

Índice

1. Introducción	5
2. Definición Técnica del Sistema Distribuido	6
2.1. Operaciones Distribuidas	6
3. Implementación del Usuario	7
4. Mecanismo de Optimización del Uso de Recursos	7
5. Uso de Archivos Intermedios	7
6. Descripción de las Consultas	8
6.1. Consulta 1: Transacciones filtradas	8
6.2. Consulta 2: Productos más vendidos y más rentables	8
6.3. Consulta 3: TPV por semestre y sucursal	9
6.4. Consulta 4: Clientes más frecuentes y cumpleaños	9
7. Vista de Escenarios	10
7.1. Casos de uso	10
8. Vista de Procesos	11
8.1. Diagramas de Actividades	11
8.1.1. Diagrama de Actividades de la primera consulta	11
8.1.2. Diagrama de Actividades de la segunda consulta	12
8.1.3. Diagrama de Actividades de la tercera consulta	13
8.1.4. Diagrama de Actividades de la cuarta consulta	14
8.2. Diagramas de Secuencia	15
8.2.1. Diagrama de Secuencia de la primera consulta	15
8.2.2. Diagrama de Secuencia de la segunda consulta	16
8.2.3. Diagrama de Secuencia de la tercera consulta	16
8.2.4. Diagrama de Secuencia de la cuarta consulta	17
9. Vista Física	18
9.1. Diagrama de Robustez	18
9.2. Diagrama de Robustez - Primera Consulta	19
9.3. Diagrama de Robustez - Segunda Consulta	19
9.4. Diagrama de Robustez - Tercera Consulta	20
9.5. Diagrama de Robustez - Cuarta Consulta	20
9.6. Diagrama de Despliegue	21
10. Vista de Desarrollo	22
10.1. Diagrama de Paquetes	22

11. Vista Lógica	23
11.1. Diagrama de Clases	23
11.1.1. Diagrama de Clases - Cleaner	23
11.1.2. Diagrama de Clases - Filter	23
11.1.3. Diagrama de Clases - Map	24
11.1.4. Diagrama de Clases - Reduce: Count y Sum	24
11.1.5. Diagrama de Clases - Sort	24
11.1.6. Diagrama de Clases - Join	25
11.1.7. Diagrama de Clases - OutputBuilder	25
12. Diagrama de Grafo Acíclico Dirigido (DAG) - Completo	26
12.1. Diagrama de Grafo Acíclico Dirigido (DAG) - Completo	26
12.2. Diagrama de Grafo Acíclico Dirigido (DAG) - Primera consulta	27
12.3. Diagrama de Grafo Acíclico Dirigido (DAG) - Segunda consulta	28
12.4. Diagrama de Grafo Acíclico Dirigido (DAG) - Tercera consulta	29
12.5. Diagrama de Grafo Acíclico Dirigido (DAG) - Cuarta consulta	30
13. Middleware	31
14. Mensajes y Protocolos	35
14.1. Protocolo	35
14.2. Mensajes	35
14.3. Formato de los mensajes	35
14.4. Tipos de mensajes y su propósito	36
14.5. Codificación (<i>encode</i>)	37
14.6. Decodificación (<i>decode</i>)	37
14.7. Ciclos de vida típicos de los mensajes	38
14.8. Ejemplos de mensajes codificados	39
14.9. Validaciones, errores y robustez	39
14.10. Supuestos y limitaciones deliberadas	39
14.11. Extensibilidad	40
14.12. Resumen	40
15. Mediciones de rendimiento del sistema implementado	41
15.1. Configuración del entorno de pruebas	41
15.2. Escenarios de prueba	41
15.3. Tiempos de procesamiento obtenidos	41
16. Mecanismos de Control	42
16.1. Mecanismos de Control de Sincronización	42
16.2. Mecanismos de Control de Señales y Finalización	42
16.3. Mecanismos de Control de Fallas	42
17. Tests	43

17.1. Tests de Middleware	43
17.2. Tests de Comparación de Outputs	43
17.3. Tests de Propagación de EOF	44
18.Cronograma teórico del desarrollo	45
18.1. Diseño	45
18.2. Escalabilidad	45
18.3. Multi-Client	45
18.4. Tolerancia	45
18.5. Paper	45
19.Cronograma real del desarrollo	46
19.1. Diseño (2 semanas)	46
19.2. Middleware y Coordinación de Procesos (2.5 semanas)	47
19.3. Multi Client (2 semanas)	48

1. Introducción

El presente documento describe el diseño de un sistema distribuido para el análisis de datos de una cadena de cafeterías en Malasia. El objetivo principal es procesar grandes volúmenes de información transaccional, de clientes, de sucursales y de productos, para obtener métricas clave que apoyen la toma de decisiones de negocio.

De acuerdo con los requisitos existentes, el sistema debe permitir responder las siguientes consultas:

1. Listado de transacciones (ID y monto) realizadas entre los años 2024 y 2025, en el rango horario de 06:00 a 23:00, con un monto total mayor o igual a 75.
2. Identificación de los productos más vendidos (nombre y cantidad) y los que más ganancias han generado (nombre y monto) para cada mes de 2024 y 2025.
3. Cálculo del *Total Payment Value* (TPV) por cada semestre en 2024 y 2025, discriminado por sucursal, considerando sólo transacciones entre 06:00 y 23:00.
4. Obtención de la fecha de cumpleaños de los tres clientes con mayor cantidad de compras en 2024 y 2025, para cada sucursal.

Además de los requerimientos funcionales, el sistema deberá cumplir con los siguientes requisitos no funcionales:

- Optimización para entornos multicomputadoras, asegurando la escalabilidad ante el crecimiento de datos.
- Inclusión de un *middleware* que abstraiga la comunicación entre nodos mediante grupos.
- Ejecución única del procesamiento con capacidad de *graceful quit* frente a señales de terminación (SIGTERM).

El diseño se realizará bajo un enfoque de arquitectura distribuida, buscando flexibilidad, robustez y capacidad de escalar en entornos reales de procesamiento de datos.

2. Definición Técnica del Sistema Distribuido

El sistema distribuido propuesto permite al usuario interactuar mediante una consola, desde la cual los usuarios envían las instrucciones y datasets necesarios para el procesamiento, y a cambio reciben como salida de la misma el resultado de las consultas realizadas. La infraestructura se implementará sobre contenedores Docker, los cuales emulan múltiples nodos computacionales. Cada nodo se especializa en la ejecución de una operación determinada, tales como filtrado, mapeo, reducción, ordenamiento o combinación de datos.

El sistema recibe como entrada un conjunto de archivos con la información a procesar:

- `menu_items.csv`
- `payment_methods.csv`
- `stores.csv`
- Múltiples archivos `transaction_items.csv`
- Múltiples archivos `transactions.csv`
- Múltiples archivos `users.csv`
- `vouchers.csv`

Estos archivos son distribuidos entre los distintos nodos según lo que requiera cada operación. La comunicación entre el nodo coordinador y los nodos de procesamiento se gestiona mediante un *middleware*, el cual se describe en detalle en una sección posterior.

2.1. Operaciones Distribuidas

Las operaciones fundamentales soportadas por el sistema son las siguientes:

1. **Cleaner:** Se encarga de la limpieza de los datos de entrada, eliminando campos que resultan innecesarios para las consultas. Es escalable.
2. **Filter:** Permite seleccionar subconjuntos de datos en función de condiciones específicas. Una misma instancia puede aplicar diferentes filtros según el criterio definido en la consulta (por ejemplo, filtrar por fechas, montos o rangos horarios). Es escalable.
3. **Map:** Transforma los datos de entrada generando nuevas columnas o reformateando la información existente. Ejemplo: Agregar un campo con el mes y año a partir de una fecha, o calcular un contador auxiliar. Es escalable.
4. **Reduce:** Agrupa datos por una clave determinada y aplica una función de agregación (como sumas, conteos o acumulaciones). Existen múltiples variantes, dependiendo de qué métrica se busque consolidar. No es escalable.
5. **Join:** Combina registros de diferentes datasets en función de un campo común, permitiendo enriquecer la información (ejemplo: Unir transacciones con usuarios para obtener la fecha de nacimiento del cliente). Es escalable.
6. **SortBy:** Ordena los registros de acuerdo con uno o más criterios, ya sea de manera ascendente o descendente. Es fundamental para identificar los elementos más significativos en cada consulta (por ejemplo, productos más vendidos). No es escalable.
7. **OutputBuilder:** Es el último paso en la cadena de operaciones, se encarga de formatear la respuesta que va a ser enviada al usuario. Es escalable.

Cada nodo de operación está diseñado para ser genérico y flexible: Puede ejecutar cualquier variante de las funciones mencionadas según el rol específico del nodo, siendo el sistema coordinador quien le envía los datos y la configuración necesaria para que realice la tarea solicitada de la mejor forma posible.

3. Implementación del Usuario

La interacción de los usuarios con el sistema distribuido se realizará mediante un esquema secuencial. Para cada usuario que se conecte, se respetará una secuencia específica de ejecución.

- **Etapa de envío de información:** En esta etapa el usuario envía toda la información en base a la que el sistema debe operar. Esto incluye todos los datasets a procesar para resolver las 4 consultas.
- **Etapa de espera:** El cliente queda a la espera de que el sistema distribuido procese la información y genere las respuestas a las consultas.
- **Etapa de recepción de resultados:** Una vez que el sistema ha procesado la información, el usuario recibe las respuestas a las consultas solicitadas mediante unos archivos generados en la carpeta `'.results/query_results'` en formato `'.txt'`. Se genera un archivo por cada consulta.

De esta manera se logra una interacción segura y escalable.

La arquitectura está diseñada para soportar múltiples usuarios en simultáneo, escalando el número de conexiones establecidas proporcionalmente a la cantidad de clientes activos.

4. Mecanismo de Optimización del Uso de Recursos

Cada computadora que participa como nodo de procesamiento en el sistema distribuido estará dedicada a ejecutar una operación determinada (por ejemplo, filtrado o reducción).

Si bien en una primera versión se teorizó la posibilidad de generar varios procesos por 'CPU' para buscar algún tipo de optimización del rendimiento de cada computadora, luego de la primera entrega con el corrector se definió que no resultaría necesario para la implementación de la resolución.

Por lo tanto, se descarta esta mejora para esta versión del sistema, pero se deja como recomendación por si se realiza una actualización a futuro en búsqueda de optimizar tiempos de procesamiento de consultas.

De ser así, se debe recordar que al implementar la paralelización, se debe hacer mediante **multiprocesos** y no mediante **multithreading**. La justificación de esta elección radica en que Python presenta limitaciones para tareas de cómputo intensivo debido al *Global Interpreter Lock* (GIL).

La librería de **multiprocessing** permite crear procesos independientes que aprovechan mejor las capacidades de CPUs con múltiples núcleos.

5. Uso de Archivos Intermedios

En el diseño del sistema se contempla la generación de archivos intermedios durante el procesamiento de las consultas. Esta decisión se fundamenta en un aspectos clave:

Respaldo de información: Los archivos intermedios actúan como puntos de control que facilitan futuras implementaciones de tolerancia a fallas. En caso de interrupciones, será posible retomar el procesamiento desde un estado parcial previamente almacenado.

El uso de archivos intermedios, si bien introduce un costo adicional de I/O, aumenta la robustez del sistema y habilita mejoras posteriores en términos de recuperación y confiabilidad.

Para esta entrega del sistema todavía no están implementados de manera funcional, pero si se deja esta consideración para luego documentar su funcionamiento cuando corresponda.

6. Descripción de las Consultas

El sistema debe ser capaz de responder a las siguientes consultas, de acuerdo con la lógica planteada en el enunciado. A continuación, se detalla el paso a paso de cada una de ellas, indicando las tablas necesarias, las columnas relevantes y la secuencia de operaciones distribuidas.

6.1. Consulta 1: Transacciones filtradas

Objetivo: Obtener las transacciones (ID y monto) realizadas durante 2024 y 2025, entre las 06:00 y las 23:00 horas, con un monto total mayor o igual a 75.

- **Tablas requeridas:** `transactions`.
- **Columnas utilizadas:** `transaction_id`, `created_at`, `final_amount`.
- **Pasos:**
 1. Filtrar transacciones entre 2024 y 2025, en el rango horario indicado.
 2. Filtrar transacciones con monto mayor o igual a 75.
 3. Generar el reporte y devolverlo al usuario.

6.2. Consulta 2: Productos más vendidos y más rentables

Objetivo: Identificar, para cada mes de 2024 y 2025, los productos más vendidos (nombre y cantidad) y los productos que más ganancias generaron (nombre y monto).

- **Tablas requeridas:** `transaction_items`, `menu_items`.
- **Columnas utilizadas:**
 - De `transaction_items`: `item_id`, `created_at`, `quantity`, `subtotal`.
 - De `menu_items`: `item_id`, `item_name`.
- **Pasos:**
 1. Filtrar los `transaction_items` de 2024 y 2025.
 2. Generar una nueva columna `year_month` a partir de la fecha.
 3. Para productos más vendidos:
 - a) Calcular contador de cantidad (`item_counter`).
 - b) Reducir por clave `year_month`, `item_id` sumando las cantidades.
 - c) Ordenar por cantidad descendente.
 4. Para productos más rentables:
 - a) Calcular monto acumulado (`total_amount_counter`).
 - b) Reducir por clave `year_month`, `item_id` sumando subtotales.
 - c) Ordenar por monto descendente.
 5. Unir con `menu_items` para obtener nombres de los productos.
 6. Generar el reporte y devolverlo al usuario.

6.3. Consulta 3: TPV por semestre y sucursal

Objetivo: Calcular el *Total Payment Value* (TPV) por cada semestre de 2024 y 2025, para cada sucursal, considerando únicamente transacciones realizadas entre 06:00 y 23:00.

- **Tablas requeridas:** `transactions` y `stores`.
- **Columnas utilizadas:** `store_id`, `created_at`, `final_amount`.
- **Pasos:**
 1. Filtrar transacciones entre 2024 y 2025, dentro del rango horario.
 2. Generar nueva columna `year_semester` a partir de la fecha.
 3. Reducir por clave `year_semester`, `store_id` sumando los montos finales.
 4. Unir con `stores` para obtener nombres de las sucursales.
 5. Generar el reporte y devolverlo al usuario.

6.4. Consulta 4: Clientes más frecuentes y cumpleaños

Objetivo: Obtener la fecha de cumpleaños de los tres clientes con mayor número de compras durante 2024 y 2025, discriminado por sucursal.

- **Tablas requeridas:** `transactions`, `users` y `stores`.
- **Columnas utilizadas:**
 - De `transactions`: `user_id`, `store_id`, `created_at`.
 - De `users`: `user_id`, `birthdate`.
- **Pasos:**
 1. Filtrar transacciones de 2024 y 2025.
 2. Generar clave combinada `store_user` con `store_id` y `user_id`.
 3. Calcular contador de compras (`buys_counter`).
 4. Reducir por clave `store_user` sumando los contadores.
 5. Ordenar en forma descendente por número de compras.
 6. Seleccionar los tres usuarios con más compras por sucursal.
 7. Unir con `users` para obtener las fechas de nacimiento.
 8. Unir con `stores` para obtener nombres de las sucursales.
 9. Generar el reporte y devolverlo al usuario.

