Documento de Arquitetura

Sistemas Distribuidos I

Grupo 23

Joaquin Mendaña	108485
Ivan Pfaab	103862

Historial de versiones

Version	Fecha
V1.0	18/09/2025
V1.1	20/09/2025
V1.2	02/10/2025
V2.0	07/10/2025
V2.1	18/10/2025
V2.3	01/10/2025
V3.0	01/12/2025
V3.1	04/12/2025

Tabla de contenidos

Alcance del sistema

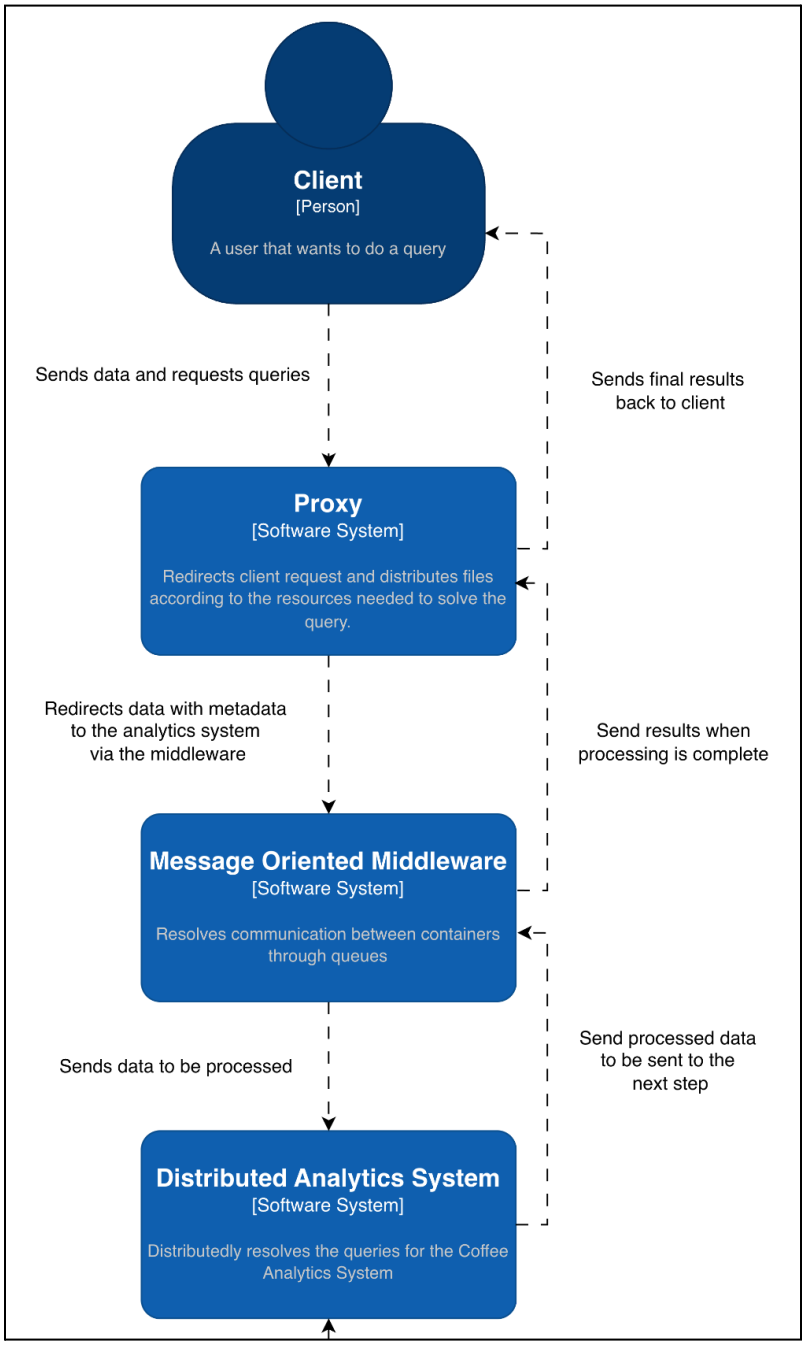
El sistema propuesto, denominado **"Distributed Coffee Analytics System"**, tiene como objetivo principal analizar la información de ventas de una cadena de cafeterías en Malasia.

Este sistema debe procesar datos transaccionales, de clientes, tiendas y productos para responder a cuatro consultas específicas:

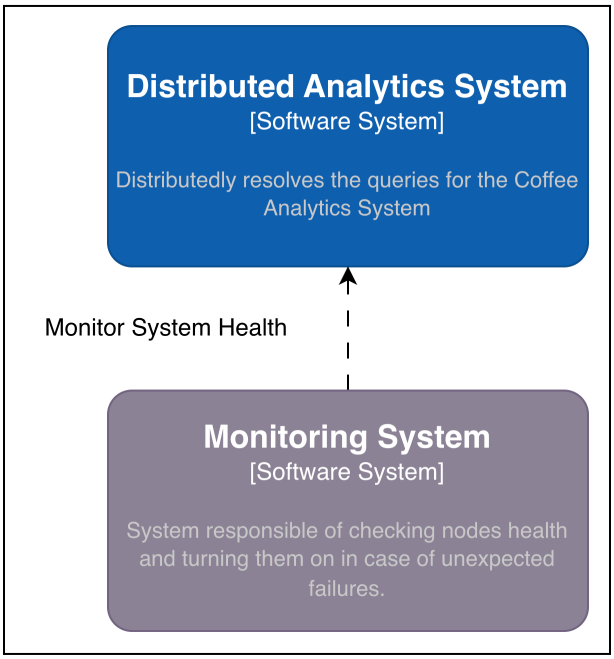
1. **Filtrado de transacciones**: Transacciones de 2024-2025, de 6 AM a 11 PM, con monto superior a $75.
2. **Análisis de productos**: Productos más vendidos y más rentables por mes en 2024-2025.
3. **TPV por sucursal**: Total Payment Value por semestre en 2024-2025, por sucursal, para transacciones de 6 AM a 11 PM.
4. **Top 3 clientes**: Fecha de cumpleaños de los 3 clientes con más compras por sucursal en 2024-2025.

Nivel 1 - Contexto General

El sistema se compone de tres elementos principales. El **Usuario** interactúa con el **Proxy** para solicitar consultas y enviar datos. El **Proxy** transforma la data para que luego el **Distributed Analytics System** se encargue de resolver las consultas. Los resultados son enviados nuevamente al **Proxy** que los disponibiliza al cliente.



Adicionalmente, un sistema por fuera de lo que es el ***Distributed Analytics System*** está encargado de monitorear y controlar la salud del mismo.
El mismo se encargará de realizar chequeos de estado de workers, junto con su levantamiento en casos donde alguno de ellos haya dejado de funcionar.



Nivel 2 - Contenedores y Responsabilidades

Para resolver la consulta, haciendo zoom-in al **Distributed Analytics System,** tenemos los siguientes containers principales que se encargan de la resolución lógica de la consulta.

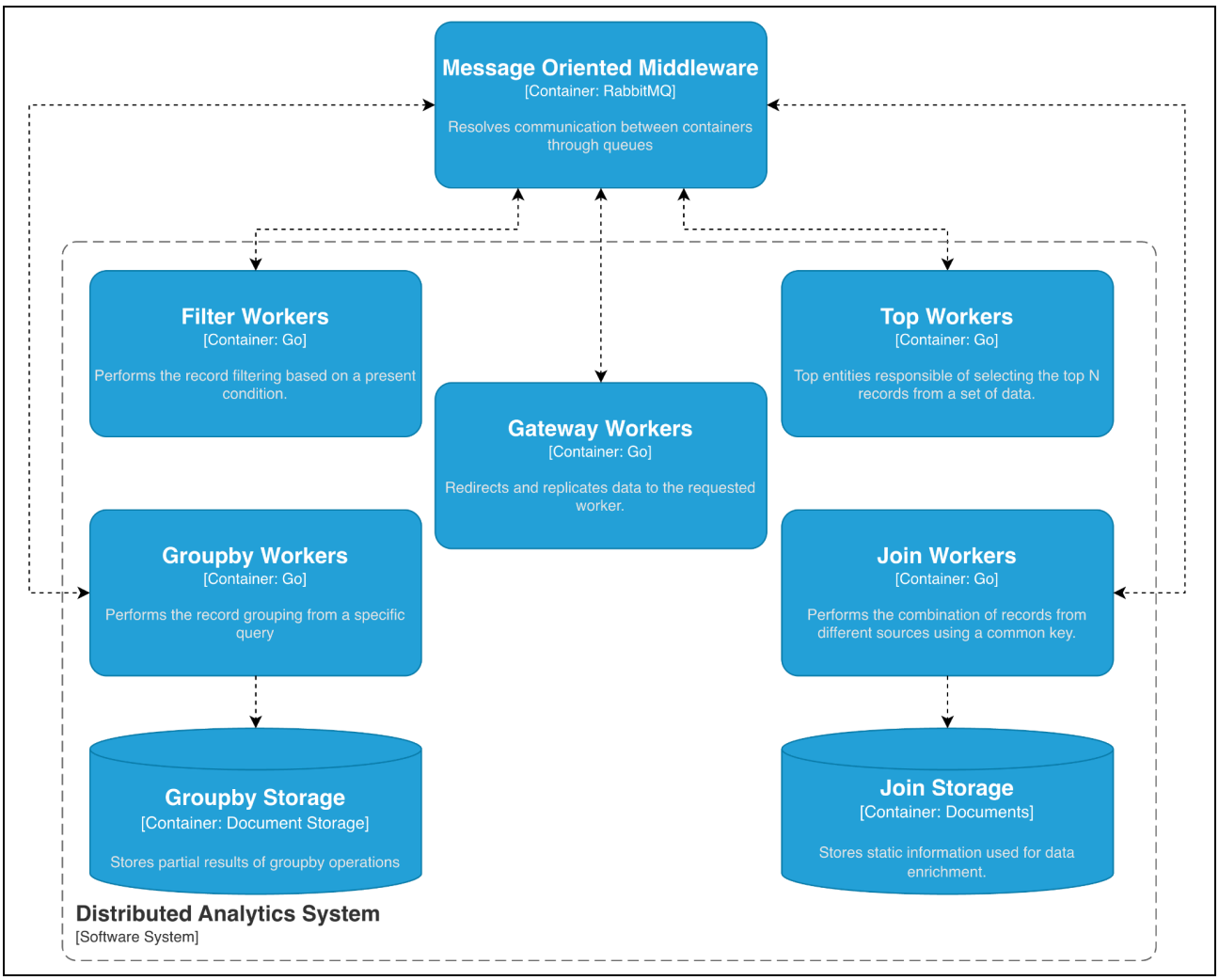


Figura 2

Entrando en detalle con cada container tenemos:

- **Client (Go)**: La interfaz del usuario para enviar solicitudes de consulta y datos.
- **Proxy (Go)**: El punto de comunicación entre el cliente y el sistema. Aquí es donde se redirigen las solicitudes y data del cliente a través del **Message Oriented Middleware,** y a través de donde el cliente recibe sus resultados.
- **Message Oriented Middleware (RabbitMQ)**: Un componente crucial que resuelve la comunicación asíncrona entre los diferentes contenedores a través de colas de mensajes, asegurando un flujo de trabajo distribuido.

