75.74 - Sistemas Distribuidos I

TP1: Escalabilidad

Grupo 08

2° Cuatrimestre 2023

Nombre y Apellido	Mail	Padrón
Prada, Joaquín	jprada@fi.uba.ar	105978
Sotelo Guerreño, Lucas Nahuel	lsotelo@fi.uba.ar	102730

1. Introducción

Se creó un sistema distribuído para analizar 6 meses de registros de precios de vuelos para proponer mejoras en la oferta a clientes.

2. Arquitectura

Objetivo de la arquitectura

El objetivo de la arquitectura es poder paralelizar lo máximo posible el filtrado y procesado de cada entrada de vuelos. Es importante que solo se lea cada entrada una única vez, ya que al ser muchos vuelos sería muy ineficiente tener que volver a leerlas. La arquitectura apunta a ser escalable, en donde se necesite más cómputo se puede escalar y acelerar su procesado.

La cantidad de datos a procesar provienen de un archivo de aproximadamente 31GB.

Flujo de la arquitectura

La arquitectura está dividida en 3 principales partes, el cliente, encargado de la subida de los archivos y recepción de resultados. El servidor, encargado de recibir los mensajes del cliente para empezar el procesamiento, como también el envío de resultados al cliente. Y por último las entidades encargadas de la modificación y procesamiento de los mensajes.

A continuación vamos a contar como es el flujo de estas partes y como en conjunto forman la arquitectura.

Cliente

El flujo de la aplicación comienza en el cliente, quien empieza por el envío de los archivos necesarios.

El cliente envía dos archivos por el mismo socket de conexión con el servidor: el dataset de aeropuertos y el dataset de vuelos. La forma en que el servidor puede diferenciarlos es mediante el primer byte del mensaje enviado por el cliente que indica de qué dataset proviene: 1 si es el dataset de aeropuertos y 2 si es el dataset de vuelos. De esta manera el servidor lee el primer byte de cada mensaje que recibe del cliente y ya puede saber cómo interpretarlo.
Cada mensaje se delimita por el siguiente string: “\r\n\r\n” y, además, pueden contener varias líneas del archivo que se delimitan por el string: “\n”.
Ejemplos:

- “1EZE,102,234\nAEP,112,221\n\r\n\r\n”
- “2vuelo1,1h30m,EZE\n\r\n\r\n”

Para indicar que se terminó de enviar el archivo completo, el cliente envía el string: “\0” (también precedido por el byte que determina el tipo de dataset).

El cliente puede envíar sus mensajes en “batches” significando enviar múltiples mensajes juntos. Esto se logra simplemente separando los mensajes por el “\n” como se explicó anteriormente, al enviar los mensajes con batches se logra un aumento muy notable de performance.

Por último, el cliente recibe los resultados y los separa en sus archivos correspondientes, creados a partir del “tag” recibido, este tag hace referencia al resultado de la “query” correspondiente.

Server

Mientras que el cliente envía sus mensajes, es el servidor el encargado de recibirlos y empezar a distribuirlos por el sistema.

Cuando recibe una conexión nueva, la empieza a manejar y escuchar sus mensajes. Cuando recibe un nuevo mensaje, lo primero que hace es ver de qué tipo se trata, como explicamos previamente, este puede ser del dataset de aeropuertos o de vuelos. Luego de averiguar el tipo se lo pasa a su respectivo *uploader*, que sabe a qué entidad pasarlo.

Por otro lado, el servidor tiene otra entidad en un proceso independiente, para el recibo y envío de los resultados hacia el cliente.
El servidor le envía un mensaje el cual es precedido por un “tag” entre corchetes, por ejemplo: “[DISTANCIA]”, que indica de qué tipo de resultado se trata. La forma de delimitar los mensajes es la misma que en la comunicación cliente -> servidor.

Processors/Utils

Por último en la arquitectura, están las entidades encargadas de la modificación y procesamiento de la información/mensajes, su forma es muy similar entre sí, la diferencia que tienen es **cómo** modifican los datos. Es por eso que acá se explica cómo se componen y cómo se comunican entre sí, que son los 2 puntos más fundamentales en la arquitectura. Sus detalles de procesamiento de los datos se encuentran explicados en los diagramas de cada caso de uso.

Comencemos primero por cómo se componen estas entidades. Hay 3 principales partes dentro de cada entidad.
Donde en primer lugar tenemos la inicialización de su configuración, que gracias a la similitud que tienen entre sí, utilizamos variables de entorno para definirlas. Estas son procesadas con un “config parser” y pasadas a las distintas abstracciones. Después inicializan su logger con su determinado nivel.
Luego está la abstracción de **Communication** donde se abstrae todo sobre cómo se comunican entre distintas entidades.

Y por último, está el procesador en sí, donde está contenida las reglas de negocio de cada entidad en particular.

Ahora, ¿cómo hacen estas entidades para comunicarse entre sí? Para eso tenemos la abstracción de **Communication**, es acá donde se esconde el corazón de la arquitectura. Hay 2 tipos de comunicación, el **Receiver**, encargado de recibir mensajes entrantes, y el **Sender**, encargado del envío de mensajes. Una entidad puede tener tantos "receiver" como "sender" desee, creados utilizando el **Communication Initializer** utilizado en la inicialización (main).
Las entidades tienen que saber también qué tipo de entrada y salida tienen, esto significa que estos tipo puede ser una cola o un exchange, es por eso que los sender y "receiver" son de tipo **Queue**, o **Exchange**.

