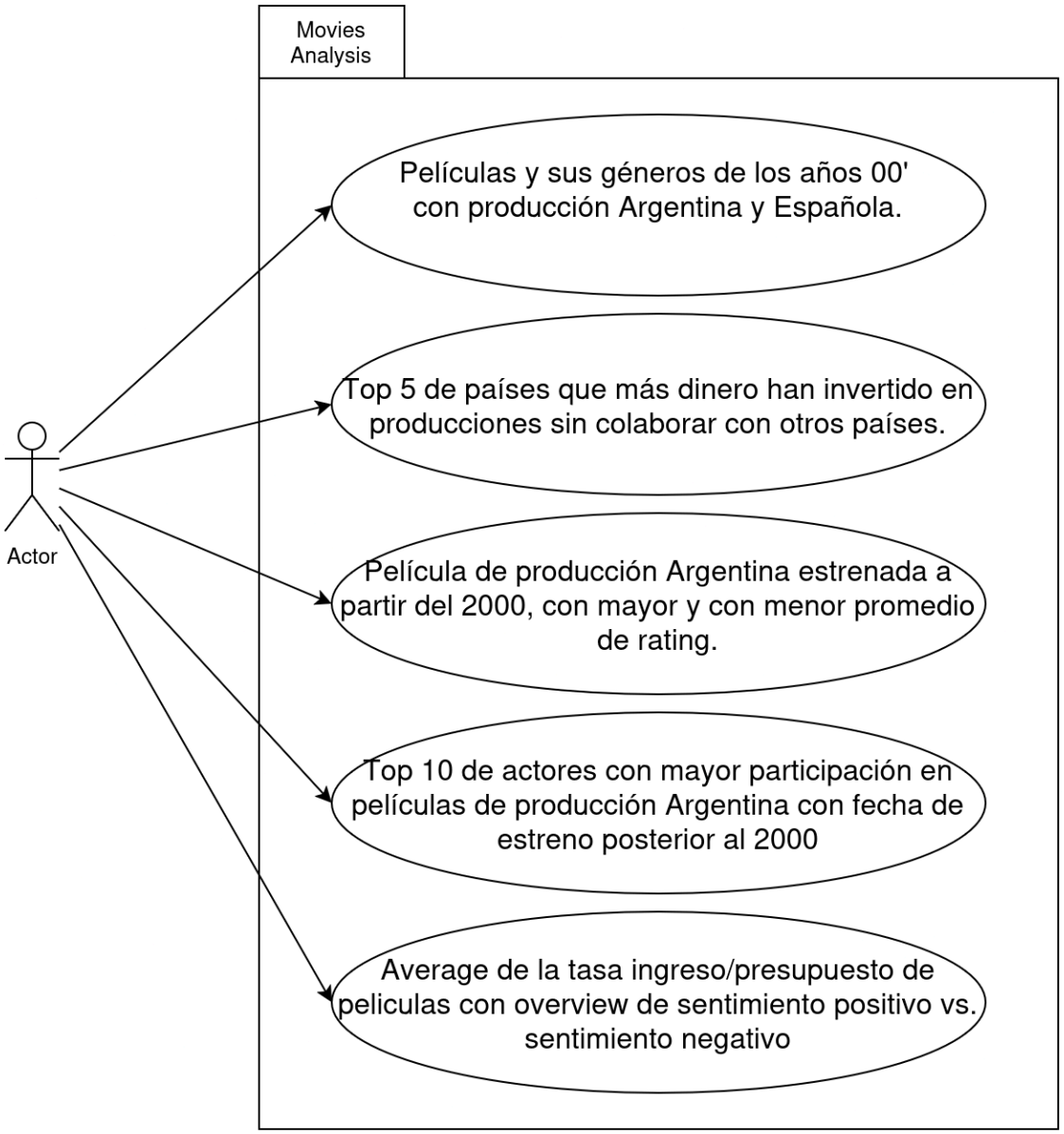
Documento de Arquitectura - Movies Analysis

Alumnos:
Belforte Paolo
Bohorquez Rubén
Gauler Ian

Índice

Casos de Uso

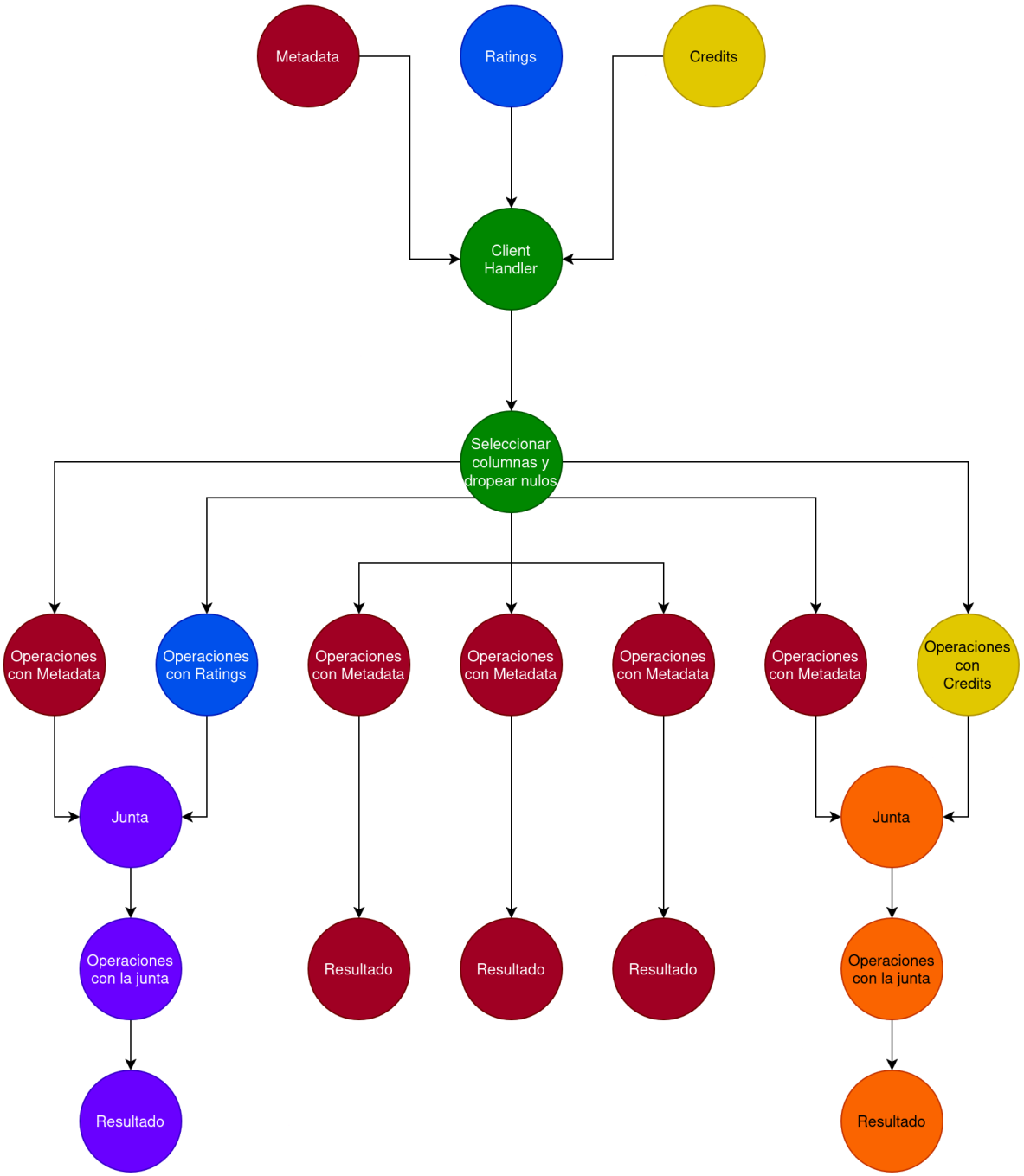
- Como usuario quiero poder obtener:
    1. Películas y sus géneros de los años 00' con producción Argentina y Española.
    2. Top 5 de países que más dinero han invertido en producciones sin colaborar con otros países.
    3. Película de producción Argentina estrenada a partir del 2000, con mayor y con menor promedio de rating.
    4. Top 10 de actores con mayor participación en películas de producción Argentina con fecha de estreno posterior al 2000
    5. Average de la tasa ingreso/presupuesto de películas con overview de sentimiento positivo vs. sentimiento negativo



