.UBAfiuba
FACULTAD DE INGENIERÍA

Sistemas Distribuidos 1
(TA050) Roca

Diseño
Coffee Shop Analysis

Fecha: 2 de octubre de 2025

Ascencio Felipe Santino - 110675

Gamberale Luciano Martín - 105892

Zielonka Axel - 110310

Índice

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 respetera 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 Çafetería", y las cuatro funcionalidades clave que el sistema debe proveer, correspondiendo directamente a las consultas de negocio definidas en los requerimientos.

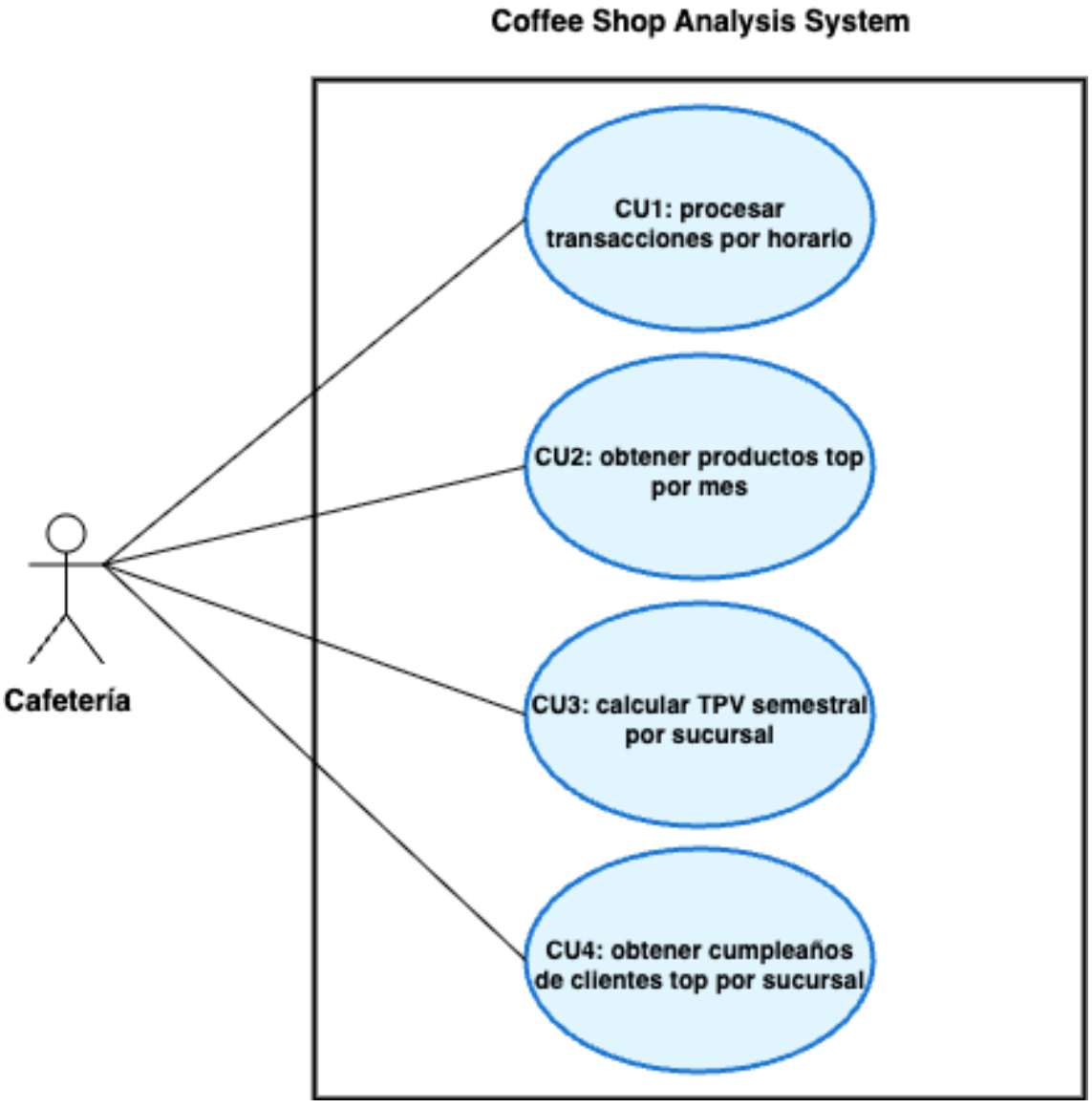


Figura 1: Casos de Uso

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