Pasemos entonces a explicar como funcionan estos **Communication**.
Empecemos con el receiver, lo que va a hacer un receiver, en su forma más básica, es leer de un "input", puede ser el nombre de una cola o un exchange (lee un predeterminado número de "prefetch" para ayudar con el *throughput)*, y cuando recibe un mensaje nuevo se lo pasa a un "callback" para que haga lo que quiera con el mensaje. Este callback, junto al eof callback tienen que indicarse al llamar a la función **bind** luego de la creación del "receiver". El mensaje que se le pasa al callback ya está "parseado", significando que recibe una adaptación del mensaje en forma de diccionario, ocultando como es el formato de envío. Es necesario igual indicarle que es lo que se está esperando con una lista de campos cuando se hace el bind.

Por otro lado, el sender, encargado de enviar los mensajes, tiene 2 funciones importantes. Primero el **send_all** utilizado para mandar una lista de mensajes aprovechando el "batching" que llega desde el cliente. Dependiendo de qué tipo es a quien le estamos enviando, puede que sea necesario indicarle una **routing key** para saber a quien enviarle.
Luego está el **send_eof** que envía la notificación de que se terminó de mandar todo, y esto es muy importante para el dominio del problema, ya que hay entidades que solo pueden actuar cuando saben que ya recibieron todo. Pasemos a explicar como funciona el protocolo de "End of File" dentro de los communications.

Una entidad llama al send_eof donde se envía un 0 + el número de mensajes enviados, cuando lo recibe un receiver inicia el protocolo, lo que se tiene que lograr es asegurarse que todos los mensajes hayan sido recibidos antes de llamar al callback del eof, puesto en el bind como se explicó anteriormente. Esto es porque pueden haber "n" réplicas escuchando cómo "workers" de la misma cola, y hasta que no estén todas sincronizadas no se avanza.
Entonces, lo primero que hace la entidad cuando recibe el primer eof es modificarlo e iniciar el requeue, para que lo agarre otra réplica. Este nuevo mensaje debe contener los mensajes enviados hasta ahora y cuantos faltan por recibir, como también un TLL indicando cuántas réplicas faltan por sincronizarse. Una vez hecho el "requeue" la réplica deja de recibir mensajes, para que los mensajes restantes y EOF reencolados sean recibidos únicamente por las próximas réplicas.

Es así como se va propagando por todas las réplicas hasta que la última se fija si faltan mensajes aún y se queda esperando, o se puede llamar ya al eof callback para continuar con la ejecución.
Previamente a llamar al callback, opcionalmente, si se le pasó el sender en el bind, la réplica actualiza el valor del sender de mensajes enviados al total de mensajes enviados por todas las réplicas, recibido en el eof. De esta manera cuando se llame al send_eof se va a mandar con el número total de mensajes enviados por todas las réplicas.

Formato de los EOF

```
Protocol first EOF:
        0     1               9
        | EOF | messages_sent |

Protocol requeued EOF:
      0       1     5           13     21                    29
        | EOF | TTL | remaining | sent | messages_sent_sender |
```

Ejemplo de algoritmo EOF

Paso 1: Una de las réplicas recibe el primer EOF, que contiene cuántos mensajes se enviaron desde la entidad previa.

Paso 2: La réplica reencola el EOF con el TTL, mensajes restantes y el número de mensajes enviados, la siguiente réplica agarra el EOF reencolado. La réplica se desconecta para evitar recibir nuevos mensajes.

Paso 3: La siguiente réplica reencola nuevamente el EOF sumado a sus mensajes recibidos y enviados y se apaga.

Paso 4: La última réplica recibe el EOF pero como los mensajes restantes junto a sus recibidos no dan 0 significa que quedan mensajes por llegar, es por eso que vuelve a encolar el EOF que recibió sin modificar ningún dato.

Paso 5: La réplica recibe los 10 mensajes que faltaban y los procesa.

Paso 6: Al recibir el EOF y esta vez si da 0 los restantes, la réplica llama al **eof_callback**.

