Heurísticas para optimización de Aggregation Pipelines en MongoDB

Maximiliano Barragán
Facultad de Ingeniería, Universidad de la República
Montevideo, Uruguay
maximiliano.barragan@fing.edu.uy

Santiago Goycoechea
Facultad de Ingeniería, Universidad de la República
Montevideo, Uruguay
santiago.goycoechea@fing.edu.uy

***Resumen*—En este documento se muestra el resultado de un relevamiento y sugerencia de heurísticas para la optimización de Aggregation Pipelines en MongoDB, y se presentan los resultados de experimentos realizados para evaluarlas. Al final, se presentan las conclusiones sobre la evaluación de los experimentos, y que recomendaciones tiene sentido seguir al momento de optimizar este tipo de consultas.**

I. Introducción

A causa de la globalización de la tecnología y el surgimiento del Big Data, el desarrollo de consultas eficientes en bases de datos se ha vuelto una temática muy relevante para la industria del software. Cuando el conjunto de datos manejado es considerablemente grande, es importante que las interacciones con las bases de datos utilizadas se realicen de la forma más eficiente posible, ya que de lo contrario el sistema podría degradarse y resultar en una experiencia de usuario inaceptable. Mientras que para las bases de datos relacionales esta es un área muy estudiada, para bases de datos no relacionales parece no haber tanto camino recorrido. En este trabajo nos concentraremos en el estudio de un sistema de bases de datos no relacionales particular, MongoDB, y más concretamente, en la optimización de consultas realizadas mediante la utilización de Aggregation Pipelines.

El resto del documento se organiza de la siguiente forma. En la Sección II se presenta un resumen de los trabajos relacionados a este tema. Luego, en la Sección III discutimos el conjunto de datos que fue utilizado para modelar y cargar la base de datos con la cual fueron realizados los experimentos a lo largo del trabajo. En la Sección IV mostramos las heurísticas de optimización que analizaremos, y en la Sección V los resultados de los experimentos que realizamos sobre ellas. Finalmente en la Sección VI se presentan las conclusiones y se discute el trabajo a futuro.

Todo el código desarrollado se encuentra disponible en el repositorio de Gitlab de la Facultad de Ingeniería de la Universidad de la República referenciado[1].

II. Trabajos relacionados

Como se mencionó previamente, no existen muchos trabajos relacionados a este tema. Este proyecto se basa en el libro [Harrison and Harrison(2021)], concretamente en su séptimo capítulo, titulado *Tuning Aggregation Pipelines*. En dicho capítulo se discuten diferentes heurísticas a aplicar a la hora de implementar consultas de Aggregation Pipeline en MongoDB, que discutiremos y evaluaremos en las siguientes secciones.

III. Conjunto de datos utilizado

Para realizar los experimentos utilizamos el conjunto de datos del Catálogo Nacional de Datos Abiertos[2] llamado *Viajes realizados en los ómnibus del Sistema de Transporte Metropolitano - STM*[3], correspondiente a los viajes realizados en el mes de junio de 2021. Este dataset se distribuye en un archivo csv, donde cada fila se corresponde con un viaje realizado. Entre la información con la que se cuenta en cada fila para cada viaje tenemos, a modo de ejemplo, el id asociado al viaje, la empresa y línea de ómnibus en que se realizó, y de qué grupo de usuarios (por ejemplo: Estudiante A, Jubilado, etc.) era el pasajero. En el Anexo A se incluye una tabla con la descripción completa de los campos del conjunto de datos.

A partir de los datos, realizamos el diseño de la base separando en distintas colecciones, dependiendo de la semántica de cada columna. De esta manera, creamos una colección con la información de los viajes, otra con la información de las empresas, otra con la información específica de los grupos de usuarios, entre otras.

Para realizar diferentes comparaciones durante la experimentación, creamos tres bases de distintos tamaños. Dado que no contamos con tantas empresas, líneas ni grupos de usuarios distintos, y que el diferencial en el tamaño de la base al separar las colecciones de la manera en que lo hicimos se da por la cantidad de viajes que podían ser considerados en cuenta para la carga de la base, es que la variación de tamaño en las tres bases se da en la cantidad de documentos de la colección **viajes**. De esta forma, se definió que **viajes** sea construida considerando los siguientes subconjuntos:

- uno con las primeras 2.000.000 entradas del dataset
- otro con las primeras 500.000 entradas
- y el último con las primeras 50.000

A partir de estos tres subconjuntos cargamos tres bases de datos que utilizamos para realizar las mediciones de tiempos

[1] https://gitlab.fing.edu.uy/maximiliano.barragan/bdnr-proyecto

[2] https://catalogodatos.gub.uy/

[3] https://catalogodatos.gub.uy/dataset/intendencia-montevideo-viajes-realizados-en-los-omnibus-del-stm

Cuadro I: Colecciones de la base de datos

Colección	Documentos	Tamaño
viajes	2.000.000	564.4 MB
empresas	4	297 B
lineas	132	10.6 KB
tipos_viajes	14	1.0 KB
grupos_usuarios	8	737 B
grupos_usuarios_especificos	21	3.8 KB

para evaluar las heurísticas recopiladas. En el Cuadro I se pueden observar la cantidad de documentos y los tamaños de las respectivas colecciones para el subconjunto con las primeras 2.000.000 entradas.

