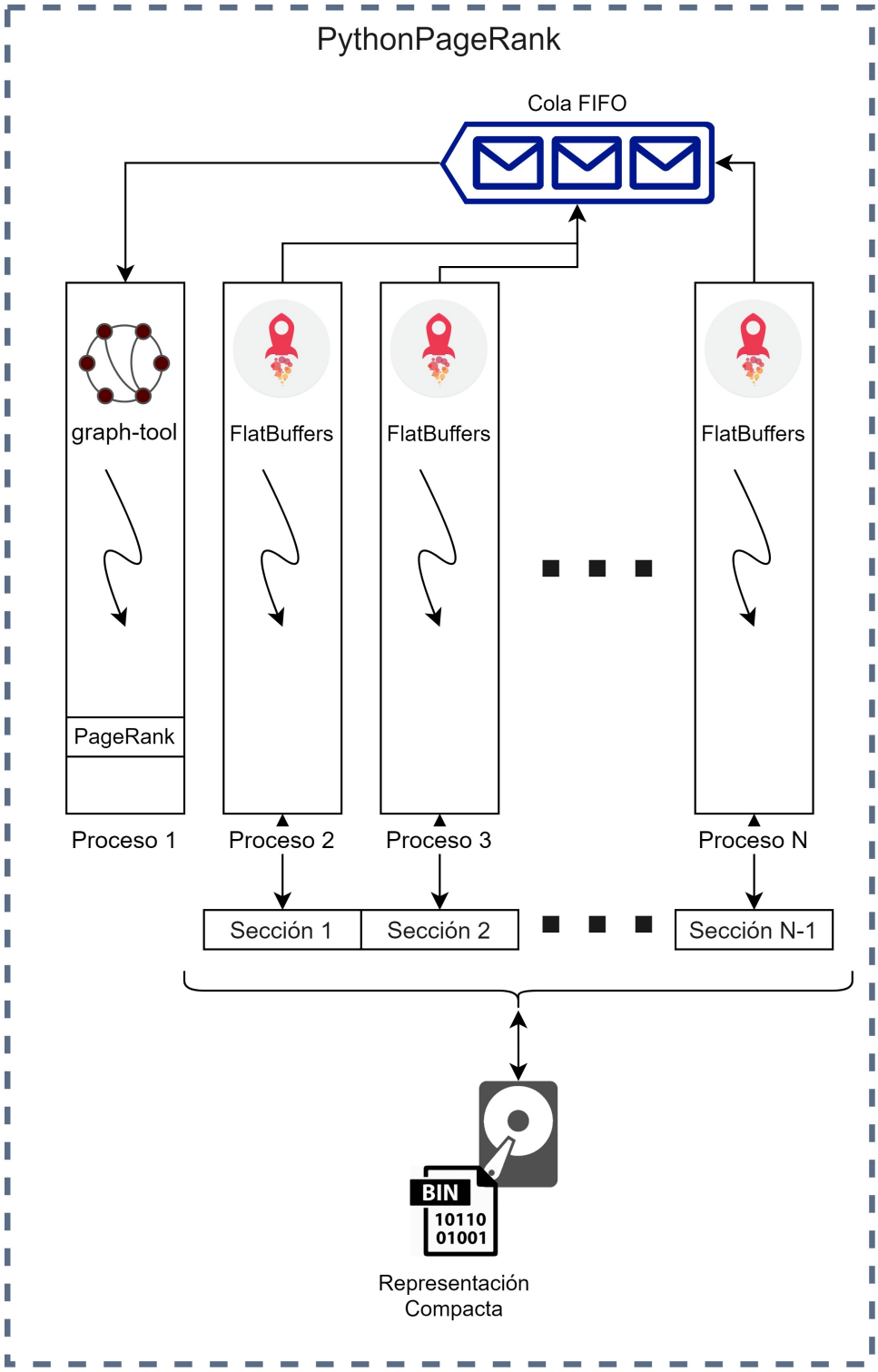


Figura 32: Estrategia de múltiples procesos.
Fuente: Elaboración Propia.

Tipo de Metrica	SingleThread	ThreadedReader	MultiProcess
Real	4m34.778s	5m17.012s	25m31.738s
User	4m34.246s	5m39.624s	32m26.096s
Sys	0m1.570s	0m21.920s	28m48.407s

Tabla 3: Tiempos en *CPU* para distintas estrategias de construcción de grafos. Fuente: Elaboración propia.

el proceso principal. Para medir esto, vamos a utilizar el comando `time` [ope, 2018] de *Unix*, este comando, nos entrega tres métricas clave sobre cómo estamos utilizando el procesador y cómo estamos interactuando con el sistema operativo. Las métricas son las siguientes:

- *Real*. Tiempo desde el inicio hasta el fin de la ejecución de un programa.
- *User*. Tiempo que la *CPU* se ha encontrado ejecutando nuestro programa en *User Space*.[32].
- *Sys*. Tiempo que la *CPU* se ha encontrado ejecutando nuestras peticiones al sistema operativo *(Kernel Space)* a través de *syscalls*[33].

Al tomar las métricas *User* y *Sys* podemos obtener una mirada de cuanta *CPU* estamos utilizando y en que tarea se esta utilizando. En la tabla 3 se pueden apreciar los valores obtenidos.

El problema es claro, si bien, tiene sentido utilizar una estrategia del tipo “dividir y conquistar” para atacar nuestro problema de rendimiento, al parecer la implementación utiliza una cantidad excesiva de recursos del sistema operativo. Para profundizar más aún en el problema, podemos revisar cuales son los recursos que estamos solicitando al sistema operativo utilizando la herramienta `strace` [Kerrisk, 2018].

Al utilizar `strace` de la forma más básica podemos obtener un reporte de todas las *syscalls* que ha realizado un determinado programa, en el cual, podemos observar el total del tiempo ejecutado, la cantidad de llamadas ejecutadas y una aproximación del tiempo que el sistema operativo demora en completar una llamada. Los datos obtenidos se pueden observar en el anexo 5.1. En base los datos generados podemos construir los gráficos presentados en las figuras 33 a la 38.

[32]Los sistemas operativos modernos, por lo general, separan la memoria disponible en dos secciones, una para el núcleo del sistema o *kernel* y otra para las aplicaciones de los usuarios. Esta división permite proteger determinada información y acceso a hardware de usuarios maliciosos o software problemático. Cuando hablamos de *kernel space* nos referimos a la sección de memoria, funciones y procesos internos del núcleo del sistema operativo que permiten su correcto funcionamiento y al hablar de *user space* hacemos referencia al resto de componentes que se ejecutan en el sistema.

[33]Una llamada al sistema (del inglés *system call* o *syscall*) es una forma programática por la cual una aplicación en *user space* solicita al sistema operativo realizar alguna funcionalidad determinada. Algunas de estas funcionalidades incluyen acceder a información desde un dispositivo de almacenamiento, enviar un paquete de datos por una interfaz de red, iniciar un nuevo proceso, entre otras. Las *syscalls* son una interfaz esencial entre un proceso y el sistema operativo en el que este se ejecuta.

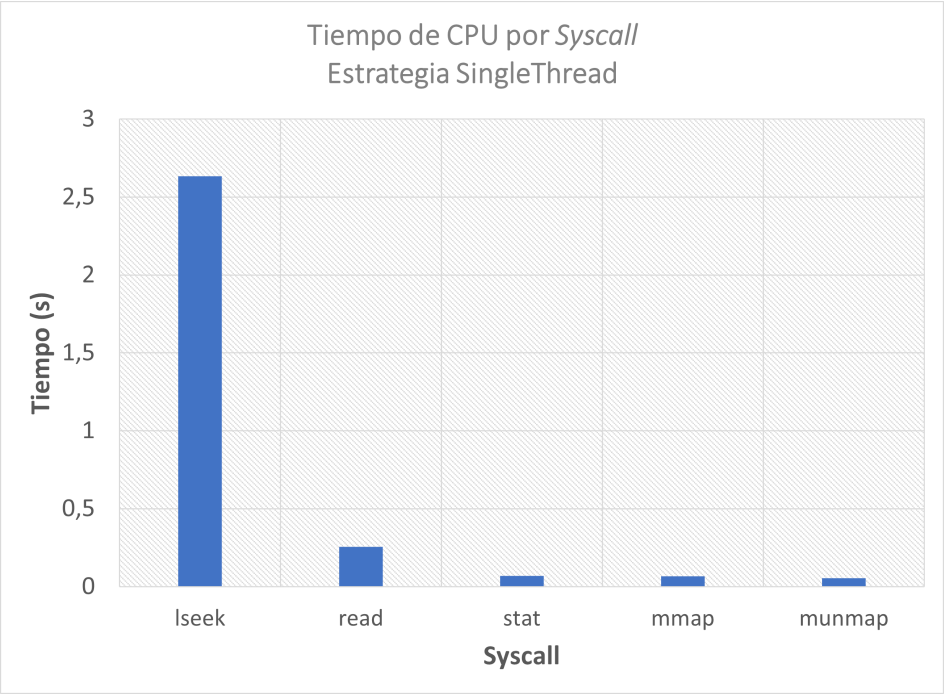


Figura 33: Tiempo de *CPU* por *syscall* para la estrategia *SingleThread*
Fuente: Elaboración Propia.

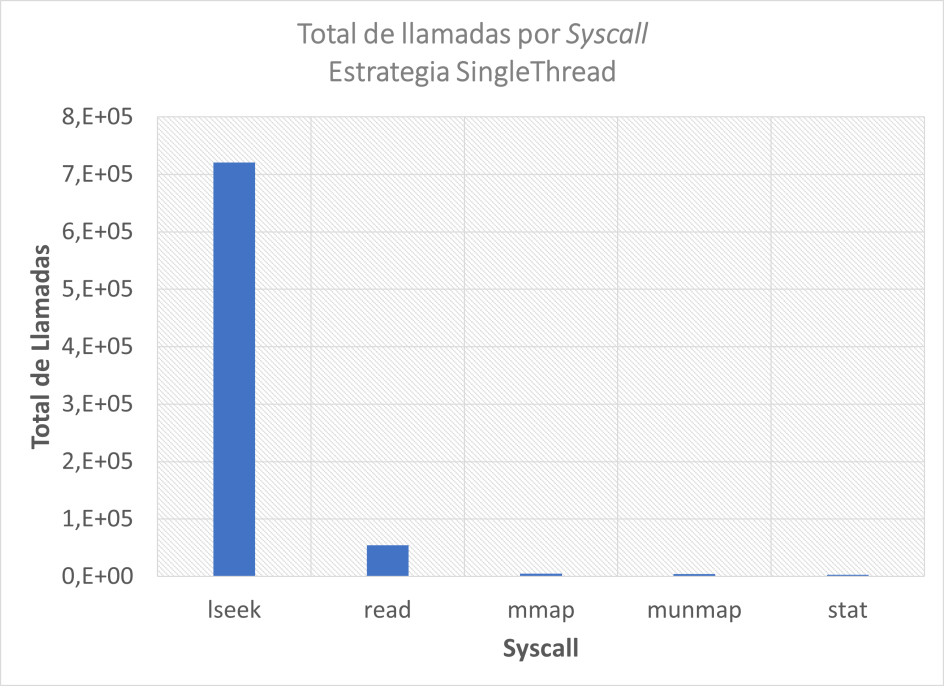


Figura 34: Total de llamadas por *syscall* para la estrategia *SingleThread*
Fuente: Elaboración Propia.

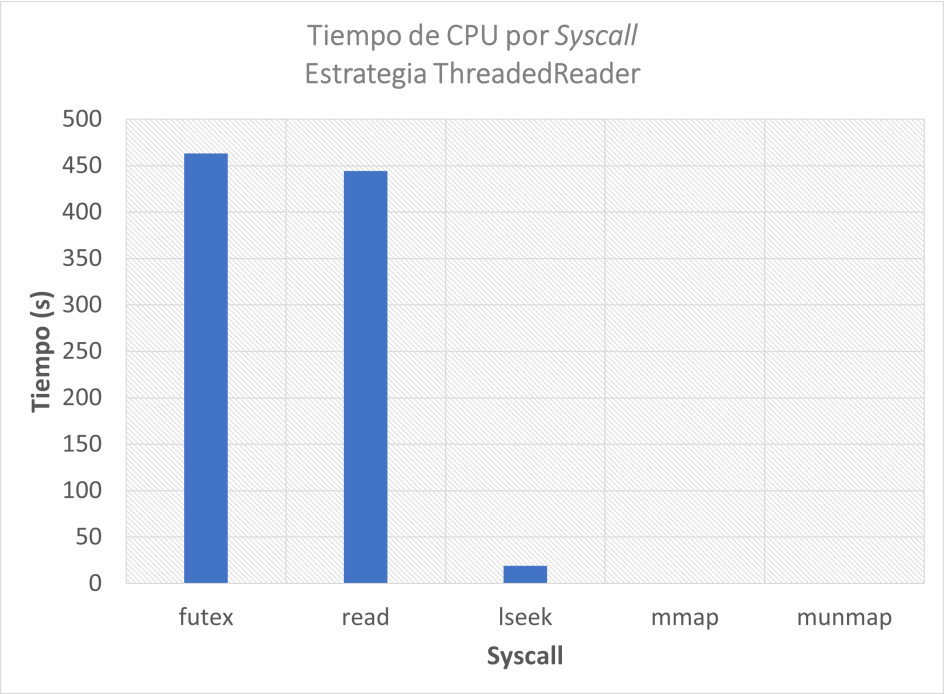


Figura 35: Tiempo de *CPU* por *syscall* para la estrategia *ThreadedReader*
Fuente: Elaboración Propia.

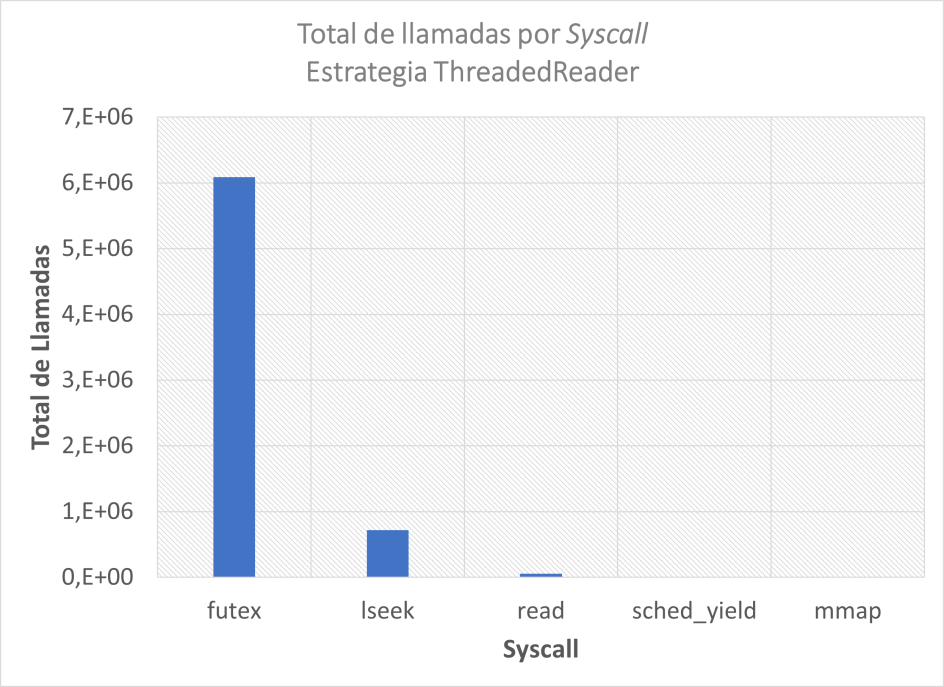


Figura 36: Total de llamadas por *syscall* para la estrategia *ThreadedReader*
Fuente: Elaboración Propia.