- **Processing Workers (Go)**: Un conjunto de componentes especializados en realizar las transformaciones y análisis requeridos por las consultas.

Flujos de Datos

Cada consulta solicitada tiene un flujo de datos específico, representado por un **Diagrama DAG** .

- **Consulta 1 (Filtro)**: Consiste en una secuencia de filtros para obtener las transacciones que cumplen las condiciones de fecha, hora y monto.

- **Consulta 2 (Productos)**: Involucra un Filter por año, un Group By por año, mes e item, y una agregación final para obtener los productos top. Para devolver el nombre del producto se hace un Join con la tabla de items.

- **Consulta 3 (TPV)**: Involucra un Filter por año y por hora del dia, un Group By por año, semestre y store ID. Para devolver el nombre de la tienda se hace un Join con la tabla de stores.

- **Consulta 4 (Clientes)**: Un flujo que filtra por año, hace un Group By por user id, identifica los top 3 por cada tienda y luego realiza un Join con la tabla de usuarios para obtener las fechas de cumpleaños.

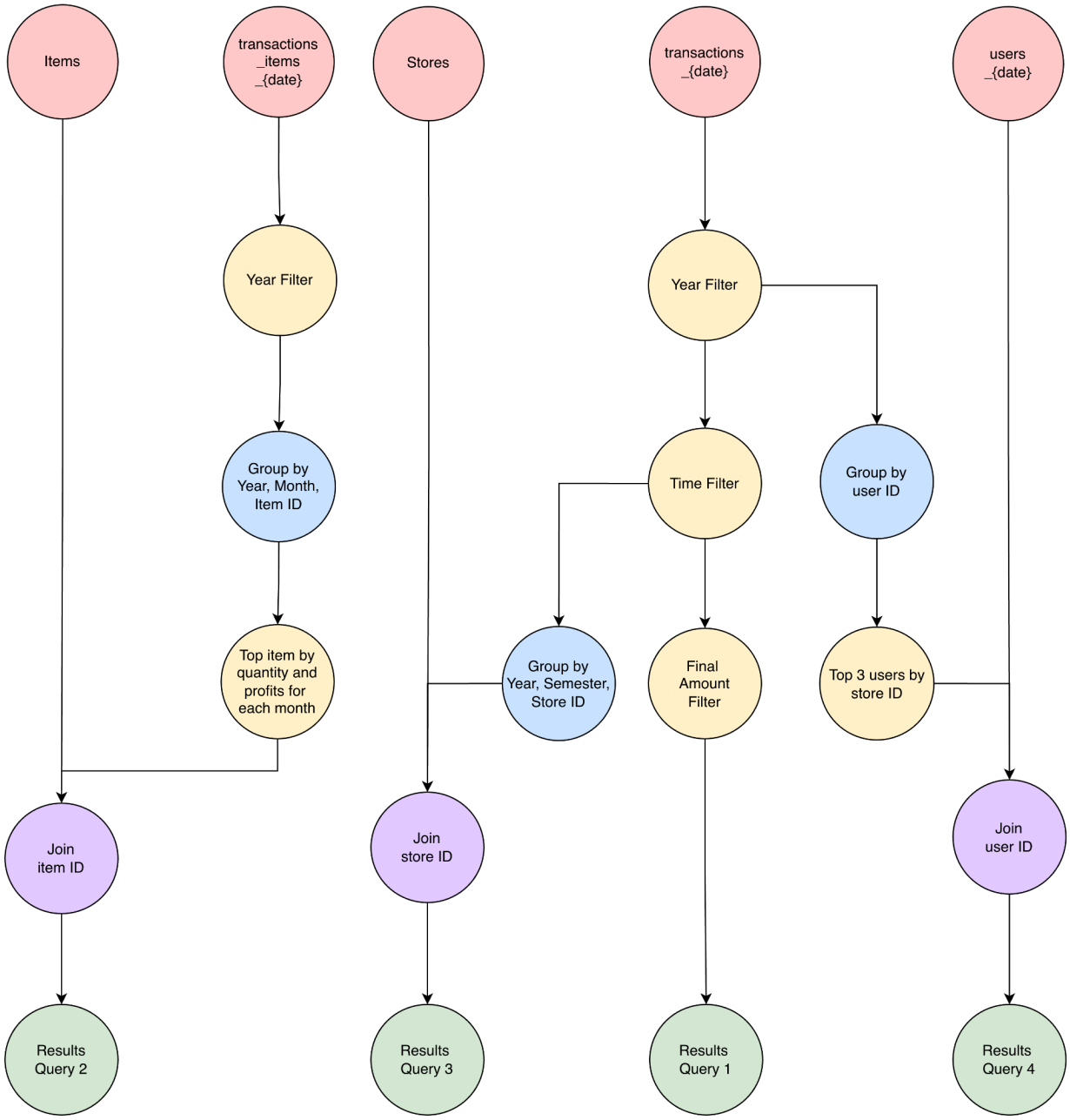

Items
transactions _items _{date}
Stores
transactions _{date}
users _{date}
Year Filter
Year Filter
Group by Year, Month, Item ID
Time Filter
Group by user ID
Top item by quantity and profits for each month
Group by Year, Semester, Store ID
Final Amount Filter
Top 3 users by store ID
Join item ID
Join store ID
Join user ID
Results Query 2
Results Query 3
Results Query 1
Results Query 4

Figura 17

Nivel 3 - Componentes

En esta sección se desglosa con más detalle los 5 contenedores principales *(se excluye el Middleware)* detallados en las dos secciones anteriores, enfocándonos en las entidades principales dentro de ellos y su relación.

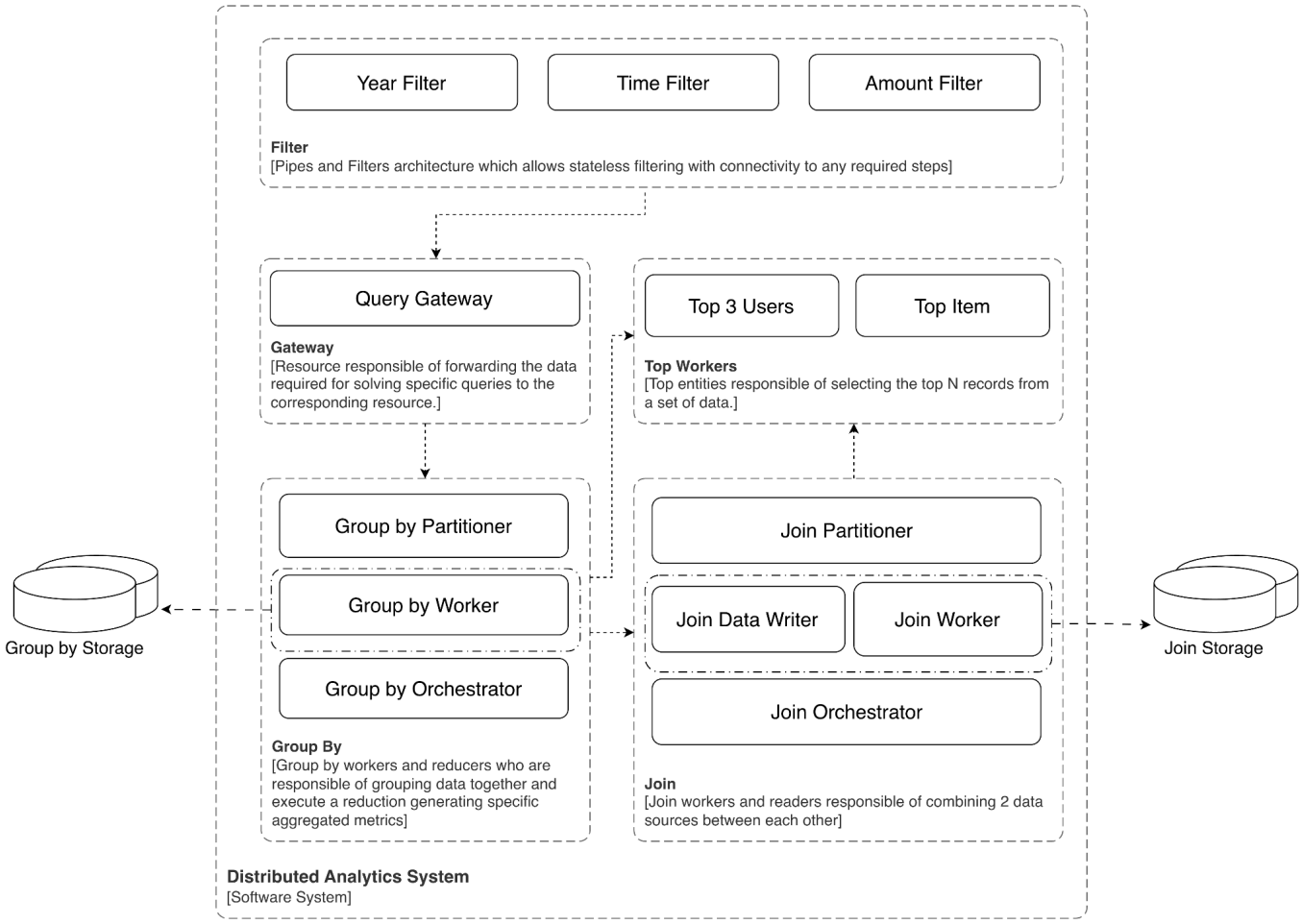


Figura 3

El cliente es el punto de inicio de la interacción con el sistema. Envía una solicitud de consulta que es recibida por el **Client Request Handler.** Esta entidad envía la data recibida al **Data Handler**, quien transformará la información recibida y agregará metadata necesaria para poder resolver las consultas. El Data Handler es también el encargado de redirigir la data a los componentes correspondientes.

Una vez que el procesamiento ha finalizado, el cliente recibe el resultado a través del Client Request Handler, quien consume del **Results Dispatcher** los resultados finales del procesamiento.