7. Vista de Escenarios

7.1. Casos de uso

Para modelar la interacción principal entre los actores y el sistema, se presenta un diagrama de casos de uso. Este diagrama identifica al actor principal/client, denominado “Cafetería”, y las cuatro funcionalidades clave que el sistema debe proveer, correspondiendo directamente a las consultas de negocio definidas en los requerimientos.

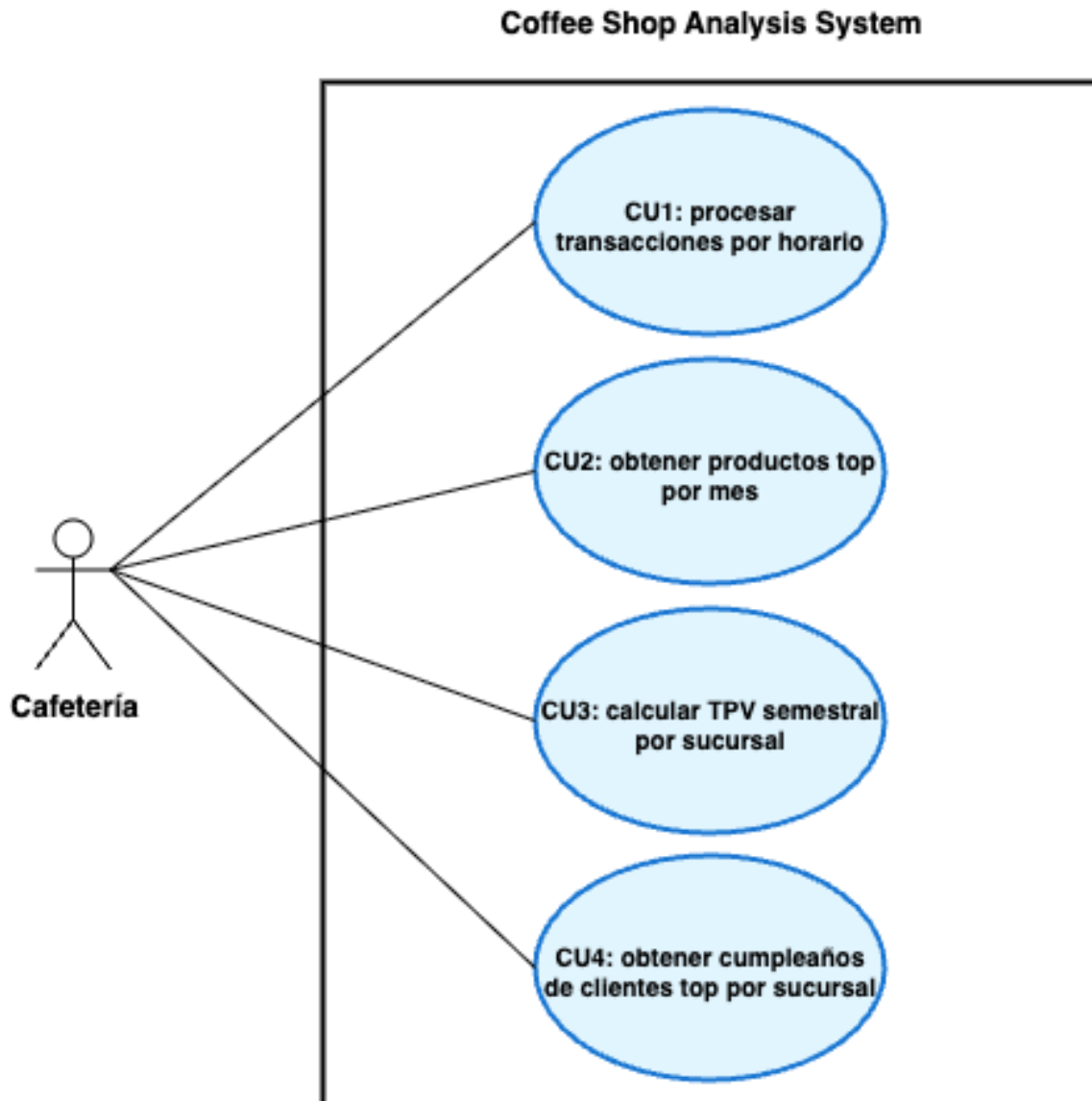


Figura 1: Casos de Uso

8. Vista de Procesos

8.1. Diagramas de Actividades

Los diagramas de actividad describen el flujo de trabajo lógico para cada una de las consultas funcionales. A continuación, se presenta un diagrama para cada consulta, detallando la secuencia de acciones y las decisiones desde el inicio de la solicitud hasta la presentación de los resultados finales.

8.1.1. Diagrama de Actividades de la primera consulta

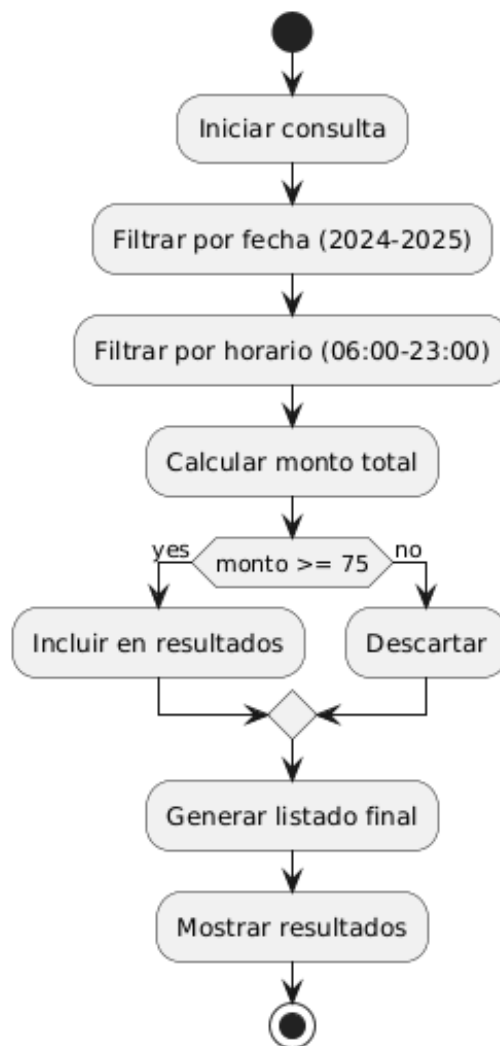


Figura 2: Actividad de la primera consulta

8.1.2. Diagrama de Actividades de la segunda consulta

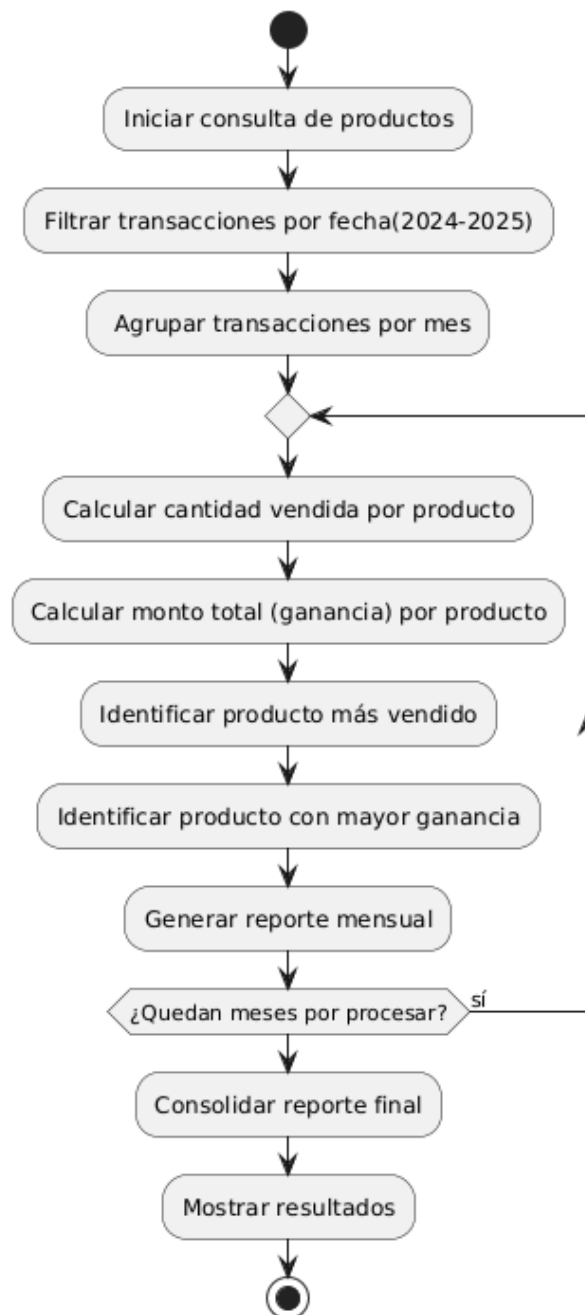


Figura 3: Actividad de la segunda consulta

8.1.3. Diagrama de Actividades de la tercera consulta

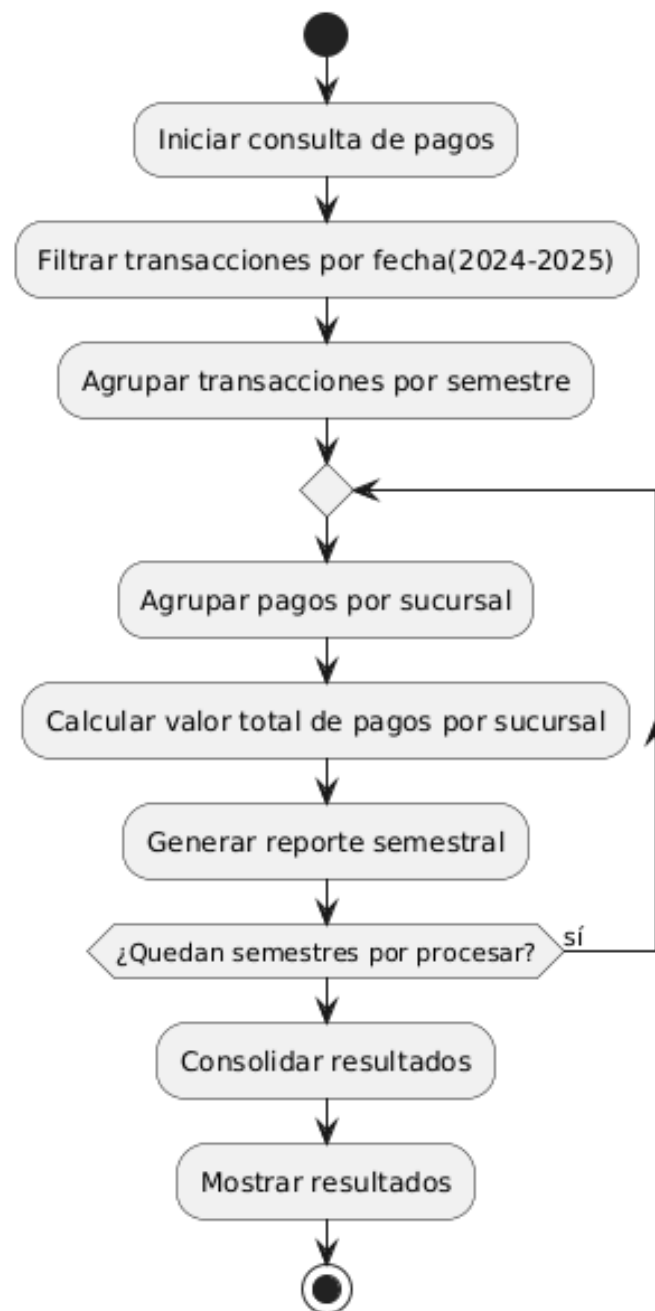


Figura 4: Actividad de la tercera consulta

8.1.4. Diagrama de Actividades de la cuarta consulta

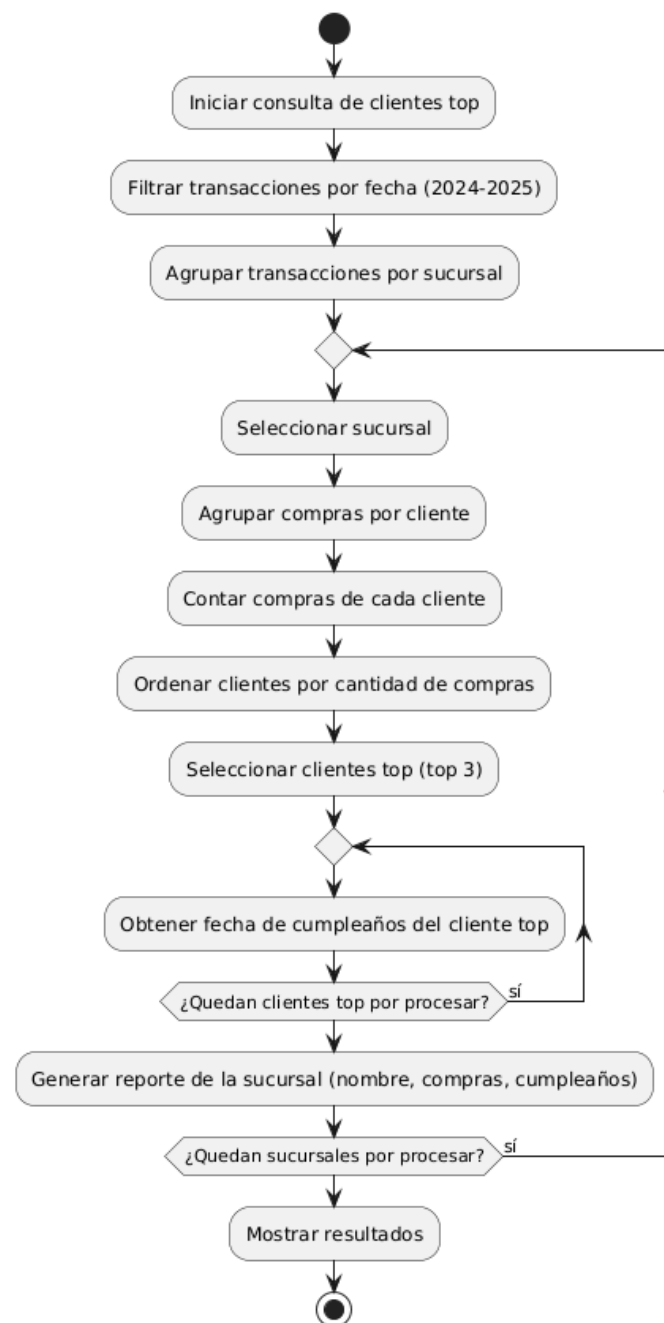


Figura 5: Actividad de la cuarta consulta

8.2. Diagramas de Secuencia

Para ilustrar la interacción temporal y el intercambio de mensajes entre los distintos componentes del sistema, se utilizan los diagramas de secuencia. Cada diagrama modela la ejecución de una consulta, mostrando cómo el Client Process envía lotes de datos al Server, el cual coordina las operaciones distribuidas entre los nodos especializados como Filter, GroupBy y ReduceBy hasta obtener y devolver el reporte final.

