La Velocidad de la LUZ

Base de Datos No Relacionales

Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Raúl Speroni - 41770478

Damián Pintos - 33352799

Sebastián Gadea - 41973703

Docentes:
Lorena Etcheverry (responsable)
Martín Giachino

Índice

Introducción

Cruzar (Sistema de información de archivos del pasado reciente) es un proyecto interdisciplinario surgido en la Universidad de la República (Udelar) en el año 2018, que involucra a docentes, egresados y estudiantes de diferentes servicios universitarios como las Facultades de Ingeniería, de Información y Comunicación (FIC) y de Ciencias Sociales, con el apoyo de la organización Madres y Familiares de Detenidos Desaparecidos y en acuerdo con el Grupo de Trabajo Verdad y Justicia. La Facultad de Ciencias Sociales fue el último servicio en integrarse, en el año 2020. El proyecto busca recuperar, sistematizar y analizar todos los documentos que conforman el Archivo Berruti -que fue encontrado durante la administración de la ministra Azucena Berrutti en el año 2007- que cuenta con más de 3 millones de imágenes. Es un archivo que se encuentra microfilmado y que se estructura como una unidad entre los rollos de microfilm y sus imágenes digitalizadas (Udelar, 2022).

Descripción del Problema

LUZ es una herramienta desarrollada en el marco del proyecto Cruzar que permite a usuarios finales e investigadores explorar el conjunto de datos mediante búsqueda de texto. Como se trata de una gran cantidad de documentos (3 millones aproximadamente), y muchas veces de baja calidad, es deseable contar con técnicas de búsqueda avanzada (como búsqueda con errores de edición, o comodines) y con una velocidad de respuesta adecuada que permita escalar a nuevos contingentes de documentos. En el presente informe cuando sea referido un documento o una página digitalizada se lo llamará hoja.

Objetivo

La intención del proyecto es comparar la performance para los métodos de almacenamiento y búsqueda de la base de datos relacional que se usa hoy en día (*PostgreSQL*), y la base documental *Elasticsearch*.

La comparación tendrá dos componentes, por un lado uno cuantitativo en el que en base a las métricas resultantes se realizará una comparación de medias de los tiempos de ejecución. Se utilizará el test Shapiro Wilk para chequear normalidad de las distribuciones. En el caso de comprobarse la normalidad de la distribución se utilizará el test t de Student’s, si no se cumpliera la hipótesis de normalidad se aplicaría el test estadístico no paramétrico de Wilcoxon.

Por otro lado, un componente cualitativo donde se comparan las performances para

las operaciones que LUZ permite realizar hoy en día. La idea es explorar qué posibilidades extra podrían habilitar otro tipo de motores de bases de datos, así como qué limitaciones se podrían introducir. Al mismo tiempo, aún si un cambio de motor estuviera justificado por las métricas elegidas, es necesario proveer de métodos prácticos para una posible migración. Esto significa trabajar en scripts de carga y consulta de información.

Bases de Datos

PostgreSQL y *Elasticsearch* son motores de bases de datos consolidados y muy utilizados en la industria. Si bien en el presente trabajo se comparan en los mismos casos de uso, la diferencias entre sí, son significativas:

1. **Modelo de Base de Datos**: La principal diferencia es que *PostgreSQL* implementa el modelo relacional y almacena los datos en filas y columnas dentro de tablas que pueden estar relacionadas. *PostgreSQL* permite además el uso de SQL para consultar los datos. *Elasticsearch* en cambio es una base documental distribuida que almacena documentos complejos serializados en formato JSON, el principal método de consulta es mediante una API Rest.
2. **Transacciones**: *PostgreSQL* soporta transacciones que aseguran la integridad de datos: el mecanismo permite la ejecución de todas las operaciones o ninguna, permitiendo volver atrás si alguna falla (rollback). En *Elasticsearch* al ser prioridad la velocidad no existe el concepto de transacción y por lo tanto no es posible hacer rollback en las operaciones.
3. **Esquemas**: *Elasticsearch* admite cualquier estructura y tipos de datos en los documentos JSON y hace un trabajo relativamente bueno infiriendo tipos como Boolean, Fechas y Números sin configuración previa, esto resulta práctico y da flexibilidad. En *PostgreSQL* las tablas deben ser creadas de antemano con columnas con tipos definidos. Este esquema rígido hace que *PostgreSQL* permita operaciones y funcionalidades que serían imposibles de otra manera.
4. **Teorema CAP:** Hay tres características que un motor de bases de datos distribuido puede ofrecer, aunque no al mismo tiempo: Consistencia (si un dato es escrito o actualizado, todas sus copias también lo serán), Disponibilidad (Cualquier pedido de un cliente tendrá siempre una respuesta), Tolerancia a la Partición (El sistema seguirá respondiendo aún si un nodo deja de estar disponible). *PostgreSQL* puede cumplir con Consistencia y Disponibilidad pero no es tolerante a fallos de un nodo y *Elasticsearch* sin embargo puede cumplir con Tolerancia a la Partición y Disponibilidad pero no consistencia inmediata en todas las réplicas.

Relacional: PostgreSQL

Las búsquedas actuales en LUZ se ejecutan sobre una base de datos *PostgreSQL* utilizando las funcionalidades de Full Text Search que se encuentran fuera del estándar SQL. Las consultas se realizan sobre dos vistas materializadas, una con la información de cada hoja y otra con atributos de las hojas tales como rollo, versión y tipo de documento.

Accedimos al código fuente de LUZ para analizar todas las secciones del código dónde se consulte a la base de datos y nos centramos en la búsqueda genérica, dado que es la principal funcionalidad de la aplicación y la que más interesa optimizar. Dejamos por fuera de este trabajo funcionalidades específicas como la validación de usuarios y usuarios del sistema o la extracción de una hoja específica, por ejemplo.

Por cada búsqueda en la aplicación se consulta en la base:

- Hojas que cumplen con las condiciones de la consulta, paginadas.
- Cantidad de hojas.
- Cantidad de hojas por cada rollo.
- Cantidad de hojas por cada versión.
- Cantidad de hojas por cada tipo de documento.
- Cantidad de hojas por cada origen.

La funcionalidad de Full Text Search permite que los textos sean preprocesados para construir un índice que es usado en tiempo de búsqueda. El preprocesamiento incluye:

- Separar texto en tokens: Es conveniente procesar y guardar las diferentes porciones de texto (en general palabras pero pueden ser números, direcciones de correo o símbolos) por separado. *PostgreSQL* usa un parser por defecto para hacer este preprocesamiento, pero se pueden configurar otros para casos específicos.
- Convertir tokens en lexemas: Un lexema es una cadena de texto pero a diferencia de un token sufre una normalización que hace que diferentes formas de una palabra (plural y singular, diferentes géneros, etc) correspondan al mismo lexema. Esto permite que las búsquedas no sean sensibles a estas variaciones de palabras. En este paso se eliminan típicamente las palabras neutras o stopwords, para lo cual se usa un diccionario de palabras para el idioma correspondiente (en español se trata de artículos y otras palabras muy frecuentes).
- Almacenamiento de los textos preprocesados: Esto puede ser por ejemplo una lista de lexemas normalizados con información adicional sobre la

posición donde aparecieron en el texto original.