Para comenzar el flujo de resolución, todos las consultas pasan por una secuencia de **Filtros.** Una de las consultas (Query 1) resulta aplicando estos filtros y puede ser retornada al cliente a través del **Results Dispatcher**. Para las consultas restantes, el conjunto de paquetes filtrados es enviado al **Query Gateway** quien se encarga de redirigirlos al flujo de agrupación que les corresponda.

El conjunto de componentes de **Group By** es responsable de agrupar registros y calcular métricas dependiendo el tipo de consulta que deban resolver. Específicamente, las métricas calculadas para la query 2 son cantidad de productos y profit generado por ellos, la query 3 son la cantidad de transacciones por store, y por último la query 4 son la cantidad de compras por usuario.

Una vez la data fue agrupada, las query 2 y 4 pasan por un **Top Worker**, responsable de quedarse únicamente con los registros con las métricas más grandes, y la query 3 es directamente redireccionada hacia los workers de **Join**.

Dentro de los workers de **Join** las query 2, 3 y 4 serán enriquecidas con data proveniente de otros archivos, para finalmente ser comunicada al **Results Dispatcher** y por último al **Cliente** a través del **Client Request Handler.**

Para ver un flujo más detallado de la resolución y flujo del sistema, por favor referenciar a la sección Flujo de Datos.

Vista de Despliegue

La solución se despliega en una arquitectura de múltiples nodos, siendo estos los siguientes:

- **Client**: El cliente, una aplicación separada, que interactúa con el sistema.
- **Server**: Contiene el Client Request Handler y el Data Handler, sirviendo como el punto de entrada y de control.
- **MO Middleware**: Alojado en un servidor independiente, RabbitMQ gestiona la comunicación entre todos los nodos de procesamiento.
- **Filters**: Nodos encargados del procesamiento de los distintos tipos de filters sobre los datos. Un nodo de filter no es capaz de realizar todos los distintos tipos de filters. Existen nodos de filters encargados de realizar el *Time Filter*, otros de realizar el *Amount Filter,* etc. Podemos configurar la cantidad de nodos que queramos para cada tipo de filter.
- **Query gateway:** Nodo encargado de redirigir los datos de las queries 2, 3, y 4, hacia el flujo de group by correspondiente a la query en cuestión.
- **Group by partitioner:** Nodos encargados de particionar los chunks (datos) recibidos para una query según corresponda. En los casos de las queries 2 y 3, los datos se particionarán según el semestre. Esto debido a que los Group By workers de estas queries se distribuyen de manera tal que un group by worker solo procesa datos de un mismo semestre. En el caso de la query 4, los datos se particionan en base al módulo del user_id. Nuevamente, esto es debido a que cada group by worker tiene asignado los usuarios con un módulo específico. Los nodos partitioner son particulares a una query en específico. Podremos definir la cantidad de partitioners para la query 2, para la query 3, y para la query 4, a conveniencia.
- **Group by worker:** El group by worker es el encargado de escribir la data en disco que luego sea agrupada. Nuevamente, podemos definir la cantidad de workers para cada query a conveniencia.
- **Group by aggregator**: El aggregator se encarga de leer la data almacenada en disco, agruparla y enviarla al siguiente paso del pipeline.
- **Group By Orchestrator:** El orchestrator se encarga de identificar cuándo hemos terminado de escribir todos los datos en disco, para que luego los mismos puedan ser agrupados por el worker mencionado en el item anterior.
- **Join Data Handler:** El Join Data Handler es el encargado de redireccionar los chunks correspondientes a los archivos a ser Joineados por los Join workers (stores, menu_items, y users_*). Podemos configurar la cantidad de nodos de este tipo a conveniencia.
- **Join Partitioner:** Nodos de cantidad configurable que se encargan de particionar los chunks recibidos a joinear en la query 4 según el módulo del user_id.
- **Join Data Writer:** Nodos de cantidad configurable que escriben en disco los archivos a ser joineados por los Join Workers. La cantidad N de nodos de este tipo define el módulo a usar para particionar los usuarios, y cada writer se encarga de un módulo i en particular, del 0 al n-1. Los archivos que escriben son definidos por {clientid}-{usermod}, por lo que un archivo no será manipulado por más de un writer a la vez.

- **Join Orchestrator:** Presentamos 2 nodos orchestrator para los Joins, que son los siguientes.
    - Join Orchestrator: Encargado de definir cuándo hemos terminado de procesar el join. Esto es necesario para poder luego limpiar los recursos de los join workers.
    - Join Writer Orchestrator: Encargado de definir cuándo hemos terminado de escribir los archivos de usuarios de un cliente en disco (query 4). Una vez hemos escrito todo, le notificamos a los join workers que pueden empezar a procesar los chunks de ese cliente.
- **Join Worker:** Nodos de cantidad configurable (puede ser distinta a la cantidad de nodos de las categorías previas) encargados del procesamiento del Join per se. Estos nodos leerán data correspondiente a su partición, ya que se buscó que cada join worker tenga un volumen relacionado al módulo
- **Results Dispatcher:** Único nodo encargado de formatear los resultados a enviar a un cliente.

La comunicación entre los nodos se realiza a través de RabbitMQ, lo que desacopla los componentes y permite la comunicación a través de colas.

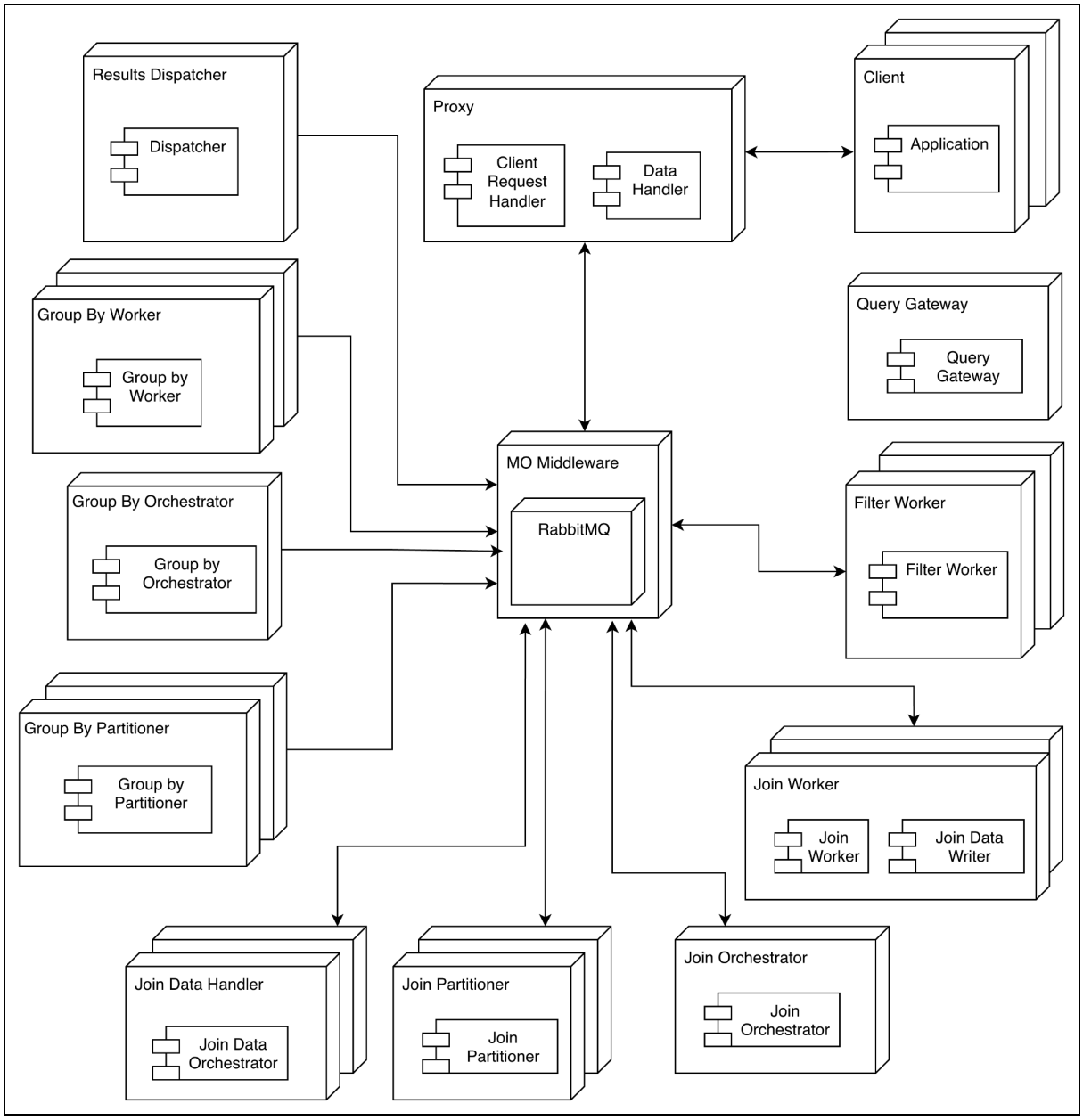
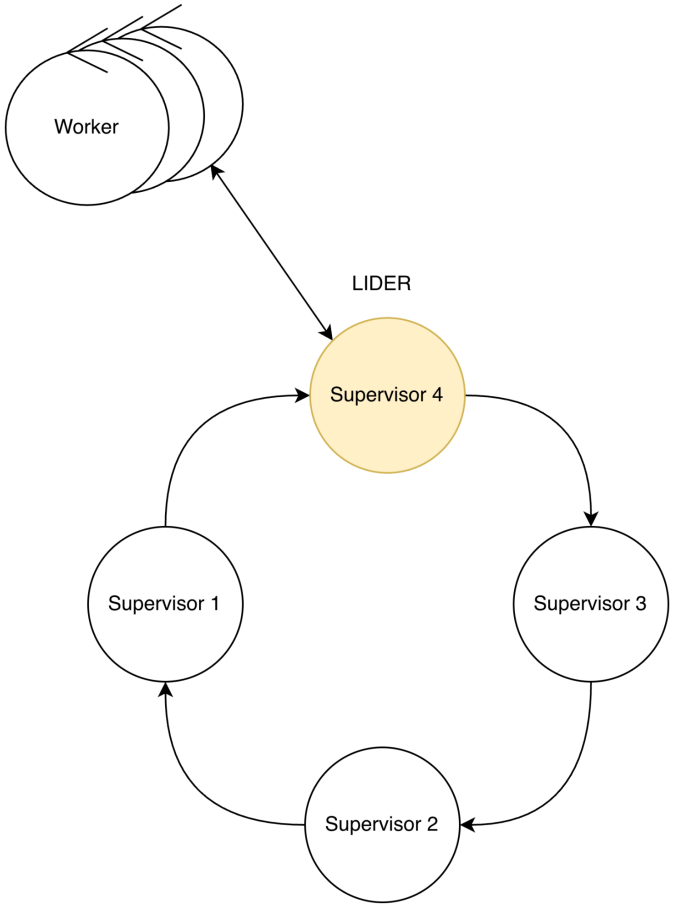