8.2.1. Diagrama de Secuencia de la primera consulta

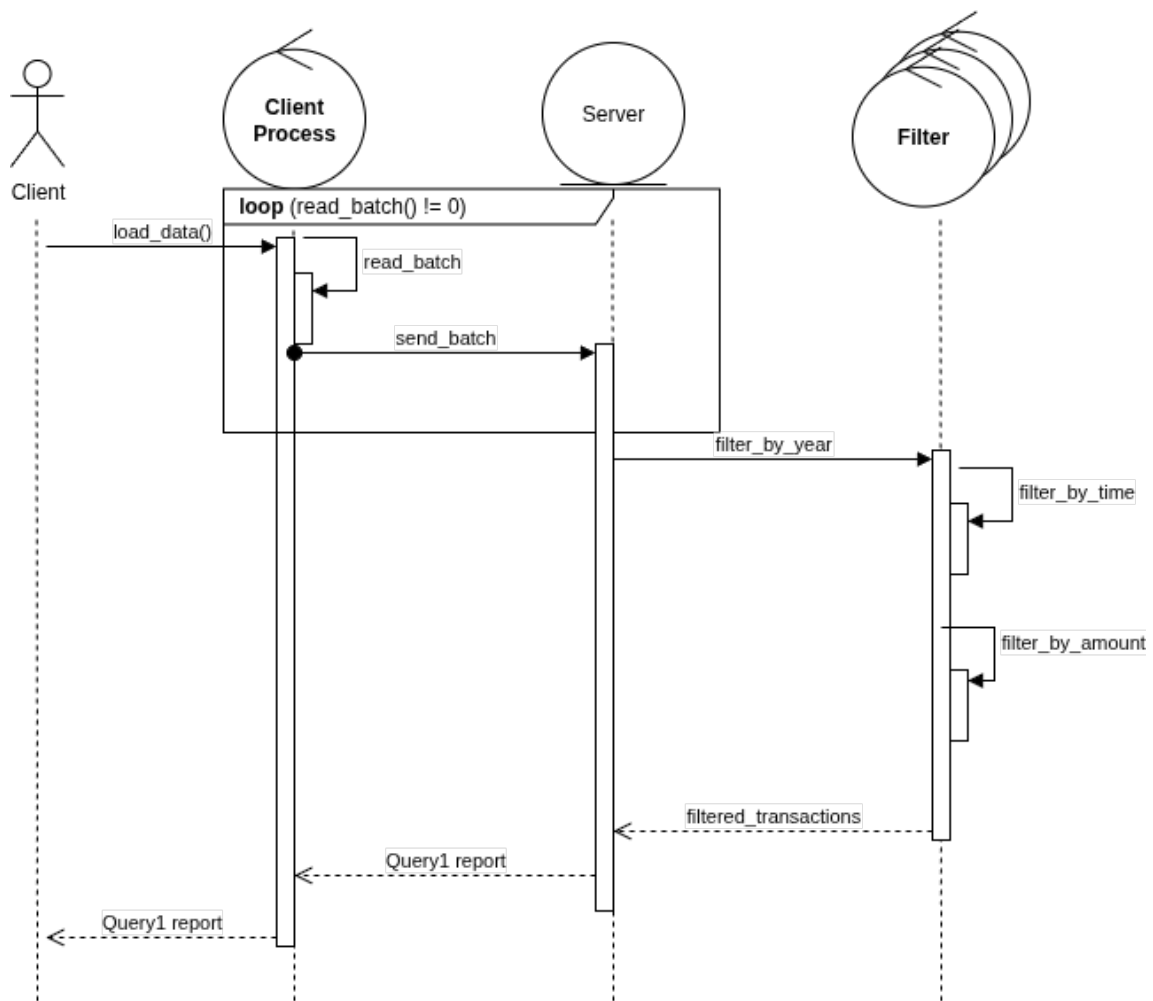


Figura 6: Secuencia de la primera consulta

8.2.2. Diagrama de Secuencia de la segunda consulta

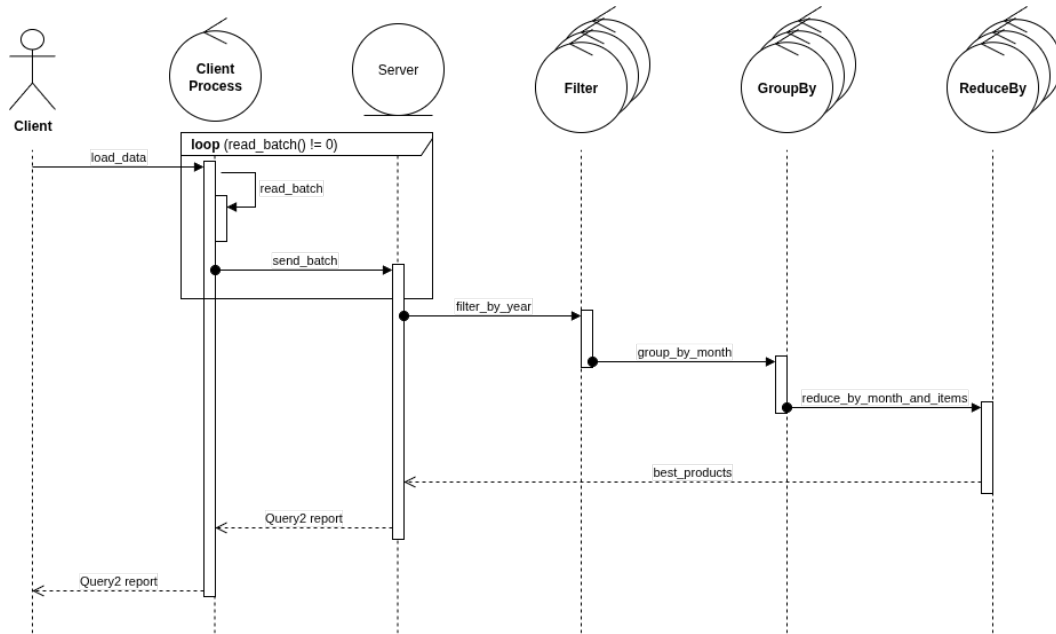


Figura 7: Secuencia de la segunda consulta

8.2.3. Diagrama de Secuencia de la tercera consulta

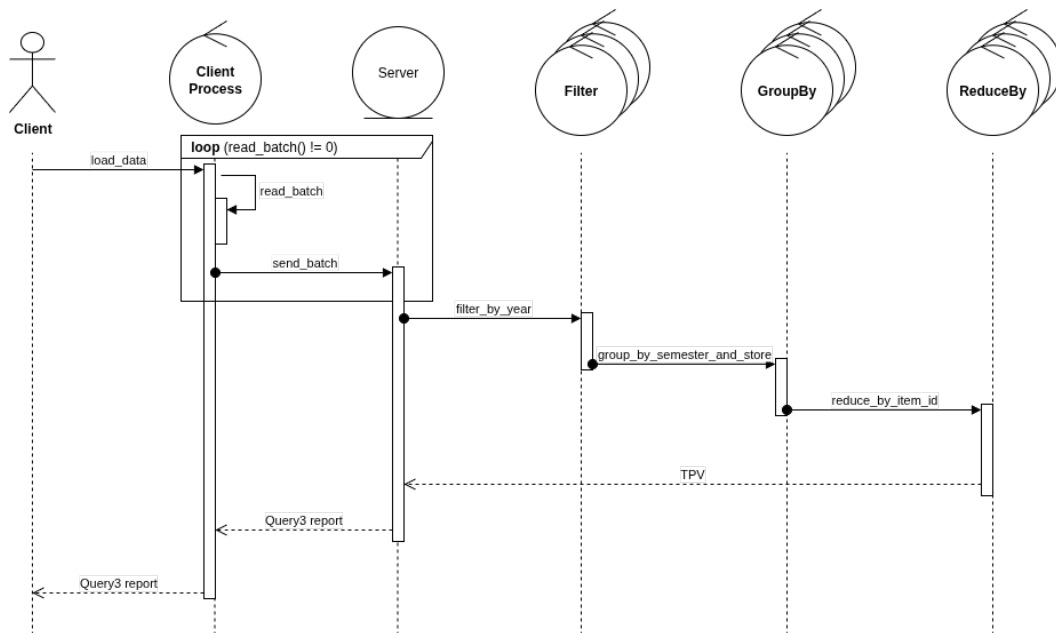


Figura 8: Secuencia de la tercera consulta

8.2.4. Diagrama de Secuencia de la cuarta consulta

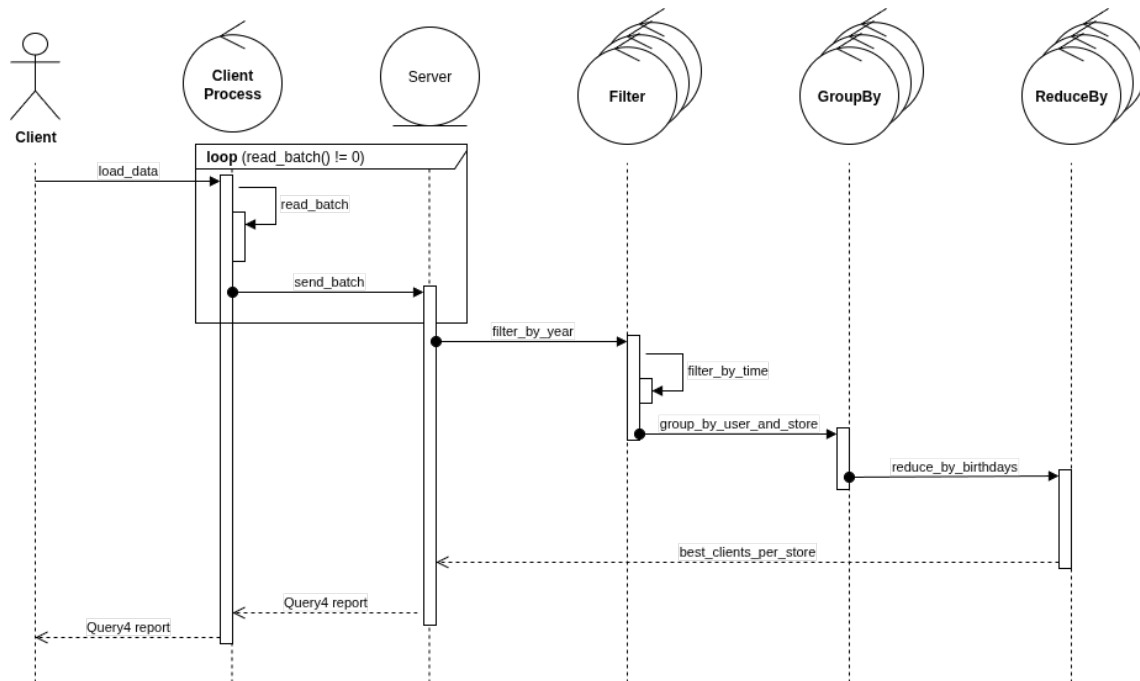


Figura 9: Secuencia de la cuarta consulta

9. Vista Física

9.1. Diagrama de Robustez

El diagrama de robustez forma parte de la vista lógica y sirve como puente entre los casos de uso y el diseño detallado. Este diagrama permite identificar y organizar los principales elementos del sistema, diferenciando entre actores externos, límites de la aplicación y las entidades o controladores que gestionan la lógica de negocio.

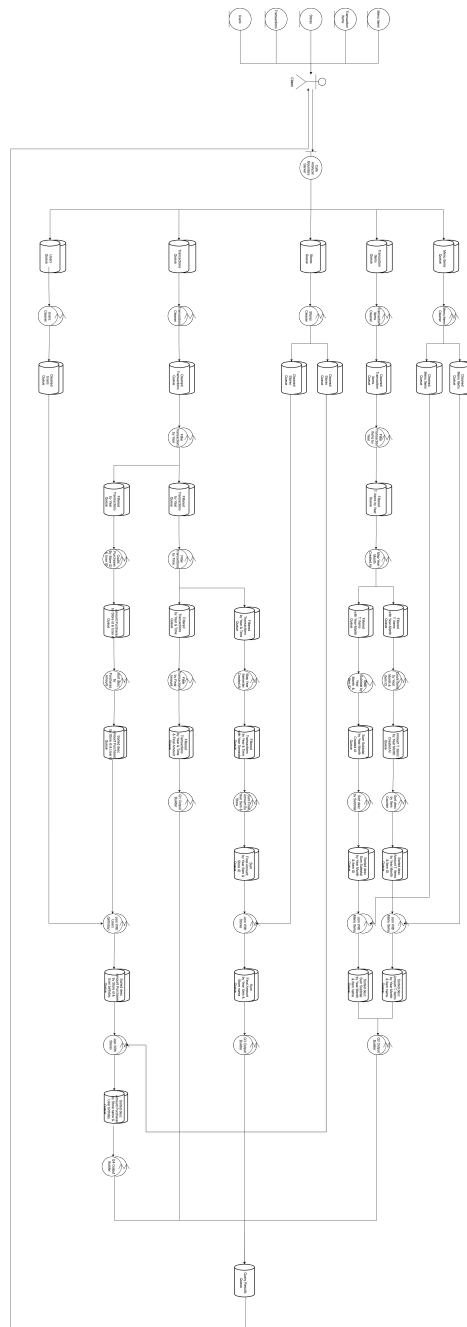
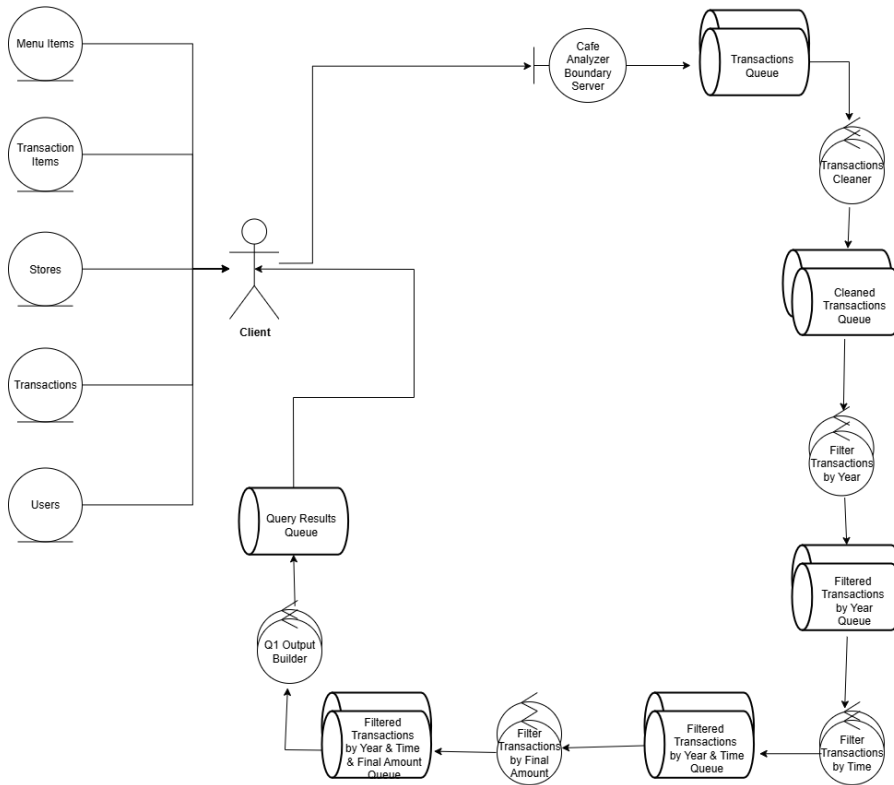
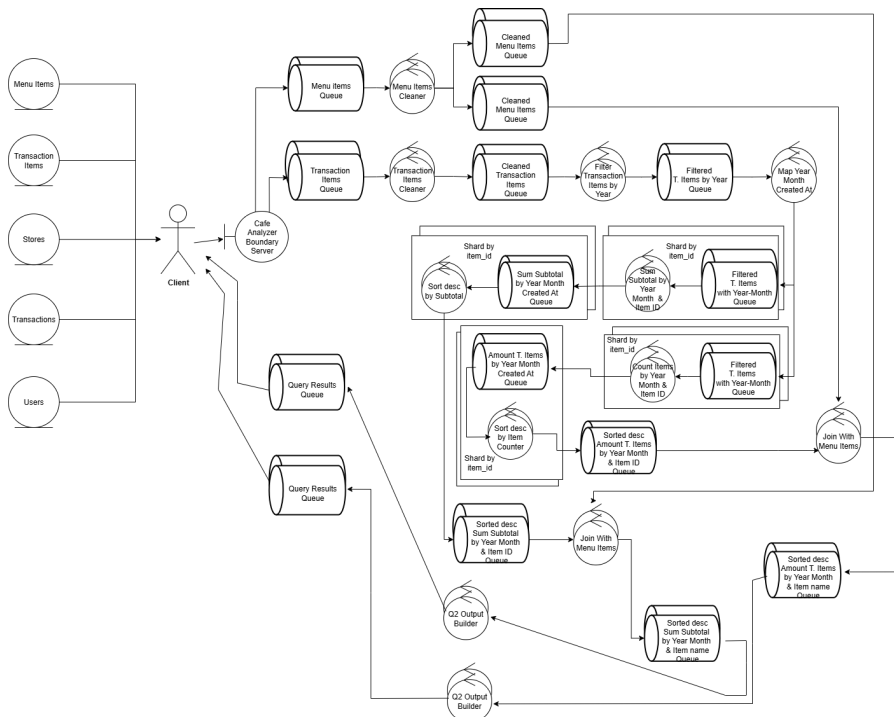


