Sistemas Distribuidos I

TP Tolerancia a Fallas

Della Vecchia, Tomas

Grin, Pedro

Singer, Joaquín

Índice

Alcance

El trabajo consiste en que dado un dataset acerca de películas se busca resolver de manera distribuida 5 queries acerca del mismo.

Este dataset contiene las tablas:

1. movies_metadata que contiene información relevante acerca de películas, entre estos datos además del título y un id identificador, se encuentran los géneros a la que pertenece, el costo de la película, la ganancia, una descripción acerca de la película y los países de la producción entre otros datos.
2. ratings que contiene un valor entre 1 y 5 de calificación que un usuario le asigna a una película.
3. credits que contiene información acerca del elenco que formó parte de la película.

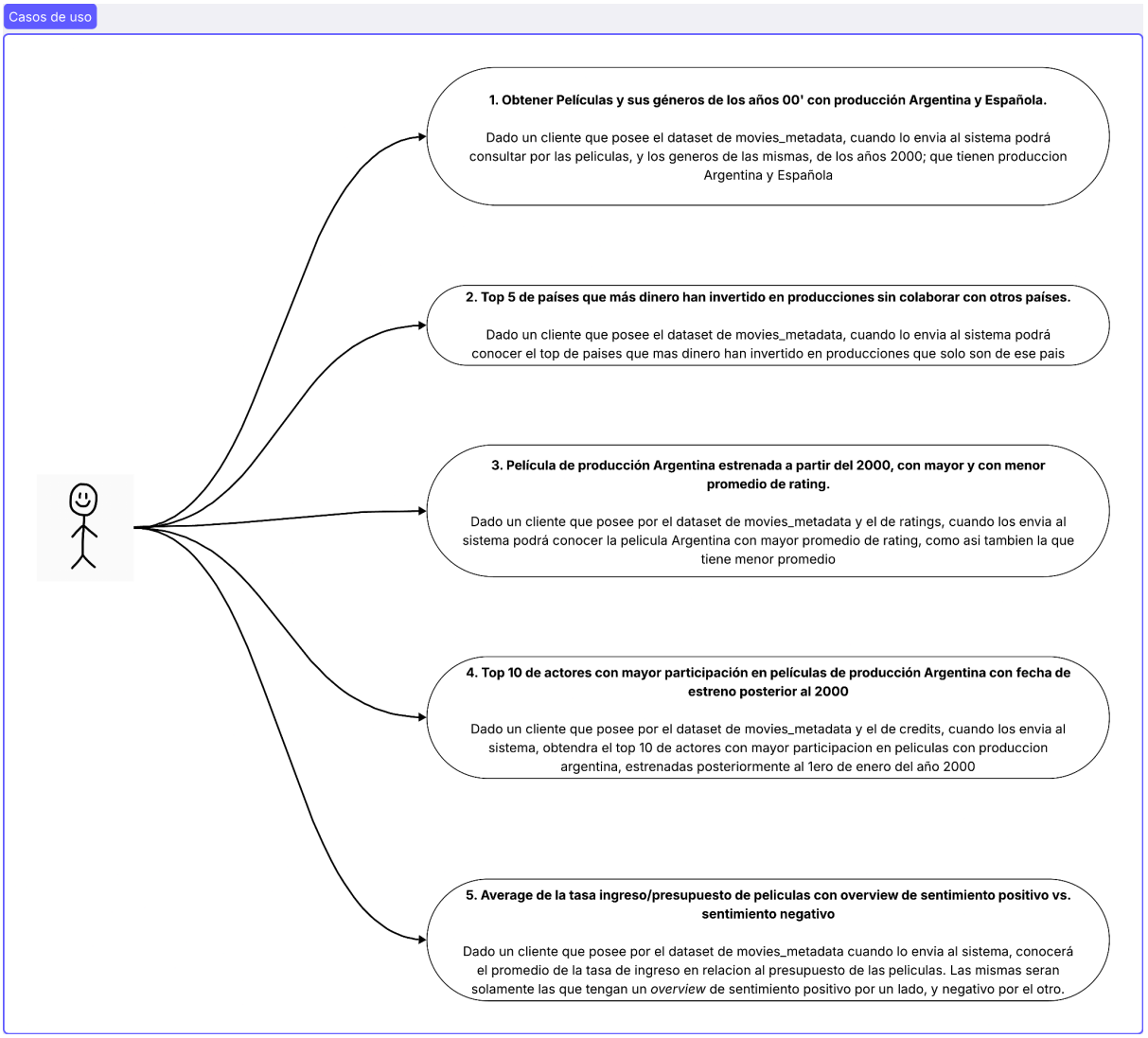
Con estos datos se buscan resolver las siguientes 5 queries:

1. Películas y sus géneros de los años 2000 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

Escenarios

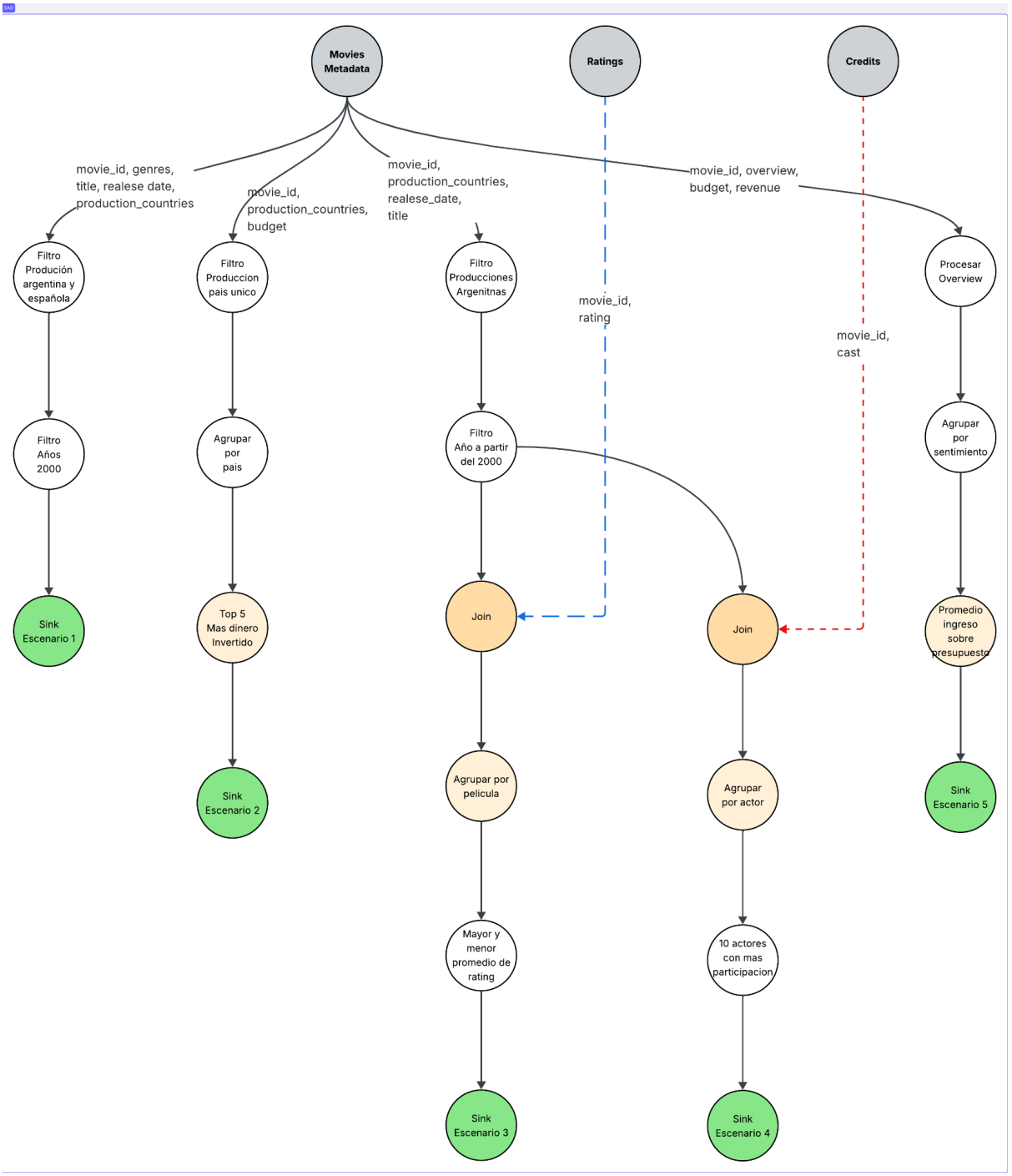
Diagrama de casos de uso

El diagrama de casos de uso incluye las 5 queries mencionadas anteriormente. Se espera que cuando un cliente realice una llamada al sistema proporcionando los datos de su interés reciba el resultado de las 5 queries.



Vista lógica

DAG



En el diagrama anterior (DAG, direct acyclic graph) podemos observar el flujo de datos del sistema. Como entrada tendremos las tablas previamente mencionadas, Movies Metadata,

Ratings y Credits. Cada una de las diferentes query reciben distintos datos de entrada provenientes de estas tablas, que se pueden observar en el diagrama.

A su vez podemos observar que tendremos diferentes operaciones a realizar como filtros, uniones, agrupaciones, promedios y ordenar según un top. Para las distintas queries la información irá atravesando distintos nodos, hasta producir el resultado esperado por el cliente. Además, cabe resaltar que no es necesario que la información atraviese todos los nodos del sistema, sino que dependerá de cada query.

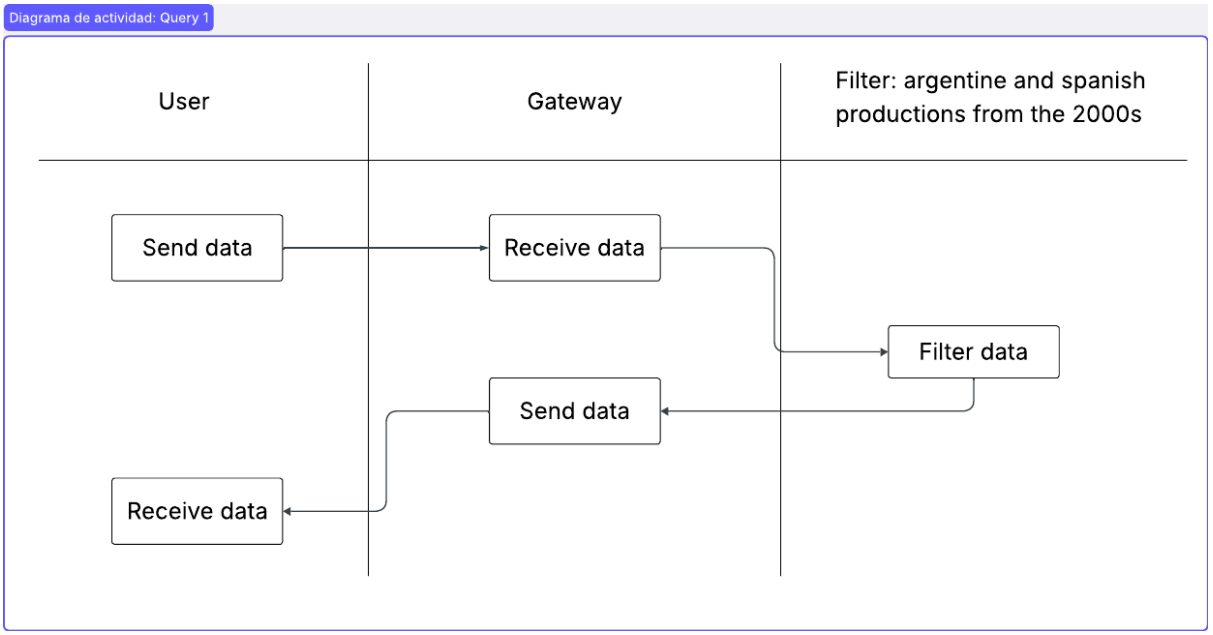
Por último, las operaciones coloreadas con color amarillo claro (top 5 más dinero invertido, agrupar por película, agrupar por actor, promedio ingreso sobre presupuesto) necesitan todos los datos previos para poder enviar la información al siguiente nodo, y por esto están coloreados de distinto color.