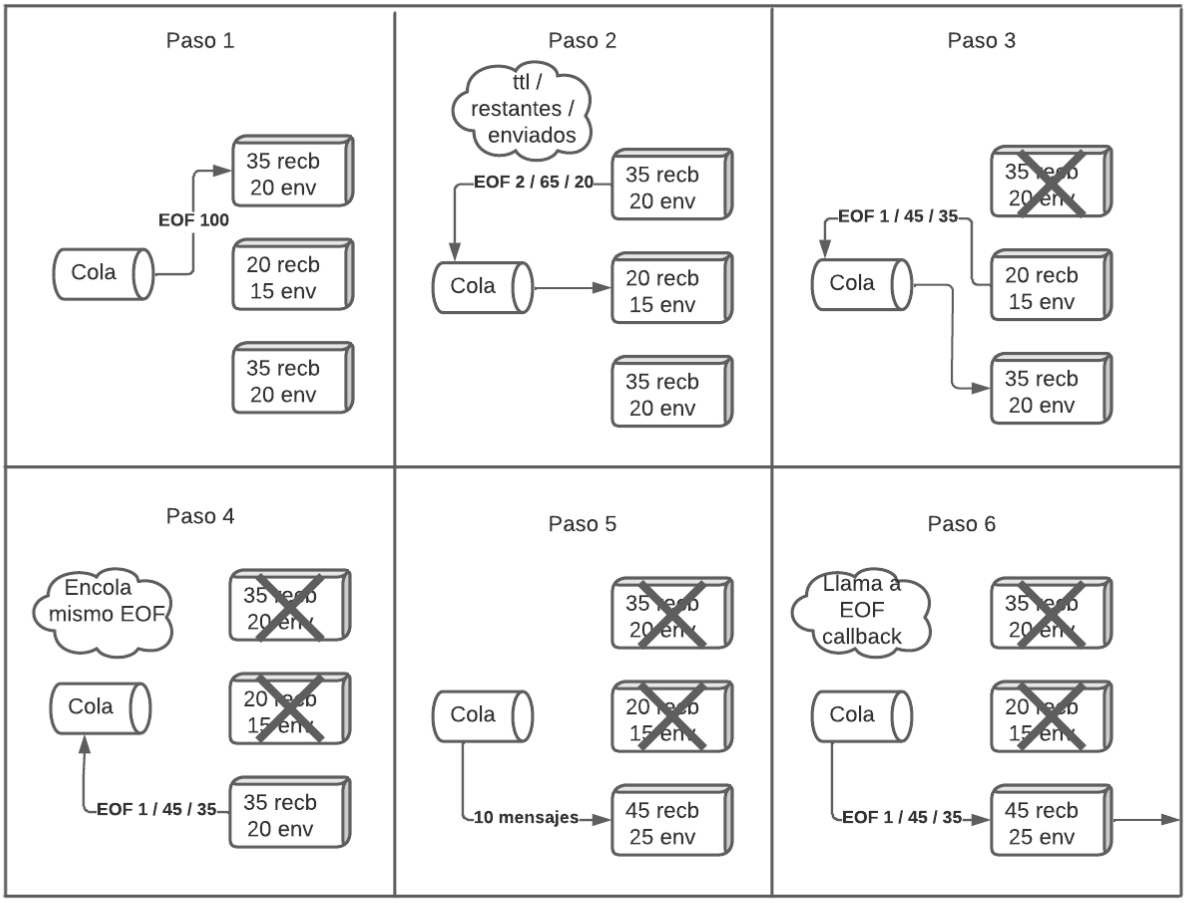

Figura 2.1 - Ejemplo del algoritmo de EOF

Ahora, ¿quien utiliza estas **communications**? Cada entidad como vimos tiene su **Processor** encargado de hacer la regla de negocio / procesamiento sobre los datos, son estos los que utilizan los *communications* para la comunicación entre entidades, como también para organizarse. Es por eso que los callbacks se “bindean” con funciones del “processor”. Si necesitamos que primero se escuche de un receiver y luego de otro, simplemente se puede hacer que escuchar del segundo sea el **eof_callback** del primer receiver, y por último el segundo eof callback podría ser el **send_eof** para continuar con el flujo.

Escalabilidad

La mayoría de las entidades anteriormente mencionadas fueron diseñadas para ser escaladas horizontalmente[1], multiplicando la cantidad de contenedores de la misma.

Esto se puede realizar a través de un archivo .env en donde se indica la cantidad de réplicas a levantar por cada entidad.

[1] Referirse al diagrama de robustez para identificar las que no son escalables horizontalmente.

3. Escenarios

Queremos cubrir las siguientes consultas:

1. ID, trayecto, precio y escalas de vuelos de 3 escalas o más.
2. ID, trayecto y distancia total de vuelos cuya distancia total sea mayor a cuatro veces la distancia directa entre puntos origen-destino.
3. ID, trayecto, escalas y duración de los 2 vuelos más rápidos para todo trayecto con algún vuelo de 3 escalas o más.
4. El precio avg y max por trayecto de los vuelos con precio mayor a la media general de precios.

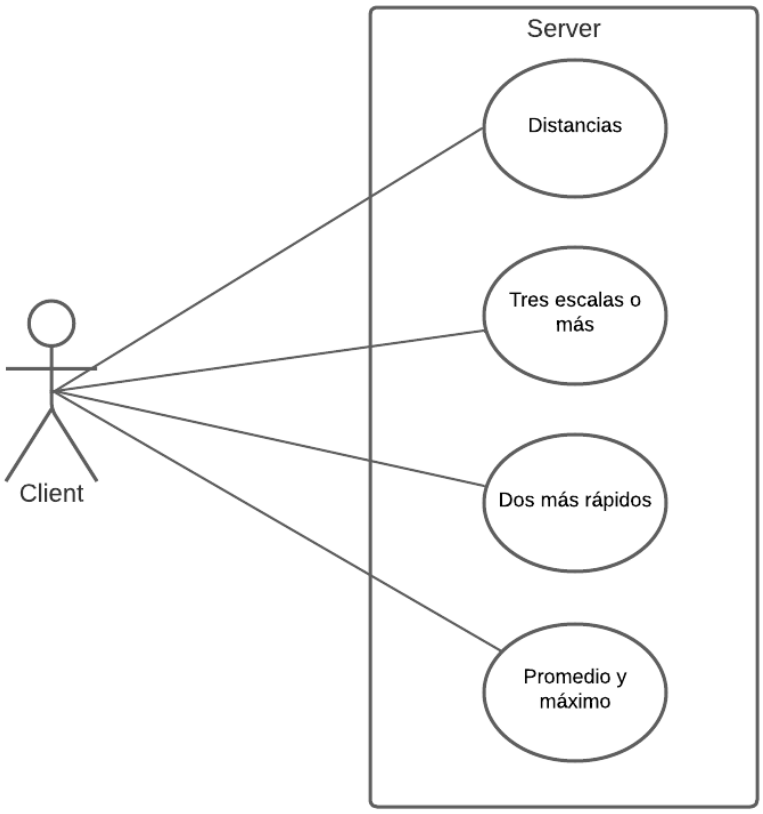

Figura 3.1 - Diagrama de casos de usos

4. Desarrollo

A continuación se detalla el flujo de resolución de cada una de las consultas a resolver y las entidades que lo conforman.

