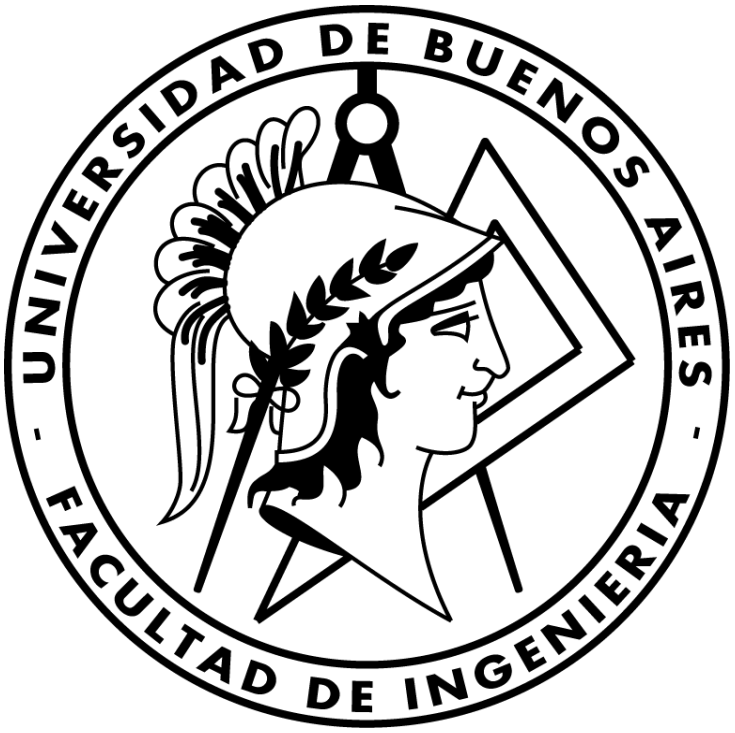


Documento de Arquitectura - Steam Analysis

Nehuén Cabibbo

Clemente Avila

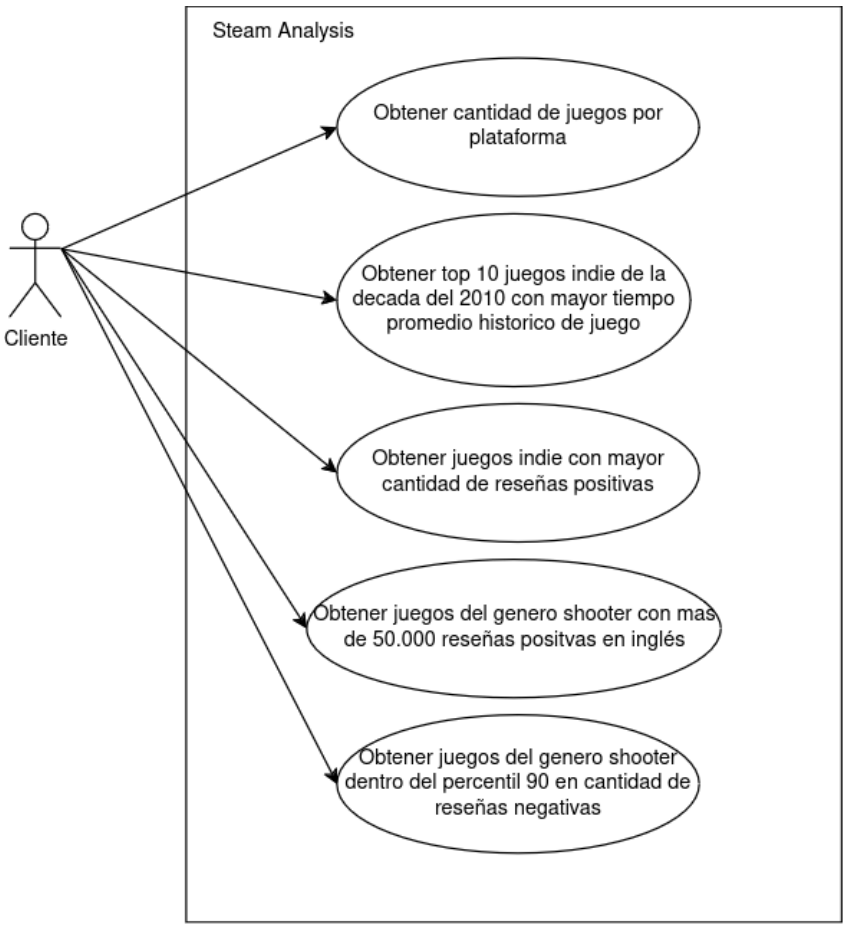
Juan Ignacio Pérez Di Chiazza

Índice

Índice **2**
Casos de uso **3**
Vista Lógica **4**
General 4
Data Pipeline Queries 1, 2 y 3 5
Data Pipeline Queries 4 y 5 6
Vista de desarrollo **7**
Diagrama de paquetes 7
Vista de procesos **8**
Diagrama de Actividad 8
Diagramas de Secuencia 9
Vista física **11**
Diagrama de robustez general 11
Client Handler and Operations (zoom in) 11
Queries 1 y 2 (zoom in) 12
Query 3 (zoom in) 13
Query 4 (zoom in) 13
Query 5 (zoom in) 14
Diagrama de despliegue 16
Manejo de END 17
División de tareas (Tentativo) 18

Casos de uso

- Como usuario quiero poder:
    1. Obtener la cantidad de juegos soportados en cada plataforma (Windows, Linux, MAC)
    2. Obtener el nombre de los juegos top10 del género "Indie" publicados en la década del 2010 con más tiempo promedio histórico de juego
    3. Obtener el nombre de los juegos top 5 del género "Indie" con más reseñas positivas
    4. Obtener el nombre de juegos del género "shooter" con más de 50.000 reseñas positivas en idioma inglés
    5. Obtener el nombre de juegos del género "shooter" dentro del percentil 90 en cantidad de reseñas negativas



Vista Lógica

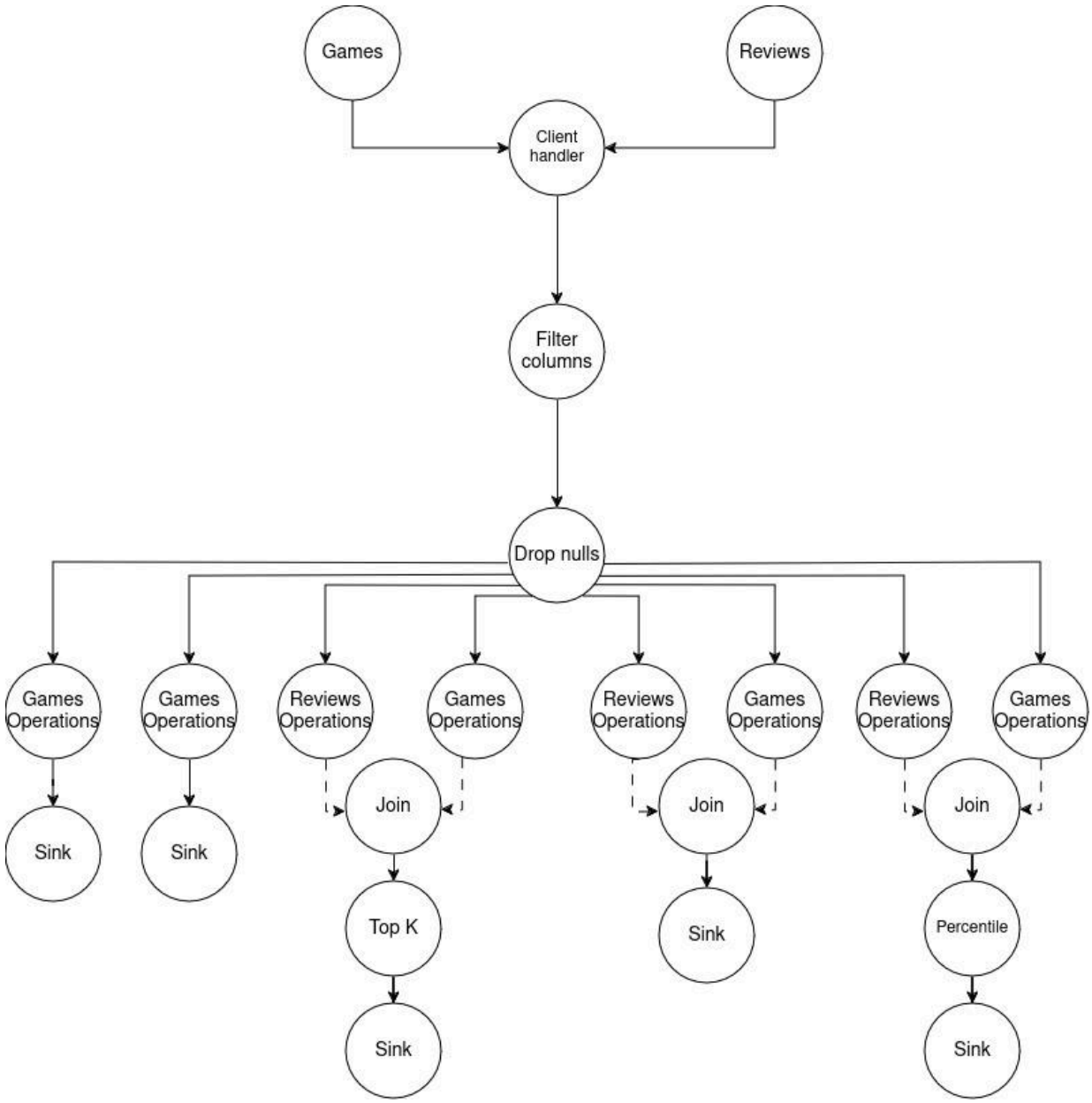
General

Se muestra un DAG del sistema completo para entender el flujo de datos que se da para resolver cada query. Como la resolución de cada query puede pensarse como un pipeline separado, también en la siguiente sección se incluye un DAG query para mayor entendimiento.