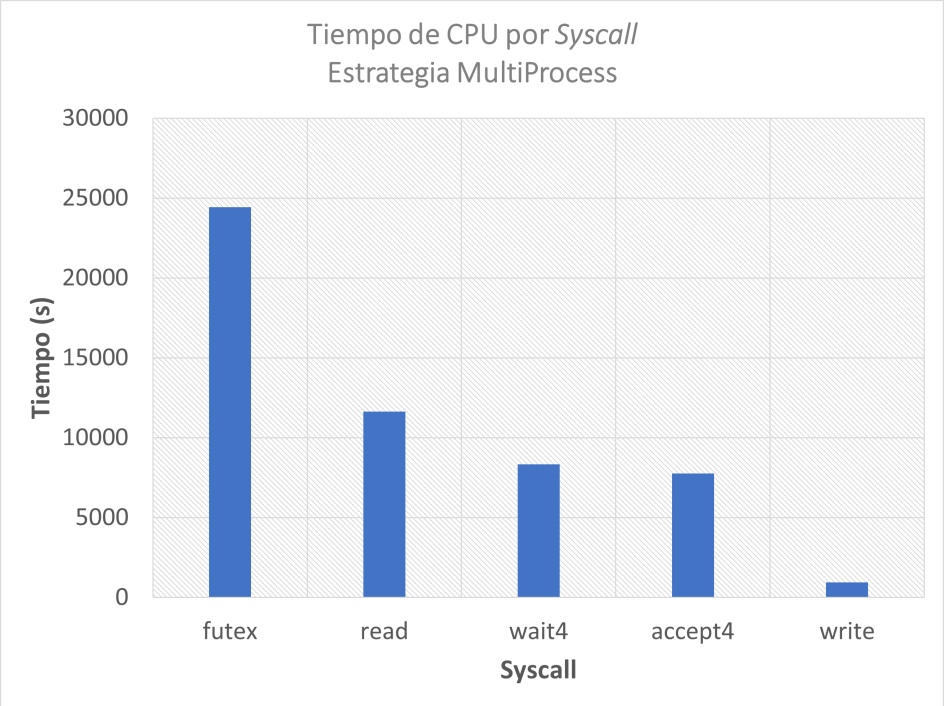


Figura 37: Tiempo de *CPU* por *syscall* para la estrategia *MultiProcess*
Fuente: Elaboración Propia.

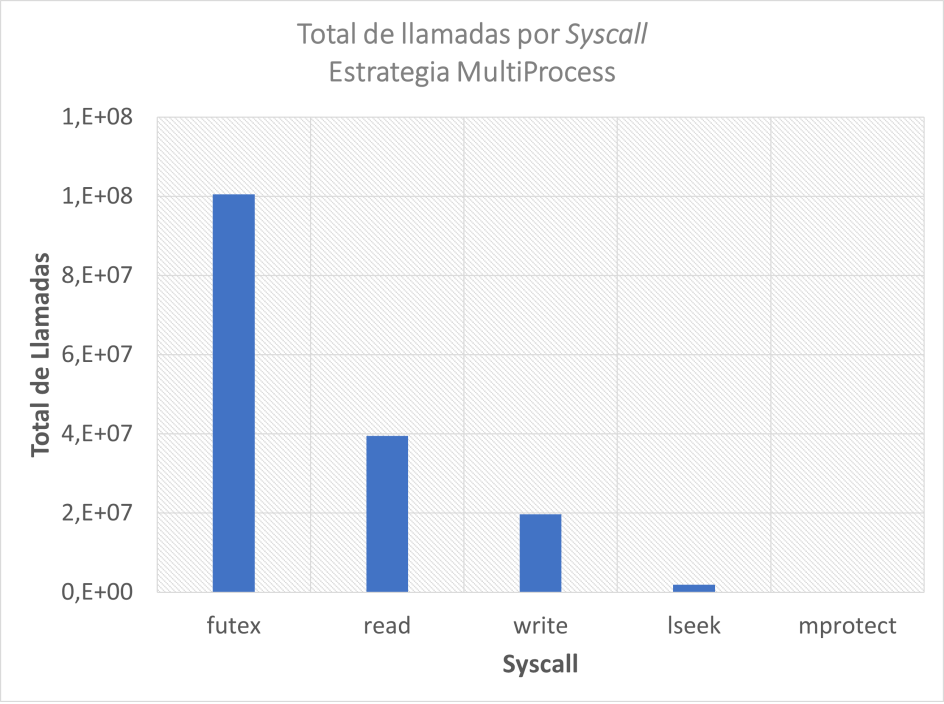


Figura 38: Total de llamadas por *syscall* para la estrategia *MultiProcess*
Fuente: Elaboración Propia.

En la figura 33, para la estrategia *SingleThread*, podemos encontrar que la mayor parte del tiempo que interactuamos con el sistema operativo es a través de llamadas a *lseek* y *read*. La *syscall lseek* nos permite ubicarnos en una posición determinada de un fichero utilizado, en nuestro caso, utilizamos esta llamada para recorrer nuestro fichero binario de representación compacta. Además, de forma natural, utilizamos *read* para obtener el contenido del fichero binario. Debido a que estas dos *syscalls* son las más utilizadas según la figura 34, no hay mucho espacio para optimizaciones por lo que no podríamos obtener mejores resultados con esta estrategia.

Al revisar la estrategia *ThreadedReader* en las figuras 35 y 36 podemos observar los efectos del *Global Interpreter Lock* de *Python* a través de la *syscall futex*. La llamada *futex* nos provee un mecanismo para esperar por una determinada condición durante la ejecución de un programa, en nuestro caso, es utilizada para sincronizar el estado global del interprete *Python* entre los hilos generados. Debido a este efecto, no es conveniente generar más hilos para paralelizar el procesamiento de la representación compacta, puesto que aumentariamos el impacto de *futex* y no obtendriamos el rendimiento esperado.

En el caso de la estrategia *MultiProcess*, podemos notar en las figuras a y b dos propiedades importantes, la primera, es la excesiva cantidad de llamadas a *futex*. Debido a que estamos utilizando procesos, no deberíamos tener problemas de exclusión mutua, por lo que al parecer estamos utilizando hilos en alguno de los componentes de la solución que no controlamos. Lo segundo que podemos observar es la inesperada cantidad de llamadas a la *syscall write*, esto, debido a que en ninguno de los flujos de esta estrategia estamos escribiendo a algún fichero u otro tipo de objeto.

Debido a que el principal cambio entre las distintas estrategias se encuentran en la comunicación y sincronización de los distintos hilos y procesos y junto a los resultados obtenidos por los reportes presentados anteriormente, podemos afirmar que el problema de la estrategia *MultiProcess* se encuentra en la comunicación entre procesos a través de la clase `multiprocessing.Queue` de *Python*. Con esto hemos obtenido un nuevo requerimiento, necesitamos un componente que nos permita comunicar múltiples procesos o componentes que se ejecutan en paralelo de forma eficiente y con un bajo impacto al rendimiento de estos. Este tipo de aplicaciones son conocidas como *message brokers*.

3.5.5. *Message Brokers*

Un *message broker* corresponde a una pieza de *software* que permite a aplicaciones, servicios y sistemas comunicarse entre ellos e intercambiar información. Esto se logra a través de un proceso de traducción de los mensajes que son enviados entre los distintos actores que se comunican con el *message broker*. Esta funcionalidad permite que múltiples servicios independientes logren comunicarse entre ellos incluso si han sido implementados en distintos lenguajes de programación y en distintas plataformas de ejecución. [Education, 2020]

Los *message brokers* son parte del grupo de intermediarios *(middlewares)* llamados *Message Oriented Middleware (MOM)*. Este tipo de *middleware* le permite a desarrolladores controlar flujos de información entre aplicaciones de forma estandarizada, permitiendo que nos podamos centrar en la lógica de los servicios y no en la problemática de la comunicación entre ellos.

Además, este tipo de componentes nos permiten validar, almacenar, enrutar y distribuir mensajes a múltiples destinos. Esto permite que aquellos componentes que generan mensajes puedan enviarlos sin necesidad de conocer quién va a recibirlos, sin saber si estos destinos se encuentran activos ni cuantos de estos destinos existen, de esta forma, se logra desacoplar múltiples procesos y servicios dentro de un sistema. Para proveer un almacenamiento de mensajes confiable y un despacho asegurado de estos, los *message brokers* utilizan colas de mensajes, las cuales, almacenan de forma ordenada los mensajes recibidos hasta que puedan ser consumidos en su totalidad por los destinos configurados.

Existen dos grupos básicos para la distribución de mensajes en los *message brokers*:

1. Punto a punto.

   Este patrón es utilizado en situaciones donde necesitamos una cola cuya relación es uno a uno entre el componente que envía el mensaje y el componente que lo recibe. Cada mensaje en la cola es enviado solo una vez y es consumido solo una vez por su receptor.

2. Publicación y suscripción.

   En este patrón de distribución, también conocido como *Pub/Sub*, el componente que genera un mensaje publica en un “tema” y múltiples consumidores se pueden suscribir a varios temas desde los cuales buscan recibir mensajes. Cada mensaje que es enviado a un tema es distribuido a todos sus suscriptores. En este patrón la relación en entre generador y receptor es uno a muchos.

Existen muchas soluciones en el mundo de los *message brokers* de código abierto, algunos de ellos son *RabbitMQ* [VMWare, 2021], *Redis* [Ltd, 2021], *Apache ActiveMQ* [Foundation, 2021a], *ZeroMQ* [authors, 2021], *Apache Kafka* [Foundation, 2021b]. En nuestro caso, buscamos los siguientes requerimientos:

- Simple. Nuestro problema ya es complicado y buscamos que este nuevo componente nos permita lograr nuestro objetivo con mayor facilidad, por lo que necesitamos que sea lo más simple posible.
- Eficiente. Actualmente estamos consumiendo una cantidad importante de recursos de cómputo con nuestra aplicación, mientras menos recursos consuma este componente mejor.

Tipo de Metrica	SingleThread	ThreadedReader	MultiProcess	MultiProcessZMQ
Real	4m34.778s	5m17.012s	25m31.738s	2m58.207s
User	4m34.246s	5m39.624s	32m26.096s	9m33.923s
Sys	0m1.570s	0m21.920s	28m48.407s	0m53.238s

Tabla 4: Tiempos en *CPU* para distintas estrategias de construcción de grafos.
Fuente: Elaboración propia.

- Rápido. La solución actual que utiliza y comunica múltiples procesos demora más tiempo en ejecutarse que una solución sin comunicación. Es vital reducir ese tiempo.

Con estos requerimientos en mente, hemos decidido utilizar *ZeroMQ* para la comunicación entre procesos. Llamaremos a esta estrategia *MultiProcessZMQ* y su diseño se puede observar en la figura 39.

Al realizar las modificaciones necesarias hemos logrado reducir el tiempo del proceso de *PageRank* a $2m59s$ para nuestra prueba de $200MB$, esto representa una reducción del $34\,\%$ con respecto a la estrategia *SingleThread*. Las métricas obtenidas por el reporte *Syscall* se pueden observar en el anexo 5.1, tabla 13 y las figuras 40 y 41. Finalmente, la tabla 4 corresponde a la comparativa de tiempos para las estrategias anteriormente revisadas.

3.5.6. Actualización del valor *PageRank*

Actualmente hemos logrado calcular el valor asignado a cada documento de nuestro índice y estos valores han sido serializados en formato *JSON*, el siguiente paso necesario antes de iniciar la construcción de los índices en *Lucene* es actualizar nuestro fichero de representación compacta con los valores *PageRank* obtenidos. Para esto, debemos cargar el documento *JSON* generado recientemente, iterar sobre cada uno de los grupos almacenados en la representación compacta, actualizar su valor y finalmente escribir el resultado. Este proceso de actualización es relativamente simple y demora aproximadamente $15m$ para el total de $43\texttt{Gb}$ en nuestro ambiente de producción. El componente encargado de esta tarea fue desarrollado en `GoLang`.

3.5.7. Construcción de índices

La última tarea que debemos realizar corresponde a generar los documentos en ambos índices de *Lucene*, entidades y propiedades. En este punto y debido a que hemos realizado todas las preparaciones necesarias a través de nuestro fichero compacto podemos iterar para cada uno de los grupos definidos en el fichero y generar su documento respectivo en *Lucene*. En nuestro ambiente productivo, esta etapa toma un tiempo de $1h30m$ generando aproxi-

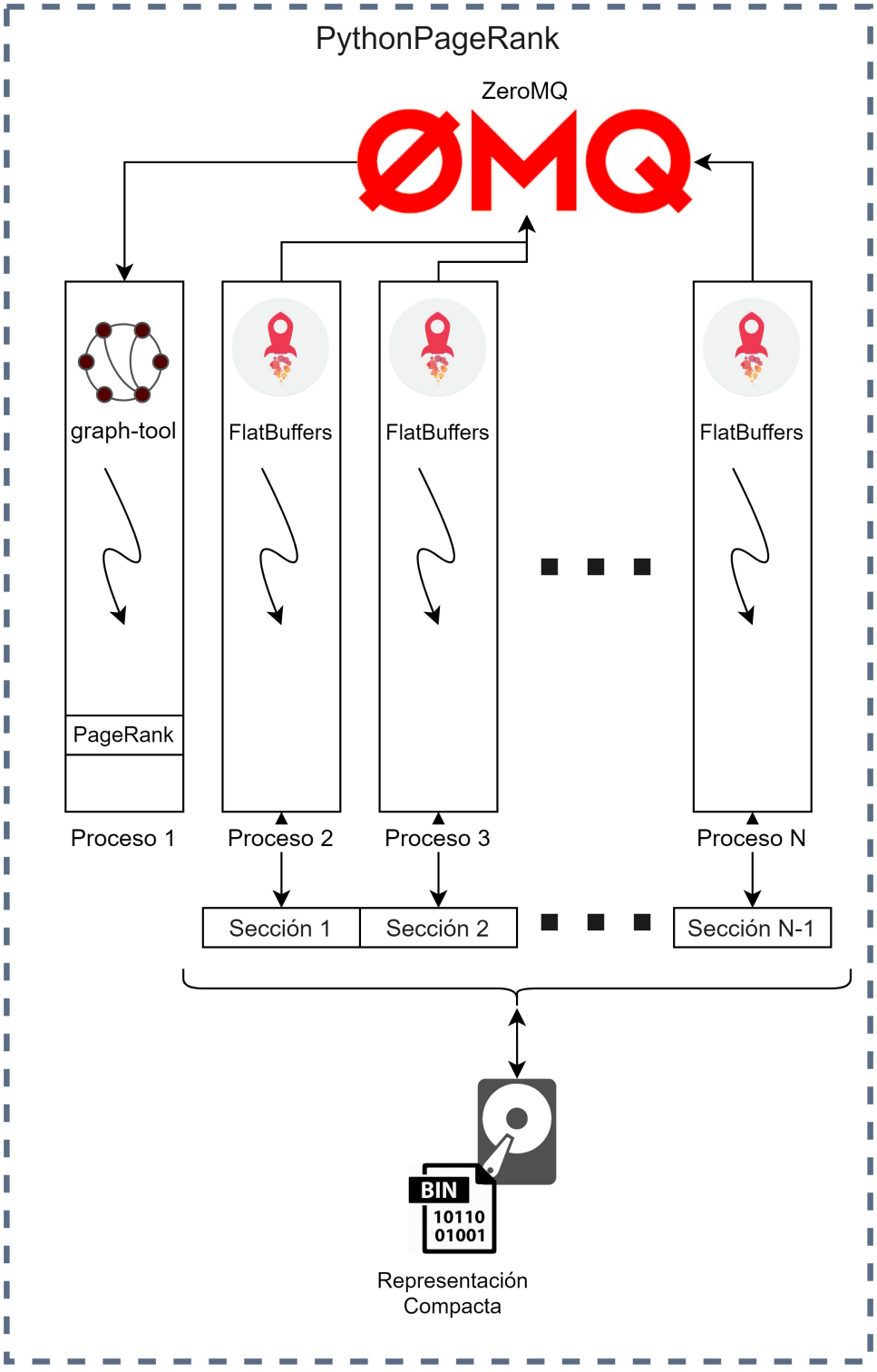


Figura 39: Estrategia de múltiples procesos y ZMQ.
Fuente: Elaboración Propia.