IV. Heurísticas de optimización

A la hora de hablar de heurísticas de optimización, es importante diferenciar las heurísticas que son aplicadas automáticamente por el optimizador del sistema manejador de bases de datos de las heurísticas que podemos aplicar nosotros a la hora de implementar consultas. Las heurísticas del optimizador de MongoDB se describen en la página de su documentación que trata el tema de Aggregation Pipeline Optimization[4]. Al pensar heurísticas para aplicar nosotros, debemos excluir las que son aplicadas automáticamente por el optimizador de MongoDB, ya que no tendría sentido aplicarlas también nosotros. Un ejemplo de ellas es la heurística de proyectar únicamente los campos que son necesarios lo antes posible, lo cual resulta bastante intuitivo.

En el libro [Harrison and Harrison(2021)], se presentan algunas recomendaciones que deberían aplicarse a la hora de implementar consultas utilizando el Aggregation Pipeline de MongoDB. El objetivo de este trabajo es evaluar las que describimos a continuación:

IV-A. Filtrar de manera temprana y frecuente

Esta optimización es bastante intuitiva: al filtrar y reducir la cantidad de datos, el costo del procesamiento que se deberá llevar a cabo en el resto del pipeline será menor. Es por esto que se debe apuntar a filtrar lo antes posible en el pipeline. Esta recomendación aplica tanto a la agregación `$match` como a la agregación `$limit`.

IV-B. Utilizar índices en los foreignField de los $lookup

Las agregaciones `$lookup` (o joins) pueden llegar a ser muy costosas si no son implementadas correctamente. Esta heurística y la siguiente apuntan a evitar esto. Lo primero que se debe verificar a la hora de hacer un `$lookup` entre colecciones de tamaño considerable es que se tenga un índice en el campo que actúa como `foreignField` en el join, ya que esto mejora el desempeño del join considerablemente.

[4]https://docs.mongodb.com/manual/core/aggregation-pipeline-optimization/

IV-C. Hacer los $lookup desde la colección más pequeña

Cuando se desea hacer un join entre dos colecciones, en el caso en que ambas tengan índices en el campo sobre el que se quiere hacer el join o ninguna lo tenga, es más eficiente hacer el join desde la colección más pequeña a la más grande. Esto ocurre porque para calcular el resultado de un join se deben recorrer todos los documentos de la colección desde la cual se parte, y entonces utilizar una colección de tamaño menor resulta en que tengamos que recorrer menos documentos.

IV-D. Hacer los $sorts que utilizan un índice lo antes posible

De forma similar a como ocurre en las bases de datos relacionales, en MongoDB los índices pueden no aprovecharse si no se usan en una etapa temprana del pipeline. Además, al realizar sorts de tamaños considerables sin un índice, se puede llegar a agotar la memoria disponible y en ese caso el sistema recurre a copiar datos al disco duro, lo cual puede resultar en una consulta muy ineficiente. Por estas dos razones se recomienda realizar los sorts que utilicen un índice lo antes posible en el pipeline.

Otras ideas...

Durante la realización de los experimentos, dadas las distintas implementaciones de consultas y resultados observados, surgieron dos posibles heurísticas más sugeridas por nosotros, que también examinamos.

IV-E. Al momento de tener un arreglo con un solo elemento, ¿es preferible utilizar $unwind o $project para extraer el elemento y utilizarlo más adelante en el pipeline?

En muchos casos, luego de un `$lookup` obtenemos como resultado un arreglo con un solo elemento en el pipeline. Por medio de un `$unwind` podemos obtener un nuevo documento por cada uno de los elementos del arreglo, por lo que resulta una manera bastante intuitiva de obtener el elemento del arreglo y tenerlo disponible en nuestro pipeline. De la misma manera, podríamos utilizar `$project` en combinación con `$arrayElemAt`.