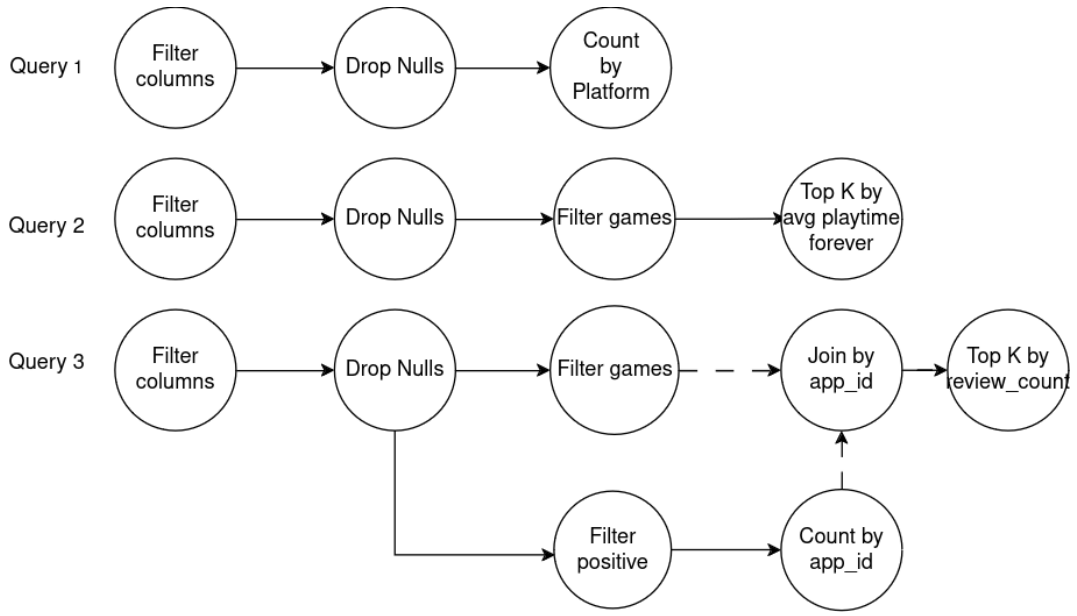
Observaciones

En aquellos lugares donde se utilicen dos flechas punteadas (que salen desde distintos nodos) y se juntan en uno, como es en los casos de join, lo que se quiere explicitar es que un nodo deberá enviar la totalidad de los datos por su parte para que se puedan empezar a recibir los datos del otro.

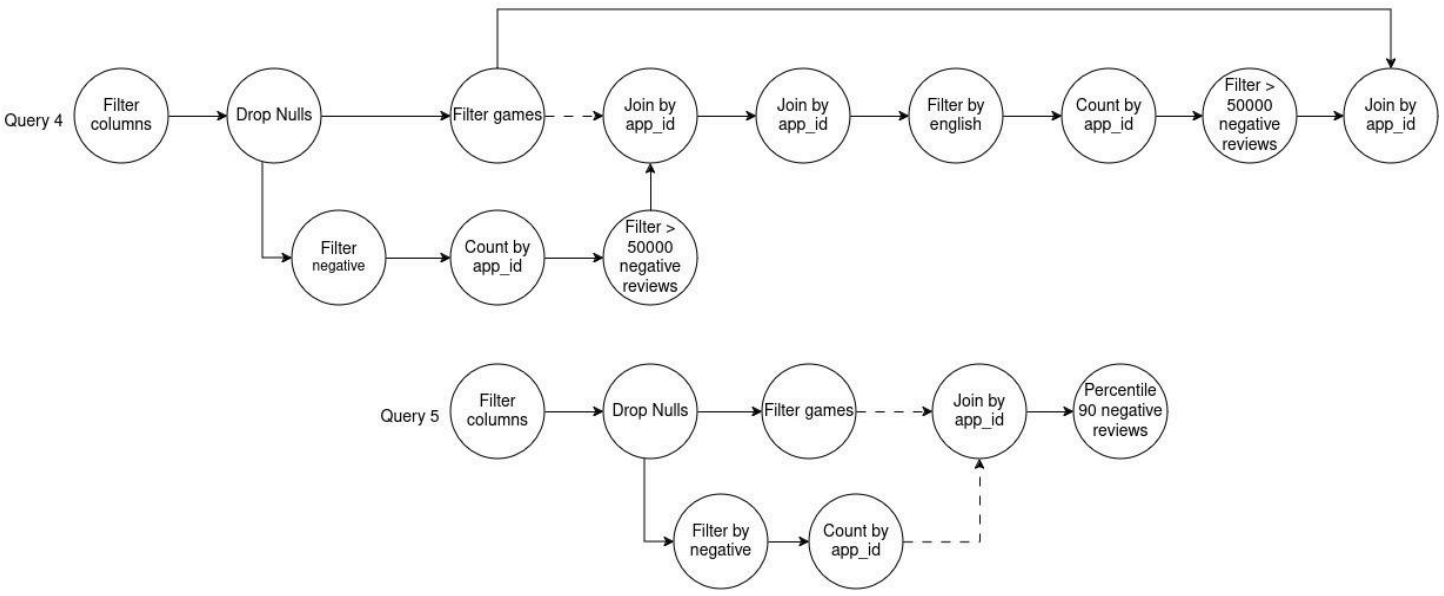
Por otro lado, en aquellos nodos que se utilice una flecha punteada (que sale de un único nodo), se busca representar que se debe obtener la totalidad de los datos por parte del mismo.


Games
Reviews
Client handler
Filter columns
Drop nulls
Games Operations
Games Operations
Reviews Operations
Games Operations
Reviews Operations
Games Operations
Reviews Operations
Games Operations
Sink
Sink
Join
Join
Join
Top K
Sink
Percentile
Sink
Sink