Figura 10: Diagrama de Robustez - Completo

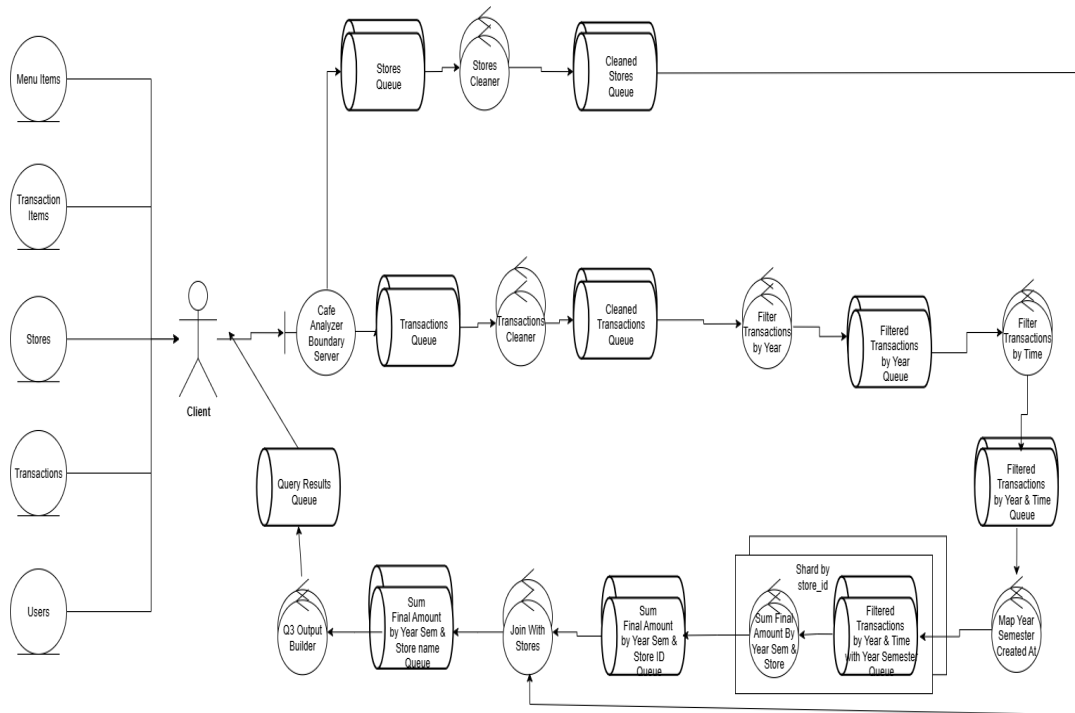
9.2. Diagrama de Robustez - Primera Consulta



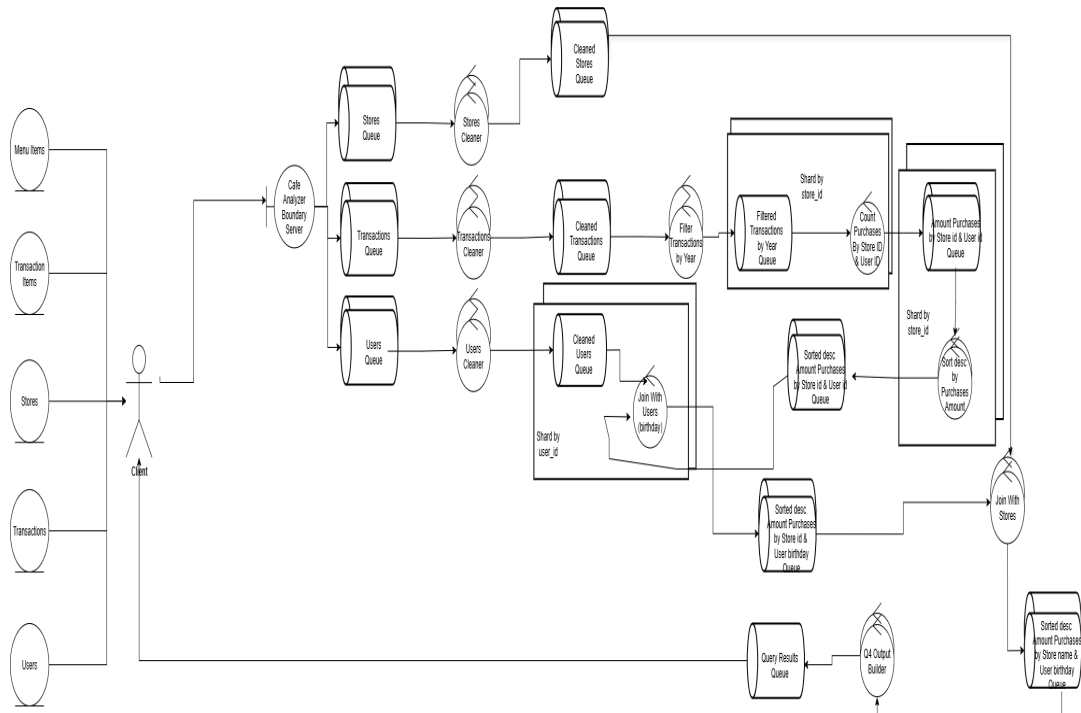
9.3. Diagrama de Robustez - Segunda Consulta



9.4. Diagrama de Robustez - Tercera Consulta



9.5. Diagrama de Robustez - Cuarta Consulta



9.6. Diagrama de Despliegue

Dentro de la vista física realizamos el diagrama de despliegue, el cual ilustra la topología del sistema distribuido en términos de sus nodos computacionales. El diagrama muestra los distintos grupos de nodos especializados (Data Cleaners, Filters, Maps, Joins, etc.) y cómo se interconectan a través de un componente central: el MOM Broker. Esta arquitectura facilita la comunicación asincrónica y el desacoplamiento entre los procesos.

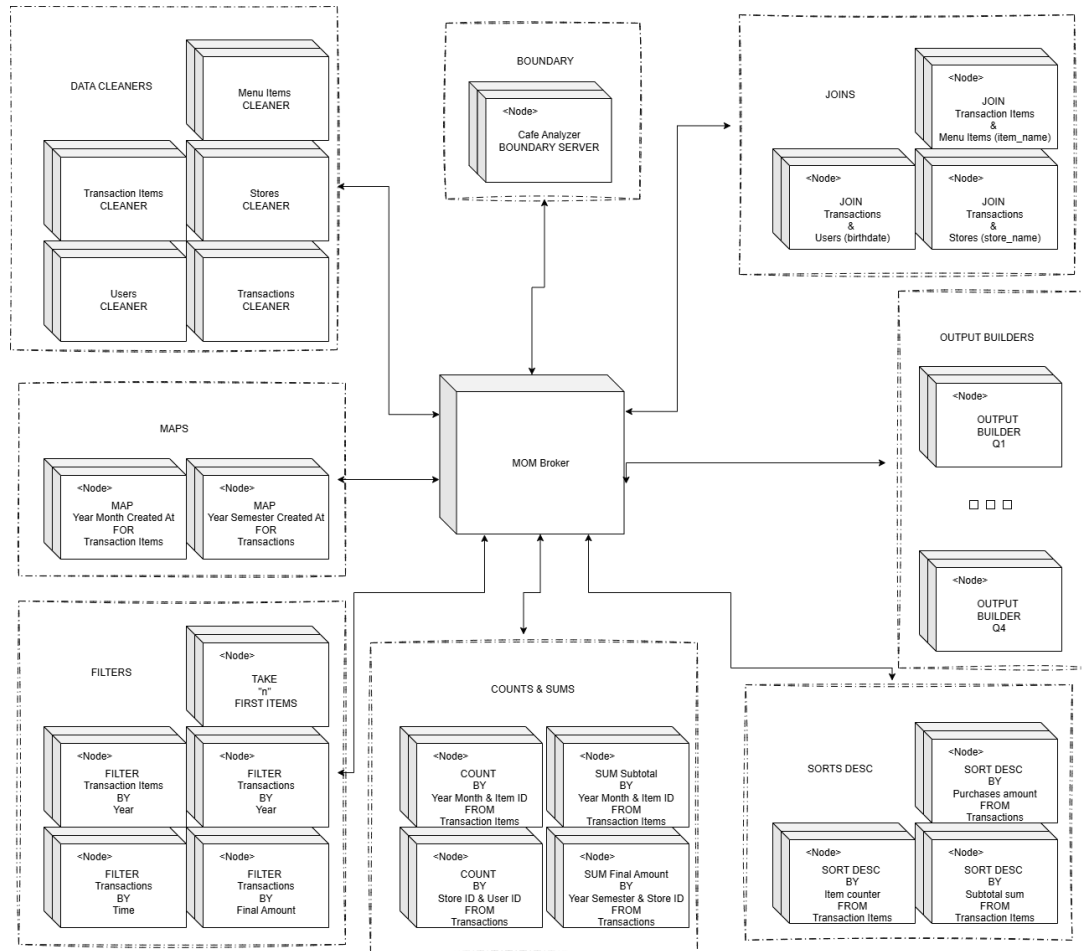


Figura 11: Diagrama de Despliegue

10. Vista de Desarrollo

10.1. Diagrama de Paquetes

La vista de desarrollo organiza el sistema en un conjunto de paquetes lógicos para promover una alta cohesión y un bajo acoplamiento. El diagrama muestra los componentes clave: el paquete `shared` contiene la lógica de comunicación común, como el `MQConnectionHandler`; el paquete `controllers` agrupa las distintas operaciones distribuidas (filtros, mapeos, etc.); y los paquetes `client` y `server` definen los puntos de entrada de la aplicación.

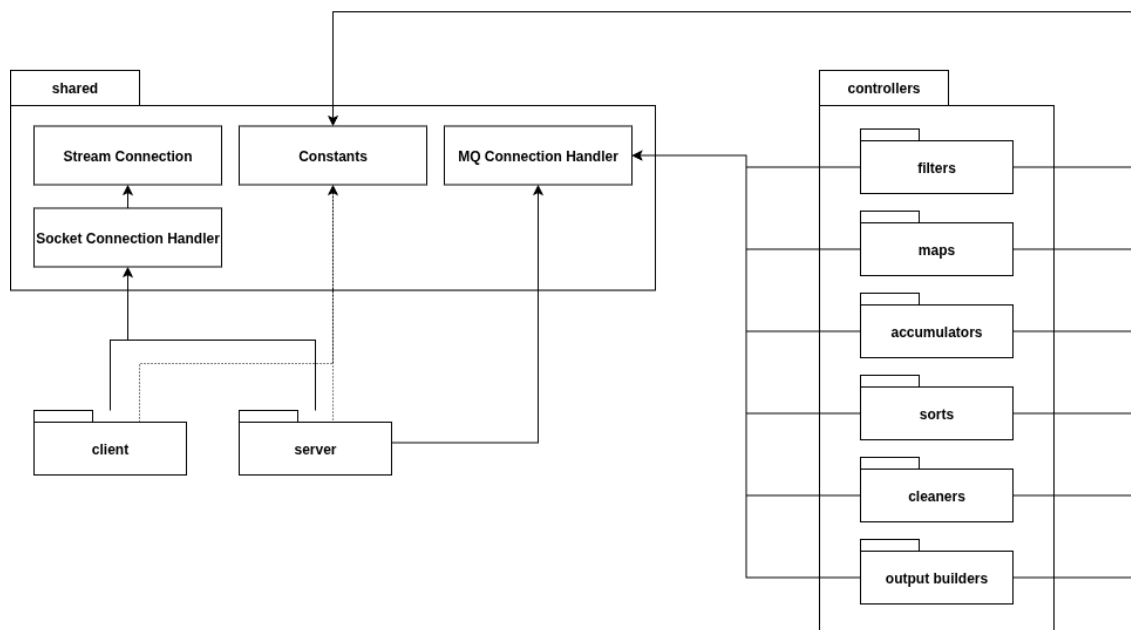


Figura 12: Vista de Desarrollo - Diagrama de Paquetes

11. Vista Lógica

11.1. Diagrama de Clases

La vista lógica detalla la estructura estática del sistema a través de diagramas de clases. El diseño se basa en la herencia y la abstracción para maximizar la reutilización de código. Como se observa en los siguientes diagramas, se define una clase base abstracta Controller de la cual heredan las distintas operaciones especializadas (Filter, Count, Sum, etc.), permitiendo que el sistema maneje diferentes tipos de tareas de manera uniforme y extensible.