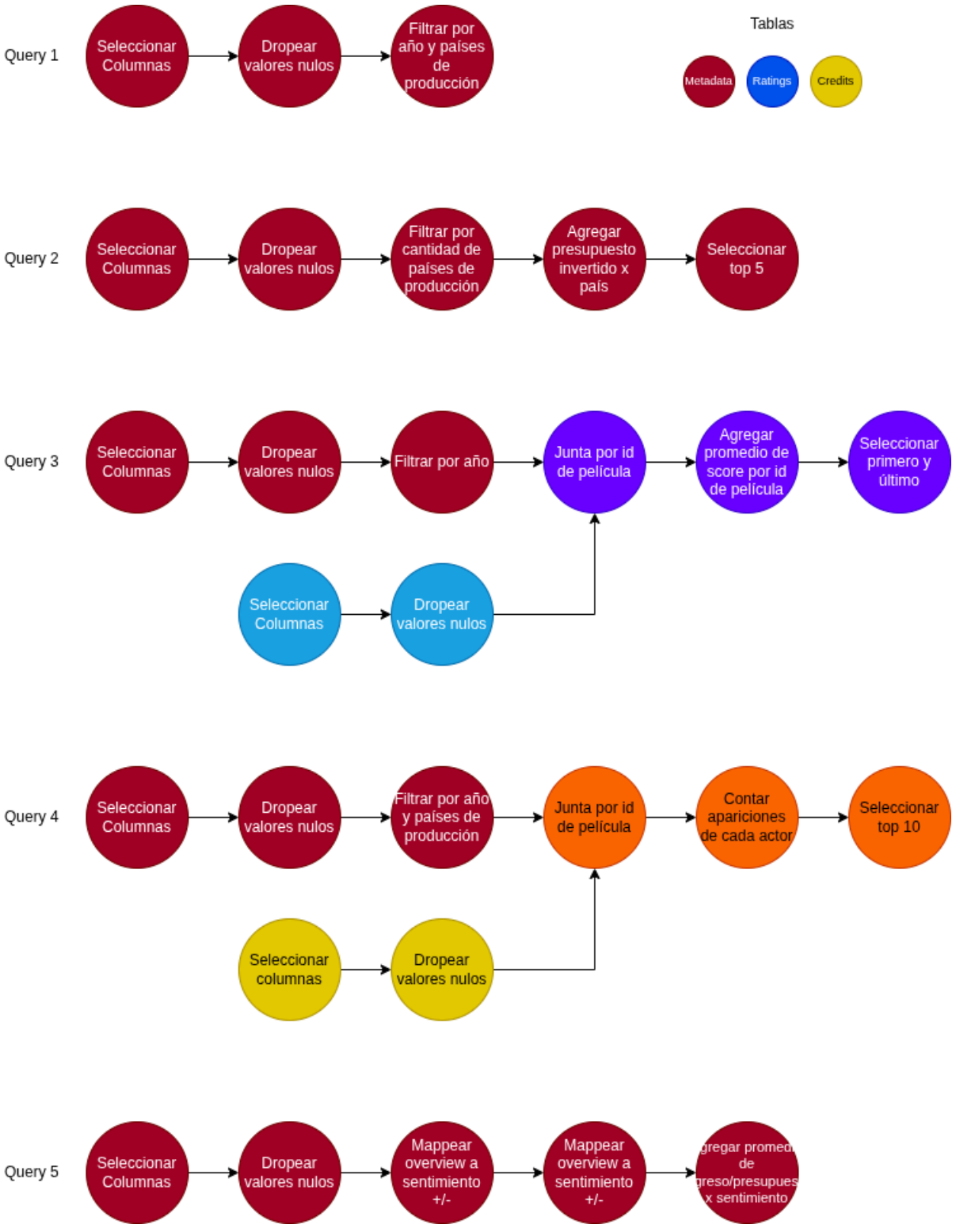
Vista Lógica

A continuación se incluye un DAG general para visualizar el flujo de datos de toda la aplicación. Para mantener los diagramas legibles, la vista general simplifica las operaciones específicas de cada query. Además, se incluyen DAGs detallados para cada query por separado. Entre los 2, se puede apreciar la secuencia completa del flujo de datos para cada query.

Flujo de la aplicación

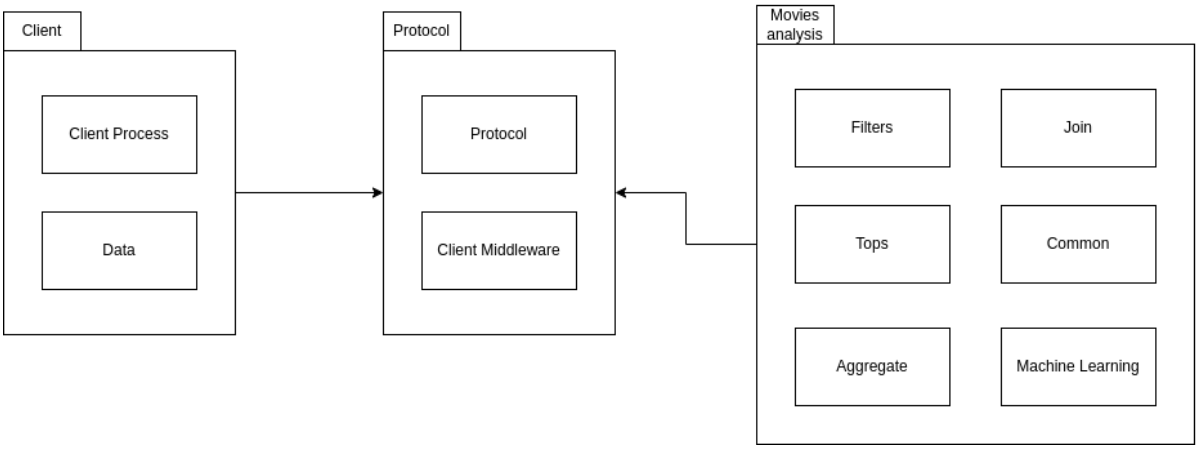


Flujo de las queries

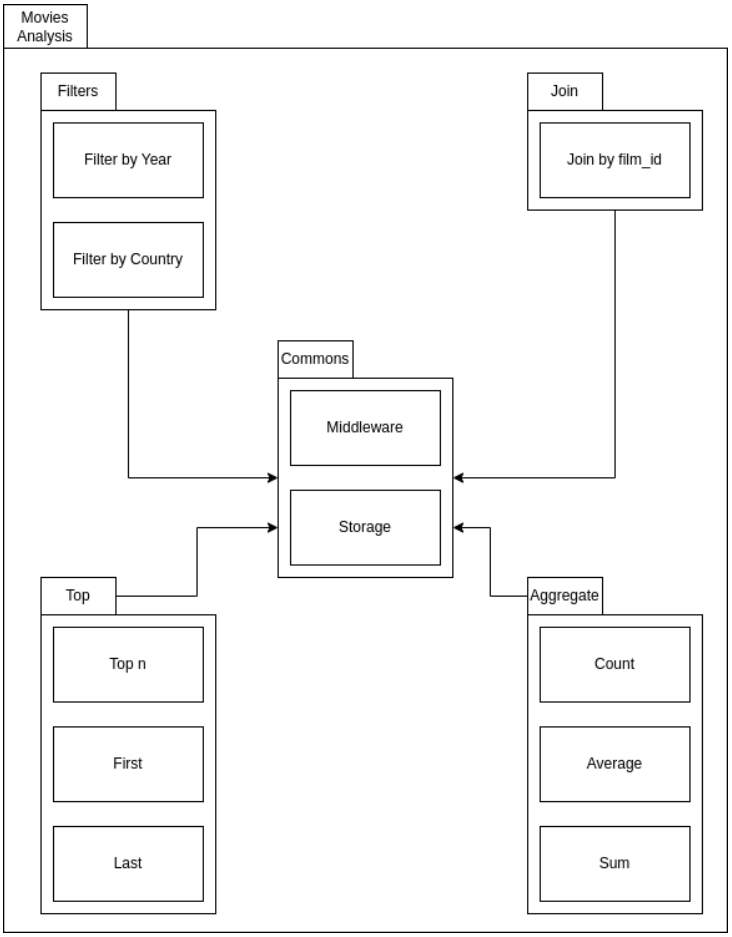


Vista de Desarrollo

Diagrama de Paquetes



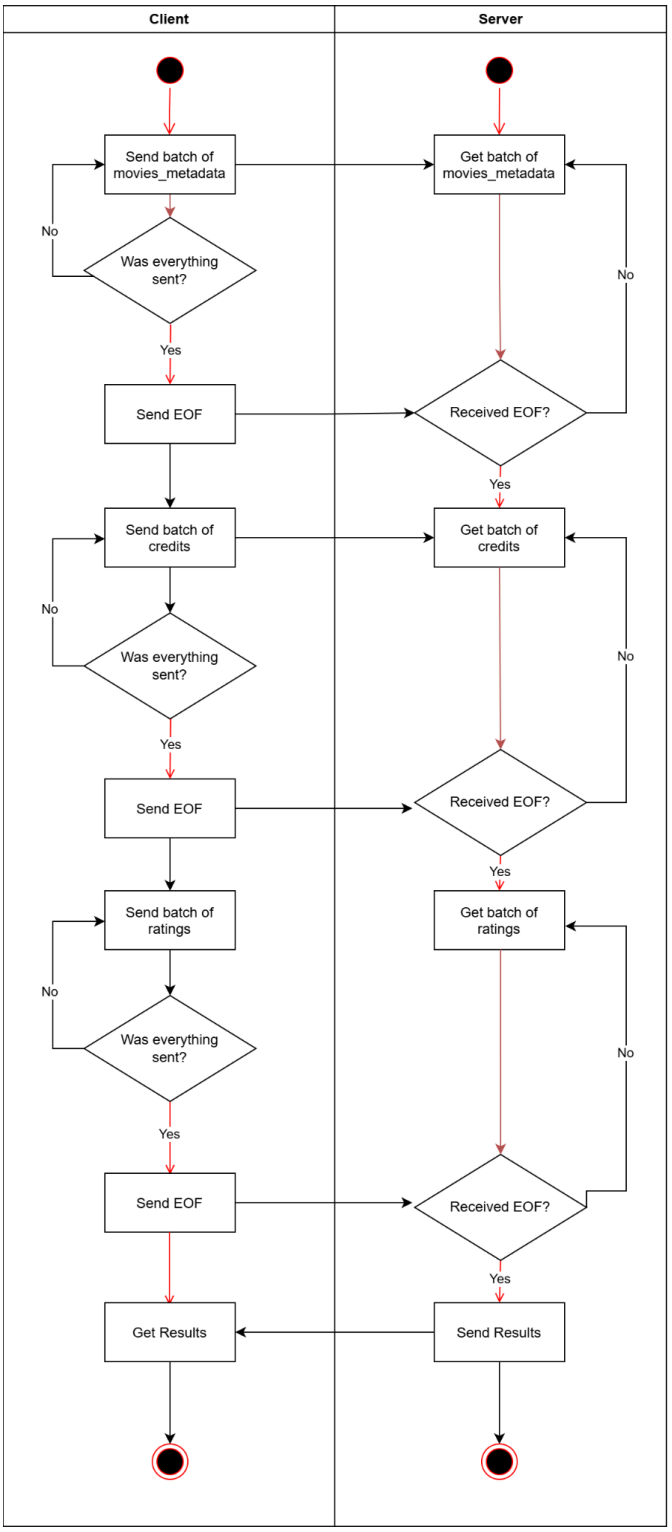
Con este diagrama se denota la relación entre nuestras dos entidades y cómo se comunican gracias al paquete que llamamos protocol.
Para el siguiente diagrama buscamos mostrar cómo los paquetes de nuestro Movie Analysis se comunican entre sí gracias al paquete Common, el cual contiene el Middleware encargado de la comunicación y un storage que sirve para la persistencia en caso de fallas en el nodo, resultados obtenidos o el procesamiento del join.