*IV-F. Al momento de hacer un $sort por fechas y tener disponible un índice, ¿es preferible que el tipo de la fecha sea **date** o **string**?*

Dado que podemos representar fechas con el tipo date al igual que con el tipo string, surge la incógnita de si al momento de utilizar la fecha como índice, sería preferible tenerla en la base con un tipo u otro, tanto desde el punto de vista del tiempo de ejecución al momento de resolver las consultas, así como del espacio utilizado por el índice.

V. Experimentación

Para cada una de las heurísticas planteamos por lo menos una consulta que la respete y una equivalente que no, para luego medir los tiempos de ejecución de las dos versiones. Ejecutamos cada consulta diez veces para evitar que alguna anomalía interfiera con los resultados. Además, ejecutamos cada una de las consultas en las tres bases de datos que

describimos en la Sección III, para evaluar si el tamaño de la base de datos puede tener alguna incidencia en la aplicación de estas recomendaciones. Los resultados obtenidos fueron muy similares para los tres tamaños de bases de datos, por lo que los datos que presentamos corresponden a la base de datos de 2.000.000 documentos en la colección **viajes**.

V-A. Filtrar de manera temprana y frecuente

En el libro [Harrison and Harrison(2021)] se presenta un ejemplo para esta heurística utilizando un `$limit` en un caso de forma temprana y, en otro, tarde. Como lo sugiere la heurística, la implementación que tiene el `$limit` antes funciona más rápido. Con el fin de evaluar esta heurística pero aplicándola de forma diferente, buscamos implementar una consulta que filtre utilizando un `$match`. Ideamos entonces una consulta que encuentre todos los viajes que tengan más de un tramo (esto ocurre para los viajes cuyo id aparece más de una vez en la colección). La implementación que hicimos siguiendo la heurística se puede ver en el Listado 1.

Listado 1 Viajes con más de un tramo

```
[{
   "$group": {
      "_id": "$id_viaje",
      "count": {
         "$sum": 1
      },
      "codigos_lineas": {
         "$push": "$linea_codigo"
      }
   }
},
{
   "$match": {
      "count": {
         "$gt": 1
      }
   }
},
{
   "$lookup": {
      "from": "lineas",
      "localField": "codigos_lineas",
      "foreignField": "linea_codigo",
      "as": "lineas"
   }
},
{
   "$project": {
      "count": "$count",
      "lineas": "$lineas.dsc_linea"
   }
}]
```

Además hicimos otra implementación que no respete la heurística, moviendo el `$match` al final del pipeline. Para nuestra sorpresa, ambas implementaciones tuvieron tiempos de ejecución muy similares, como se puede ver en la Figura 1. Observando la salida del comando *explain* de MongoDB, vimos que ambas implementaciones terminan siendo iguales tras pasar por el optimizador automático, ya que en el caso en que el `$match` está al final, este se mueve automáticamente a donde se encuentra en la otra versión de la consulta. De esto dedujimos que MongoDB aplica más optimizaciones de las que se listan en la documentación que nombramos en la Sección IV, ya que en dicha página nunca se habla de una optimización que permita mover un `$match` antes de un `$lookup`, como ocurre en este caso. De todas formas, aunque en este caso la heurística no suponga una mejora en el desempeño de la consulta, creemos que siempre vale la pena aplicarla porque en otros casos (como el presentado en el capítulo de referencia) sí funciona, y la aplicación de la misma nunca debería causar un desempeño sustancialmente peor.

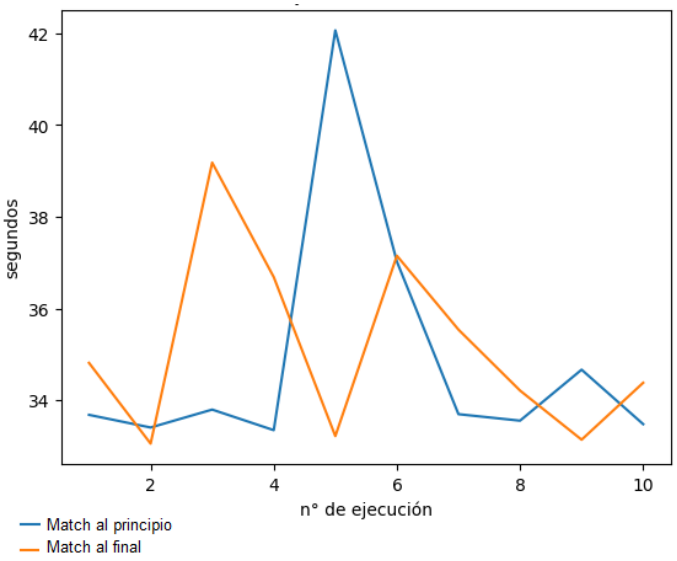

Figura 1: Tiempos de ejecución - heurística de filtrar temprano

V-B. Utilizar índices en los foreignField de los $lookup

Con el fin de evaluar esta heurística implementamos el pipeline del Listado 2, que encuentra las diez líneas con más viajes, partiendo desde la colección `lineas`.