Query 1: Tres escalas o más

Vista Lógica

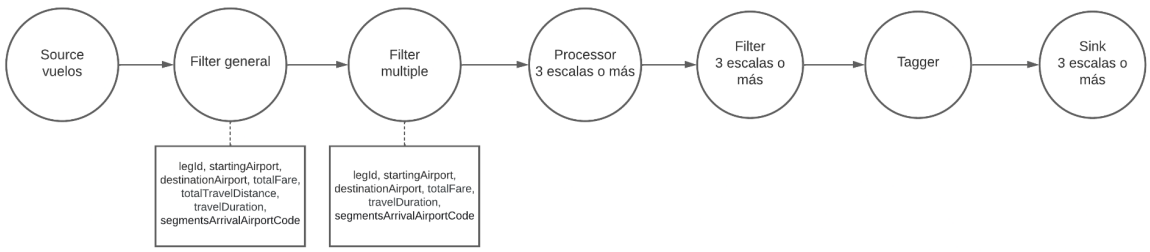

Figura 4.1 - DAG de la consulta de vuelos con tres escalas o más

En esta consulta los vuelos primero pasan por dos **Filters** para que el **Processor** tenga solamente la cantidad de columnas necesarias para el procesamiento (son dos **Filters** porque el primero es compartido por otras consultas). Una vez que el **Processor** recibe un vuelo lo que hace es verificar si el mismo realiza tres escalas o más, en caso que así sea lo entrega a la siguiente entidad, en caso contrario lo descarta.

La siguiente entidad es un **Filter** que elimina las columnas innecesarias para la respuesta y envía el vuelo al **Tagger**. Finalmente, el **Tagger** le agrega el tag correspondiente al vuelo y lo envía al **Sink**.

Vista de Proceso

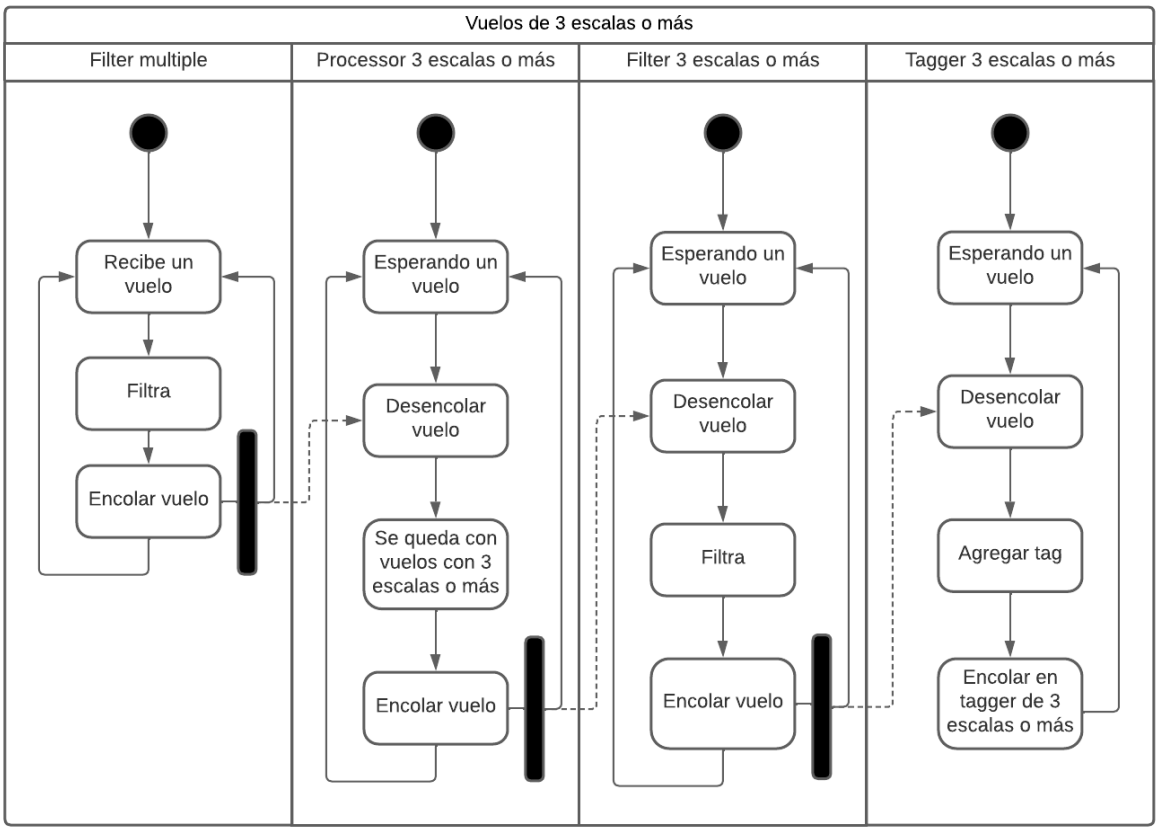

Figura 4.2 - Diagrama de actividad de la consulta de vuelos con tres escalas o más

Como se puede apreciar en el diagrama de actividad de la Figura 4.2, el flujo es bastante sencillo, simplemente van pasándose los vuelos y cada uno realiza un acción sobre el mismo.