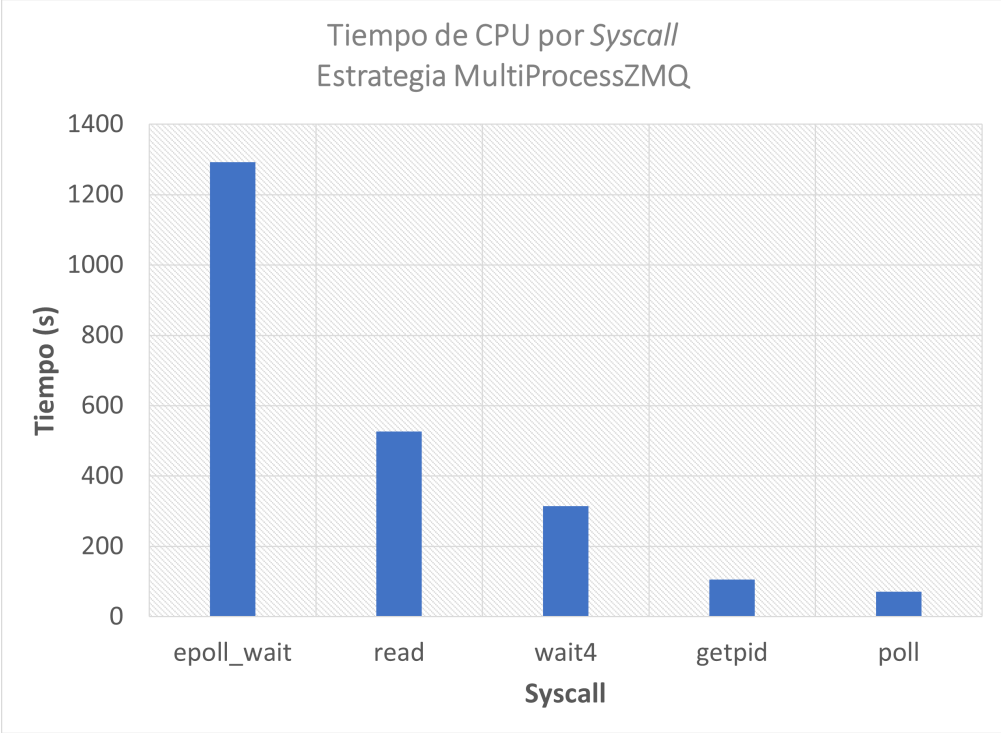


Figura 40: Tiempo de *CPU* por *syscall* para la estrategia *MultiProcessZMQ*
Fuente: Elaboración Propia.

madamente 91 millones de documentos en el índice de entidades y 8900 documentos en el índice de propiedades.

Resultados

En base a las modificaciones anteriores, hemos logrado reducir el tiempo de ejecución de nuestro proceso desde la etapa `(1.a)` hasta la `(4.c)` para todos los componentes que hemos implementado a un tiempo aproximado de $6h30m$ en nuestro ambiente de producción. Esto representa una optimización del $94\,\%$ en el tiempo de arranque de la aplicación original. Con esto, hemos logrado cumplir el objetivo de nuestra iteración al reducir en más de un $50\,\%$ el tiempo de ejecución de la aplicación, además, estamos a un paso de lograr todos nuestros objetivos específicos y objetivo general de nuestro trabajo, puesto que, solo nos queda revisar el proceso de actualización en tiempo real desde *Wikidata*. Finalmente, el diseño de arquitectura como resultado de esta iteración se puede apreciar en la figura 42.

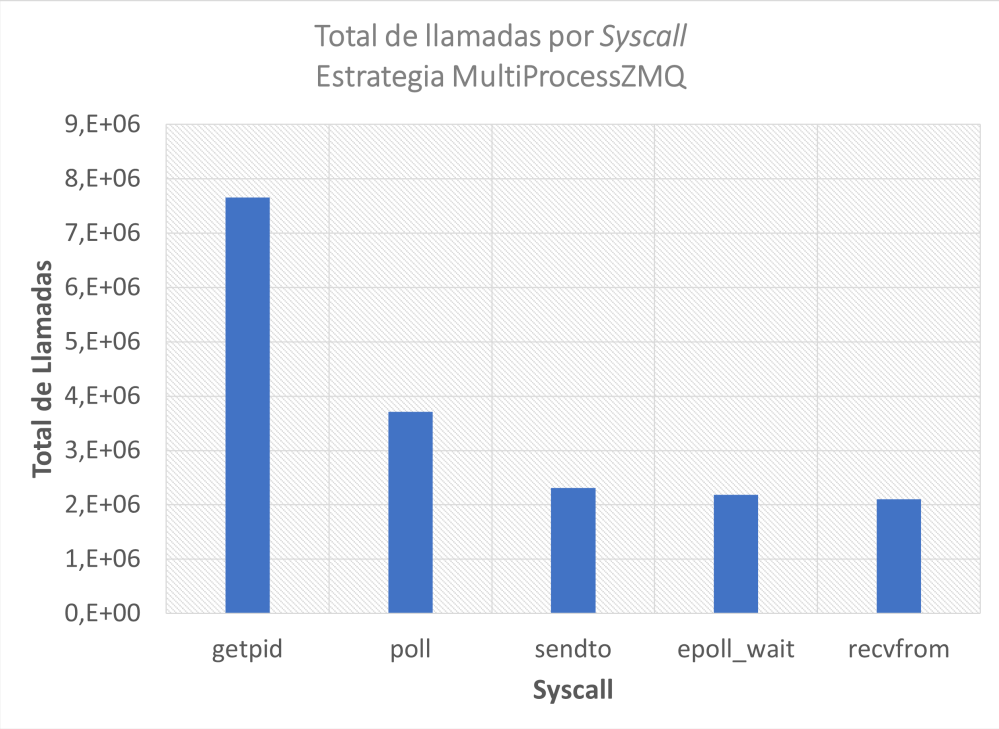


Figura 41: Total de llamadas por *syscall* para la estrategia *MultiProcessZMQ*
Fuente: Elaboración Propia.

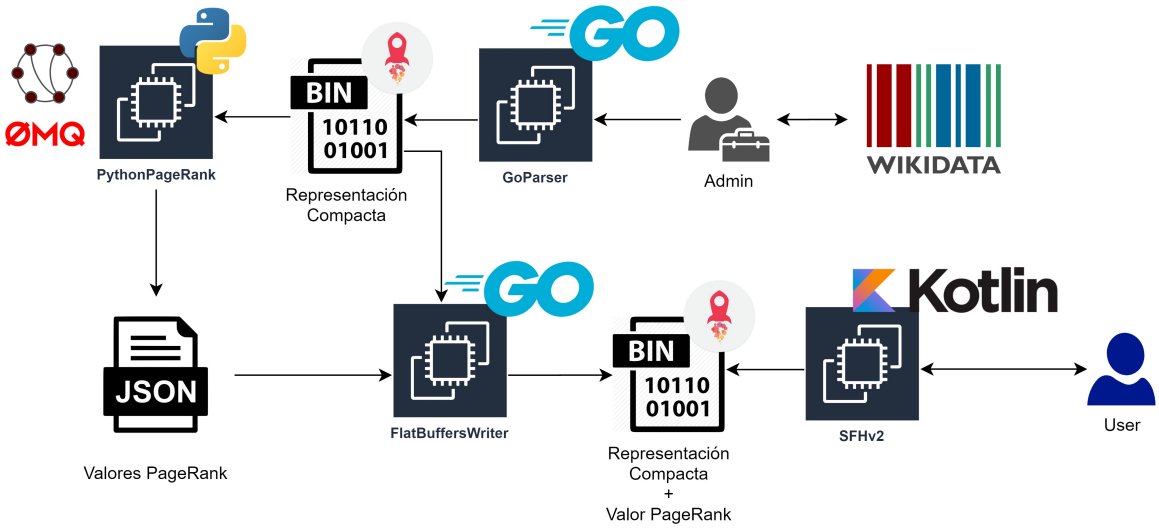


Figura 42: Diagrama de arquitectura para la solución.
Fuente: Elaboración Propia.

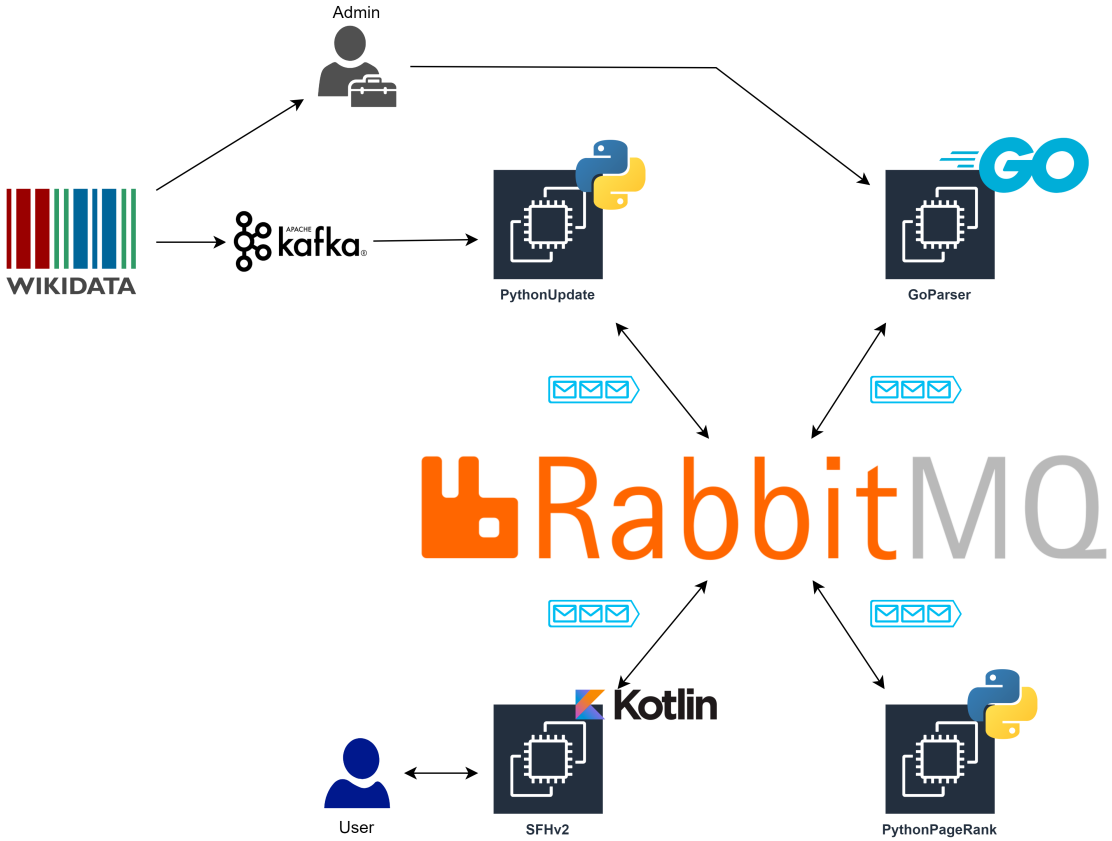


Figura 43: Diagrama de arquitectura orientado a microservicios para la solución.
Fuente: Elaboración Propia.

3.6. Cuarta iteración: Microservicios, paralelización y actualizaciones en tiempo real

Planificación

Hemos logrado reducir en un $94\,\%$ el tiempo de nuestra implementación en comparación a la solución original. Sin embargo, creemos que podemos hacerlo mejor a través de determinadas mejoras en nuestro proceso.

Actualmente, estamos realizando todo el proceso inicial de forma secuencial, una etapa tras otra. Esto se debe a la forma en la que hemos decidido desacoplar los componentes de nuestra aplicación, sin embargo, podemos reestructurar estos componentes para acercar la aplicación a una arquitectura orientada a microservicios. En esta iteración, buscamos lograr el resultado presentado en la figura 43.

Este diagrama representa la siguiente lista de cambios:

1. Eliminación de los ficheros PageRank y representación compacta intermedia entre los componentes “FlatBuffersWriter” y “SFHv2”. Utilizaremos una conexión directa entre un nuevo *message broker*, *RabbitMQ* y los componentes de la aplicación. Con esto, buscamos reducir la cantidad de operaciones de lectura y escritura realizadas al disco duro durante la ejecución del proceso inicial.
2. El componente “PythonPageRank” va a recibir de forma constante nuevos elementos desde el componente “GoParser” junto con una solicitud para calcular el algoritmo PageRank y publicar sus resultados.
3. En este nuevo modelo, el componente “GoParser” debe inyectar de forma constante los elementos obtenidos al procesar el fichero *gzip* de $40GB$ hacia el *message broker*. Estos mensajes serán recibidos directamente por los componentes “PythonPageRank” y “SFHv2”, eliminando la necesidad de escribir y leer a través del disco los resultados obtenidos al ejecutar “GoParser”.
4. Generamos el nuevo componente “PythonUpdate” orientado a lograr nuestro objetivo de mantener actualizados los registros de nuestra aplicación a través del tiempo. Este componente se deberá encargar de escuchar los cambios que son realizados en los servidores de *Wikidata*, para interpretar la ultima versión generada a través de los procesos que hemos presentado anteriormente.

Evaluaremos el impacto en nuestro tiempo de ejecución de estas modificaciones, junto con los resultados del nuevo componente “PythonUpdate” y la funcionalidad de actualización continua.