Listado 2 Líneas con más viajes

```
[{
   "$lookup": {
      "from": "viajes",
      "localField": "linea_codigo",
      "foreignField": "linea_codigo",
      "as": "viajes"
   }
},
{
   "$project": {
      "linea_codigo": "$linea_codigo",
      "dsc_linea": "$dsc_linea",
      "num_viajes": {
         "$size": "$viajes"
      }
   }
},
{
   "$sort": {
      "num_viajes": -1
   }
},
{
   "$limit": 10
}]
```

Como esperábamos, el uso de un índice en el campo que actúa como `foreignField`, en este caso `linea_codigo`, mejora enormemente el desempeño de la consulta, como se puede ver en la Figura 2.

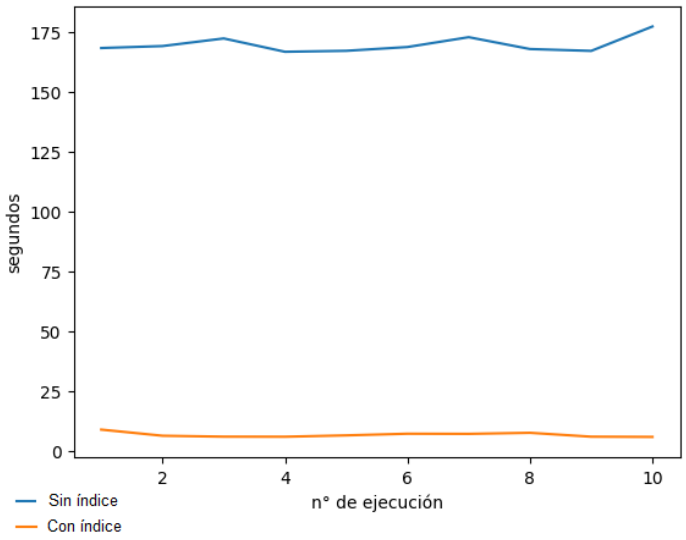

Figura 2: Tiempos de ejecución - heurística de índice en el join

V-C. Hacer los $lookup desde la colección más pequeña

Para evaluar esta heurística consideramos una consulta que encuentre el código de las paradas donde más estudiantes de tipo A asciendan a un ómnibus. Implementamos dicha consulta de dos formas, la primera partiendo desde la colección más grande, **viajes**, mostrada en el Listado 3.

Listado 3 Paradas donde más estudiantes tipo A suben - desde viajes

```
[{
   "$lookup": {
      "from": "grupos_usuarios_especificos",
      "localField": "grupo_usuario_especifico",
      "foreignField": "grupo_usuario_especifico",
      "as": "grupo_usuario_especifico"
   }
},
{
   "$project": {
      "id_viaje": "$id_viaje",
      "ordinal_de_tramo": "$ordinal_de_tramo",
      "codigo_parada_origen": "$codigo_parada_origen",
      "grupo_usuario_especifico": {
         "$arrayElemAt": [
            "$grupo_usuario_especifico.
               descripcion_grupo_usuario_espe",
            0
         ]
      }
   }
},
{
   "$match": {
      "grupo_usuario_especifico": "ESTUDIANTE A"
   }
},
{
   "$group": {
      "_id": "$codigo_parada_origen",
      "count": {
         "$sum": 1
      }
   }
},
{
   "$sort": {
      "count": -1
   }
},
{
   "$limit": 10
}
]
```

La segunda implementación parte de la colección más pequeña, que es **grupos_usuarios_especificos**, y se puede ver en el Listado 4.

Listado 4 Paradas donde más estudiantes tipo A suben - desde grupos_usuarios_especificos

```
[{
   "$lookup": {
      "from": "viajes",
      "localField": "grupo_usuario_especifico",
      "foreignField": "grupo_usuario_especifico",
      "as": "viajes"
   }
},
{
   "$unwind": {
      "path": "$viajes"
   }
},
{
   "$match": {
      "descripcion_grupo_usuario_espe": "ESTUDIANTE A"
   }
},
{
   "$group": {
      "_id": "$viajes.codigo_parada_origen",
      "count": {
         "$sum": 1
      }
   }
},
{
   "$sort": {
      "count": -1
   }
},
{
   "$limit": 10
}
]
```

Ambas consultas son similares, pero una de ellas utiliza un `$project` mientras que la otra usa un `$unwind`. Para asegurarnos de que esto no causa diferencias en los tiempos de ejecución, evaluamos otro par de consultas donde la única diferencia era de este estilo. Haciéndolo confirmamos que los tiempos de ejecución son muy parecidos para los tres tamaños de bases de datos, por lo que la diferencia entre usar `$unwind` o `$project` no supone un problema al evaluar la heurística, ya que hay una diferencia mucho mayor de tiempos. Los resultados de esta comparación se pueden ver en la subsección V-E.