Hay varias funciones de *PostgreSQL* que permiten hacer las búsquedas sobre el campo de texto preprocesado haciendo uso del índice almacenado (PostgreSQL, 2022a). En el caso particular de LUZ se utiliza la función *websearch_to_tsquery* para convertir el texto ingresado por quienes realizan la búsqueda en la aplicación web. Esta función tiene la desventaja de ser bastante limitada en cuanto a las opciones de búsqueda que admite, pero tiene la ventaja de nunca dar por inválido un texto de consulta. Para poder utilizar algún método más potente como *to_tsquery* es necesaria programación adicional que convierta el texto de entrada en una secuencia válida para el motor de bases de datos.

Los modificadores *websearch_to_tsquery* que admite son:

- texto normal
  busca por cada uno de los términos y en el resultado tienen que estar todos los términos.
- texto entre comillas
  busca los términos pero cada uno tiene que estar a continuación de otro y en el mismo orden.
- OR
  disyunción.
- -:
  elimina del conjunto de resultados los documentos que contengan un término o una expresión.

Documental: Elasticsearch

Más allá de las diferencias fundamentales entre motores relacionales y documentales como lo son *PostgreSQL* y *Elasticsearch*, en cuanto al indexado de texto para soportar búsquedas ambos motores preprocesan el texto de manera similar.

En el caso de *Elasticsearch*, sin necesidad de habilitar ninguna funcionalidad extra, se pueden definir *analyzers* para cada campo de texto. Estos *analyzers* realizan el preprocesamiento al indexar documentos, y hacen lo mismo para los textos que ingresen como términos de una consulta. Los *analyzers* consisten en diferentes pasos configurables, como la tokenización, conversión a lexemas y eliminación de palabras neutras. Al definir que un campo es de tipo texto se aplica un *analyzer* por defecto sin necesidad de configuración extra.

En caso de que se quiera personalizar la forma en que los campos de texto son tratados, por ejemplo guardando diferentes versiones preprocesadas de un mismo campo para hacer diferentes tipos de búsquedas en simultáneo, basta con definir en

la configuración del índice antes de indexar los documentos.

Metodología

Infraestructura

Las pruebas y experimentos se hicieron sobre contenedores en una MacBook Pro con chip M1 y 32GB de RAM. Se optó por correr las imágenes oficiales de Postgres (12.5) y Elasticsearch (8.2.2) con un límite de 5 cores y 22GB de RAM.

Ingesta de Datos

PostgreSQL

Se intentó replicar lo mejor posible el ambiente de producción de LUZ. Para ello el equipo de LUZ nos permitió la descarga de un volcado de los datos con sentencias *COPY*. Después de importar los datos fue necesario refrescar varias de las vistas materializadas de las que depende LUZ para las búsquedas.

Adicionalmente, para permitir el correcto funcionamiento de los vectores utilizados para permitir full text search en *PostgreSQL*, se tuvo que importar un archivo de palabras neutras, o stopwords, en español utilizado por el *PostgreSQL* de LUZ.

A continuación se testearon todas las consultas que utiliza LUZ y que se pueden ver en los Anexos 1 al 6. Si bien para refrescar estas vistas son utilizadas varias tablas y vistas de la base de datos, las búsquedas se realizan exclusivamente sobre las vistas materializadas **luz.textosporhoja** y **lblme.imgjsondata**.

Vista	Entradas	Espacio en disco
textosporhoja	2082494	7.02 GB
imgjsondata	75950	28 MB

En cada fila de **textosporhoja** se encuentran principalmente los campos rollo, hoja, versión, calificación, original, tipo y textohoja. En cada fila de **imgjsondata** se encuentran rollo, imagen, año, operador, tipodoc, origen y fecha. A continuación se describe la función de cada campo y cómo se relacionan.

Campo	Descripción	Relación
textosporhoja.rollo	Número de rollo	imgjsondata.rollo
textosporhoja.hoja	Número de hoja dentro de rollo	imgjsondata.imagen
textosporhoja.version	Versión	
textosporhoja.calificacion	Score asignado a la calidad del OCR	
textosporhoja.original	Título del rollo	
textosporhoja.tipo	Tipo	
textosporhoja.textohoja	Texto de la hoja	
imgjsondata.rollo	Número de rollo	textosporhoja.rollo
imgjsondata.imagen	Número de hoja dentro del rollo	textosporhoja.hoja
imgjsondata.año	Año	
imgjsondata.operador	Operador	
imgjsondata.tipodoc	Tipo	
imgjsondata.origen	Origen	

En general las consultas SQL que hace LUZ (Anexos 1 al 6) devuelven campos sobre la relación entre **imgjsondata** y **textosporhoja** cuando coinciden los campos rollo y página. Como puede haber varios tipos y orígenes por cada hoja de cada rollo en general las consultas devuelven un array de orígenes y tipos por cada hoja.

Sobre las consultas que ejecuta hoy LUZ encontramos dos oportunidades de posible optimización:

- convertir en campos de la vista los valores que se encuentran en **lblme.imgjsondata** en formato json.
- los joins son innecesarios cuando no se consulta por los campos que se obtienen. Este cambio implica más programación y un código fuente un tanto más engorroso.

Si bien la consulta es optimizable el objetivo del presente trabajo es el de comparar

la solución actual con una solución en *Elasticsearch*.

Para tomar la información que está en *PostgreSQL* y volcarla en la base *Elasticsearch* ejecutamos una única consulta que devuelve por cada resultado una hoja y toda la información necesaria tanto para su visualización como para la búsqueda. La consulta necesaria para el volcado está disponible en el Anexo 7.

Elasticsearch

Para la ingesta de datos en *Elasticsearch* el primer paso fue decidir una estructura para la información resultante de las consultas a *PostgreSQL*. Se optó por cargar cada hoja de cada rollo como un documento independiente agregando los campos vinculados de **imgjsondata** a cada documento. La ausencia de consultas relacionales tipo join en *Elasticsearch* hace que esta sea una estructura razonable, ya que los campos que se repiten entre documentos no son numerosos ni pesados. Esta estructura permite además diseñar consultas equivalentes a las realizadas por LUZ en *PostgreSQL*.

Índice	Entradas	Espacio en disco
luz	2082494	5.76 GB

El procedimiento para la carga de datos fue exportar un archivo csv a partir de una consulta en *PostgreSQL* (Anexo 7) y cargar cada fila como documento en Elasticsearch usando la API de Elasticsearch y un script de Python (Anexo 8).

Consulta de Datos

Para cumplir con las funcionalidades de búsqueda actuales LUZ realiza una serie de consultas SQL con una serie de filtros dependiendo lo que se esté buscando. Para todos los experimentos usamos las mismas consultas en el mismo orden para *PostgreSQL*, y diseñamos consultas para *Elasticsearch* que devolvieran los mismos campos, algunas veces en menos pasos.

Los filtros consisten en limitar el espacio de búsqueda en el momento de hacer la consulta. Estos filtros están configurados y funcionan para ambos motores: en SQL mediante cláusulas where y en Elasticsearch mediante el bloque filter. De todas formas los experimentos se hicieron sobre el espacio de búsqueda lo más grande posible por lo que no se usaron filtros adicionales al de la búsqueda por palabras clave.