Query 2: Distancias

Vista Lógica

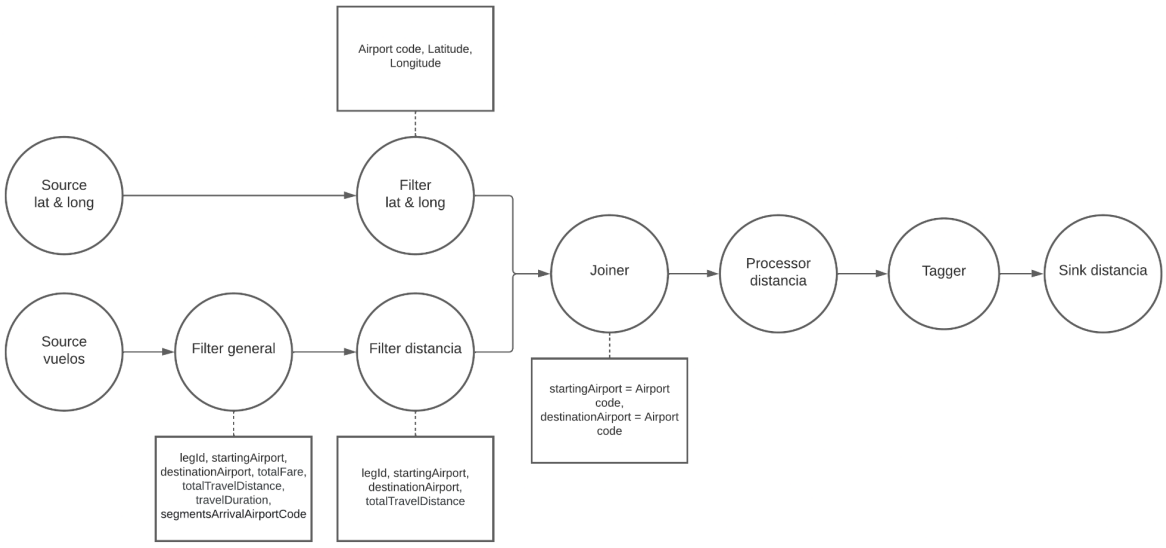

Figura 4.3 - DAG de la consulta de vuelos cuya distancia total sea mayor a cuatro veces la distancia directa entre puntos origen-destino

Para realizar esta query necesitamos la latitud y longitud de cada aeropuerto, que no se encuentran en el dataset de vuelos. Por esto realizamos una junta de cada vuelo con dichos datos que provienen de otro dataset.

El **Joiner** es el encargado de hacer la junta, por lo tanto, primero espera recibir todos los datos del dataset de latitud y longitud, al ser pequeño (~875kB) lo mantiene guardado en memoria.

Una vez recibido todo el dataset de latitud y longitud, comienza a escuchar pedidos del dataset de vuelos y por cada vuelo recibido le agrega la latitud y longitud del aeropuerto de salida y el aeropuerto de llegada.

El vuelo junto a las latitudes y longitudes se envían al **Processor** de distancias que compara la distancia total con la distancia directa y si cumple la condición es enviado al **Tagger**. El **Processor** de distancias mantiene un diccionario en memoria que funciona como *cache* en el cuál se van guardando las distancias entre aeropuertos ya calculadas, esto se realiza porque dicho cálculo consume un tiempo de procesamiento no despreciable. Al tener una baja cardinalidad de aeropuertos, este *cache* no crece demasiado y no representa un problema en el uso de la memoria.

El **Tagger** finalmente agrega un tag al vuelo para indicar que es un resultado de este tipo de query y lo envía al **Sink** correspondiente.

Vista de Proceso

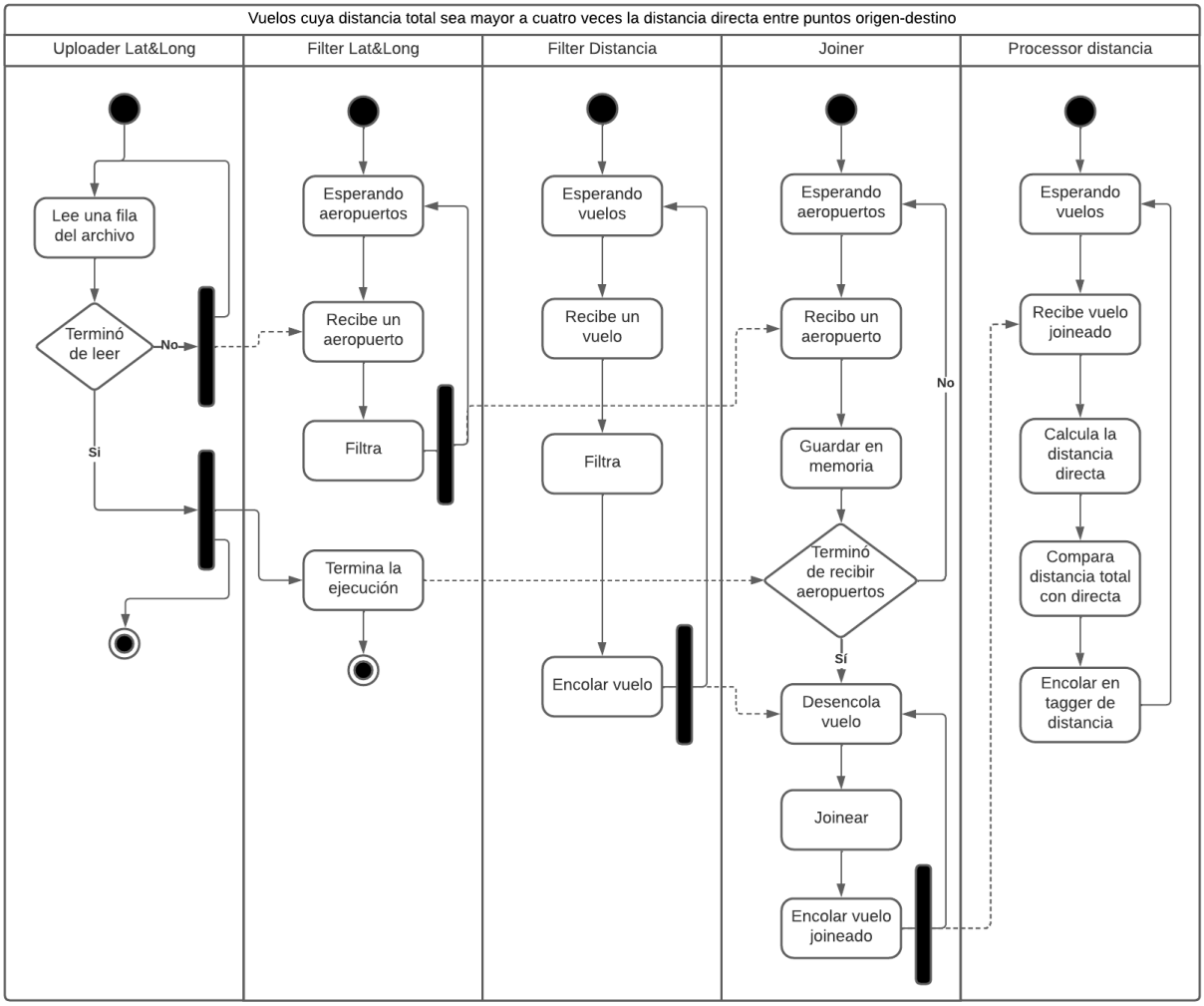

Figura 4.4 - Diagrama de actividades de la consulta de vuelos cuya distancia total sea mayor a cuatro veces la distancia directa entre puntos origen-destino