Como se puede ver en la Figura 3, partir desde la colección más pequeña resulta en un desempeño mucho mejor, como lo indica la heurística.

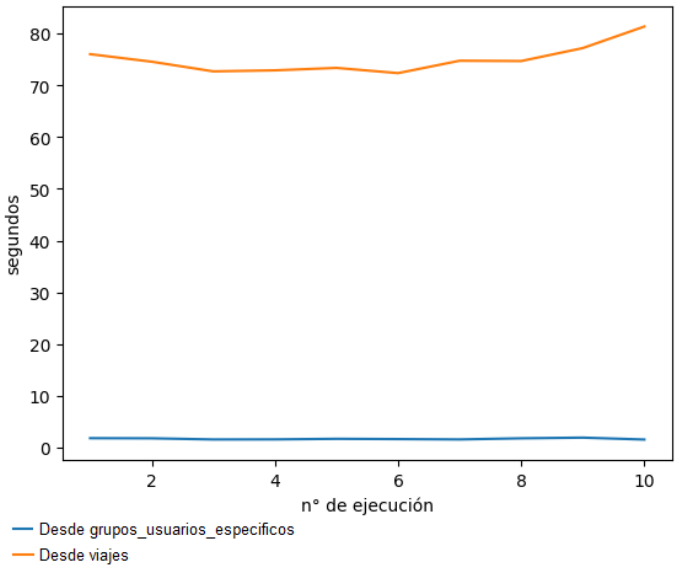

Figura 3: Tiempos de ejecución - heurística de join desde la colección pequeña

V-D. Hacer los $sorts que utilizan un índice lo antes posible

En este caso utilizamos una consulta que encuentre las últimas cinco líneas que hayan realizado un viaje, permitiendo repetidos. En las implementaciones utilizamos un `$sort` sobre el campo `fecha_evento` de la colección **viajes**, para el cual creamos un índice. En el Listado 5 mostramos la implementación de dicha consulta que tiene en cuenta la heurística. La otra implementación a considerar, que no aplica la heurística, es igual pero el `$sort` se ejecuta inmediatamente antes del `$limit`, en vez de al principio.

Listado 5 Últimas líneas con viajes

```
[{
   "$sort": {
      "fecha_evento": -1
   }
},
{
   "$lookup": {
      "from": "lineas",
      "localField": "linea_codigo",
      "foreignField": "linea_codigo",
      "as": "linea"
   }
},
{
   "$project": {
      "_id": 0,
      "linea": {
         "$arrayElemAt": [
            "$linea.dsc_linea",
            0
         ]
      },
      "fecha_evento": "$fecha_evento"
   }
},
{
   "$limit": 5
}
]
```

Esta heurística también resultó efectiva: como se puede ver en la Figura 4 los tiempos son muchos menores para la implementación que ordena al principio.

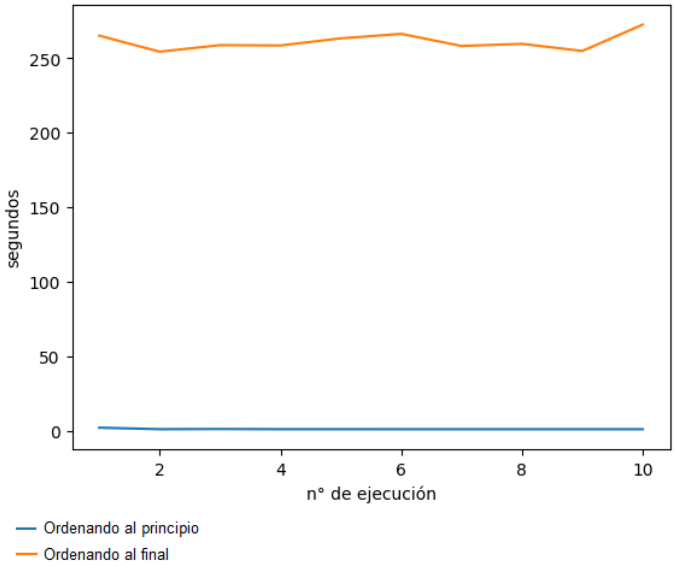

Figura 4: Tiempos de ejecución - heurística de sort al principio

V-E. Comparación entre $unwind y $project

Para evaluar esto implementamos también dos versiones de la misma consulta. En el Listado 6 se puede ver la implementación que utiliza el `$unwind`, mientras que en el Listado 7 podemos ver qué se debe sustituir en esta implementación para obtener la otra versión (simplemente se cambia el `$unwind` por el `$project`).