Data Pipeline Queries 1, 2 y 3



Data Pipeline Queries 4 y 5

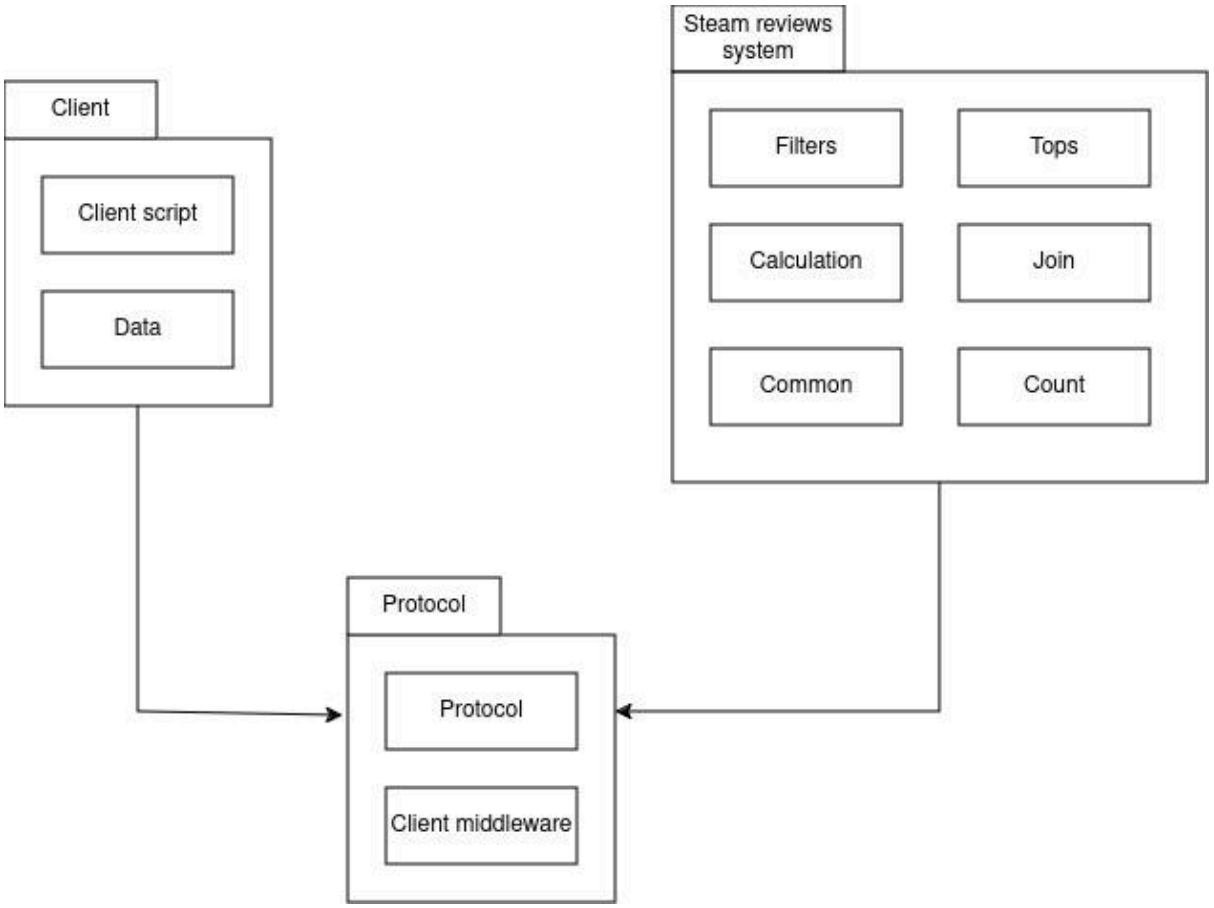


Vista de desarrollo

Diagrama de paquetes

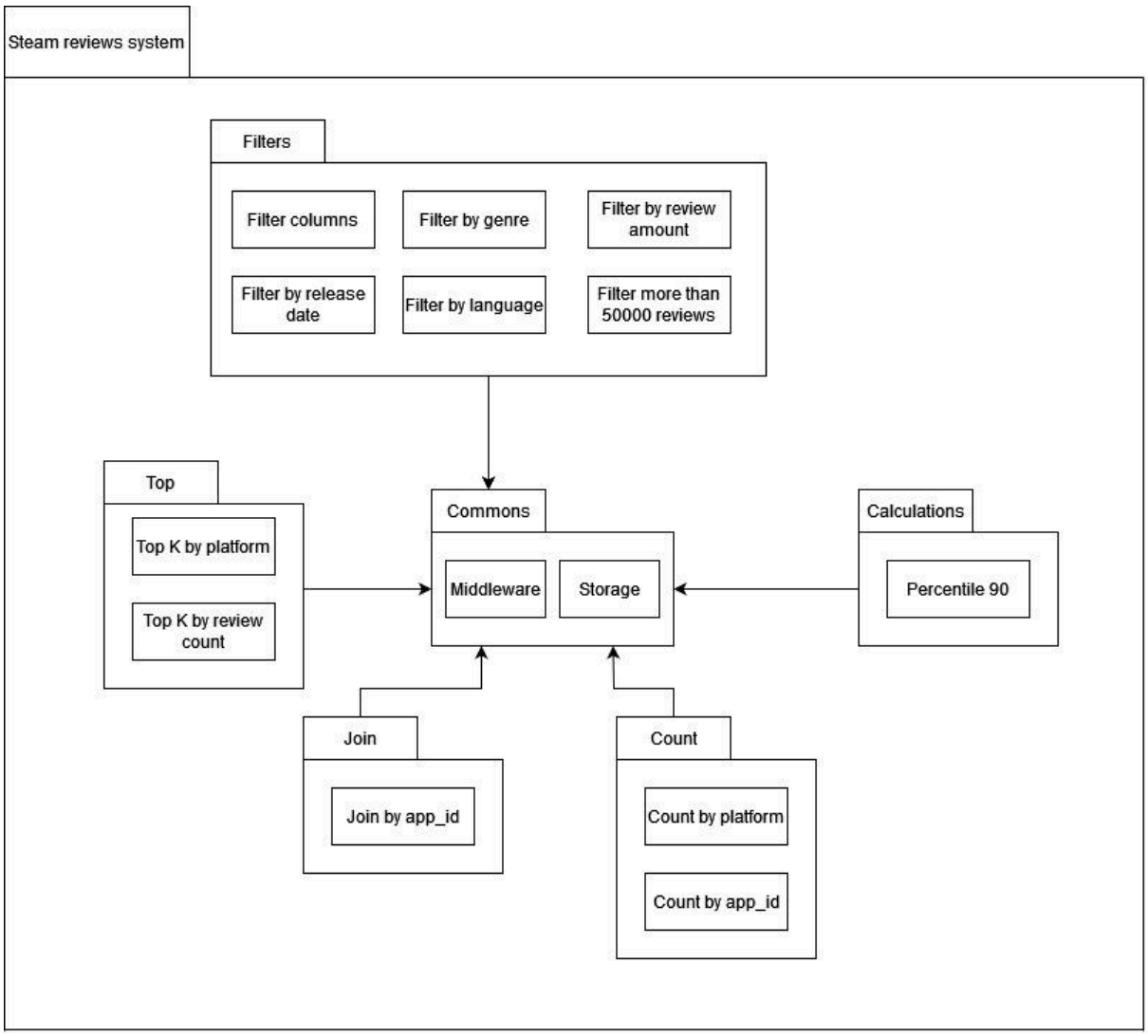
Se expone el diagrama de paquetes para denotar la relación entre ciertos elementos del sistema de similar funcionalidad, y a nivel conjunto como estos se relacionan con otros grupos de elementos.

Con este primer diagrama se busca denotar las dos entidades que existen, el cliente y el *Steam reviews system*. Con este se busca denotar como ambos se relacionan utilizando un paquete llamado *protocol*, que encapsula la comunicación entre ambos.



En este siguiente diagrama se busca entrar en detalle en los componentes del *Steam reviews system*, y cómo se relacionan entre ellos.

A destacar está el paquete *common*, el cual contiene el *middleware* a utilizar para comunicación, al igual que un elemento *storage*, el cual se encarga de encapsular toda la lógica relacionada a la persistencia del sistema.


Steam reviews system
Filters
Filter columns
Filter by genre
Filter by review amount
Filter by release date
Filter by language
Filter more than 50000 reviews
Top
Top K by platform
Top K by review count
Commons
Middleware
Storage
Calculations
Percentile 90
Join
Join by app_id
Count
Count by platform
Count by app_id

Vista de procesos

Diagrama de Actividad

Diagrama de actividad de la comunicación cliente-servidor

El siguiente diagrama se muestra la comunicación a grandes rasgos entre el cliente y el servidor

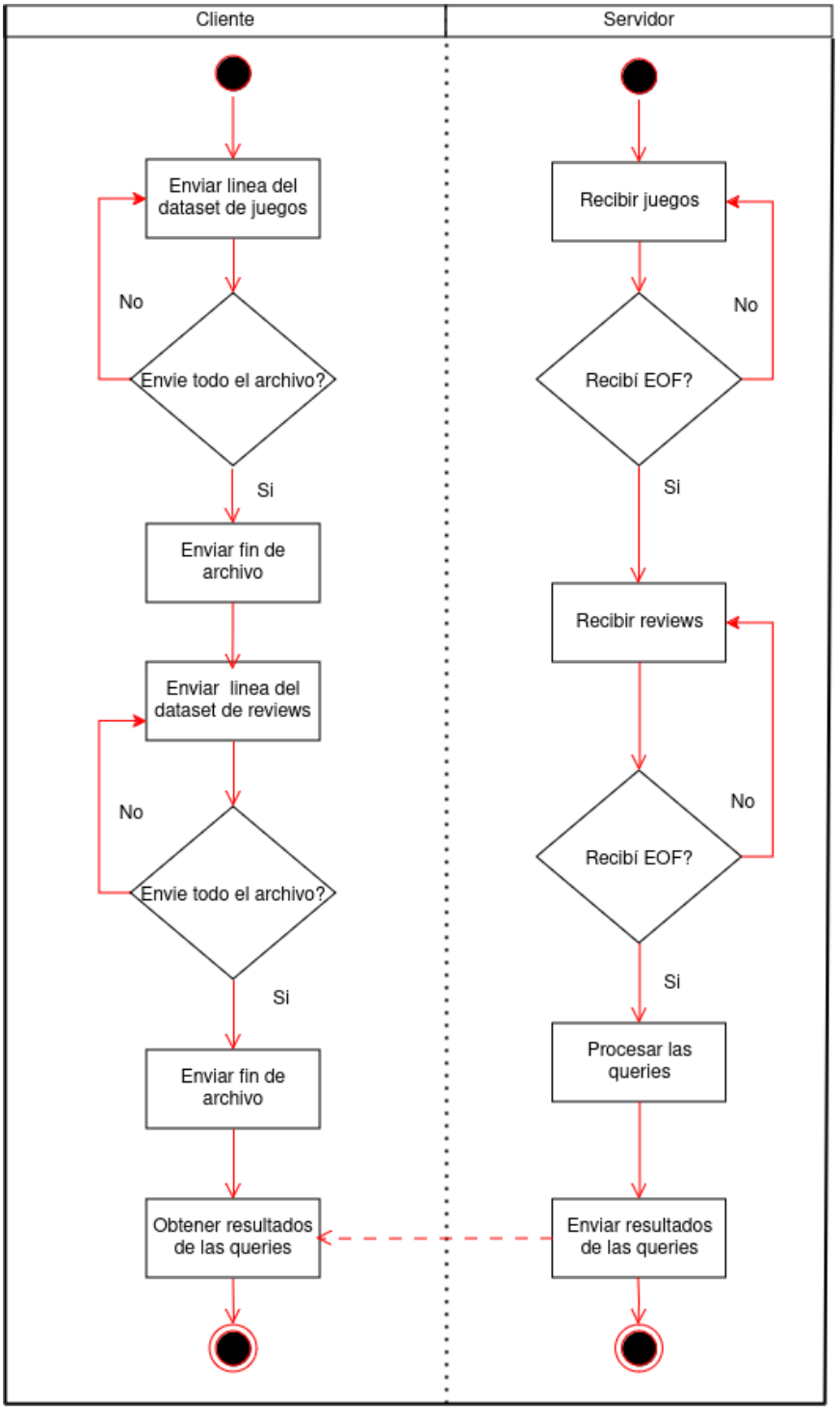


Diagrama de actividad del pre-filtrado de la query 3

El diagrama de actividad muestra el filtrado que es común para todas las queries, solo mostrandolo para el caso específico de la siguiente query: “Nombre de los juegos Indie top 5 en cantidad de reseñas positivas”

