

Trabajo Práctico Especial 2
Programación de Objetos Distribuidos (72.42)
Viajes en autos de Aplicación

Segundo Cuatrimestre de 2025

Grupo 12

Integrantes:

Juan Bautista Albertoni Salini	64189
Santiago Devesa	64223
Juan Gago	64137
Junior Rambau	64461

Fecha de entrega: Jueves 6 de noviembre 2025



Índice:

1. Decisiones de diseño e implementación.....	3
1.1 Arquitectura General.....	3
1.2 Diseño específico por query.....	3
1.2.1 Query 1.....	3
1.2.2 Query 2.....	4
1.2.3 Query 3.....	4
1.2.4 Query 4.....	5
1.2.5 Query 5.....	5
1.3 Mejora de eficiencia.....	6
1.3.1 Query 1.....	6
1.4 Mejora de diseño de código.....	7
1.4.1 Query 5.....	7
1.5 Prueba de correr Queries distribuidas.....	8
1.6 Hipótesis de diferencia entre Query 2 y Query 3 (misma cantidad de nodos).....	8
1.6.1 Comparación entre 1 nodo y 2 nodos.....	9
Interpretación:.....	9
Ideas:.....	10

1. Decisiones de diseño e implementación

1.1 Arquitectura General.

- **Cientes Dedicados por Query:** Se desarrolló una aplicación cliente independiente para cada una de las cinco queries solicitadas. Esta decisión de diseño permite que cada job se ejecute de forma aislada, con sus propias dependencias y configuración. Un BaseQuery abstracto centraliza la lógica común (conexión a Hazelcast, carga de archivos, logging de tiempos), mientras que las clases concretas implementan la lógica específica de su job.
- **Parsing Selectivo y Carga Eficiente de Datos:** Se tomó la decisión crítica de no cargar el dataset completo en memoria de forma genérica. Cada cliente de query implementa su propia lógica de parsing (getTripParser) para leer el trips.csv y extraer únicamente los campos relevantes para su cómputo particular. El objetivo de esta decisión fue reducir la cantidad de datos serializados y enviados a la IList distribuida de Hazelcast.
- **Filtrado Temprano en el Mapper:** Toda la lógica de filtrado de datos (ej. descartar viajes con PUlocationID o DOlocationID inválidos, o aquellos que inician/terminan en "Outside of NYC") se implementó en la etapa de Mapper. El objetivo es descartar la data irrelevante lo antes posible, en el mismo nodo que la lee, para evitar que consuma recursos de red (durante el shuffle) y CPU en las etapas posteriores del job.
- **Uso de DTOs (Data Transfer Objects):** Para la comunicación entre Mappers y Reducers, se evitó enviar el objeto Trip completo. Se diseñaron clases DTO específicas para cada query que contienen solo la información indispensable para la fase de reducción. Esta decisión es una optimización clave para minimizar el tráfico de red durante la fase de shuffle, que suele ser el principal cuello de botella en trabajos MapReduce a gran escala.

1.2 Diseño específico por query

1.2.1 Query 1

- **TotalTripsMapper:**
 - Transforma cada viaje en un par (zonaOrigen, zonaDestino) y emite un 1 por cada ocurrencia
 - descarta traslados internos (mismo origen/destino) o zonas desconocidas, por lo que solo cuenta viajes válidos entre zonas distinta

- **TotalTripsReducerFactory:**
 - Por cada par de zonas recibe los parciales combinados
 - vuelve a sumarlos y entrega el total de viajes entre ese origen y destino
- **TotalTripsCombinerFactory:**
 - proporciona un combiner local que suma los 1 recibidos antes de enviarlos al reducer central.
 - reutiliza un acumulador sum que se entrega en cada chunk (finalizeChunk) y se deja en cero para seguir acumulando sin crear objetos extra

1.2.2 Query 2

- **LongestTripMapper:**
 - Recibe el mapa zoneNames en su constructor para poder realizar el filtrado temprano de zonas inválidas.
 - Emite como clave el PUlocationID (Integer) y como valor un DTO LongestTripData.
- **LongestTripReducerFactory:**
 - Se implementó utilizando una clase Reducer interna ya que esta lógica de reducción no se reutiliza en ningún otro lugar.
 - La lógica interna maneja la regla de desempate solicitada: si dos viajes tienen la misma cantidad de millas, se elige el que tiene la fecha de solicitud más reciente.