Vista de Procesos

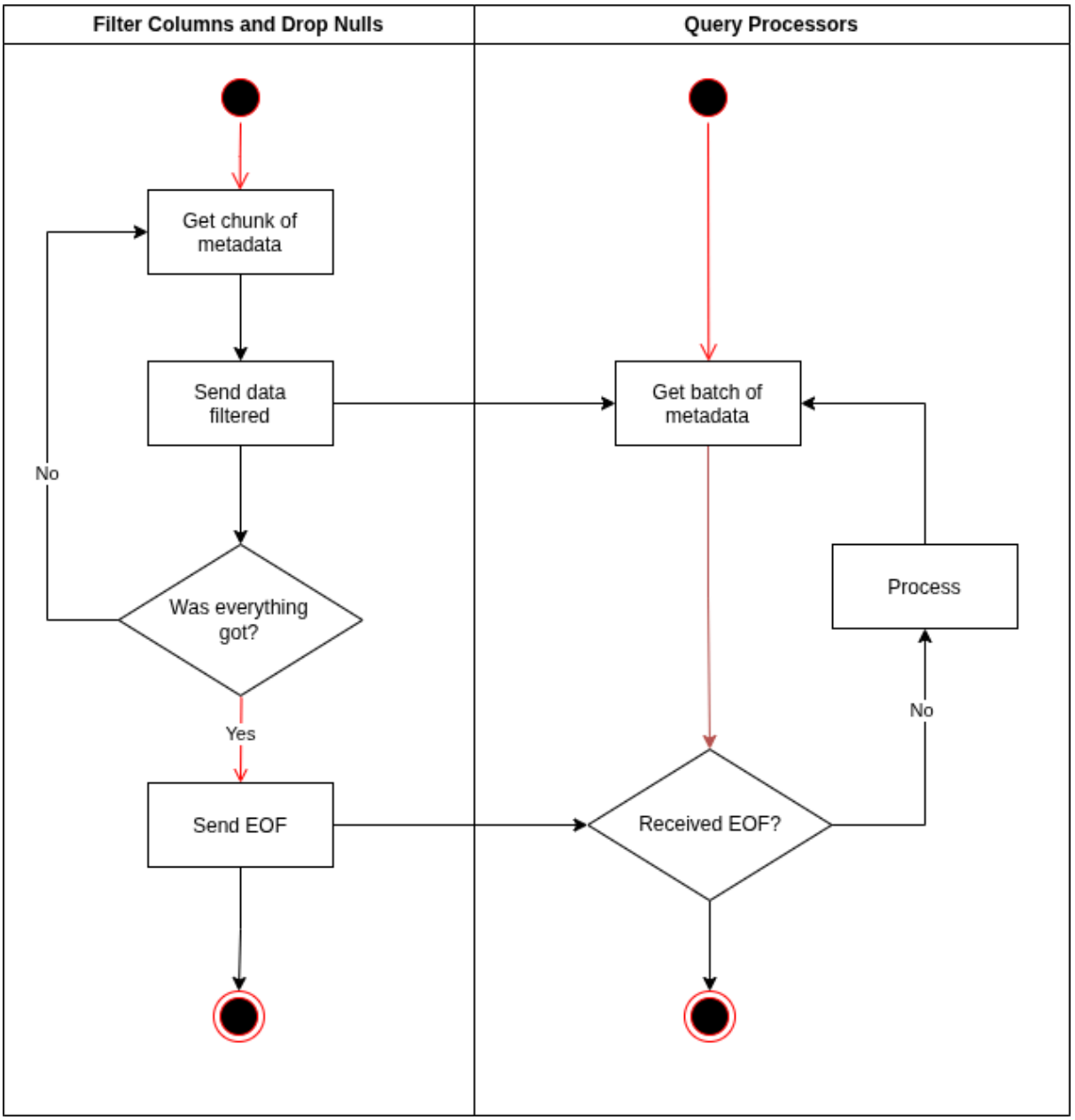
Diagrama de Actividad

Diagrama de actividad general



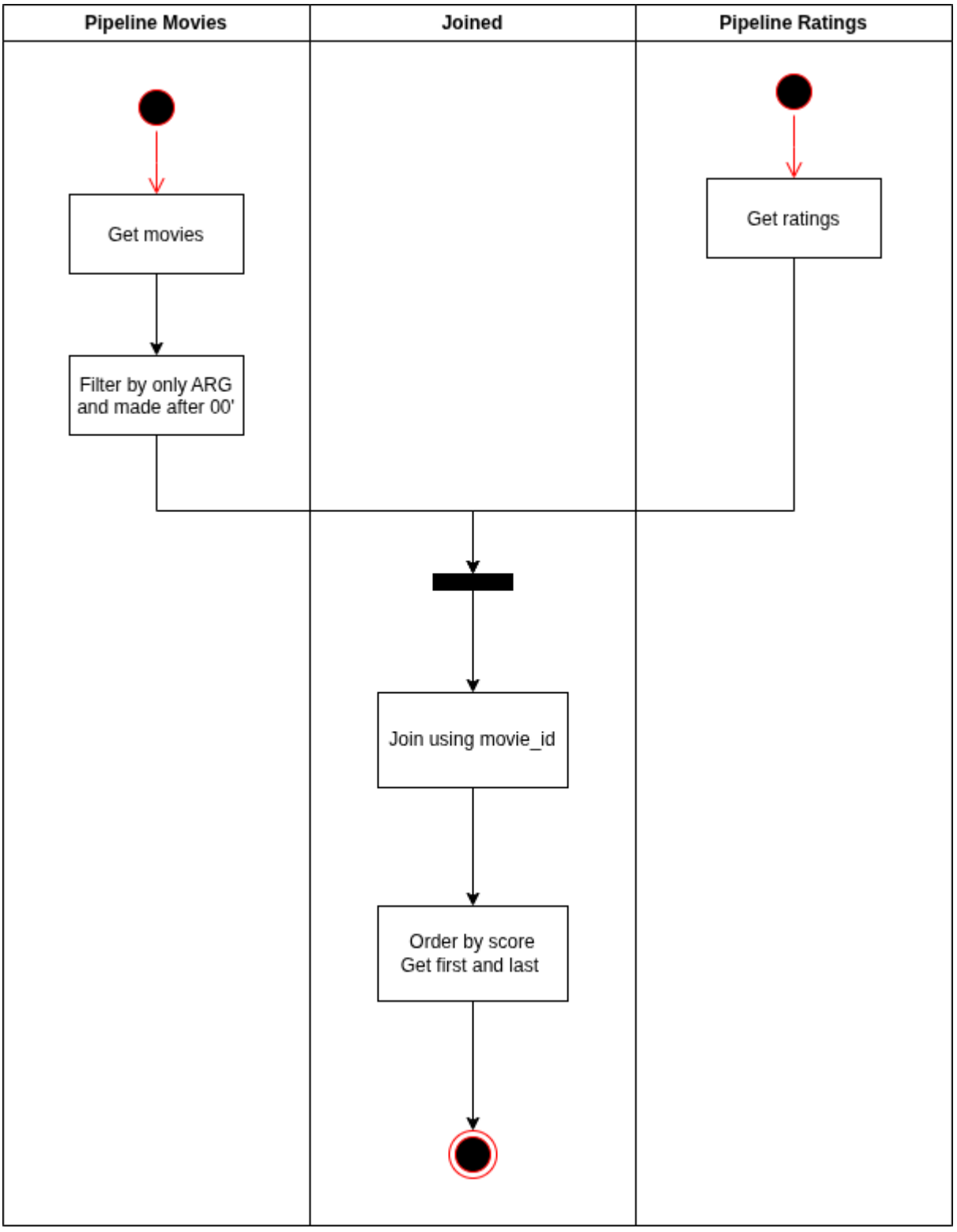
En el diagrama se puede observar cómo funciona la comunicación entre el cliente y el servidor para el envío de los datos y recibir los resultados de las querys

Diagrama de actividad pre filtrado para las querys en el Client Handler



El prefiltro es común para todas las querys, en este se sacan datos innecesarios de los csv y se quitan los valores nulos. Además se filtran aquellas filas que no cumplan el formato debido para la transformación de tipos de datos, como por ejemplo de string a int.