11.1.1. Diagrama de Clases - Cleaner

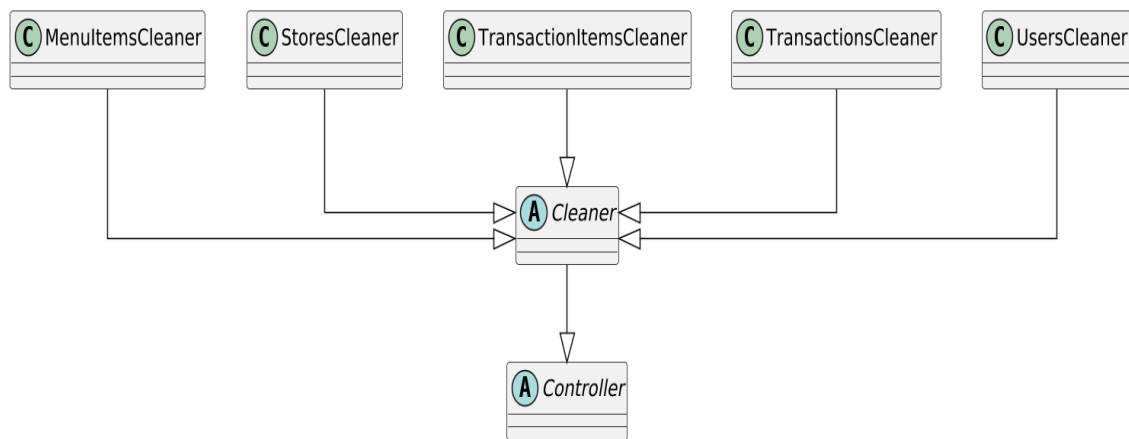


Figura 13: Clase Cleaner

11.1.2. Diagrama de Clases - Filter

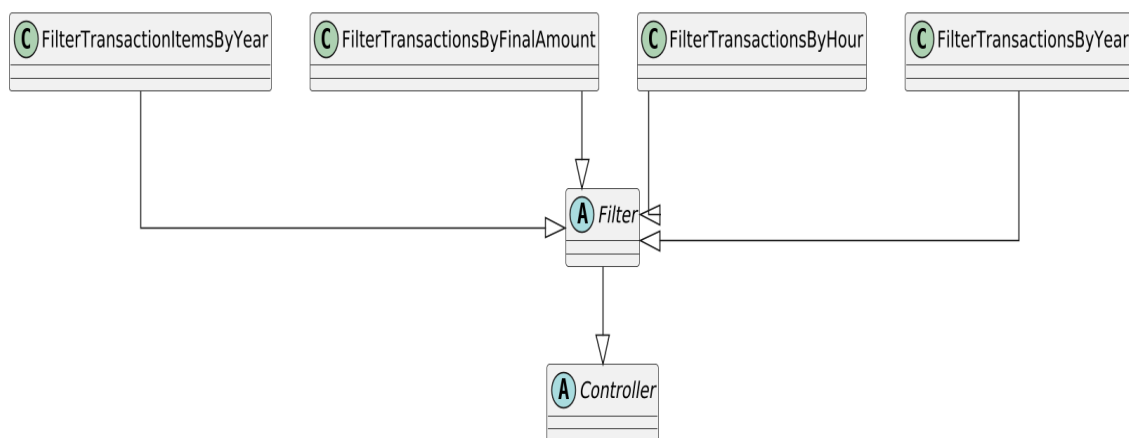


Figura 14: Clase Filter

11.1.3. Diagrama de Clases - Map

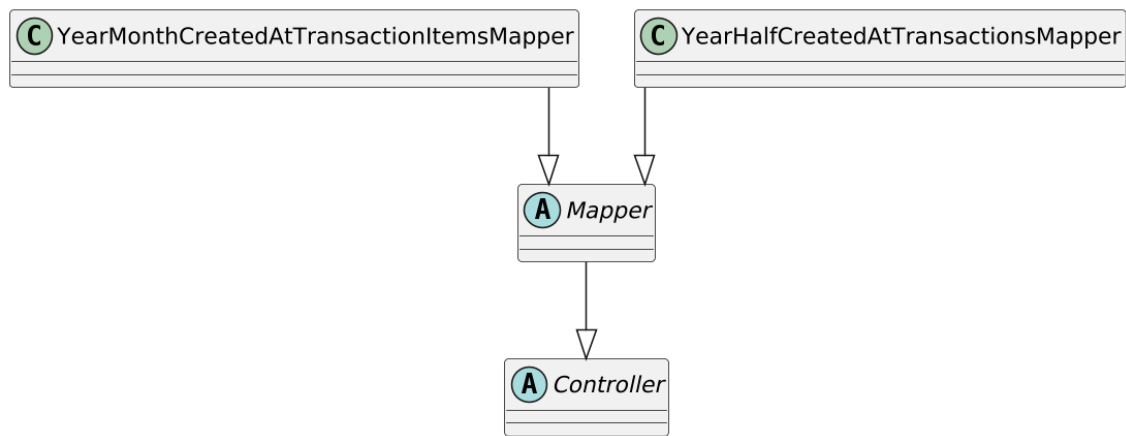


Figura 15: Clase Map

11.1.4. Diagrama de Clases - Reduce: Count y Sum

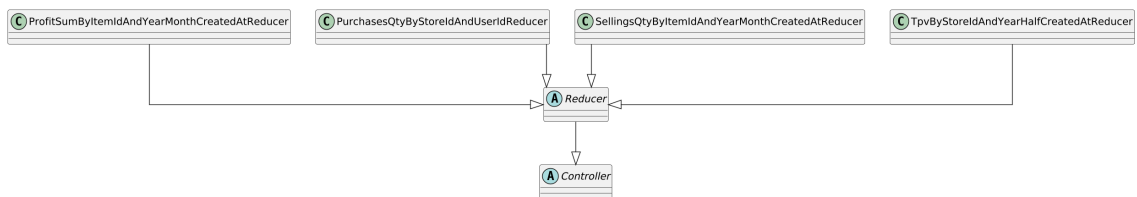


Figura 16: Clases Reduce: Count y Sum

11.1.5. Diagrama de Clases - Sort

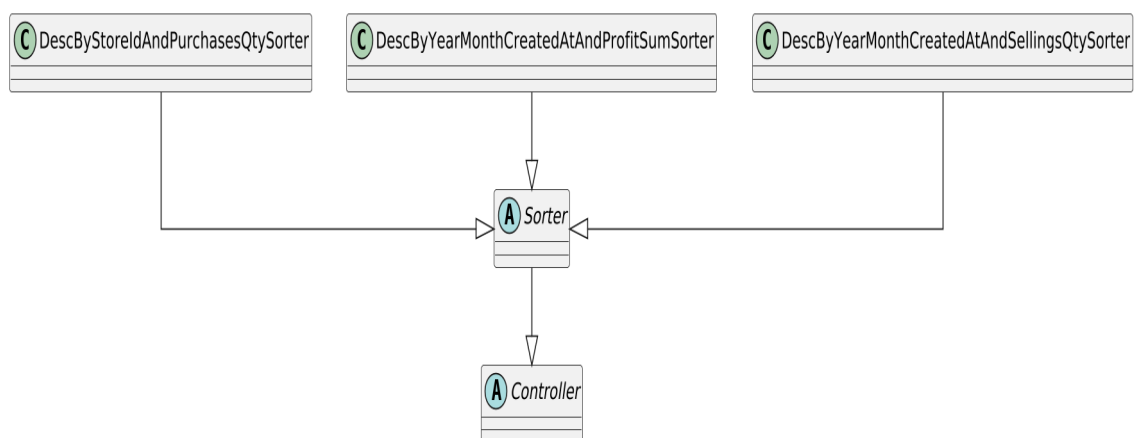


Figura 17: Clase Sort

11.1.6. Diagrama de Clases - Join

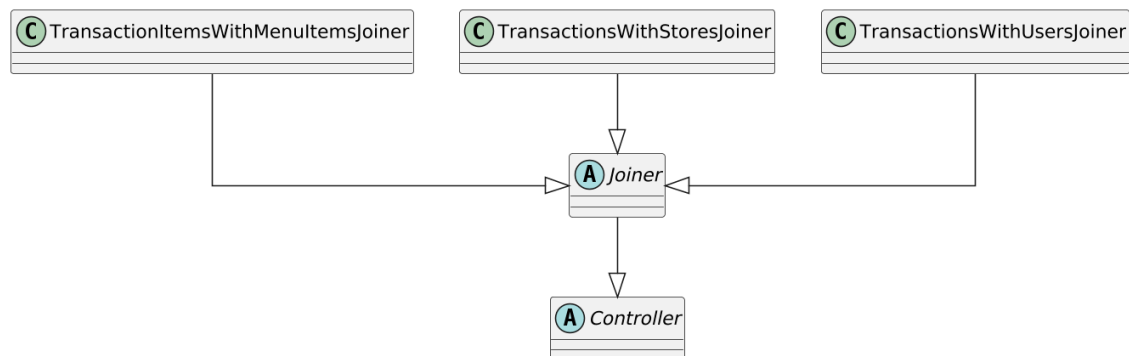


Figura 18: Clase Join

11.1.7. Diagrama de Clases - OutputBuilder

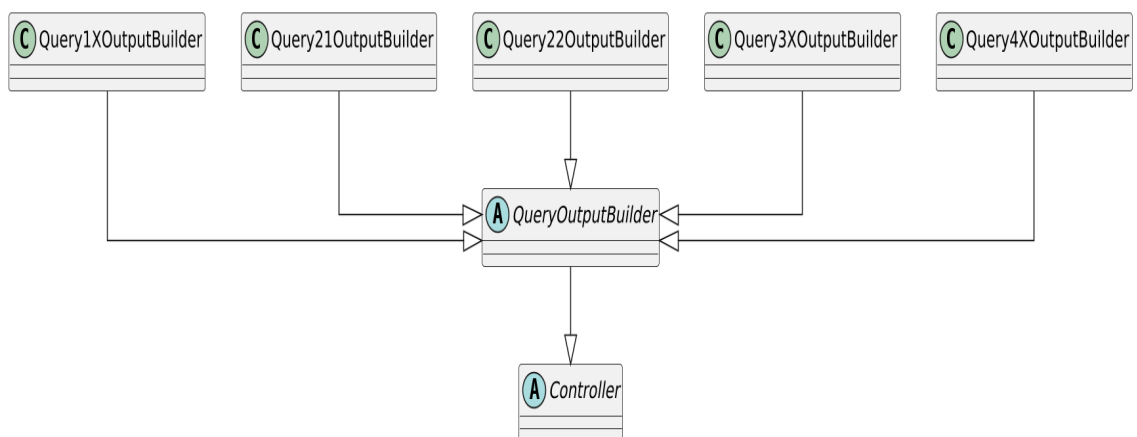


Figura 19: Clase OutputBuilder

12. Diagrama de Grafo Acíclico Dirigido (DAG) - Completo

El flujo de procesamiento completo para resolver las consultas se modela como un Grafo Acíclico Dirigido (DAG). Este diagrama visualiza las dependencias entre las distintas operaciones distribuidas. Cada nodo en el grafo representa una tarea (ej. Clean, Filter by Year, Count Items), y las aristas indican el flujo de datos desde una etapa hacia la siguiente, mostrando cómo se combinan las distintas fuentes de datos para producir los resultados finales (Q1, Q2, Q3, Q4).