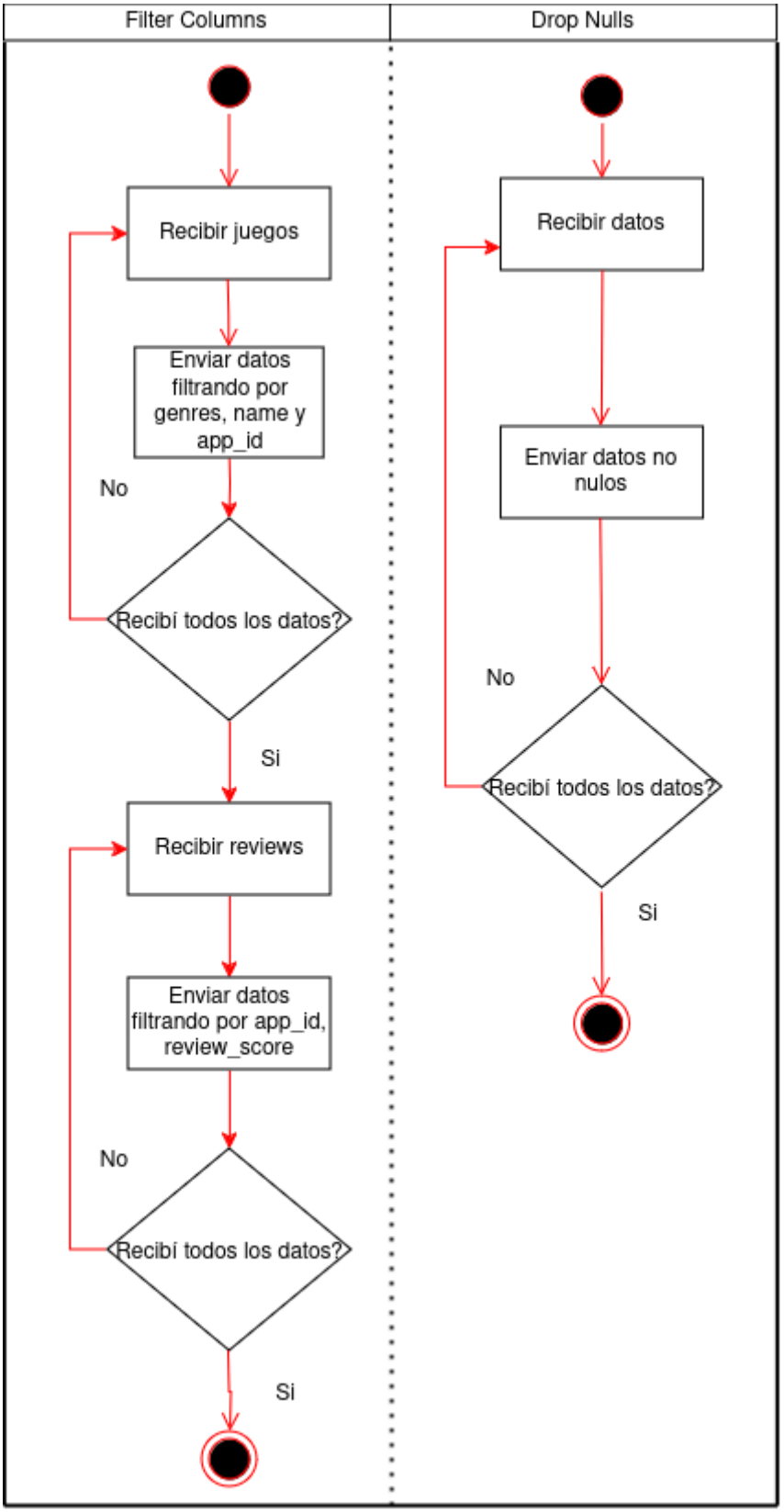


Diagrama de actividad de la query 3

Este diagrama muestra cómo se ejecuta la query 3 luego del filtrado inicial. En este caso se simplificaron algunas de las actividades para poder hacerlo más legible

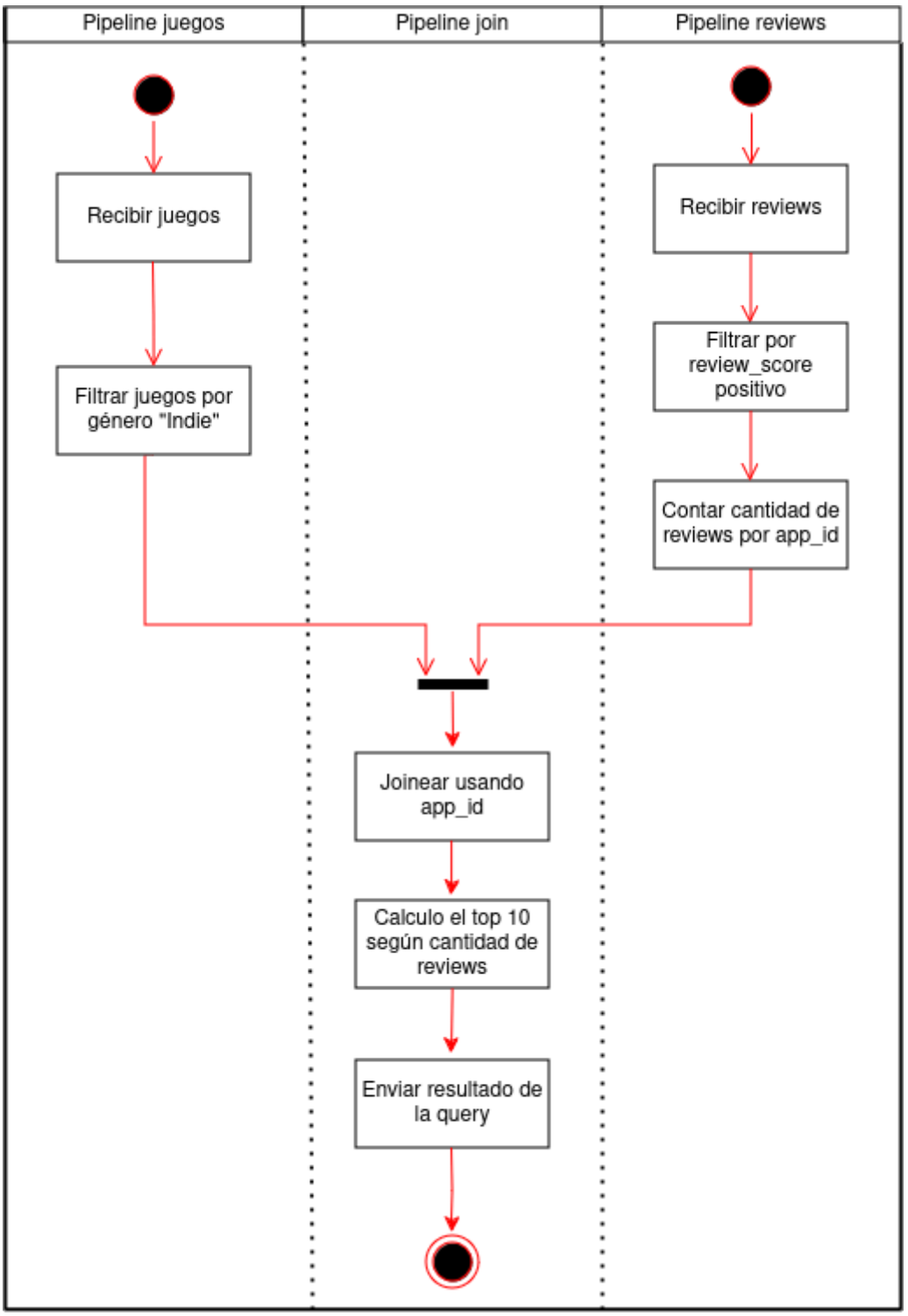
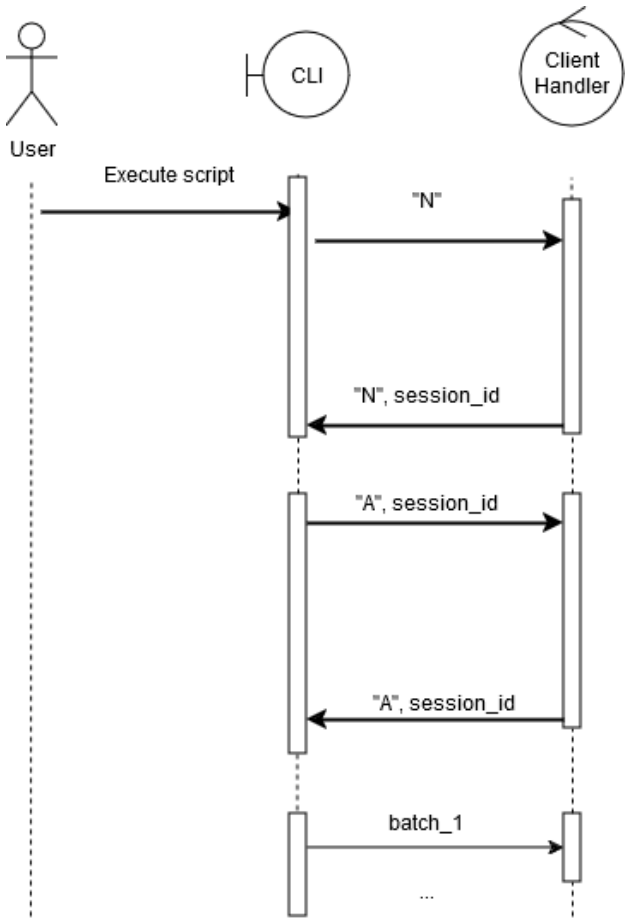
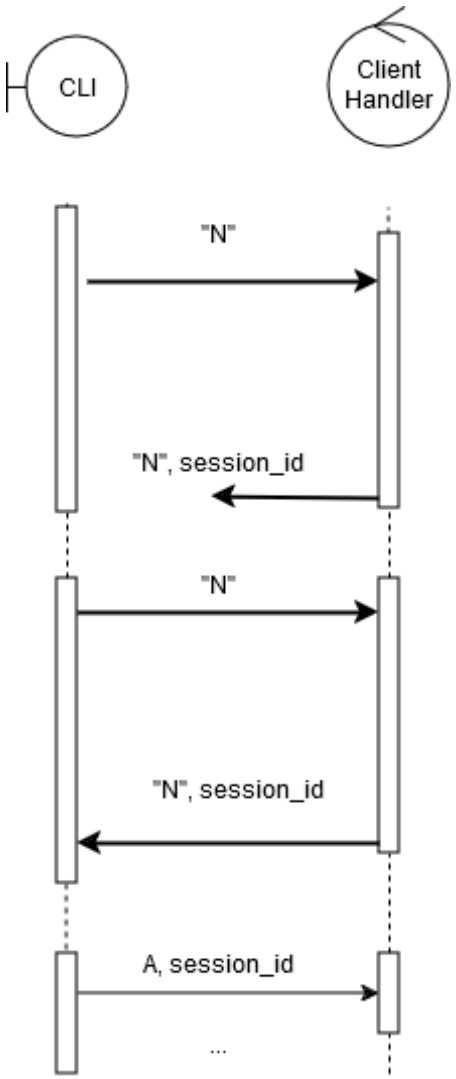