Vista de procesos

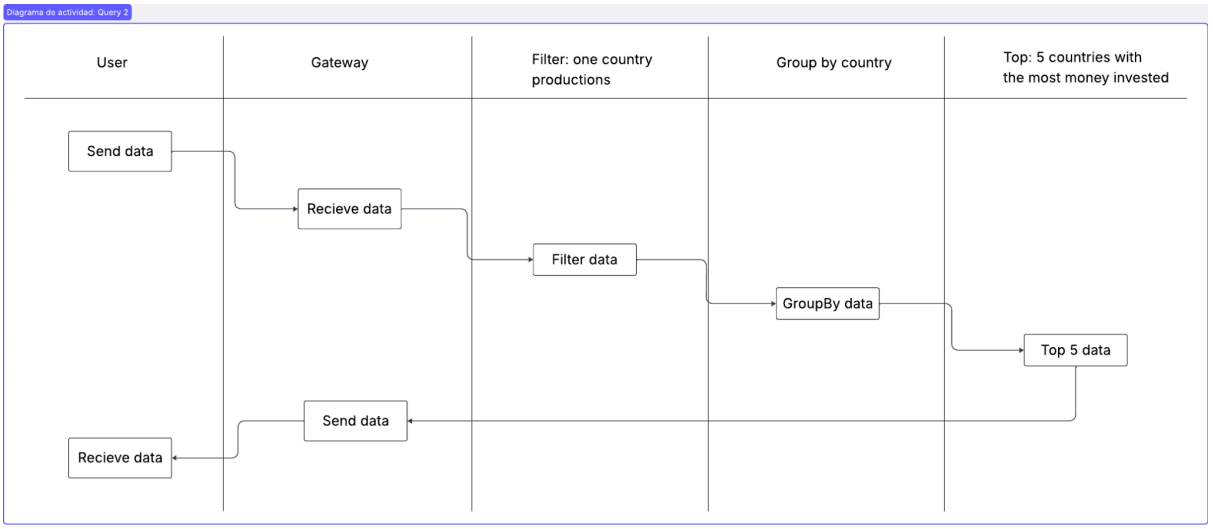
Diagramas de actividad

A continuación se observarán los diagramas de actividad de cada query en particular.

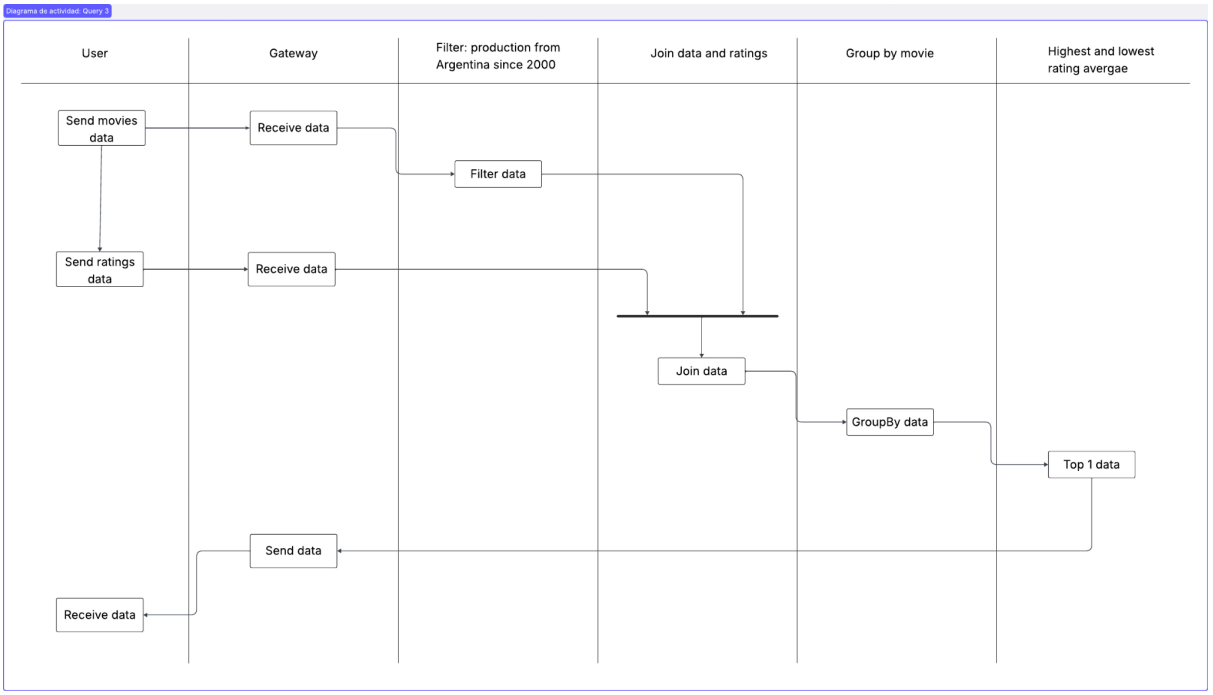
Query 1: Películas y sus géneros de los años 2000 con producción Argentina y Española.



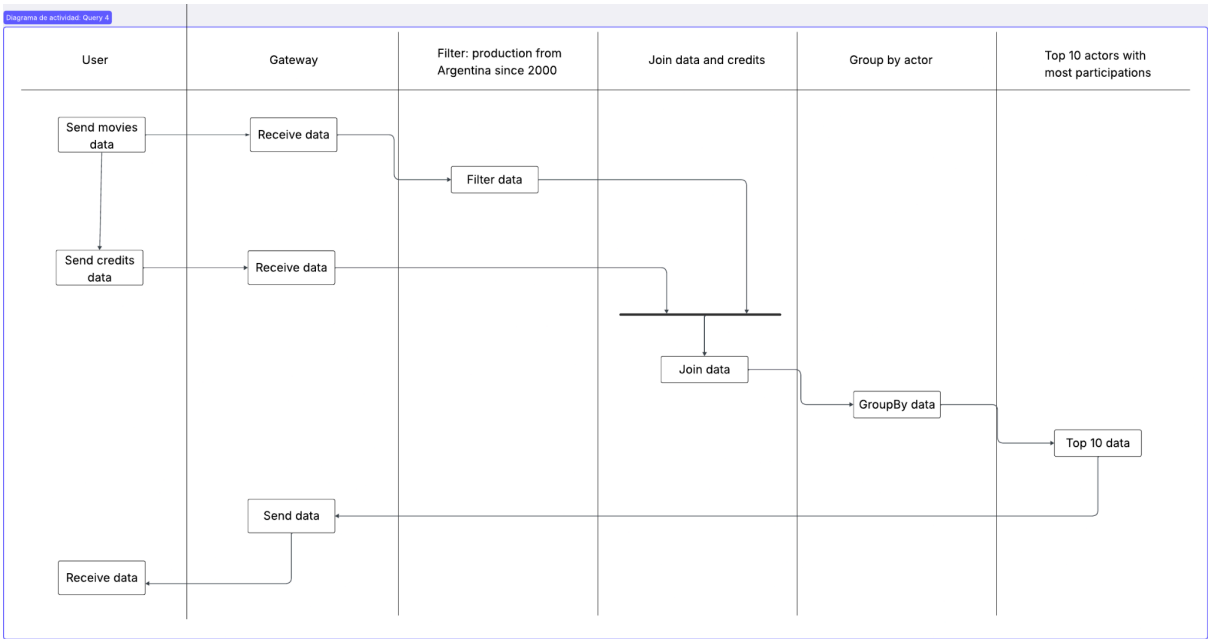
Query 2: Top 5 de países que más dinero han invertido en producciones sin colaborar con otros países