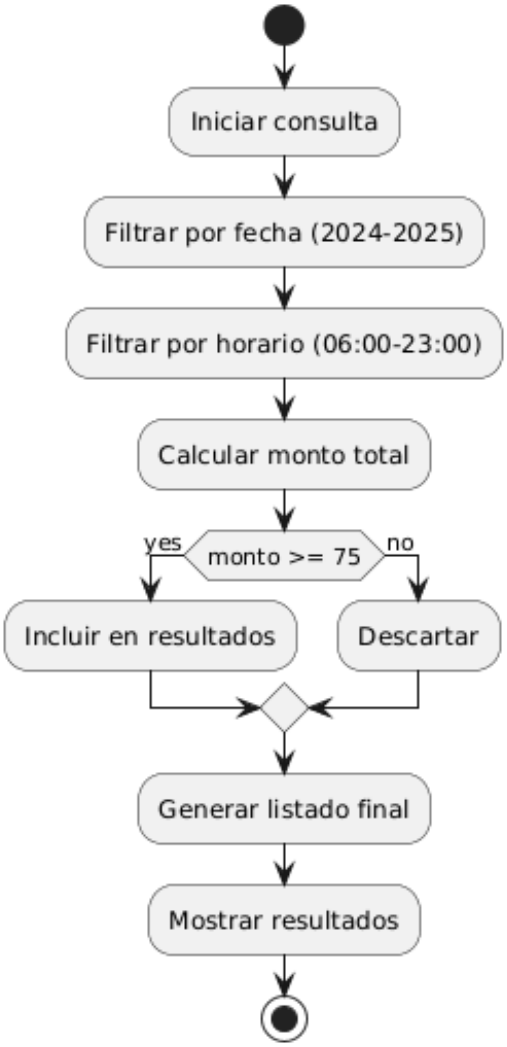


Figura 2: Actividad de la primera consulta

8.1.2. Diagrama de Actividades de la segunda consulta

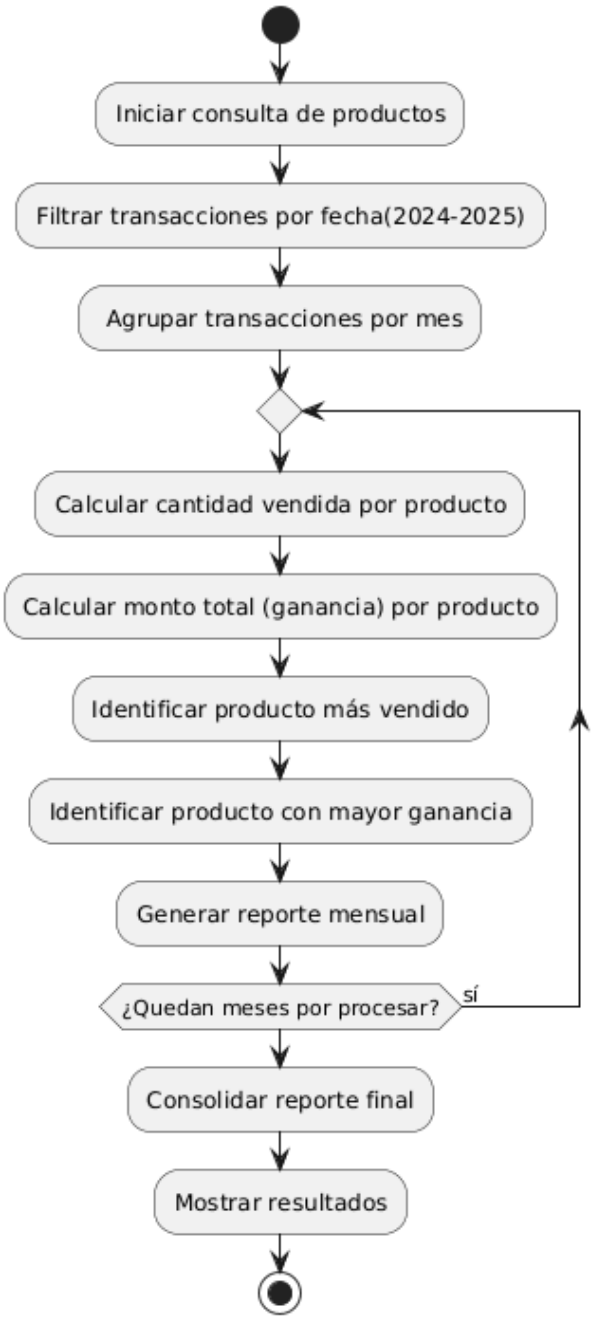


Figura 3: Actividad de la segunda consulta

8.1.3. Diagrama de Actividades de la tercera consulta

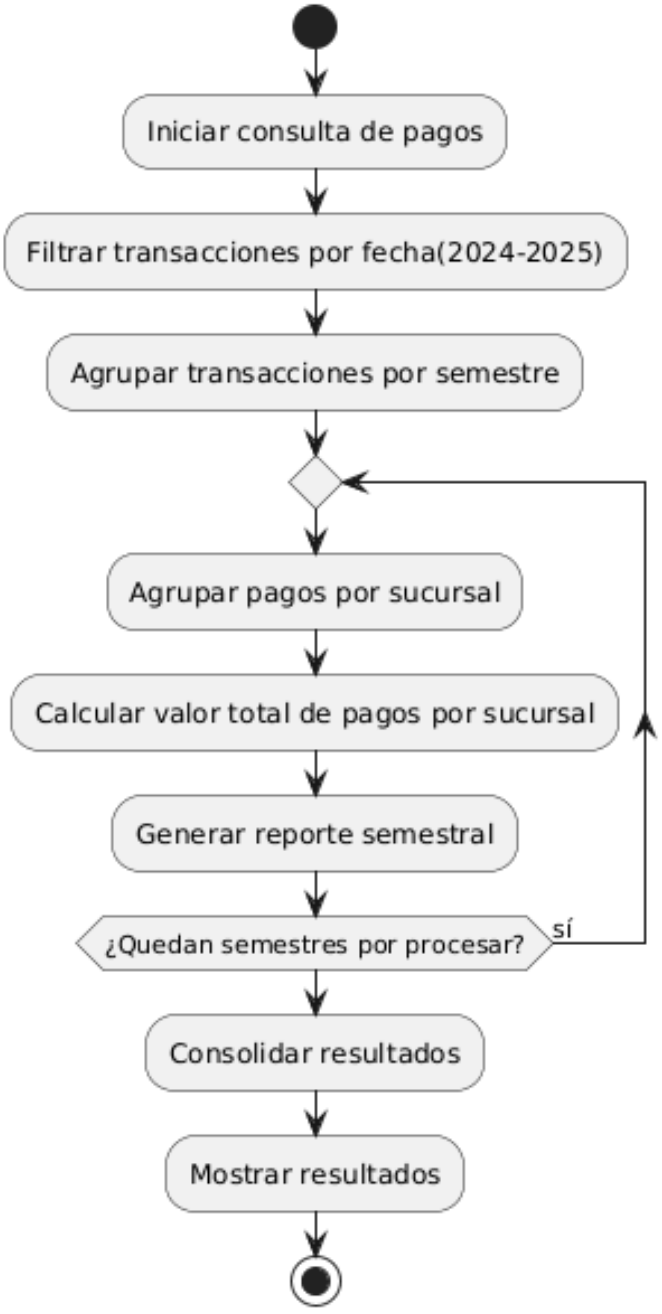


Figura 4: Actividad de la tercera consulta

8.1.4. Diagrama de Actividades de la cuarta consulta

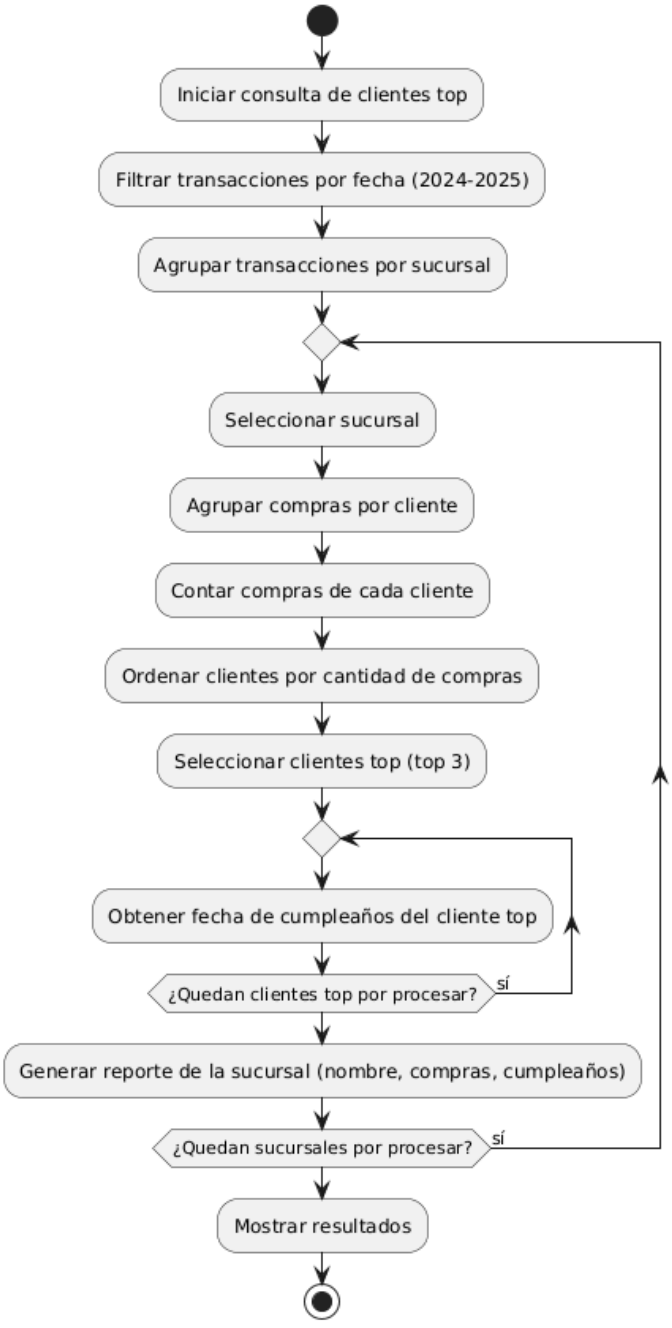


Figura 5: Actividad de la cuarta consulta

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

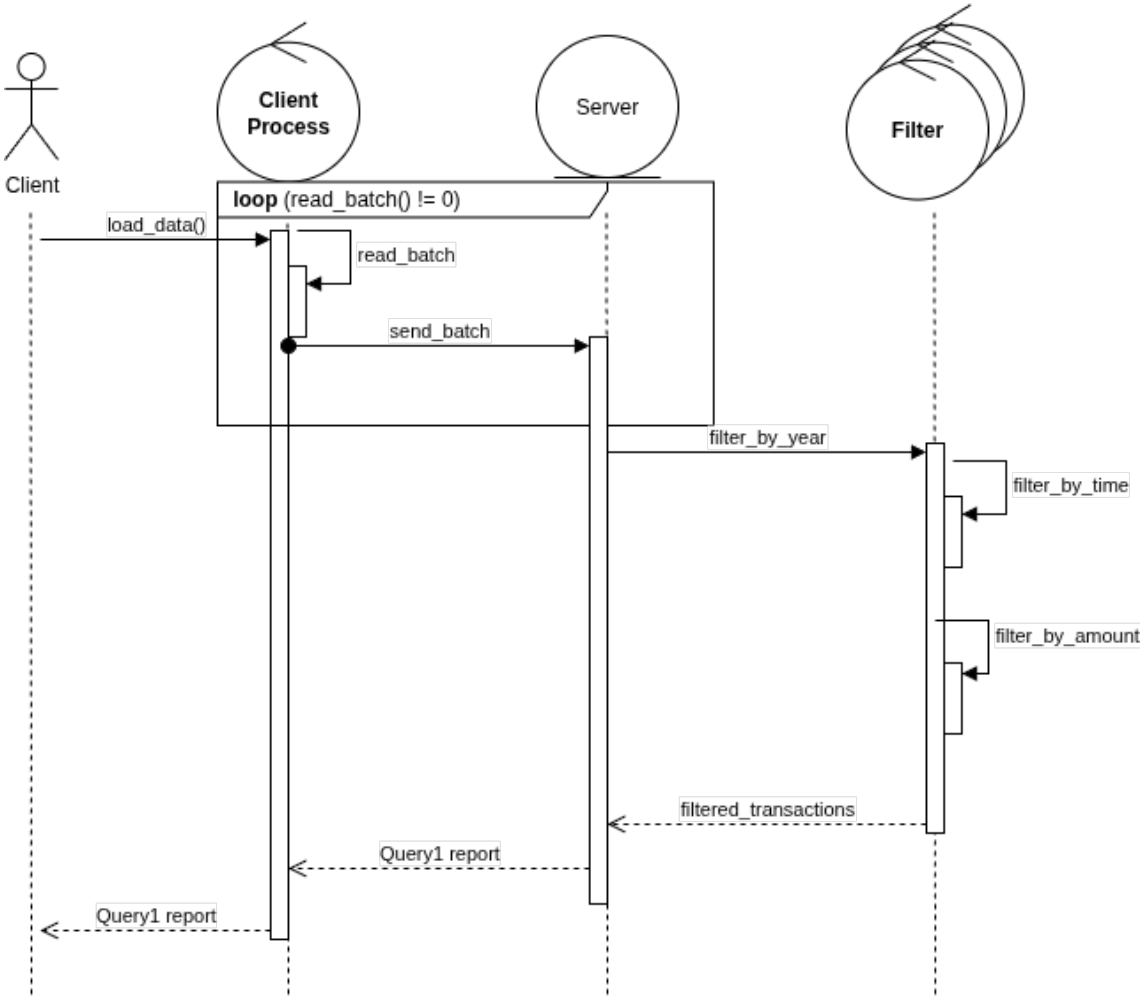