Diagrama de Secuencia

Diagramas de secuencia de conexión entre el cliente y el servidor

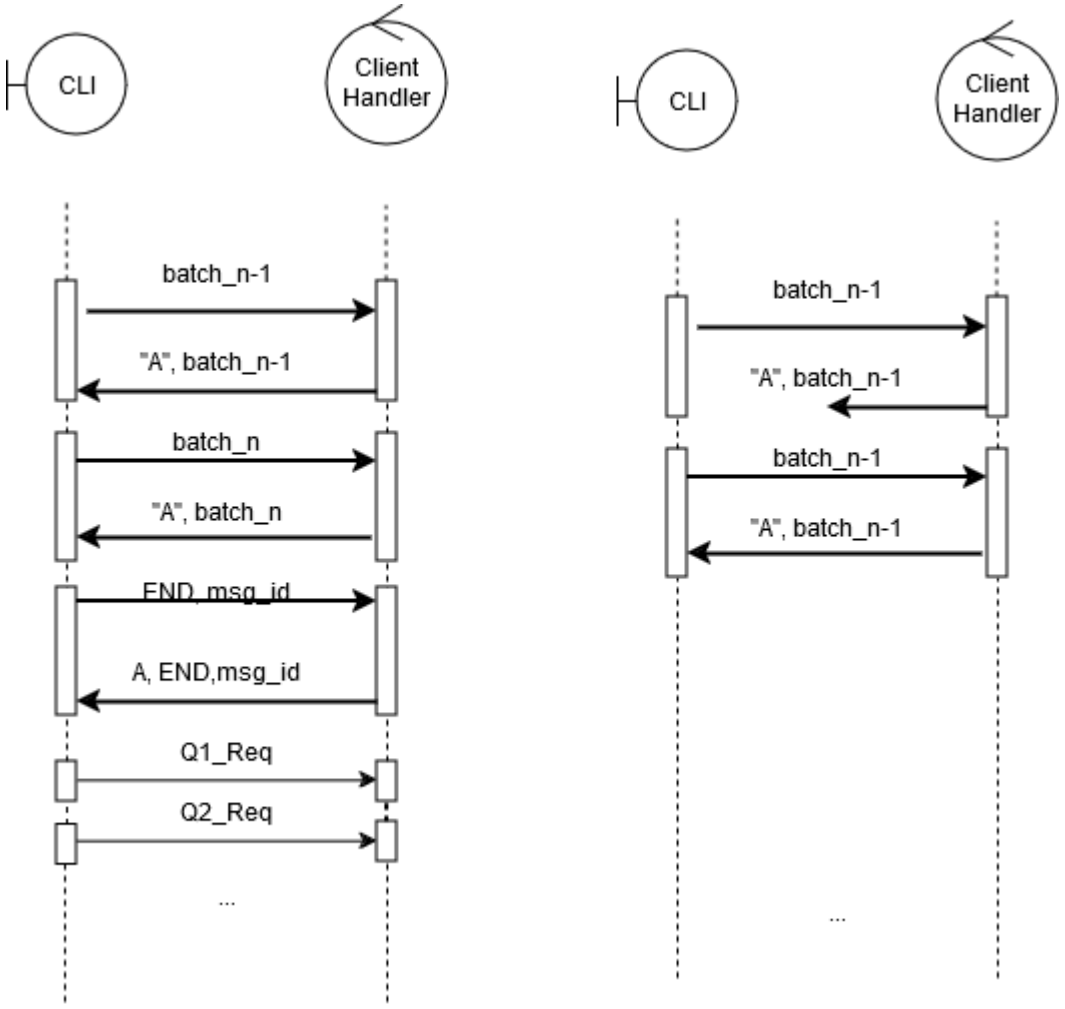


Este diagrama muestra el flujo normal de conexión entre un cliente y un servidor, en el que el primer mensaje representa una petición de un ID de sesión



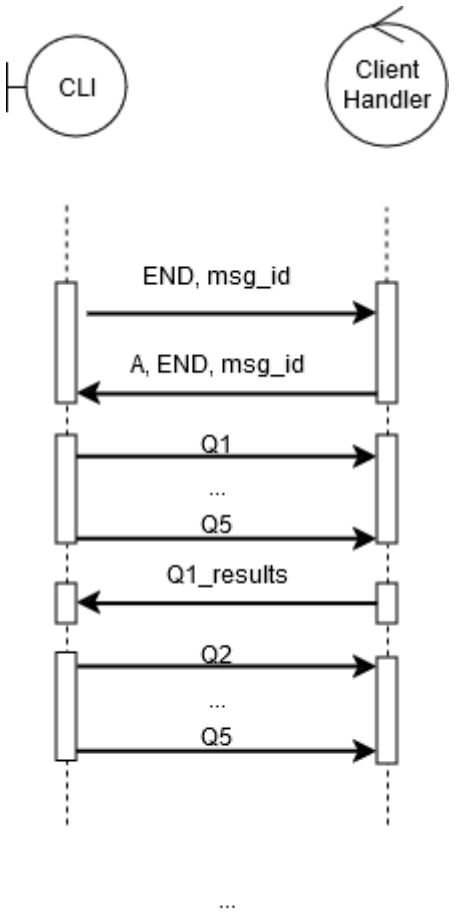
En este diagrama se muestra la conexión entre el cliente y el servidor, habiendo pérdida de mensajes y reconexión en el medio

Diagramas de secuencia de flujo de datos entre cliente y servidor



El diagrama de secuencia de la izquierda muestra el flujo normal de envío de *reviews* entre el cliente y el servidor. Se puede observar que luego del “END” y su respectivo *acknowledge* (“A”), el cliente empieza a solicitar los resultados haciendo *short polling.* Por otro lado, en el diagrama de la derecha, se muestra la eventual pérdida de un mensaje, lo que provoca la retransmisión del mismo.

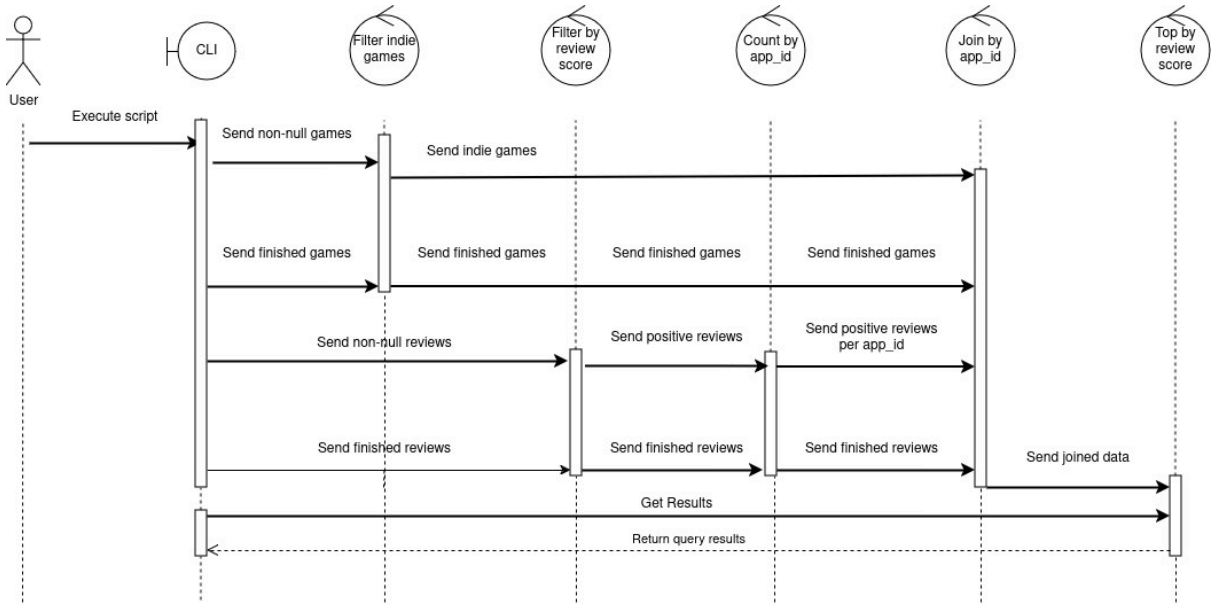
Diagrama de secuencia de solicitudes de resolución de consultas



Este diagrama de secuencia muestra, que luego de obtener los resultados de una consulta, no se volverán a solicitar de nuevo, sino que se seguirá con aquellas consultas que no fueron resueltas.

Diagrama de secuencia de consulta ejemplo

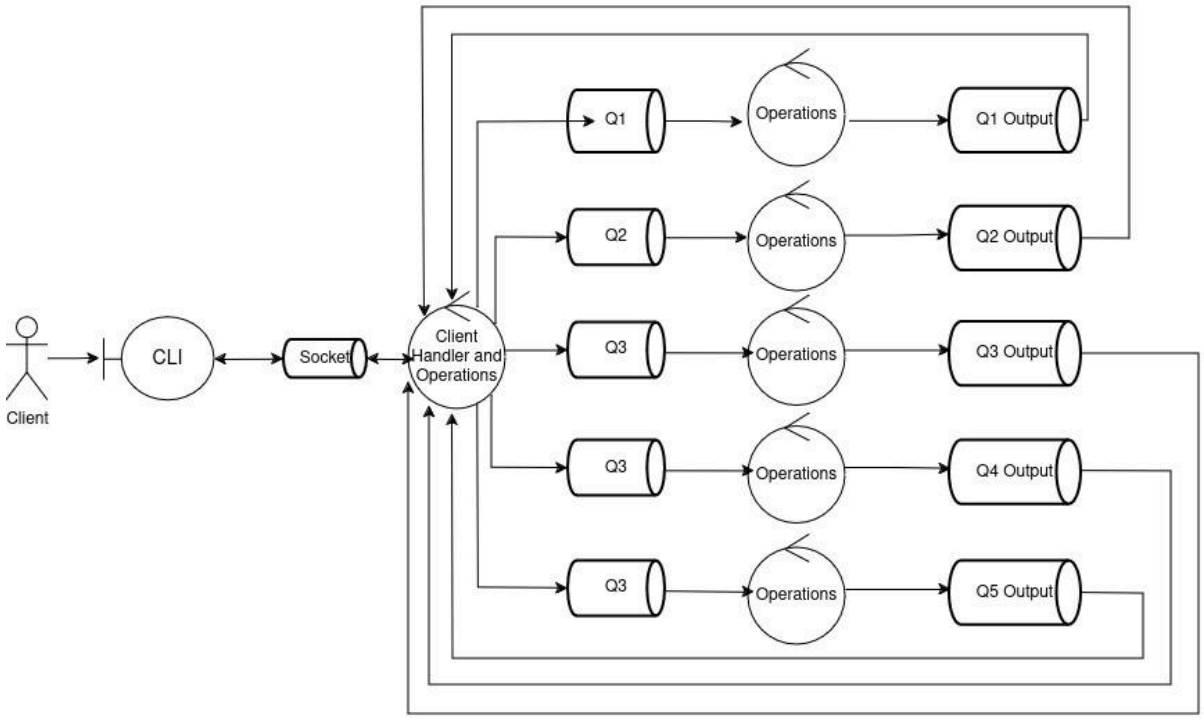
El diagrama de secuencia muestra la ejecución de la la query “Nombre de los juegos Indie top 5 en cantidad de reseñas positivas”, por la misma razón descrita en el de actividad, este “engloba” a las demás queries.