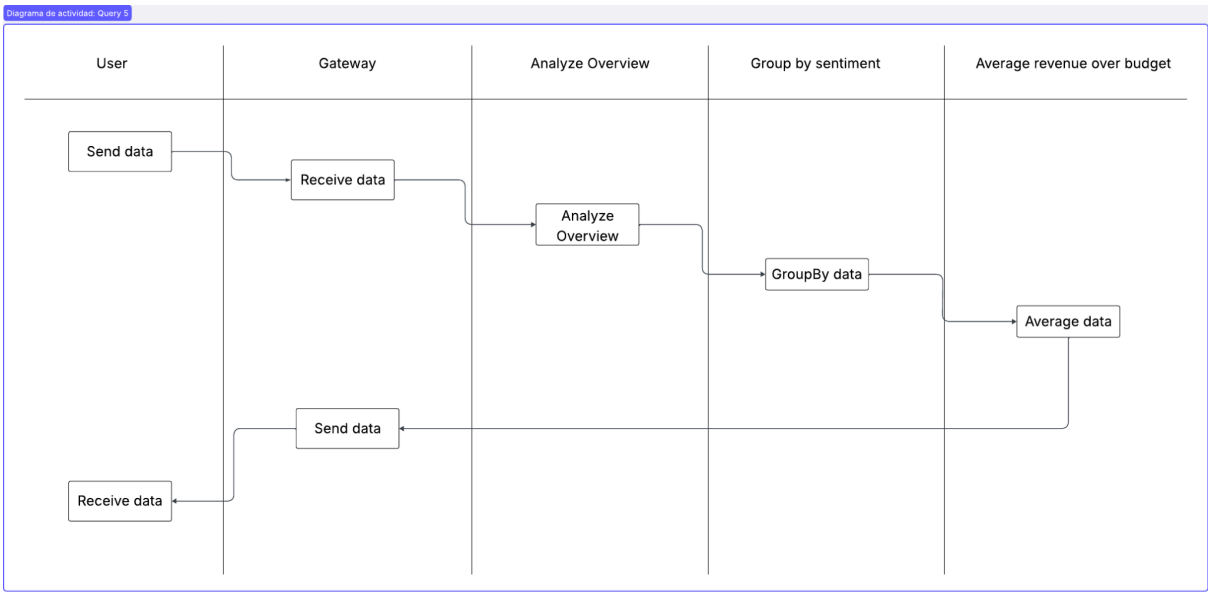
Query 3: Película de producción Argentina estrenada a partir del 2000, con mayor y con menor promedio de rating.



Query 4: Top 10 de actores con mayor participación en películas de producción Argentina con fecha de estreno posterior al 2000.



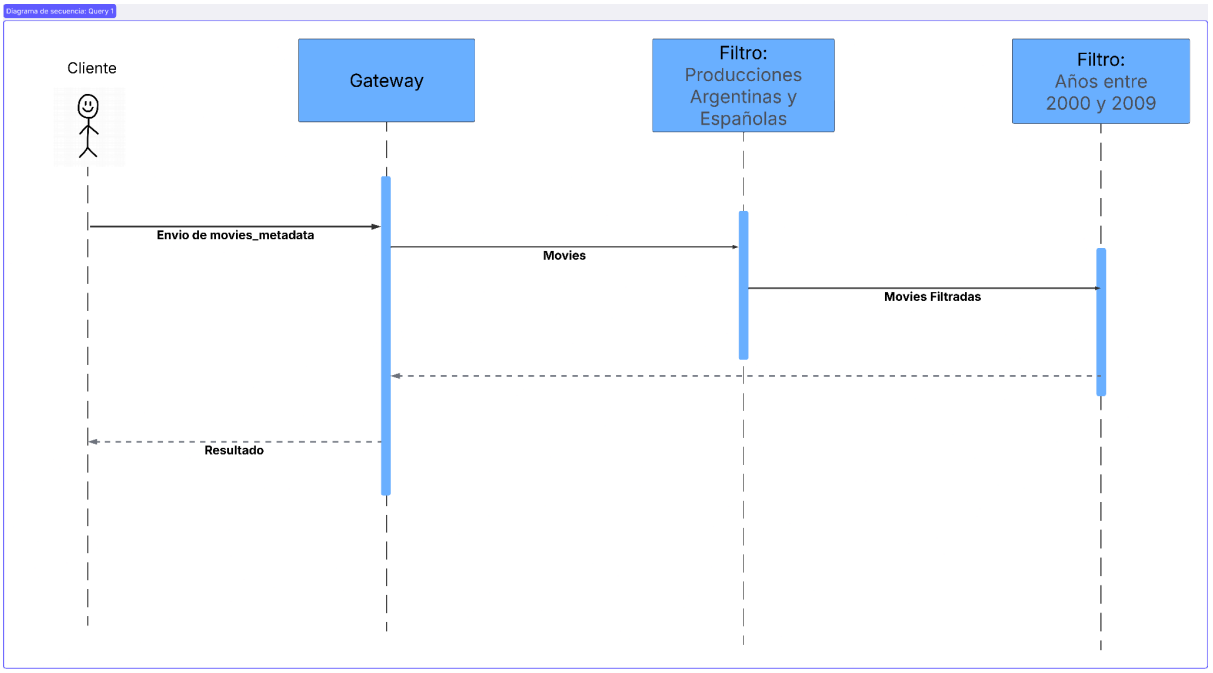
Query 5: Average de la tasa ingreso/presupuesto de películas con overview de sentimiento positivo vs. sentimiento negativo.

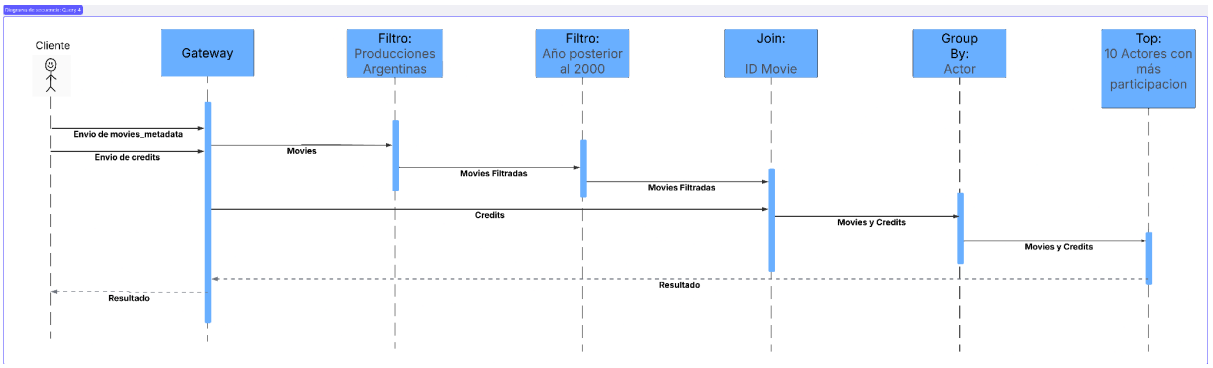


Diagramas de secuencia

Para evitar la redundancia, aquí detallamos sólo dos de los diagramas de secuencia, uno para la Query 1 (*Obtener Películas y sus géneros de los años 00' con producción Argentina y Española*) y otro para la Query 4 (*Top 10 de actores con mayor participación en películas de producción Argentina con fecha de estreno posterior al 2000*).

