.UBAfiuba
FACULTAD DE INGENIERÍA

Tolerancia a Fallos – Coffee Shop Analysis

Sistemas Distribuidos I (75.74)

Fecha: 7 de diciembre de 2025 **Corrector:** Nicolas Zulaica

Nombre	Padrón
Bubuli, Pedro	103452
Cuevas, Juan Francisco	107963
Zimbimbakis, Francisco Manuel	103295

Índice

1. Diagramas

1.1. DAG

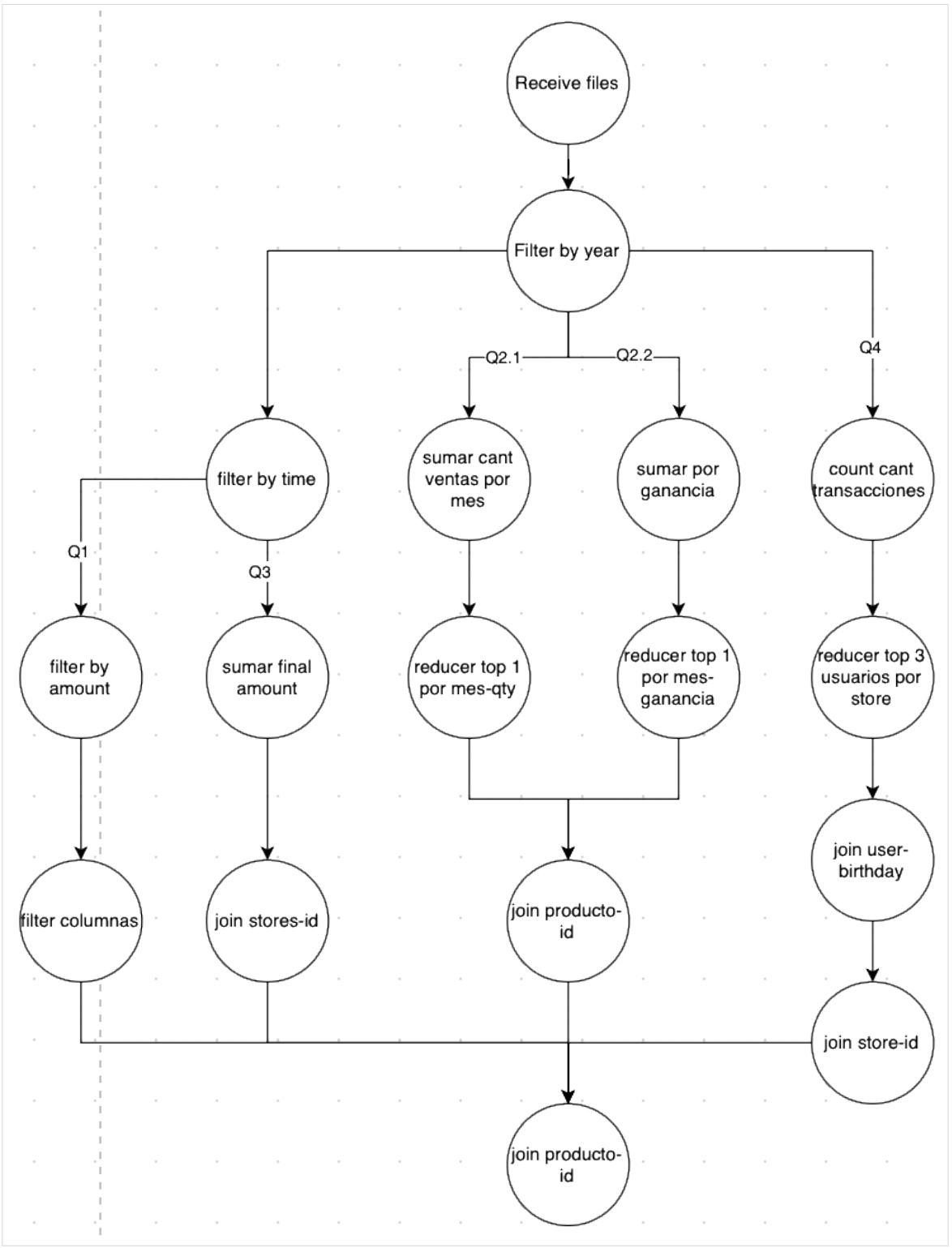


Figura 1: DAG

Este diagrama corresponde al DAG (Directed Acyclic Graph) del sistema y representa la secuencia de operaciones que se ejecutan desde la recepción de los archivos de entrada hasta la obtención del resultado final. A partir del bloque inicial de recepción de datos, se despliegan distintos caminos en paralelo que aplican filtros, agrupamientos, agregaciones y uniones, según la lógica requerida por cada consulta. Por ejemplo, en una rama se filtra por rango horario y luego por monto, mientras que en otra se realizan sumatorias de cantidades y subtotales que después se agrupan por mes y producto, para finalmente unirse con la información del menú. También se incluye un flujo específico para identificar los clientes más relevantes mediante conteo de transacciones, ordenamiento y ranking. Todas estas transformaciones confluyen en los resultados parciales de las consultas Q1, Q2, Q3 y Q4, que luego se integran en un resultado final.