Una búsqueda típica en LUZ

Un caso de uso típico de LUZ es intentar recuperar todos los documentos que mencionen a determinada persona. Por ejemplo, para buscar hojas que contengan las palabras “Zelmar” y “Michelini” en *PostgreSQL,* LUZ hace actualmente 5 consultas secuenciales:

1. Cuántas hojas hay como resultados de la búsqueda.
2. Cuántos resultados (hojas) hay con tipo “ROLLO” en el campo tipodoc.
3. Cuántos resultados (hojas) hay con tipo “VERSION” en el campo tipodoc.
4. Cuántos resultados (hojas) hay por cada TIPODOC
5. Cuántos resultados (hojas) hay por cada ORIGEN
6. Devolver las primeras X hojas de los resultados de búsqueda.[1]

Para hacer la misma búsqueda en *Elasticsearch* se deben hacer las siguientes consultas:

1. Cuántos resultados (hojas) hay con tipo “ROLLO” en el campo tipodoc[2].
2. Cuántos resultados (hojas) hay con tipo “VERSION” en el campo tipodoc.
3. Cuántos resultados (hojas) hay por cada TIPODOC
4. Cuántos resultados (hojas) hay por cada origen
5. Devolver las primeras X hojas de los resultados de búsqueda mostrando el total de resultados.

Otras búsquedas posibles en LUZ

Como se explicó antes en LUZ actualmente es posible buscar hojas que contengan todas las palabras buscadas (AND), alguna de las palabras (OR), frases exactas, y disyunciones. Si bien *PostgreSQL* permite hacer búsquedas más complejas, como por prefijos, distancia entre palabras y otro tipo de mecanismos que podrían ser útiles por ejemplo para la búsqueda de nombres propios, hoy LUZ no los utiliza, y para hacerlo requeriría la implementación de un parser no trivial que traduzca el contenido del cuadro de búsqueda a las consultas deseadas.

En *Elasticsearch* en cambio se está utilizando el mecanismo de consulta *query_string* que habilita una serie de operadores dentro de las consultas que pueden aumentar significativamente las posibilidades de LUZ entre los que se

---

[1] Por cómo se muestran los resultados en la página web, LUZ le pide a *PostgreSQL* solo los primeros 200 carácteres de cada página de los resultados. Para los experimentos quitamos esta restricción para que ambos motores sean comparables.

[2] Es posible pedir el número total de matcheos en cualquier consulta de *Elasticsearch* además de la página que se quiere recuperar.

destacan el uso de comodines y distancia entre términos.

Adicionalmente ambos motores pueden ser configurados para usar múltiples estrategias de indexado de texto, de manera de expandir las posibilidades de búsqueda. Un buen ejemplo son los *pipelines* de *Elasticsearch* que permiten especificar cómo realizar la tokenización de un *analyzer*. Usando esos *pipelines* se puede guardar una versión de los textos con cierta transformación como la versión fonética de las palabras. Al hacer la búsqueda sobre esos campos expandidos el término de búsqueda también sufre la misma transformación. En este escenario, las palabras “Zelmar” y “Selmar” serían guardadas como “SLMR” (además de su versión original) habilitando una forma de búsqueda más laxa sobre ese campo. Estas posibilidades parecen potencialmente útiles para documentos resultantes de OCR sobre PDFs de mala calidad como los de LUZ.

Todos los experimentos reportados en la sección resultados son sobre consultas que ambos motores pueden resolver y usando las estrategias de indexado por defecto de cada motor.

Diseño de Experimentos

Siguiendo con el criterio de hacer consultas del tipo que hace LUZ hoy en día se decidió consultar sobre las siguientes dimensiones:

1. Cuatro personas para las cuales hubiera diferente número de hojas en la base:
    a. Lucía Topolansky (63 hojas)
    b. Zelmar Michelini (entre 1460 y 1483 hojas)
    c. Wilson Ferreira (entre 5512 y 5519 hojas)
    d. María Rodriguez (entre 10100 y 10125 hojas)
2. Tres cantidades de hojas a recuperar en cada página:
    a. 10
    b. 1000
    c. 10000
3. Mostrar o no dónde se produjo el “match” dentro del texto:
    a. Si
    b. No

Las cantidades de hojas a recuperar en cada llamado, lo que comunmente se conoce como paginación, es relevante en el contexto de internet -donde se usará LUZ- ya que el tráfico de datos afecta en la performance significativamente. En este

caso, como los experimentos se hicieron localmente, esta dimensión no es tan importante.

Decidimos agregar la tercera dimensión, una funcionalidad comunmente conocida como “highlight” porque es posible implementarla tanto en *PostgreSQL* y *Elasticsearch*. Aunque LUZ no la esté utilizando actualmente, parece razonable querer mostrarle al usuario en que posición de la hoja se encontró la palabra.

Lo anterior da como resultado 4*3*2 = **24** experimentos a realizar. Cada experimento consiste en ejecutar secuencialmente las consultas involucradas para cada uno de los dos motores. Además de analizar los tiempos de ejecución se analizaron las hojas recuperadas por los dos motores, qué hojas fueron recuperadas por ambos, y cuáles fueron recuperadas por sólo uno de ellos. En la sección resultados se analizarán los tiempos de respuesta y las posibles razones para la diferencia en los conjuntos de resultados. Cada experimento fue ejecutado **200** veces.

Resultados

En esta sección se presentan los resultados de las comparaciones de ambos motores, por un lado de los tiempos en las ejecuciones de las búsquedas, y por otro la composición de los conjuntos de hojas recuperadas por cada uno.

Tiempos de ejecución

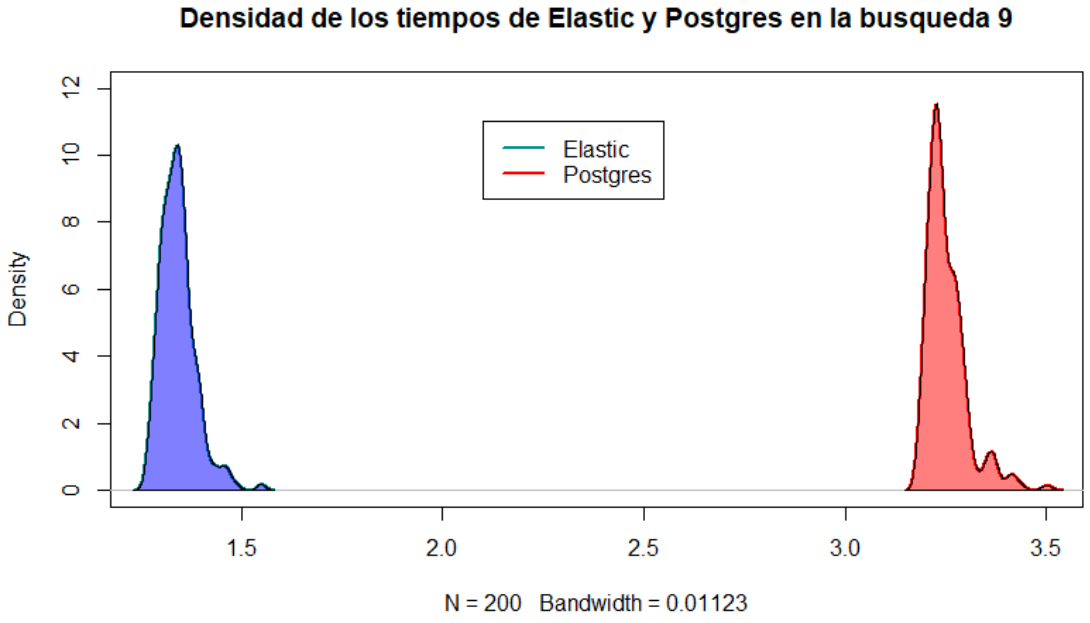
Para las comparaciones de performance de tiempos, se eligieron las búsquedas *“Zelmar Michelini”, “Wilson Ferreira”, “María Rodriguez” y “Lucía Topolansky*”. Para cada una de ellas se realizan las comparaciones de tiempos eligiendo 3 diferentes cantidades de hojas a recuperar. Para los primeros tres casos de cada búsqueda se pide al motor de búsqueda que resalte las palabras que produjeron el resultado.