Figura 6: Secuencia de la primera consulta

8.2.2. Diagrama de Secuencia de la segunda consulta

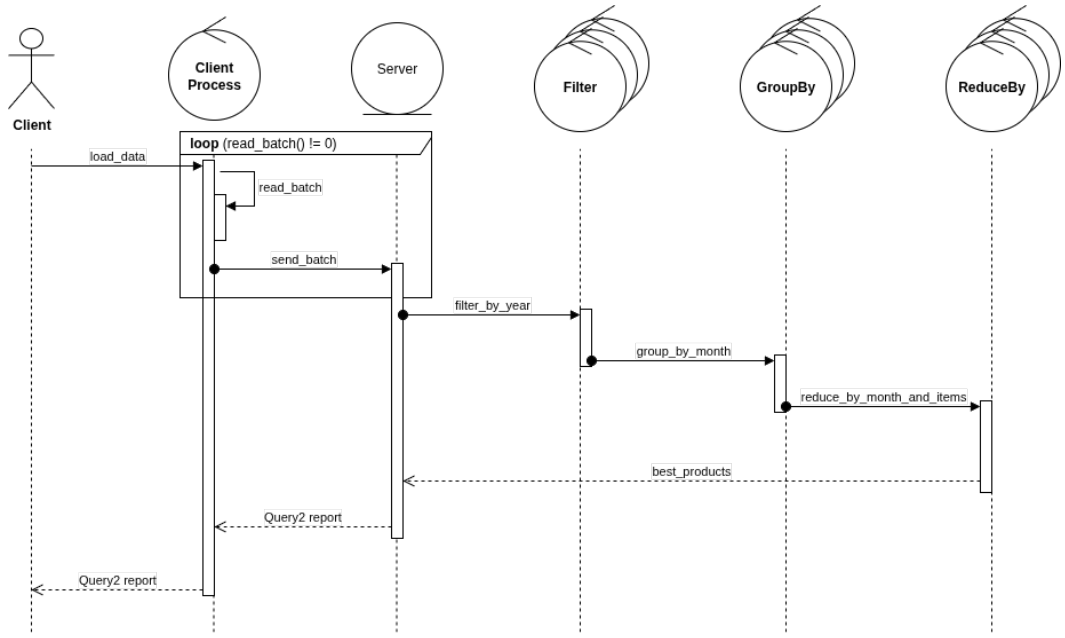


Figura 7: Secuencia de la segunda consulta

8.2.3. Diagrama de Secuencia de la tercera consulta

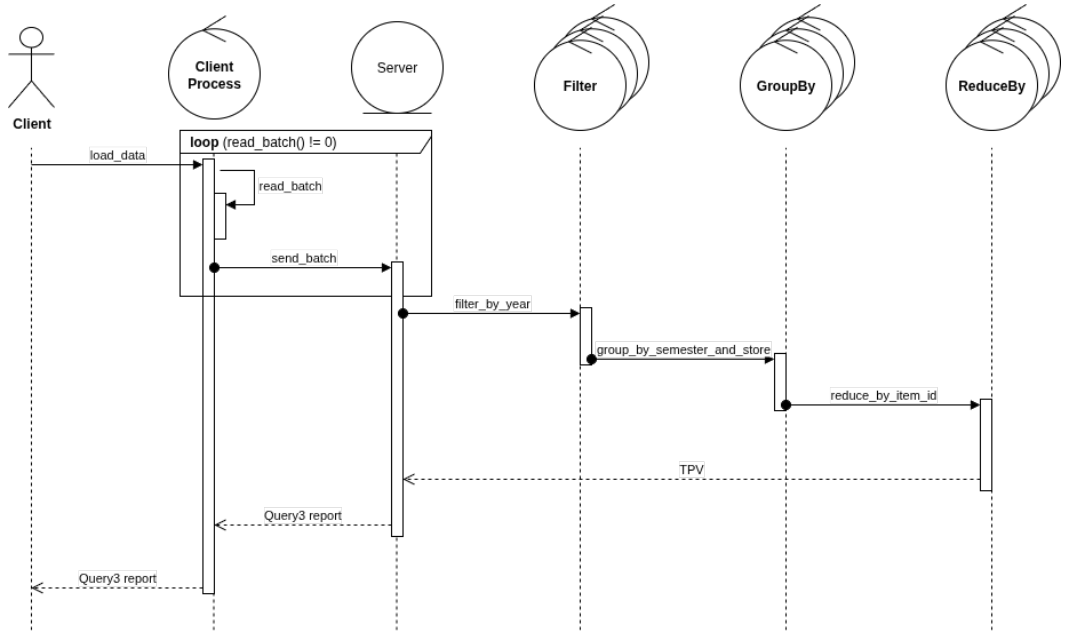


Figura 8: Secuencia de la tercera consulta

8.2.4. Diagrama de Secuencia de la cuarta consulta

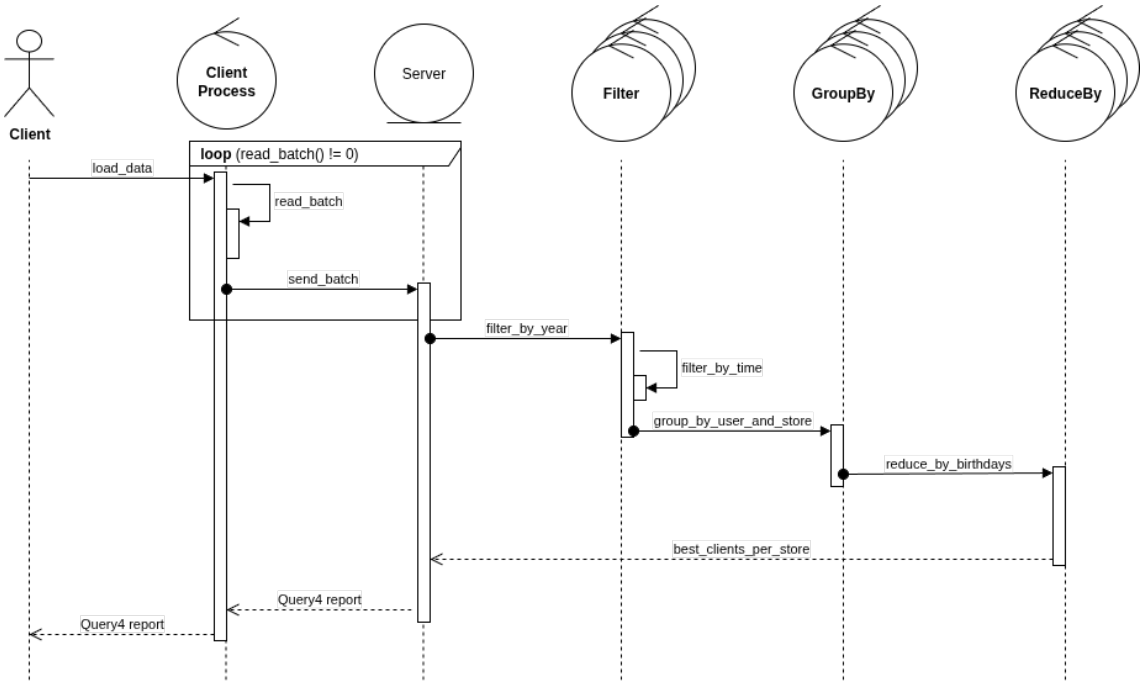


Figura 9: Secuencia de la cuarta consulta

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.

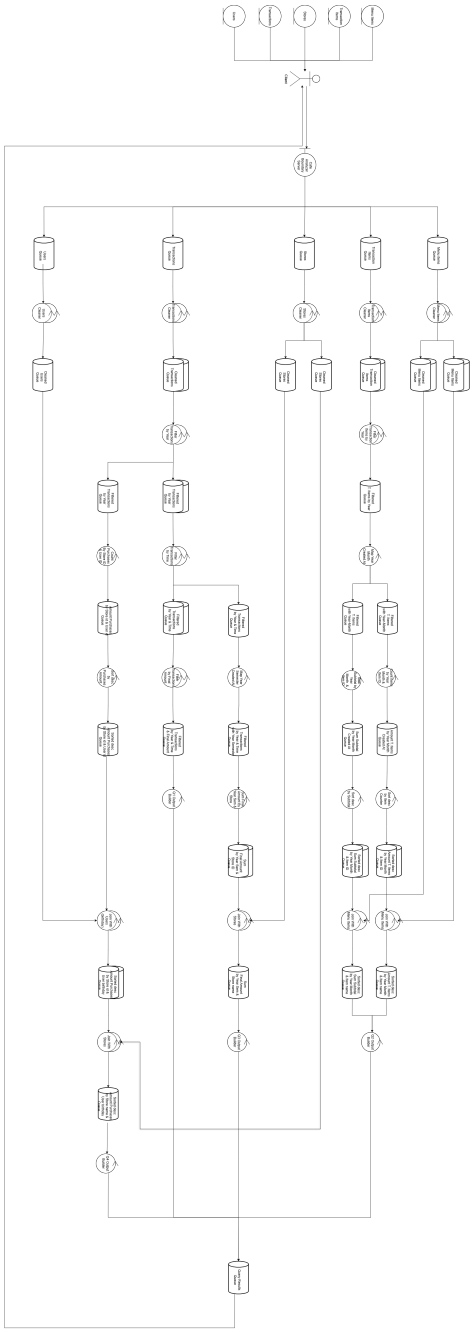
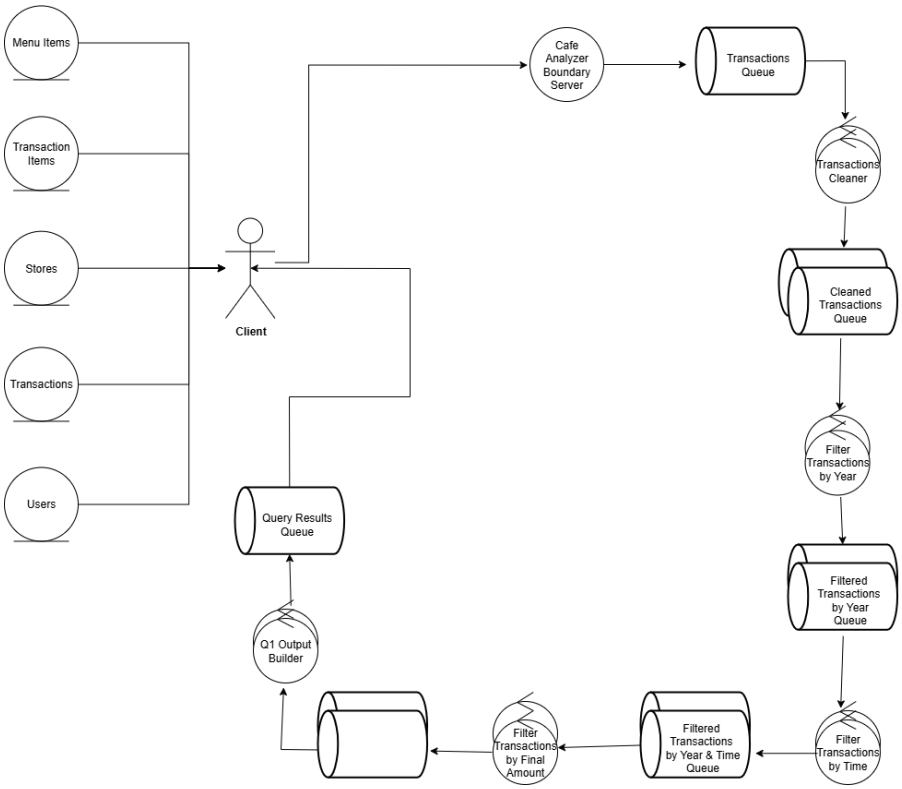
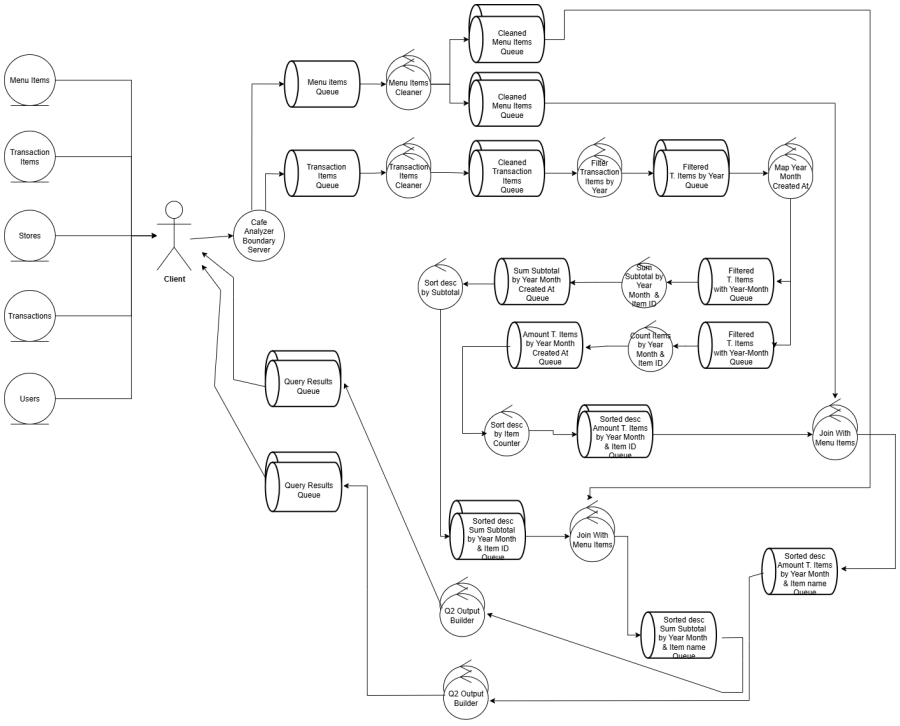