1.2. Diagrama de secuencia: Envío y recepción de batches

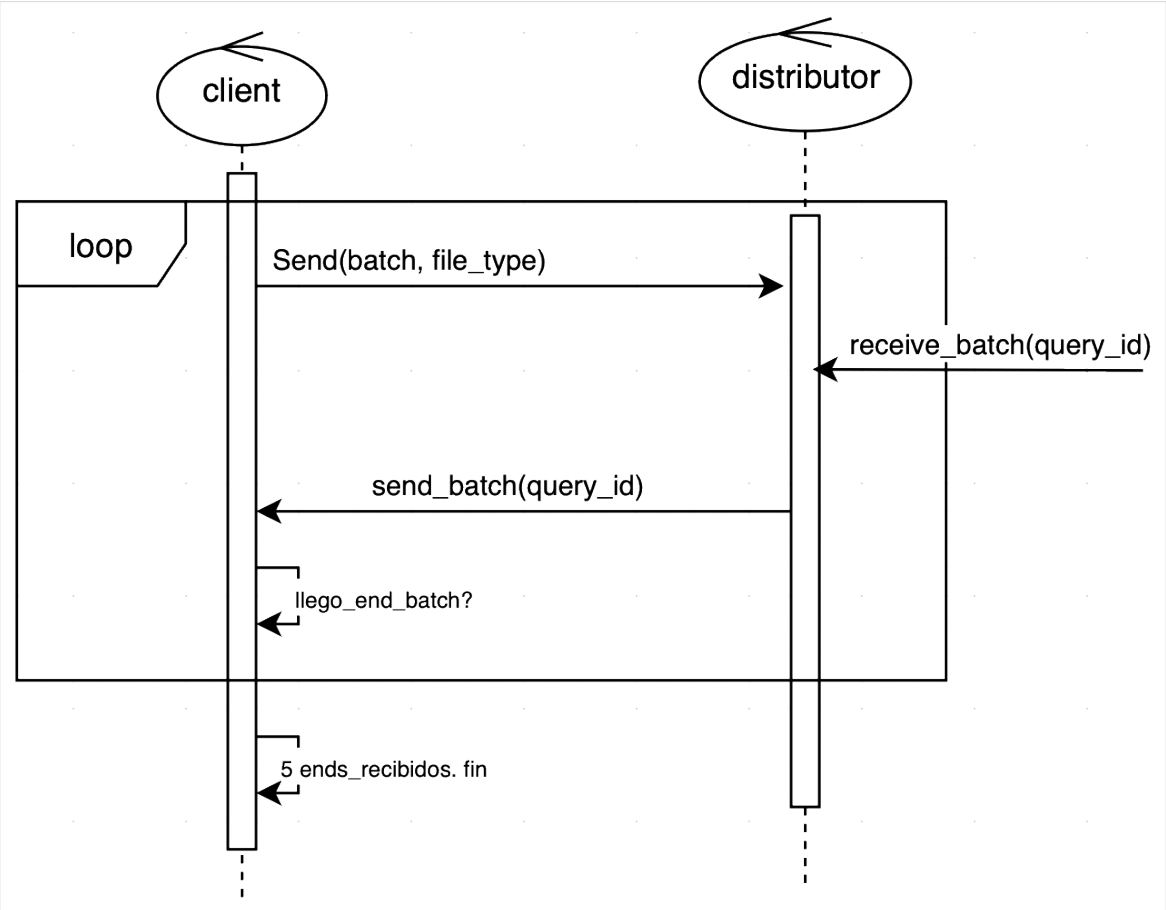


Figura 2: Diagrama de secuencia: envío y recepción de batches

1.3. Arquitectura por módulos y componentes

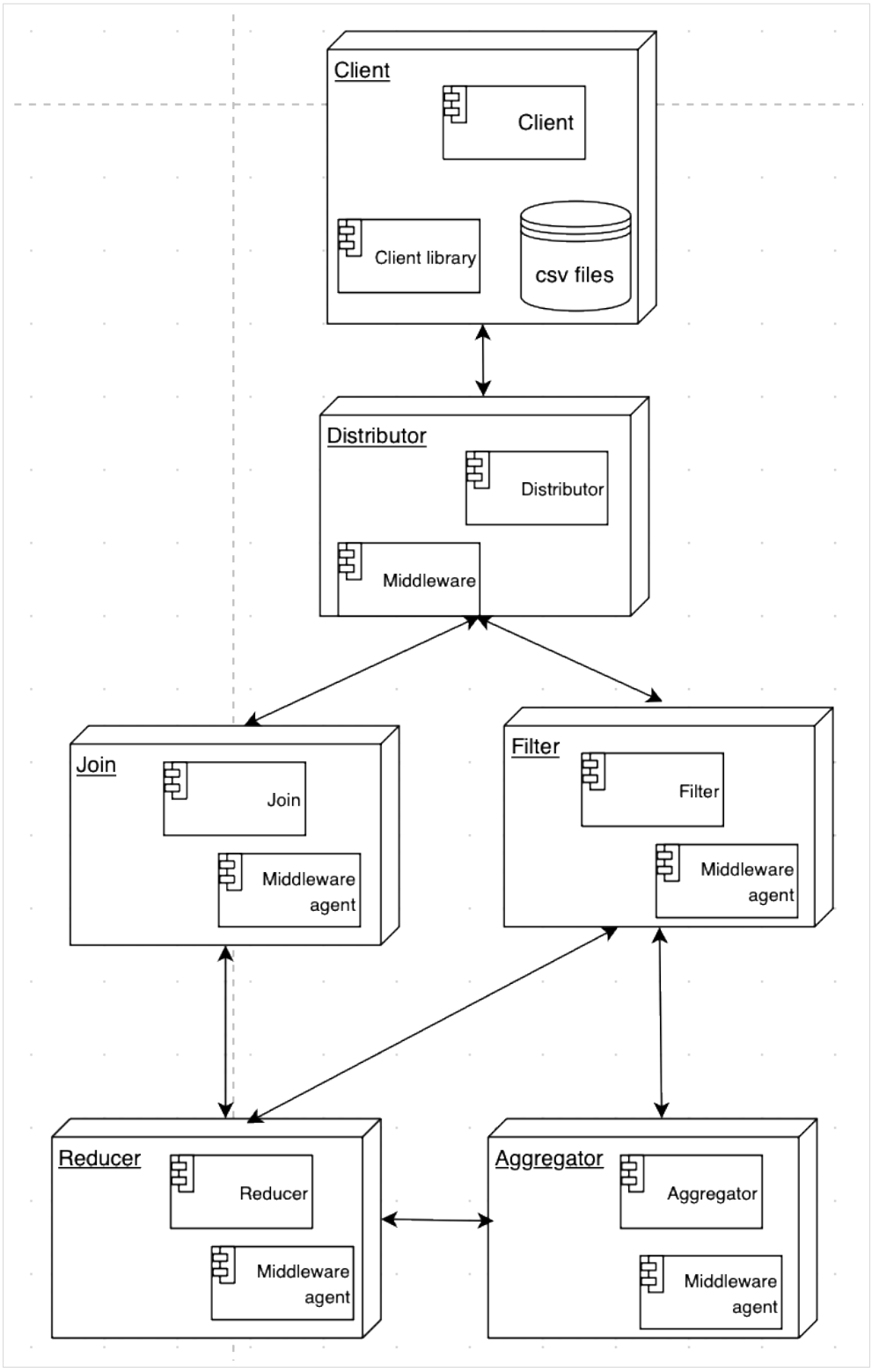
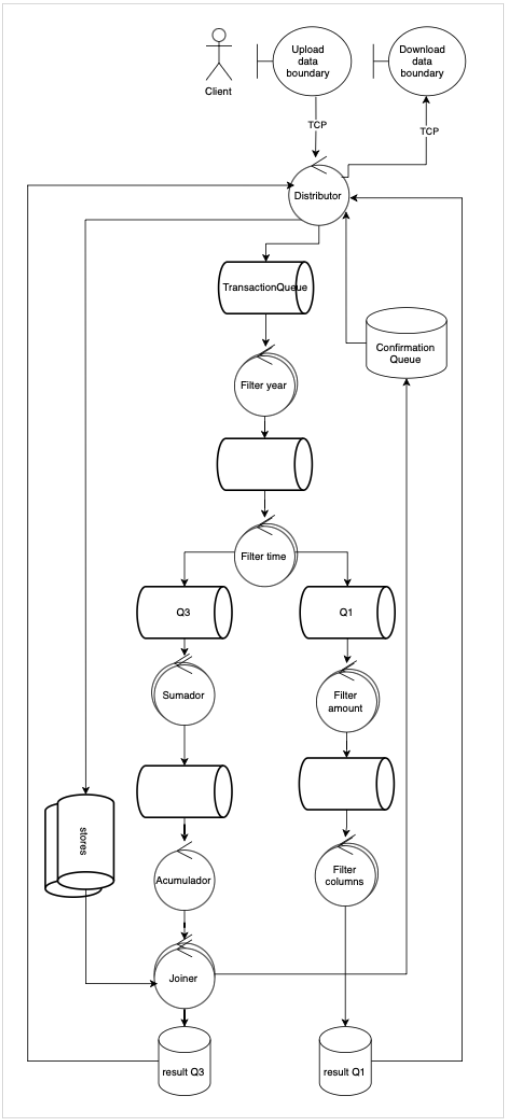