12.1. Diagrama de Grafo Acíclico Dirigido (DAG) - Completo

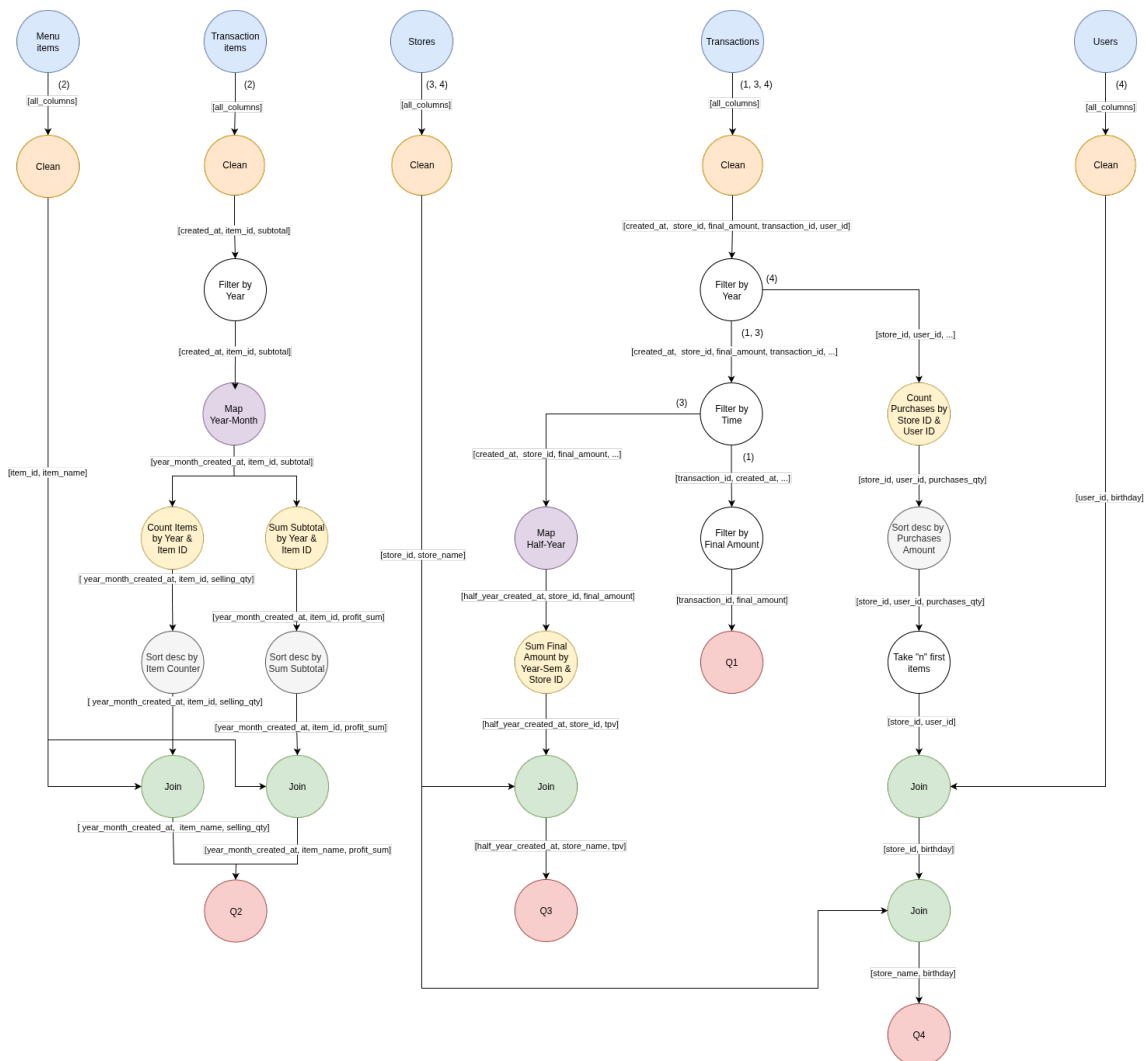


Figura 20: Diagrama de Grafo Acíclico Dirigido (DAG) - Completo

12.2. Diagrama de Grafo Acíclico Dirigido (DAG) - Primera consulta

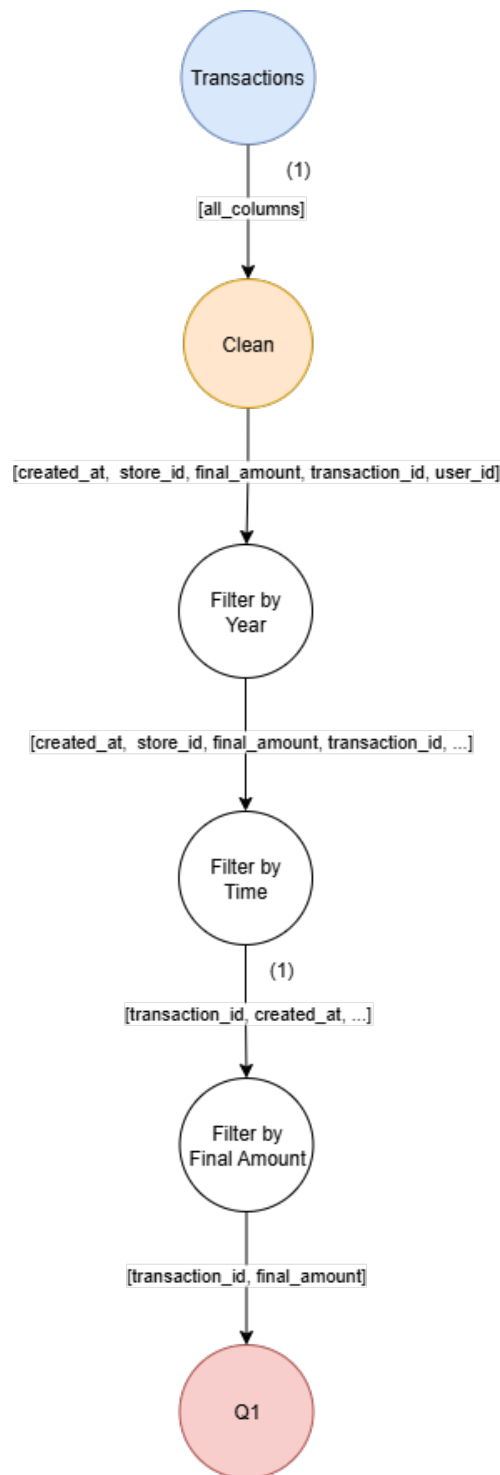


Figura 21: Diagrama de Grafo Acíclico Dirigido (DAG) - Primera consulta

12.3. Diagrama de Grafo Acíclico Dirigido (DAG) - Segunda consulta

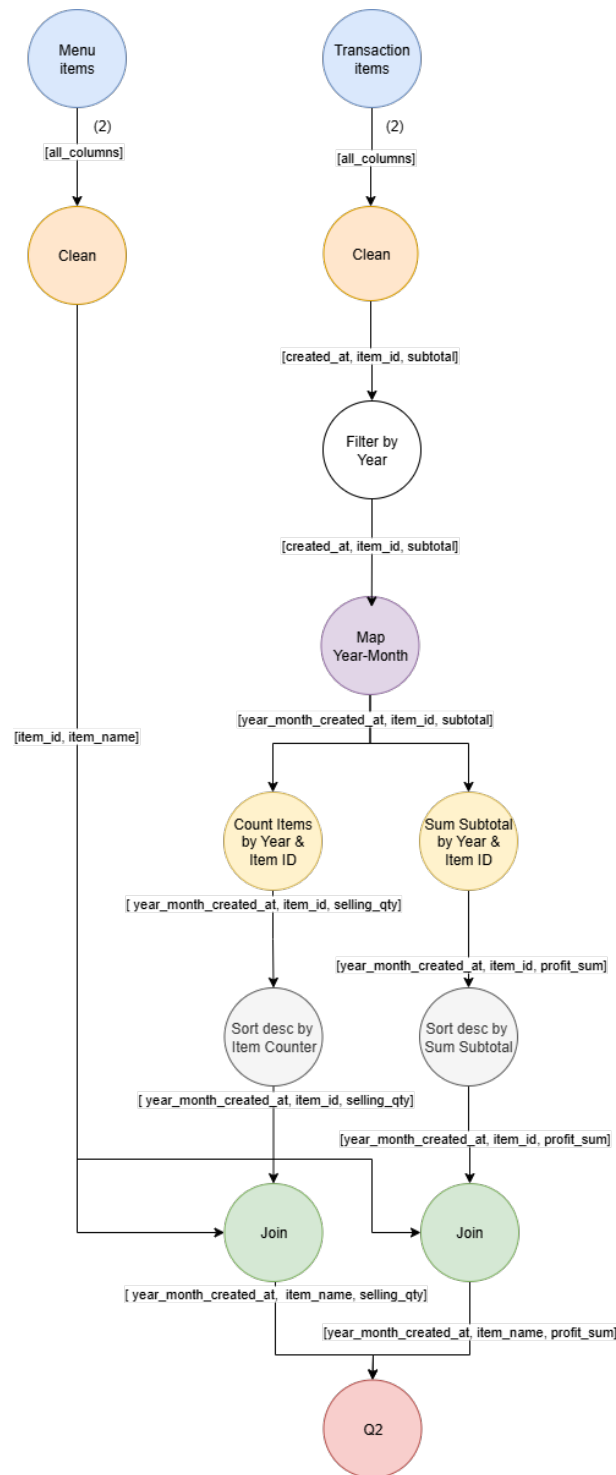


Figura 22: Diagrama de Grafo Acíclico Dirigido (DAG) - Segunda consulta

12.4. Diagrama de Grafo Acíclico Dirigido (DAG) - Tercera consulta

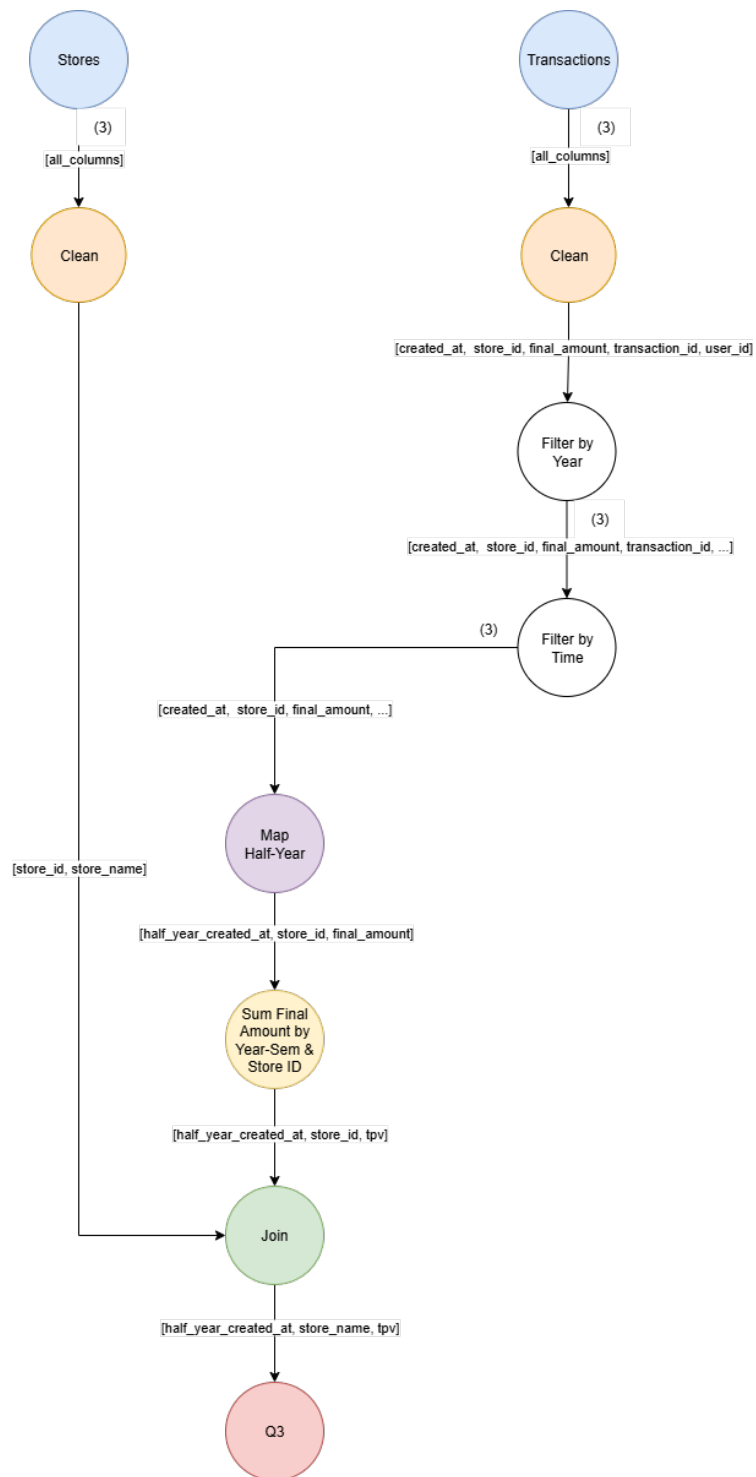


Figura 23: Diagrama de Grafo Acíclico Dirigido (DAG) - Tercera consulta

12.5. Diagrama de Grafo Acíclico Dirigido (DAG) - Cuarta consulta

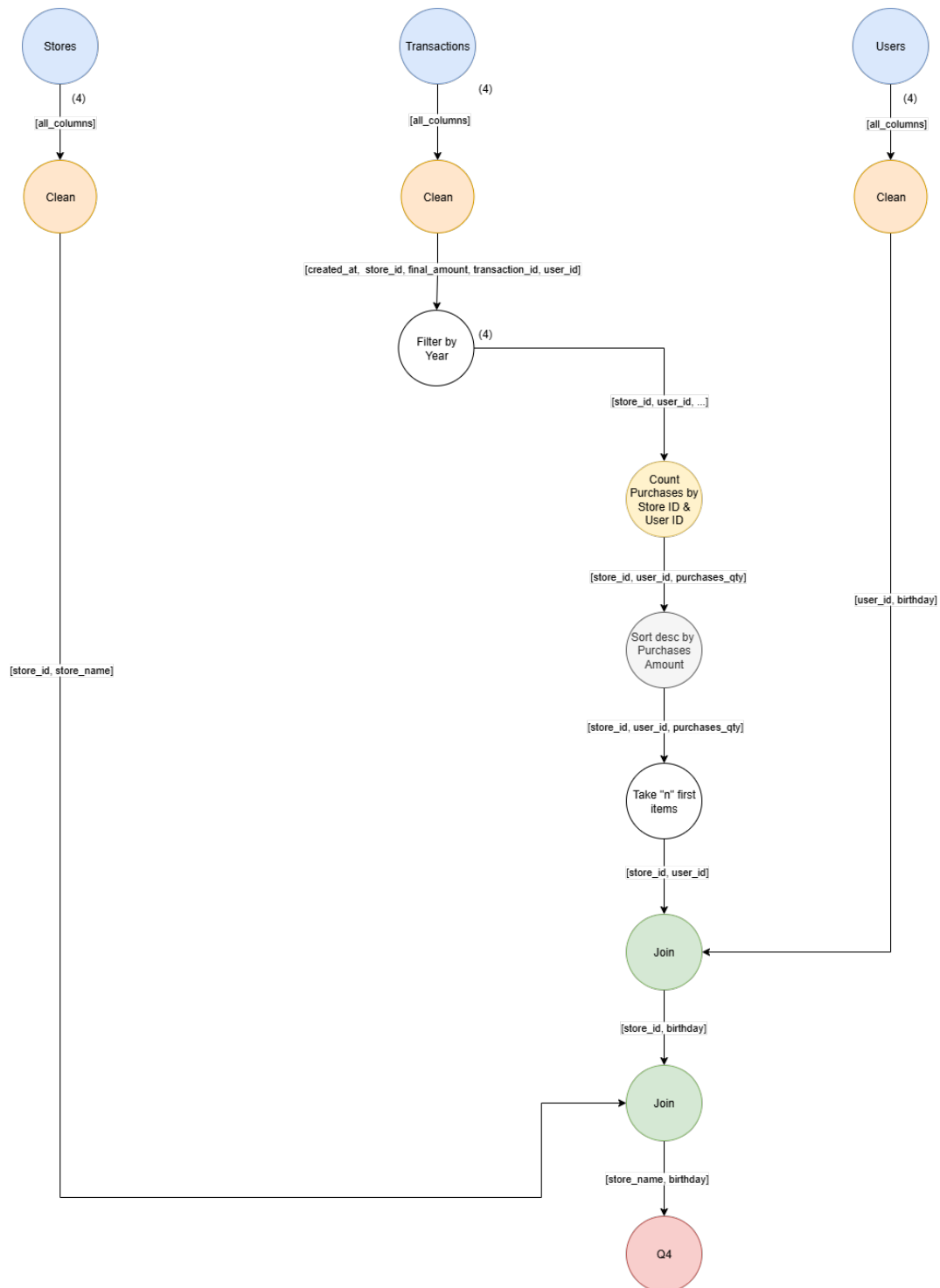


Figura 24: Diagrama de Grafo Acíclico Dirigido (DAG) - Cuarta consulta

13. Middleware

El middleware interno constituye el pilar fundamental de la comunicación en el sistema *Coffee Shop Analysis*, siendo el responsable de orquestar la interacción entre el servidor central y los múltiples nodos de procesamiento. Su arquitectura se basa en un modelo de Middleware Orientado a Mensajes (MOM), implementado mediante la herramienta **RabbitMQ**, que actúa como bróker central y mediador confiable entre los distintos procesos distribuidos, tal como se visualiza en el Diagrama de Despliegue.

La adopción de un middleware de este tipo permite abstraer completamente la complejidad de la comunicación en red, ocultando al programador los detalles de bajo nivel relacionados con sockets, serialización de datos o control de concurrencia. En lugar de conexiones rígidas punto a punto, los componentes se comunican a través de colas y *exchanges* definidos estratégicamente para cada escenario. Este enfoque incrementa notablemente la flexibilidad del sistema, ya que la incorporación de un nuevo nodo de procesamiento, o la eliminación de uno existente, no requiere modificar la lógica de negocio del resto de los participantes.

Otro aspecto central es la asincronía. Gracias al middleware, los nodos no dependen de la disponibilidad inmediata de sus contrapartes, sino que pueden depositar mensajes en una cola para que el receptor los procese cuando corresponda. Esto aporta robustez frente a picos de carga y tolerancia a fallos parciales, ya que los datos no se pierden aunque un nodo se encuentre momentáneamente inactivo. Asimismo, las colas actúan como un mecanismo de *buffering*, desacoplando el ritmo de producción y consumo de datos entre los distintos nodos.

La lógica de conexión se implementa en el componente `MQConnectionHandler`, ubicado en el paquete `shared`, que centraliza la configuración de colas, *exchanges* y canales de comunicación. De este modo, se asegura un manejo uniforme y estandarizado de las operaciones de envío y recepción de mensajes. Sobre esta capa de mensajería se diseñó además un protocolo de aplicación específico que define la estructura de los mensajes para el envío de datasets, la transmisión de resultados y el manejo de errores, lo que permite garantizar la correcta interpretación de la información en todas las etapas del procesamiento.

Finalmente, para interactuar con el middleware se emplearon las interfaces provistas por la cátedra, complementadas con la implementación de pruebas unitarias exhaustivas que validaron su correcto funcionamiento bajo diferentes escenarios de carga. De esta manera, el middleware cumple con el requisito planteado en el enunciado de abstraer la comunicación entre los nodos del sistema distribuido, aportando una base sólida para la escalabilidad, el desacoplamiento y la mantenibilidad del sistema en su conjunto.

Esquemas de comunicación

Durante el desarrollo se identificaron distintos patrones de comunicación, y para cada uno se diseñó una solución particular apoyada en las herramientas de RabbitMQ. A continuación, se describen los principales casos implementados:

Comunicación mediante colas dedicadas

En los nodos escalables, se definió una cola por cada instancia de nodo. Esto asegura la ausencia de *race conditions* y permite direccionar la información de forma controlada, garantizando el funcionamiento correcto y balanceado del sistema.