Cabe destacar que no se denota la parte del lado del cliente, sino que se toma un enfoque completamente desde el lado del servidor.

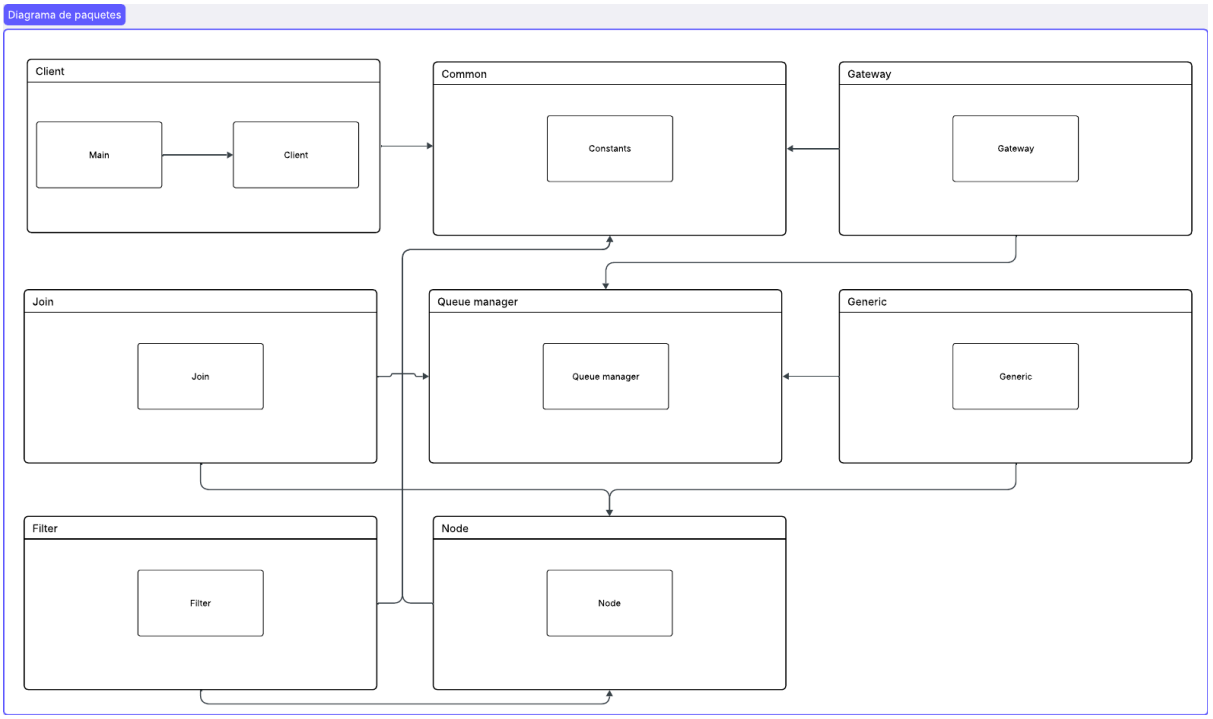




Como se observa en ambos diagramas, el cliente posee los datasets y comienza una comunicación con el servidor, más precisamente con el gateway. Ambos procesos, tanto cliente como gateway viven hasta recibir los resultados de la query.
Se pueden observar diferentes procesos, dependiendo de qué query se trata.
Exceptuando la comunicación entre el cliente y el gateway, que se comunicarán mediante TCP, todos los demás procesos se comunicarán mediante queues.

Vista de desarrollo

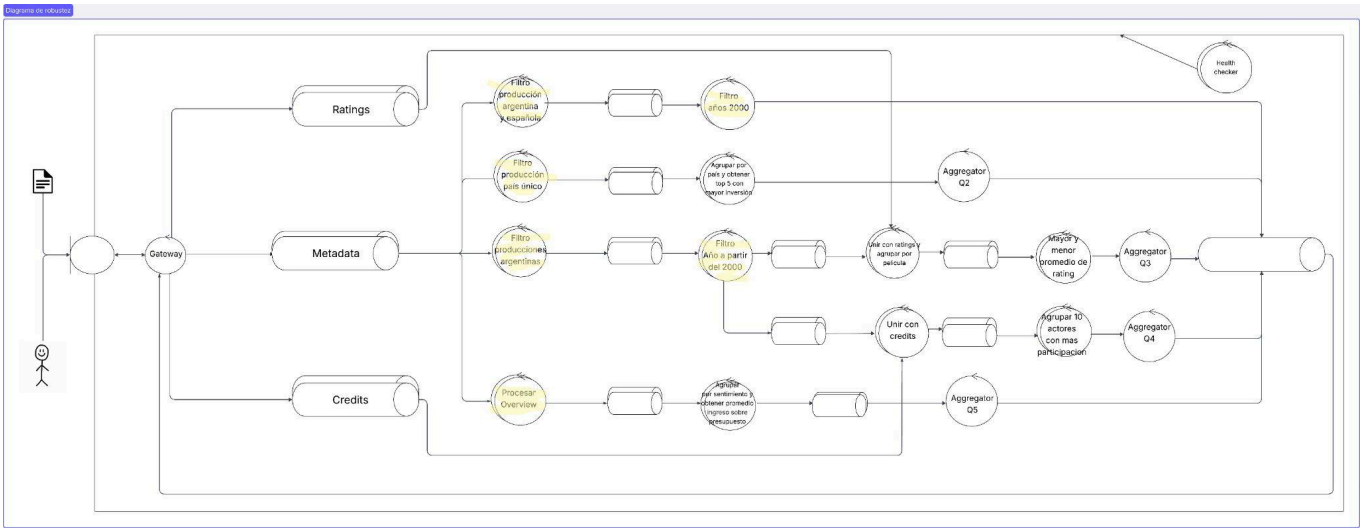
Diagrama de paquetes



En el diagrama se detalla la interacción entre paquetes del sistema. Estos son: Client, Common, Gateway, Join, Queue manager, Generic, Filter y Node.
El paquete Client modela al cliente del sistema y su comunicación con el servidor. En cuanto al servidor el paquete que interactúa con la conexión con el cliente y da inicio a las queries es el Gateway.
El paquete Queue Manager es una abstracción para el uso de RabbitMQ. Los paquetes que interactúan con él son Node y Generic.
Node es un paquete que modela el comportamiento general de un nodo del sistema, por este motivo los nodos de Filter, Join y Generic lo utilizan.
Filter, Join y Generic son paquetes que sirven para modelar el comportamiento de un nodo puntual del sistema. Tanto Filter cómo Join existen más de uno, por lo que es un paquete aparte. En cambio, Generic sirve para modelar nodos que son únicos dentro del sistema (por ejemplo el procesamiento de overviews de la query 5).