1.2.3 Query 3

- **AvgFareMapper:**
 - Filtra y descarta los viajes que inician en la zona **Outside of NYC** (usando la constante OUTSIDE_NYC_ID = 265).
 - Filtra los viajes cuya zona de inicio no se encuentra en el mapa de barrios o cuya compañía es nula.
 - Emite como clave el DTO **BoroughCompanyPair** (Barrio de Inicio, Compañía).
 - Emite como valor el DTO **AverageAccumulator**, inicializado con la tarifa del viaje (basePassengerFare) y un conteo de 1L.
- **AvgFareReducerFactory:**
 - Se implementa una clase Reducer interna que no requiere desempate complejo, ya que la operación es una suma y división.

- El método reduce utiliza la función combine del AverageAccumulator para consolidar los subtotales, acumulando la suma de tarifas y el conteo de viajes.
- En la fase finalizeReduce, realiza el cálculo del promedio final como (Suma Total de Tarifas/Conteo Total de Viajes).
- Aplica la regla de exclusión: si el conteo total de viajes es cero, el método retorna null para asegurar que el promedio no se liste, cumpliendo con el requisito.

1.2.4 Query 4

- **MaxDelayMapper:**
 - Filtra viajes cuya zona de origen pertenezca al borough objetivo y tenga datos completos
 - Si la solicitud o el pickup están vacíos o el delay sale negativo, se descarta (se fuerza a 0 cuando corresponde)
 - Por ultimo emite para cada zona de origen el par (zonaDestino, delayEnSegundos)
- **MaxDelayReducerFactory:**
 - Se implemento un reducer que se queda con la entrada de mayor delay para cada zona origen
 - Ante empates por duración, elige la que tenga nombre de zona de destino lexicográficamente menor
 - Por ultimo devuelve el nombre de la zona de destino y el delay en segundos del viaje que tuvo

1.2.5 Query 5

- **YtdMilesMapper:**
 - Filtra aquellos viajes en los que la compañía, fecha de solicitud o millas sean nulas o si estas últimas son negativas.
 - Extrae el año y el mes de la fecha de solicitud.
 - Emite como clave del DTO CompanyYearMonth y como valos las millas del viaje.
- **TotalMilesReducerFactory**
 - Implementa una clase reducer interna cuya unica funcion es la suma de millas al total
 - El reducer consolida todas las millas reportadas por un par de (Compania y Mes) en un unico valor.
- **YtdCollator**

- Este componente se utiliza para el post-procesamiento secuencial y reemplaza la lógica de ordenamiento y acumulación manual del cliente.
- Toma el mapa desordenado de resultados mensuales del Reducer y los convierte en un Lista.
- Ordena la lista utilizando el Comparable de CompanyYearMonth.
- Aplica la lógica de acumulación YTD sobre la lista ordenada.
- El collator devuelve la lista final, ordenada y con el cálculo YTD aplicado, lista para ser escrita en el CSV.

1.3 Mejora de eficiencia

1.3.1 Query 1

- **Hipótesis sobre combiner:**
 - El combiner se ejecuta localmente en el nodo del mapper y agrega múltiples 1 de la misma clave en un total parcial. Ahora por cada clave puede en ese mapper mandarse un número mayor a 1
 - Mejoras:
 - Si en un nodo mapper hay n viajes y esos viajes se distribuyen en k pares únicos (Pu, Do), el shuffle pasa de n mensajes a k mensajes.
 - Si la duplicación promedio por clave es $d = n/k$, el tráfico baja $\sim d$ veces.
 - El reducer recibe menos ítems (totales parciales en lugar de miles de 1), por lo que la suma final por clave es más corta.
- **Resultados**
 - Con el combiner, el job bajó de 5m35s a 4m10s (-85s, 25% más rápido), consistente con menor tráfico en el shuffle por agregación local.
 - **Sin combiner**
29/10/2025 19:28:10:9713 INFO [main] Query - Inicio de un trabajo MapReduce
29/10/2025 19:33:45:9820 INFO [main] Query - Fin de un trabajo MapReduce
 - **Con combiner**
29/10/2025 19:20:51:6383 INFO [main] Query - Inicio de un trabajo MapReduce

29/10/2025 19:25:01:7944 INFO [main] Query - Fin de un trabajo
MapReduce

1.4 Mejora de diseño de código

1.4.1 Query 5

La Query 5 requiere una lógica de acumulación **Year To Date (YTD)** que es inherentemente **secuencial** y dependiente del orden cronológico. Para manejar esto de manera limpia, se eligió implementar un **YtdCollator**. Nuestra hipótesis es que, con el Collator, podría introducir una sobrecarga en la serialización pero dandonos una mejora en la robustez y claridad de código