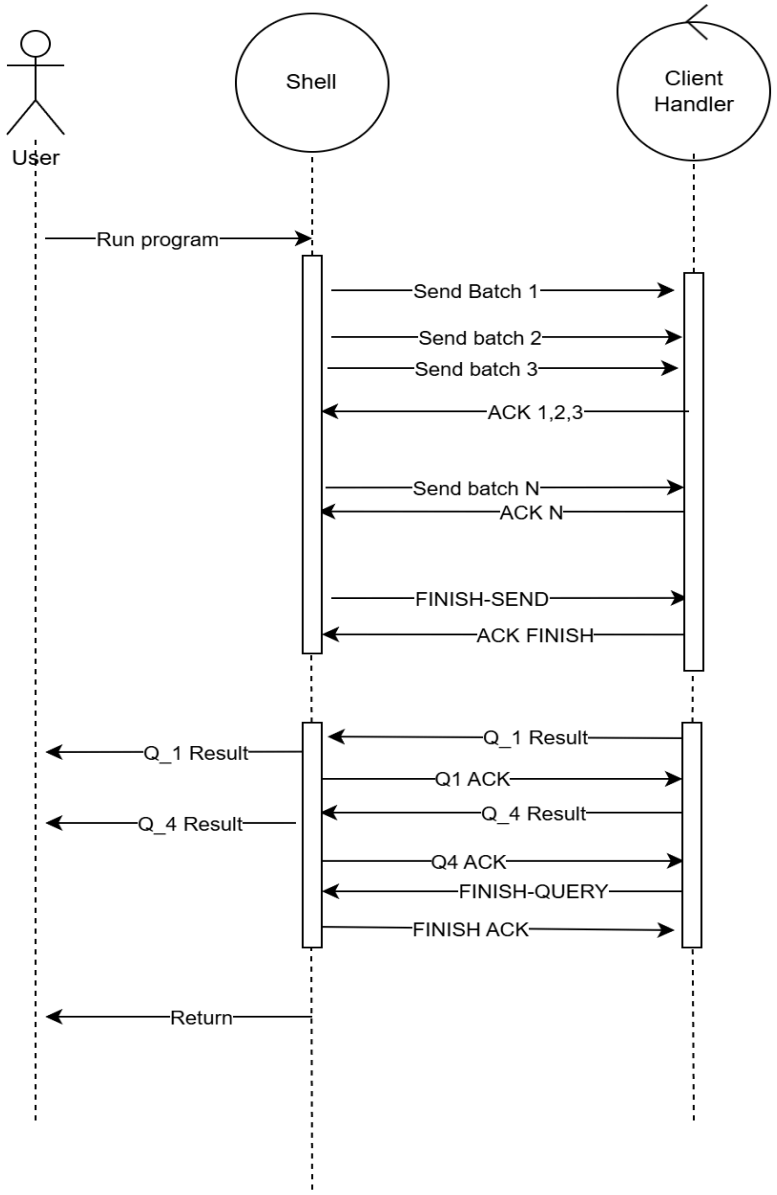
Diagrama de actividad de una query



En este último diagrama de actividad, se pone como ejemplo la Query3, en el cual de manera simplificada se ven los pasos que toma, filtrando las películas y agrupando los ratings para unirlos y obtener el resultado

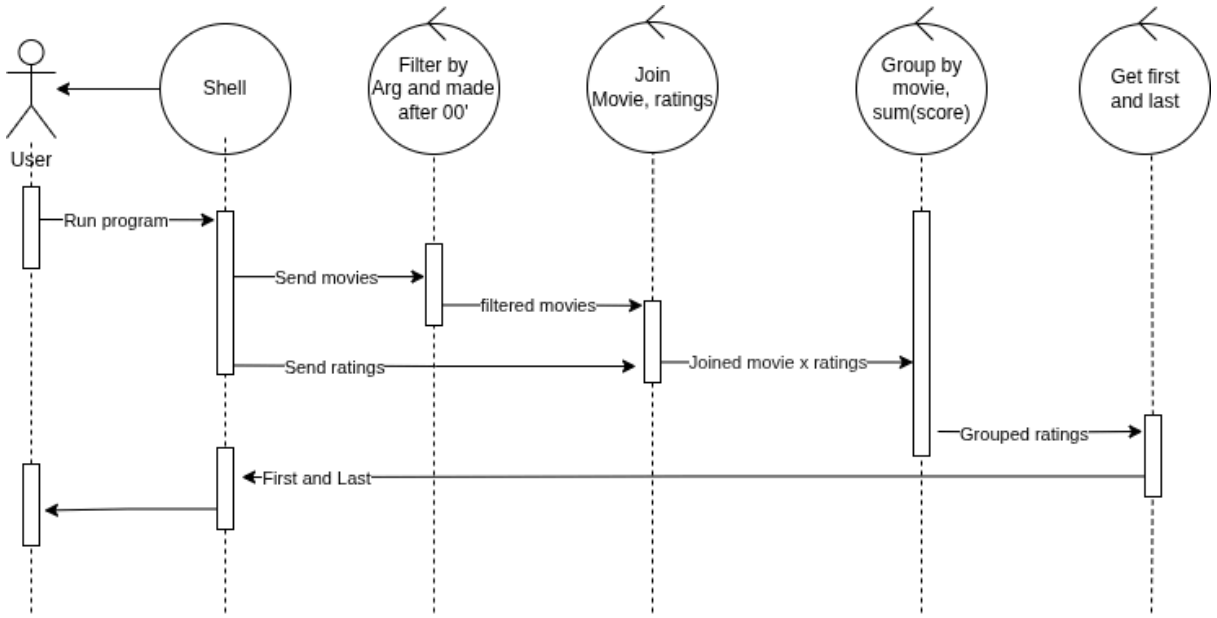
Diagrama de Secuencia

Diagrama de secuencia general Client y Client Handler



El envío de datos se realiza en forma de batches. El cliente envía batches de tamaño configurable, el client handler los toma, realiza los filtros básicos pertinentes, que corresponden a la limpieza de filas con valores que deberían ser numéricos y no lo son, o valores faltantes.
Luego realiza un ruteo, el mismo que realizan todos los nodos siguientes en el pipelining, para determinar a qué nodo enviar la data (ya sea movies, ratings o credits) hasheado por el id de la movie. Con esto aseguramos que cada línea del csv se dirija de forma determinística por un solo camino de nodo y salvar algunos casos bordes que surgían al usar una única cola (working queue) para todos los nodos de un mismo tipo. Este tema se tratará en el último apartado.

Diagrama de secuencia de una query



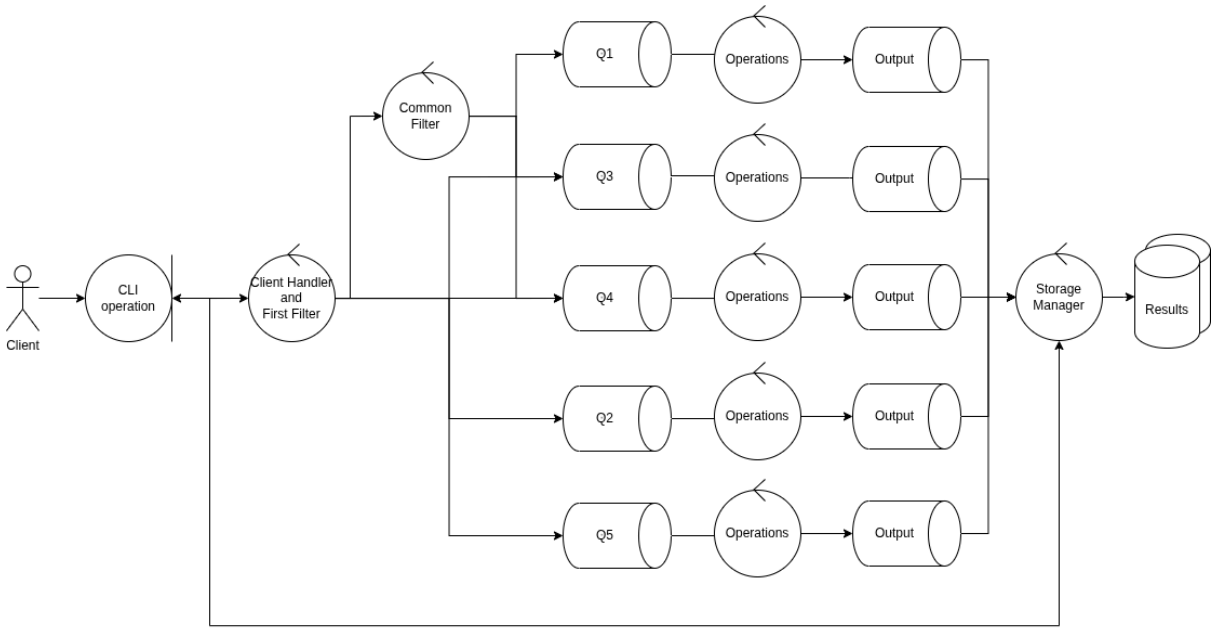
Se eligió mostrar como ejemplo el caso de la Query2, pero todas las demás funcionan de manera similar, realizando cada nodo de la cascada del pipelining un pequeño trabajo, hasta llegar al último nodo de la línea. Todos los nodos trabajan con el patrón direct, a pesar de estar usando un exchange de tipo topic, excepto el último nodo del pipelining, ya que el Client Handler escucha de una cola con el routing key de *.results. Con lo que el último nodo de cada query envía un resultado con el routing key de, en el caso de Get First and Last “query2.results”. De esta forma el Client Handler sabe qué query está tratando sin tener que incluirlo en el payload, ya que la routing key viene incluida en la metadata del mensaje. Así es como el Client Handler envía el resultado al Cliente y éste último sabe de qué query se refiere el resultado.

Vista Física

Diagrama de robustez

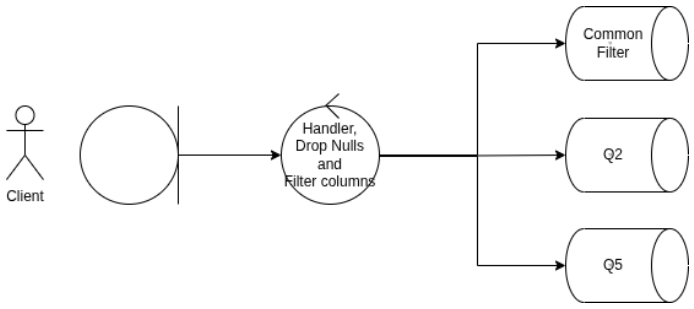
A continuación se incluyen varios diagramas de robustez, a distintos niveles de detalle y de distintas partes del sistema. Entre todos, proporcionan una vista a alto nivel de todo el sistema

Diagrama de robustez general



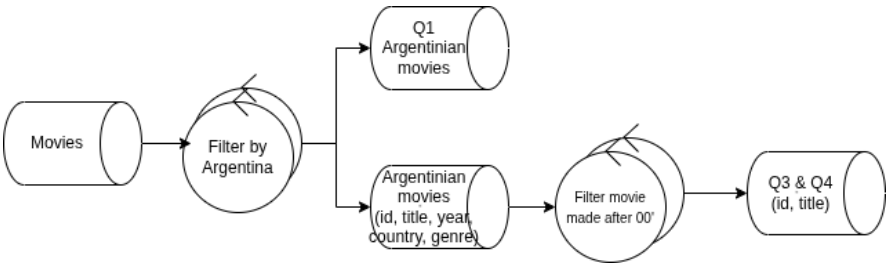
En el diagrama de arriba podemos observar cómo funciona a grandes rasgos el pipelining del sistema. Tenemos el Client Handler que se comunica directamente con los nodos de la Query 2 y 5 enviando el dataset de Movies. Pero en el caso de la Query 3 y 4 está conectado enviando Ratings para el caso de la Q3 y Credits para el caso de la Q4. Las Queries 3 y 4, así como la Query 1 reciben las movies mediante el procesamiento del Common Filter, que no es solamente un nodo. Sino que en este diagrama se representa de forma generalizada, pero en el diagrama de Common Filter podemos ver cómo los nodos se comunican con los siguientes.