Vista física

Diagrama de robustez



El siguiente diagrama muestra la robustez del sistema y su escalabilidad. Se pueden observar los nodos que participan en el sistema, y las queues que en algunos casos son múltiples.

Podemos observar que el cliente enviará la información al gateway que determinará en qué queue almacenar los datos dependiendo de qué tabla provengan.

Cabe destacar que los nodos de color amarillo son los que no tienen ni persisten ningun estado personalizado.

Luego, para las query 1 a 4 habrá una etapa de filtrado de los datos relevantes para la query. En cambio, en el último caso habrá una etapa de procesamiento de lenguaje natural para determinar si la overview de una película es positiva o no.

Para la query 1, luego de este paso ya se obtiene el resultado y se lo almacena en una queue final que acumulará los resultados de las diferentes queries.

Para la query 3 y 4, se realizará un join con la tabla correspondiente en cada caso. En su momento analizamos la alternativa de realizar primero la agrupación sobre las tablas ratings y credits dependiendo el caso y finalmente el join, pero creemos que será mejor hacerlo a la inversa ya que aprovechamos los datos ya filtrados para reducir la cantidad de información que tenemos. A comparación del diagrama inicial, para la entrega de escalabilidad nos dimos cuenta que podíamos aprovechar el join que se estaba realizando para agrupar, por

lo que estas operaciones se juntaron en un único nodo, que luego de pasar por nodos de top se devuelve el resultado final.

Para el resto de las queries se realiza directamente la agrupación junto con diferentes operaciones por el mismo motivo previamente mencionado, y así devolver el resultado a la queue final.

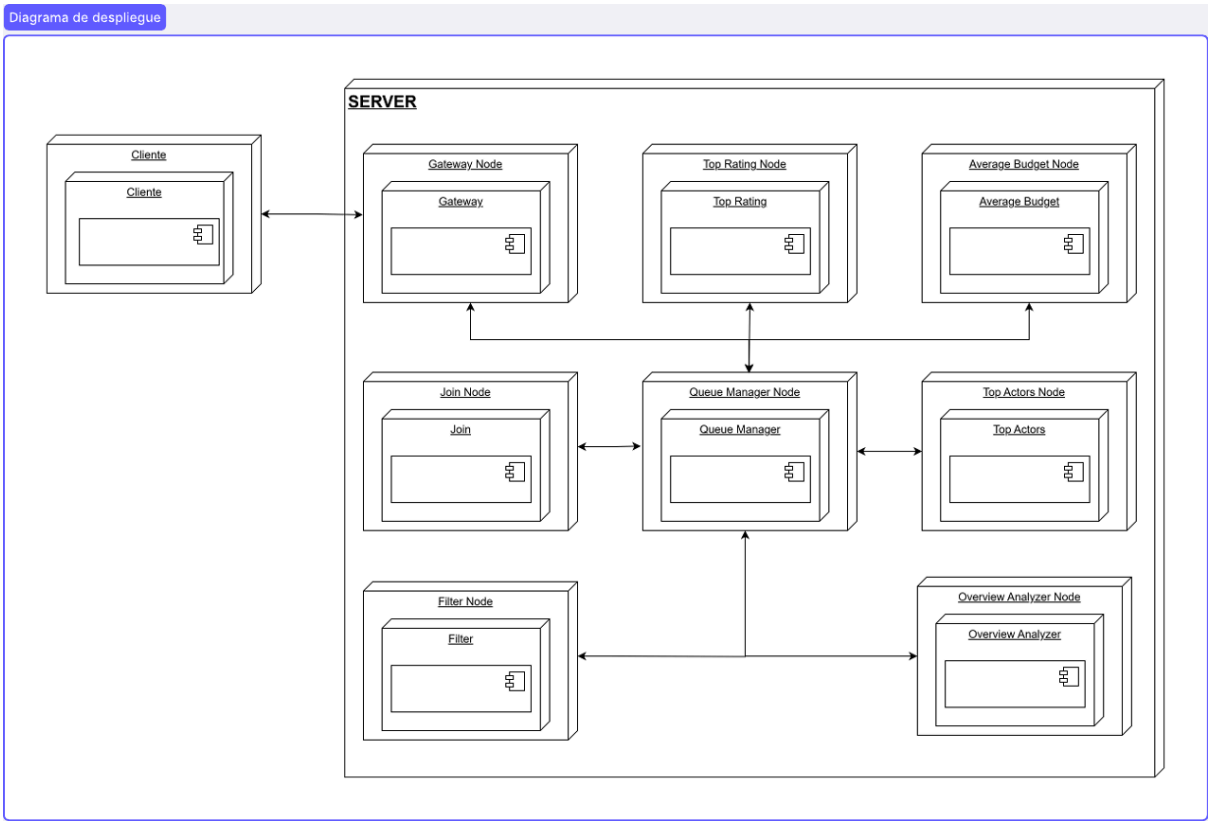
Para la entrega de escalabilidad consideramos agregar que los nodos finales (nodo anterior a aggregator) de las queries 2, 3, 4 y 5 sean escalables, pero al ser un único cliente creímos que no sería necesario.
En cambio, para la entrega de multi-client consideramos que sería mejor que estos últimos nodos sean escalables y agregar un nodo de aggregator como final de cada query. De esta forma, al poder existir la posibilidad de muchos clientes ya todos los nodos son escalables, lo que nos permite tener un sistema más robusto.

Para los nodos de aggregator tuvimos en cuenta el siguiente razonamiento:

- Para la query 2 van a llegar N x 5 resultados
- Para la query 3 van a llegar N x 2 resultados
- Para la query 4 van a llegar N x 10 resultados
- Para la query 5 van a llegar N x 2 resultados

Siendo N la cantidad de nodos deployados de cada nodo final previo al aggregator y esto multiplicado por M clientes, no sentimos que valga la pena que estos nodos aggregator sean escalables, ya que en cada caso será: M x N x 2 ó 5 ó 10 resultados.