Características de la búsqueda			Tiempo promedio		Resultados					Wilcoxon rank sum test	Shapiro-Wilk normality test	
n°	query	hojas	Elastic	Postgres	En ambos	Elastic	Postgres	Solo en Elastic	Solo en Postgres	p valor	p valor - Elastic	p valor - Postgres
1	Zelmar Michelini	10	0,01043	1,25599	1	10	10	9	9	< 2.2e-16	< 2.2e-16	2.472e-15
2	Zelmar Michelini	1000	0,21524	1,33576	700	1000	1000	300	300	< 2.2e-16	8.832e-15	2.442e-13
3	Zelmar Michelini	10000	0,38796	1,37641	1459	1483	1460	24	1	< 2.2e-16	2.178e-09	2.649e-13
4	Zelmar Michelini	10	0,01187	0,64071	1	10	10	9	9	< 2.2e-16	1.233e-07	2.835e-15
5	Zelmar Michelini	1000	0,13998	0,72366	700	1000	1000	300	300	< 2.2e-16	2.606e-13	7.247e-14
6	Zelmar Michelini	10000	0,26289	0,79236	1459	1483	1460	24	1	< 2.2e-16	6.906e-12	0.0001341
7	Wilson Ferreira	10	0,01747	2,88117	1	10	10	9	9	< 2.2e-16	< 2.2e-16	1.954e-13
8	Wilson Ferreira	1000	0,20876	2,95100	545	1000	1000	455	455	< 2.2e-16	2.08e-15	9.065e-11
9	Wilson Ferreira	10000	1,34332	3,25218	5509	5519	5512	10	3	< 2.2e-16	8.294e-08	7.321e-13
10	Wilson Ferreira	10	0,01369	0,78056	1	10	10	9	9	< 2.2e-16	4.302e-05	4.302e-05
11	Wilson Ferreira	1000	0,12091	0,85158	545	1000	1000	455	455	< 2.2e-16	< 2.2e-16	< 2.2e-16
12	Wilson Ferreira	10000	0,80177	3,77614	5509	5519	5512	10	3	< 2.2e-16	0.000532	< 2.2e-16
13	María Rodriguez	10	0,01586	2,34494	4	10	10	6	6	< 2.2e-16	5.045e-14	< 2.2e-16
14	María Rodriguez	1000	0,22614	2,40213	451	1000	1000	549	549	< 2.2e-16	4.112e-13	5.069e-16
15	María Rodriguez	10000	2,16075	3,01922	9871	10000	10000	129	129	< 2.2e-16	1.742e-07	6.055e-16
16	María Rodriguez	10	0,01645	0,75519	4	10	10	6	6	< 2.2e-16	< 2.2e-16	5.543e-12
17	María Rodriguez	1000	0,13359	0,80329	451	1000	1000	549	549	< 2.2e-16	9.843e-16	1.067e-11
18	María Rodriguez	10000	1,23053	1,37632	9871	10000	10000	129	129	< 2.2e-16	5.272e-05	1.641e-05
19	Lucía Topolansky	10	0,01094	0,63178	2	10	10	8	8	< 2.2e-16	< 2.2e-16	< 2.2e-16
20	Lucía Topolansky	1000	0,02372	0,65343	63	63	63	0	0	< 2.2e-16	1.236e-09	1.011e-11
21	Lucía Topolansky	10000	0,02514	0,65315	63	63	63	0	0	< 2.2e-16	1.347e-13	< 2.2e-16
22	Lucía Topolansky	10	0,01087	0,61860	2	10	10	8	8	< 2.2e-16	7.152e-10	7.14e-14
23	Lucía Topolansky	1000	0,01907	0,61974	63	63	63	0	0	< 2.2e-16	1.052e-13	< 2.2e-16
24	Lucía Topolansky	10000	0,01887	0,62339	63	63	63	0	0	< 2.2e-16	< 2.2e-16	4.509e-13

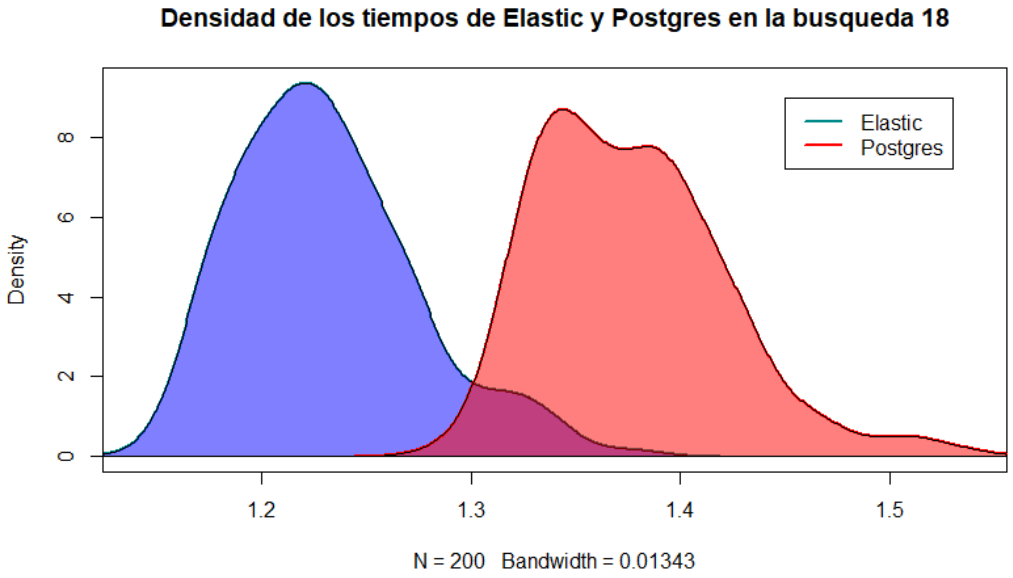
Como fue mencionado anteriormente, previo a la realización de la comparación de medias en los tiempos de ejecución de las búsquedas se realizó el test Shapiro Wilk sobre las distribuciones estimadas, para determinar si se utilizaba el test paramétrico t de Student's ó el no paramétrico Wilcoxon rank sum. Lo que finalmente se obtuvo fue que se rechaza la hipótesis nula de normalidad para todas distribuciones de tiempos de ambos motores de búsqueda, en test Shapiro Wilk, por lo que fue utilizado finalmente el test no paramétrico Wilcoxon rank sum, para la comparación de medias.