Detalle - Filtro inicial

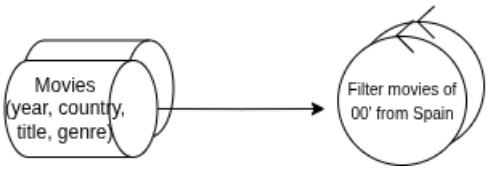


Detalle - Common Filter

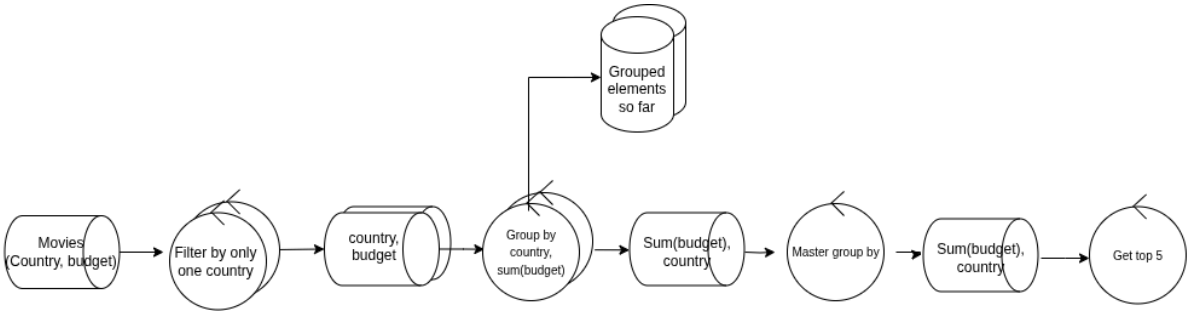
Como se mencionó anteriormente, el Common Filter recibe las movies del Client Handler. Y el Common Filter está conformado por los filtros de Argentina y Movies After 2000. El Filtro de Argentina envía las movies filtradas al siguiente nodo que vemos en la Query 1 y al Filtro Movies After 2000, éstos últimos nodos envían a los nodos de Join de la Query 3 y 4 respectivamente.



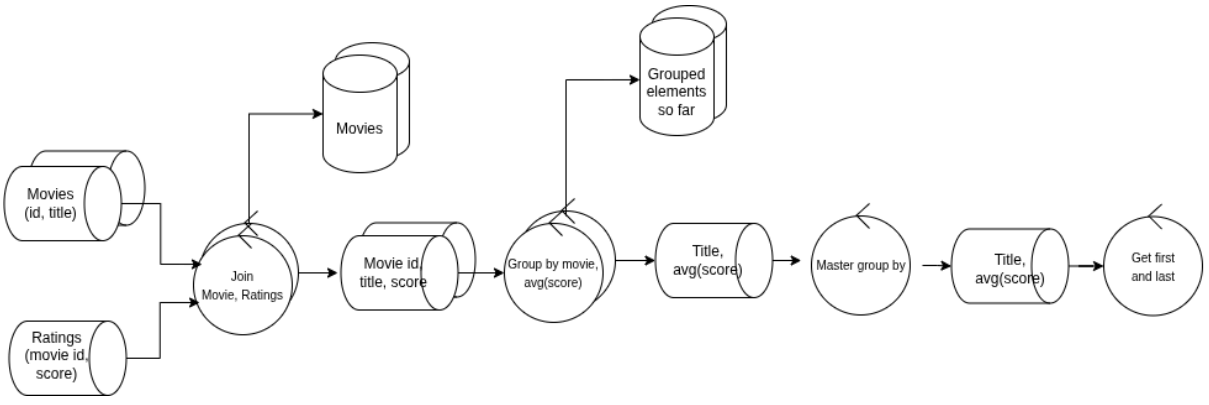
Detalle - Query 1



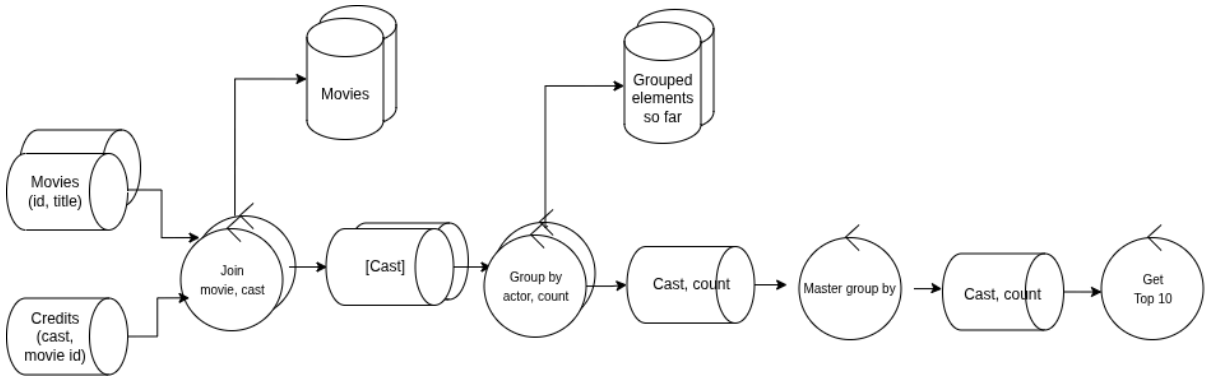
Detalle - Query 2



Detalle - Query 3



Detalle - Query 4



Detalle - Query 5

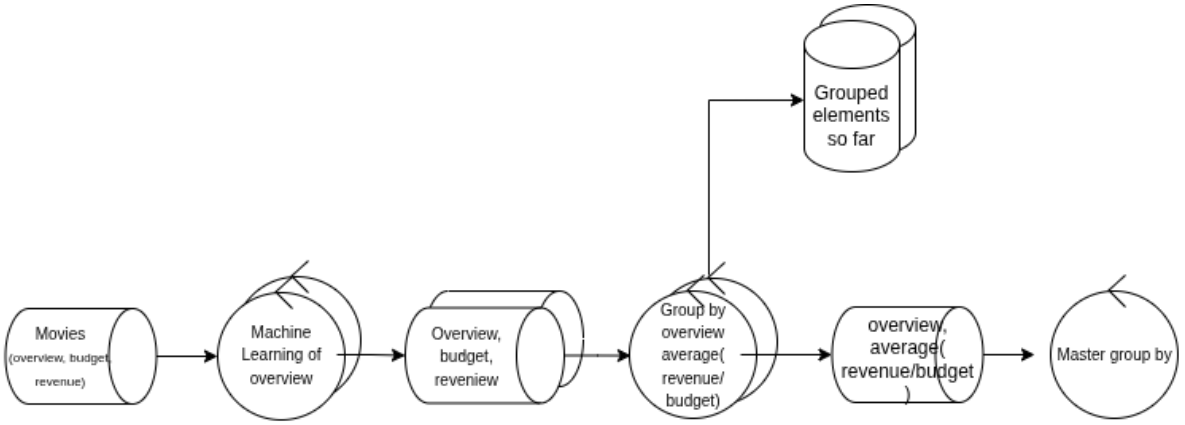
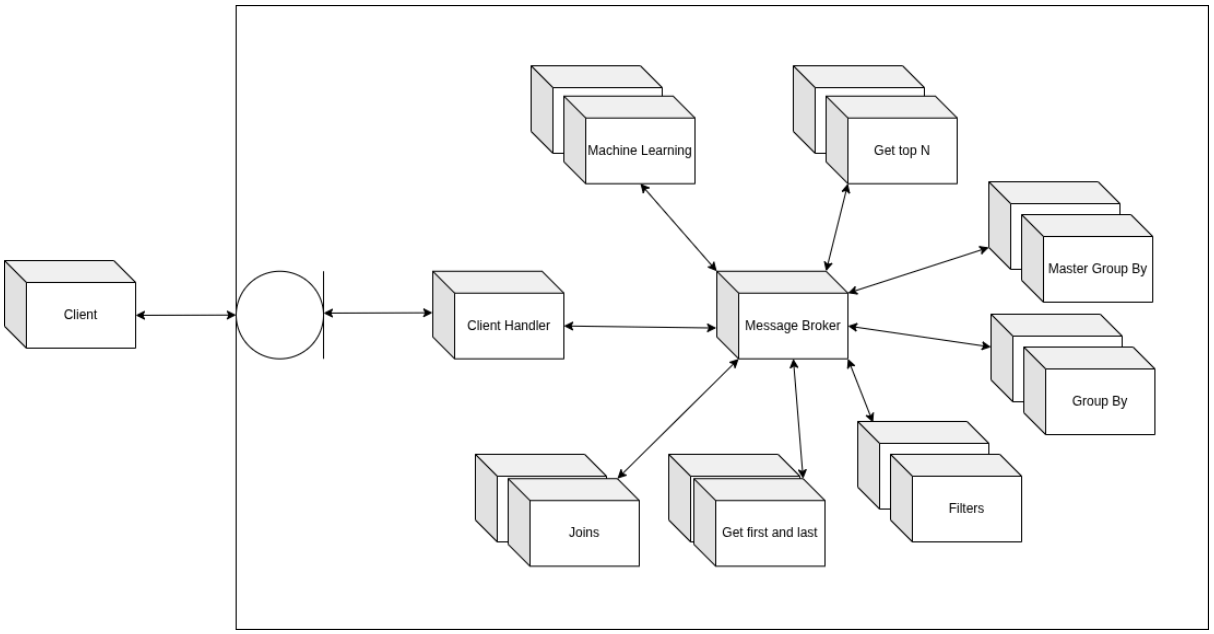