Results Dispatcher
Dispatcher
Proxy
Client Request Handler
Data Handler
Client
Application
Group By Worker
Group by Worker
Query Gateway
Query Gateway
MO Middleware
RabbitMQ
Group By Orchestrator
Group by Orchestrator
Filter Worker
Filter Worker
Group By Partitioner
Group by Partitioner
Join Worker
Join Worker
Join Data Writer
Join Data Handler
Join Data Orchestrator
Join Partitioner
Join Partitioner
Join Orchestrator
Join Orchestrator

Figura 18

Control de fallas

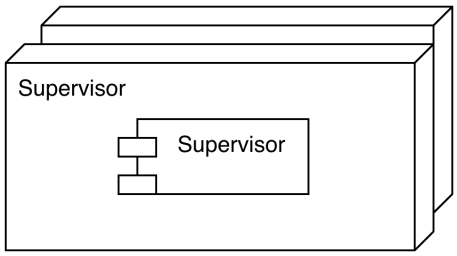
Un sistema de supervisores con elección de líder monitorea la salud de los workers mediante ***probes***. Al detectar fallos, el supervisor líder reinicia automáticamente los containers fallidos. Los workers recuperan su estado al reiniciar y continúan el procesamiento sin pérdida de datos.



Únicamente el nodo líder es quien levantará un worker caído. En caso de que el líder se caiga, una nueva elección comenzará.

El sistema implementa tolerancia a fallos mediante persistencia de estado y detección de duplicados. Los workers que mantienen estado por cliente (orchestrators, dispatchers, workers de agregación) persisten metadata en disco. Al reiniciar, reconstruyen el estado utilizando esta información almacenada y continúan desde donde quedaron, procesando solo clientes incompletos. Todos los workers mantienen registros persistentes de mensajes procesados. Antes de procesar, verifican si el mensaje ya fue procesado, evitando duplicación tras reinicios o reenvíos de RabbitMQ. El sistema detecta y corrige automáticamente escrituras incompletas en disco y particiones causadas por crashes durante operaciones de escritura, garantizando integridad de datos.

Cuando un cliente completa todas sus queries, se dispara un proceso de limpieza que elimina los recursos en disco asociados a la tolerancia a fallos.

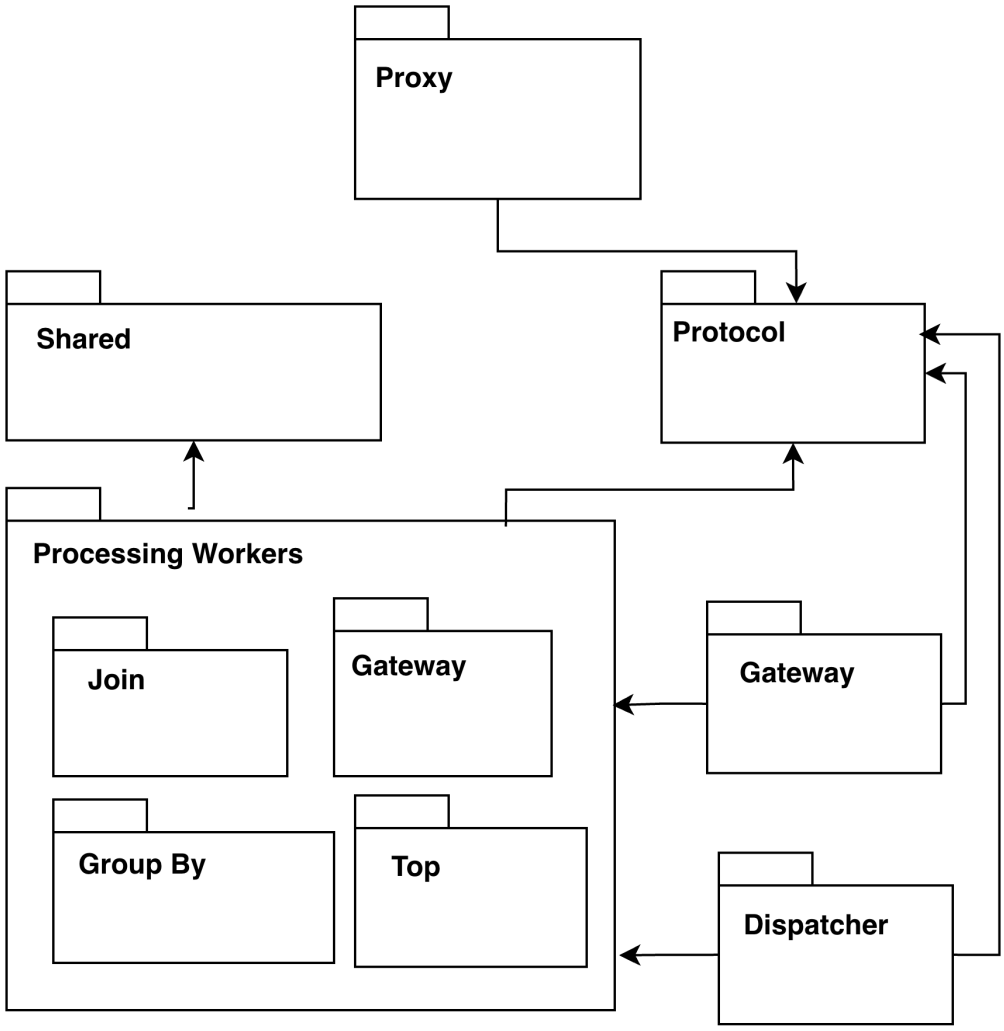


Al ser un subsistema independiente, el despliegue del mismo se puede hacer de manera independiente, **con la salvedad de que el sistema no será tolerante a fallos hasta que este componente haya sido correctamente desplegado.**