El cliente al momento de enviar los datos correspondientes se mantendrá haciendo *polling* para pedir los resultados que pueda obtener con los datos enviados hasta el momento. Esto no se explicito en todo el gráfico por motivos de legibilidad, en cambio, se optó por dejar, a modo de ilustración, al final del envío de datos

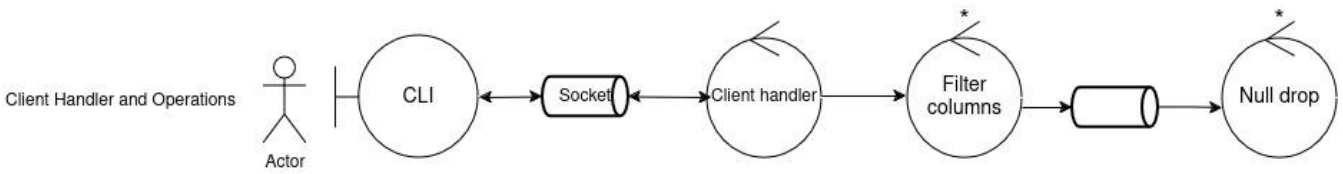
Vista física

Diagrama de robustez general

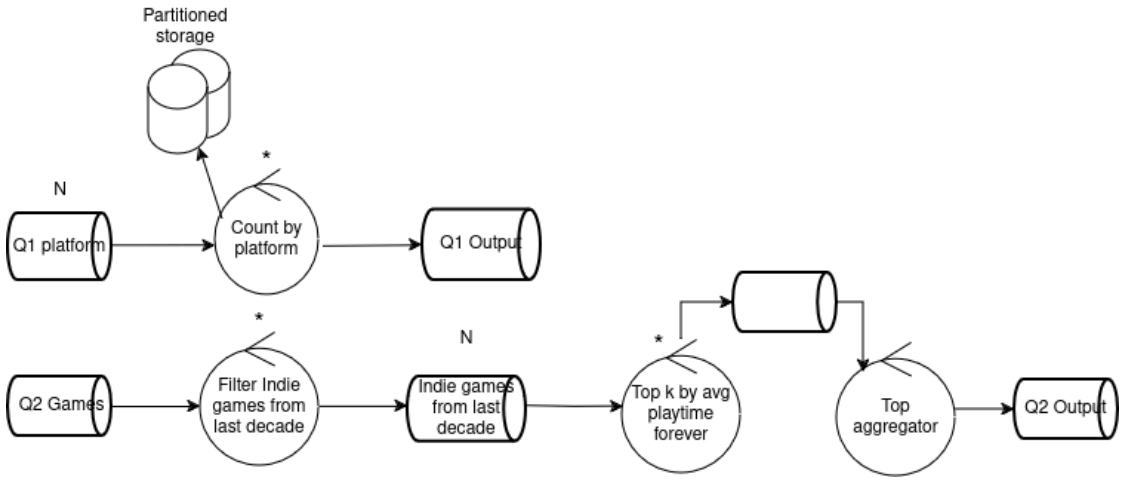


- Q1, Q2, …, Q5 son abreviaturas de QueryN (N el número de la *query*)

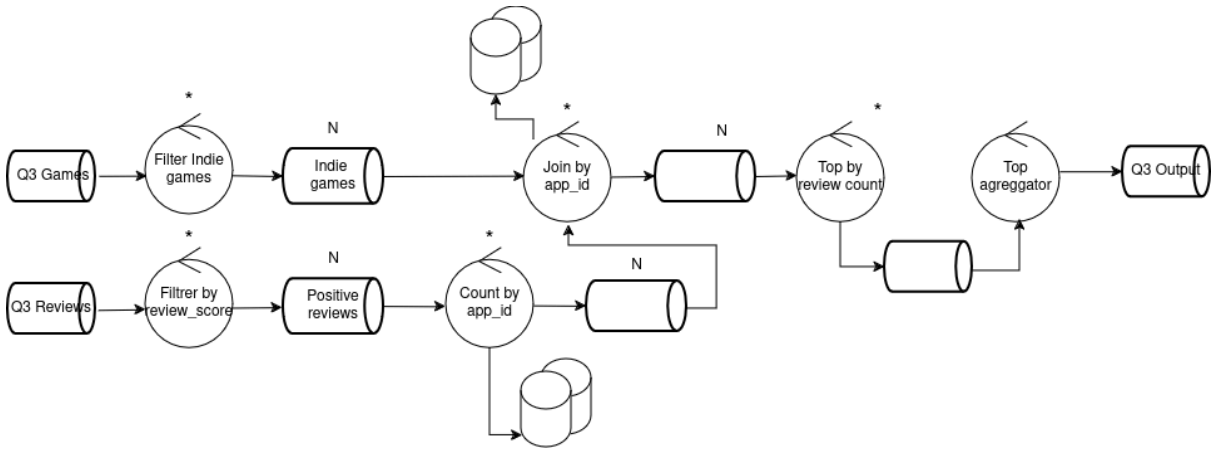
Client Handler and Operations (Acercamiento)



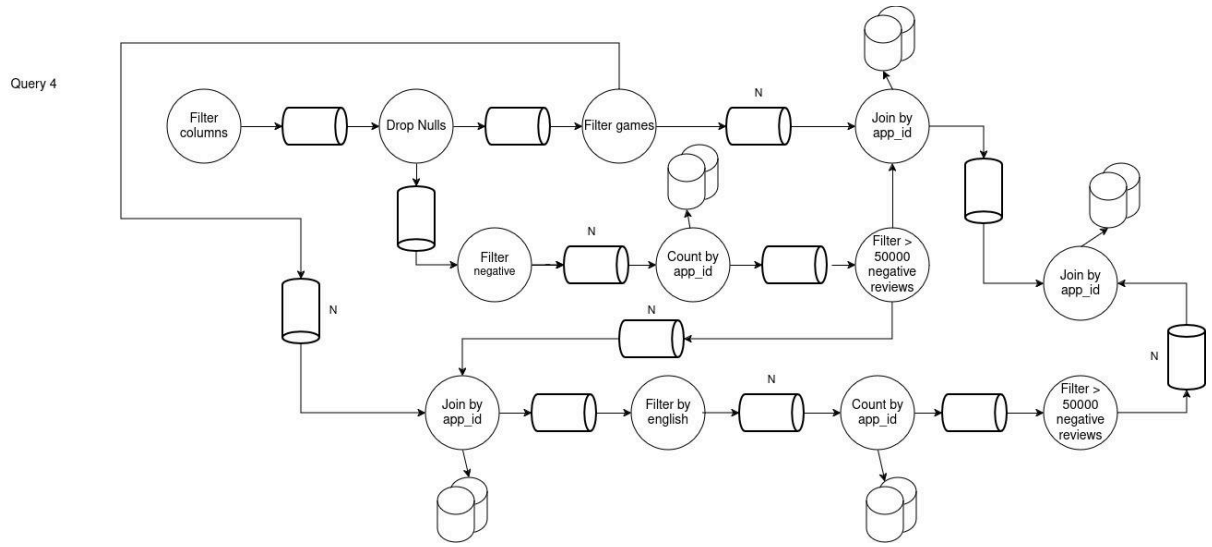
Queries 1 y 2 (Acercamiento)



Query 3 (Acercamiento)



Query 4 (Acercamiento)