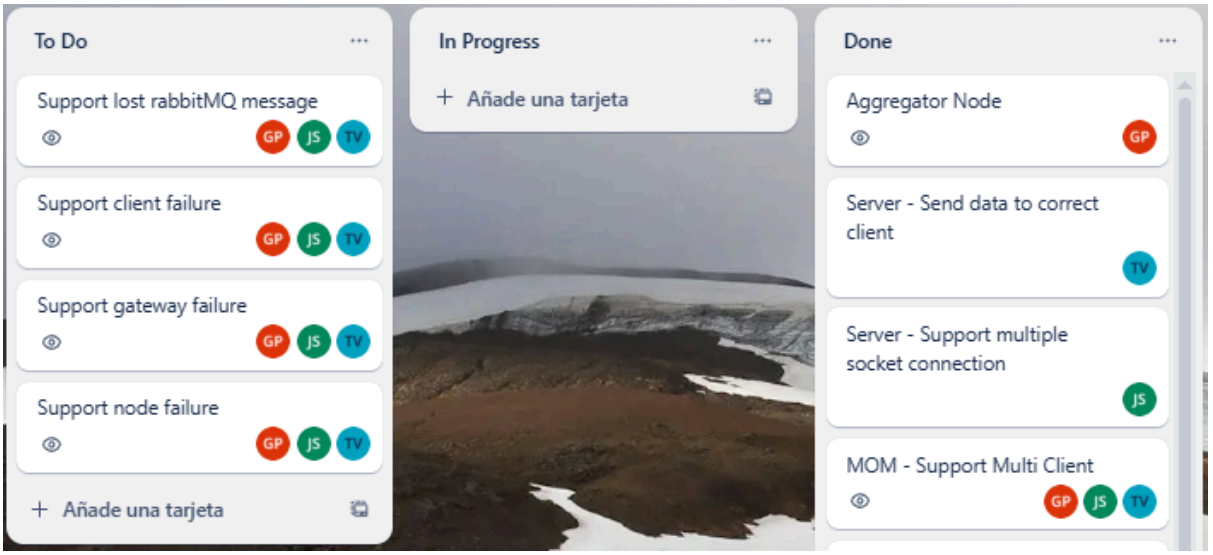
Diagrama de despliegue



En el diagrama de despliegue podemos observar todos los procesos del sistema. El cliente se comunica al servidor a través del gateway, mientras que los procesos internos del servidor se comunican entre ellos a través de RabbitMQ utilizando el Queue Manager Node.

Tareas

Para la división de tareas utilizaremos la herramienta Trello.
En esta herramienta se ven asignadas las actividades y los usuarios asignadas a estas, donde los usuarios se identifican por sus iniciales. JS: Singer, Joaquín. GP: Grin, Pedro. TV: Della Vecchia, Tomas.



Estas son las tareas pendientes para la última entrega, que irán siendo modificadas seguramente a medida que nos adentramos en la solución. En la sección de Done únicamente se ven las tareas realizadas para la entrega de multi-client, ya que sino la lista se volvía muy larga para insertar en este documento.

Tolerancia a fallos

Health Check

Se implementó una serie de nodos destinados a monitorear el estado del sistema mediante el envio de mensajes por UDP

Los nodos a monitorear cuando comienzan tienen otro proceso que se conecta a un health checker mediante UDP y cada cierto tiempo le mandan un heartbeat. El health checker al que le mandan el heartbeat está definido en el docker compose, pero cada health checker tendrá encargado algunos nodos, así se distribuye el trabajo entre health checkers (en el caso que haya uno solo tendrá que supervisar a todos logicamente).

Por su parte los health checker tienen dos procesos por detrás. Un primer proceso está en loop recibiendo todos los heartbeats de los nodos que controla. Y otro proceso que revisa que un todos los nodos se hayan comunicado antes de cumplir con un timeout.

Para el caso en el que se detecte que un nodo dio timeout se lo reinicia. Esto es seguro ya que un nodo sólo se comunica con un health checker evitando que exista un reinicio de más de un health checker al nodo que dio timeout.

Además, los health checker se monitorean entre sí. Para esto cada health checker conoce su id y conoce cuántas instancias de health checker existen.
Por ejemplo ID = 1, HC_SIZE = 3. Inicialmente cada HC conoce cuales otros hay y espera recibir heartbeats de estos (los HC mandan heartbeats a sus otras réplicas cuando chequea que ningún servicio se haya caído).
Cuando se chequea si algún servicio dio timeout también se chequean los HC pero como estos son servicios que pueden ser monitoreados por más de un HC (cada HC monitorea a todos los otros) hay que tener un cuidado diferente. Si se detecta que un HC se cayo se envía un mensaje a todos los otros HC avisandoles eso y que lo va a reiniciar. Para evitar tener una condición de carrera, se definió que el HC con id más chico sea el encargado de los reinicios. Entonces si un HC con ID 2 se cae, todos los HC van a detectarlo y enviar el mensaje de que lo van a reiniciar. Pero como el HC con ID 1 es el mas chico, cuando el HC con ID 3 reciba este mensaje va a dejar de ser el encargado de reiniciarlo. El mensaje de quien reinicia a quien se manda varias veces hasta llegar un timeout, para asegurarse que todos los otros HC tengan en claro quién lo va a reiniciar. Cuando se cumple el timeout el HC 1 lo reinicia.

Además, el HC que no se encarga de reiniciarlo va a saber que se reinició correctamente cuando el HC antes caido envíe un nuevo heartbeat. En caso de que luego de 2 * timeout no se reciba el heartbeat del HC caido el proceso comenzará de nuevo para decidir nuevamente quien lo va a reiniciar. Esto es para handlear un caso borde en el cual por ejemplo el HC 3 se cae, todos los otros HC saben que el HC 1 lo reniciara ya que les llega su aviso pero antes de reiniciarlo el HC 1 se cae también, de esta forma luego de 2 * timeout se asume que el HC 3 no se reinicio correctamente y los HC vivos comenzarán nuevamente la búsqueda de quién será el HC encargado de reiniciarlo, así a la larga se reinicia el HC 1 y el HC 3.

Otro punto importante es que los HC en sus mensajes entre si se comparten los procesos que monitorean y los procesos que saben que el otro monitorea. Esto es para que cada HC vaya construyendo quienes son los procesos que los otros HC monitorean, y si se cae un HC y antes de que se reinicie se cae un proceso que monitorea ese HC caído cuando se reinicie el HC los otros HC le compartirán cuales eran los procesos que monitorea, y finalmente detectara que el proceso caído luego de el supero el timeout y asi puede reiniciarlo.

Con estas consideraciones, el sistema siempre termina reiniciando a todos los nodos (incluidos HC) mientras al menos un HC siga vivo.

Caída de Clientes

La detección de una caída de cliente es responsabilidad exclusiva del **gateway**. Este componente es el encargado de identificar un *timeout* en la comunicación con un cliente, y, al hacerlo, propagará un mensaje a través de todas las *queues*, informando del evento. Este mensaje contendrá el **ID del cliente** caído.

Cada nodo que reciba dicho mensaje deberá:

- Eliminar cualquier información en su estado relacionada con ese cliente.
- Registrar el ID del cliente en un diccionario, con el fin de **ignorar cualquier mensaje futuro** proveniente de dicho cliente.
- Propagar el mensaje al siguiente nodo.