Figura 10: Diagrama de Robustez - Completo

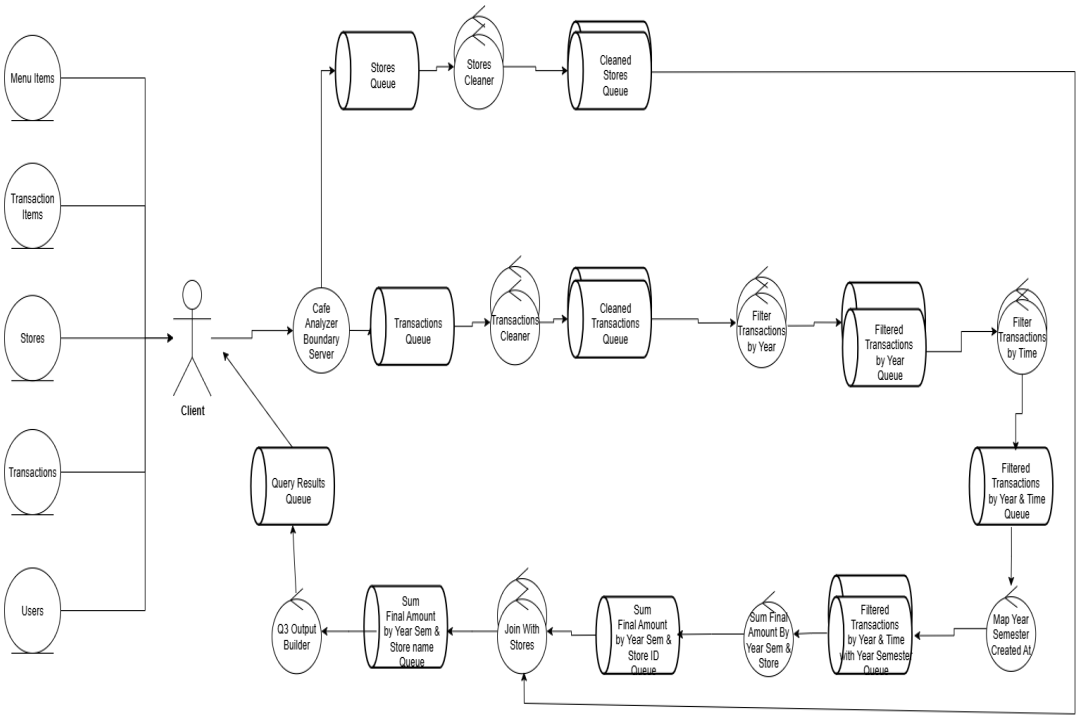
9.2. Diagrama de Robustez - Primera Consulta



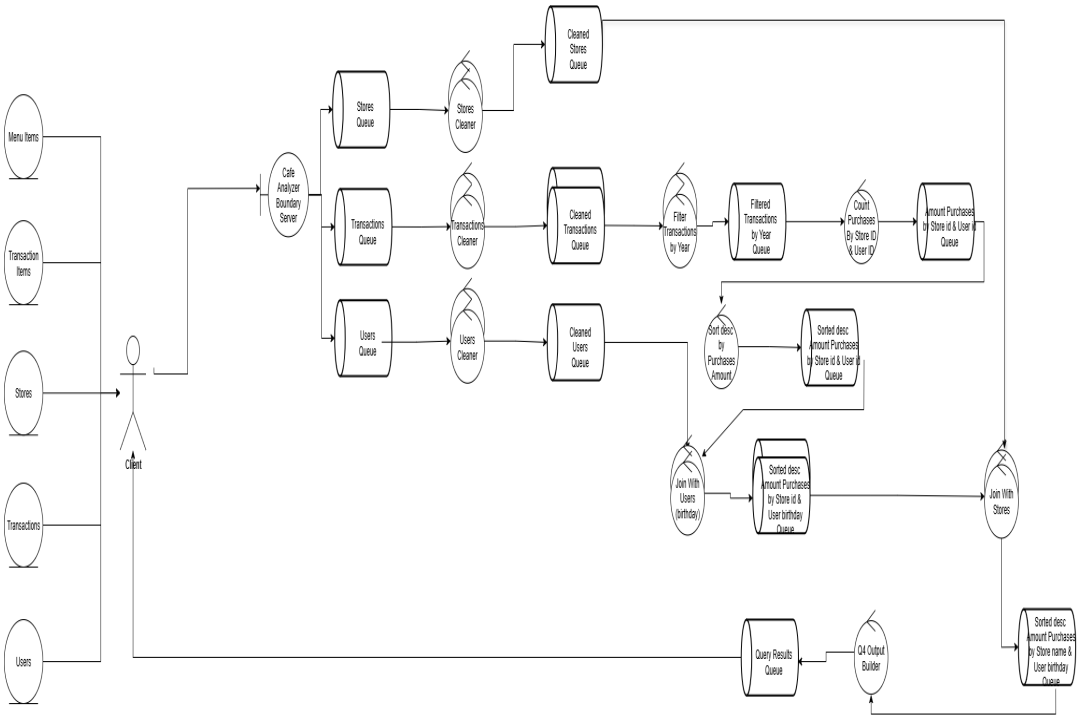
9.3. Diagrama de Robustez - Segunda Consulta



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.

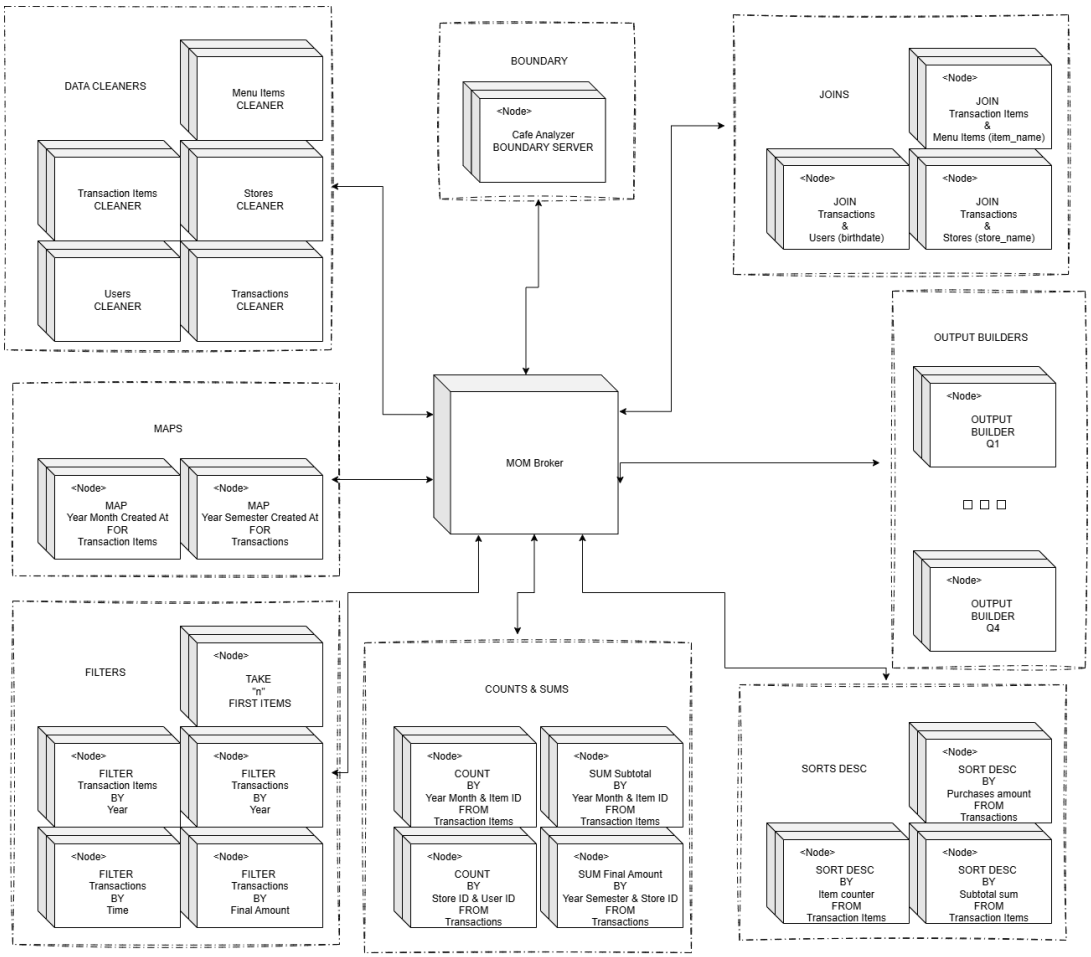


Figura 11: Diagrama de Despliegue

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.