Figura 3: Arquitectura por módulos y componentes

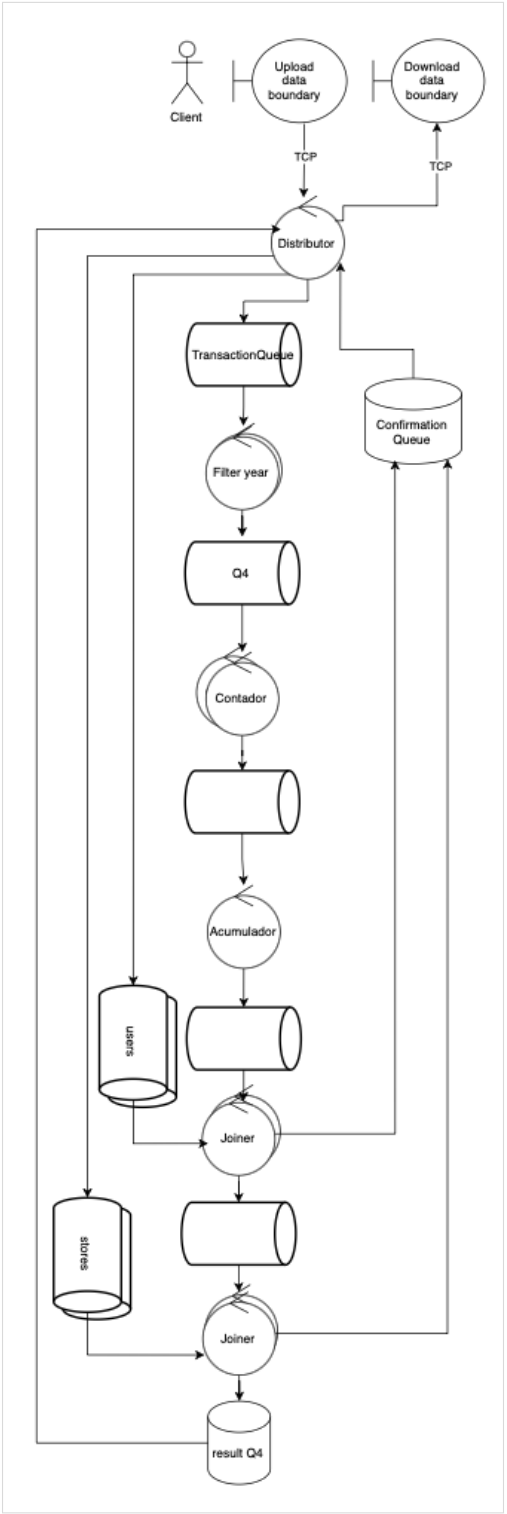
Este diagrama de despliegue representa la vista física del sistema. Muestra cómo el cliente interactúa con el gateway, que funciona como balanceador dentro del middleware. Desde allí se distribuyen las tareas hacia distintos nodos especializados, como Join, Filter o Group by, cada uno con su agente de middleware para coordinar la ejecución. De esta manera se refleja la

arquitectura distribuida del sistema y cómo se asignan los procesos en diferentes componentes físicos.

