# 72.42 - Programación de Objetos Distribuidos

## Trabajo Práctico N° 2

## “Viajes en autos de aplicación en Nueva York”

Grupo 5


### Profesores:

Roman Meola, Franco

Turrin, Marcelo

### Integrantes del grupo:

| | | |
|---|---|---|
| 64396 | Menshikoff, Katia | kmenshikoff@itba.edu.ar |
| 62108 | Peral Belmont, Javier Bautista | jperalbelmont@itba.edu.ar |
| 63167 | Pinausig Castillo, Tomas | tpinausigcastillo@itba.edu.ar |
| 64047 | Squillari, María Agostina | msquillari@itba.edu.ar |

# Índice

# 1. Decisiones de diseño e implementación de los servicios

## Diseño de los componentes

El sistema desarrollado implementa un esquema de procesamiento distribuido basado en el modelo MapReduce, utilizando el framework Hazelcast 3.8.6. La solución se estructura en tres módulos principales: API, Server y Client, siguiendo una arquitectura modular y extensible.

- Módulo API: Contiene las clases compartidas que definen las abstracciones comunes necesarias para la ejecución de los jobs, incluyendo las interfaces y clases base de Mapper, Combiner, Reducer y Collator.

- Módulo Server: Representa los nodos del cluster distribuido encargados de ejecutar las tareas de procesamiento y almacenar los datos en estructuras distribuidas de Hazelcast.

- Módulo Client: Se encarga de la carga inicial de datos y la ejecución de las distintas consultas MapReduce sobre el conjunto distribuido.

El diseño favorece la separación de responsabilidades y el desacoplamiento entre componentes, permitiendo agregar nuevas queries o estrategias de procesamiento sin modificar la estructura base. Esta modularidad asegura escalabilidad, facilidad de mantenimiento y reutilización del código.

Para optimizar el proceso de carga de los datasets desde archivos CSV, se implementó una estrategia de batching, que agrupa los registros en lotes de 1000 elementos antes de insertarlos en el mapa distribuido mediante la operación putAll().

Esta técnica reduce significativamente el número de llamadas remotas y el tráfico de red, mejorando la eficiencia de la inicialización del conjunto de datos.

Además, se empleó un HashMap pre-dimensionado para almacenar temporalmente los registros antes de cada inserción, evitando operaciones de rehashing y contribuyendo a un mejor aprovechamiento de la memoria y del tiempo de ejecución.

Con el objetivo de optimizar el uso de memoria, se reemplazaron estructuras de gran tamaño por representaciones compactas. Por ejemplo, para la Query 1, se creó la clase MinimalTrip, que almacena únicamente dos valores short (zona de origen y destino), ocupando apenas 4 bytes por registro, en comparación con los aproximadamente 200 bytes de la estructura original. Esta optimización representó una reducción del 96% en el tamaño de los objetos almacenados.

Cada query fue implementada a partir de una combinación particular de Mapper, Combiner, Reducer y Collator, seleccionados y diseñados para los objetivos analíticos de cada consulta.

El flujo general de ejecución de las consultas replica el patrón propuesto por el modelo MapReduce de Hazelcast, garantizando procesamiento paralelo, data locality y tolerancia a fallos dentro del cluster.

A continuación, se detalla el diseño y la lógica de ejecución de cada una de las cinco consultas implementadas mediante el modelo MapReduce.

## Query 1: Total de viajes por zona

El objetivo de esta consulta es contabilizar el número total de viajes agrupados por zona de origen y zona de destino.

El Mapper emite pares clave-valor del tipo (TotalKeyOut(pickUpZone, dropOffZone), 1L), donde cada par representa un viaje entre ambas zonas.
Tanto el Combiner como el Reducer suman los valores asociados a cada clave, obteniendo así el total de viajes por combinación de zonas.
Finalmente, el Collator organiza y formatea los resultados para su escritura en el archivo CSV de salida.

Durante el proceso de carga de datos se incorporó un filtro de preprocesamiento que descarta los viajes cuyo origen y destino coinciden, reduciendo el número de registros a procesar y optimizando el rendimiento global de la consulta.

## Query 2: Viaje más largo por zona

Esta consulta busca identificar el viaje más largo, en millas, correspondiente a cada zona de inicio.

El Mapper emite pares (pickUpZone, LongestTrip) que encapsulan la distancia recorrida y la fecha de solicitud del viaje.
El Combiner conserva localmente el viaje de mayor distancia por zona, aplicando un criterio de desempate por fecha más reciente en caso de igualdad de distancias.
El Reducer aplica la misma lógica de comparación para determinar el máximo global por zona.
El Collator se encarga de estructurar los resultados finales según el formato requerido.