Listado 6 Cantidad total de viajes realizados por cada grupo de usuario - `$unwind`

```
[
{
   '$lookup': {
      'from': 'grupos_usuarios_especificos',
      'localField': 'grupo_usuario_especifico',
      'foreignField': 'grupo_usuario_especifico',
      'as': 'usuarios'
   }
}, {
   '$unwind': {
      'path': '$usuarios'
   }
}, {
   '$group': {
      '_id': {
         'usuario': '$usuarios.descripcion_grupo_usuario_espe'
      },
      'total_viajes': {
         '$sum': 1
      }
   }
}, {
   '$project': {
      '_id': 0,
      'usuario': '$_id.usuario',
      'total_viajes': '$total_viajes'
   }
}
]
```

Listado 7 Cantidad total de viajes realizados por cada grupo de usuario - `$project`

```
[
...
{
   '$project': {
      '_id': 0,
      'usuarios': {
         '$arrayElemAt': [
            '$usuarios', 0
         ]
      }
   }
},
...
]
```

Además, dado que de la salida del `$unwind` se obtienen todos los campos del documento entrante en la salida, además del elemento que sale del arreglo, implementamos otra versión del `$project` (que llamaremos "Project completo") donde en el `$project` también se agregan el resto de los campos, para que ambos tengan la misma salida. Como se puede ver en la Figura 5 (se muestra una gráfica ampliada, dado que hay un dato anómalo en uno de los tiempos de ejecución para el "Project completo", que a escala normal no permite visualizar las diferencias en los tiempos), vemos que en las 10 ejecuciones, la consulta con `$unwind` se comporta mejor que las versiones con `$project`. Otra ventaja además del tiempo de ejecución es que, como se puede ver en cada uno de los pipelines, utilizar el `$unwind` resulta en una implementación más simple, dado que simplemente tenemos que especificar el *path* correcto. A pesar de esto, aunque la diferencia pueda resultar significativa, utilizar `$project` puede resultar más intuitivo, además de que podemos aprovechar y en este `$project` filtrar por los campos que nos interesan en el resto del pipeline, en lugar de hacer `$unwind` y tener que proyectar en un paso posterior.

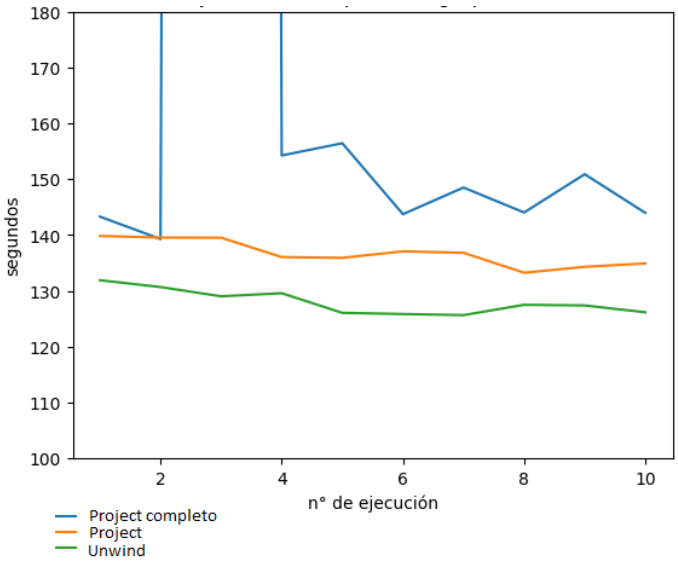

Figura 5: Tiempos de ejecución - `$unwind` vs `$project`

V-F. Comparación entre date y string como índices para fechas

Para evaluar esto utilizamos la misma consulta que se puede ver en el Listado 5, pero creamos dos réplicas de la colección **viajes**: una con `fecha_evento` como string, y otra con `fecha_evento` como date, y creamos índices en ambos casos. De las ejecuciones realizadas, podemos ver dos cosas interesantes. Para el caso de la base de mayor tamaño, con 2.000.000 documentos, vemos que el índice del tipo **date** en general funciona mejor que el índice de tipo **string**. Esto se puede ver en la Figura 6.