1.4. Diagramas de Robustez

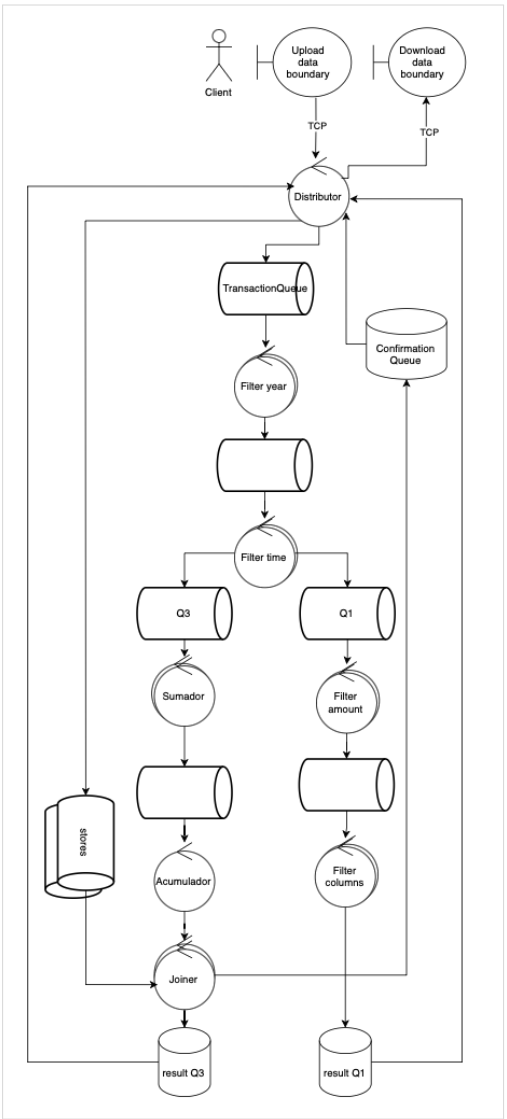


(a) Diagrama de Robustez 1

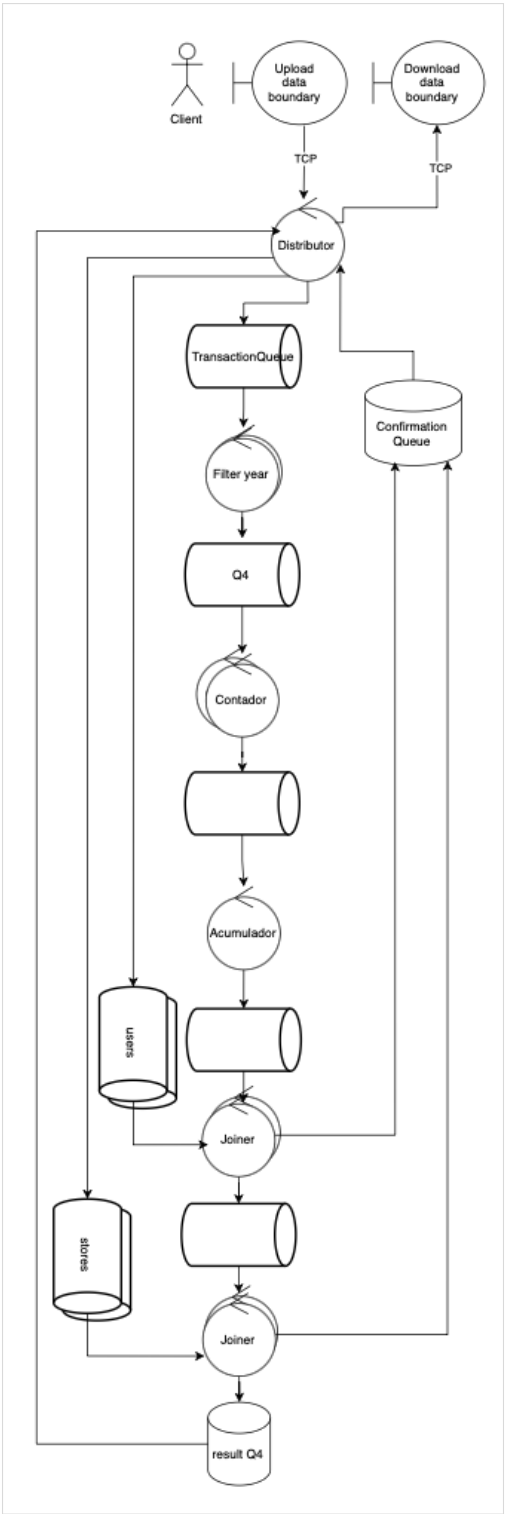


(b) Diagrama de Robustez 2

Figura 4: Diagramas de Robustez (Q1–Q4)



(a) Diagrama de Robustez 1



(b) Diagrama de Robustez 2

Figura 5: Diagramas de Robustez (Q1–Q4)

2. Confirmation Queue (joiners)

Dado que la cantidad de usuarios puede ser mucho mayor que la cantidad de transacciones, se implementó un mecanismo de coordinación para que los *joiners* indiquen cuándo están listos para comenzar a recibir los datos de un `client_id` y poder construir correctamente los resultados de las queries.