Diagrama de despliegue



El anterior diagrama busca mostrar los componentes principales del sistema, cada nodo representa uno de los workers vistos en el diagrama de robustez (filtros y joins por ejemplo) y como se comunican con el broker de mensajes

División de Tareas

La siguiente división es tentativa y está sujeta a cambios según los horarios de los integrantes y posibles modificaciones al diseño

Todos: Middleware y storage
Paolo:
Semana 1 (8/4 - 15/4) :
- Eliminar duplicados
- Eliminar nulos

Semana 2 (16/4 - 22/4):
- NLP
- Obtener top N / Primero y último

Ian:
Semana 1 (8/4 - 15/4):
- Enviar archivos al server
- Obtener resultados

Semana 2 (16/4 - 22/4):
- Filtros

Ruben:
Semana 1 (8/4 - 15/4):
- Junta
- Agregación

Semana 2 (16/4 - 22/4):
- Almacenamiento
- Integración con el almacenamiento

Consideraciones

Existe una pequeña consideración que queremos dejar en claro, y es que tuvimos una larga historia de cambios en la arquitectura para la distribución de mensajes a los nodos workers.
La primera decisión de arquitectura que al inicio del proyecto fue usar muchas colas, con distintos exchanges que dividirían cada tipo de nodo. Por ejemplo para el caso de la conexión entre el master group by country sum y get top 5, el exchange tenía un nombre declarativo para el mismo, y había una cola por cada nodo réplica del get top 5, con routing keys declaradas como get_top_5.1, De forma tal que sea declarativo.
Luego intentamos usar working queues, en donde todos los nodos de un mismo tipo, por ejemplo todas las réplicas de Filtro Argentina compartían la misma cola. Esto nos trajo muchos problemas al manejar los casos bordes por EOF, ya que la caída de un nodo que había tomado un EOF traería problemas en el reencolado, ya que podríamos tener desordenamiento, y por consiguiente, malos resultados de procesamiento. Con esa arquitectura la idea fue dividir el Exchange en Data y Control. Estos dos exchanges servirían para dividir la data y la lógica de manejo del EOF, de forma tal que toda la data circularía por el Data Exchange, incluído el EOF del archivo, y una vez que un nodo recibiría este EOF se convertiría en líder, informando a todas las demás réplicas del mismo tipo que el EOF habría llegado, y que respondan al líder que puede enviar el EOF al siguiente nodo una vez que sus compañeros hayan terminado de procesar el archivo. Esta decisión fue

descartada por diversos problemas y su alto nivel de complejidad. Aunque a su vez traía mucha ventaja, como la posibilidad de que el trabajo sea perfectamente equitativo entre nodos réplica, ya que RabbitMQ realiza un Round Robbin entre nodos para consumir el mensaje, y a su vez si un nodo caía, otros nodos estarían consumiendo de la cola, sin quedar en espera esta cola con mensajes hasta que un nodo se levante, como en el caso de una cola por nodo

Por último decidimos usar un solo Exchange de Data y de tipo topic para el manejo de routing keys, cada nodo tiene una cola asociada, y tenemos un módulo llamado hasher que es el encargado de decidir a qué cola de las réplicas enviar el mensaje. La contra de este método es que si un nodo cae por alguna razón, hay que esperar a que se levante de nuevo para que sean consumidos los mensajes de su respectiva cola. La ventaja es que nos aseguramos que solamente un nodo consume de una cola, por lo que el desordenamiento de los mensajes ante caídas es mínimo (el único caso borde es explicado en el apartado de casos bordes). Y por último, el uso de solamente un exchange de tipo topic siendo que se usa para todos los nodos el tipo direct, excepto para la comunicación de los últimos nodos del pipelining con el Client Handler donde si se usan los wildcards, es porque en la documentación no mencionan una diferencia en la performance entre el uso de direct y topics. Aunque se podría pensar que direct tiene menos lógica asociada, con lo que podría ser más rápida. Si fuera el caso, podríamos separar los exchanges y dejar el exchange de topic solamente para los resultados, o incluso podríamos incluir en el payload del resultado el número de query a la que se está respondiendo.

Detalles del protocolo

El protocolo que utilizamos para los mensajes que llegan a rabbit es uno en el cual los mensajes están formados de a batches por líneas que contienen 3 o 4 partes principales dependiendo el tipo. La primera es el id del cliente que envía el mensaje, luego es el id único de ese mensaje y luego viene el payload. Si el mensaje es un EOF el payload simplemente serán los caracteres de EOF, pero si es un mensaje de datos va a estar primero el id de la película a la que corresponde ese mensaje y luego el resto del payload, se deja un ejemplo de cómo se vería un mensaje EOF y un mensaje de datos

```
b709d2b9|690ae4b2|EOF
f1a982e1|aa58a4e9|100|PAYLOAD
```

Asumimos que cuando un mensaje ya está dentro de rabbit, no se va a corromper por lo que no vamos a encontrarnos con mensajes con formatos incorrectos dentro de nuestros nodos.
También queremos aclarar que utilizamos ids de mensajes únicos utilizando UUID4, puede ocurrir que se repita un mismo uuid? Si. Pero la probabilidad de que ocurra es tan baja que simplemente vamos a dejar lo que dice la función de UUID4 de go “Randomly generated UUIDs have 122 random bits. One’s annual risk of being hit by a meteorite is estimated to be one chance in 17 billion, that means the probability is about 0.00000000006 (6×10^{-11}), equivalent to the odds of creating a few tens of trillions of UUIDs in a year and having one duplicate” creemos que no hace falta decir más nada.
Por otro lado, algo que se realizó para el envió de mensajes, es que siempre se rutean los mensajes a través de un hash utilizando un ID_PELICULA, esto es que si tenemos 3 nodos, cuando queramos enviar un mensaje este se envía a el nodo que corresponde haciendo ID_PELICULA % (len(nodos_que_tengo_delante)) esto nos asegura que siempre se envían los mismos mensajes a los mismos nodos, tiene la desventaja de que no estamos enviando los mensajes a través de una working queue donde todos los mismos nodos puedan tomar los datos y procesarlos on demand, pero simplifica enormemente el trabajo y los casos bordes que podemos encontrar.

Casos bordes

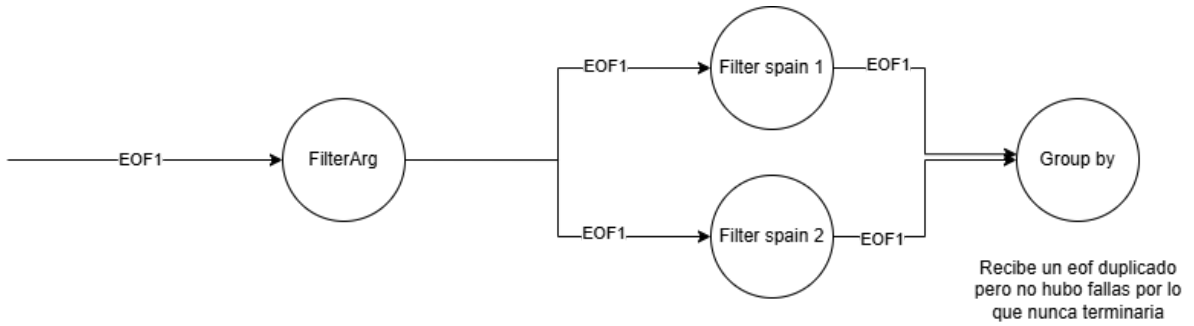
En esta sección veremos distintos casos bordes que fuimos encontrando en el tp y como es que los mitigamos, contaremos los casos de nodos con estado ya que los filtros o el machine learning simplemente enviaran el mismo mensaje al próximo nodo que tengan delante y ese próximo nodo será el encargado de verificar duplicados si lo necesita.
Vamos a empezar explicando un caso borde que manejamos solo bajo una condición, y es importante que quien use nuestra aplicación lo sepa.
El caso borde ocurre cuando se toma un dato de la queue de rabbit de un nodo que guarde estado, y el siguiente a ese dato es un EOF y nuestro nodo muere con ese mensaje sin ackear. Si el programa se corre con la opción de variable de entorno de CLI_WORKER_MAXMESSAGES=1, este caso no hay problema ya que simplemente se va a reencolar el mensaje al inicio y lo vamos a poder manejar.