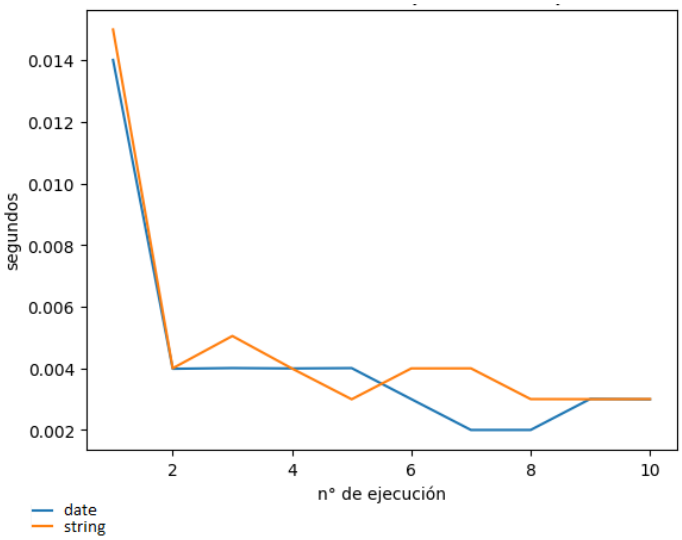

Figura 6: Tiempos de ejecución - índice date vs string 2.000.000 de documentos

En cambio, para las bases de menor tamaño, tanto la de 50.000 (Figura 7) como la de 500.000 (Figura 8) documentos, se puede observar que, contrario a lo que uno podría intuir y a lo que pasa en la base de mayor tamaño, se obtiene un mejor desempeño con el índice de tipo **string**.

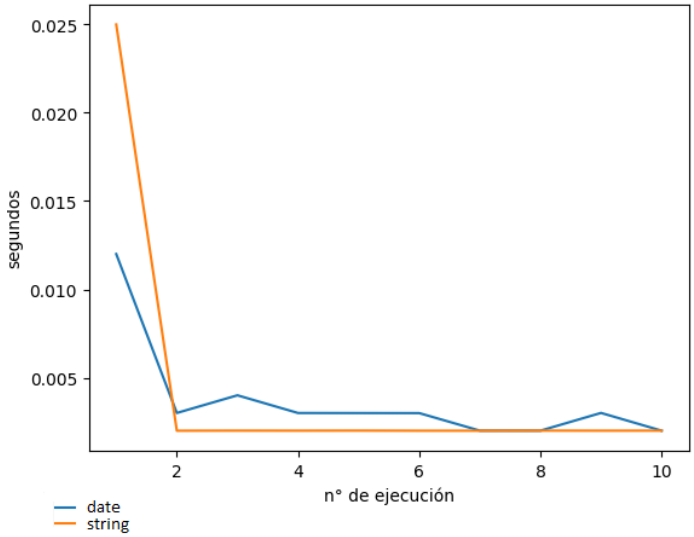

Figura 7: Tiempos de ejecución - índice date vs string 50.000 documentos

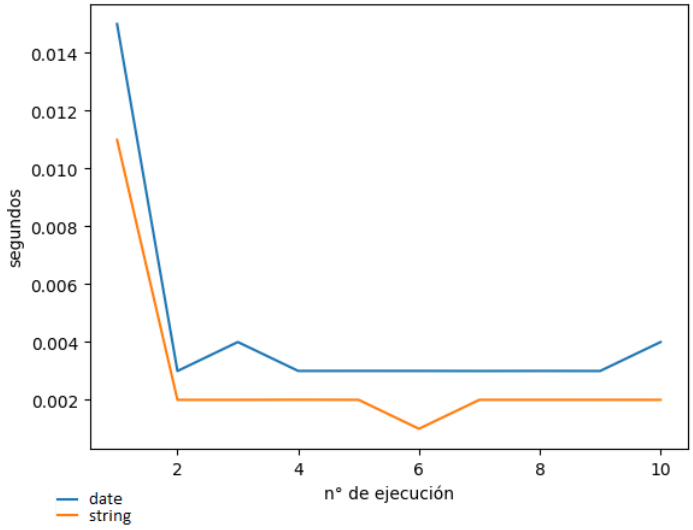

Figura 8: Tiempos de ejecución - índice date vs string 500.000 documentos

Por otro lado, más allá de los tiempos de ejecución de una opción u otra, resulta bastante interesante (e importante) observar es el tamaño que ocupan los dos tipos de índices. En este caso, en la Figura 9 vemos que el índice de tipo string ocupa prácticamente el doble de tamaño que el índice de tipo date, lo que en algunos casos puede resultar tan determinante o más, que los tiempos de ejecución.