Query 3: Dos vuelos más rápidos

Vista Lógica

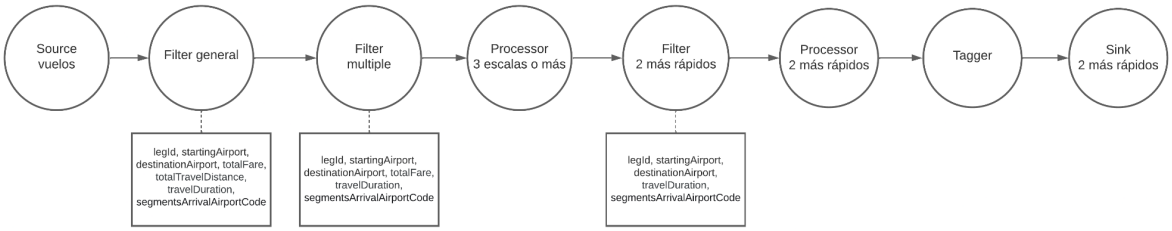

Figura 4.5 - DAG de la consulta de los dos vuelos más rápidos que tengan tres escalas o más

Para esta consulta se reutiliza lo procesado para la consulta de tres escalas o más hasta el **Processor** de tres escalas o más. La salida de dicho **Processor** pasa por un **Filter** para eliminar columnas innecesarias y luego lo toma el **Processor** de dos más rápidos.

Este **Processor** lo que hace es ir guardando los dos vuelos más rápidos por trayecto en un diccionario en memoria, donde la clave es el trayecto y el valor una lista ordenada con los dos vuelos más rápidos de dicho trayecto. El valor de los vuelos más rápidos se va pisando, si corresponde, a medida que van llegando los vuelos, hasta que llegue el mensaje de *end of file* que señala el fin del proceso del dataset de vuelos. En ese caso, el **Processor** envía al **Tagger** el resultado de cada trayecto. El **Tagger** le agrega el tag que corresponde y lo envía al **Sink**.