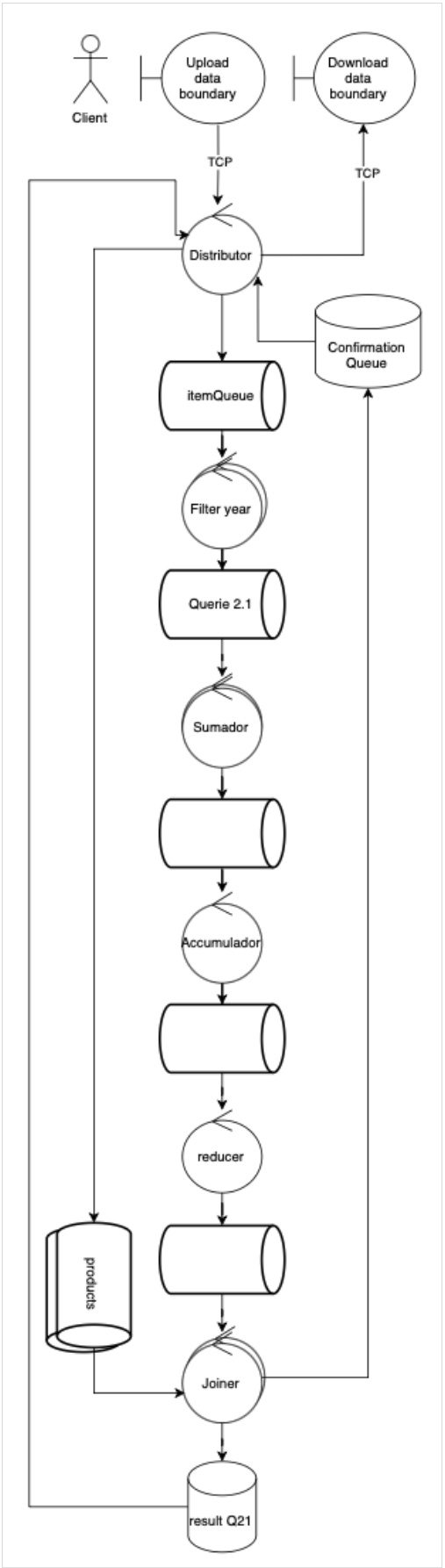
El mecanismo es sencillo: el *distributor* mantiene una cola de confirmación donde cada *joiner* envía una notificación cuando recibe el `end_batch` correspondiente a los datos a *joinear* de un cliente. Una vez que el *distributor* recibe todas las confirmaciones esperadas, puede informar al cliente que está habilitado para comenzar a enviar las transacciones que serán procesadas.

Cada *joiner* opera con dos hilos:

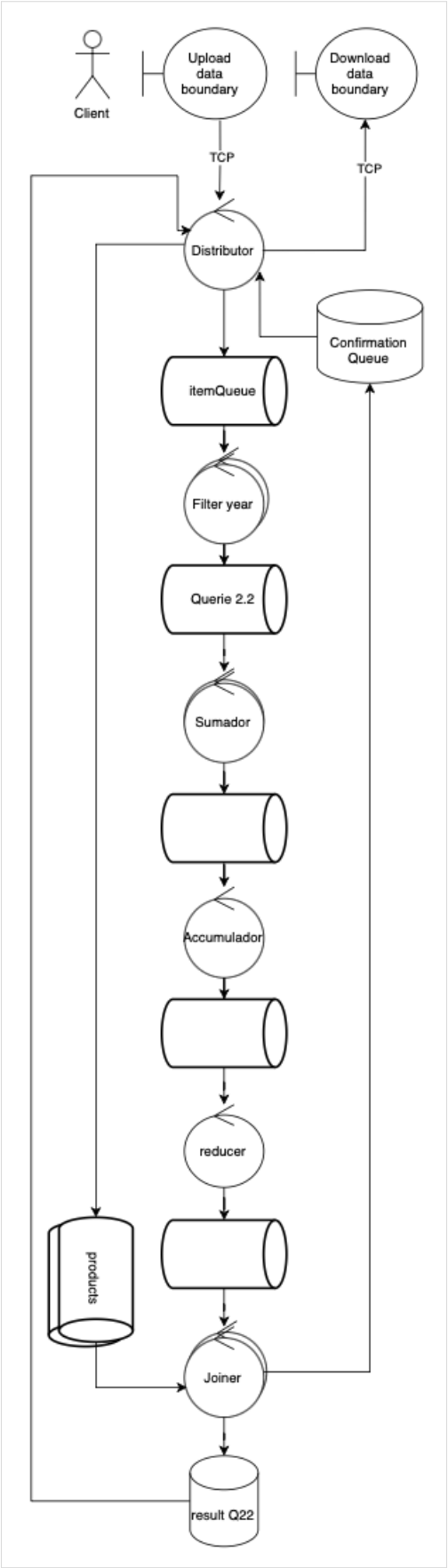
- Hilo principal, que maneja la *callback* del flujo de *batches* correspondientes a las queries.
- Hilo secundario, dedicado a recibir los pares clave–valor necesarios para construir el diccionario (es decir, recibe el id–valor que posteriormente será usado para reemplazar el dato en el hilo principal).

Además, los nodos de *join* aplican *sharding* para distribuir la carga y evitar que un único nodo concentre todo el trabajo.

Finalmente, cuando el *distributor* ha recibido la cantidad esperada de `end_batch` en la *confirmation queue*, el sistema puede considerar que la fase de preparación está completa y que todos los *joiners* están listos. En ese punto, el *distributor* comienza a *flushear* las transacciones hacia los nodos para su procesamiento.



(a) Diagrama de Robustez 3



(b) Diagrama de Robustez 4

Figura 6: Diagramas de Robustez (Q21–Q22)

3. Filtro de duplicados e `IDRangeCounter`

Los nodos que no realizan *buffering* (como *filters* y *aggregators*) no necesitan verificar duplicados, ya que procesan cada *batch* de manera independiente. Estos nodos no buscan resultados globales, sino resultados locales dentro de cada *batch*, por lo que el control de duplicados no es necesario.

En cambio, el nodo *reducer* y el nodo *accumulator* (este último forma parte del procesamiento interno del *aggregator*) sí deben evitar reprocesar *batches*. Para esto utilizan una estructura especializada llamada `IDRangeCounter`, encargada de gestionar la detección y el almacenamiento eficiente de los IDs ya procesados.

La motivación detrás de `IDRangeCounter`, en lugar de mantener simplemente una lista de IDs vistos, es optimizar tanto el tiempo de búsqueda como el consumo de memoria. Las funciones más importantes de esta clase son:

- `already_processed(id)`: determina si un ID ya fue procesado.
- `add_id(id)`: marca un ID como procesado.

Ambas funciones son, en la práctica, aproximadamente $O(1)$. Esto es posible porque internamente la estructura usa un `set` para almacenar IDs aislados y una lista ordenada de rangos para representar intervalos contiguos de IDs.

Dado el ordenamiento natural de los *batches* en el sistema (se originan ordenados y rara vez se desordenan), esta lista de rangos suele mantenerse extremadamente pequeña —típicamente entre 1 y 3 elementos— lo que permite tratar su búsqueda como constante. La misma propiedad aplica al consumo de memoria: la estructura prácticamente no crece y se mantiene de tamaño constante.