Implementación

3.6.1. *RabbitMQ*

Uno de los cambios más importantes que podemos notar, es la utilización de *RabbitMQ* en vez de *ZeroMQ* como *message broker*. Este cambio se debe a las complicaciones encontradas a la hora de implementar un sistema de colas entre los distintos componentes con *ZeroMQ*, esto ocurre por las características distribuidas de *ZeroMQ* y a nuestra necesidad de utilizar múltiples lenguajes de programación para nuestros componentes.

En *ZeroMQ* no existe un servicio centralizado encargado de recibir, enrutar y transmitir los mensajes generados por los componentes productores, sino que cada productor y consumidor ejecuta una versión del motor de mensajes del *broker*. Cuando utilizamos un único

proceso o distintos programas desarrollados con el mismo lenguaje de programación la solución funciona sin problemas, pero a la hora de utilizar distintos lenguajes de programación nos podemos encontrar con uno o más de los siguientes problemas:

- Diferencias en las versiones de la biblioteca utilizada entre lenguajes.
- Existencia de *bugs* en la biblioteca de un lenguaje especifico.
- Implementación incompleta de los protocolos definidos en un lenguaje especifico.
- Documentación incompleta o difícil de utilizar.

Estos problemas existen debido a la falta de bibliotecas oficiales de parte del proyecto *ZeroMQ* y a la falta de consenso en la comunidad de desarrolladores sobre cual biblioteca utilizar en cada lenguaje de programación soportado.

Por otro lado, *RabbitMQ* no sufre de estos problemas, puesto que existe solo una distribución del *broker* y solo una biblioteca oficial soportada por toda la comunidad del proyecto. Esto facilita y acelera bastante el desarrollo de aplicaciones distribuidas, con el *trade-off* de la existencia de un componente central y que puede representar un *Single Point of Failure*. En nuestro caso, estamos dispuestos a aceptar el *trade-off* en favor de la simplicidad del desarrollo.

3.6.2. Arquitectura orientada a eventos

Las mejoras que buscamos implementar requieren de un cambio fundamental en el proceso inicial de la aplicación definido en la sección 3.3, anteriormente hemos dicho que la unidad fundamental del proceso es un fichero *gzip* de $40GB$, pero en el nuevo diseño, la unidad mínima corresponde a un grupo el cual representa a una entidad en *Wikidata*. Además, debido a la naturaleza distribuida y paralela de esta versión, ya no existe solo un componente en ejecución durante un momento determinado, sino que los cuatro componentes existentes se ejecutan de forma paralela en todo momento. Considerando esto, hemos generado los siguientes procesos para cada componente:

- GoParser
    1. Obtener un grupo desde fichero *gzip*.
    2. Filtra y validar líneas en formato *N-Triples*.
    3. Creación de propiedades inversas.
    4. Serialización en formato binario a través de *FlatBuffers*.
    5. Publicación de grupo en *broker* para componentes *PythonPageRank* y *SFHv2*.

6. Esperar por mensaje desde componente *PythonUpdate*.
7. Obtener grupo actualizado desde *Wikidata*.
8. Filtra y validar líneas en formato *N-Triples*.
9. Creación de propiedades inversas.
10. Serialización en formato binario a través de *FlatBuffers*.
11. Publicación de grupo actualizado en *broker* para componente *SFHv2*.

- PythonPageRank

1. Esperar por mensaje desde componente *GoParser*.
2. Almacenar información del grupo en grafo.
3. Esperar por mensaje desde componente *SFHv2* solicitando valores *PageRank*.
4. Realizar algoritmo *PageRank* utilizando biblioteca *graph-tool*.
5. Serialización de valores *PageRank* a través de *FlatBuffers*.
6. Publicación de valores *PageRank* en *broker* para componente *SFHv2*.

- PythonUpdate

1. Esperar por mensaje desde componente *SFHv2* solicitando actualizaciones.
2. Esperar por una actualización desde *Wikidata* a través de *Kafka*.
3. Filtro de actualización según tipo.
4. Publicación de actualización en *broker* para componente *GoParser*.

- SFHv2

1. Esperar por grupo desde componente *GoParser*.
2. Creación de documento en índice de entidades.
3. Creación de documento en índice de propiedades.
4. Esperar por mensaje desde componente *GoParser* indicando que se han procesado todos los grupos.
5. Enviar solicitud de valores *PageRank* a componente *PythonPageRank*.
6. Calcular frecuencias en índice de propiedades.
7. Esperar valores *PageRank* desde componente *PythonPageRank*.
8. Asignación de valores *PageRank* en documentos del índice para entidades.
9. Enviar solicitud de actualizaciones a componente *PythonUpdate*.
10. Esperar por grupo desde componente *GoParser*.
11. Actualización de documento en índice de entidades.

La representación gráfica de las interacciones entre estos procesos se puede apreciar en el diagrama de flujo de la figura 44.

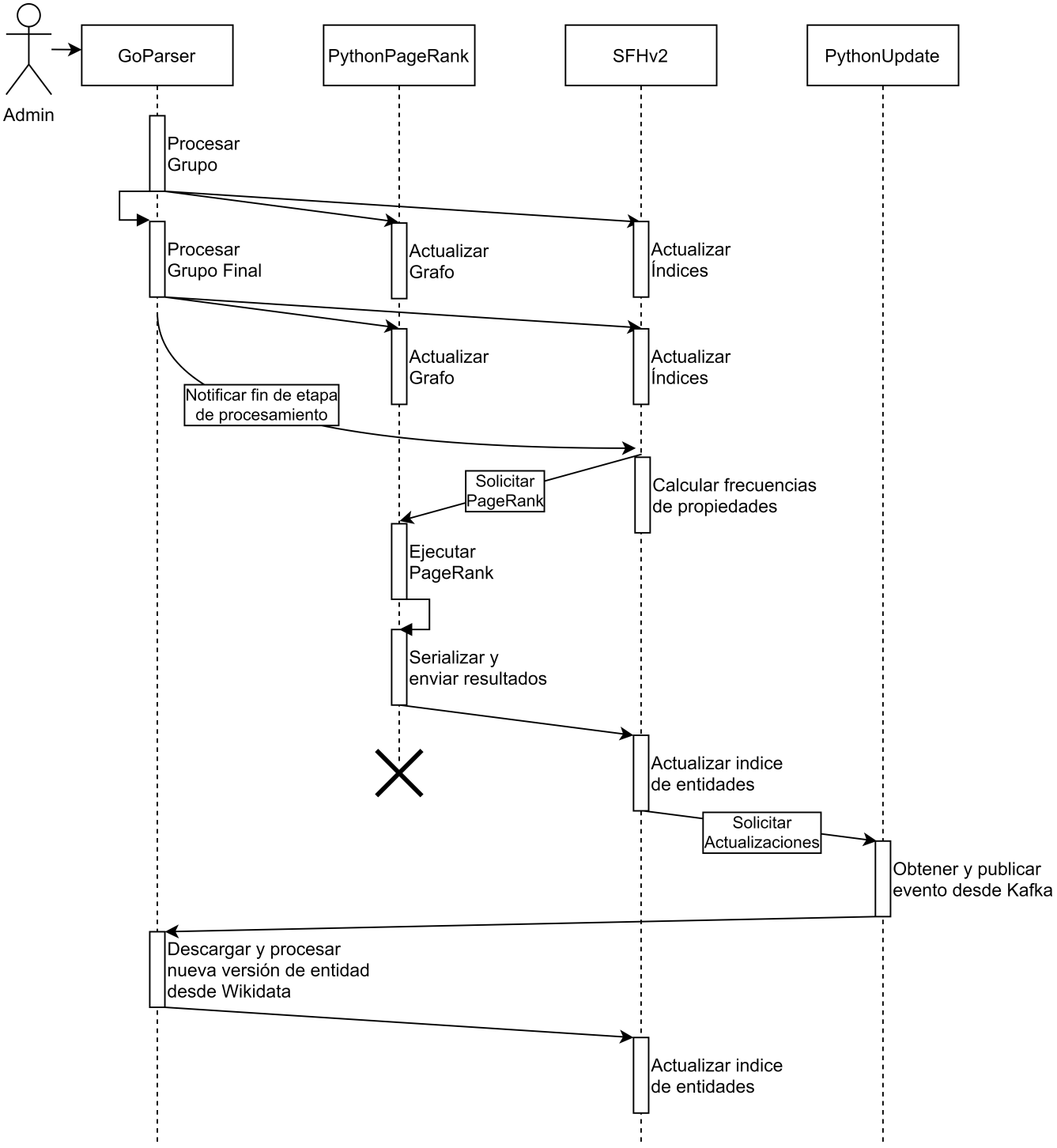


Figura 44: Interacción entre componentes a través de la ejecución de la aplicación
Fuente: Elaboración Propia.

3.6.3. Actualización en tiempo real

El procedimiento para obtener versiones actualizadas de los documentos procesados en la etapa inicial consiste en suscribirse a una cola de eventos llamada *EventStreams* publicada por los servidores de *Wikimedia*. Este servicio ha sido construido utilizando el mundialmente conocido bus de eventos *Apache Kafka* [Garg, 2013], el cual es capaz de servir millones de eventos a múltiples clientes sin problemas. La cola *EventStreams* nos provee múltiples temas a los cuales nos podemos suscribir y recibir información, para nuestro caso de uso, nos interesan los temas `recentchange` y `revision-create`, los cuales nos permiten identificar modificaciones a documentos en cualquiera de los servicios de *Wikimedia*.

Según las actuales estadísticas de *Wikimedia*, se realizan un total de entre 45 a 60 millones de ediciones a través de todos los sitios del proyecto [Wikipedia, 2022], por lo tanto es necesario algún tipo de filtrado para lograr procesar solo la información que necesitamos. Hemos definido las siguientes reglas de filtrado:

- El servidor donde se ha realizado el cambio corresponde a `www.wikidata.org`
- El cambio se encuentra clasificado como tipo `edit`.
- El cambio no corresponde a una modificación clasificada como `minor`.
- El cambio no ha sido realizado por un *bot*.
- Se ha asignado un identificador de revisión a la nueva versión generada por el cambio obtenido.
- El titulo del documento modificado cumple la expresión regular `^Q\d+$`. Ej. `Q1`.

Un evento que cumple las reglas anteriores se puede apreciar en la figura 45.

Con cada uno de estos eventos que recibimos podemos construir la siguiente *URL*: “https://www.wikidata.org/wiki/Special:EntityData/<qid>.nt?flavor=simple&revision=<rev>”, en la cual, debemos reemplazar los valores *qid* con el identificador obtenido en el titulo del evento y *rev* por el identificador de la nueva versión generada.

Al acceder a este enlace, podemos obtener la representación en formato *N-Triples* de la entidad que ha sido modificada en *Wikidata*, luego, podemos enviar estos datos directamente al componente *GoParser* de nuestra solución, el cual, transformará los datos crudos en un grupo serializado con *FlatBuffers*, el siguiente paso es publicar la versión actualizada en el *broker* para finalmente ser consumido por el componente *SFHv2*.

Una vez que *SFHv2* tiene el nuevo grupo, el proceso de actualización de un documento en el índice de entidades es bastante sencillo. Lo primero, es obtener la versión anterior en el

```
{
    "$schema": "/mediawiki/recentchange/1.0.0",
    "meta": {
        "uri": "https://www.wikidata.org/wiki/Q9381816",
        "request_id": "77087a5b-c56b-4bf2-a175-76604707fd20",
        "id": "7aa99b9f-a32f-46d6-939a-c64ec6b3cf0f",
        "dt": "2022-01-02T20:50:23Z",
        "domain": "www.wikidata.org",
        "stream": "mediawiki.recentchange",
        "topic": "eqiad.mediawiki.recentchange",
        "partition": 0,
        "offset": 3542816478
    },
    "id": 1604000617,
    "type": "edit",
    "namespace": 0,
    "title": "Q9381816",
    "comment": "[[Property:P166]]: [[Q946960]], #drag-n-drop",
    "timestamp": 1641156623,
    "user": "Niegodzisie",
    "bot": false,
    "minor": false,
    "patrolled": true,
    "length": {
        "old": 34038,
        "new": 34450
    },
    "revision": {
        "old": 1555000076,
        "new": 1555000233
    },
    "server_url": "https://www.wikidata.org",
    "server_name": "www.wikidata.org",
    "server_script_path": "/w",
    "wiki": "wikidatawiki",
    "parsedcomment": "[[Property:P166]]: [[Q946960]], #drag-n-drop"
}
```


Figura 45: Evento obtenido desde *EventStreams* en formato *JSON*
Fuente: Elaboración propia.

Figura 46: Reducción inesperada del rendimiento en *broker*
Fuente: Elaboración Propia.

índice para el grupo que hemos recibido, lo siguiente es obtener el valor *PageRank* y almacenarlo temporalmente, luego, creamos el nuevo documento en el índice en base a los datos obtenidos desde el *broker*, por ultimo, asignamos el valor *PageRank* al nuevo documento y eliminamos su versión anterior.

Al realizar este proceso de forma continua, podemos actualizar los registros en nuestro índice de entidades en solo instantes de que los cambios han sido publicados y propagados por *Kafka* y la cola *EventStreams* desde *Wikidata*.

Resultados

3.6.4. Rendimiento de instancias *AWS*

Durante los primeros minutos de la ejecución de la nueva solución, logramos procesar un total de $30{,}000$ eventos por segundo, sin embargo, luego de un tiempo el rendimiento decae inesperadamente a $15{,}000$ eventos por segundo según lo visto en la figura 46. Al investigar en la instancia virtual, podemos notar que el rendimiento de la *CPU* también se ha reducido, desde un $70\,\%$ hasta un $40\,\%$ fijo, como se puede observar en la figura 47.

Al investigar en la documentación oficial de *AWS* [34] [35], sobre las métricas de créditos de

[34] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-performance-instances.html
[35] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-credits-baseline-concepts.html

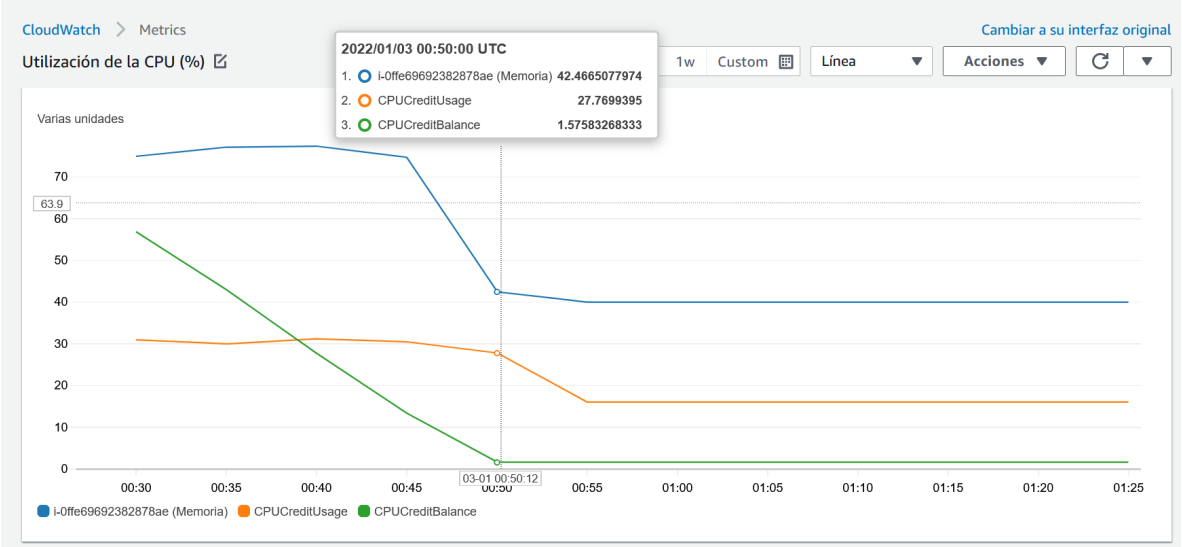


Figura 47: Utilización de *CPU* y créditos de *CPU* en *AWS* para la instancia `t3.2xlarge`
Fuente: Elaboración Propia.