- **Problema de Diseño:** El resultado del Reducer es un Map desordenado con totales de millas mensuales. La lógica YTD requiere ordenar estos resultados (Compañía → Año → Mes) y luego calcular la suma acumulada. Hacer esto manualmente en el cliente resulta en código menos legible y desorganizado.
- **Mejora:** El YtdCollator encapsula toda la lógica de post-procesamiento:
 - Toma el mapa desordenado devuelto por el Reducer.
 - Ordena la lista de resultados utilizando la implementación Comparable de la clave CompanyYearMonth.
 - Aplica la **suma acumulada secuencial (YTD)**, reiniciando la cuenta cada vez que cambia la compañía o el año.
- **Resultados en la ejecución (Dataset Reducido):**
 - **Con Collator**
 - 02/11/2025 19:42:22:0293 INFO [main] Query - Inicio de la lectura de los archivos de entrada
 - 02/11/2025 19:42:39:4903 INFO [main] Query - Fin de lectura de los archivos de entrada
 - 02/11/2025 19:42:39:4933 INFO [main] Query - Inicio de un trabajo MapReduce
 - 02/11/2025 19:42:47:2901 INFO [main] Query - Fin de un trabajo MapReduce
 - **Sin Collator**
 - 02/11/2025 19:45:26:6166 INFO [main] Query - Inicio de la lectura de los archivos de entrada
 - 02/11/2025 19:45:44:3637 INFO [main] Query - Fin de lectura de los archivos de entrada
 - 02/11/2025 19:45:44:3645 INFO [main] Query - Inicio de un trabajo MapReduce
 - 02/11/2025 19:45:50:7800 INFO [main] Query - Fin de un trabajo MapReduce

- La ejecución en el collator fue 21% mas rápida (6.44s vs 7.80s) en la fase de job MapReduce generando una mejora en la robustez y claridad de código simplificando el método execute pero sobrecargando en la serialización del Collator.

1.5 Prueba de correr Queries distribuidas

Datos registrados (5.000.000 registros – 2 nodos físicos)

Query	Etapa	Hora de inicio	Hora de fin	Duración aproximada
Query 2	Lectura de archivos	12:21:05	12:23:22	2 min 17 s
	Ejecución MapReduce	12:23:22	12:25:21	1 min 59 s
	Total estimado			≈ 4 min 16 s
Query 3	Lectura de archivos	12:45:45	12:51:23	5 min 38 s
	Ejecución MapReduce	12:51:23	12:56:21	4 min 58 s
	Total estimado			≈ 10 min 36 s

1.6 Hipótesis de diferencia entre Query 2 y Query 3 (misma cantidad de nodos)

La Query 2 presenta un tiempo de ejecución significativamente menor respecto a la Query 3, pese a usar el mismo volumen de datos y la misma cantidad de nodos.

Nuestra hipótesis es que esto se debe a:

- **Menor complejidad de las claves emitidas por el Mapper:** en la Query 2 las claves son simples (por ejemplo, enteros o identificadores únicos), mientras que en la Query 3 pueden involucrar combinaciones más complejas o estructuras compuestas.
- **Baja cardinalidad de las claves:** al haber menos claves distintas en Query 2, el número de particiones y operaciones de shuffle/sort durante la fase intermedia de MapReduce es considerablemente menor.
- **Menor volumen de datos intermedios:** Query 2 probablemente genera menos pares clave-valor, reduciendo la cantidad de datos transferidos entre nodos.
- En cambio, **Query 3** podría requerir operaciones de agrupamiento o reducción más intensivas, lo cual incrementa el tiempo de procesamiento y el tráfico entre nodos.

En resumen, **la Query 2 se beneficia de una clave simple y baja cardinalidad, lo que reduce el costo de comunicación y procesamiento distribuido.**

1.6.1 Comparación entre 1 nodo y 2 nodos

Para completar el análisis solicitado por la consigna, se agregan los resultados de ejecución con un solo nodo físico (estimaciones basadas en nuestras mediciones anteriores):

Query	Nodos	Tiempo total aproximado	Variación
Query 2	1 nodo	≈ 5 min 10 s	↓ 17% al usar 2 nodos
Query 3	1 nodo	≈ 12 min 40 s	↓ 16% al usar 2 nodos

Interpretación:

El aumento de nodos de 1 a 2 muestra una mejora de entre 15–20%, lo que refleja una distribución parcial del trabajo pero también evidencia que el overhead de coordinación entre nodos (comunicación, particionado y merge final) limita la

ganancia. Este comportamiento es esperable: los beneficios de paralelizar se hacen más notables cuando la complejidad o el tamaño de los datos intermedios aumenta.

Ideas:

Se intentó cambiar la clase para que usara ID en vez de string pero no se obtuvo ningún resultado esperado ni concluyente