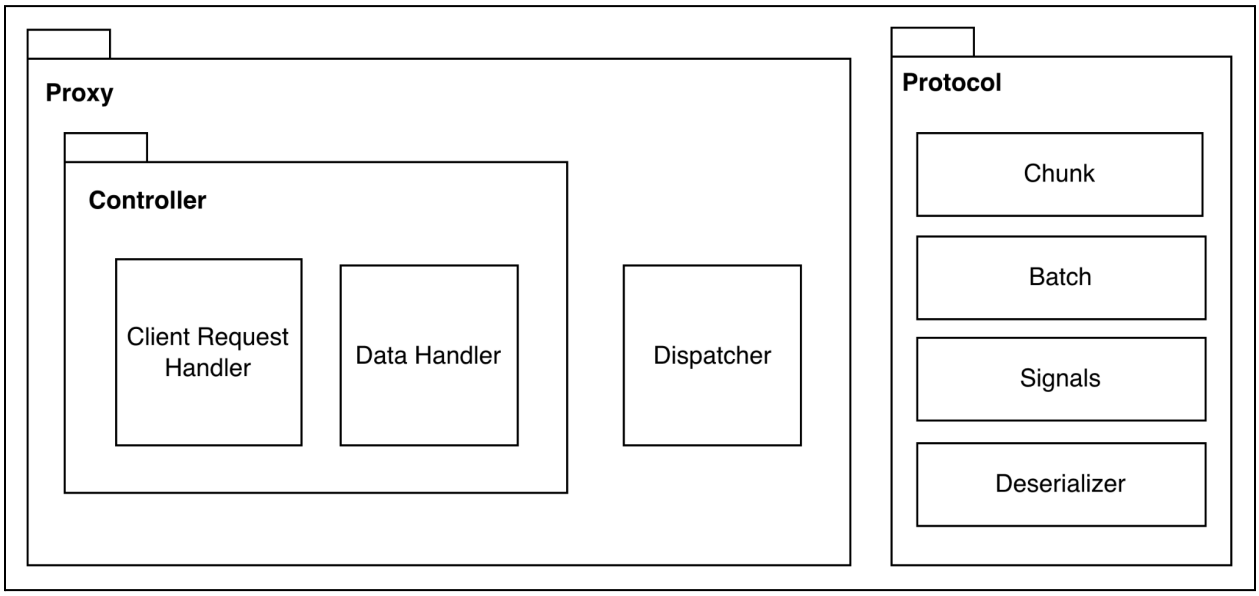
Nivel 4 - Código

Estructura de Paquetes

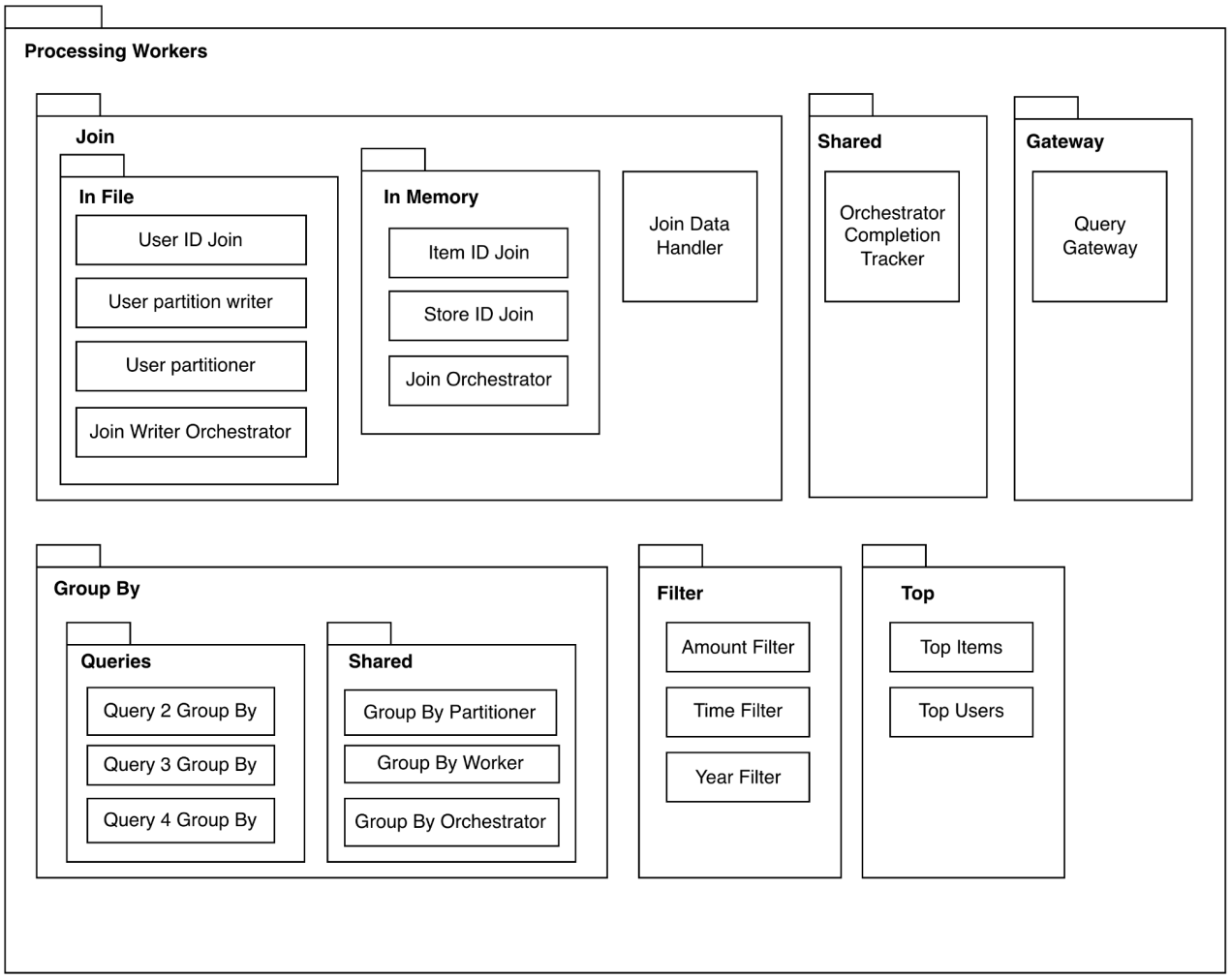
Nuestro sistema se organiza en los siguientes paquetes.



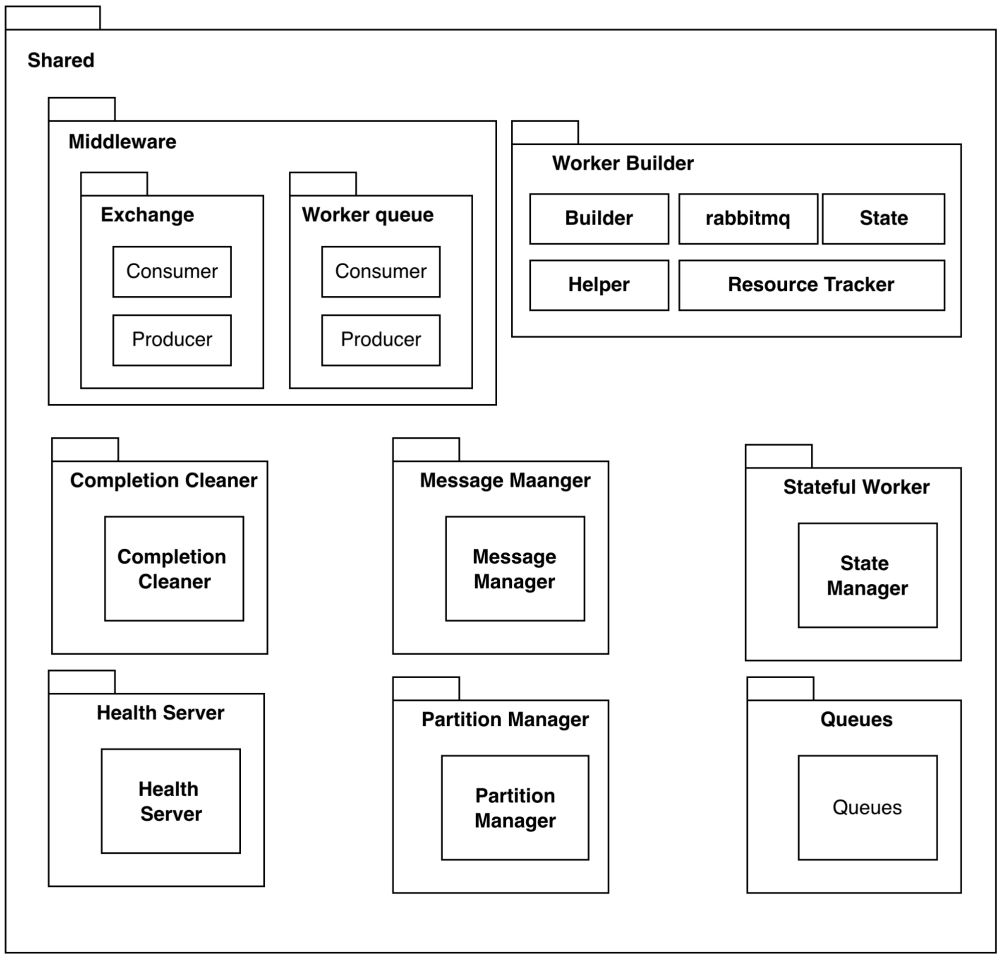
- **Protocol**: Maneja el protocolo de mensajería. Se encarga de la serialización y deserialización de los distintos tipos de mensajes que se envían entre los componentes de nuestro sistema.
- **Proxy**: Contiene el Data Handler y Client Request Handler, responsables de manejar la comunicación con el cliente y redireccionar los datos envíados por y hacia el mismo.
- **Dispatcher**: El Dispatcher o Response Dispatcher es el encargado de formatear y redireccionar la respuesta a enviar a los clientes. Envía la respuesta lista al Client Request Handler quien será el encargado de enviársela al cliente a través de su conexión TCP con el mismo.



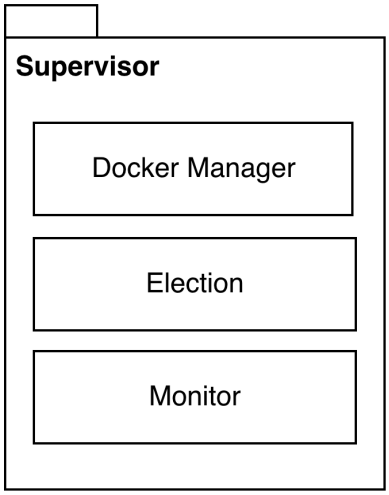
- **Processing Workers**: Agrupa los componentes que realizan las operaciones analíticas.
    - Group By: El Group By presenta una serie de paquetes compartidos para el procesamiento de las distintas queries, así como también paquetes específicos a cada query.
    - Join: presenta paquetes diferenciados para la lógica de join que ocurre en memoria (Query 2, 3) y aquella que consume data en disco (Query 4).
    - Filter: Cada tipo de filter es un paquete diferente.
    - Shared: Se presenta un componente compartido, el Orchestrator Completion Tracker. Nos abstrae y generaliza la lógica de contar los chunks procesados y los archivos completados en una etapa del procesamiento, para poder determinar cuándo ese paso ha sido completado.



- **Shared:** Múltiples recursos compartidos por todos los workers, utilizados principalmente para abstraer lógica que se replica en multiples sectores



- **Supervisor**: Presenta paquetes encargados de manejar las operaciones de docker, determinar la elección del líder y monitoreo de nodos.



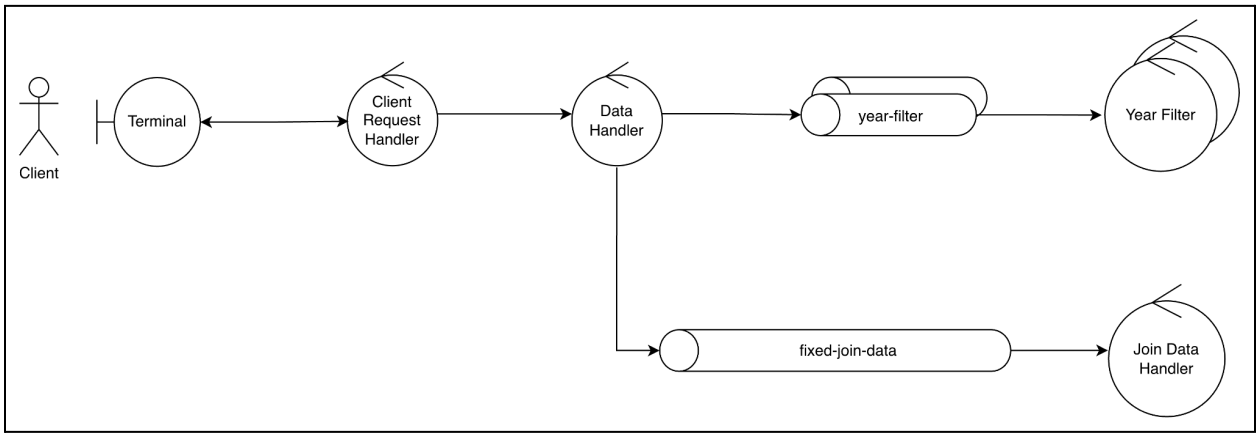
Flujos de Datos

Inicio de consulta

El sistema está diseñado para manejar consultas a través de un **flujo de trabajo de tipo DAG** (Grafo Acíclico Dirigido), garantizando la **paralelización** de determinadas operaciones y procesamiento dirigido orientado hacia el resultado de la consulta.

El flujo comienza con el cliente subiendo los archivos en batches. Estos son interpretados por el Client Request Handler y luego transformados por el Data Handler.

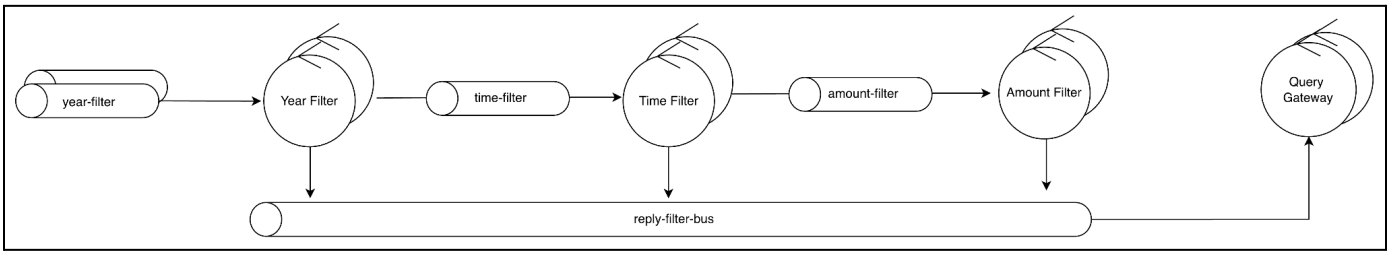
Desde el data handler hay dos flujos posibles, enviar la información hacia los workers de filtrado (esto sucede para los archivos de *transactions* y de *transaction_items*) o enviar los datos hacia el Join Data Handler (esto sucede para los archivos de *stores, users, menu_items*).