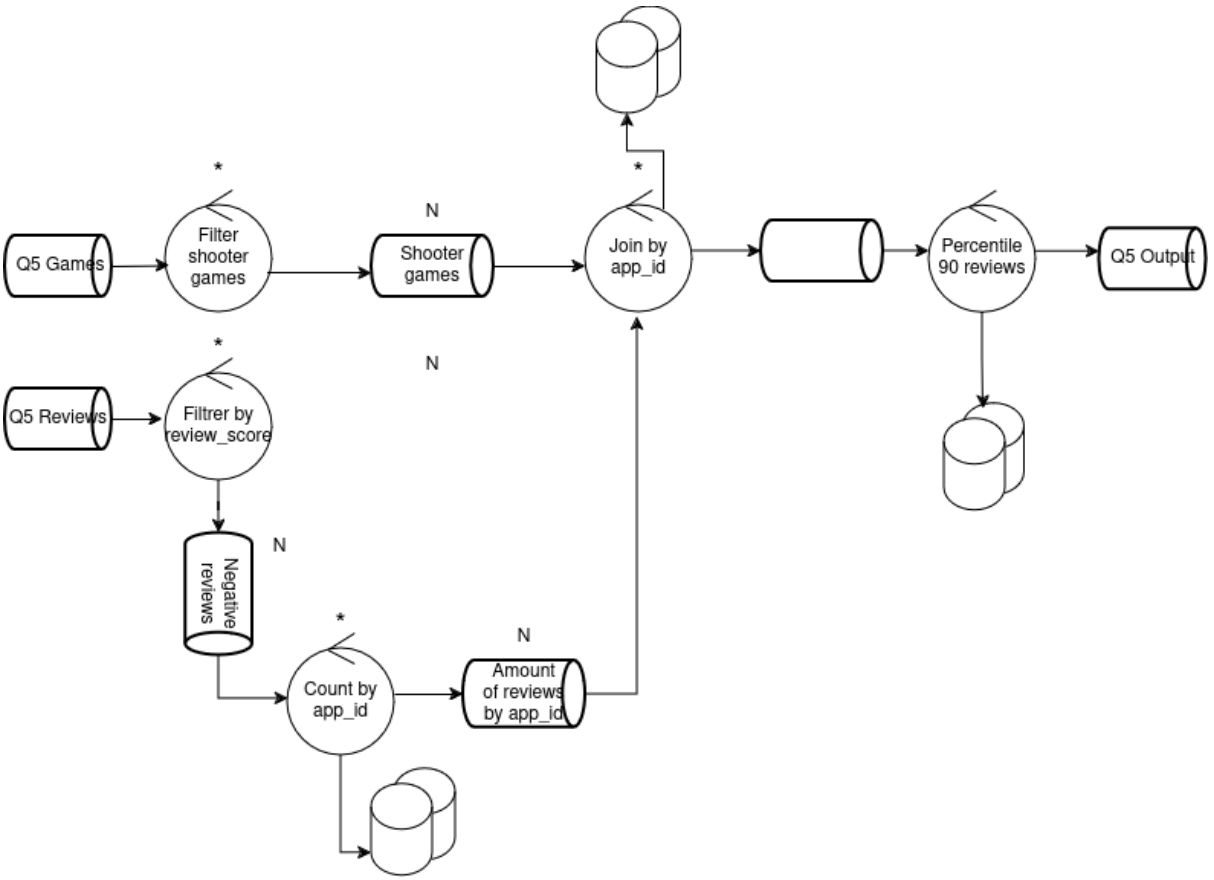


Observaciones

- Para un determinado client_id y app_id se manda a un determinado nodo de join by app_id

- Para un determinado cliente se manda a un determinado nodo de top

Query 5 (Acercamiento)



Observaciones

- Para un determinado cliente se manda a un determinado nodo de percentile 90 reviews
- Para un determinado client_id y app_id se manda a un determinado nodo de join by app_id

Observaciones:

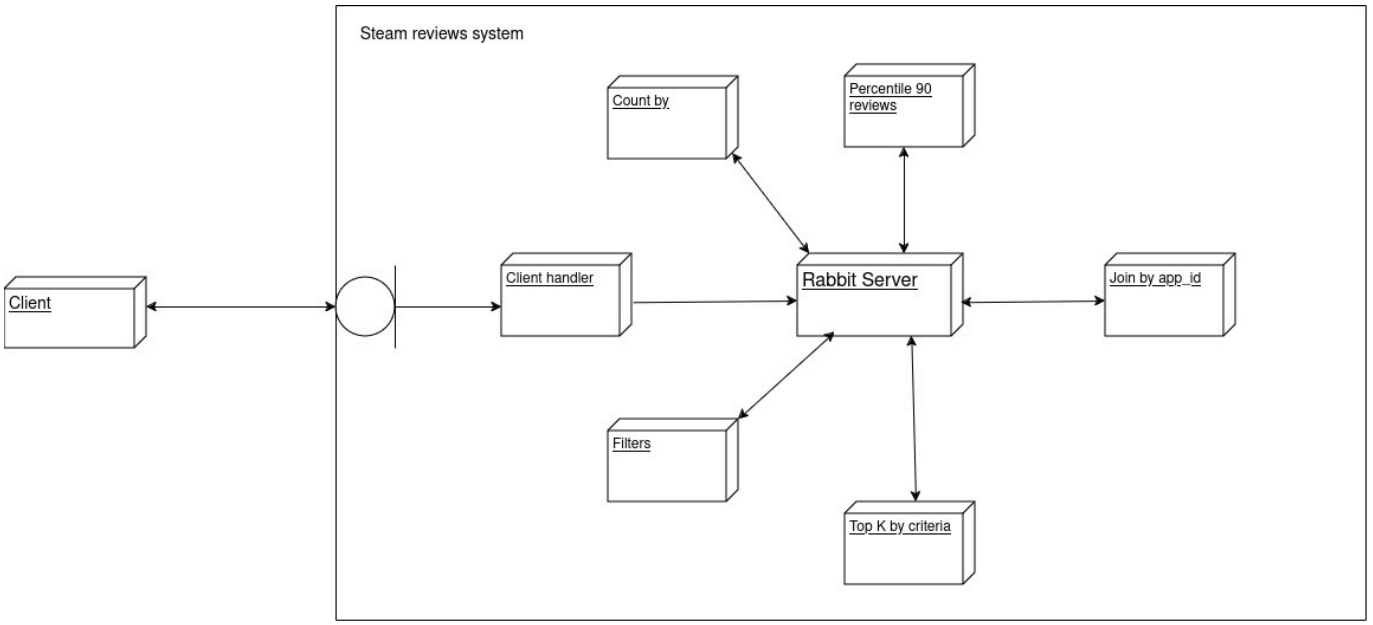
- En aquellos lugares que se utilice la notación "*Partitioned Storage"* se referira a que ese node deberá persistir temporalmente datos de forma particionada, de forma tal de que se pueda cumplir con la query sin tomar supuestos en cuanto a la memoria principal del mismo (véase los casos de *join*, *count*, o aquellos en los que se lleven a cabo funciones de agregación).
- Las particiones se harán según sea conveniente en cada nodo.
- En aquellos nodos en los que se explicite, los mismos serán categorizados utilizando una *sharding key*, siguiendo un patrón de *sharding*.
    - Las colas que tienen un "N" arriba, representan que son para cada *sharding key.*
    - Las *sharding keys* se harán en base al client_id, y un atributo más que dependerá del nodo. Dependiendo de la *query* este se tomará en rangos, por ejemplo app_id (que como se ve en el *dataset* son ids incrementales).
    - La ventaja de esto es que cada nodo, para cada cliente, tendrá los valores absolutos de dicho atributo por el que se busca almacenar, evitando así la necesidad de un agregador o un nodo líder que junte los resultados.
    - La pregunta que surge naturalmente es "qué pasa si la distribución de aquellos atributos elegidos no es uniforme"? En ese caso se podrían tomar métricas del sistema, y crear más instancias de aquellos nodos que tengan más carga.
- La complejidad inherente a la comunicación para el *sharding* será encapsulada por el *middleware*.

Diagrama de despliegue

A continuación se presenta el diagrama de despliegue del sistema.

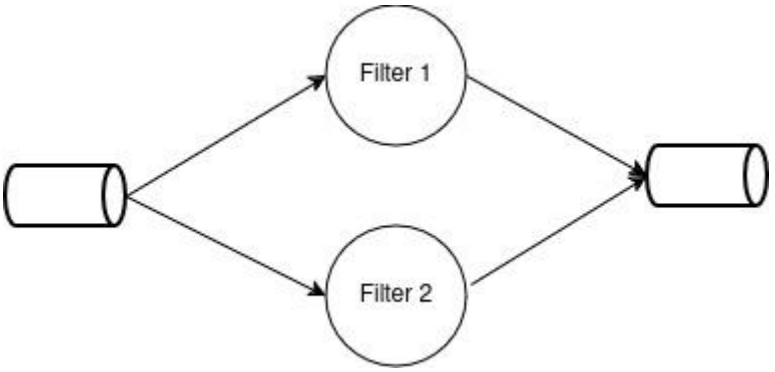
Se busca mostrar en este diagrama los componentes físicos del sistema, y cuáles son estos.

Dado que tiene mucha similitud con el de robustez, se engloban elementos similares que preservan las mismas relaciones.



Manejo de END

A la hora de enviar un mensaje para informar sobre la finalización del dataset para un determinado cliente (vamos a llamarlo END), puede surgir el problema presentado a continuación:



Si se tiene la situación anterior, donde múltiples workers de un tipo de nodo (en este caso de un filtro genérico) consumen de una misma cola, puede ocurrir que uno reciba el END y lo propague, mientras el otro siga procesando data, lo cual podría ocasionar que se crea que se terminó la data antes de lo esperado, perdiendo así datos en el camino.

Para que esto no ocurra, tiene que haber un consenso entre los *workers* para mandar el END, solamente uno de estos debería propagar el END, una vez todos los demás ya lo hayan recibido. Para solucionar este problema, se propone el siguiente algoritmo:

Se asume que todos los *workers* saben la totalidad de workers que hay, para este caso, sería dos.

1) Si se recibe un mensaje de END, se chequea el *payload* del mismo
2) Si no hay tantos ids como *workers,* y el id del *worker* no se encuentra en el payload del mensaje, entonces se agrega al mismo el id del *worker*
3) Si hay tantos ids como *workers*, entonces todos los *workers* recibieron el mensaje, por lo tanto este *worker* propaga el mensaje hacia la cola correspondiente

Monitoreo

Elección de líder

Para garantizar que haya un nodo monitoreando, se realiza una elección de líder entre los monitores.
Para la elección de líder, lo que hicimos fue usar un algoritmo similar al bully. Lo único diferente es que, cuando un nodo se conecta o se reconecta, se va a conectar al líder que esté activo y, en el caso de que no haya ninguno, inicia una nueva elección.

Leader Discovery

Para facilitar la conexión de los distintos nodos al monitor líder utilizamos lo que llamamos *LeaderDiscovery*. Para esto, todos los nodos tienen un hilo escuchando por conexiones para responder quién es el monitor líder y, en el caso de preguntar durante una elección, estos responden con un mensaje indicándolo.