CPU disponible, hemos notado que el tipo de instancia seleccionado para nuestro ambiente productivo es parte del grupo de instancias *burstable*, las cuales son excelentes para cargas de trabajo que no utilizan de forma agresiva la *CPU* o que necesitan *peaks* de rendimiento durante pequeños periodos de tiempo. Nuestra aplicación necesita consumir toda la *CPU* posible durante un largo periodo de tiempo, por lo que la instancia `t3.2xlarge` no es la más indicada para nuestra solución.

La documentación de *AWS* [36] [37] [38] recomienda utilizar instancias de tipo `M5` para aquellas cargas de trabajo balanceadas entre utilización de memoria y *CPU*. En nuestro caso, utilizaremos una instancia `m5.2xlarge` la cual entrega la misma cantidad de núcleos (8) y la misma memoria RAM $(32GB)$ que nuestra instancia anterior.

Una vez iniciado el proceso con la nueva instancia, no notamos la existencia del anterior cuello de botella, sin embargo, transcurrido suficiente tiempo y durante la etapa de ejecución del algoritmo *PageRank* encontramos un error *OutOfMemory*, por lo que hemos decidido aumentar el tamaño de nuestra infraestructura por una instancia `m5.4xlarge` la cual entrega 16 núcleos y $64GB$ de memoria.

Con esta ultima modificación podemos observar en la figura 48 que nuestro rendimiento se mantiene constante durante el tiempo y sin perdidas inesperadas, pero nuestra *CPU* solo está siendo utilizada en un $50\,\%$ según la figura 49 debido al aumento en el tamaño de la instancia, por lo que es posible encontrar una nueva combinación de recursos que permita obtener resultados de forma más eficiente.

[36] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/general-purpose-instances.html
[37] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/compute-optimized-instances.html
[38] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/memory-optimized-instances.html

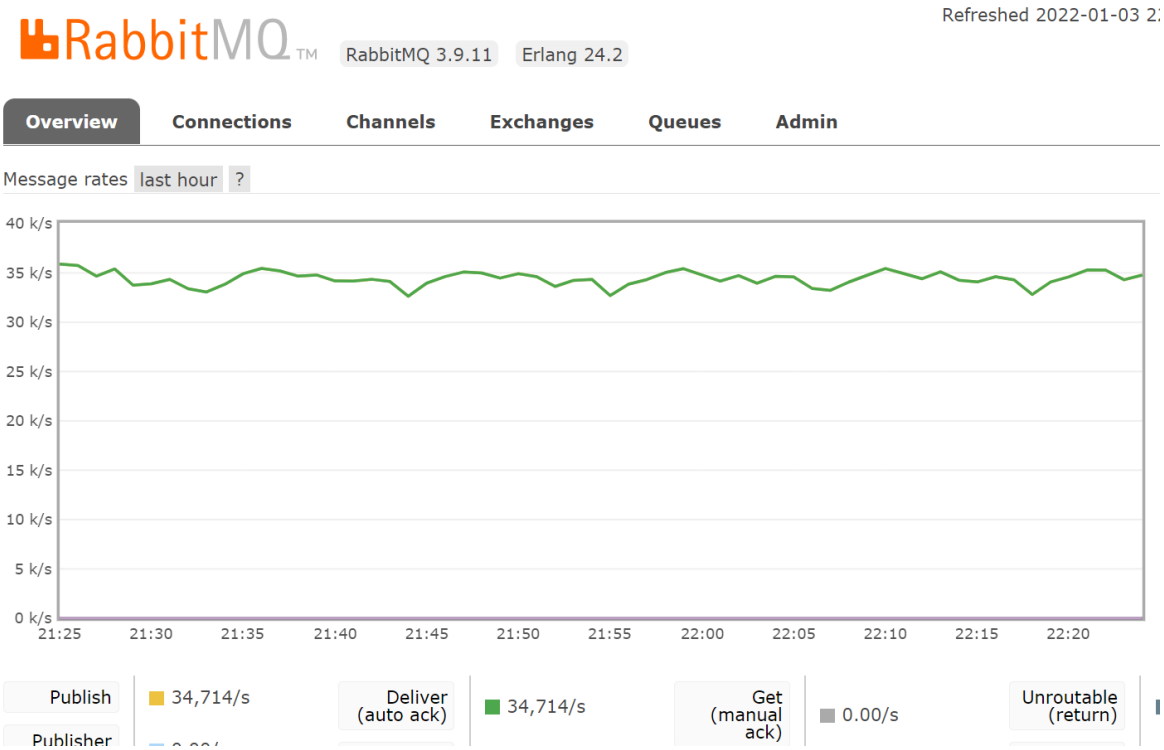


Figura 48: Rendimiento en *broker* utilizando una instancia *m5.4xlarge*
Fuente: Elaboración Propia.

Figura 49: Rendimiento de la instancia *m5.4xlarge*
Fuente: Elaboración Propia.

Proceso	**Tiempo de ejecución**
Procesamiento de fichero *dump* en formato *gzip*	1h30m
Construcción de grafo para entidades	15m
Ejecución de algoritmo *PageRank*	4m
Actualización del índice de entidades	39m
Total	2h28m

Tabla 5: Tiempos obtenidos en ambiente productivo.
Fuente: Elaboración propia.

3.6.5. Tiempos obtenidos

Con las últimas modificaciones hemos logrado reducir aún más el tiempo necesario para procesar el fichero *dump* de *Wikidata*, junto con implementar un mecanismo simple para mantener actualizados estos registros procesados a través del tiempo y en pocos instantes de ser modificados en nuestra fuente de datos.

Los siguientes tiempos de la tabla 5, separados por proceso, fueron obtenidos al promediar tres ejecuciones de la aplicación en nuestro ambiente productivo desde un estado inicial limpio. El tiempo total de la nueva versión es de $2h28m$, un $37\,\%$ menos que nuestra versión anterior y un $97{,}7\,\%$ más rápido que la versión original en la que se basa este trabajo.

3.6.6. Actualizaciones

Una vez iniciada la aplicación, pasamos a procesar notificaciones de cambios desde *Kafka*. Al ejecutar esta etapa durante una hora, hemos recopilado la siguiente información, resumida en la tabla 6 y categorizada de la siguiente forma:

- Actualización exitosa. La notificación de actualización ha sido procesada desde *Kafka* y se ha actualizado el registro de la entidad en nuestro índice de entidades.
- Actualización fallida. La notificación de actualización ha sido procesada desde *Kafka*, pero no ha sido posible actualizar el registro de la entidad en nuestro índice, ya sea debido a un problema durante la ejecución de la aplicación o porque simplemente no existía la entidad en nuestro índice.

Finalmente, podemos comentar que nuestro proceso de actualización de entidades tiene un nivel de éxito del $84{,}5\,\%$ con un rendimiento promedio de $4{,}8$ actualizaciones procesadas por segundo.

Resultado	**Cantidad de actualizaciones**
Exitosas	14710
Fallidas	2694
Total	17404

Tabla 6: Cantidad de notificaciones procesadas desde *Kafka*
Fuente: Elaboración propia.

CAPÍTULO 4
VALIDACIÓN DE LA SOLUCIÓN

4.1. Diseño e implementación final de la solución

Los procesos y arquitectura de la solución se pueden observar en las figuras 44 y 43 respectivamente. Los componentes declarados fueron implementados utilizando los siguientes lenguajes de programación y bibliotecas:

- *GoParser*. Encargado de procesar datos originales obtenidos desde *Wikidata*.
  - Lenguaje de programación: *GoLang*
  - Dependencias
    - *go-resty/resty*. Cliente *HTTP/HTTPS*.
    - *google/flatbuffers*. Codificación binaria.
    - *klauspost/pgzip*. Descompresión *gzip*.
    - *rabbitmq/amqp091-go*. Comunicación con *broker* de mensajes.
    - *sirupsen/logrus*. Utilidades para registros.
    - *stretchr/testify*. Utilidades para pruebas unitarias.
- PythonPageRank. Encargado de construir, almacenar el grafo de entidades, junto con ejecutar el algoritmo *PageRank*.
  - Lenguaje de programación: *Python*
  - Dependencias
    - *flatbuffers*.
    - *pika*. Comunicación con *broker* de mensajes.
    - *loguru*. Utilidades para registros.
    - *graph-tool*. Operaciones de grafos.
- PythonUpdate. Encargado de procesar las notificaciones de cambio provenientes de los servidores de *Wikidata* y notificar a otros componentes para actualizar el índice de entidades.
  - Lenguaje de programación: *Python*
  - Dependencias
    - *pika*.
    - *loguru*
    - *pywikibot*. Comunicación con servidores *Kafka* de *Wikidata*.

- SFHv2
  - Lenguaje de programación: *Kotlin*
  - Dependencias
    - *flatbuffers*.
    - *amqp-client*. Comunicación con *broker* de mensajes.
    - *log4j*. Utilidades para registros.
    - *kotlin-logging-jvm*. Utilidades para registros.
    - *kotlinx-coroutines-core*. Generación de tareas paralelas.
    - *kotlinx-serialization-json*. Serialización de clases a *JSON*.
    - *lucente-core*. Generación de índices.
    - *lucente-queries*. Consultas a índices.
    - *lucente-queryparser*. Generación de consultas en base a texto.
    - *clikt*. Utilidades para el procesamiento de la linea de comandos en la aplicación.
    - *junit*. Utilidades para pruebas unitarias.

Todas las fuentes del proyecto, junto con su historial de modificaciones, se encuentran disponible en el repositorio *git* hospedado en la plataforma pública *GitHub* y se puede acceder a ellas a través del siguiente enlace https://github.com/capgadsx/SPARQLforHumans.

4.2. Despliegue de la solución

Para facilitar y agilizar el despliegue de la aplicación hemos decidido utilizar las herramientas *docker* y *docker-compose* para la compilación, distribución y ejecución de nuestra solución. El hecho de utilizar estas tecnologías nos asegura la facilidad y reproducibilidad de la aplicación en cualquier ambiente que soporte ejecutar contenedores *docker*. Dicho esto, el proceso de despliegue de nuestra solución se puede resumir en cuatro simples pasos:

1. Obtener un fichero *dump* inicial con los datos a trabajar desde *Wikidata*.
2. Clonar el repositorio de fuentes desde *GitHub*.
3. Ejecutar `docker-compose build` para compilar los componentes y construir los contenedores.
4. Ejecutar `docker-compose up` para iniciar la aplicación

Una vez realizados estos pasos, *docker-compose* iniciará los componentes en su orden correspondiente y orquestará la comunicación entre ellos. Es importante notar, que debido a

las modificaciones realizadas durante el desarrollo de este trabajo, los requerimientos mínimos del servidor o instancia donde será ejecutada la solución son los siguientes:[39]

- $37GB$ de *RAM* para el procesamiento de datos y almacenar el grafo de entidades.
- $200GB$ de espacio en disco para almacenar el fichero *dump* inicial, contenedores creados, los índices generados y datos del algoritmo *PageRank*.

Para el desarrollo de este trabajo, hemos utilizado las siguientes instancias del servicio *AWS EC2*:

- m5.4xlarge. 16 núcleos, $64GB$ de memoria.
- m5.8xlarge. 32 núcleos, $128GB$ de memoria.
- c5.9xlarge. 36 núcleos, $72GB$ de memoria.
- c5.12xlarge. 48 núcleos, $96GB$ de memoria.
- r5.2xlarge. 8 núcleos, $64GB$ de memoria.
- r5.4xlarge. 16 núcleos, $128GB$ de memoria.

4.3. Pruebas unitarias

Para validar que nuestra solución logra el objetivo de replicar las funcionalidades del proyecto original hemos decidido migrar una serie de pruebas automatizadas desde el proyecto base a nuestras solución.

De un total de 218 pruebas existentes 34 fueron implementadas, las cuales, se enfocan en los aspectos de procesamiento de texto junto con generación y consulta de entidades y propiedades en índices *Lucene*. El resto de pruebas no fueron implementadas por alguna de las siguientes razones:

- Funcionalidad relacionada a la compresión de texto a formato *gzip*. No hemos migrado estas pruebas debido a que nuestra solución solo realiza descompresión de datos.
- Funcionalidad relacionada al proceso de ordenamiento de grupos. Nuestra solución elimina la necesidad de ordenar los objetos una vez son procesados, por lo que no necesitamos estas pruebas.

[39] Se asume que se utilizará el fichero *dump* de *Wikidata* completo, si se reduce el tamaño de la entrada de la aplicación los requerimientos de almacenamiento y memoria también se reducen de forma considerable.

- Funcionalidad relacionada a la ejecución del algoritmo *PageRank*. Ha diferencia de la solución original, nosotros utilizamos una biblioteca externa para realizar esta tarea, por lo que no necesitamos probar su funcionamiento.
- Funcionalidad relacionada a la interacción entre el *backend* y *frontend*. Nuestro trabajo se enfoca en las mejoras al *backend*, por lo que todas las funcionalidades del *frontend* han sido ignoradas. No necesitamos migrar estas pruebas.
- Funcionalidad relacionada a la utilización de estructuras de datos internas. En nuestra solución utilizamos solamente estructuras de datos definidas en los lenguajes de programación de los distintos componentes, en comparación, la solución base genera estructuras de datos que para nosotros no son necesarias, puesto que, son implementadas de forma nativa por el lenguaje utilizado.

Con esto y considerando que las 34 pruebas fueron implementadas, ejecutadas y sus resultados son exitosos, podemos validar que cumplimos con la misma funcionalidad que nuestro proyecto base.