Los resultados son contundentes en la comparación de medias, ya que para todos los escenarios planteados los tiempos del motor de búsqueda *Elasticsearch* son menores estadísticamente significativos, en relación a los tiempos de ejecución de *PostgreSQL.*

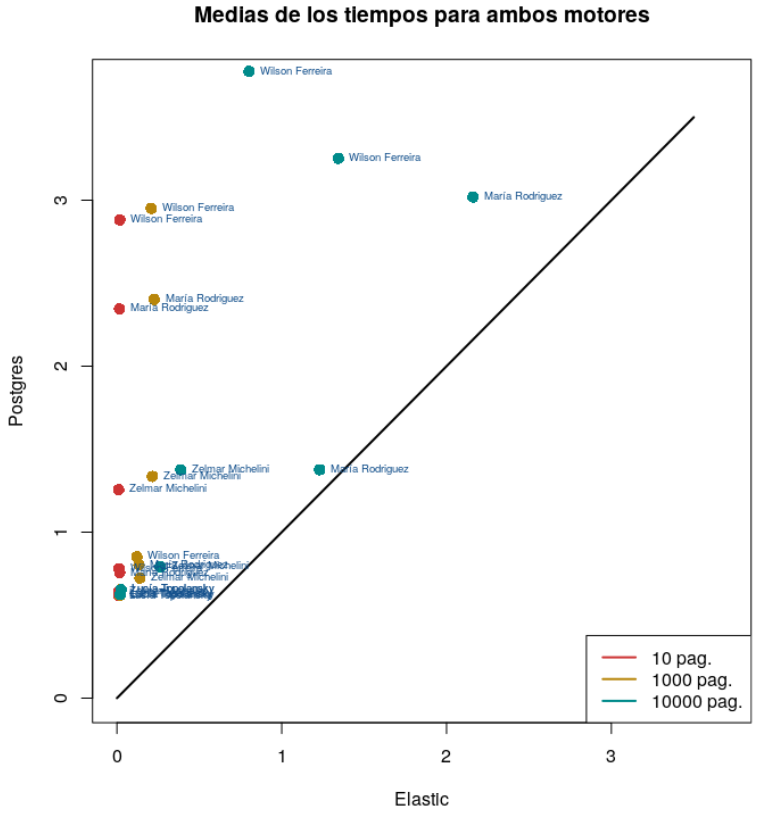
A continuación se realizan algunas visualizaciones a modo de ejemplo:



La siguiente figura muestra la comparación con menos diferencia entre las medias, donde igualmente los tiempos del motor de búsqueda *Elasticsearch* son menores a los tiempos de ejecución de *PostgreSQL.*

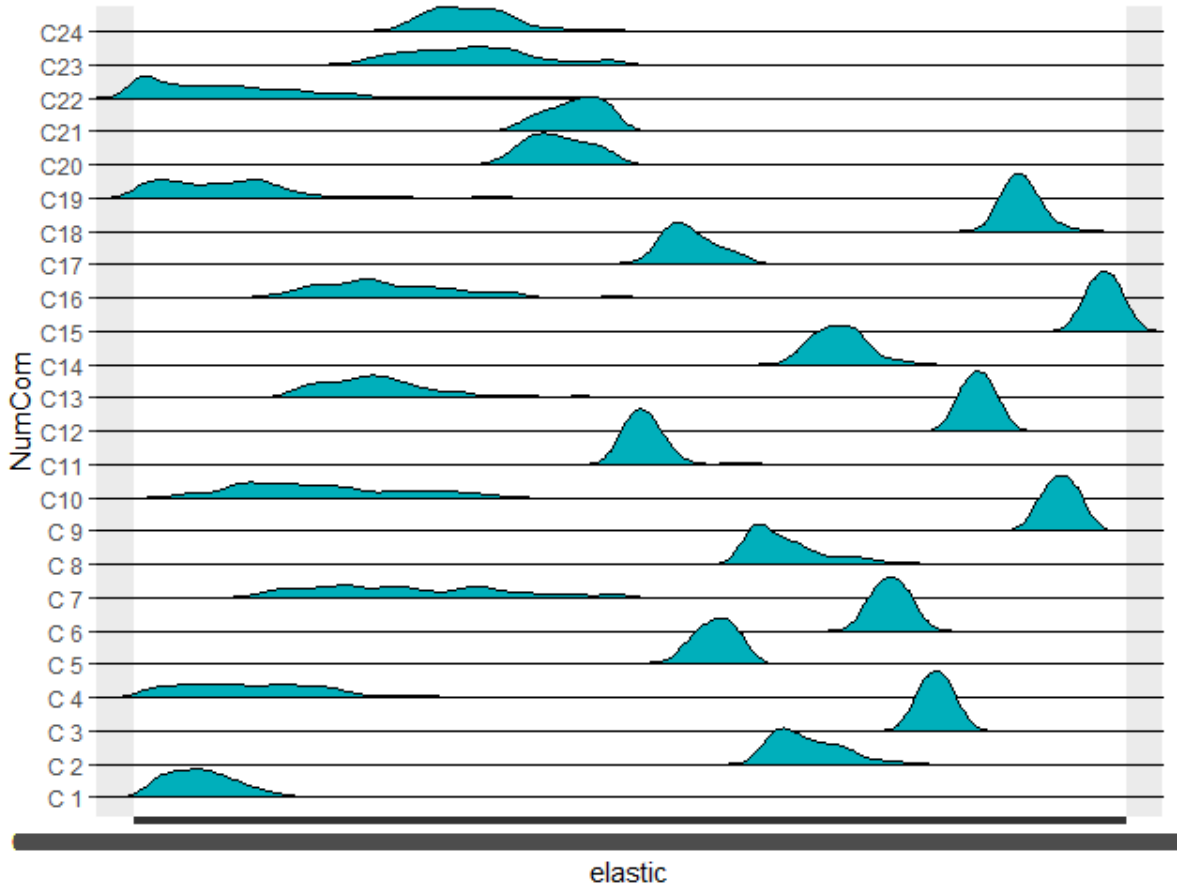


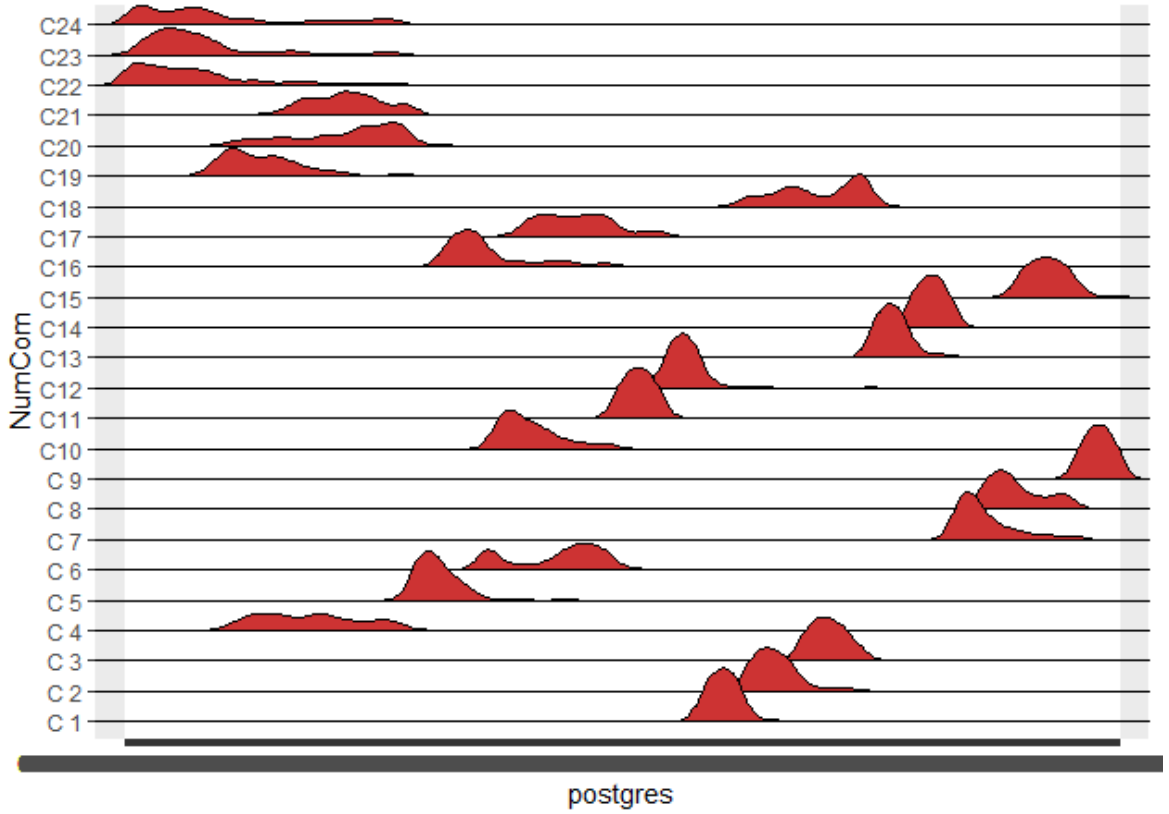
Se grafican las medias de los tiempos de ambos motores de búsqueda a modo de resumen, para mostrar cómo todas las medias de los tiempos de *Elasticsearch* son menores a las de *PostgreSQL.*