Para decidir a qué cola enviar cada batch de datos se utilizaron dos estrategias:

1. **Round Robin:** Cada emisor mantiene un contador que se incrementa en cada envío, distribuyendo los lotes de datos de forma equitativa entre las colas disponibles. Una vez alcanzado el último nodo, el contador retorna al primero. Esta técnica fue utilizada en situaciones como:
 - Servidor → Cleaners.
 - Cleaners → Filters.
 - Filters → Filters.
 - Filters → Output Builder.
 - Join Users → Join Stores.
 - Filters → Map Year Month / Map Year Semester.
2. **Hashing por clave (Sharding):** Se empleó en los casos en que los datos debían ser dirigidos a un nodo específico según una clave. Por ejemplo, en la comunicación de:
 - Cleaners de Usuarios → Joins de Usuarios.

La asignación de transacciones a los nodos de Join se resolvió aplicando una función hash sobre el ID de usuario.

Concretamente, se utilizó el resto de la división entera del ID por la cantidad de nodos shardeados, lo que asegura que todas las transacciones de un mismo usuario lleguen al mismo nodo de Join.

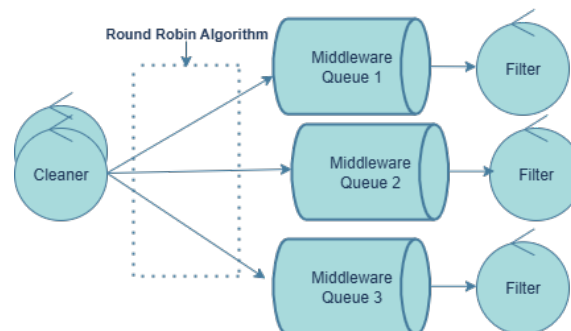


Figura 25: Esquema de comunicación mediante colas dedicadas.

Comunicación mediante exchanges

En situaciones donde la misma información debía ser replicada en múltiples colas, se utilizó un **exchange** de RabbitMQ. Este patrón resultó especialmente útil en la **Query 2**, donde se debía dividir la información para dos subconsultas distintas: los ítems más vendidos y los que generaron mayor facturación.

En este caso, el exchange distribuye la información procesada por el nodo *Map Year Month* hacia las colas de:

- Count Items.
- Sum Items.

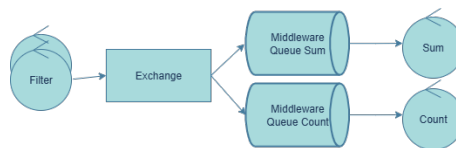


Figura 26: Esquema de comunicación mediante exchanges.

Comunicación de nodos escalables hacia un nodo no escalable

En escenarios donde múltiples nodos escalados debían enviar resultados a un nodo único no escalable (generalmente de tipo *Reduce*), se diseñó una cola de recolección centralizada. Dicho nodo recopilaba toda la información y generaba el reporte final una vez recibidos los resultados de todos los emisores.

Este esquema fue utilizado en situaciones como:

- Filters → Count Purchases.

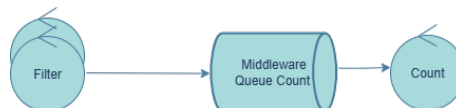


Figura 27: Esquema de comunicación de nodos escalables hacia un nodo único no escalable.

Comunicación 1 a 1

Finalmente, en los casos donde la comunicación era estrictamente secuencial entre nodos no escalables, se implementaron colas directas de 1 a 1. Los principales casos fueron:

- Count Items → Sort Items.
- Count Purchases → Sort Purchases.
- Sum Items → Sort Items.



Figura 28: Esquema de comunicación 1 a 1.

Detección y propagación de EOF

En la nueva versión del sistema, con soporte **MultiClient**, cada cliente opera dentro de su propio *contexto de sesión*, identificado por un `session_id` (UUID). Esto implica que la detección y propagación de los mensajes de fin de flujo (EOF) debe realizarse de forma independiente por sesión, garantizando el cierre correcto de cada flujo de datos asociado a un cliente específico.

El mecanismo general conserva la misma lógica que en la versión de un solo cliente, pero se amplía para manejar múltiples sesiones concurrentes:

- Cada nodo mantiene un **registro por sesión** de los emisores de los que espera recibir EOF. Es decir, no sólo sabe qué nodos le preceden, sino también a qué *session_id* pertenece cada flujo.
- Por cada `session_id`, el nodo espera recibir la cantidad exacta de mensajes EOF correspondientes a sus emisores aguas arriba.
- Cuando un nodo ha recibido todos los EOF esperados para una sesión dada, puede concluir que no habrá más datos de entrada para esa sesión en particular.
- En ese momento, el nodo genera y propaga su propio EOF —incluyendo el mismo `session_id`— hacia todos los nodos que le suceden, utilizando el mismo esquema de enrutamiento:
 - Enviando a una cola directa (1 a 1).
 - Replicando en un *exchange* compartido.
 - Distribuyendo por *Round Robin* entre varias colas consumidoras.
- Este proceso ocurre de forma independiente para cada sesión activa, permitiendo que distintos clientes finalicen sus flujos en momentos diferentes sin interferir entre sí.

De esta manera, el sistema logra una **propagación de EOF por sesión**, garantizando que cada cliente tenga un cierre completo y ordenado de su pipeline de procesamiento. Además, los nodos pueden liberar recursos por sesión una vez propagado el último EOF, mejorando la eficiencia y evitando bloqueos globales entre flujos concurrentes.

14. Mensajes y Protocolos

14.1. Protocolo

Para la implementación del sistema distribuido se actualizó el protocolo de comunicación a un esquema **MultiClient**, que permite multiplexar múltiples clientes concurrentes sobre el mismo backend mediante un *identificador de sesión* embebido en cada mensaje.

Este protocolo define las reglas de interacción entre el nodo coordinador, los nodos de procesamiento y los clientes del sistema. La comunicación se basa en el intercambio de mensajes *auto-contenidos* con un encabezado de tipo fijo, un *identificador de sesión* y un *payload* estructurado. Los mensajes expresan operaciones, datos a procesar y resultados obtenidos, y contemplan el envío por lotes (*batch*) de datasets, una negociación inicial de sesión (HSK) y señales de fin de flujo por tipo (EOF).

14.2. Mensajes

Como parte del protocolo, se mantiene un conjunto de tipos de mensajes de longitud fija (3 caracteres), orientados a cubrir las operaciones básicas del sistema: envío de datasets por lotes y resultados de queries. Con MultiClient, **todo** mensaje lleva además un *session id* entre el tipo y el payload. Cada mensaje posee:

- Un **tipo** de 3 caracteres (p.ej., HSK, MIT, TRN).
- Un **delimitador** de sesión | que separa tipo y *session id*.
- Un **identificador de sesión** (*session id*) opaco, provisto por el cliente.
- Un **delimitador** de inicio de payload [y de fin].
- Un **payload** cuyo formato depende del tipo (texto libre o lote estructurado).

A continuación se detalla el funcionamiento concreto, a partir del código provisto.

14.3. Formato de los mensajes

Estructura general. Todo mensaje sigue ahora el formato:

<TYPE>|<SESSION_ID>[<PAYLOAD>]

Donde:

- <TYPE> es un prefijo de **3 caracteres** (MESSAGE_TYPE_LENGTH = 3).
- <SESSION_ID> es una cadena opaca definida por el cliente. El separador entre tipo y sesión es SESSION_ID_DELIMITER=.
- El **payload** va entre corchetes: [...] (MSG_START_DELIMITER y MSG_END_DELIMITER).

Payload por lotes (batch). Para los mensajes que transportan registros, el payload es una **secuencia de filas**, cada una **encapsulada en llaves** y separada por punto y coma:

```
{<FIELDS>};{<FIELDS>};...
```

Cada <FIELDS> es una secuencia de pares clave-valor separados por comas:

```
<FIELD>,<FIELD>,...
```

Un <FIELD> tiene la forma (comillas **dobles**):

```
"key":"value"
```

Delimitadores usados en el batch:

- Inicio/fin de **cada fila**: BATCH_START_DELIMITER={, BATCH_END_DELIMITER=}
- Separador de filas: BATCH_ROW_SEPARATOR=;
- Separador de campos: ROW_FIELD_SEPARATOR=,

Ejemplo de batch con dos filas (dentro del payload):

```
{"id":"m001","name":"Latte","price":"4.50"};{"id":"m002","name":"Espresso","price":"3.20"}
```

Sin escape de caracteres. No se realiza *escaping*; por lo tanto, las claves y valores no pueden contener comillas dobles ", dos puntos :, comas ,, punto y coma ;, llaves , corchetes [] ni el delimitador de sesión |.

14.4. Tipos de mensajes y su propósito

Tipo (3 chars)	Uso
HSK	<i>Handshake</i> de inicio de sesión MultiClient (negociación).
MIT	Lote de <i>menu items</i> .
STR	Lote de <i>stores</i> .
TIT	Lote de <i>transaction items</i> .
TRN	Lote de <i>transactions</i> .
USR	Lote de <i>users</i> .
Q1X	Resultado para Query 1 (<i>variant X</i>).
Q21	Resultado para Query 2.1.
Q22	Resultado para Query 2.2.
Q3X	Resultado para Query 3 (<i>variant X</i>).
Q4X	Resultado para Query 4 (<i>variant X</i>).
EOF	Señal de fin de flujo (payload: el tipo de flujo que finaliza).

Nota: en esta versión, ACK y QRY no forman parte del *wire protocol*. La coordinación se realiza con HSK y flujos por sesión. Se expone además el *alias* ALL_QUERIES = "Q1X;Q21;Q22;Q3X;Q4X" para expresar capacidades o intereses en el *handshake*.

14.5. Codificación (*encode*)

Mensajes genéricos con sesión. La función privada `_encode_message(type, session_id, payload)` compone:

```
type
|
session_id
[
payload
]
type|session_id[payload]
```

A partir de ella se exponen:

- `encode_handshake_message(id: str, payload: str) -> str:`
Genera `HSKID[payload]`.
- `encode_eof_message(session_id: str, message_type: str) -> str:`
Genera `EOF\textbar session_id[<message_type>]`, donde el *payload* es el tipo de flujo que se declara finalizado (p.ej., `EOF\textbar s1[TRN]`).

Mensajes por lotes. `encode_batch_message(batch_msg_type, session_id, batch):`

1. Para cada `row: dict[str, str]`, codifica campos como `"key":"value"` unidos con `,` y los envuelve con `{}`.
2. Une las **filas codificadas** con `;` (*no hay llaves globales del batch*).
3. Envuelve con el tipo, `|session_id` y corchetes de mensaje.

14.6. Decodificación (*decode*)

Acceso a tipo, sesión y payload.

- `get_message_type(msg):` Devuelve los primeros 3 caracteres. Valida largo mínimo; lanza `ValueError` si el mensaje es demasiado corto.
- `get_message_session_id(msg):` Extrae el *session id* entre `|` y `[`.
- `get_message_payload(msg):` Remueve tipo, `|session_id` y `[]` exteriores, retornando la cadena interna.
- `message_without_payload(msg):` Verdadero si el payload está vacío.

Validación estructural. `_assert_message_format(expected_type, msg)` verifica:

1. Que el tipo decodificado sea el esperado.
2. Que exista el delimitador de sesión `|` y los corchetes `[,]` exteriores.

En caso contrario, lanza `ValueError`.

Decodificación de lotes. `decode_batch_message(msg)`:

1. Obtiene el payload.
2. Separa **filas** por `;`.
3. Para cada fila, aplica `_decode_row`, que:
 - Remueve llaves `{}` en extremos.
 - Separa campos por `,`.
 - Divide cada campo por el primer `:` en clave y valor.
 - Quita comillas dobles exteriores de clave y valor.
4. Devuelve `list[dict[str,str]]`.

Handshake y fin de flujo.

- `decode_handshake_message(msg)` valida HSK y retorna (`session_id`, `payload`). El payload puede expresar capacidades, p.ej. `ALL_QUERIES`.
- `decode_eof_message(msg)` valida EOF y retorna el payload (el *identificador de tipo de flujo* que finaliza; p.ej., `TRN`, `USR`, etc.).

14.7. Ciclos de vida típicos de los mensajes

Establecimiento de sesión (MultiClient).

1. El cliente genera un **session id** (p.ej., `s007`) y emite `HSK|s007[...]` con parámetros de negociación (p.ej., `ALL_QUERIES`).
2. El backend asocia la sesión y, a partir de entonces, **todo** mensaje de/para ese cliente incluirá `s007[, ,]`.

Ingesta de datasets por lotes. Para cada dataset (p.ej., `MIT`, `TRN`):

1. El productor envía uno o más mensajes **batch** del tipo correspondiente, incluyendo su *session id* (p.ej., `MIT|s007[...]`).
2. Al finalizar el stream de ese dataset, el productor envía `EOF|s007[<TIPO>]`, por ejemplo `EOF|s007[TRN]`.

Consultas y resultados.

1. Un cliente puede declarar interés/capacidades en HSK (p.ej., `ALL_QUERIES`).
2. Los resultados se retornan *por sesión* en mensajes `Q1X`, `Q21`, `Q22`, `Q3X` o `Q4X`, con payload acorde, por ejemplo `Q21|s007[...]`.

14.8. Ejemplos de mensajes codificados

Handshake con capacidades.

```
HSK|s007[Q1X;Q21;Q22;Q3X;Q4X]
```

Batch de menu items (2 filas).

```
MIT|s007[{"id":"m01","name":"Latte","price":"4.50"};{"id":"m002","name":"Doble","price":"3.20"}]
```

Fin de flujo de transacciones.

```
EOF|s007[TRN]
```

Resultado de Query 2.1 (payload libre en este TP).

```
Q21|s007[{"store_id":"s007","metric":"top_seller","value":"m001"}]
```

14.9. Validaciones, errores y robustez

- **Tipo y formato:** Todo decodificador específico verifica que el tipo de mensaje coincida con el esperado, que exista el delimitador de sesión | y que el payload esté entre [, ,]. Inconsistencias lanzan `ValueError`.
- **Longitud mínima:** `get_message_type` exige al menos 3 caracteres (tipo). Mensajes más cortos disparan `ValueError`.
- **Payload vacío:** Admitido (`message_without_payload`).
- **Complejidad:** Las rutinas de encode/decode son lineales en la longitud del mensaje, con operaciones de `split/join` sobre separadores fijos, facilitando el procesamiento por streaming.

14.10. Supuestos y limitaciones deliberadas

- **Sin escape de caracteres:** Claves y valores no deben contener comillas dobles ", dos puntos :, comas ,, punto y coma ;, llaves/corchetes, ni el delimitador de sesión |. Esto simplifica y acelera el parser a costa de restringir el dominio de valores posibles (suficiente para este TP).
- **Tipos de datos:** Todos los valores se tratan como cadenas (`str`); la tipificación semántica queda a cargo de las capas de negocio.
- **Orden y entrega:** El protocolo describe *formato* y *semántica de alto nivel* (handshake, batches, EOF) y la multiplexación por sesión. Las garantías de orden, reintentos o *at-least-once/exactly-once* pertenecen a la capa de transporte y orquestación (p.ej., colas), tratadas en otra sección del informe.

14.11. Extensibilidad

El diseño con prefijos de 3 caracteres y *session id* explícito facilita agregar nuevos tipos de mensajes sin afectar a los existentes. Para incorporar uno nuevo basta con:

1. Declarar la constante del tipo (3 letras).
2. Implementar, si corresponde, *helpers* de encode/decode análogos a los de batch.
3. Documentar el payload asociado (texto libre o esquema de campos del batch).