Filtrado

Los chunks que son enviados para filtrado, pasan por un proceso de pipe and filtering que es determinado por el tipo de query que les corresponde. Cada query va a tener su set de chunks que irán transitando los distintos filtros donde:

- Query 1: Pasa por los filtros de Year, Time y Amount.
- Query 2: Pasa por el filtro de Year.
- Query 3: Pasa por el filtro de Year y Time.
- Query 4: Pasa por el filtro de Year.



Estos nodos de filtrado son stateless y no guardan ningún dato en memoria, únicamente reciben un chunk, aplican el filtro y lo devuelven.

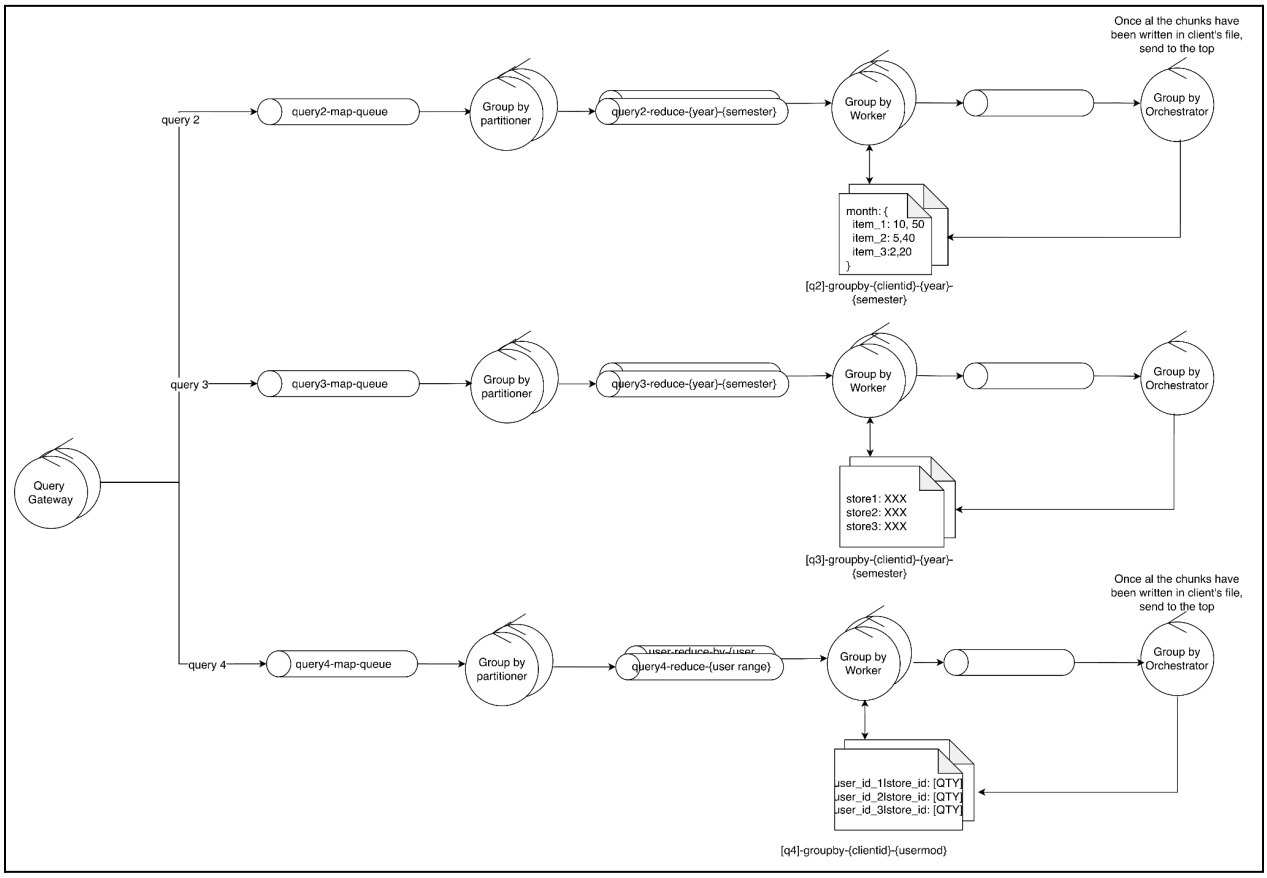
Una vez aplicados los filtros, todos los chunks tienen como destinatario final el controller **Query Gateway**. Este controller se encarga de routear, dependiendo del número de query, los chunks a la cola correspondiente.

- Query 1: ya fue resuelta en su completitud y puede ser enviada al cliente
- Query 2, 3 y 4: tienen que seguir pasando por el pipeline para aplicar otras transformaciones.

Agrupación

Cada una de las queries consume de su propia cola, donde un controller de partición se encarga de consumir los chunks.
Este controller llamado **Group by partitioner** se encarga de subdividir el chunk recibido, en chunks más pequeños, que luego serán consumidos por **Group by Workers**.

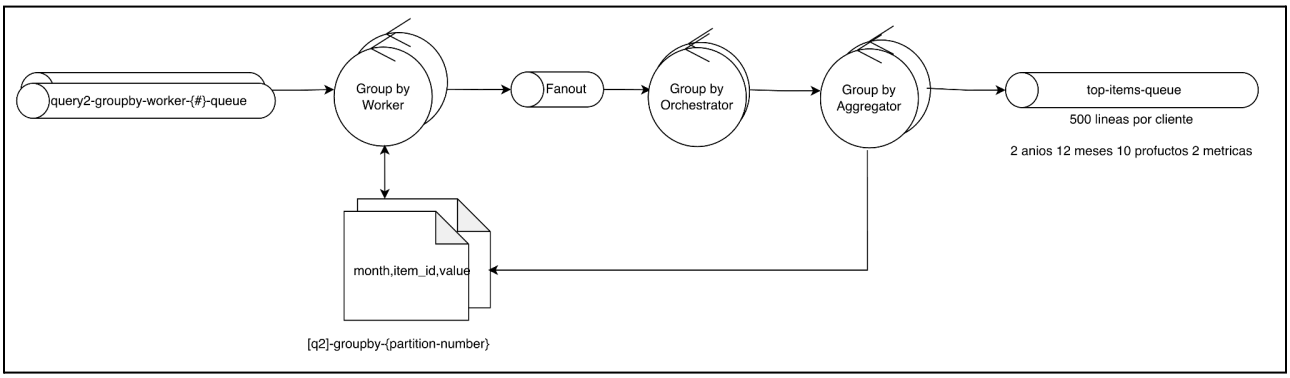
La decisión de tener múltiples particiones y que cada chunks sea redirigido a distintas colas se da principalmente para reducir la dimensión del chunk a procesar,, lo cual permite generar conjuntos disjuntos de escritura en el storage generado en el siguiente paso.



Se decidió bajar a disco y no mantener en memoria las agrupaciones, ya que esto nos permite aumentar la cantidad de clientes que se manejan en paralelo, sin la necesidad de mantener un estado parcial para cada uno en memoria.

La query 2 tiene que agrupar por mes e item, calculando cuantos se compraron y que profit generaron.
Para ello, cada vez que llega un chunk, el **Group by Worker** lee, o crea en caso de que no

exista, de disco, un archivo que sigue la siguiente nomenclatura: *Q2-groupby-{clientID}-{partition-number}*.



El proceso de agrupado es:

1. Se recibe un chunk
1. Se deserializa
2. Se escribe la información en la/las particiones correspondientes
3. Se envia una notificacion de escritura exitosa al group by orchestrator
4. Una vez se terminaron de escribir todos los chunks, el orchestrator avisa al aggregator que ya puede comenzar a agrupar
5. El agregador lee el disco, y envía chunks agrupados.

La metadata del chunk procesado sirve para que luego, el **Group by Orchestrator**, pueda llevar registro *(en memoria)* de cuantos chunks ya fueron procesados para un cliente particular, para poder determinar cuándo hemos terminado de procesar el group by para un cliente.

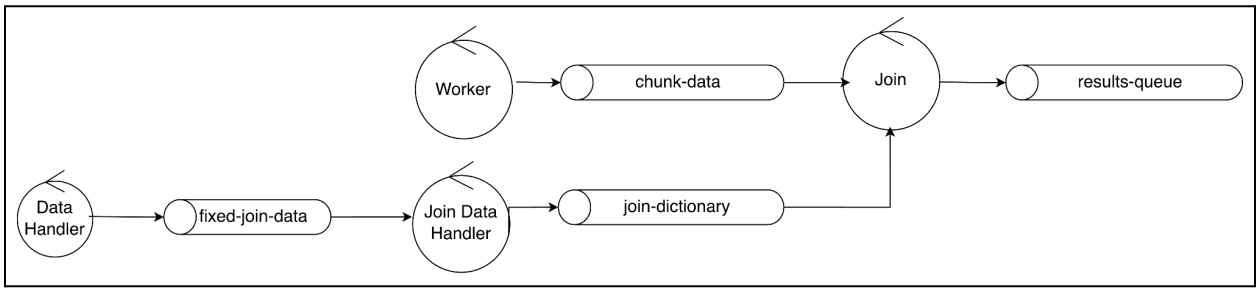
El Group By Orchestrator logra esto contando la cantidad de chunks procesados y esperando a recibir el chunk con isLastChunk = True. Cuando esto ocurre, compara el ChunkNumber de dicho chunk con la cantidad de chunks procesados. Si este número coincide, luego hemos finalizado, si no, se queda esperando a más chunks para este cliente hasta que la cantidad de chunks procesados sea igual al ChunkNumber del último chunk

Cuando el Orchestrator determina que hemos completado el procesamiento para un cliente, el agregator se encarga de leer los archivos que están almacenados en disco para ese cliente, generar un chunk, enviar la data al siguiente paso y eliminar los archivos que habían sido almacenados en disco. Las queries 3 y 4 realizan el mismo proceso que se menciona anteriormente.

Joineado

Como se menciona en el inicio de consulta, all comenzar el pipeline, los datos podrán ser redireccionados a dos posibles flujos, uno el del filtrado y el otro el envío hacia el **Join Data Handler**.