A modo de visualización general se grafican todas las densidades estimadas, pero

con un diferente valor mayor de bandwidth:





En esta gráfica puede verse cómo aumenta el valor de las medias al aumentar la cantidad de hojas utilizadas. Por ejemplo, en las tres primeras comparaciones (C1, C2 y C3), relacionadas a *"Zelmar Michelini"* vemos un aumento de la media de los tiempos cuando se pasa de la comparación C1 con 100 hojas, a la C2 con 1000 y a la C3 con 10.000 hojas. La misma lectura puede hacerse cada tres comparaciones.

Resultados recuperados

A modo de ejemplo, en el tercer experimento, para la búsqueda de "Zelmar Michelini" que trae todos los resultados encontrados en cada motor, nos encontramos con que hay 1460 hojas que son devueltas por ambos motores, 26 que sólo fueron encontradas por *Elasticsearch*, y 1 que fue encontrada sólo por *PostgreSQL*.

Se analiza el tercer experimento porque es el que trae todos los resultados. En los dos primeros, al estar paginado el conjunto de resultados depende del orden asignado por cada motor, por lo que la comparación de conjuntos deja de tener sentido.

Veamos una que solo fue encontrada en *PostgreSQL*:

- Casa de la Cultura "***Zelmar***:***Michelini***" . de la lista

Veamos algunas de los encontradas solo en Elasticsearch:

- Senadores por eb Partido Demáécratas _ Cristiano Sublema Movimiento por el Gobierno del L Pueblo-***Zelmar Michelini***-Movimiento Socialista.Li | -ta 99-Corresponde al Sector Socialista dentro del Frente Amplio
- de Colonia 3 “Dor el PDC-Sublema "Movimiento por el Gobierno del ,Lista 999-Corresponde al Pueblo-***Zelmar Michelini***" Sector Socialista- dentro del F:Amplio para las E.

En ambos casos el paso de tokenización falló en separar los símbolos, como los dos puntos o los guiones, de las palabras haciendo que los tokens almacenados sean para el primer caso “*Zelmar:Michelini” y* el segundo caso “*Pueblo-Zelmar*”. Esta situación se puede resolver para ambos motores modificando las reglas sobre cómo debe hacerse la tokenización.

Conclusiones

Teniendo en cuenta que LUZ es un sitio creado para buscar documentos, tiene sentido utilizar un motor de base de datos que permita realizar múltiples tipos de búsquedas en el contexto web que permitan la más completa y eficaz recuperación de información para los investigadores.

Desde el punto de vista cuantitativo, es claro que los tiempos de ejecuciòn de las consultas del motor de búsqueda *Elasticsearch* son menores a los de *PostgreSQL,* para cualquier paginado, si se les pide que subrayen o no la búsqueda y si es una búsqueda que aparece mucho en los documentos o no.

Desde el punto de vista de la infraestructura, ambos motores tienen requisitos similares, y el conjunto de datos ocupa un espacio similar en disco. Sin embargo, y como se puede apreciar en los scripts de carga y búsqueda, la configuración y la programación necesarias para ejecutar las búsquedas resultan más sencillas y claras en *Elasticsearch* que en *PostgreSQL*.

Desde el punto de vista cualitativo si bien las consultas sobre *PostgreSQL* se pueden optimizar, y las funcionalidades se pueden extender hasta cierto punto, resulta evidente que la implementación resulta forzada y el costo de programación elevado. *Elasticsearch* por otra parte facilita las cosas por su API rest, sus diferentes tipos de query para búsqueda de texto, y las facilidades para búsquedas con errores de edición, comodines y otras funcionalidades.

Anexo

Anexo 1 - Consulta Principal PostgreSQL

```
SELECT DISTINCT th.indice,
                th.original,
                th.rollo,
                th.idhoja,
                th.version,
                th.tipo,
                th.calificacion,
                SUBSTRING(th.textohoja, 1, 200) AS textohoja
FROM luz.textosporhoja th
LEFT JOIN (SELECT *,
           UNNEST(STRING_TO_ARRAY(tipodoc, '|')) tipodoc_simple
           FROM lblme.imgjsondata) lth
ON th.rollo = lth.rollo AND th.idhoja = lth.imagen
LEFT JOIN (SELECT *,
           UNNEST(STRING_TO_ARRAY(origen, '|')) origen_simple
           FROM lblme.imgjsondata) loh
ON th.rollo = loh.rollo AND th.idhoja = loh.imagen
WHERE ($4 = FALSE OR th.hojalexemes @@ websearch_to_tsquery('public.mh_simple', $3))
AND ($6 = FALSE OR th.rollo IN ($5:csv))
AND ($8 = FALSE OR
     CONCAT(th.rollo::text, ' - ', th.version::text, ' - ',
     th.original) IN ($7:csv))
AND ($10 = FALSE OR tipodoc_simple IN ($9:csv))
AND ($12 = FALSE OR origen_simple IN ($11:csv))
ORDER BY ${orderQuery}
LIMIT $1 OFFSET $2;
```

Anexo 2 - Cantidad de hojas, PostgreSQL

```
SELECT COUNT(DISTINCT (th.rollo, th.idhoja, th.version)) AS recordsfiltered
FROM luz.textosporhoja th
LEFT JOIN (SELECT *,
           UNNEST(STRING_TO_ARRAY(tipodoc, '|')) tipodoc_simple
           FROM lblme.imgjsondata) lth
ON th.rollo = lth.rollo AND th.idhoja = lth.imagen
LEFT JOIN (SELECT *,
           UNNEST(STRING_TO_ARRAY(origen, '|')) origen_simple
           FROM lblme.imgjsondata) loh
ON th.rollo = loh.rollo AND th.idhoja = loh.imagen
WHERE
  ($4 = FALSE OR
    th.hojalexemes @@ websearch_to_tsquery('public.mh_simple', $3))
AND
       ($6 = FALSE OR th.rollo IN ($5:csv))
AND
       ($8 = FALSE OR CONCAT(th.rollo::text, ' - ',
```

```
th.version::text, ' - ', th.original) IN ($7:csv))
AND
       ($10 = FALSE OR tipodoc_simple IN ($9:csv))
AND
       ($12 = FALSE OR origen_simple IN ($11:csv));
```