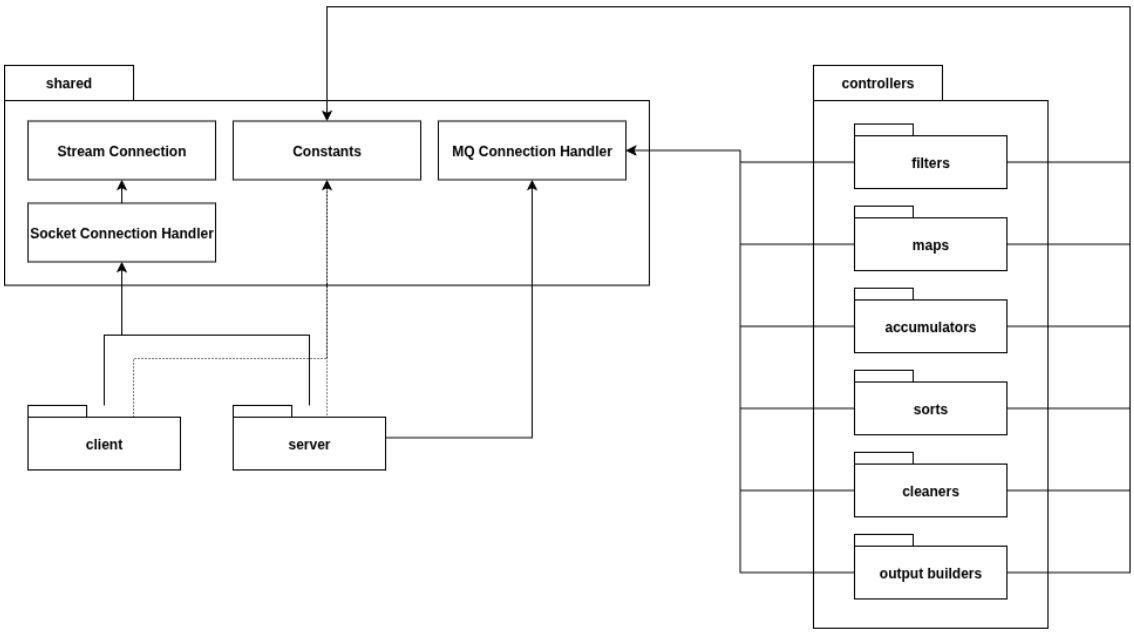


Figura 12: Vista de Desarrollo - Diagrama de Paquetes

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 - DataCleaner

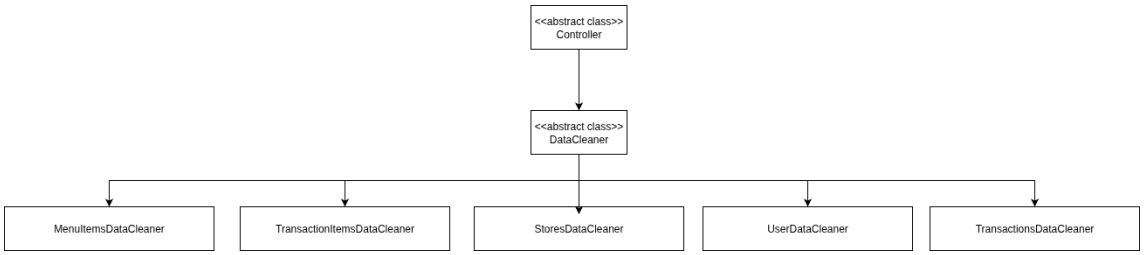


Figura 13: Clase DataCleaner

11.1.2. Diagrama de Clases - Filter

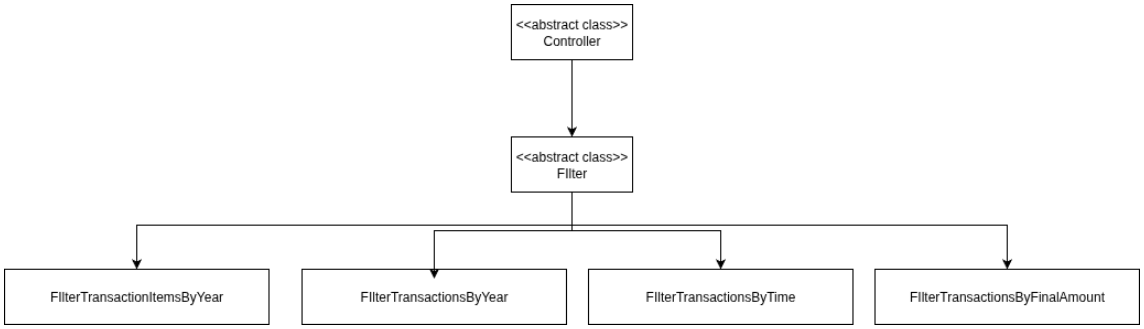


Figura 14: Clase Filter

11.1.3. Diagrama de Clases - Count y Sum

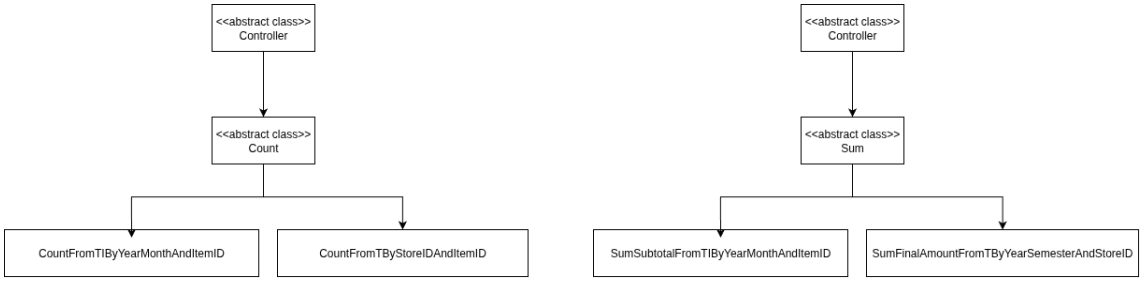


Figura 15: Clases Count y Sum

11.1.4. Diagrama de Clases - SortDesc

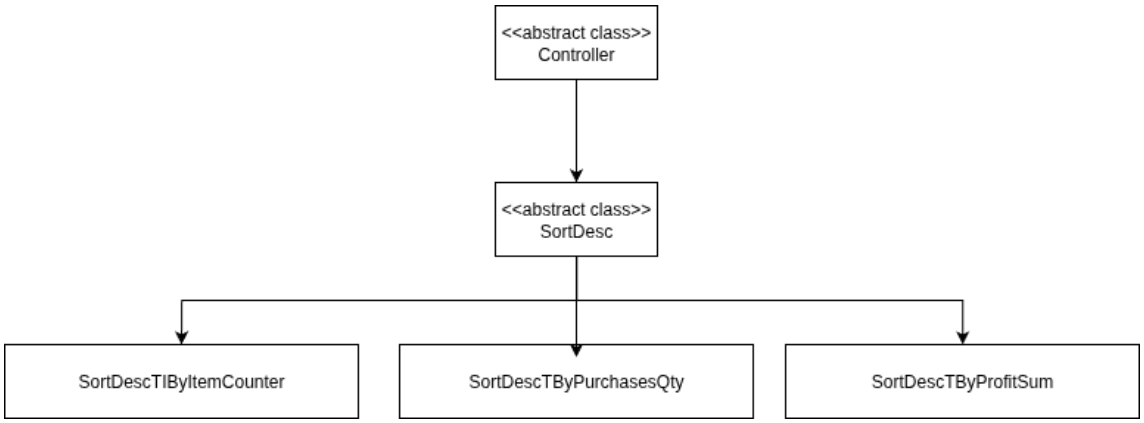


Figura 16: Clase SortDesc

11.1.5. Diagrama de Clases - Join

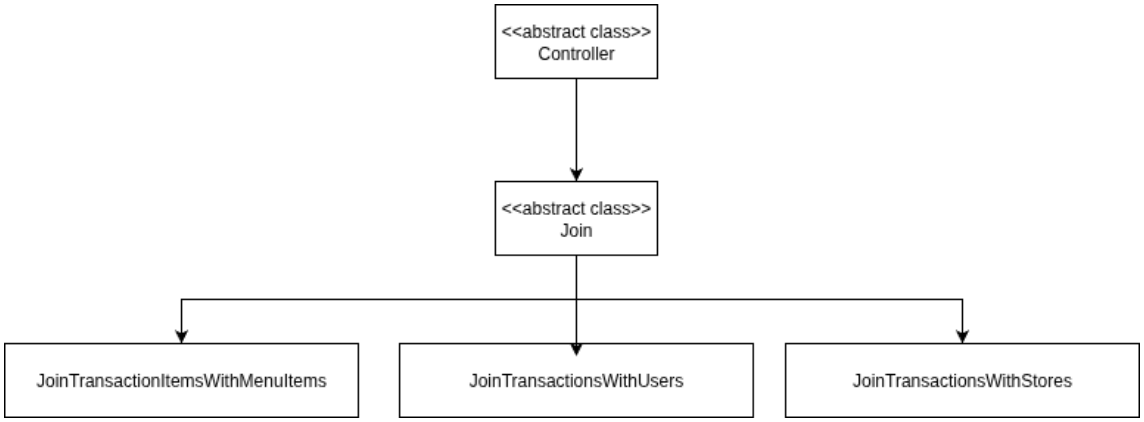


Figura 17: Clase Join

11.1.6. Diagrama de Clases - OutputBuilder

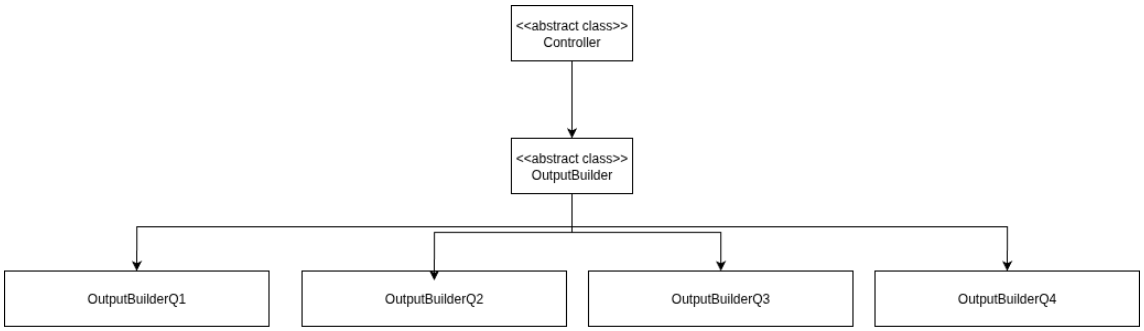