Volviendo al comportamiento de los nodos: tanto el *reducer* como el *accumulator* utilizan `IDRangeCounter` para verificar si el *batch* recibido en la *callback* ya fue procesado. Si ya está registrado, se hace *ack* y se descarta; de lo contrario, se procesa normalmente.

3.1. Diagrama de Secuencia: cierre de joins

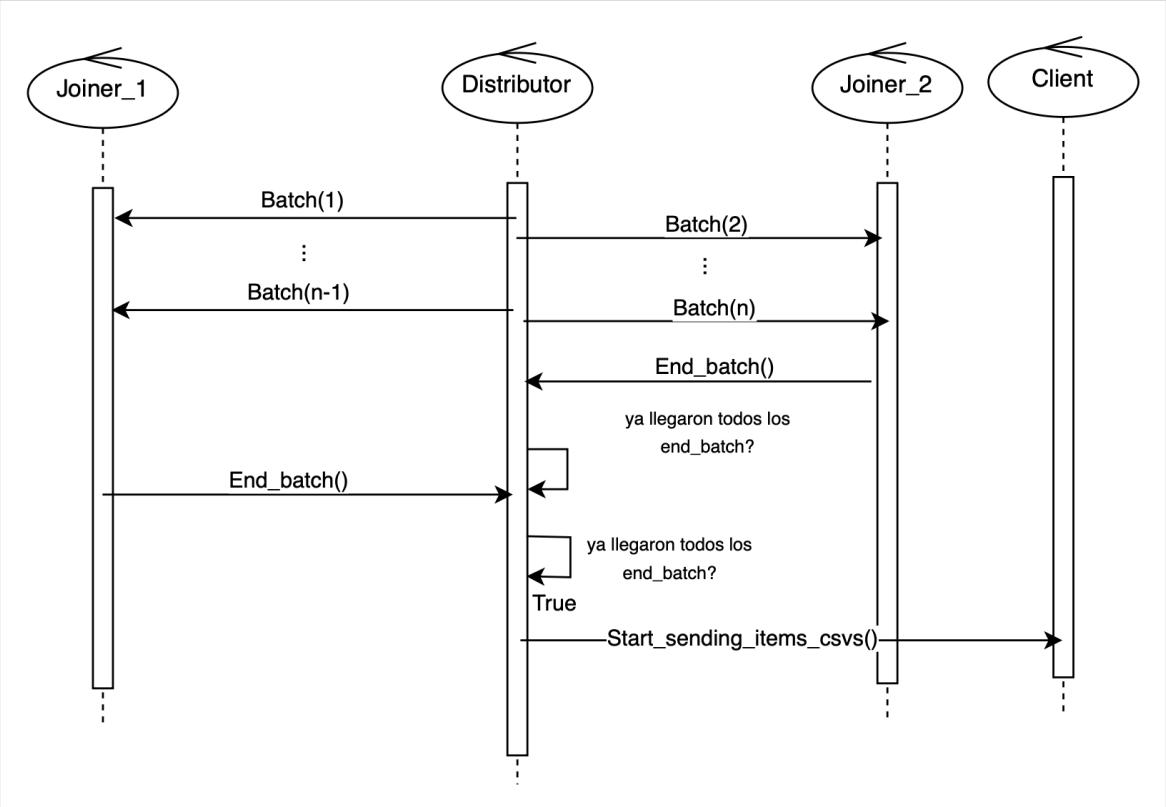


Figura 7: Diagrama de Secuencia: cierre de joins; al recibir todos los `end_batch`, el distributor envía los resultados al cliente

3.2. Diagrama de Secuencia: procesamiento en filtros

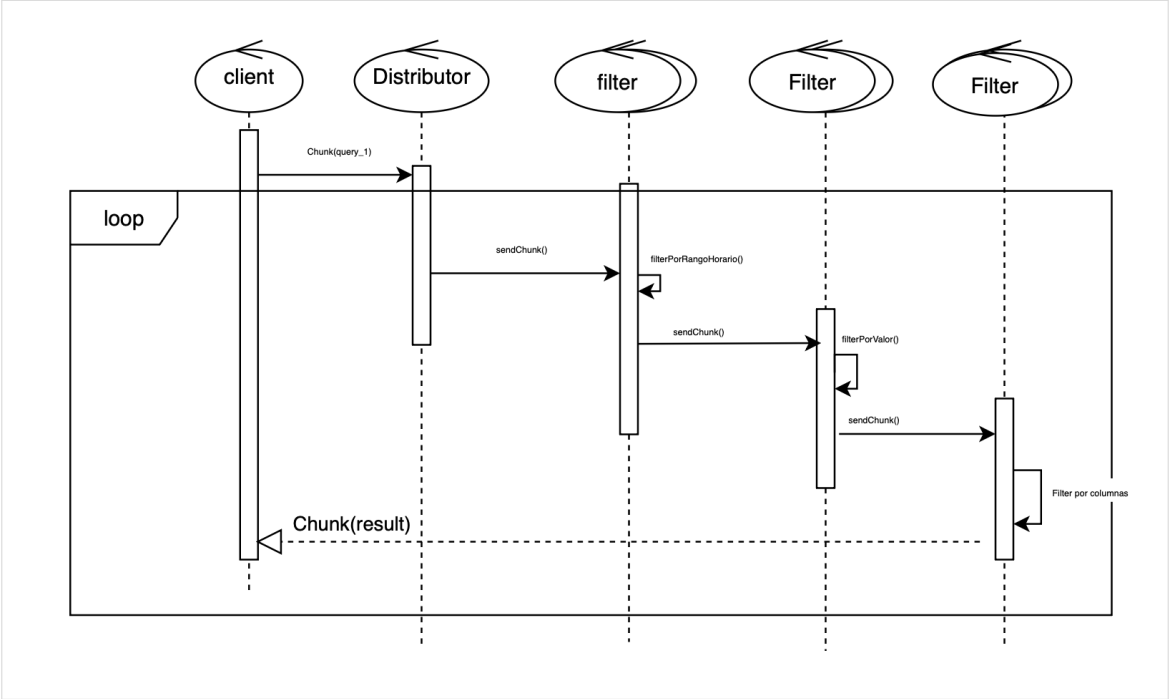


Figura 8: Diagrama de Secuencia: cada filtro procesa y reencola el chunk hasta devolver el resultado al cliente.