14.12. Resumen

El protocolo MultiClient define un **formato compacto, determinista y con multiplexación**:

- encabezado de 3 caracteres para el tipo
- separador | y *session id* obligatorio
- delimitadores exteriores [, ,]
- para datasets, un **batch** como secuencia *fila-a-fila*: {} ; con separadores simples

Se proveen *helpers* específicos para codificar/decodificar los distintos datasets, un **HSK** para iniciar la sesión y una señal **EOF** que marca el fin de cada flujo lógico por sesión. Las validaciones estructurales minimizan errores de parseo y el diseño favorece procesamiento lineal, multi-cliente y extensibilidad controlada para futuras necesidades del TP.

15. Mediciones de rendimiento del sistema implementado

Para evaluar el rendimiento del sistema distribuido implementado, se realizaron diversas mediciones bajo diferentes configuraciones y cargas de trabajo, poniendo especial énfasis en la capacidad del sistema para manejar múltiples clientes en ejecución simultánea (*MultiClient*) sobre el dataset completo.

A continuación, se detallan los resultados obtenidos y su análisis correspondiente.

15.1. Configuración del entorno de pruebas

Las pruebas se llevaron a cabo en un entorno controlado, utilizando una red local con varios nodos de cómputo interconectados. Durante las mediciones, el sistema fue desplegado de manera que todos los nodos escalables ajustaran automáticamente su cantidad de instancias según lo considerado óptimo para mantener un equilibrio entre rendimiento, utilización de recursos y tiempo de respuesta.

En otras palabras, no se fijó una cantidad estática de nodos de cómputo por servicio, sino que el sistema escaló dinámicamente cada componente según la demanda generada por la cantidad de clientes concurrentes y el volumen de procesamiento requerido en cada fase de ejecución.

15.2. Escenarios de prueba

Se midió el comportamiento del sistema con **1, 2 y 3 clientes** en ejecución simultánea, todos procesando el **dataset completo** y ejecutando las **cuatro consultas** definidas por la cátedra. Cada cliente opera de manera aislada dentro de su propia sesión (`session_id`), manteniendo así independencia lógica entre los flujos y permitiendo evaluar la escalabilidad real del sistema frente a cargas concurrentes.

15.3. Tiempos de procesamiento obtenidos

Los resultados medidos fueron los siguientes:

- **1 cliente:** El tiempo total de procesamiento fue **X minutos y X segundos**.
- **2 clientes:** El tiempo total de procesamiento fue **X minutos y X segundos**.
- **3 clientes:** El tiempo total de procesamiento fue **X minutos y X segundos**.

Se observa que, si bien el tiempo total aumenta con la cantidad de clientes activos, el incremento no es lineal. El sistema logra mantener una alta eficiencia en la distribución de tareas gracias a la paralelización de los flujos y al escalado dinámico de los nodos de cómputo.

16. Mecanismos de Control

16.1. Mecanismos de Control de Sincronización

En el diseño del sistema distribuido cuenta con la incorporación de mecanismos de control de sincronización, con el objetivo de evitar problemas asociados a la concurrencia, tales como *race conditions* o *deadlocks*.

Estos mecanismos garantizan un acceso ordenado a los recursos compartidos y la correcta coordinación entre los procesos en ejecución.

16.2. Mecanismos de Control de Señales y Finalización

El sistema incorpora un manejo explícito de señales a fin de asegurar una finalización correcta y segura de los procesos.

En particular, se controla la señal **SIGTERM**, permitiendo un *graceful quit* que libera recursos, cierra todos los archivos abiertos, se asegura de que todos los file descriptors sean cerrados de forma correcta, mediante a los métodos de 'Close' y 'Delete' brindados por el Middleware.

Este mecanismo previene pérdidas de datos, mantiene la consistencia del sistema y asegura el correcto uso de los recursos del sistema operativo.

16.3. Mecanismos de Control de Fallas

Se contemplará el desarrollo de mecanismos de control de fallas para aumentar la robustez y confiabilidad del sistema. Estos mecanismos estarán orientados a gestionar situaciones adversas como caídas de procesos, pérdida de nodos de cómputo o interrupciones inesperadas durante la ejecución.

El objetivo es garantizar, en la medida de lo posible, la continuidad del procesamiento y la recuperación parcial de información. En futuras versiones del documento se describirán las estrategias de detección, mitigación y recuperación ante fallas.

17. Tests

Con el objetivo de garantizar la correcta operación, consistencia y determinismo del sistema distribuido desarrollado, se implementó un conjunto integral de pruebas (*tests*) que abordan distintos niveles de validación funcional y estructural.

En total, el sistema cuenta con **tres grandes grupos de tests**:

- **Tests de Middleware**
- **Tests de Comparación de Outputs**
- **Tests de Propagación de EOF**

A continuación, se describen los objetivos, el alcance y los criterios de validación de cada grupo de pruebas.

17.1. Tests de Middleware

Estos tests se encargan de verificar la **correcta implementación y funcionamiento del Middleware** desarrollado sobre la biblioteca de comunicación **RabbitMQ**, siguiendo la interfaz y las especificaciones brindadas por la cátedra.

El objetivo principal de este conjunto de pruebas es asegurar que la capa de comunicación entre nodos funcione de manera estable, confiable y conforme al protocolo definido, tanto para el envío como para la recepción de mensajes. Entre los aspectos validados se incluyen:

- Correcta creación y vinculación de colas y *exchanges*.
- Enrutamiento adecuado de mensajes según el tipo de flujo o dataset.
- Manejo de *acknowledgements* y confirmaciones de entrega.
- Comportamiento esperado del middleware bajo condiciones de carga y escalado.
- Gestión de excepciones.

Estos tests fueron desarrollados originalmente para la entrega “**Escalabilidad, Middleware y Coordinación de Procesos**”, y sirvieron de base para validar la estabilidad de la infraestructura de mensajería del sistema.

17.2. Tests de Comparación de Outputs

El segundo conjunto de pruebas está orientado a la **validación funcional de los resultados** obtenidos por el sistema distribuido, comparando las salidas producidas por las consultas con los resultados esperados.

Dado que en un entorno distribuido las tareas se dividen entre nodos escalables sin garantizar un orden específico en la entrega de resultados, fue necesario diseñar técnicas de comparación **insensibles al orden** para validar el correcto comportamiento determinístico del sistema. De esta forma, las pruebas se centran en comprobar la equivalencia semántica de los resultados más allá del orden particular de las filas.

Validación por consulta.

- **Query 1:** Verifica que la cantidad de líneas generadas sea exactamente la misma que la esperada, confirmando que el volumen de resultados es correcto.
- **Query 2:** Compara tanto la cantidad de filas como el contenido de cada una, sin considerar el orden en que fueron entregadas. De este modo, se valida que los resultados sean completos y correctos, independientemente del orden de procesamiento distribuido.
- **Query 3:** Aplica los mismos criterios que en la Query 2, validando que las filas generadas contengan la misma información y que no haya pérdidas ni duplicados, aún cuando el orden de emisión varíe entre ejecuciones.
- **Query 4:** Verifica que:
 - La cantidad total de filas coincida con la esperada.
 - Exista la misma cantidad de respuestas por cada tienda.
 - Los clientes identificados como los más compradores por tienda coincidan en cantidad de compras.

No se consideran las diferencias en nombres o fechas de nacimiento, ya que en caso de empates el orden de los tres máximos puede variar entre ejecuciones. Lo relevante es que los valores reportados correspondan a los tres mayores por tienda, garantizando así la corrección lógica de la consulta.

Estos tests fueron desarrollados inicialmente para la entrega “**Escalabilidad, Middleware y Coordinación de Procesos**” y luego **mejorados para la entrega “MultiClient”**.

17.3. Tests de Propagación de EOF

Finalmente, se implementó un conjunto de pruebas específicas para validar el **mecanismo de propagación de mensajes EOF** dentro del sistema *MultiClient*.

El objetivo de estos tests es comprobar que:

- Cada controlador reciba correctamente los EOF asociados a los flujos que le corresponden, diferenciados por `session_id`.
- La propagación de EOF se realice de forma completa y ordenada hacia los nodos sucesores en la cadena de procesamiento.
- No existan pérdidas, duplicados ni propagaciones cruzadas entre sesiones.

De esta manera, se valida que el mecanismo de cierre de flujos por sesión funcione correctamente y que el sistema distribuido finalice el procesamiento de cada cliente de forma independiente y controlada, sin afectar el estado de los demás flujos activos.

Estos tests fueron desarrollados como parte de la entrega “**MultiClient**”, a pedido de la cátedra, para asegurar la correcta implementación del sistema de detección y propagación de fin de flujo dentro del entorno multiusuario.

18. Cronograma teórico del desarrollo

18.1. Diseño

Previo al inicio del desarrollo efectivo del diseño, el equipo realizó una instancia de planificación para organizar la distribución de responsabilidades. El objetivo fue asegurar que cada integrante asumiera un conjunto de tareas equilibrado, alineado tanto con sus fortalezas como con las necesidades del proyecto.

La planificación se estructuró en función de las vistas arquitectónicas requeridas, los diagramas de modelado, la documentación conceptual y las definiciones de base.

18.2. Escalabilidad

El desarrollo de esta entrega cuenta con un tiempo estimado de 2.5 semanas. Durante la primera semana se realizará un análisis detallado de los criterios de escalabilidad, los requisitos técnicos y el alcance de la entrega, así como la planificación y distribución de tareas entre los integrantes del equipo.

En las semanas posteriores se definirán y ejecutarán las labores específicas de cada integrante, las cuales se documentarán en versiones futuras de este informe. La presente sección cumple únicamente la función de establecer la planificación inicial.

18.3. Multi-Client

El desarrollo de esta entrega cuenta con un tiempo estimado de 2 semanas. La primera semana estará destinada al análisis de los criterios, requisitos y alcance de la funcionalidad de múltiples clientes, junto con la planificación y asignación de tareas a cada integrante del equipo.

En la semana restante se abordará la ejecución de las labores planificadas, que se detallarán en próximas versiones de este documento. En esta instancia se deja asentada únicamente la planificación teórica.

18.4. Tolerancia

El desarrollo de esta entrega cuenta con un tiempo estimado de 3.5 semanas. La primera semana se dedicará al análisis de criterios, requisitos de tolerancia a fallos y alcance de la entrega, además de la planificación y distribución de responsabilidades entre los integrantes.

Las semanas siguientes estarán orientadas a la implementación progresiva de los mecanismos planificados. Las labores específicas por integrante se incluirán en futuras actualizaciones de este documento.

18.5. Paper

El desarrollo de esta entrega cuenta con un tiempo estimado de 1.5 semanas. Durante la primera semana se llevará a cabo el análisis de criterios, requisitos de formato y alcance de la entrega, así como la planificación y reparto de tareas.

El tiempo restante será utilizado para la redacción y consolidación del documento final, cuyos detalles se incorporarán en versiones posteriores de este informe. La presente sección constituye únicamente la documentación inicial de la planificación.

19. Cronograma real del desarrollo

19.1. Diseño (2 semanas)

Durante la **primera semana**, el grupo se enfocó en interpretar en profundidad los requisitos planteados en el enunciado, analizar la lógica de las consultas y definir el diseño conceptual del sistema. Asimismo, en esta etapa se realizó la repartición de tareas entre los integrantes, de manera de optimizar el tiempo de ejecución restante.

En la **segunda semana**, cada integrante se abocó a las actividades asignadas, según el siguiente detalle:

Luciano

- Vista Lógica.
- Vista de Desarrollo.
- Vista Física.
- Vista de Procesos.
- Diseño inicial/conceptual del Middleware.
- Diagrama de Paquetes.

Axel

- Escenarios de uso (Casos de Uso).
- Diagrama de Secuencia.
- Diagrama de Actividades.
- Diagrama de Robustez.
- DAG (Directed Acyclic Graph).

Felipe

- Redacción de las definiciones iniciales.
- Creación de los cronogramas.
- Descripción detallada de la ejecución de las consultas.
- Explicaciones introductorias de los mecanismos de control y protocolos.

19.2. Middleware y Coordinación de Procesos (2.5 semanas)

Durante la **primera semana**, el grupo se enfocó en interpretar en profundidad los requisitos planteados en el enunciado, analizar posibles implementaciones para las consultas solicitadas, hacer las correcciones de diseño necesarias, pactar protocolos e identificar bibliotecas necesarias para el desarrollo.

En la **segunda semana**, cada integrante se abocó a las actividades asignadas, detalladas a continuación:

Luciano

- Diseño funcional integral de colas y exchanges en el sistema para la topología diseñada.
- Investigación y selección de la biblioteca para la comunicación mediante RabbitMQ.
- Investigación y selección de la biblioteca para la serialización/deserialización de mensajes.
- Implementación del Middleware.
- Creación de los tests unitarios para validar funcionamiento del Middleware.
- Establecimiento de conexiones entre Cliente, Servidor y Middleware.
- Redacción de protocolos de comunicación.
- Integración del Middleware con los controladores.
- Construcción del ecosistema de nodos de cómputo distribuido mediante a Dockerfiles.

Axel

- Diseño funcional integral de colas y exchanges en el sistema para la topología diseñada.
- Implementación de los controladores:
 - Join.
 - Sorts.
 - Maps.
 - Filters.
- Integración de los controladores con el Middleware.
- Construcción del ecosistema de nodos de cómputo distribuido mediante a Dockerfiles.
- Medición del rendimiento del sistema.

Felipe

- Diseño funcional integral de colas y exchanges en el sistema para la topología diseñada.
- Implementación de los controladores:
 - Cleaners.
 - Output Builders.
 - Reduces.
- Integración de los controladores con el Middleware.
- Redacción de la nueva documentación del sistema.
- Relevamiento y control de los mecanismos de cierre 'graceful' de los nodos de cómputo.

19.3. Multi Client (2 semanas)

Durante la **primera semana**, repitiendo la operativa realizada en entregas previas, el grupo se enfocó en interpretar en profundidad los requisitos planteados en el enunciado, analizar posibles implementaciones para las consultas solicitadas, hacer las correcciones de diseño necesarias, pactar protocolos e identificar bibliotecas necesarias para el desarrollo.

En la **segunda semana**, cada integrante se abocó a las actividades asignadas, detalladas a continuación:

Luciano

- Planificación y diseño de la arquitectura multi cliente.
- Implementación de la lógica multi cliente adaptada al Servidor.
- Implementación de la lógica multi cliente adaptada a los Controladores Statefull.
- Redacción del nuevo protocolo de comunicación multi cliente.
- Refactorización de los controladores para soportar múltiples sesiones concurrentes.
- Adaptación del sistema para configurar la escalabilidad en base al archivo '.env'.

Axel

- Planificación y diseño de la arquitectura multi cliente.
- Implementación de la lógica multi cliente adaptada al Cliente.
- Implementación de la lógica multi cliente adaptada a los Controladores Stateless.
- Refactorización de los controladores para soportar múltiples sesiones concurrentes.
- Mediciones de rendimiento para múltiples clientes concurrentes.
- Redacción de datasets reducidos para la demostración del sistema en la defensa.

Felipe

- Planificación y diseño de la arquitectura multi cliente.
- Análisis de puntos escalables para optimización del rendimiento.
- Redacción de la nueva documentación del sistema.
- Redacción de los nuevos tests de validación funcional de outputs y propagación de EOF.
- Redacción de la presentación para la defensa de la entrega.
- Relevamiento y control de cumplimiento de los criterios de aceptación.