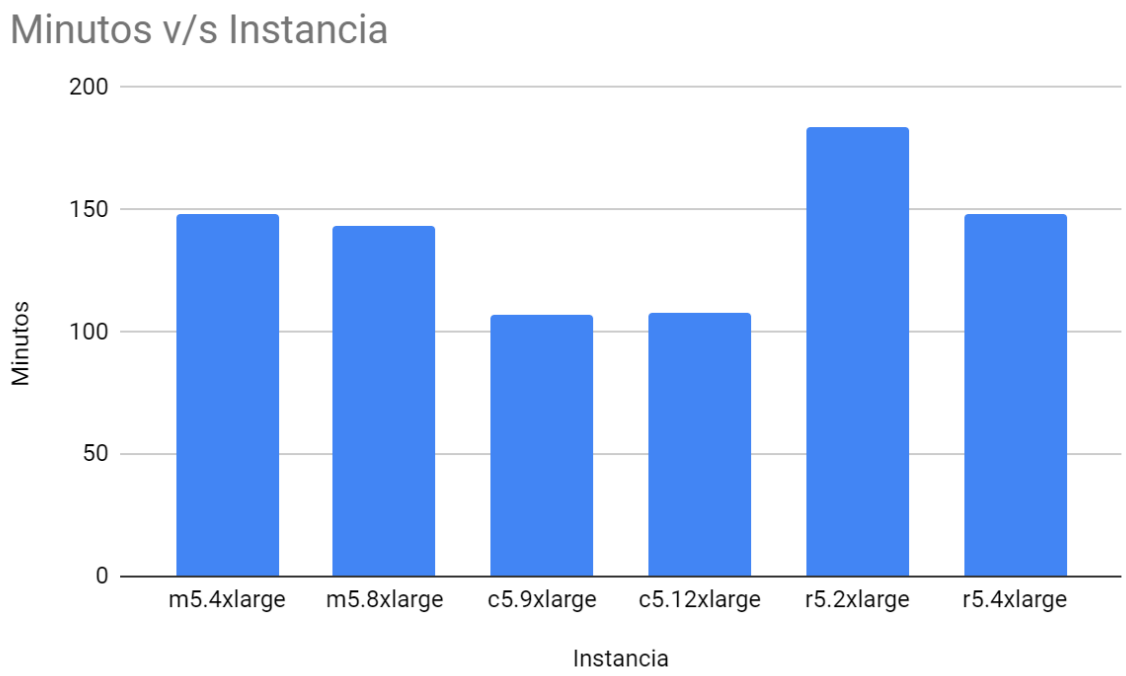


Figura 50: Tiempo de ejecución en distintas instancias
Fuente: Elaboración Propia.

Proceso	Tiempo necesario
Pre-procesamiento	35h
PageRank	11h
Generación del índice de entidades	58h
Generación del índice de propiedades	5h
Total	109h

Tabla 7: Tiempos necesarios para procesar la información de *Wikidata*
Fuente: [De la Parra, 2020].

Instancia	Tiempo total de ejecución
`m5.4xlarge`	2h28m
`m5.8xlarge`	2h23m
`c5.9xlarge`	1h47m
`c5.12xlarge`	1h48m
`r5.2xlarge`	3h4m
`r5.4xlarge`	2h26m

Tabla 8: Tiempos obtenidos en múltiples tipos y tamaños de instancias
Fuente: Elaboración Propia.

4.4. Comparación

Una vez validadas las funcionalidades necesarias, podemos comparar nuestros resultados con la versión base, para esto, consideramos los siguientes datos iniciales, presentados en la tabla 7.

Utilizaremos la métrica del tiempo total de ejecución para realizar nuestra comparación. En la tabla 8 se puede observar el tiempo total de ejecución de nuestra solución[40] en múltiples tipos y tamaños de instancias, además, en el gráfico 50 se puede observar como se distribuyen estos tiempos entre las instancias.

Con estos resultados, logramos cumplir con nuestro objetivo general, reducir el tiempo necesario para obtener resultados desde la aplicación. Además, hemos logrado mejorar la aplicación original con dos nuevas características: la actualización de los registros en los índices de forma continua y en tiempo real junto con demostrar la escalabilidad vertical de la nueva solución al implementar una arquitectura orientada a microservicios.

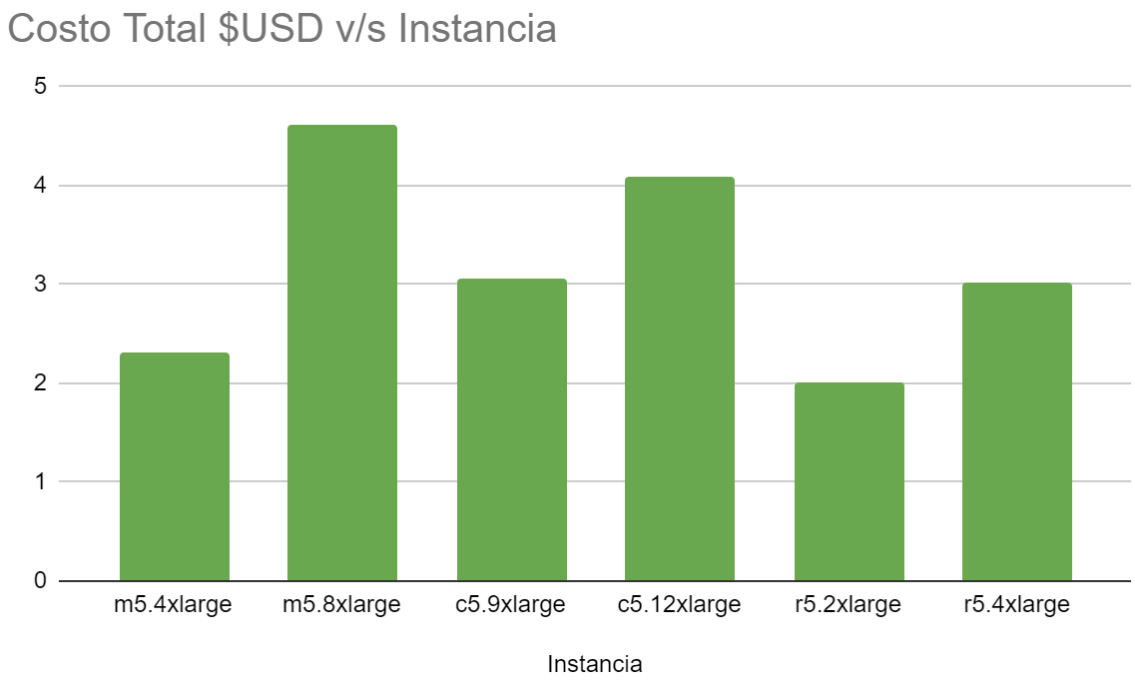


Figura 51: Costo total de ejecución en distintas instancias
Fuente: Elaboración Propia.

4.5. Costos de la solución

Una dimensión que no debemos dejar a un lado en nuestra validación es el costo de utilizar los servicios de la nube pública *Amazon Web Services (AWS)*. Debemos considerar dos servicios para realizar un análisis de los costos totales de nuestra solución: almacenamiento *(Amazon Elastic Block Store)* y computo *(Amazon Elastic Cloud Compute)*.

En nuestro caso, el costo de todas las pruebas realizadas en las distintas instancias corresponde a un costo fijo por almacenamiento más un costo variable según la instancia *EC2* utilizada.

Considerando las siguientes características de nuestro almacenamiento en *EBS*, el costo fijo es de *USD* $\$60$.

- Cantidad de almacenamiento. $200GB$.
- Tipo de disco. *General Purpose SSD (gp3)*.
- Operaciones de entrada/salida máximas *IOPS*. 12000.
- Velocidad máxima de transferencia. $256MBps$.

[40] Sin considerar la funcionalidad de actualización de datos en tiempo real.

Instancia	Costo por hora en *$USD*
`m5.4xlarge`	0,768
`m5.8xlarge`	1,536
`c5.9xlarge`	1,53
`c5.12xlarge`	2,04
`r5.2xlarge`	0,504
`r5.4xlarge`	1,008

Tabla 9: Costo por hora según tipo de instancia en *AWS*
Fuente: https://aws.amazon.com/es/ec2/pricing/on-demand/.

Para obtener el valor del costo variable, utilizamos el costo por hora de cada tipo de instancia utilizada multiplicado por el tiempo total de ejecución de la aplicación en intervalos de una hora. El costo de cada instancia se puede observar en la tabla 9.

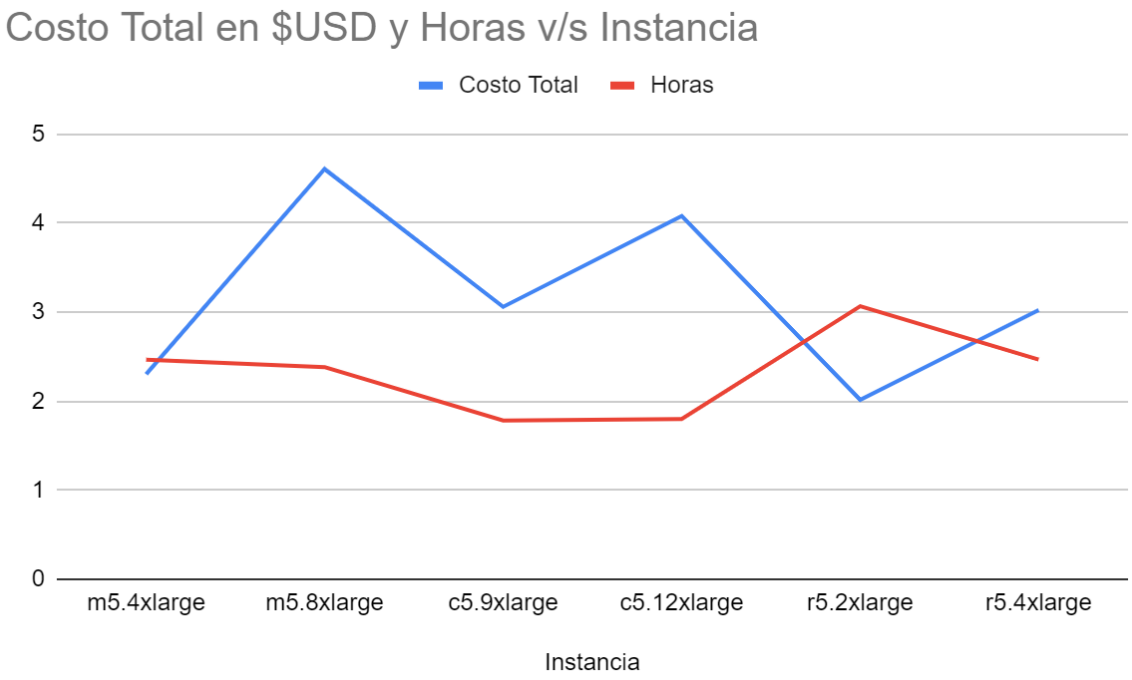


Figura 52: Costo total y tiempo de ejecución en distintas instancias
Fuente: Elaboración Propia.

Con estos datos, podemos realizar el gráfico de la figura 51. Además, si cruzamos esta información con el gráfico de la figura 50, podemos encontrar la combinación más eficiente en tiempo y costo, la cual corresponde al tipo de instancia `c5.9xlarge`, como se puede observar en la figura 52.

CAPÍTULO 5
CONCLUSIONES

En este trabajo, presentamos el proceso de optimización al rendimiento de una aplicación monolítica, comenzando por un análisis de sus funcionalidades, diseño e implementación, continuando con un rediseño y reimplementación de sus procesos, para finalizar con un conjunto de mejoras y pruebas en múltiples ambientes de ejecución en una nube pública, todo esto, realizando un conjunto de iteraciones enfocadas a la mejora continua de nuestros resultados utilizando la metodología ágil *PDCA*.

Como consecuencia de nuestras modificaciones, hemos logrado reducir en un $97\,\%$ el tiempo necesario para obtener resultados de la aplicación en comparación a la solución original. Además, nuestra solución logra agregar valor a la aplicación a través de la implementación de un proceso de actualización continua de sus datos en tiempo real.

También podemos comentar que nuestro problema de optimización se encuentra en la clásica situación del compromiso espacio-temporal o más conocido como *space-time complexity trade-off* en inglés. En nuestro caso, buscamos aumentar los recursos espaciales con el objetivo de reducir el tiempo de ejecución de nuestros programas. Consideramos que esta decisión le entrega un gran valor al usuario final, puesto que ahora tiene el poder de elegir si desea utilizar más recursos o esperar un tiempo mayor para obtener resultados, opción que no existe en la solución original.

Además, en base al análisis de costo y tiempo realizado en la sección anterior, podemos afirmar que la combinación de recursos más eficientes para procesar nuestros datos de entrada corresponden a los existentes en una instancia tipo `c5.9xlarge` del servicio *Amazon Elastic Compute Cloud* disponible en la nube pública de *Amazon Web Services*.

Dicho esto, podemos concluir que no solo hemos logrado nuestros objetivos específicos y general, sino que además, hemos logrado generar valor en nuestra solución a través de dos nuevas características: actualizaciones en tiempo real y escalabilidad vertical.

5.1. Discusión

A continuación, presentamos una serie de limitaciones de nuestra solución, junto con comentarios sobre la implementación realizada.

5.1.1. Optimizaciones

La fuente de la mayor parte de nuestras optimizaciones corresponde a eliminar componentes, flujos o funciones que no vamos a utilizar. En gran parte, esto se debe a los ajustes realizados para que nuestra solución solo fuera compatible con las estructuras, formatos y entidades existentes en la base de datos *RDF* de *Wikidata*, a diferencia de la aplicación original, la cual, permite integrarse con múltiples repositorios.

Otra de las optimizaciones que ha permitido reducir más tiempo es la utilización de un mecanismo externo para obtener los valores del algoritmo *PageRank* aplicado al grafo dirigido de entidades, esto se debe a que la implementación original utilizaba un método secuencial para ejecutar el algoritmo y que además no era posible paralelizar, a diferencia de nuestra solución, la cual hace uso del proyecto *graph-tool*.

5.1.2. *Time-space complexity trade-off*

Un aspecto criticable del trabajo realizado es el aumento importante en la cantidad de memoria *RAM* necesaria para ejecutar la aplicación. En el trabajo original [De la Parra, 2020], se declara que es posible obtener resultados con $16GB$ de memoria y en nuestro caso necesitamos al menos $37BG$ de *RAM*, lo cual, representa un aumento del $130\,\%$. Ahora bien, si contrastamos este aumento con la reducción del tiempo original, el cual corresponde a un $97\,\%$, creemos que vale la pena el aumento en los recursos necesarios.

Además, con nuestra implementación, hemos obtenido el poder de elegir cuánto tiempo, aproximado, nos vamos a tomar en procesar nuestros datos. Esta posibilidad no existe en la solución original, debido a su naturaleza secuencial y a que no fue diseñada con el foco de ser verticalmente escalable.

5.1.3. *Lucene*

Una sección del tiempo de ejecución de nuestra solución final corresponde a la interacción con nuestros índices de entidades y propiedades, operaciones como por ejemplo creación, actualización, consultas y sincronización en disco comienzan a tomar protagonismo cuando trabajamos con más de 90 millones de documentos.

En base a esto, uno se puede cuestionar la idea de utilizar *Lucene* de forma directa. Una alternativa a esta estrategia es apoyarse de un sistema basado en *Lucene* y que tenga la capacidad de operar de forma asíncrona y escalable, como por ejemplo, *Elasticsearch* [Kononenko et al., 2014], *OpenSearch* [41] o *Apache Solr* [Smiley et al., 2015].

[41] https://aws.amazon.com/es/opensearch-service/the-elk-stack/what-is-opensearch/

5.2. Trabajo futuro

Aún es posible continuar mejorando nuestra solución implementada, a continuación proponemos una serie de temas que creemos pueden aportar a obtener aún mejores resultados, categorizados en los siguientes ejes: nuevas tecnologías, mejoras al rendimiento de los componentes, nuevas funcionalidades, despliegue de la aplicación e interacción con el usuario.