Anexo 3 - Cantidad de hojas por cada rollo, PostgreSQL

```
SELECT fh.valor_faceta::int AS value,
CASE WHEN fhr.count IS NULL THEN 0 ELSE fhr.count END AS count
FROM luz.facetashoja fh
LEFT JOIN (
  SELECT DISTINCT facet_rollo.rollo AS value,
        (COUNT(*) OVER (PARTITION BY facet_rollo.rollo)) AS count
  FROM (
       SELECT DISTINCT th.rollo, th.idhoja, th.version
       FROM luz.textosporhoja th
       LEFT JOIN (SELECT *,
                  UNNEST(STRING_TO_ARRAY(tipodoc, '|'))
                      tipodoc_simple
                  FROM lblme.imgjsondata) lth
            ON th.rollo = lth.rollo AND th.idhoja = lth.imagen
       LEFT JOIN (SELECT *,
                  UNNEST(STRING_TO_ARRAY(origen, '|')) origen_simple
                  FROM lblme.imgjsondata) loh
            ON th.rollo = loh.rollo AND th.idhoja = loh.imagen
       WHERE
         ($4 = FALSE OR th.hojalexemes @@
               websearch_to_tsquery('public.mh_simple', $3))
         AND ($6 = FALSE OR th.rollo IN ($5:csv))
         AND ($8 = FALSE OR CONCAT(th.rollo::text, ' - ',
             th.version::text, ' - ', th.original) IN ($7:csv))
         AND ($10 = FALSE OR tipodoc_simple IN ($9:csv))
         AND ($12 = FALSE OR origen_simple IN ($11:csv))
       ) AS facet_rollo) fhr
    ON fh.valor_faceta = fhr.value::text
 WHERE fh.tipo_faceta = 'ROLLO'
 ORDER BY fhr.value;
```

Anexo 4 - Cantidad de hojas por cada versión, PostgreSQL

```
SELECT fh.valor_faceta AS value,
    CASE WHEN fhr.count IS NULL THEN 0 ELSE fhr.count END AS count
```

```
FROM luz.facetashoja fh
LEFT JOIN (
  SELECT DISTINCT CONCAT(facet_version.rollo::text, ' - ',
                         facet_version.version::text, ' - ',
                         facet_version.original) AS value,
        (COUNT(*) OVER (PARTITION BY facet_version.rollo,
                        facet_version.version)) AS count
  FROM (
    SELECT DISTINCT th.rollo, th.idhoja, th.version, th.original
    FROM luz.textosporhoja th
    LEFT JOIN (SELECT *,
                UNNEST(STRING_TO_ARRAY(tipodoc, '|')) tipodoc_simple
               FROM lblme.imgjsondata) lth
    ON th.rollo = lth.rollo AND th.idhoja = lth.imagen
    LEFT JOIN (SELECT *,
               UNNEST(STRING_TO_ARRAY(origen, '|')) origen_simple
               FROM lblme.imgjsondata) loh
    ON th.rollo = loh.rollo AND th.idhoja = loh.imagen
    WHERE ($4 = FALSE OR th.hojalexemes @@
           websearch_to_tsquery('public.mh_simple', $3))
    AND ($6 = FALSE OR th.rollo IN ($5:csv))
    AND ($8 = FALSE OR
         CONCAT(th.rollo::text, ' - ', th.version::text, ' - ',
         th.original) IN ($7:csv))
    AND ($10 = FALSE OR tipodoc_simple IN ($9:csv))
    AND ($12 = FALSE OR origen_simple IN ($11:csv))
    ) AS facet_version) fhr
ON fh.valor_faceta = fhr.value::text
WHERE fh.tipo_faceta = 'VERSION'
ORDER BY value;
```

Anexo 5 - Cantidad de hojas tipo de documento, PostgreSQL

```
SELECT fh.valor_faceta AS value,
     CASE WHEN fhr.count IS NULL THEN 0 ELSE fhr.count END AS count
FROM luz.facetashoja fh
LEFT JOIN (
  SELECT DISTINCT faceta_tipodoc_simple.tipodoc_simple AS value,
       (COUNT(*) OVER (PARTITION BY
                       faceta_tipodoc_simple.tipodoc_simple))
       AS count
  FROM (
    SELECT DISTINCT th.rollo, th.idhoja, th.version, tipodoc_simple
    FROM luz.textosporhoja th
    LEFT JOIN (SELECT *,
               UNNEST(STRING_TO_ARRAY(tipodoc, '|')) tipodoc_simple
               FROM lblme.imgjsondata) lth
    ON th.rollo = lth.rollo AND th.idhoja = lth.imagen
    LEFT JOIN (SELECT *,
               UNNEST(STRING_TO_ARRAY(origen, '|')) origen_simple
               FROM lblme.imgjsondata) loh
    ON th.rollo = loh.rollo AND th.idhoja = loh.imagen
```

```
    WHERE ($4 = FALSE OR th.hojalexemes @@
             websearch_to_tsquery('public.mh_simple', $3))
    AND ($6 = FALSE OR th.rollo IN ($5:csv))
    AND ($8 = FALSE OR
         CONCAT(th.rollo::text, ' - ', th.version::text, ' - ',
         th.original) IN ($7:csv))
    AND ($10 = FALSE OR tipodoc_simple IN ($9:csv))
    AND ($12 = FALSE OR origen_simple IN ($11:csv))
    ) AS faceta_tipodoc_simple) fhr
ON fh.valor_faceta = fhr.value::text
WHERE fh.tipo_faceta = 'TIPODOC'
ORDER BY value;
```

Anexo 6 - Cantidad de hojas por cada origen, PostgreSQL

```
SELECT fh.valor_faceta AS value,
      CASE WHEN fhr.count IS NULL THEN 0 ELSE fhr.count END AS count
FROM luz.facetashoja fh
LEFT JOIN (
  SELECT DISTINCT faceta_origen_simple.origen_simple
         AS value,
         (COUNT(*) OVER (PARTITION BY
         faceta_origen_simple.origen_simple)) AS count
  FROM (
    SELECT DISTINCT th.rollo, th.idhoja, th.version, origen_simple
    FROM luz.textosporhoja th
    LEFT JOIN
      (SELECT *,
       UNNEST(STRING_TO_ARRAY(tipodoc, '|')) tipodoc_simple
       FROM lblme.imgjsondata) lth
    ON th.rollo = lth.rollo AND th.idhoja = lth.imagen
    LEFT JOIN (SELECT *,
               UNNEST(STRING_TO_ARRAY(origen, '|')) origen_simple
               FROM lblme.imgjsondata) loh
    ON th.rollo = loh.rollo AND th.idhoja = loh.imagen
    WHERE ($4 = FALSE OR th.hojalexemes @@
        websearch_to_tsquery('public.mh_simple', $3))
    AND ($6 = FALSE OR th.rollo IN ($5:csv))
    AND ($8 = FALSE OR
          CONCAT(th.rollo::text, ' - ', th.version::text, ' - ',
          th.original) IN ($7:csv))
    AND ($10 = FALSE OR tipodoc_simple IN ($9:csv))
    AND ($12 = FALSE OR origen_simple IN ($11:csv))
    ) AS faceta_origen_simple) fhr
ON fh.valor_faceta = fhr.value::text
WHERE fh.tipo_faceta = 'ORIGEN'
ORDER BY value;
```