El controller de **Join Data Handler** se encarga de redireccionar los chunks que se utilizaran como data **fija/referencia** a la hora de joinear.



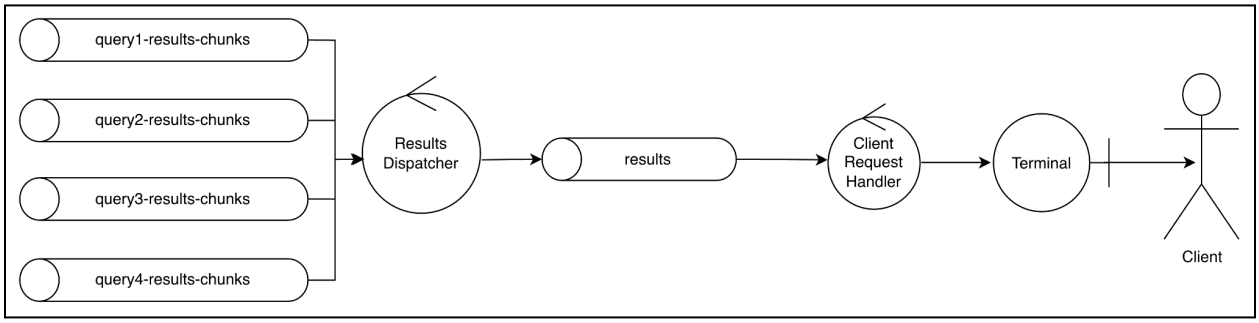
Existen 3 posibles flujos:

- Query 2: Se encarga de hacer un fanout a todos los Join by ItemId quienes recibirán un único chunk y guardarán sus datos en memoria. Esta redundancia permite que cualquier controller pueda resolver un join que necesite información de producto. Esta redundancia se permite ya que el diccionario de productos contiene únicamente 10 productos.
  La información de dichos productos pesa alrededor de 100 bytes. Al ser solo 10 productos por cliente, cada diccionario pesa alrededor de 1000 bytes, equivalente a un kB.
- Query 3: Se encarga de hacer un fanout a todos los Join by StoreId quienes recibieron un único chunk y guardar los datos en memoria. El tamaño y justificación es el mismo que para el Query 2.
- Query 4: Para esta query la cantidad de usuarios es mucho mayor a una dimensión que pueda ser manejada en memoria para múltiples clientes, con lo cual se decidió bajar los archivos directamente a disco. Para poder coordinar mejor y ser más eficiente a la hora de escribir, se decidió agregar múltiples nodos de escritura que permiten paralelizar el trabajo. Para ello se utilizaron las siguientes técnicas:
  - Habran multiples writers (configurable en el docker-compose).
  - Cada writer tendra una cola dedicada.
  - El Join user partition será el responsable de conformar nuevos chunks y routearlos adecuadamente
    - La técnica de enrutamiento utiliza el módulo del user id, ya que el mismo tiene un valor numérico.

- Los writers escribirán en disco múltiples particiones, que usaran el módulo del user id y seguirán la siguiente nomenclatura: {clientid}-{num partition}-users
- Dentro de esos archivos se encontraran todos los usuarios cuyo módulo de id pertenezca a la partición.
- Esta logica simplifica la lectura a la hora del join, ya que la misma se aplica al Join by userID, permitiendo identificar en qué archivo se encuentra dicho usuario.

Retorno de Resultados

Por último, el **Results Dispatcher** se encarga de leer los chunks recibidos, formatearlos y enviarlos al Client Request Handler para que los transmita al cliente.



Cuando el **Client Request Handler** recibe 4 señales de finalización provenientes del Results Dispatcher viniendo de los resultados del mismo cliente, sabe que terminó de resolver las 4 queries y finaliza la conexión.

Este flujo de trabajo se ilustra en el **Diagrama de Actividades** que se encuentra a continuación. El proceso inicia con el cliente, pasa por el Client Request Handler, se interpreta y transforma la data para que sea utilizada en el Data Handler *(se transforma de Batch a Chunk)*, enviada al los Processing Workers y por último retornada utilizando el **Results Dispatcher.**

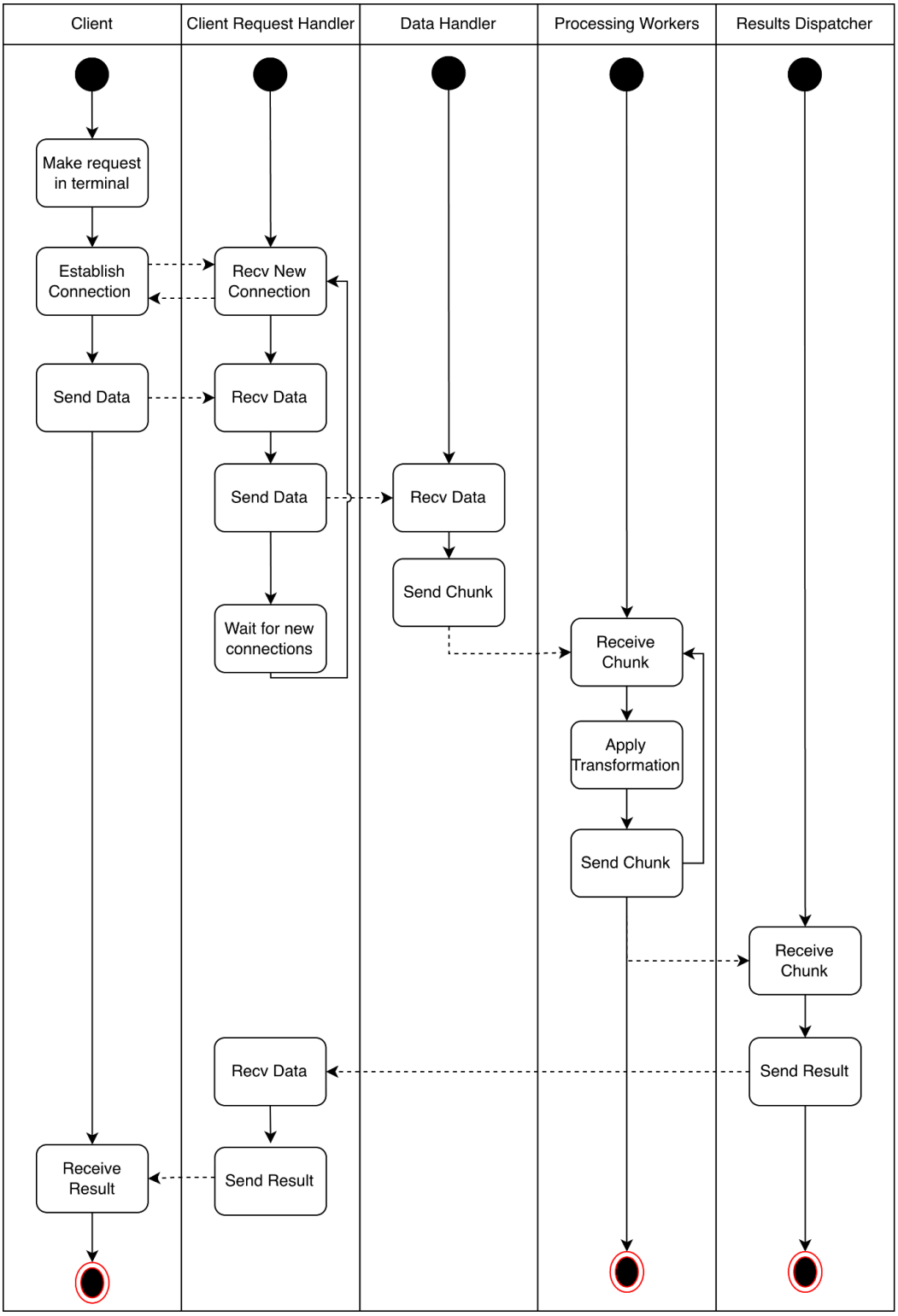

Client
Client Request Handler
Data Handler
Processing Workers
Results Dispatcher
Make request in terminal
Establish Connection
Recv New Connection
Send Data
Recv Data
Send Data
Recv Data
Send Chunk
Wait for new connections
Receive Chunk
Apply Transformation
Send Chunk
Receive Chunk
Recv Data
Send Result
Receive Result
Send Result

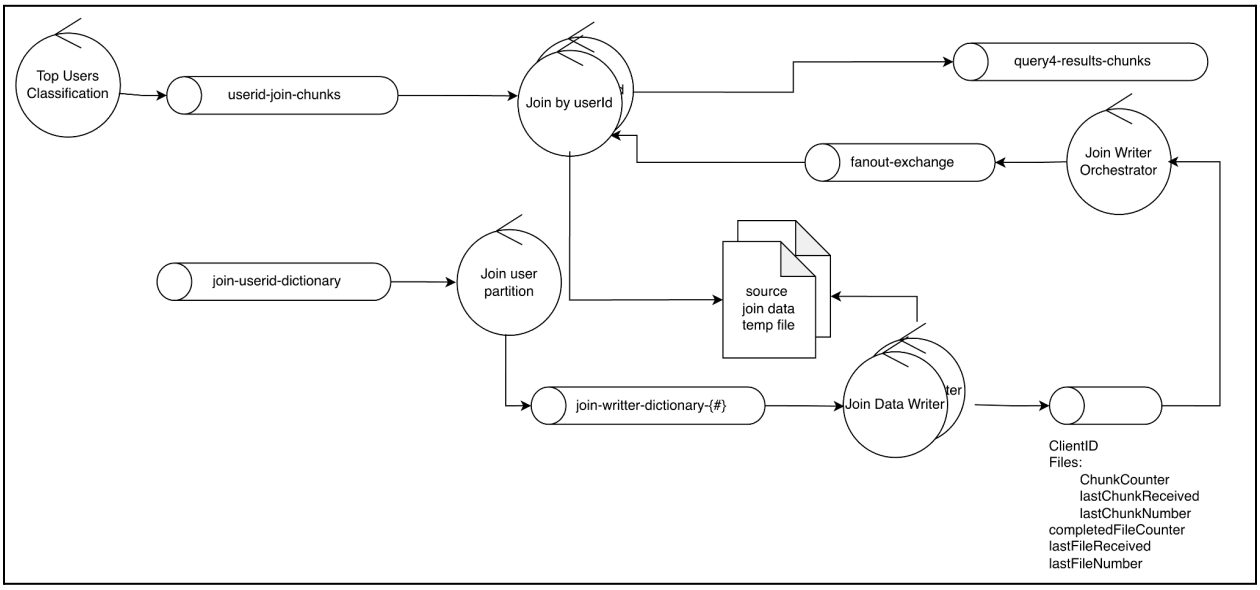
Figura 16

Guardado de Data en Disco

Joiners

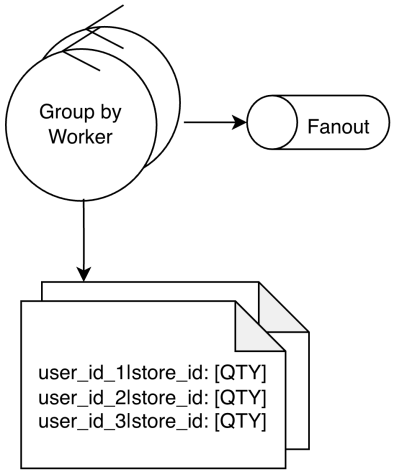
Los múltiples workers de Join by userID están consumiendo desde una única cola chunks de usuarios que ya fueron procesados por el worker de clasificación. Los paquetes comenzarán a ser NACKeados hasta que el **Join Data Writer** haya escrito todos los archvios.
Una vez **todos los archivos de usuarios fueron escritos en disco** el Join Writer Orchestrator comunicará a todos los Join by userID workers el cliente que terminó de escribir todos los archivos de usuarios, y únicamente los workers que tengan data para dicho cliente procesaran lo que les corresponda y enviaran los resultados.