Vista de Proceso

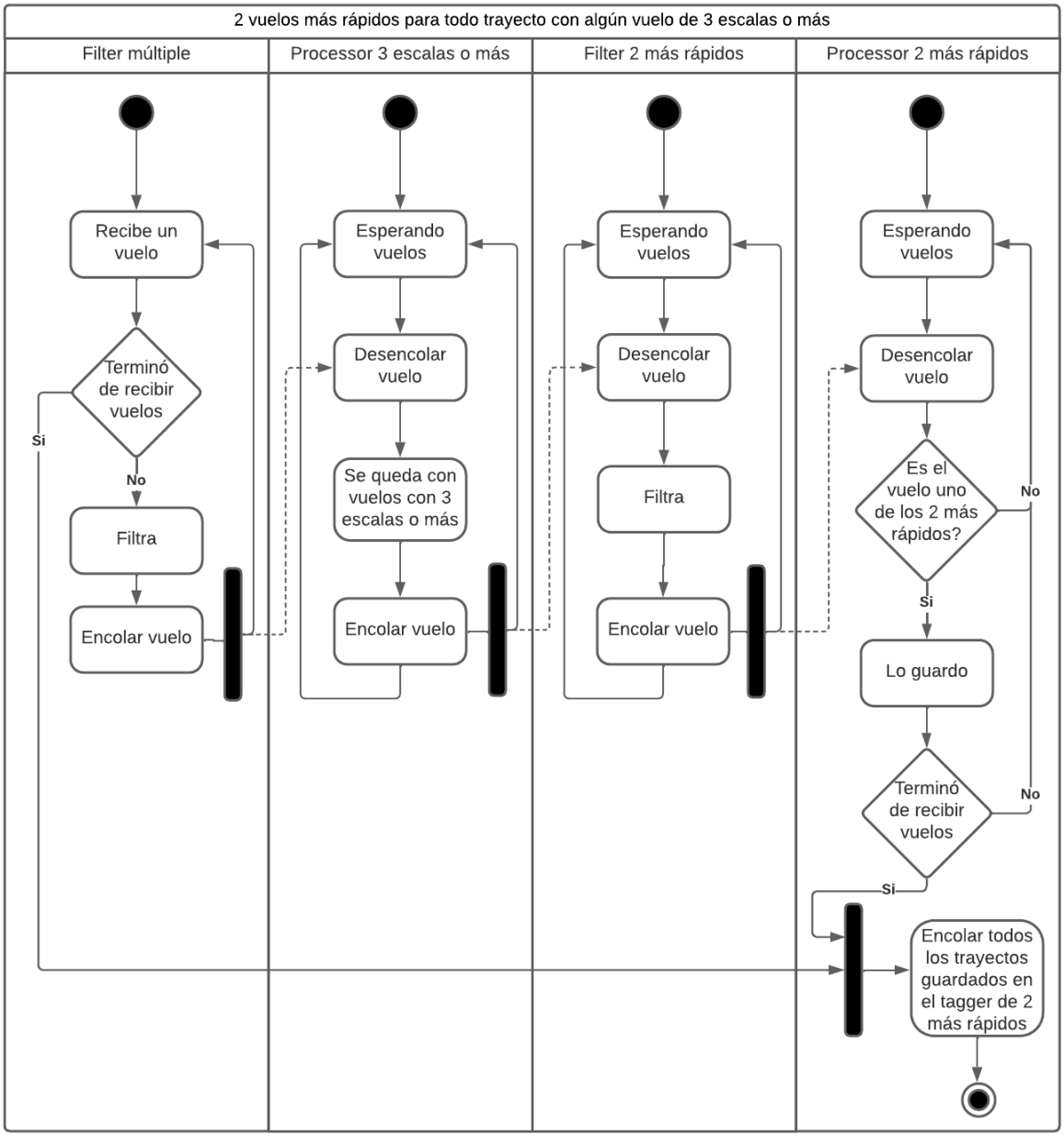

Figura 4.6 - Diagrama de actividades de la consulta de los dos vuelos más rápidos que tengan tres escalas o más

Query 4: Promedio y máximo de precios

Vista Lógica

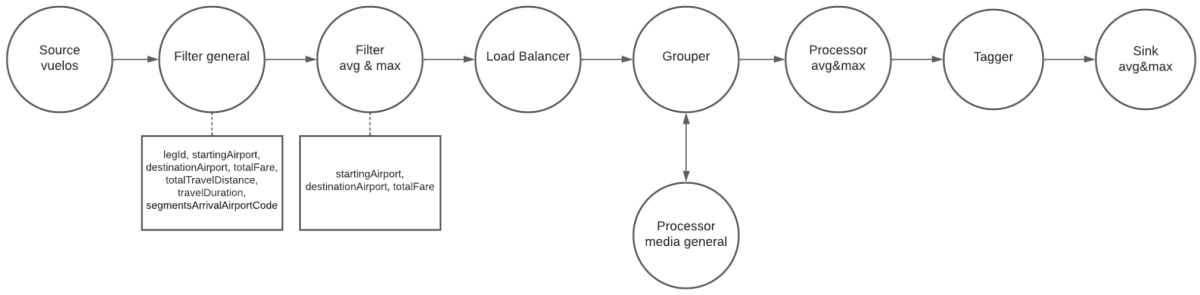

Figura 4.7 - DAG de la consulta del promedio y máximo de precios cuyo precio está por encima de la media general (por trayecto)

Para esta consulta contamos con la entidad **Load Balancer** que realiza un balanceo de carga particionando el dataset de vuelos hacia cada instancia de **Grouper**. Cada **Grouper** es encargado de procesar una partición del dataset de vuelos, esta partición se hace por trayecto: se aplica una función de hash al trayecto (combinación entre aeropuerto de salida y aeropuerto de destino) y a ese resultado se le aplica un módulo por la cantidad de instancias de **Groupers** para decidir a cuál de ellos debería ser enviado.

Los **Groupers** agrupan los precios de los vuelos por trayecto en un diccionario en memoria, donde la clave es el trayecto y el valor es la lista de precios. Una vez recibido el *end of file* que indica la finalización del dataset de vuelos, cada **Grouper** suma todos los precios de todos los trayectos y calcula la media. Este valor es envíado al **Processor** de media general.

Cuando el **Processor** de media general recibe todas las medias parciales de los **Groupers**, calcula la media general y le envía la respuesta a cada **Grouper**.

Cada **Grouper** recibe la media general y pasa a realizar un filtrado quedándose con los precios que se encuentran por encima de dicha media. Luego, envía un mensaje por trayecto, con los precios que quedaron, al **Processor** avg & max.

Este **Processor** simplemente calcula la media de precios y el precio máximo de cada trayecto que recibe (recordar que en este punto un mensaje equivale a un trayecto). Por último, envía este resultado al **Tagger** que finalmente lo envía al **Sink** con su tag correspondiente.