3.3. Diagrama de Secuencia: distributor-cliente

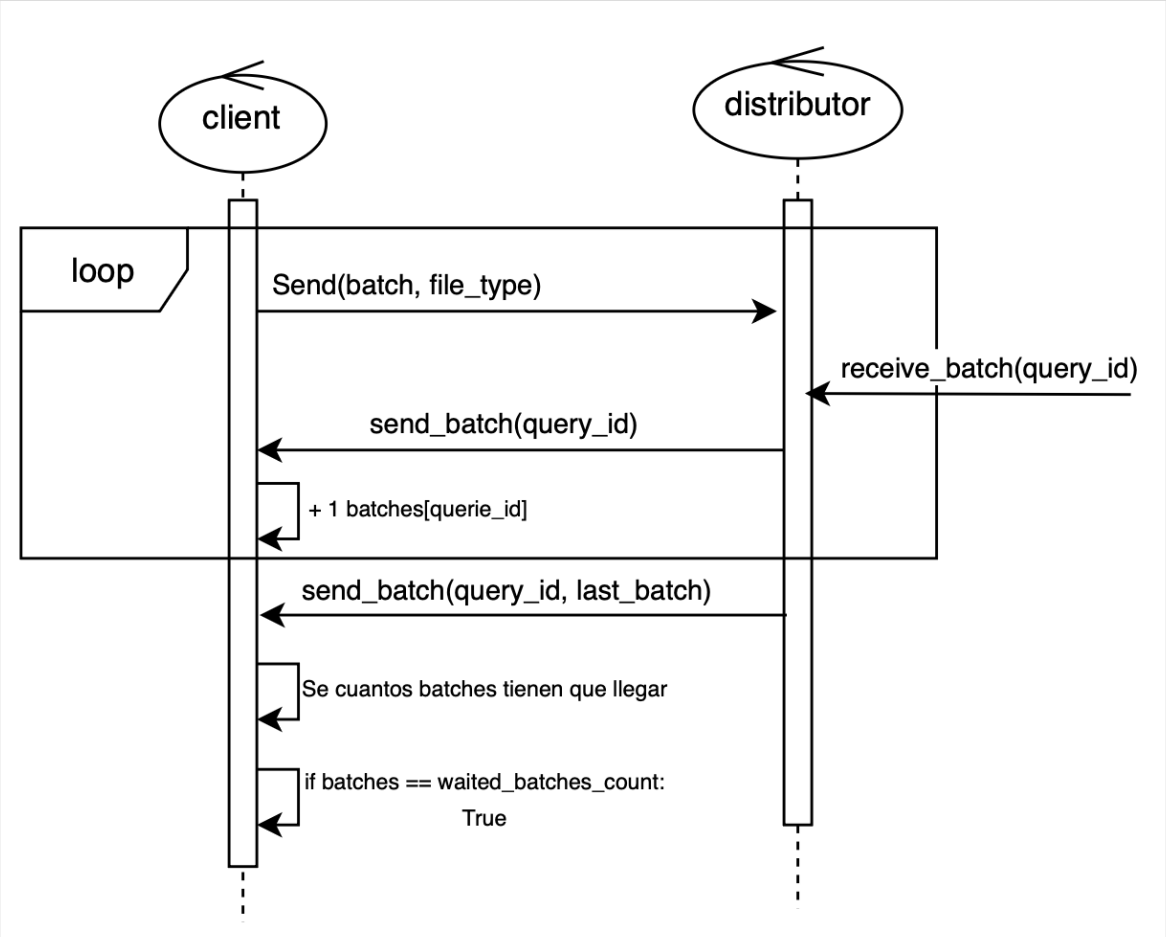


Figura 9: Diagrama de Secuencia: el cliente envía *batches*; el distributor los cuenta por query y, al recibir `last_batch`, compara `waited_batches_count` para darla por completa