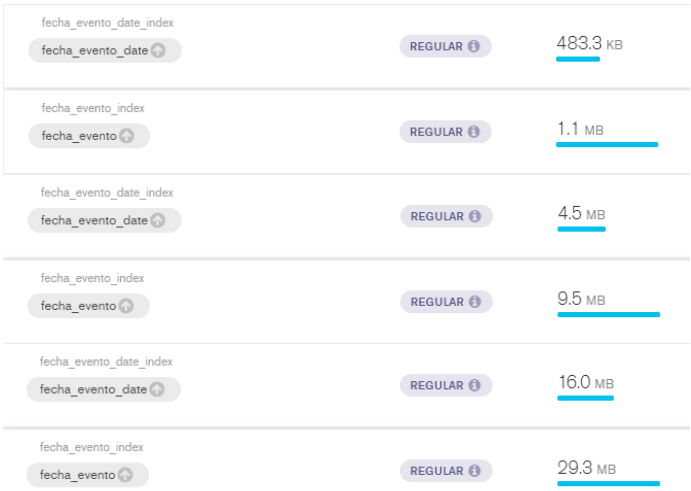

Figura 9: Tamaño que ocupa cada tipo de índice para las colecciones de 50.000 (arriba), 500.000 (medio) y 2.000.000 (abajo) de documentos

VI. Conclusiones y trabajo futuro

Relevamos y evaluamos cuatro heurísticas para la optimización de Aggregation Pipelines en MongoDB. Implementando consultas para poner a prueba la efectividad de las heurísticas, comprobamos que la heurística de *"filtrar seguido y de forma temprana"* no siempre supone una mejora, ya que a veces es aplicada por el optimizador automático de MongoDB (y descubrimos que el mismo a veces aplica heurísticas que no están detalladas en su documentación), mientras que las tres restantes relevadas del libro **sí** resultan en desempeños considerablemente mejores en nuestros experimentos.

Por otro lado, evaluamos dos heurísticas propuestas por nosotros. En la primera de ellas observamos que en caso de tener un arreglo de un solo elemento, al menos en nuestro caso particular, podemos obtener un mejor desempeño en la consulta al obtener el elemento mediante un `$unwind`, respecto a obtenerlo utilizando `$project`. A pesar de esto, cabe destacar que al utilizar `$project` podemos también filtrar otros campos del pipeline, que puede resultarnos beneficioso.

En cuanto a la comparación realizada respecto a utilizar índices de fecha de tipo string o tipo date, podemos concluir que para bases con muchos documentos, parece tener un mejor desempeño utilizar índices de tipo date. Esto sumado al hecho de que cuantos más documentos tenemos, más espacio van a ocupar nuestros índices, y a que los índices de tipo date ocupan alrededor de la mitad de espacio que los de tipo string, decantarse por índices de tipo date parece ser la mejor opción. En cambio, para colecciones con menor cantidad de documentos, a pesar de ocupar más espacio, los índices de tipo string parecen tener mejores tiempos de ejecución.

En el libro [Harrison and Harrison(2021)] se realizan algunas otras recomendaciones que no evaluamos ya que consideramos que son de menor importancia o aplicabilidad. Como trabajo futuro sería interesante evaluar estas otras heurísticas, y además realizar más pruebas sobre las heurísticas que ya

evaluamos y las posibles combinaciones entre ellas, así como explorar la interacción de estas recomendaciones con otros pasos distintos del pipeline.

REFERENCIAS

[Harrison and Harrison(2021)] Guy Harrison and Michael Harrison. *MongoDB Performance Tuning*. Apress, 2021.

Anexo A - Campos del conjunto de datos

id_viaje	Identifica de forma única a un viaje dentro del mes. Se define un viaje como todos los tramos realizados por el/los pasajeros con un único pago
con_tarjeta	0 = viajes sin tarjeta, 1 = viajes con tarjeta
fecha_evento	Fecha y hora en la cual se registra el tramo del viaje
tipo_viaje	Código del tipo de viaje
descripcion_tipo_viaje	Descripción del tipo de viaje (por ejemplo '1 hora')
grupo_usuario	Código del grupo de usuario
descripcion_grupo_usuario	Descripción del grupo de usuario (por ejemplo 'estudiante')
grupo_usuario_especifico	Código del grupo de usuario específico (subgrupo dentro del grupo de usuario)
descrip_grupo_usuario_espe	Descripción del grupo de usuario específico (por ejemplo 'estudiante A')
ordinal_de_tramo	El ordinal del tramo dentro del viaje (para viajes con tarjeta)
cantidad_pasajeros	Cantidad de personas que realizan el tramo
codigo_parada_origen	Código de la parada de ascenso
cod_empresa	Código de la empresa a la que pertenece el ómnibus
descrip_empresa	Descripción de la empresa a la que pertenece el ómnibus
linea_codigo	Código de la línea
dsc_linea	Nombre público de la línea (por ejemplo '185')
sevar_codigo	Código de la variante dentro de la línea