Vista de Proceso

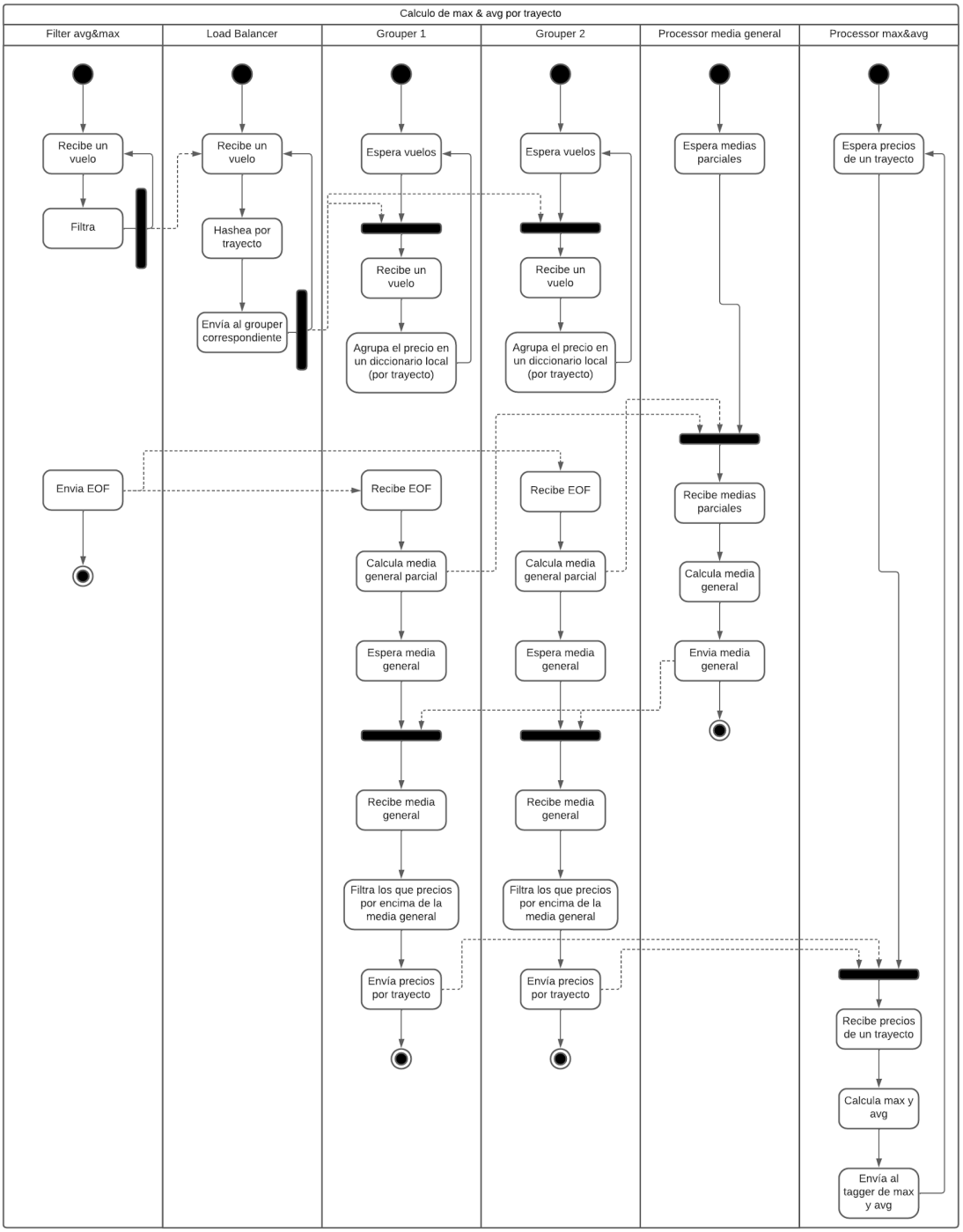

Figura 4.8 - Diagrama de actividades de la consulta del promedio y máximo de precios cuyo precio está por encima de la media general (por trayecto)

5. Vista Física

Diagrama de despliegue

Figura 5.1 - Diagrama de despliegue de todo el sistema

Diagrama de robustez

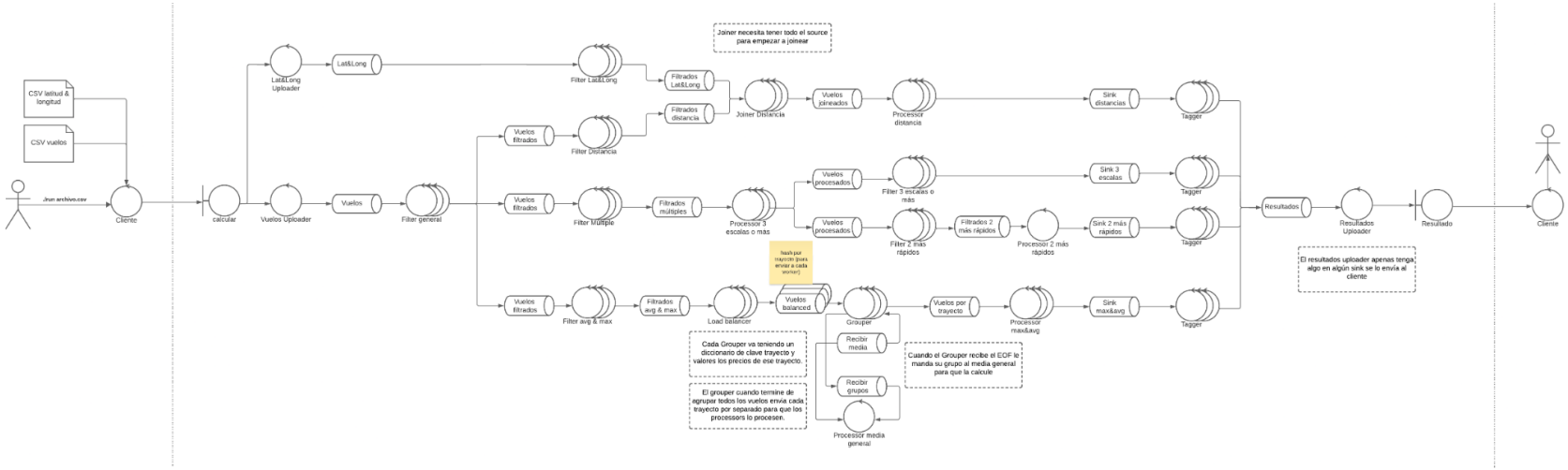

Figura 5.2 - Diagrama de robustez de todo el sistema

5. Performance

Se analizó el performance de la arquitectura con los siguientes escenarios:

- **1er escenario**: Dataset reducido de 2 millones de líneas
    - Se completó la ejecución en 1 minuto y 30 segundos.
    - Se utilizan alrededor 890 megabytes de RAM, contenida en su mayoría en **Rabbit** (500mb) y de nuestro dominio en los **groupers** (100mb) distribuído entre todos.

- **2do escenario**: Dataset entero de 50 millones de líneas
    - Se completó la ejecución en 14 minutos y 30 segundos.
    - Se utilizan alrededor de 3 gigabytes y 600 megabytes de RAM, contenida en su mayoría en **Rabbit** (500mb) y de nuestro dominio en los **groupers** (3gb) distribuído entre todos.

Como se puede apreciar el rendimiento no parece ser lineal, ya que a medida que suben la cantidad de líneas, tarda menos que su factor de crecimiento.

Es importante notar que los “benchmarks” se realizaron con un disco SSD, con un disco duro los rendimientos pueden verse afectados generando un cuello de botella, más que nada en el guardado de resultados. El mismo fue realizado utilizando un procesador AMD® Ryzen 5 1600 six-core processor.