Figura 18: Clase OutputBuilder

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

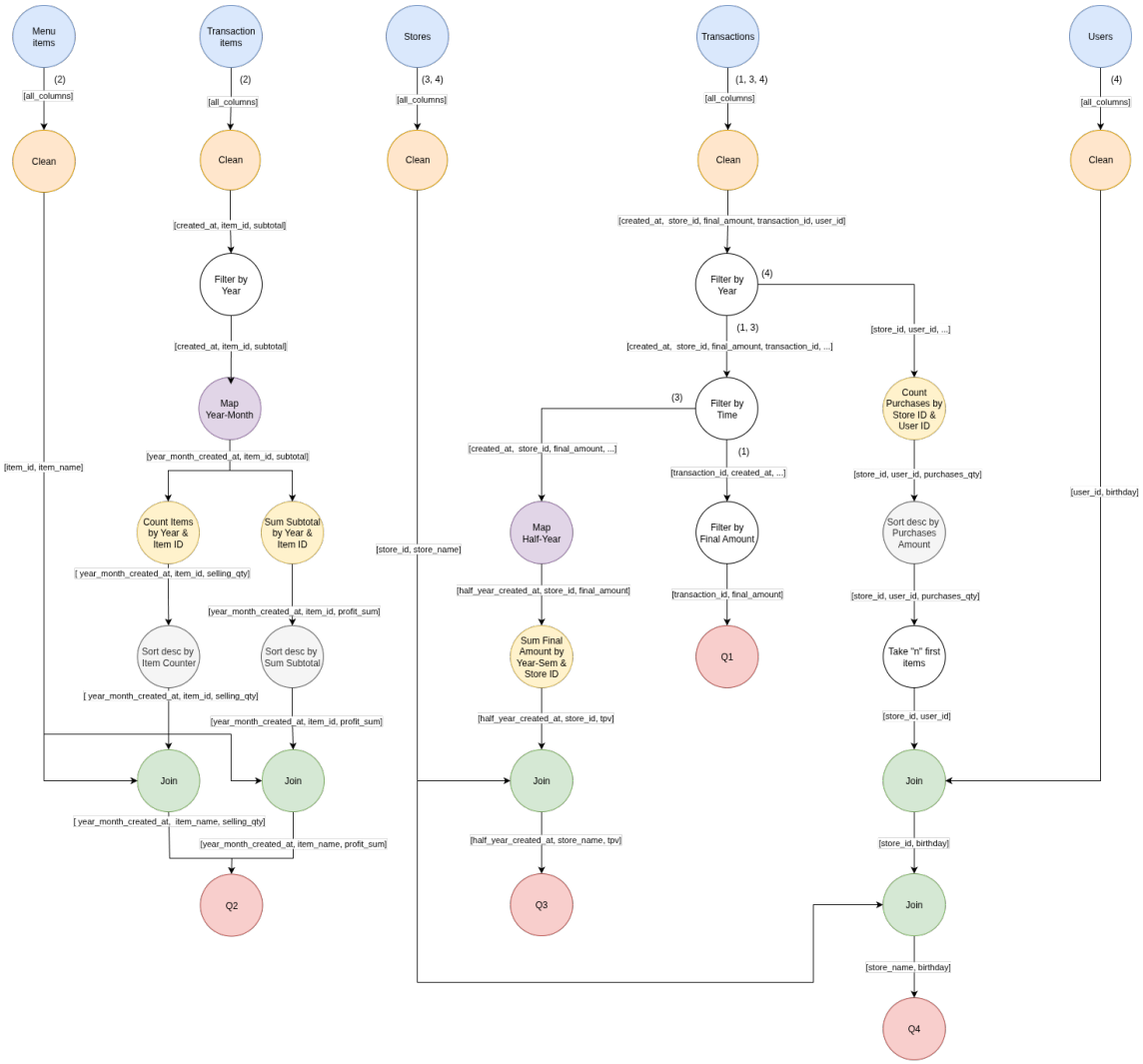


Figura 19: Diagrama de Grafo Acíclico Dirigido (DAG) - Completo

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

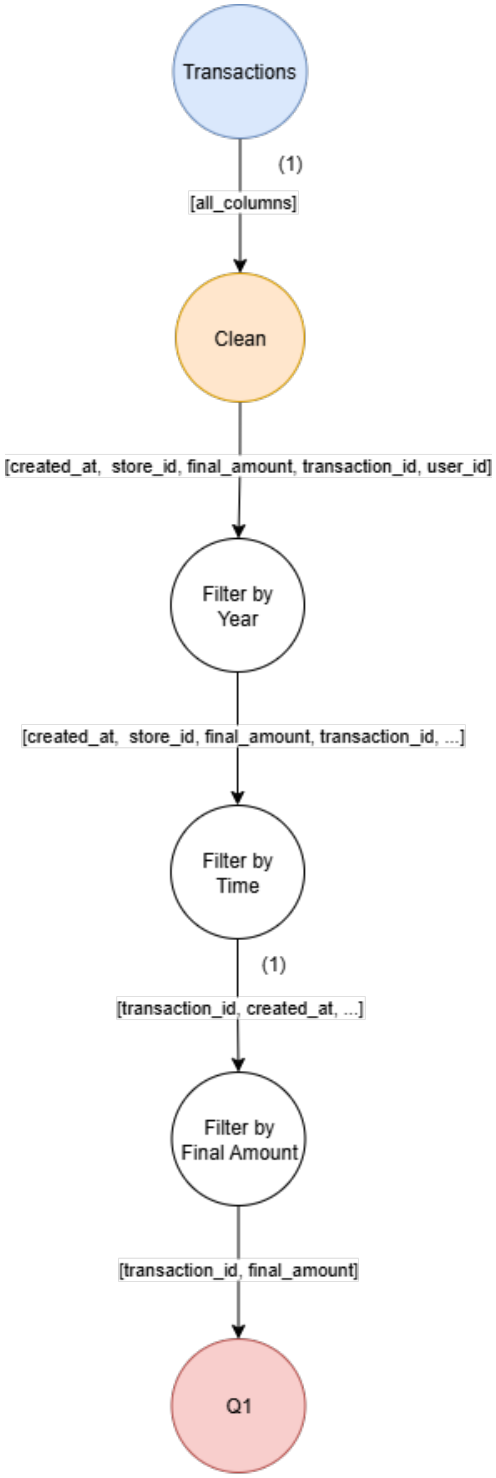


Figura 20: Diagrama de Grafo Acíclico Dirigido (DAG) - Primera consulta

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

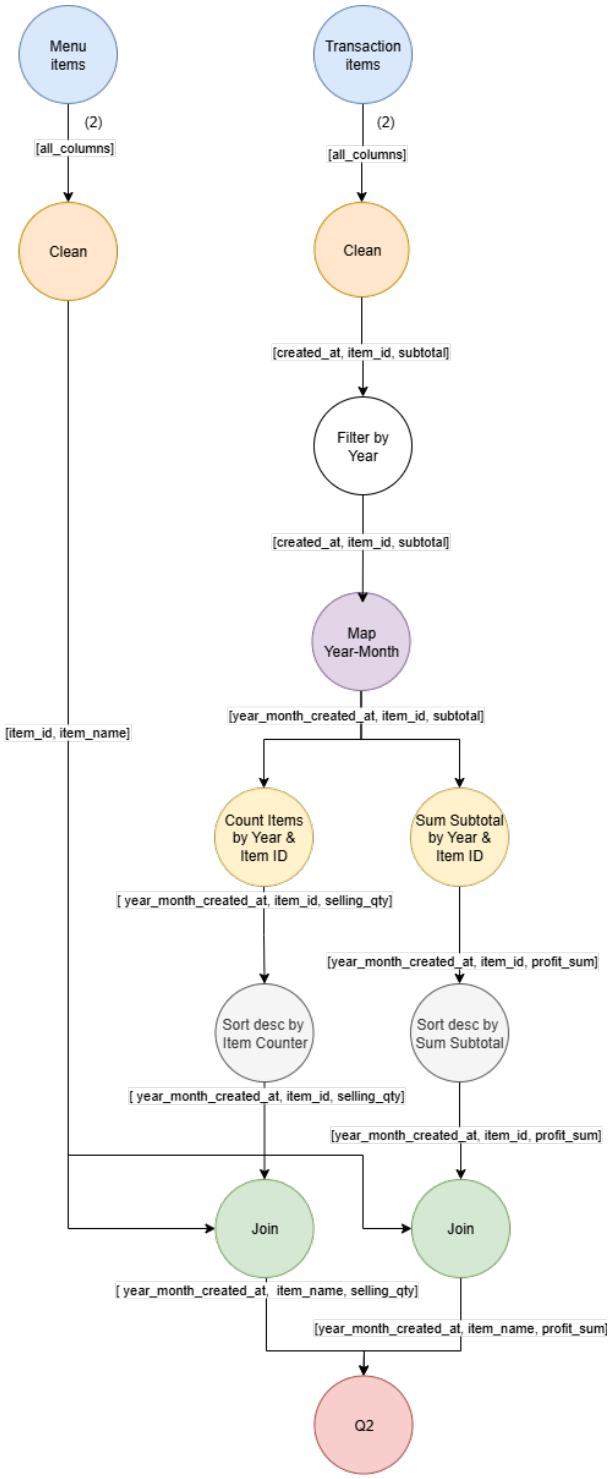


Figura 21: Diagrama de Grafo Acíclico Dirigido (DAG) - Segunda consulta

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

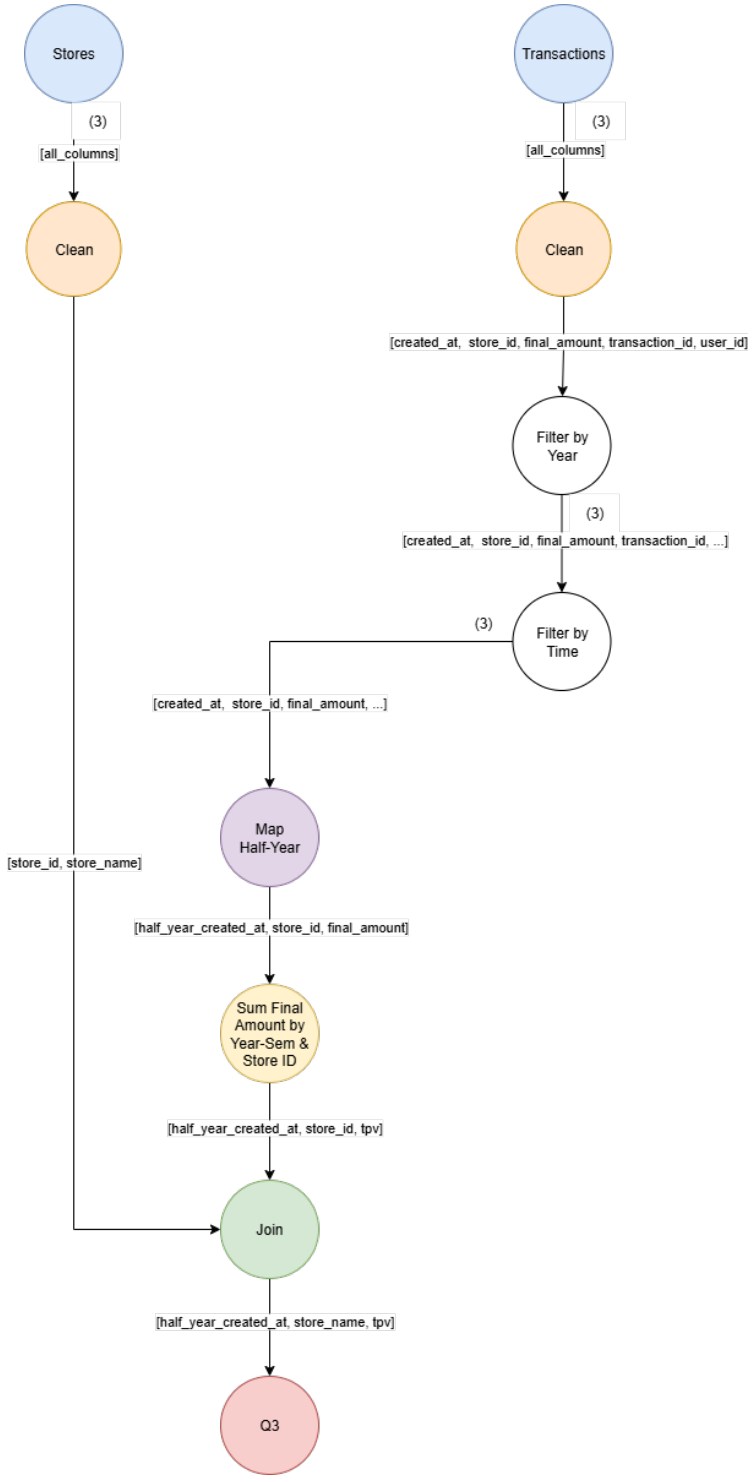


Figura 22: Diagrama de Grafo Acíclico Dirigido (DAG) - Tercera consulta