5.2.1. Nuevas tecnologías

Una de las nuevas herramientas existentes en el ecosistema de aplicaciones basadas en la *Java Virtual Machine* es *GraalVM* [42], una nueva implementación de la *JVM* creada por *Oracle*. El proyecto declara que se obtienen mejoras al rendimiento de las aplicaciones en un $10\,\%$ sin cambios en el código fuente. *Facebook* y *Twitter* han obtenido niveles de mejora en un $10\,\%$ y $11\,\%$ respectivamente en sus aplicación de análisis de datos [43].

Otro posible experimento es utilizar el lenguaje de programación *Julia* en vez de *Python* para realizar el algoritmo *PageRank*. En el último tiempo *Julia* ha obtenido buena fama en la comunidad de desarrolladores científicos y computación de alto rendimiento.

En el nivel del sistema operativo, la nueva funcionalidad para realizar operaciones de entrada y salida asíncronas en el *kernel Linux* llamada `io_uring` es una interesante apuesta para evaluar el impacto en el rendimiento de la aplicación cuando nos enfocamos en el rendimiento de la capa de almacenamiento, dimensión a la cual no le hemos prestado mucha atención en el desarrollo de este trabajo.

La última mejora que podemos proponer en el ámbito tecnológico es investigar e implementar ajustes relacionados al rendimiento del *broker* de mensajes, *RabbitMQ*. Además, se podría evaluar utilizar un servicio administrado en vez de nuestra propia instancia para procesar mensajes.

5.2.2. Mejoras al rendimiento

Debido al diseño orientado a microservicios realizado, es posible mejorar cada componente de forma independiente de los otros, con tal de que se respeten los protocolos de comunicación entre el *broker* de mensajes y el componente que ha sido modificado, no existirá ningún problema con potenciales cambios.

Dicho esto, una herramienta que puede permitir obtener una mejor visibilidad del rendi-

[42] https://www.graalvm.org/use-cases/
[43] https://medium.com/graalvm/graalvm-at-facebook-af09338ac519

miento en cada uno de los componentes es *OpenTracing* [44]. El proyecto *OpenTracing* es una biblioteca disponible en múltiples lenguajes de programación, la cual permite generar reportes y registros sobre los tiempos de ejecución de nuestras funciones y líneas de código, junto con herramientas para analizar estos reportes.

Esta información nos permite entrar al mundo de las micro-optimizaciones de nuestros componentes y de esta forma, podemos lograr extraer el máximo rendimiento a nuestros recursos disponibles.

5.2.3. Nuevas funcionalidades

Para el eje de nuevas funcionalidades, podemos considerar la actualización del grafo dirigido de entidades en tiempo real. En la solución propuesta, solo se actualiza el contenido del índice de entidades con los datos de las actualizaciones provenientes de los servidores de *Wikidata*, pero al momento de actualizar esta información se reutiliza el valor de *PageRank* que fue calculado en el proceso inicial. Un desafío no menor e interesante es actualizar no solo los datos de las entidades, sino que también el valor de *PageRank* asignado a la entidad que ha sido actualizada.

Además de esto, podemos considerar aumentar la cantidad de pruebas automatizadas que han sido implementadas. Si bien las actuales pruebas logran demostrar la funcionalidad de nuestra solución, es necesario medir el porcentaje total de cobertura de estas y de ser posible, aumentarlo. De esta forma, logramos encontrar errores y solucionarlos junto con mejorar el nivel de calidad de la solución.

5.2.4. Despliegue de la aplicación

Con respecto al despliegue de la aplicación, podemos notar que tanto la solución original como nuestra solución no son altamente disponibles, debido a que ambas son desplegadas en un solo servidor. Un tema interesante es revisar la posibilidad de replicar y comunicar los componentes para actuar de forma distribuida y altamente disponible, aumentando de esta forma la resiliencia de todo el sistema y agregando una nueva dimensión a la escalabilidad de nuestra solución propuesta, escalabilidad horizontal.

Otro tema importante es diseñar e implementar un proceso automatizado para el despliegue de la solución. Este proceso debe ser capaz de construir los contenedores de nuestros componentes, ejecutar las pruebas automatizadas e instalar y reiniciar los servicios en el o los servidores donde será instalada la solución. Además de esto, es posible integrar herramientas automatizadas para la detección de errores y malas prácticas en el código fuente,

[44] https://opentracing.io/

como por ejemplo, *SonarQube* [45].

5.2.5. Interacción con el usuario

En el ámbito de la interacción del usuario debemos recordar la existencia del proyecto *RDFExplorer*, el cual, ha sido integrado sin problemas en el proyecto original pero no en nuestra solución debido a que se encuentra fuera de nuestro enfoque.

Además de esta integración, es posible revisar la posibilidad de utilizar técnicas de *caching* entre *RDFExplorer* y nuestra solución. Para esto, será necesario integrar el proceso de actualizaciones en tiempo real y algún componente que almacene aquellas entidades más utilizadas, como por ejemplo *Redis* [46] o *Memcached* [47].

[45] https://www.sonarqube.org/
[46] https://redis.io/
[47] https://memcached.org/

ANEXOS

5.1. Reporte de *Syscalls* para estrategias de construcción de grafos.

5.1.1. Estrategia SingleThread

% time	seconds	usecs / call	calls	errors	syscall
82,13	2,633515	3	720836	12	lseek
7,95	0,255053	4	54775	0	read
2,14	0,068684	23	2871	476	stat
2,08	0,066749	12	5253	0	mmap
1,70	0,054642	12	4485	0	munmap
0,98	0,031363	20	1512	0	fstat
0,77	0,024701	26	944	42	openat
0,65	0,020755	22	919	0	close
0,39	0,012634	19	636	616	ioctl
0,25	0,007906	36	215	17	futex
0,24	0,007569	302	25	0	write
0,19	0,006199	17	358	0	brk
0,18	0,005882	25	230	0	mprotect
0,17	0,005432	30	180	0	getdents64
0,06	0,001812	139	13	0	clone
0,02	0,000645	17	36	0	getpid
0,02	0,000518	32	16	0	unlink
0,01	0,000193	2	68	0	rt_sigaction
0,01	0,000396	18	22	0	pread64
0,01	0,000402	23	17	8	lstat
0,01	0,000266	33	8	0	link
0,01	0,000175	25	7	0	pipe2
0,00	0,000106	21	5	0	getrandom
0,00	0,000139	34	4	2	readlink
0,00	0,000091	22	4	0	uname
0,00	0,000143	47	3	0	getcwd
0,00	0,000122	40	3	0	fcntl
0,00	0,000090	30	3	1	access
0,00	0,000078	26	3	0	wait4
0,00	0,000055	18	3	0	sched_getaffinity
0,00	0,000047	15	3	0	sigaltstack
0,00	0,000000	0	3	0	dup
0,00	0,000057	28	2	0	prlimit64

Tabla 10, continuación de la página anterior

% time	seconds	usecs / call	calls	errors	syscall
0,00	0,000000	0	2	1	arch_prctl
0,00	0,000048	48	1	1	mkdir
0,00	0,000039	39	1	0	sysinfo
0,00	0,000038	38	1	0	rt_sigprocmask
0,00	0,000038	38	1	0	set_tid_address
0,00	0,000038	38	1	0	set_robust_list
0,00	0,000022	22	1	0	statfs
0,00	0,000000	0	1	0	execve
0,00	0,000000	0	1	0	getuid
0,00	0,000000	0	1	0	getgid
0,00	0,000000	0	1	0	geteuid
0,00	0,000000	0	1	0	getegid
0,00	0,000000	0	1	0	gettid
0,00	0,000000	0	1	0	geteuid
0,00	0,000000	0	1	0	getegid
0,00	0,000000	0	1	0	gettid
0,00	0,000030	30	1	0	getegid
0,00	0,000029	29	1	0	geteuid
0,00	0,000027	27	1	0	restart_syscall
0,00	0,000000	0	8	0	link
0,00	0,000000	0	3	0	sigaltstack
0,00	0,000000	0	1	0	execve
0,00	0,000000	0	1	1	mkdir
0,00	0,000000	0	1	0	statfs

Tabla 10: Reporte generado por `strace` para la estrategia SingleThread.

5.1.2. Estrategia ThreadedReader

% time	seconds	usecs / call	calls	errors	syscall
49,93	463,136627	76	6086105	826234	futex
47,91	444,353283	8103	54832	0	read
2,07	19,177303	26	720836	12	lseek
0,02	0,17949	27	6576	0	mmap
0,02	0,162522	28	5770	0	munmap
0,02	0,142847	17	8021	0	sched_yield
0,01	0,109215	38	2863	469	stat
0,01	0,055643	36	1539	0	fstat
0	0,044293	45	973	42	openat

Tabla 11, continuación de la página anterior

% time	seconds	usecs / call	calls	errors	syscall
0	0,040182	41	962	0	close
0	0,038093	586	65	0	write
0	0,021223	7074	3	0	wait4
0	0,018707	29	636	616	ioctl
0	0,01025	41	246	0	mprotect
0	0,009519	51	186	0	getdents64
0	0,006606	17	382	0	brk
0	0,005076	461	11	5	execve
0	0,002104	150	14	0	clone
0	0,002081	23	88	0	rt_sigaction
0	0,001368	85	16	0	unlink
0	0,001234	32	38	0	getpid
0	0,000942	55	17	8	lstat
0	0,000694	46	15	0	set_robust_list
0	0,000635	635	1	0	rt_sigprocmask
0	0,000483	60	8	0	link
0	0,000163	32	5	0	getrandom
0	0,000152	25	6	2	readlink
0	0,000152	50	3	0	gettid
0	0,000131	43	3	0	dup
0	0,000112	22	5	0	getcwd
0	0,00011	13	8	0	madvise
0	0,000096	16	6	4	access
0	0,000073	6	12	6	arch_prctl
0	0,000065	65	1	0	set_tid_address
0	0,000061	8	7	0	uname
0	0,000058	19	3	0	sched_getaffinity
0	0,000056	28	2	0	prlimit64
0	0,000055	55	1	1	mkdir
0	0,000037	5	7	0	fcntl
0	0,000026	0	40	0	pread64
0	0,000019	3	6	6	mbind
0	0,000006	1	5	0	geteuid
0	0,000005	1	3	0	getuid
0	0,000004	1	3	0	getgid
0	0,000004	1	3	0	getegid
0	0,000003	1	2	0	getppid
0	0	0	8	0	dup2
0	0	0	1	0	sysinfo
0	0	0	3	0	sigaltstack

Tabla 11, continuación de la página anterior

% time	seconds	usecs / call	calls	errors	syscall
0	0	0	1	0	statfs
0	0	0	7	0	pipe2
0,00	0,000027	27	1	0	restart_syscall
0,00	0,000000	0	8	0	link
0,00	0,000000	0	3	0	sigaltstack
0,00	0,000000	0	1	0	execve
0,00	0,000000	0	1	1	mkdir
0,00	0,000000	0	1	0	statfs

Tabla 11: Reporte generado por `strace` para la estrategia ThreadedReader.

5.1.3. Estrategia MultiProcess

% time	seconds	usecs / call	calls	errors	syscall
45,96	24433,867005	243	100508452	13972080	futex
21,89	11635,990083	294	39468891	4	read
15,69	8342,285687	595877549	14	0	wait4
14,59	7758,105083	517207005	15	0	accept4
1,80	958,030034	48	19690200	1	write
0,06	30,552761	15	1943208	12	lseek
0,00	0,427184	60	7061	0	mprotect
0,00	0,184344	45	4047	0	mmap
0,00	0,180841	54	3309	0	sched_yield
0,00	0,162868	50	3206	0	munmap
0,00	0,075373	26	2884	469	stat
0,00	0,035020	22	1561	0	fstat
0,00	0,025659	25	992	42	openat
0,00	0,024737	23	1034	0	close
0,00	0,015846	23	685	634	ioctl
0,00	0,009816	297	33	0	clone
0,00	0,008317	22	375	0	brk
0,00	0,004795	25	188	0	getdents64
0,00	0,002512	25	98	0	getpid
0,00	0,002499	28	89	0	rt_sigaction
0,00	0,002116	96	22	0	madvise
0,00	0,001263	37	34	0	set_robust_list
0,00	0,001067	26	41	0	getrandom
0,00	0,000851	38	22	0	gettid
0,00	0,000616	15	40	0	pread64

Tabla 12, continuación de la página anterior

% time	seconds	usecs / call	calls	errors	syscall
0,00	0,000597	33	18	0	unlink
0,00	0,000596	37	16	0	getsockname
0,00	0,000515	32	16	0	pipe2
0,00	0,000491	25	19	9	lstat
0,00	0,000333	22	15	0	connect
0,00	0,000294	49	6	6	mbind
0,00	0,000253	15	16	0	socket
0,00	0,000233	23	10	0	fcntl
0,00	0,000223	27	8	0	link
0,00	0,000175	175	1	0	rmdir
0,00	0,000118	16	7	0	uname
0,00	0,000084	28	3	0	sched_getaffinity
0,00	0,000079	26	3	0	dup
0,00	0,000075	6	12	6	arch_prctl
0,00	0,000074	12	6	2	readlink
0,00	0,000049	9	5	0	geteuid
0,00	0,000043	7	6	4	access
0,00	0,000035	5	6	0	getcwd
0,00	0,000026	8	3	0	getuid
0,00	0,000026	8	3	0	getegid
0,00	0,000025	8	3	0	getgid
0,00	0,000018	18	1	0	statfs
0,00	0,000000	0	1	0	poll
0,00	0,000000	0	1	0	rt_sigprocmask
0,00	0,000000	0	8	0	dup2
0,00	0,000000	0	1	0	bind
0,00	0,000000	0	1	0	listen
0,00	0,000000	0	1	0	setsockopt
0,00	0,000000	0	12	6	execve
0,00	0,000000	0	2	1	mkdir
0,00	0,000000	0	1	0	sysinfo
0,00	0,000000	0	2	0	getppid
0,00	0,000000	0	3	0	sigaltstack
0,00	0,000000	0	1	0	set_tid_address
0,00	0,000000	0	2	0	prlimit64
0,00	0,000000	0	3	0	sched_getaffinity
0,00	0,000039	19	2	0	prlimit64
0,00	0,000000	0	2	0	getppid
0,00	0,000063	63	1	0	bind
0,00	0,000040	40	1	0	listen

Tabla 12, continuación de la página anterior

% time	seconds	usecs / call	calls	errors	syscall
0,00	0,000040	40	1	0	sysinfo
0,00	0,000037	37	1	0	set_tid_address
0,00	0,000024	24	1	0	restart_syscall
0,00	0,000020	20	1	1	mkdir
0,00	0,000000	0	1	0	statfs

Tabla 12: Reporte generado por `strace` para la estrategia MultiProcess.