Ahora, si se usa un valor de la env var mayor a 1, estaríamos haciendo que el programa sea mucho más rápido, pero puede ocurrir la posibilidad que tomamos por ejemplo 5 mensajes, luego tomamos el eof, y nuestro nodo muere antes de guardar el estado y ackear los mensajes. Si esto ocurre, rabbit no nos asegura que los mensajes vuelvan a tener el orden de primero los 5 mensajes y al final el eof, puede ser aleatorio, por lo que aunque es una forma de que el programa corra mucho más rápido, no nos asegura una resistencia completa a los fallos siempre que el prefetch sea distinto de 1. Se dejó la opción de utilizar el programa de esta manera, pero a riesgo del usuario.
Vamos a nombrar los casos bordes que encontramos, pero en las explicaciones de los siguientes nodos está cómo manejamos estos casos.
Si se reciben mensajes con ids repetidos está manejado, también el caso de eofs repetidos y si los nodos groupers reenvían mensajes que no deberían por no haber guardado el eof al final

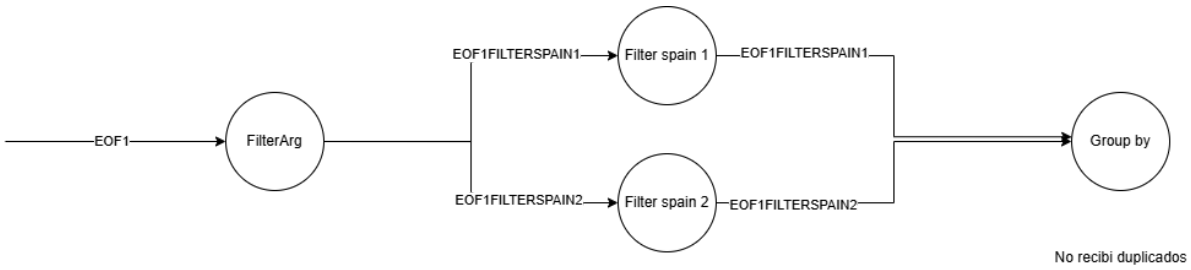
Nodos sin bajada a disco

Explicaremos en esta parte los nodos que no guardan ningún tipo de estado, siendo estos los filtros y el machine learning.
Empezando por los filtros, estos cumplen la interfaz de Filter que les da la función RunWorker, la cual va a leer los mensajes de la cola de rabbit, si es un mensaje de datos va a llamar la función de filtro que depende de la implementación de cada filtro en particular y luego va a enviar el mensaje. Lo que hay que tener en cuenta es que el mensaje que estamos procesando lo estamos guardando en una estructura “Hasher”, esta se encarga de hacer el hasheo por el id de película y mantenerla, luego cuando queramos enviar el mensaje cuando ya se filtro el batch, llamaremos a GetMessages del hasher y enviaremos a las routing keys los mensajes que nos diga esta estructura. Si el mensaje es un EOF, se envía el EOF a todos los nodos siguientes que tengamos con su message_id. Excepto el filtro por argentina, que tiene una función de envío de los eofs diferente a los demás ya que es el único filtro conectado a otros filtros. Si nosotros enviamos el eof de id=1 a todos los filtros de españa, todos recibieron este eof con id=1 y volverían a reenviar este eof de id=1 ahora a todos los group bys siguientes, esto generaría mensajes duplicados para la vista de los group by, por lo que este nodo además del id de mensaje del eof, le agrega la routing_key del nodo al que está por enviarle.

Caso si no se utiliza la routing_key para el id



Solucion



El machine learning funciona de la misma manera que un filtro normal, solo que en vez de filtrar mensajes por alguna función particular, le calcula su sentimiento si es positivo o negativo. El manejo de EOFs es idéntico a un caso normal de filtro

Nodos con bajada a disco

En este apartado se explicará todo lo relacionado a los nodos con estado interno, siendo estos los group-by, master-group-by, topN y joins.

Funciones genéricas

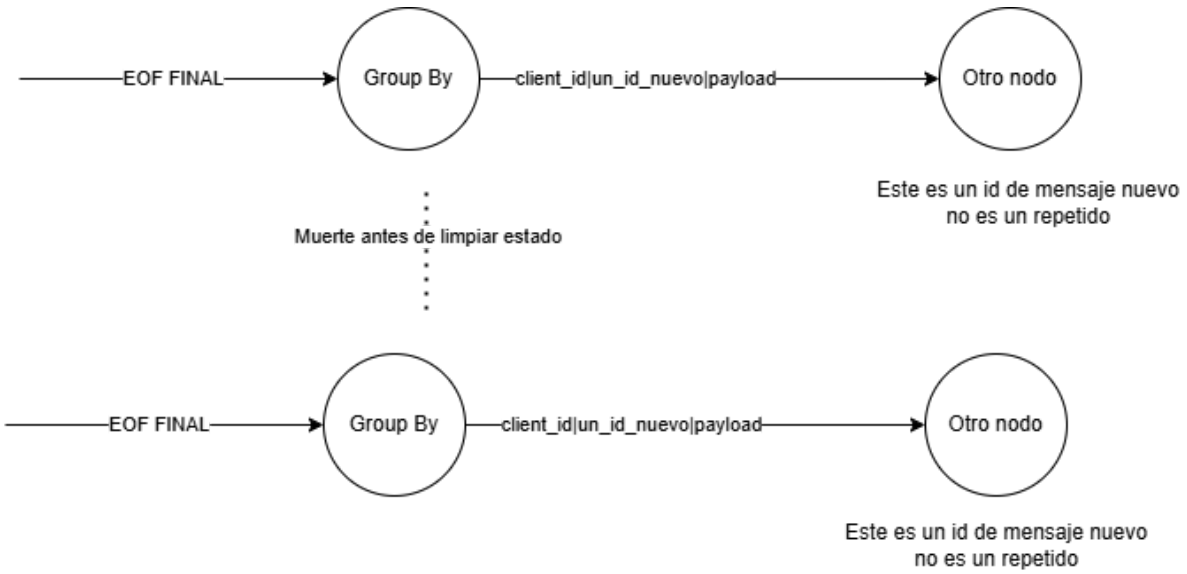
Para empezar, explicaremos las funciones genéricas que utilizan los distintos nodos para bajar y leer de disco

- ‘genericStoreElements’, su funcionamiento es el siguiente. Primero, creamos un archivo temporal en el cual escribiremos todo el estado que tenemos, para esto se decidió guardar en formato json, ya que todos los nodos guardaban su estado en un hashmap o alguna estructura que podía pasarse a hashmap. Luego, se escribe algún mensaje adicional si es necesario con la función de ‘after_write_function’, (que luego explicaremos el caso en el que es utilizada), se hace un flush de lo que se escribió y se utiliza la syscall rename para comitear el archivo, ya que esta es atómica, por lo que estamos seguros que si algo se escribió con el formato de archivo de client_id_commited.txt, sabemos que va a tener un estado valido guardado.
- ‘genericWriteToFile’ esta función simplemente escribe a un archivo con ciertos flags, su uso es en 2 partes. Cuando debemos appendear ids de mensajes y para escribir ids del nodo (se explicará a continuación el por qué). Algo para notar es que el archivo de ids appendeados no lo borramos cada cierta cantidad de mensajes, ya que en las pruebas que realizamos la cantidad de mensajes que se envían es bastante baja, ya que todos los datos que pasan a group bys pasaron por filtros previamente que reducen la cantidad de mensajes.
- ‘genericGetElements’, recorre todos los archivos en el directorio especificado y lee los datos. Utiliza el nombre del archivo para obtener el id del cliente y genera el estado como un hashmap de los ids de clientes y su estado que fue escrito como json. Además, si encuentra el mensaje como LAST_ID al final, devuelve la lista de los últimos mensajes que se guardaron en ese archivo para ese cliente.

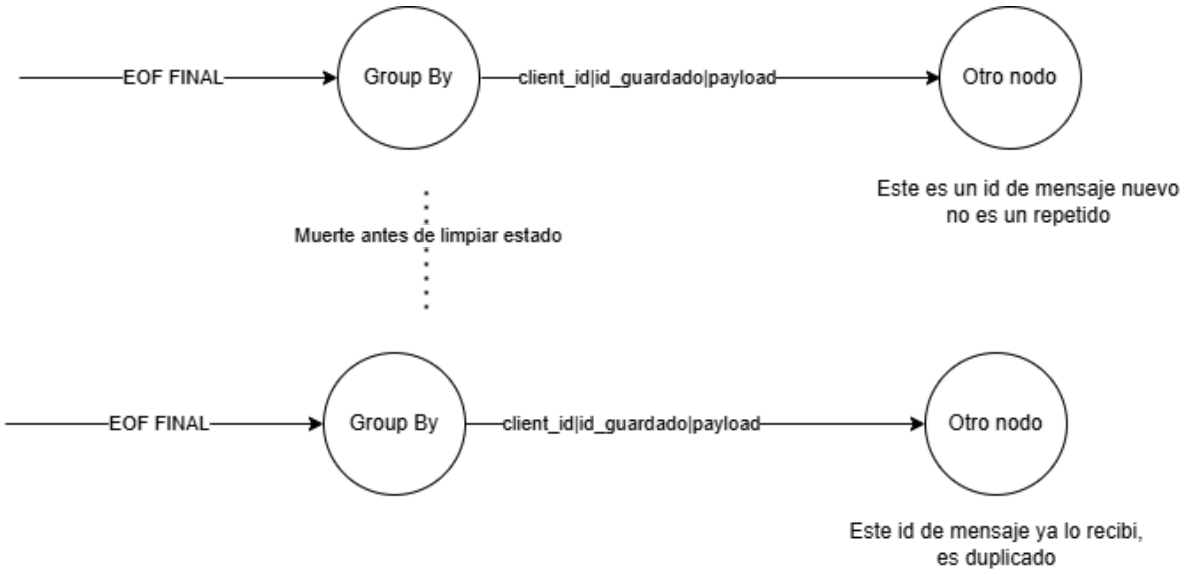
- ‘genericCleanState’ es la función para borrar todos los archivos de algún cliente en particular de un directorio.

Empezando por los group-by y master-group-by, éstos al levantarse inicialmente sin ningún estado, van a empezar a recibir mensajes por la función RunWorker ya que cumplen con la interfaz de StatefullWorker. Aca se van a empezar a recibir mensajes, lo primero es separar el mensaje en sus 3 partes y verificar que exista o no el cliente, si es la primera vez que envía un mensaje se le va a asignar su estado interno y se va a generar el id de mensaje que enviara al final el group by y se guarda en el estado. Es necesario guardar esto en estado ya que los group by envían 2 mensajes “nuevos”, no tienen un id que reutilizar de mensajes anteriores, por lo que sí generan uno nuevo cada vez que envían un mensaje para ese cliente, si el nodo muere en el momento justo luego de enviar el mensaje, cuando vuelva a intentar enviar va a ser un mensaje con un id distinto y el nodo que esté siguiente no lo va a poder detectar como duplicado.