El uso del Combiner permitió reducir significativamente el volumen de datos transmitidos entre nodos, pasando de un orden de complejidad O(n) a O(cantidad de zonas), con un impacto directo en la disminución del tráfico de red y los tiempos de ejecución.

## Query 3: Precio promedio por borough y compañía

El objetivo de esta query es calcular el precio promedio de los viajes agrupado por barrio (borough) y compañía.

El Mapper emite pares del tipo (AverageKeyOut(borough, company), basePassengerFare).

El Combiner acumula localmente la suma de tarifas y el conteo de viajes, y el Reducer realiza la agregación final de estos acumuladores, calculando el promedio correspondiente a cada par (borough, company).

El Collator ordena los resultados de acuerdo con los criterios del enunciado: de forma descendente por precio promedio y alfabéticamente por borough y compañía.

Se implementó un patrón de acumulador distribuido, garantizando precisión numérica y consistencia en los resultados, incluso en escenarios de alta concurrencia y paralelismo.

## Query 4: Mayor demora por zona (con filtro de borough)

Esta consulta calcula el mayor tiempo de espera (en segundos) entre la solicitud y el inicio del viaje, agrupado por zona de origen, dentro de un borough específico indicado por parámetro.

El Combiner determina localmente el valor máximo de demora, aplicando desempate alfabético por la zona de destino.

El Reducer obtiene el máximo global combinando los resultados parciales.

El Collator convierte los tiempos a segundos y genera la salida final con el formato requerido.

Se introdujo una optimización relevante en la etapa de carga: el filtrado por borough se realiza antes de insertar los datos en Hazelcast, reduciendo la cantidad de registros procesados en el cluster. Para ello, se precalcularon los LocationID pertenecientes al borough especificado y se descartaron los viajes no pertinentes durante el parseo del archivo CSV.

## Query 5: Millas YTD por compañía

El propósito de esta query es calcular las millas acumuladas Year-To-Date (YTD) por compañía, año y mes.

El Mapper analiza la fecha de solicitud del viaje y emite pares (CompanyYearMonth, miles).

Tanto el Combiner como el Reducer suman las millas recorridas local y globalmente.

El Collator se encarga de computar los acumulados YTD, ordenando los resultados cronológicamente por compañía, año y mes.

La clave CompanyYearMonth fue diseñada para implementar las interfaces DataSerializable y Comparable, optimizando la serialización de objetos durante la transmisión y facilitando el ordenamiento natural de los resultados en la etapa final.

**Consideraciones generales sobre combiners y patrones de diseño**

Todas las queries implementan combiners con el fin de reducir el tráfico de red, disminuir la carga sobre los reducers y optimizar el aprovechamiento del paralelismo local.

Asimismo, se aplicaron patrones de diseño como *Factory*, *Strategy* y *Template Method*, utilizados respectivamente para la creación dinámica de instancias, el parseo configurable por query y la reutilización de flujos comunes entre distintos procesos.

Finalmente, se priorizó el uso de DataSerializable por sobre Serializable, debido a su mayor eficiencia y control mediante los métodos writeData() y readData(). Esta decisión permitió minimizar la reflexión y reducir la sobrecarga durante la transferencia de objetos entre nodos, mejorando el rendimiento general del sistema distribuido.

## 2. Análisis de tiempos de ejecución

Los tiempos calculados a continuación se realizaron utilizando el archivo proporcionado por la cátedra que contiene alrededor de 20 millones de líneas. Todas las mediciones dentro de la tabla se realizaron luego de implementar la optimización del código.

| Query Strategy | Tiempo de carga de datos (s) | Tiempo de ejecución de la query (s) | Cantidad de nodos | Uso de Combiner |
|---|---|---|---|---|
| 1 | ~21,73 | ~346,003 | 3 | ✅ |
| 1 | ~18,04 | ~379,014 | 3 | ✅ |
| 1 | ~17,53 | ~403,793 | 3 | ✅ |
| 1 | ~20,147 | ~421,84 | 3 | ❌ |
| 1 | ~18,386 | ~396,128 | 3 | ❌ |

| | | | | |
|---|---|---|---|---|
| **1** | **~16,766** | **~378,644** | **3** | ❌ |
| **Nodos físicos con CSV de 1M líneas (trips.csv)** | | | | |
| **1** | **~19,021** | **~203,792** | **2** | ✅ |
| **1** | **~14,651** | **~121,594** | **3** | ✅ |
| **1** | **~2,041** | **~16,876** | **1** | ✅ |
| **2** | **~24,196** | **~10,305** | **3** | ✅ |
| **2** | **~23,955** | **~12,141** | **2** | ✅ |
| **2** | **~2,673** | **~0,744** | **1** | ✅ |
| **3** | **~24,292** | **~14,031** | **3** | ✅ |
| **3** | **~2,281** | **~0,782** | **1** | ✅ |
| **4** | **~4,851** | **~0,029** | **3** | ✅ |
| **4** | **~13,141** | **~0,039** | **2** | ✅ |
| **4** | **~1,798** | **~0,032** | **1** | ✅ |
| **5** | **~25,340** | **~8,904** | **3** | ✅ |
| **5** | **~1,825** | **~1,142** | **1** | ✅ |