Por el lado de los nodos, al iniciar estos utilizan el *LeaderDiscovery* para obtener el id del líder y una vez lo obtienen, se conectan a este para ser monitoreados. En el caso de que el monitor líder se caiga, los nodos volverán a usar el *LeaderDiscovery* para conectarse al nuevo líder.

Detección de caídas

Caídas de nodos

Hay varios casos de caídas de nodos para los monitores:

- **Caída de un nodo que se conecto al monitor líder**: En este caso, el monitor va a detectar casi inmediatamente la caída del nodo (debido a que usa TCP) y va a volver a levantarlo. Si por algún motivo, el nodo vuelve a fallar antes de la reconexión, el monitor va a seguir levantandolo hasta que eventualmente se reconecte.

- **Caída de un nodo durante una elección de líder**: En el caso de que un nodo se caiga durante una elección de líder, una vez termine la elección, el nodo líder va a detectar que un nodo nunca se volvió a conectar, luego de un determinado tiempo y va a levantar el nodo.

Caidas de nodos monitores

En cuanto a las caídas de los distintos nodos monitores, tenemos los siguientes casos

- **Se cae el líder**: En este caso los demás nodos van a detectar la caída del líder y van a iniciar una nueva elección.
-
- **Caída de un nodo que no es líder**: Si un nodo que no es líder se cae, el monitor líder va a detectar que el nodo está caído, ya que no envió heartbeats al monitor líder y lo va a levantar
-
- **Caída de un nodo de mayor ID durante la elección de líder**: Si pasa esto, los demás nodos que participan de la elección van a detectar que, luego de un tiempo, no se eligió ningún líder y van a volver a lanzar la elección

En todos los casos, el líder detecta que otro monitor está caído si no recibió ningún heartbeat luego de un tiempo.

Recuperación

Integridad del sistema de archivos

Para evitar lidiar con archivos corruptos, toda modificación a hacer sobre un archivo se utiliza un archivo temporal, sobre el cual se escriben todos los datos del archivo original (este es, **siempre**, muy acotado), y se hacen las modificaciones pertinentes. Luego, para efectivamente modificar el archivo se utiliza **os.replace()**, operación la cual para todos los

sistemas de archivos modernos es **atómica,** evitandonos así lidiar con la corrupción de los archivos a cambio del costo de tener que reescribir el archivo original.

Cantidad de ENDs

Este es un estado propio del *Top K, Counter by app id, Join y Percentile,* los cuales necesitan una **determinada cantidad de END para completar su operación.** El estado es un diccionario en memoria, el cual para cada id de cliente, mantiene la cantidad de mensajes de END que se recibieron. Para recuperar este estado, se mantiene, **por cliente**, en una carpeta que tiene de nombre su id, un archivo llamado ***ends.txt.***

Este tiene, en cada línea, un id de mensaje de END que fue procesado (para luego poder filtrar duplicados con estos tal como se explica en la parte de *filtro de duplicados*). Con esto, recuperar el diccionario original es trivial.

Middleware

Dado que se publica de a *batches*, se va almacenando **en memoria mensajes por cola** para enviar, por un periodo de tiempo, hasta que **se llegue al *batch size***, ahí **efectivamente se publica el *batch***. Esto implica mantener un estado más, ya que si un *batch* no se completó con un mensaje que llegó, a este se le hizo *ACK*, y se cayó el proceso correspondiente, efectivamente perdí el mensaje anterior.

El estado que se mantiene en memoria, es un **diccionario**, cuyas **claves** son los **nombres de las colas** a las cuales publicar, y cuyo **valor** son los **mensajes acumulados** hasta el momento.

Para solucionar esto se hizo algo muy similar a la recuperación de la cantidad de ENDs. Se mantiene **por cada cola** un **archivo de mismo nombre**, el cual contiene los mensajes acumulados hasta el momento por línea.

Filtro de duplicados

Es un problema que surge por las caídas, dado que si se envía un mensaje antes de hacerle *ACK,* rabbit lo va a re-encolar, y al retomar su actividad el nodo correspondiente va a efectivamente enviar el mismo mensaje dos veces, generando un estado invalido en el sistema.

Solamente los nodos con estado (*Counter*, *Counter by AppId*, *Join*, *Top K* y *Percentile*) y el *Client Handler* se encargan de filtrar duplicados.

Los filtros pasan el problema al siguiente nodo con estado, y esta cadena de “pateo” del problema se termina en el *Client Handler,* que es quien finalmente termina enviando los resultados al cliente.

Dado que no podemos identificar cada *batch*, decidimos identificar mensajes duplicados, para esto, el cliente en su *middleware,* agrega un id a cada mensaje.

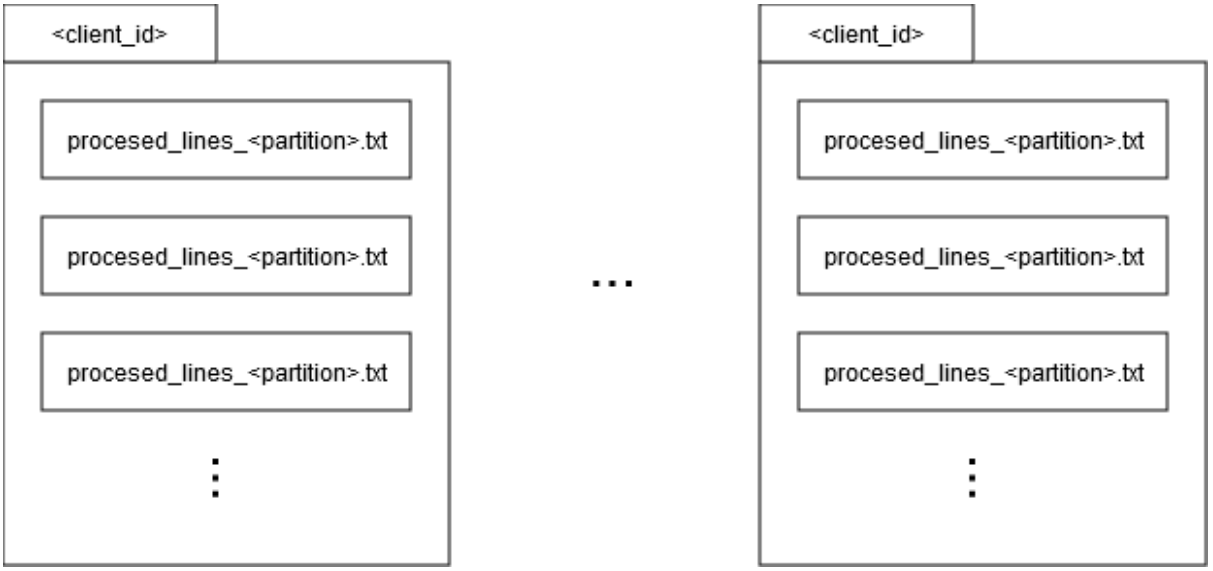
Filtros

Como se dijo anteriormente, estos nodos solamente “patean” el problema.

Counters (*by platform* y by *app id)*

Dado que cada mensaje “genera” un nuevo estado (i.e: si cuento por *app id*, y me llegan los mensajes de id 2 y 3, estos me generaron el estado app id, 2), se necesita poder identificar, para cada cliente, cuáles fueron todos los ids de mensajes que fueron procesados históricamente para el mismo, para así no contabilizar dos veces un mensaje duplicado.

Para esto, se implementó un *log* que contiene las líneas procesadas por un determinado cliente, la estructura del sistema de archivo del mismo es la siguiente:



Cada **carpeta corresponde a un cliente**, y cada archivo ***processed_lines_<partition>.txt*** tiene los **ids de mensaje ya procesados**. Se hacen particiones para que chequear si un determinado id de mensaje fue procesado o no (lo cual es una operación muy frecuente), sea eficiente.

Hay un problema con esto, y es que un estado, puede verse modificado por **múltiples ids de mensaje**, al estar *loggeando* en distintos archivos, una caída en la mitad del *logging* implicaría no estar *loggeando* distintos archivos.

Para solucionar esto, se tiene un ***log_general.bin***, en donde **se *loggean* todos los ids de mensaje a procesar juntos**, tal que si se cae en medio, internamente el *log* puede utilizar este *log general* para recuperar todos los ids de mensaje procesados.

Top/Percentil

En ambos de estos nodos el filtro de duplicados se simplifica mucho, ya que al estar ambos insertando en un archivo ordenado para realizar sus respectivas operaciones, se puede simplemente, a lo que guardan, añadirles como clave primaria el id de mensaje. A la hora

de insertar, se chequea si el id de mensaje se encuentra presente, en caso de estarlo, se descarta el mensaje actual, caso contrario, se procesa.

Join/Client Handler

Al igual en el *Top* y el *Percentil*, el filtro de duplicados se simplifica aquí, ya que se agregan al final de un archivo las *reviews* y los *juegos* (O los resultados para el caso del *Client Handler*) que van llegando cuando es necesario. Por lo explicado en la sección de **integridad de archivos** de **recuperación**, simplemente basta con ir chequeando cada mensaje leído, si su id de mensaje coincide con ya sea el *juego* o la *review* a guardar.

División de tareas (Tentativo)

Todos: Middleware y storage

Nehuén:
Semana 1 (26/9 - 01/10) :

- Drop columns
- Null drops
- Filter by column

Semana 2 (02/10 - 09/10):

- Filter by language
- Top

Clemente
Semana 1(26/9 - 01/10):

- Envio de juegos desde el cliente
- Envío de reviews desde el cliente
- Polling para obtener resultados desde el cliente

Semana 2(02/10 - 09/10):

- Count sin sharding
- Count usando sharding
- Percentile 90

Juani
Semana 1 (26/9 - 01/10)**:**

- Join

- Hash by client_id, app_id

Semana 2 (02/10 - 09/10)**:**

- Guardar de manera particionada
- Obtención de info que está particionada