4. Healthchecker

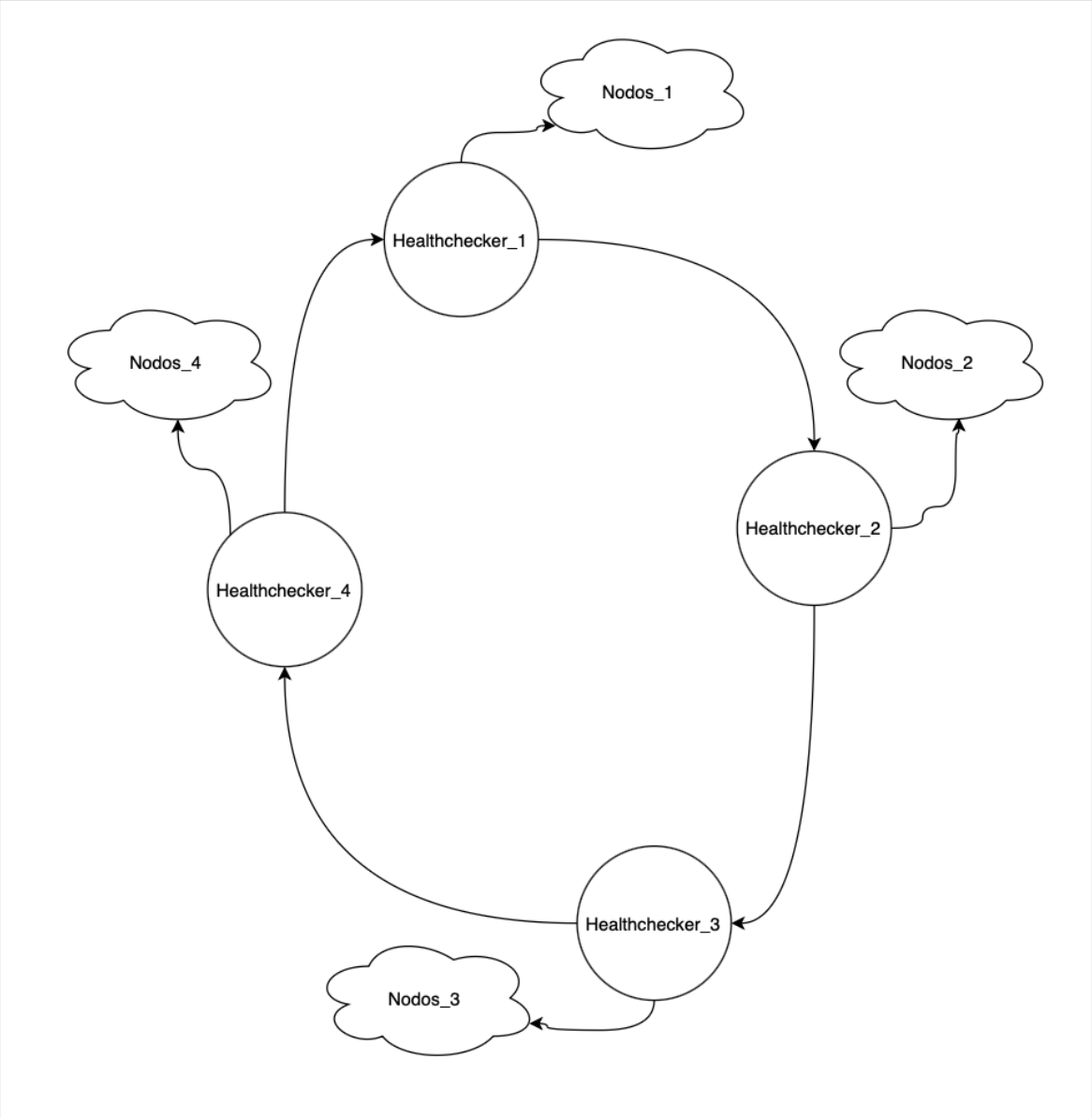


Figura 10: Healthchecker

El sistema cuenta con un anillo de nodos llamados *healthcheckers*, responsables de detectar caídas de nodos y revivirlos cuando sea necesario. Cada healthchecker mantiene una lista con los nombres de los nodos que debe monitorear. Cada nodo aparece una única vez en estas listas, garantizando que no existan dos healthcheckers controlando el mismo nodo.

Además, ninguna lista contiene todos los nodos de un mismo tipo. Esto asegura que, en caso de que se caigan simultáneamente todos los nodos de un tipo particular junto con un healthchecker, al menos otro healthchecker seguirá activo y podrá revivir rápidamente alguno de esos nodos, evitando demoras en la recuperación del sistema.

Junto con los nodos workers, cada lista incluye también el healthchecker siguiente en el anillo. Por ejemplo, `healthchecker_1` supervisa a `healthchecker_2`, este supervisa al que sigue, y así sucesivamente, mientras que el último supervisa al primero. De este modo, los healthcheckers se monitorean entre sí formando un anillo de control mutuo.

Persistencia y mecanismos de recuperación

La persistencia del estado en los accumulators, encargados de bufferizar lo preprocesado por los aggregators, se implementa mediante un Write-Ahead Log (WAL) complementado con snapshots periódicos. Cada vez que se recibe un batch, el nodo registra de forma atómica la operación en un archivo WAL correspondiente al cliente. Este registro incluye tanto las filas procesadas como el identificador del batch, lo que permite asegurar que, ante una caída inesperada, el nodo pueda recuperar exactamente el mismo estado previo reproduciendo los bloques de operaciones en el orden en que fueron confirmados.

Si únicamente se acumularan entradas en el WAL, el archivo crecería de manera indefinida. Para evitarlo, el nodo realiza una compactación cada cierta cantidad de operaciones procesadas. En ese momento, el estado acumulado se serializa en un snapshot en formato JSON. Este snapshot contiene:

- el número total de batches esperados,
- cuántos ya fueron recibidos,
- el tipo de operación,
- el resultado acumulado hasta ese momento,
- y el estado del IDRangeCounter.

El snapshot se escribe mediante un archivo temporal y la función `replace` del sistema operativo, garantizando una actualización atómica. Una vez generado el snapshot, el WAL incremental se elimina y se reinicia desde cero, evitando crecimientos ilimitados y reduciendo el tiempo de recuperación: ante un reinicio, primero se carga el snapshot (si existe) y luego solo las operaciones registradas en el WAL posteriores al último checkpoint.

Este esquema garantiza que:

- nunca se pierda estado, incluso ante fallas entre batches,
- la recuperación sea eficiente, sin necesidad de ejecutar todo el historial,
- el tamaño del WAL se mantiene acotado, ya que se compacta regularmente.

El joiner también utiliza un WAL para registrar las claves y valores recibidos y poder determinar qué datos deben reemplazarse en caso de reinicio. A diferencia del accumulator, no utiliza snapshots: en su lugar emplea una *stopword*, que permite identificar si una línea del WAL está incompleta o corrupta y, por lo tanto, debe ser ignorada durante la recuperación. Esto funciona porque el estado del joiner es efímero y está directamente vinculado al flujo de batches entrantes, por lo que no justifica la complejidad de generar snapshots.

El reducer, en cambio, siempre genera snapshots, independientemente de la cantidad de batches procesados. Esto es posible porque su estado interno es pequeño y estable: consiste en agregados parciales y estructuras compactas que no crecen proporcionalmente al volumen de datos procesados. Dado que el costo de serializar este estado es bajo, el reducer puede realizar un snapshot en cada actualización, lo que garantiza una recuperación extremadamente rápida y simple ante fallas.

En resumen, cada nodo emplea un mecanismo de persistencia distinto porque su comportamiento, volumen de estado y patrón de acceso son también distintos. Este enfoque mejora la durabilidad, el uso de memoria, el tiempo de recuperación y el costo operativo de cada componente del sistema, aplicando en cada caso la estrategia más eficiente para la naturaleza del nodo.