Para la Query 1 – “Total de viajes por zona de inicio y finalización”, se evaluó el impacto del uso de Combiner ejecutando el trabajo en un cluster de tres nodos.

En promedio, el tiempo de carga de datos se mantuvo cercano a los 19 segundos, mientras que el tiempo de ejecución de la query fue de aproximadamente 377 segundos con Combiner y 399 segundos sin Combiner.

Se observa así una mejora cercana al 6 % en el tiempo total de ejecución al habilitarlo, debido a la reducción del tráfico de red entre mappers y reducers.

El uso de Combiner resultó beneficioso para esta query, dado que implica una operación de conteo con un alto número de claves repetidas.

Por otro lado, se evaluó el comportamiento del sistema al ejecutar las queries con distintos números de nodos físicos (1, 2 y 3) utilizando un archivo trips.csv de 1 millón de líneas, adaptado al de la catedra.

En todos los casos se mantuvo el uso del Combiner para asegurar condiciones comparables.

Los resultados muestran una disminución significativa del tiempo de ejecución al comparar la ejecución con 2 nodos contra 3 nodos. Por ejemplo, para la Query 1 el tiempo promedio pasó de aproximadamente 203 segundos con 2 nodos a 121 segundos con 3 nodos. Esto evidencia la escalabilidad del sistema y su capacidad de aprovechar mayores recursos.

Por otro lado, vemos en los resultados a lo largo de las 3 querys que se midieron tiempos significativamente menores en las ejecuciones con un solo nodo físico. Creemos que esto es producto de la conexión subóptima de nuestra configuración. Las pruebas sobre nodos físicos se realizaron con 2 de las máquinas conectadas a la red por wifi, y la última por cable de red. La conexión disponible ralentizó bastante el tiempo de carga de csv. En la vida real, esta no sería una situación óptima.

Las demás queries presentaron un comportamiento similar: al incrementar de 2 a 3 los nodos físicos, los tiempos de ejecución se redujeron de manera casi proporcional, especialmente en consultas de mayor carga de datos.

Por otra parte, antes de realizar las optimizaciones, intentamos ejecutar el programa utilizando el archivo CSV de 20 millones de líneas, pero el proceso fallaba debido a un error de memoria, mostrando el mensaje **“Out of Memory Error: Java heap space”**. Esto nos indicó que el código no estaba manejando eficientemente el uso de memoria para volúmenes de datos tan grandes. A partir de este descubrimiento decidimos modificar muchas de nuestras clases de datos para minimizar el uso de memoria.

Además, nos llevó a correr las pruebas usando los flags de java -Xms y -Xmx para asignar memoria a los nodos del server.

## 3. Potenciales puntos de mejora y/o expansión

Durante las pruebas realizadas, se observó que en algunas de las queries el tiempo de carga de datos representa una porción considerable del tiempo total de ejecución. En un entorno de producción real, donde los datos suelen encontrarse precargados en el cluster o en memoria distribuida, este costo inicial podría eliminarse casi por completo, permitiendo que las operaciones de MapReduce se ejecuten directamente sobre los datos ya disponibles. En Hazelcast, esto se puede lograr por medio de la clase MapLoader, siempre que el archivo del que se está cargando sea accesible directamente por el cluster.

Por otro lado, la implementación de una paralelización de la carga de datos —dividiendo el dataset entre múltiples clientes— permitiría escalar horizontalmente la inicialización del sistema y aprovechar mejor los recursos distribuidos. También se identifican posibles mejoras en el cacheo de parseos de fechas, la externalización de configuraciones (por ejemplo, tamaño de lote o formato de entrada) y la incorporación de testing automatizado para validar el rendimiento y la estabilidad de las optimizaciones implementadas.

En conjunto, estas mejoras apuntan a reducir los cuellos de botella iniciales, incrementar la eficiencia de las operaciones distribuidas y consolidar una arquitectura más robusta, flexible y preparada para escenarios de procesamiento continuo o en tiempo real.