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

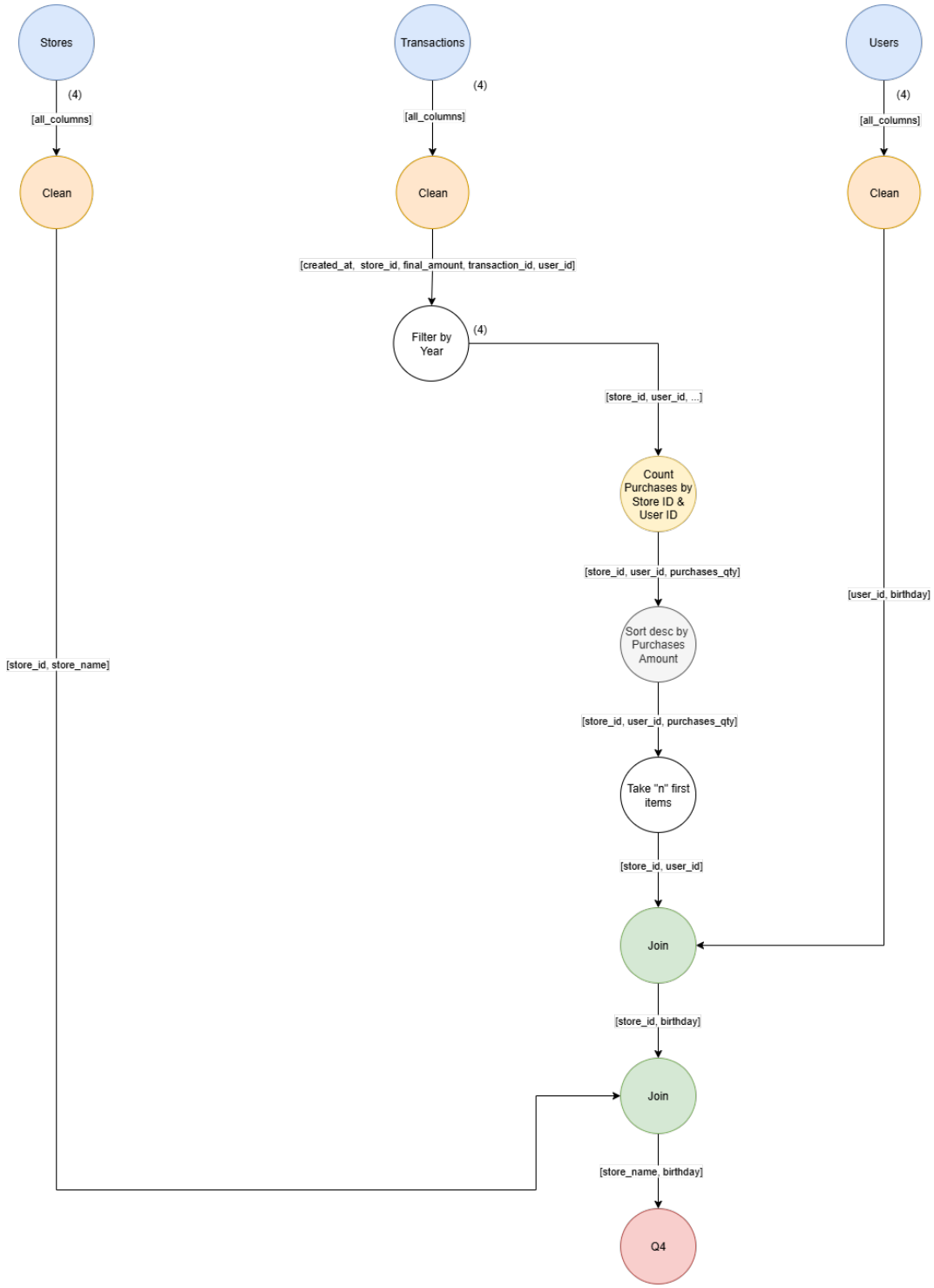


Figura 23: Diagrama de Grafo Acíclico Dirigido (DAG) - Cuarta consulta

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.

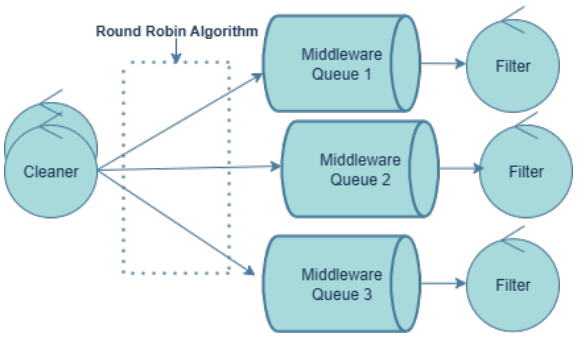


Figura 24: Esquema de comunicación mediante colas dedicadas.

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.

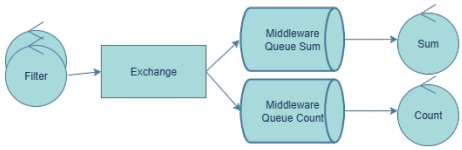


Figura 25: Esquema de comunicación mediante exchanges.

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.

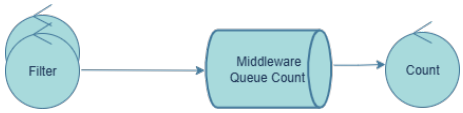


Figura 26: Esquema de comunicación de nodos escalables hacia un nodo único no escalable.

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 27: Esquema de comunicación 1 a 1.