Caso posible si no se guarda el id del nodo, generando un problema de mensaje duplicado sin manejar



Solución implementada



Siguiendo con el RunWorker, se verifica si el mensaje es un EOF o no, empecemos por el caso que no es un eof si no un dato a guardar.
Si es un dato a guardar se va a llamar a la función UpdateState, en la cual cada nodo va a guardar en su estado interno ese mensaje siempre y cuando ese mensaje en su message_id no sea uno que tenga guardado, si es asi simplemente lo va a descartar y hackear. Luego de guardar estado interno, se va a llamar al método HandleCommit, el cual primero va a guardar el mensaje de rabbit y lo va a guardar en su lista de mensajes sin ackear y sumar en uno el contador de mensajes recibidos. Si se llegó a la cantidad sin ackear permitida, se va a guardar el estado a disco junto con los últimos ids de mensajes guardados, y luego se appendean los ids. Cuando se termina esto, se ackean todos los mensajes guardados. Si el mensaje por el otro lado era un EOF entramos a la función handleEOF, la cual primero va a verificar si el eof recibido esta duplicado, si no lo esta se va a agregar el id de ese eof en el estado interno en memoria de ese cliente, luego se guardan los eofs y el estado interno hasta el momento junto con el id de ese eof recibido y se ackean todos los mensajes para ese cliente. Si el eof recibido es el ultimo eof esperado, se va a guardar el estado a disco, se van a ackear todos los mensajes y finalmente se va a enviar el resultado final y limpiar el estado interno. Si llega a morir en el ultimo ack, como este ack no se guarda a estado, se va a recibir devuelta, se va a enviar todo y luego se va a limpiar como debería. El nodo de adelante se va a dar cuenta del mensaje repetido por lo que tenemos al inicio de setear un id al mensaje para ese cliente apenas lo recibimos.
Si un group by muere, va a levantar todo su estado con las funciones GetElements para obtener el estado interno que le pertenece a cada cliente y los últimos mensajes que se recibieron de cada cliente, luego se obtienen los ids y los eofs. Se hace una verificación de restaurar el estado de los appends si es necesario (es decir, se escribió el ultimo mensaje a estado pero no se llegó a appendear antes de morir) y si hubo algún cambio, se actualiza. También se intenta obtener el id propio de envío de mensajes para los clientes y finalmente se puede iniciar el proceso.

Siguiendo con los topN, éstos tienen un manejo mucho mas sencillo. Utilizan la misma interfaz que los group by por lo que el orden es idéntico. Al recibir un mensaje siguen el mismo camino de crear un id único para ese cliente, cuanto actualizan su estado tienen una

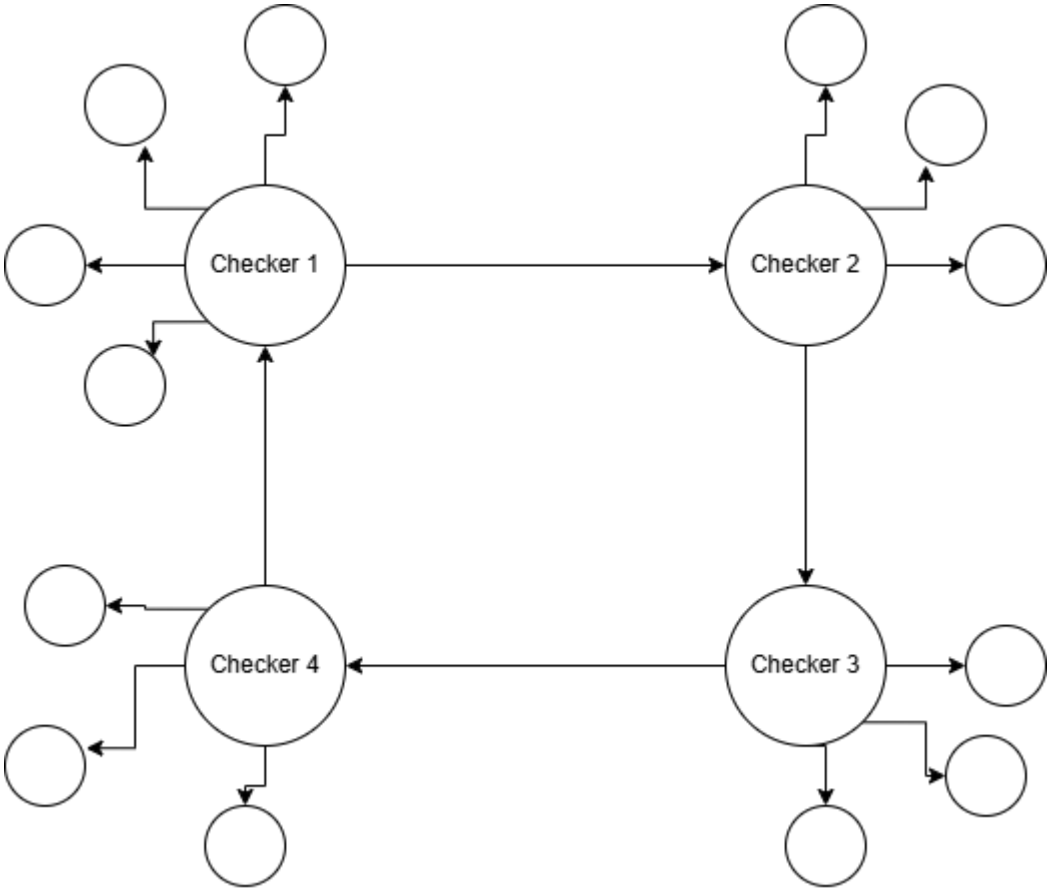
verificación propia de mensajes repetidos que no es por id, si no por el propio estado en sí, ya que claramente no puede estar 2 veces el mismo dato en un top, ya se hacía una verificación de este dato sin tener que manejar ids. Otra diferencia es que los topN van a commitear a disco todo el tiempo, pero utilizando la función StoreElements, la cual no guarda los últimos mensajes, solamente el estado interno y tampoco al manejar eofs es necesario tener un contador, ya que todos los topN están enfrente de un master group by que les enviara un eof y estos enviaron su top al client handler. Si envían de vuelta el mensaje, como va a tener el mismo id por su seteo al inicio, el client handler se encarga de filtrar ese repetido recibido. Si un topN muere, va a tomar todo su estado interno y su id, no tiene que mantener la lista de ids de mensajes recibidos.

Finalizando con los joins, estos son un caso particular de worker que guarda estado, ya que la mitad de su estado lo necesita guardar siempre (movies) y la otra mitad solo si vienen en desorden (credits/ratings), pero ahora explicaremos en detalle el proceso.
Los joins, al igual que los otros nodos empiezan con la función RunWorker para ir recibiendo mensajes de rabbit, pero no de la interfaz si no una particular, ya que la lógica es distinta al recibir de 2 queues al mismo tiempo. Para empezar, cuando llega un cliente se va a crear su estado en memoria si no existía, pero en comparación a los group bys no necesita crearse un id de mensaje único, ya que los joins hacen pipelining, enviando los credits/ratings que reciben.
Nosotros en nuestro programa siempre enviamos primero el dataset de movies y luego el resto, por esto los joins las movies van a ser lo que primero recibieron (se tiene en cuenta los casos en lo que no ocurre, se explicará luego). Si el mensaje que se recibe es un dato, simplemente se va a guardar a el estado interno y luego se guardará en disco, pero sin ningún message_id, ya que los joins no les importa si les llega la misma película repetida, si esto ocurre actualizarán su dato por el mismo dato. Si lo que se recibe es un EOF de movies se va a guardar ese id de mensaje de eof y se flushea a disco, si el eof recibido fue el último, se van a enviar todos los datos que se hayan recibido en desorden, los llamaremos pendings.
Pero, ¿qué es un pending? Estos son datos recibidos por la cola de credits o ratings que llegaron antes de recibir todos los eofs de movies, así que si esto ocurre se va a guardar en el estado interno y luego se flushea a disco ese dato. No tenemos ninguna verificación particular de repetidos ya que los joins al enviar los datos funciona como si fuera un filtro.
Ahora, por el lado de la segunda cola cuando recibimos todas las movies, simplemente haremos la función de join correspondiente y enviamos cada dato que recibimos, no guardamos a disco, y si llega un EOF de esta cola, que va a ser solo uno por cliente, se va a enviar el EOF a todos los nodos siguientes y se va a limpiar el estado interno del join.

Nodos Health Checkers

Los nodos health checkers son un nuevo tipo que se tuvieron que implementar para la tolerancia a fallos. Estos reciben por una variable de entorno el nombre de los contenedores que se van a encargar de monitorear y cada cierto tiempo le envían un mensaje de “PING” por UDP a los nodos, si no responden luego de ciertos intentos con el mensaje “PONG”, se asume que esta caído el nodo y proceden a hacer docker start <nodo>. Los nodos por su parte tienen implementado un pequeño servidor UDP que se encarga de recibir el mensaje “PING” por un puerto particular y devolver el mensaje “PONG”. Algo para notar que los

nodos que verifican si se utiliza el archivo de generar compose, hace que cada health checker monitoree un subconjunto distinto de nodos así no hay solapamiento.



Versiones anteriores

Este apartado es para hablar de la implementación que pensamos inicialmente para la tolerancia a fallos y por qué decidimos cambiar a la actual.

La implementación que pensamos antes de terminar con la que tenemos actualmente era una utilizando working queues, en la que todos los nodos podían tomar datos de una cola, esto sería muy bueno para temas de performance ya que cada nodo toma al máximo la carga que pueda manejar pero nos encontramos con problemas con un nodo que toma un dato, luego otro nodo toma el eof, y muere el nodo que tomó el dato y se re-encola. Esto nos generaba problemas al desordenar los mensajes, por lo que al final optamos por la arquitectura actual en el que cada nodo tiene una cola propia.

Conclusiones

Al realizar este tp nos encontramos con muchísimos problemas que iban surgiendo, infinitos casos bordes y código complicado de entender, pero al pasar las semanas logramos armar

un programa que se levanta todo en docker, que se le puede lanzar una bomba apagando todos los nodos menos un checker y puede revivir, nos parece increíble lo que llegamos a lograr en estos meses de cursada, nos llevamos varias enseñanzas de este tp y conocimientos sobre la palabra clave de la materia: ‘Middlewares’
Muchas gracias por leer el informe, sabemos que tiene partes bastante “densas”, esperamos que el tp sea de su agrado