Anexo 7 - Consulta para el volcado de datos, PostgreSQL

```
select h.rollo, h.idhoja, h.version, h.indice, h, calificacion, h.original,
h.tipo, h.textohoja, r.valor_faceta as faceta_rollo,
v.valor_faceta as faceta_version, lth.tipodoc_simple,loh.origen_simple
from luz.textosporhoja h
left join luz.facetashoja r
on r.count_rollo=h.rollo and r.tipo_faceta = 'ROLLO'
left join luz.facetashoja v
on v.valor_faceta=CONCAT(h.rollo::text, ' - ',
                         h.version::text, ' - ',
                         h.original) and v.tipo_faceta = 'VERSION'
left JOIN (SELECT *,
           tipodoc as tipodoc_simple
           FROM lblme.imgjsondata) lth
ON h.rollo = lth.rollo AND h.idhoja = lth.imagen
left JOIN (SELECT *,
               origen as origen_simple
               FROM lblme.imgjsondata) loh
ON h.rollo = loh.rollo AND h.idhoja = loh.imagen
```

Anexo 8 - Indexado en Elasticsearch

```
from elasticsearch import Elasticsearch, helpers

def index_many(os: Elasticsearch, os_index: str, documents, id_field: str):
   """
   Indexes many documents
   """
   actions = [
       {
           "_index": os_index,
           "_id": document[id_field],
           "_source": document,
       }
       for document in documents
   ]

   return helpers.bulk(os, actions, chunk_size=100, request_timeout=300)

es = Elasticsearch('localhost')
```

```
es.indices.create(index="luz")

df = pd.read_csv('datos_luz/select_h_rollo_h_idhoja_.csv')

df = df.fillna("")
df["origen"] = df.apply(lambda row: list(set(row['origen'].split('|'))) if
row['origen'] != "" else [], axis=1)
df["tipodoc"] = df.apply(lambda row: list(set(row['tipodoc'].split('|'))) if
row['tipodoc'] != "" else [], axis=1)
paginas = df.to_dict(orient='records')

index_many(es, 'luz', paginas, 'indice')
```

Anexo 9: Consultas en Elasticsearch

```
from elasticsearch import Elasticsearch
import time
import pandas as pd

from utils import time_wrapper

es = Elasticsearch("localhost")


def count_documents(index="luz"):
   start = time.time()
   result = es.count(index=index)
   end = time.time()
   return end - start, result["count"]


def get_query(query, rollo_array=None, faceta_array=None, tipodoc_array=None,
origen_array=None):
   query = {
       "bool": {
           "must": [
               {
                   "query_string": {
                       "query": query,
                       "fields": ["textohoja"],
                       "default_operator": "AND",
                   }
```

```
                }
            ],
            "filter": {"bool": {"must": []}},
        }
    }

    if rollo_array:
        query["bool"]["filter"]["bool"]["must"].append(
            {"terms": {"rollo": rollo_array}}
        )
    if faceta_array:
        query["bool"]["filter"]["bool"]["must"].append(
            {"terms": {"faceta_version": faceta_array}}
        )
    if tipodoc_array:
        query["bool"]["filter"]["bool"]["must"].append(
            {"terms": {"tipodoc.keyword": tipodoc_array}}
        )
    if origen_array:
        query["bool"]["filter"]["bool"]["must"].append(
            {"terms": {"origen.keyword": origen_array}}
        )

    return query


@time_wrapper
def search_documents_by_query_string_elastic(
    query,
    index="luz",
    limit=10000,
    rollo_array=None,
    faceta_array=None,
    tipodoc_array=None,
    origen_array=None,
    score=True,
    highlight=False,
):

    # Query con filtros
    query = get_query(
        query, rollo_array, faceta_array, tipodoc_array, origen_array
    )

    # Total count and results
```

```
body = {
    "size": limit,
    "track_total_hits": True,
    "_source": [
        "indice",
        "original",
        "rollo",
        "idhoja",
        "version",
        "tipo",
        "calificacion",
        "textohoja",
    ],
    "query": query,
}

if highlight:
    body["highlight"] = {"fields": {"textohoja": {}}}

# Total count por cada faceta_rollo
body_faceta_rollo = {
    "size": 0,
    "query": query,
    "aggs": {
        "faceta_rollo": {
            "terms": {"field": "faceta_rollo.keyword", "size": 10000},
        }
    },
}

# Total count por cada faceta_version
body_faceta_version = {
    "size": 0,
    "query": query,
    "aggs": {
        "faceta_version": {
            "terms": {"field": "faceta_version.keyword", "size": 10000},
        }
    },
}

# Total count por cada tipodoc
body_tipodoc = {
    "size": 0,
    "query": query,
```

```
    "aggs": {
        "tipo_doc": {
            "terms": {"field": "tipo_doc.keyword", "size": 10000},
        }
    },
}

# Total count por cada origen
body_origen = {
    "size": 0,
    "query": query,
    "aggs": {
        "origen": {
            "terms": {"field": "origen.keyword", "size": 10000},
        }
    },
}

start = time.time()
result = es.search(index=index, body=body)
result_faceta_rollo = es.search(index=index, body=body_faceta_rollo)
result_faceta_version = es.search(index=index, body=body_faceta_version)
result_tipodoc = es.search(index=index, body=body_tipodoc)
result_origen = es.search(index=index, body=body_origen)
end = time.time()

resultados = []
for hit in result["hits"]["hits"]:
    resultado = hit["_source"]
    resultado["score"] = hit["_score"]
    if highlight:
        resultado["highlights"] = hit["highlight"]["textohoja"]
    resultados.append(resultado)

result = {
    "total_time": end - start,
    "total_count": result["hits"]["total"]["value"],
    "count_rollo": pd.DataFrame(
        result_faceta_rollo["aggregations"]["faceta_rollo"]["buckets"]
    ).to_dict("records"),
    "count_version": pd.DataFrame(
        result_faceta_version["aggregations"]["faceta_version"]["buckets"]
    ).to_dict("records"),
    "count_tipodoc": pd.DataFrame(
        result_tipodoc["aggregations"]["tipo_doc"]["buckets"]
```

```
        ).to_dict("records"),
        "count_origen": pd.DataFrame(
            result_origen["aggregations"]["origen"]["buckets"]
        ).to_dict("records"),
        "resultados": resultados,
    }

    return result
```

REFERENCIAS BIBLIOGRÁFICAS

- Portal Udelar (2022). Proyecto Cruzar: “desde la interdisciplina aporta para «reconstruir la verdad”. Disponible en: https://udelar.edu.uy/portal/2022/05/proyecto-cruzar-desde-la-interdisciplina-aporta-para-reconstruir-la-verdad/
- Postgresql (2022a). Controlling Text Search, Chapter 9 Functions and Operators. Disponible en: https://www.postgresql.org/docs/current/functions-textsearch.html
- Postgresql (2022b). Controlling Text Search, Chapter 12 Full Text Search. Disponible en: https://www.postgresql.org/docs/current/textsearch-controls.html
- Hevodata (2022). Elasticsearch PostgreSQL Comparison: 6 Critical Differences. Disponible en: https://hevodata.com/learn/elasticsearch-postgresql/
- Patrick Royston (1982). An extension of Shapiro and Wilk's W test for normality to large samples. Applied Statistics, 31, 115–124. doi: 10.2307/2347973.
- Patrick Royston (1982). Algorithm AS 181: The W test for Normality. Applied Statistics, 31, 176–180. doi: 10.2307/2347986.
- Patrick Royston (1995). Remark AS R94: A remark on Algorithm AS 181: The W test for normality. Applied Statistics, 44, 547–551. doi: 10.2307/2986146.
- Myles Hollander and Douglas A. Wolfe (1973). Nonparametric Statistical Methods. New York: John Wiley & Sons. Pages 27–33 (one-sample), 68–75 (two-sample). Or second edition (1999).