Detección y propagación de EOF

Todos los controladores del sistema conocen tanto los nodos que les preceden como los que les suceden en la cadena de procesamiento. Esta información es fundamental para la correcta propagación de los mensajes de fin de flujo (`EOF`).

El mecanismo es el siguiente:

- Cada nodo espera recibir la cantidad exacta de mensajes `EOF` correspondientes a los emisores que tiene detrás suyo.
- Una vez que se han recibido todos los `EOF`, el nodo puede concluir que no habrá más información de entrada.
- En ese momento, genera y propaga su propio `EOF` hacia todos los nodos que le suceden, ya sea:
  - Enviando a una cola directa (1 a 1).
  - Replicando en un exchange.
  - Distribuyendo por Round Robin entre varias colas.

De esta manera, se garantiza que cada etapa del sistema procese exactamente todos los datos antes de finalizar, manteniendo un estado consistente y evitando pérdidas o confusiones en la comunicación.

Conclusión

La combinación de estas estrategias de comunicación permitió cumplir con los requisitos de abstracción y desacoplamiento planteados, al mismo tiempo que se aprovecharon al máximo las capacidades de RabbitMQ. Gracias a esta arquitectura, el sistema puede escalar horizontalmente de manera sencilla y mantener la coherencia en el flujo de datos, asegurando un comportamiento correcto incluso en escenarios de alta concurrencia.

14. Mensajes y Protocolos

14.1. Protocolo

Para la implementación del sistema distribuido se desarrolló un protocolo de comunicación específico, adaptado a las necesidades del presente trabajo.

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 *autocontenidos* con un encabezado de tipo fijo y un *payload* estructurado. Los mensajes expresan operaciones, datos a procesar y resultados obtenidos, y contemplan el envío por lotes (*batch*) de datasets, el intercambio de *ACKs* y señales de fin de flujo.

La implementación completa del protocolo se encuentra en: `src/shared/communication_protocol.py`.

14.2. Mensajes

Como parte del protocolo, se estableció un conjunto de tipos de mensajes de longitud fija (3 caracteres), orientados a cubrir las operaciones básicas del sistema: envío de datasets por lotes, resultados de queries, reconocimientos (`ACK`) y señalización de fin de flujo (`EOF`). Cada mensaje posee:

- Un **tipo** de 3 caracteres (p.ej., `MIT`, `TRN`, `ACK`).
- 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 un formato simple y siempre igual:

```
<TYPE>[<PAYLOAD>]
```

Donde:

- `<TYPE>` es un prefijo de **3 caracteres** (`MESSAGE_TYPE_LENGTH = 3`).
- 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 un *batch*:

```
{<ROW>;<ROW>;...;<ROW>}
```

Cada `<ROW>` es una secuencia de pares clave–valor separados por comas:

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

Un `<FIELD>` tiene la forma:

```
{'<key>':'<value>'}
```

Delimitadores usados en el batch:

- Inicio/fin de batch: `BATCH_START_DELIMITER={`, `BATCH_END_DELIMITER=}`
- Separador de filas: `BATCH_ROW_SEPARATOR=;`
- Separador de campos: `ROW_FIELD_SEPARATOR=,`

Ejemplo de batch con dos filas:

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

Sin escape de caracteres: Las claves y valores no pueden contener comillas dobles, dos puntos `:`, comas `,`, punto y coma `;`, ni llaves o corchetes.

14.4. Tipos de mensajes y su propósito

Tipo (3 chars)	Uso
`QRY`	Mensaje de consulta del cliente al sistema.
`ACK`	Reconocimiento genérico (éxito/recepción).
`MIT`	Lote de *menu items*.
`STR`	Lote de *stores*.
`TIT`	Lote de *transaction items*.
`TRN`	Lote de *transactions*.
`USR`	Lote de *users*.
`Q1X`	Resultado para Query 1 (*variant X*).
`Q21`	Resultado para Query 2.1.
`Q22`	Resultado para Query 2.2.
`Q3X`	Resultado para Query 3 (*variant X*).
`Q4X`	Resultado para Query 4 (*variant X*).
`EOF`	Señal de fin de flujo (payload: el tipo de flujo que finaliza).

Los tipos `MIT`, `STR`, `TIT`, `TRN` y `USR` son *batch messages*. Los tipos `Q1X`, `Q21`, `Q22`, `Q3X`, `Q4X` representan resultados de consultas y pueden usar el mismo formato general de mensaje (el payload queda a criterio del productor del resultado en este TP). `QRY` y `ACK` son mensajes de control ligeros (texto breve o vacío).

14.5. Codificación (*encode*)

Mensajes genéricos. La función privada `__encode_message(type, payload)` compone:

`type` ‖ `[` ‖ `payload` ‖ `]`

A partir de ella se exponen:

- `encode_ack_message(msg: str) ->str`: Genera `ACK[msg]`.
- `encode_eof_message(message_type: str) ->str`: Genera `EOF[<message_type>]`, donde el *payload* es el tipo de flujo que se declara finalizado (p.ej., `EOF[TRN]`).

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

1. Para cada `row: dict[str, str]`, codifica campos como `'key':'value'` unidos con `','`.
2. Une filas con `';'` y encapsula con `'{}'`.
3. Envuelve con el tipo y corchetes de mensaje.

Se proveen *helpers* tipados: `encode_menu_items_batch_message`,`encode_stores_batch_message`, `encode_transaction_items_batch_message`, `encode_transactions_batch_message`, `encode_users_batch_message`.

14.6. Decodificación (*decode*)

Acceso al payload y tipo.

- `decode_message_type(msg)`: Devuelve los primeros 3 caracteres. Valida largo mínimo; lanza `ValueError` si el mensaje es demasiado corto.
- `get_message_payload(msg)`: Remueve el prefijo de tipo y `[ ]` exteriores, retornando la cadena interna.
- `decode_is_empty_message(msg)`: Verdadero si el payload está vacío.

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

1. Que el tipo decodificado sea el esperado.
2. Que el mensaje comience con `<expected>[` y termine con `]`.

En caso contrario, lanza `ValueError`.

Decodificación de lotes. `decode_batch_message(msg)`:

1. Obtiene el payload (que globalmente está entre {}).
2. Separa filas por `;`.
3. Para cada fila, `__decode_row`:
   a) Remueve llaves {} en extremos (si las hubiera).
   b) Separa campos por `,`.
   c) Divide cada campo por el primer `:` en `key:value`.
   d) Quita comillas dobles exteriores de clave y valor.
4. Devuelve `list[dict[str,str]]`.

Se incluyen variantes tipadas que además validan el tipo del mensaje: `decode_menu_items_batch_message`, `decode_stores_batch_message`, `decode_transaction_items_batch_message`, `decode_transactions_batch_message`, `decode_users_batch_message`.

Señal de fin de flujo. `decode_eof_message(msg)` valida que el tipo sea `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

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

1. El productor envía uno o más mensajes **batch** del tipo correspondiente.
2. Opcionalmente, el receptor responde con `ACK[...]` para marcar recepción/éxito.
3. Al finalizar el stream de ese dataset, el productor envía `EOF[<TIPO>]`, por ejemplo `EOF[TRN]`.

Consultas y resultados.

1. Un cliente emite `QRY[...]` con parámetros de consulta (formato libre en este TP).
2. Los resultados se retornan en mensajes `Q1X`, `Q21`, `Q22`, `Q3X` o `Q4X`, según el ejercicio, con payload acorde.
3. Se puede usar `ACK[...]` para confirmar recepción o estado (p.ej., *accepted*, *done*).

14.8. Ejemplos de mensajes codificados

ACK sin payload.

```
ACK[]
```

Batch de menu items (2 filas).

```
MIT[{'id':'m001','name':'Latte','price':'4.50';'id':'m002','name':'Espresso','price':'3.20'}]
```

Fin de flujo de transacciones.

```
EOF[TRN]
```

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

```
Q21[{'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 y que existan los corchetes exteriores. Inconsistencias lanzan `ValueError`.
- **Longitud mínima:** `decode_message_type` exige al menos 3 caracteres (tipo). Mensajes más cortos disparan `ValueError`.
- **Payload vacío:** Admitido (`ACK[]` o keep-alives).
- **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 `;`, ni llaves/corchetes. 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* (batches, ACK, EOF). 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 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 define un **formato compacto y determinista**:

- encabezado de 3 caracteres para el tipo
- delimitadores exteriores `[ ]`
- para datasets, un **batch** con `{}` y separadores simples

Se proveen *helpers* específicos para codificar/decodificar los distintos datasets y una señal `EOF` que marca el fin de cada flujo lógico. Las validaciones estructurales minimizan errores de parseo y el diseño favorece procesamiento lineal 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 a lo largo de todo el desarrollo.

A continuación, se detallan los resultados obtenidos al finalizar el mismo, 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 múltiples nodos de cómputo.

Todos los nodos escalables, fueron configurados para levantar 2 nodos de cómputo cada uno.

Respecto a los nodos no escalables, solo se levantó un nodo de cómputo por cada uno (Es decir, para esta entrega no se implementó redundancia en los nodos no escalables, pero no se descarta para futuras versiones del desarrollo).

15.2. Tiempo de procesamiento total

Para cumplir con los requisitos brindados por la cátedra, donde se pedia que el sistema tarde menos de una hora en realizar todo el procesamiento de las 4 consultas, con el dataset completo, se realizó la medición del tiempo total de procesamiento al finalizar el desarrollo completo de la solución.

El tiempo total de procesamiento medido fue de exactamente ’29’ minutos con ’32’ segundos, cumpliendo así con el requisito establecido.

Con esto, se puede concluir que el sistema implementado es capaz de manejar eficientemente el procesamiento de las consultas dentro del tiempo límite especificado, demostrando su efectividad y capacidad para trabajar con grandes volúmenes de datos de manera distribuida.

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. Cronograma teórico del desarrollo

17.1. 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.

17.2. 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.

17.3. 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.

17.4. 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.

17.5. 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.

18. Cronograma real del desarrollo (2 semanas)

18.1. Diseño

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.

18.2. Middleware y Coordinación de Procesos (2.5 semanas)

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.

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.