Por su parte, el **gateway**, tras enviar el mensaje en las tres *queues*, eliminará cualquier resultado previamente recibido del cliente y descartará todos los mensajes posteriores que lleguen a la *queue* de resultados relacionados con ese cliente.

Mensajes repetidos

Los mensajes viajan con un ID de mensaje, el cual es generado automática e incrementalmente, para evitar la repetición del procesamiento de mensajes en caso de que llegue un mensaje duplicado frente a una eventual caída de un nodo emisor.

Además, el ID está acompañado de un identificador del nodo que envió el mensaje, entonces el receptor puede almacenar el último mensaje recibido de cada nodo y así detectar como duplicado uno con el mismo ID o anterior (ya que el orden siempre se preserva).

Los filtros no necesitan detectar duplicados, y los joins solo necesitan detectarlos realmente para credits y ratings. La parte de metadata simplemente ignora una película cuando ya había registrado ese ID.

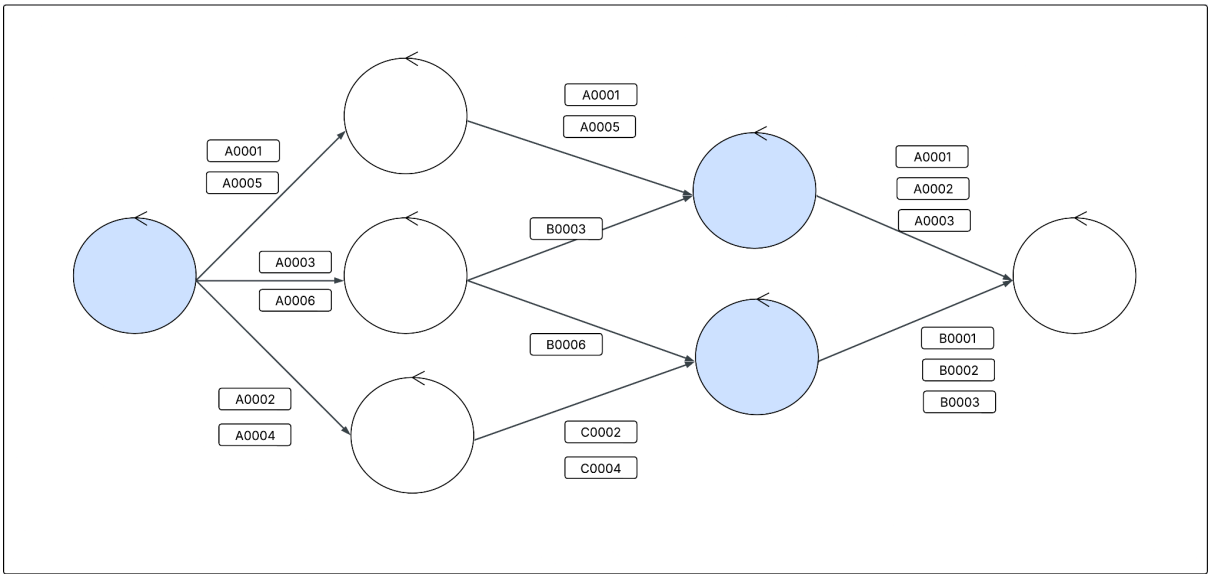
El gateway garantiza envío ordenado con IDs incrementales, y luego cada nodo que filtra duplicados, emite un nuevo ID, esto no es un problema ya que en todos esos casos donde hay que hacerlo, el envío de mensajes se produce al terminar de procesar la data.

El gateway detecta duplicados si recibe dos mensajes iguales para un mismo cliente, ignorando simplemente esos casos.

Casos por query:

Q1 → Son solo filtros, así que con detectar duplicados en el gateway alcanza.
Q2 → El primer filtro no necesita detectar duplicados, y propaga el mismo ID que recibe. Hay detección de duplicados y regeneración de IDs en el group by/top, y luego detección nuevamente en el aggregator.
Q3y4 → Los filtros no necesitan detectar duplicados y el join ignora películas repetidas, solo detecta duplicados de credits y ratings, que vienen con su ID incremental del gateway, luego regenera IDs al enviar la data. Los tops/group by detectan y regeneran, y el aggregator detecta.
Q5 → El overview processor no necesita detectar duplicados, y propaga el mismo ID que recibe. Hay detección de duplicados y regeneración de IDs en el group by/average, y luego detección nuevamente en el aggregator.

Un potencial problema que detectamos era que si propagamos IDs por más de dos nodos, hay riesgo de que se desordenen cuando no haya coincidencia de cardinalidad de nodos. Afortunadamente esto no es un problema ya que el único lugar donde podía ocurrir era en los filtros de las querys 3 y 4, pero al ignorar las películas repetidas en el filtro, se pueden ignorar esos IDs.



Nuestros escenarios terminan siendo como los de este modelo, donde los nodos azules regeneran ids, y todos los nodos mandan su identificador con el ID. Si el primero (el gateway) envía de forma ordenada los IDs a la siguiente etapa, estos nodos pueden utilizarlos y por más que haya cambio de cardinalidad descendiente, mientras la tercera etapa regenere IDs nunca se perderá el orden.

Persistencia

Todos los nodos cuentan con un volumen de almacenamiento persistente para guardar su información. Cada nodo persiste **dos estructuras de datos clave** para su funcionamiento. Estas estructuras se guardan en archivos individuales.

1. Diccionario de clientes detectados como caídos

- **Clave**: ID del cliente.
- **Valor**: Timestamp correspondiente al momento en que el nodo recibió el mensaje de caída.

Este diccionario permite, con el tiempo, limpiar los registros obsoletos, bajo el supuesto de que ya no se recibirán más mensajes de ese cliente.
Se persiste en disco **únicamente la primera vez** que se recibe el mensaje de caída para cada cliente.

2. Diccionario de clientes con procesamiento finalizado

- **Clave**: ID del cliente.
- **Valor**: Lista de *binds* por los que se ha recibido el mensaje de finalización de procesamiento.

Este diccionario se limpia automáticamente cuando se han recibido todos los *binds* esperados del cliente, o también cuando se detecta que el cliente ha caído.

Además, cada nodo puede tener estructuras adicionales según su función específica.
Por ejemplo, el nodo de **Top Ratings** mantiene su propio diccionario para almacenar los ratings de las películas.

Optimización de Escrituras

Para evitar escrituras frecuentes en disco por cada mensaje recibido, los nodos implementan un sistema de **procesamiento por lotes (batches)**, cuyo tamaño es configurable.

Además, se ha incorporado un mecanismo de persistencia con historial: se guardan las últimas **N snapshots** (por defecto, 5). Esto permite recuperar el estado anterior en caso de corrupción del archivo principal.