Una vez los resultados fueron enviados, el Join by userID eliminará todos los archivos correspondientes al cliente procesado.



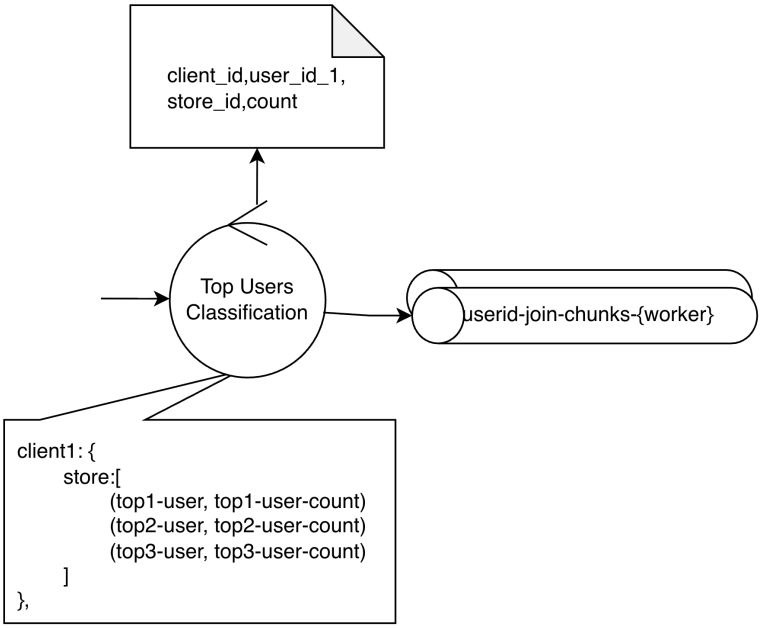
Groupers

Los workers de Group By almacenan datos agregados en archivos CSV particionados. Cada worker escribe en su directorio dedicado /app/groupby-data/q{queryType}/worker-{id}/, creando un archivo por partición con el formato {clientID}-q{queryType}-partition-{XXX}.csv. Los datos se escriben de forma incremental mediante append, donde cada chunk agrega registros a las particiones correspondientes. El sistema utiliza PartitionManager para garantizar escrituras tolerantes a fallos, detectando y corrigiendo automáticamente líneas incompletas. En el primer chunk tras un reinicio, se comparan las últimas líneas escritas para evitar duplicados antes de continuar con la escritura normal.



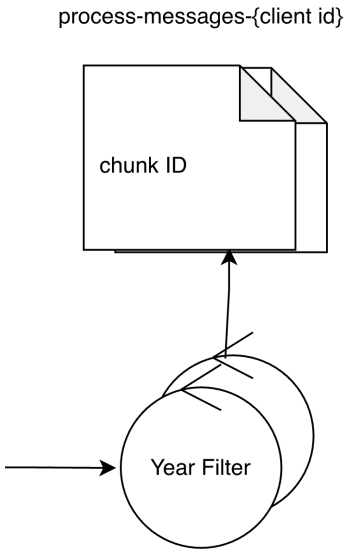
Workers con Estado

Los workers con estado, como ejemplo **Top Users Classification Worker**, mantienen estado por cliente para calcular el y poder recrear a posteriori el estado actual. Persiste metadata en archivos CSV ubicados en /app/worker-data/metadata/, donde cada cliente tiene su propio archivo {clientID}.csv con unicamente la metadata necesaria para replicar el estado posterior, como ejemplo, *msg_id,chunk_number,user_id,store_id,purchase_count*. Al procesar cada chunk, el worker actualiza el estado en memoria y escribe todas las filas del chunk en batch al CSV correspondiente. En caso de caida,, el worker lee estos archivos línea por línea, reconstruye el estado y marca qué chunks ya fueron recibidos. Una vez que un cliente recibe todos sus chunks esperados, el worker genera los resultados finales y elimina automáticamente su archivo de metadata.



Deduplicado de Mensajes

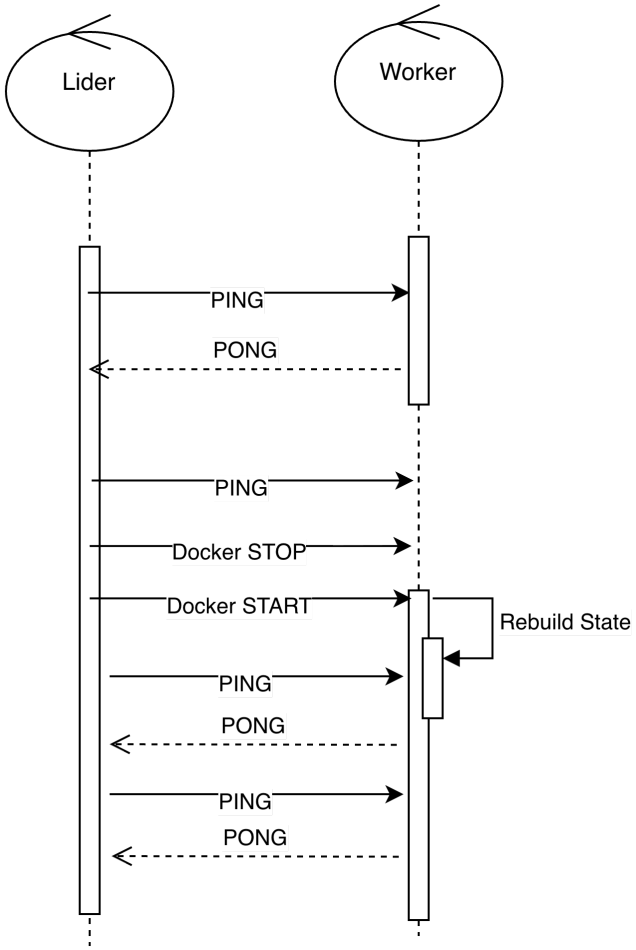
El sistema de deduplicación utiliza archivos de texto plano para mantener registros persistentes de mensajes procesados. Cada worker que requiere deduplicación mantiene archivos en /app/worker-data/ (o subdirectorios específicos) con el formato processed-ids-{clientID}.txt, donde cada línea contiene un ID de mensaje procesado. Al marcar un mensaje como procesado, se escribe el ID al final del archivo correspondiente. Esta estrategia permite verificar rápidamente si un mensaje ya fue procesado, evitando reprocesamiento tras reinicios o reenvíos.



Recuperación de Nodos Caídos

El sistema implementa recuperación automática mediante supervisores distribuidos que monitorean la salud de los workers y reinician containers fallidos. El proceso funciona en dos niveles: recuperación de workers de aplicación y recuperación de supervisores.

Recuperación de Workers de Aplicación: El supervisor líder monitorea continuamente la salud de todos los workers configurados mediante health checks TCP periódicos. Cada worker expone un servidor de salud en el puerto 8888 que responde "PONG" a mensajes "PING". El HealthMonitor realiza estos checks en intervalos configurables (típicamente cada 5 segundos) y marca un worker como fallido tras 2 fallos consecutivos. Cuando detecta un fallo, el supervisor líder ejecuta automáticamente el reinicio del container mediante DockerManager, que realiza un stop seguido de un start del container. Al reiniciar, los workers recuperan su estado desde los archivos CSV persistidos y continúan procesando desde donde quedaron, evitando duplicados mediante los mecanismos de detección implementados.



Recuperación de Supervisores: El sistema mantiene múltiples supervisores (típicamente 3) para alta disponibilidad. Solo uno actúa como líder y ejecuta el monitoreo de workers. Si el supervisor líder falla, los supervisores restantes detectan la ausencia de heartbeats y ejecutan una elección de líder para seleccionar un nuevo líder que continúe con el monitoreo. Esta arquitectura garantiza que el sistema pueda recuperarse incluso si falla el supervisor activo.

Algoritmo Bully para elección de líder

El sistema utiliza el algoritmo Bully para la elección de líder entre los supervisores. Cada supervisor tiene un ID numérico único, y el supervisor con el ID más alto gana la elección.

Inicio de Elección: Una elección se inicia cuando un supervisor detecta que no hay líder (al iniciar el sistema) o cuando el heartbeat del líder actual excede un timeout de 15 segundos.

Proceso de Elección: Cuando un supervisor inicia una elección, envía mensajes ELECTION a todos los supervisores con ID mayor que el suyo. Si recibe un mensaje OK de algún supervisor con ID mayor, sabe que no puede ser líder y espera el anuncio del nuevo líder. Si no recibe ningún OK después de 3 segundos, se declara líder. Al convertirse en líder, el supervisor anuncia su liderazgo mediante mensajes LEADER a todos los demás supervisores y comienza a ejecutar el monitoreo de workers. Los supervisores no líderes detienen su monitoreo y solo escuchan los heartbeats del líder.

Mantenimiento de Liderazgo: El líder envía heartbeats periódicos (cada 5 segundos) mediante mensajes LEADER a todos los supervisores. Los supervisores no líderes actualizan su timestamp del último heartbeat recibido. Si un supervisor no recibe heartbeats del líder por más de 15 segundos, asume que el líder falló e inicia una nueva elección. Este mecanismo garantiza que el sistema siempre tenga un líder activo y pueda recuperarse automáticamente de fallos del supervisor líder.