5.1.4. Estrategia MultiProcessZMQ

% time	seconds	usecs / call	calls	errors	syscall
53,00	1292,246149	591	2184868	0	epoll_wait
21,61	526,819552	5942	88659	2	read
12,91	314,767770	34974196	9	0	wait4
4,33	105,468760	13	7654168	0	getpid
2,92	71,152446	19	3715088	1	poll
1,64	40,098137	20	1943198	12	lseek
1,45	35,424908	16	2104548	14	recvfrom
1,36	33,102162	14	2307736	0	sendto
0,70	17,042166	10	1581111	0	epoll_ctl
0,07	1,586830	1548	1025	255	futex
0,01	0,205097	61	3341	0	sched_yield
0,00	0,074487	25	2866	469	stat
0,00	0,043288	23	1855	0	mmap
0,00	0,037157	23	1549	0	fstat
0,00	0,022996	21	1050	0	munmap
0,00	0,023776	22	1038	0	close
0,00	0,028010	28	984	42	openat
0,00	0,015451	23	647	627	ioctl
0,00	0,009739	26	370	0	brk
0,00	0,005731	22	259	0	mprotect
0,00	0,005154	27	186	0	getdents64
0,00	0,002559	19	130	0	fcntl
0,00	0,003377	36	93	0	write
0,00	0,002595	29	88	0	rt_sigaction
0,00	0,000573	14	40	0	pread64
0,00	0,000295	11	25	0	set_robust_list
0,00	0,005034	209	24	0	clone
0,00	0,000223	13	17	0	unlink

Tabla 13, continuación de la página anterior

% time	seconds	usecs / call	calls	errors	syscall
0,00	0,000030	1	17	8	lstat
0,00	0,000459	28	16	0	socketpair
0,00	0,000050	3	15	0	madvise
0,00	0,000523	40	13	0	pipe2
0,00	0,000000	0	12	6	execve
0,00	0,000000	0	12	6	arch_prctl
0,00	0,000313	34	9	0	rt_sigprocmask
0,00	0,000101	11	9	0	getpeername
0,00	0,000062	6	9	0	getsockopt
0,00	0,000222	27	8	0	sched_setscheduler
0,00	0,000204	25	8	0	sched_getscheduler
0,00	0,000175	21	8	0	sched_getparam
0,00	0,000163	20	8	0	epoll_create
0,00	0,000140	17	8	0	getrandom
0,00	0,000000	0	8	0	dup2
0,00	0,000000	0	8	0	link
0,00	0,001606	229	7	0	getsockname
0,00	0,000102	14	7	0	uname
0,00	0,000260	43	6	2	readlink
0,00	0,000074	12	6	6	mbind
0,00	0,000067	11	6	4	access
0,00	0,000112	22	5	0	gettid
0,00	0,000078	15	5	0	getcwd
0,00	0,000033	6	5	0	geteuid
0,00	0,000102	25	4	0	socket
0,00	0,000119	39	3	0	accept
0,00	0,000114	38	3	0	dup
0,00	0,000057	19	3	0	sigaltstack
0,00	0,000038	12	3	0	getuid
0,00	0,000036	12	3	0	getgid
0,00	0,000036	12	3	0	getegid
0,00	0,000021	7	3	0	connect
0,00	0,000000	0	3	0	sched_getaffinity
0,00	0,000039	19	2	0	prlimit64
0,00	0,000000	0	2	0	getppid
0,00	0,000063	63	1	0	bind
0,00	0,000040	40	1	0	listen
0,00	0,000040	40	1	0	sysinfo
0,00	0,000037	37	1	0	set_tid_address
0,00	0,000024	24	1	0	restart_syscall

Tabla 13, continuación de la página anterior

% time	seconds	usecs / call	calls	errors	syscall
0,00	0,000020	20	1	1	mkdir
0,00	0,000000	0	1	0	statfs

Tabla 13: Reporte generado por `strace` para la estrategia MultiProcessZMQ.

REFERENCIAS BIBLIOGRÁFICAS

[ope, 2018] (2018). The single unix specification, issue 7. In *The Single UNIX Specification*. The Open Group.

[Antoniou and Van Harmelen, 2004] Antoniou, G. and Van Harmelen, F. (2004). Web ontology language: Owl. In *Handbook on ontologies*, pages 67–92. Springer.

[Apache, 2012] Apache, S. F. (2012). Welcome to apache lucene.

[Araújo et al., 2009] Araújo, S., Schwabe, D., and Barbosa, S. (2009). Experimenting with explorator: a direct manipulation generic rdf browser and querying tool. *Visual Interfaces to the Social and the Semantic Web (VISSW 2009)*, *Sanibel Island*, *Florida*.

[Auer et al., 2009] Auer, S., Dietzold, S., Lehmann, J., Hellmann, S., and Aumueller, D. (2009). Triplify: light-weight linked data publication from relational databases. In *Proceedings of the 18th international conference on World wide web*, pages 621–630.

[authors, 2021] authors, T. Z. (2021). Zeromq. an open-source universal messaging library.

[Beazley, 2010] Beazley, D. (2010). Understanding the python gil. In *PyCON Python Conference. Atlanta*, *Georgia*.

[Beckett et al., 2014] Beckett, D., Berners-Lee, T., Prud'hommeaux, E., and Carothers, G. (2014). Rdf 1.1 turtle. *World Wide Web Consortium*.

[Beckett and McBride, 2004] Beckett, D. and McBride, B. (2004). Rdf/xml syntax specification (revised). *W3C recommendation*, 10(2.3).

[Berners-Lee et al., 2006] Berners-Lee, T., Chen, Y., Chilton, L., Connolly, D., Dhanaraj, R., Hollenbach, J., Lerer, A., and Sheets, D. (2006). Tabulator: Exploring and analyzing linked data on the semantic web. In *Proceedings of the 3rd international semantic web user interaction workshop*, volume 2006, page 159. Athens, Georgia.

[Berners-Lee et al., 1998] Berners-Lee, T. et al. (1998). Semantic web road map.

[Berners-Lee et al., 2001] Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific american*, 284(5):34–43.

[Bizer and Cyganiak, 2006] Bizer, C. and Cyganiak, R. (2006). D2r server-publishing relational databases on the semantic web. In *Poster at the 5th international semantic web conference*, volume 175.

[Bizer et al., 2011] Bizer, C., Heath, T., and Berners-Lee, T. (2011). Linked data: The story so far. In *Semantic services, interoperability and web applications: emerging concepts*, pages 205–227. IGI Global.

[Boley et al., 2010] Boley, H., Hallmark, G., Kifer, M., Paschke, A., Polleres, A., and Reynolds, D. (2010). Rif core dialect. w3c recommendation, 22 june 2010. *World Wide Web Consortium, June*.

[Broekstra et al., 2002] Broekstra, J., Kampman, A., and Van Harmelen, F. (2002). Sesame: A generic architecture for storing and querying rdf and rdf schema. In *International semantic web conference*, pages 54–68. Springer.

[Bruch et al., 2009] Bruch, M., Monperrus, M., and Mezini, M. (2009). Learning from examples to improve code completion systems. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 213–222.

[Business, 2021] Business, D. (2021). Phva: Planear, hacer, verificar y actuar.

[Campinas, 2014] Campinas, S. (2014). Live sparql auto-completion. In *International Semantic Web Conference (Posters & Demos)*, pages 477–480.

[Carroll, 2003] Carroll, J. J. (2003). Signing rdf graphs. In *International Semantic Web Conference*, pages 369–384. Springer.

[De la Parra, 2020] De la Parra, G. (2020). Autocompletion for sparql-based exploration and querying. *URL https://github.com/gabrieldelaparra/SPARQLforHumans*.

[Deutsch, 1996] Deutsch, L. P. (1996). GZIP file format specification version 4.3. RFC 1952.

[Dividino et al., 2009a] Dividino, R., Sizov, S., Staab, S., and Schueler, B. (2009a). Querying for provenance, trust, uncertainty and other meta knowledge in rdf. *Journal of Web Semantics*, 7(3):204–219.

[Dividino et al., 2009b] Dividino, R. Q., Schenk, S., Sizov, S., and Staab, S. (2009b). Provenance, trust, explanations-and all that other meta knowledge. *KI*, 23(2):24–30.

[Education, 2020] Education, I. C. (2020). What are message brokers?

[Eggen and Eggen, 2019] Eggen, R. and Eggen, M. (2019). Thread and process efficiency in python. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 32–36. The Steering Committee of The World Congress in Computer Science.

[Erxleben et al., 2014] Erxleben, F., Günther, M., Krötzsch, M., Mendez, J., and Vrandečić, D. (2014). Introducing wikidata to the linked data web. In *International semantic web conference*, pages 50–65. Springer.

[Ferré, 2017] Ferré, S. (2017). Sparklis: an expressive query builder for sparql endpoints with guidance in natural language. *Semantic Web*, 8(3):405–418.

[Flouris et al., 2009] Flouris, G., Fundulaki, I., Pediaditis, P., Theoharis, Y., and Christophides, V. (2009). Coloring rdf triples to capture provenance. In *International Semantic Web Conference*, pages 196–212. Springer.

[Foundation, 2021a] Foundation, A. S. (2021a). Apache activemq. flexible and powerful open source multi-protocol messaging.

[Foundation, 2021b] Foundation, A. S. (2021b). Apache kafka. an open-source distributed event streaming platform.

[Garg, 2013] Garg, N. (2013). *Apache kafka*. Packt Publishing Birmingham, UK.

[Harth et al., 2011] Harth, A., Janik, M., and Staab, S. (2011). Semantic web architecture. In *Handbook of Semantic Web Technologies*.

[Kerrisk, 2018] Kerrisk, M. (2018). System call tracing with strace. In *NDC TechTown 2018. Kongsberg, Norway*.

[Khare, 2018] Khare, K. (2018). Json vs protocol buffers vs flatbuffers.

[Kifer, 2008] Kifer, M. (2008). Rule interchange format: The framework. In *International Conference on Web Reasoning and Rule Systems*, pages 1–11. Springer.

[Kifer and Boley, 2013] Kifer, M. and Boley, H. (2013). Rif overview. *W3C working draft, W3C,(October 2009)*. *http://www.w3.org/TR/rif-overview*.

[Kiryakov et al., 2005] Kiryakov, A., Ognyanov, D., and Manov, D. (2005). Owlim–a pragmatic semantic repository for owl. In *International Conference on Web Information Systems Engineering*, pages 182–192. Springer.

[Kleinberg et al., 1998] Kleinberg, J. M. et al. (1998). Authoritative sources in a hyperlinked environment. In *SODA*, volume 98, pages 668–677. Citeseer.

[Kononenko et al., 2014] Kononenko, O., Baysal, O., Holmes, R., and Godfrey, M. W. (2014). Mining modern repositories with elasticsearch. In *Proceedings of the 11th working conference on mining software repositories*, pages 328–331.

[Kramer et al., 2013] Kramer, K., Dividino, R. Q., and Gröner, G. (2013). Space: Sparql index for efficient autocompletion. In *International Semantic Web Conference (Posters & Demos)*, pages 157–160. Citeseer.

[Ltd, 2021] Ltd, R. (2021). Redis. the real-time data platform.

[McBride, 2001] McBride, B. (2001). Jena: Implementing the rdf model and syntax specification. In *Proceedings of the Second International Conference on Semantic Web-Volume 40*, pages 23–28. CEUR-WS. org.

[McCarthy et al., 2012] McCarthy, L., Vandervalk, B., and Wilkinson, M. (2012). Sparql assist language-neutral query composer. *BMC bioinformatics*, 13(1):S2.

[Mihalcea, 2004] Mihalcea, R. (2004). Graph-based ranking algorithms for sentence extraction, applied to text summarization. In *Proceedings of the ACL interactive poster and demonstration sessions*, pages 170–173.

[Moreau et al., 2008] Moreau, L., Freire, J., Futrelle, J., McGrath, R. E., Myers, J., and Paulson, P. (2008). The open provenance model: An overview. In *International Provenance and Annotation Workshop*, pages 323–326. Springer.

[Neumann and Weikum, 2010] Neumann, T. and Weikum, G. (2010). The rdf-3x engine for scalable management of rdf data. *The VLDB Journal*, 19(1):91–113.

[Openlink, 2015] Openlink, V. (2015). Virtuoso universal server. *Openlink Software*.

[Oracle, 2021a] Oracle (2021a). Java platform, standard edition java flight recorder runtime guide.

[Oracle, 2021b] Oracle (2021b). The structure of the java virtual machine.

[Oracle, 2021c] Oracle (2021c). Understand the outofmemoryerror exception.

[Page et al., 1999] Page, L., Brin, S., Motwani, R., and Winograd, T. (1999). The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab.

[Proos and Carlsson, 2020] Proos, D. P. and Carlsson, N. (2020). Performance comparison of messaging protocols and serialization formats for digital twins in iov. In *2020 IFIP networking conference (networking)*, pages 10–18. IEEE.

[Rafes et al., 2018] Rafes, K., Abiteboul, S., Cohen-Boulakia, S., and Rance, B. (2018). Designing scientific sparql queries using autocompletion by snippets. In *2018 IEEE 14th International Conference on e-Science (e-Science)*, pages 234–244. IEEE.

[Recordon and Reed, 2006] Recordon, D. and Reed, D. (2006). Openid 2.0: a platform for user-centric identity management. In *Proceedings of the second ACM workshop on Digital identity management*, pages 11–16.

[Rescorla, 2000] Rescorla, E. (2000). Rfc2818: Http over tls.

[Rietveld and Hoekstra, 2017] Rietveld, L. and Hoekstra, R. (2017). The yasgui family of sparql clients 1. *Semantic Web*, 8(3):373–383.

[Sayers, 2004] Sayers, C. (2004). Node-centric rdf graph visualization. *Mobile and Media Systems Laboratory, HP Labs*, 71.

[Smiley et al., 2015] Smiley, D., Pugh, E., Parisa, K., and Mitchell, M. (2015). *Apache Solr enterprise search server*. Packt Publishing Ltd.

[Thompson et al., 2016] Thompson, B., Personick, M., and Cutcher, M. (2016). The bigdata® rdf graph database. In *Linked Data Management*, pages 221–266. Chapman and Hall/CRC.

[Valsecchi et al., 2015] Valsecchi, F., Abrate, M., Bacciu, C., Tesconi, M., and Marchetti, A. (2015). Dbpedia atlas: Mapping the uncharted lands of linked data. In *LDOW@ WWW*.

[Vargas, 2019] Vargas, H. (2019). Rdf explorer: A visual sparql query builder. *URL https://github.com/hvarg/RDFExplorer*.

[Vargas et al., 2019] Vargas, H., Buil-Aranda, C., Hogan, A., and López, C. (2019). Rdf explorer: A visual sparql query builder. In *International Semantic Web Conference*, pages 647–663. Springer.

[Venners, 1998] Venners, B. (1998). The java virtual machine. *Java and the Java virtual machine: definition, verification, validation*.

[VMWare, 2021] VMWare (2021). Rabbitmq open source enterprise messaging.

[Vrandečić and Krötzsch, 2014] Vrandečić, D. and Krötzsch, M. (2014). Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85.

[W3C et al., 2013] W3C et al. (2013). Sparql 1.1 query language. *World Wide Web Consortium. URL https://www.w3.org/TR/2013/REC-sparql11-query-20130321/*.

[W3C et al., 2014] W3C et al. (2014). Rdf 1.1 concepts and abstract syntax. *World Wide Web Consortium. URL https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/*.

[Wikipedia, 2022] Wikipedia (2022). Wikipedia:Statistics — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Wikipedia%3AStatistics&oldid=1062669931. [Online; accessed 08-January-2022].