# Instituto Tecnológico de Buenos Aires

# MapReducing Parking Tickets

*72.42 Programación Orientada a Objetos - 2024Q1*


## Alumnos:

**Tomás Santiago Marengo**, 61587

**Abril Occhipinti**, 61159

**Santino Ranucci**, 62092

**Agustin Zakalik**, 62068

## Profesores:

**Ing. Marcelo Turrín**

**Ing. Franco Román Meola**

# Índice

# 1. Introducción

En este trabajo, se explora el diseño y la implementación de un sistema distribuido utilizando Hazelcast y el paradigma MapReduce para procesar grandes volúmenes de datos de infracciones de tránsito. El objetivo principal es desarrollar consultas eficientes y escalables, optimizando el rendimiento y evaluando estrategias de procesamiento distribuido. Se tomaron decisiones críticas en la arquitectura del sistema para maximizar la eficiencia y minimizar la latencia, evaluando alternativas de diseño con base en la escalabilidad y robustez.

Se analizó el impacto de utilizar, o no, combiners, de diferentes implementaciones y se realizaron pruebas con conjuntos de datos variados, para probar cómo esto afecta el rendimiento. Estos experimentos permiten obtener una comprensión más profunda de cómo diferentes factores influyen en la eficiencia del procesamiento distribuido.

# 2. Diseño de los componentes de cada MapReduce

## 2.1. Elección de la Key del Mapa Inicial

En el diseño de un sistema MapReduce, **la elección de la clave (key) para el mapeo inicial es fundamental para asegurar una distribución equilibrada de la carga entre los nodos**. La clave debe ser seleccionada teniendo en cuenta la naturaleza del dataset para evitar sobrecargar un solo nodo. En un escenario ideal con un número considerable de nodos, la distribución equitativa de los datos es crucial.

Como ejemplo, para la consulta de las top 3 infracciones por barrio, inicialmente podríamos pensar en utilizar el barrio como clave. Sin embargo, esta elección puede llevar a una distribución desigual si ciertos barrios concentran un alto porcentaje de las infracciones. Por ejemplo, en un dataset de infracciones de tránsito en una gran estado, algunas ciudades de mayor concentración demográfica o industrial, pueden tener una mayor densidad de población y, por lo tanto, más infracciones. Si el barrio se utiliza como clave, los nodos que procesan esos barrios podrían sobrecargarse. El tiempo en un proceso en paralelo es el tiempo del que más tarda, por lo que en el peor caso podría ser como no distribuir la data.

Para evitar este problema, **se decidió utilizar una clave primaria (PK) autoincremental** como clave para la distribución de los datos. Esto asegura que los tickets se distribuyan aleatoriamente entre los nodos, equilibrando la carga de trabajo. Sin embargo, dependiendo de la naturaleza de la agrupación requerida por la consulta, puede requerir una etapa adicional de procesamiento para agrupar los datos de interés.

## 2.2. Componentes

Primero se detalla la idea general, que en principio valdría para cualquier consulta, salvo excepciones. En tal caso se nombrarán aparte.

Los **mappers** toman cada ticket con un ID particular y emiten pares clave-valor en la forma (clave de interés, 1). Cada ticket se sube a la red con la información pertinente para cada consulta. Esto es simplemente para agilizar los testeos y no sobrecargar la red, aunque se entiende que en un ambiente operativo y del día a día estos datos ya están cargados.

Los **combiners** realizan una operación similar a la de los reducers pero a nivel local en cada nodo. Agrupan los valores (1s) localmente para reducir la cantidad de datos que necesitan ser transferidos a través de la red en la fase de reducción.

Los **reducers** agrupan los valores recibidos de los mappers y combiners, sumando los 1s para cada clave de interés. Este proceso consolida los datos a nivel global.

Los **collators** trabajan con un orden de magnitud mucho menor después de la fase de reducción, realizando operaciones finales como el ordenamiento y selección de los top N elementos que son poco costosas dada la magnitud de la información que le llega, pero no es siempre así...

#### 2.2.1. ¿Y si el collator trabaja con mucha información?

En el caso de la Query 4, que requiere encontrar las placas de peor comportamiento por barrio, el orden de magnitud de los datos que llegan al collator es considerablemente alto. Los datos están compuestos por pares (barrio, placa), donde la cantidad de barrios es relativamente pequeña pero la cantidad de placas puede ser muy grande.

En ese caso hay dos opciones:

1. Para manejar este gran volumen de datos, se puede realizar un MapReduce adicional. Si el collator recibiera un dataset con un orden de magnitud muy alto, el procesamiento se volvería ineficiente, por ejemplo un ordenamiento sería muy costoso. Por lo tanto, se optó por dividir el trabajo en dos fases de MapReduce.

2. Se puede pensar desde cero el MapReduce, utilizando otra key que no sea el entero incremental, y que permita reducir la magnitud de la información en pasos anteriores como el mapper o el reducer.

#### 2.2.2. ¿KeyPredicate?

En este caso no usamos KeyPredicate porque este sirve para filtrar por las keys y las queries no pedían nada de filtrado previo que pueda ser usado en claves. Existe un filtrado de fechas para la Query 4 pero este se realiza en el Mapper para no emitir cosas de más. Se podría haber utilizado si se pidiera un filtrado como por ejemplo “los barrios pero que empiecen con A”, en ese caso podría haber estado bueno utilizar barrio como key.

### 2.3. Queries

A continuación se resume la lógica de cada query, recordando que resuelve cada una.

#### 2.3.1. Query 1

La consulta 1 es bastante simple. Pide el total de multas por infracción (definición).

1. El mapper emite (definición, 1).

2. El combiner simplemente suma lo que el mapper le mandó por cada definición, emite (definición, suma).

3. El reducer hace exactamente lo mismo que el combiner pero a nivel global.

4. El collator entrega el resultado final en un TreeSet para ser iterado y escrito en el CSV de salida.

### 2.3.2. Query 2

La consulta 2 pide top 3 infracciones más populares de cada barrio. En la sección 3.3 se menciona una colección propia que implementamos para esta query. Esta query si bien pide el top 3, se le puede pasar el mismo parámetro “Dn=” y que sea cualquier tipo de top.

1. El mapper emite (<barrio, definición>, 1).

2. El combiner simplemente suma lo que el mapper le mandó por cada par, emite (<barrio, definición>, suma).

3. El reducer hace exactamente lo mismo que el combiner pero a nivel global.

4. El collator entrega el resultado final en un TreeSet para ser iterado y escrito en el CSV de salida. Utiliza TopNSet para ordenar las infracciones por barrio.

### 2.3.3. Query 3

La consulta 3 pide top N agencias con mayor porcentaje de recaudación.

1. El mapper emite (agencia, monto).

2. El combiner simplemente suma lo que el mapper le mandó por cada agencia, emite (agencia, suma-Monto).

3. El reducer hace exactamente lo mismo que el combiner pero a nivel global.

4. Debido que el orden de magnitud de la información que recibe el collator no es tan grande (cantidad de agencias), este se encarga de calcular los porcentajes sobre el total del monto.

### 2.3.4. Query 4

La consulta 4 nos retorna la patente con más infracciones de cada barrio en el rango [from, to]. Esta en primera instancia se hizo con dos MapReduce.

1. El mapper1 filtra por el rango de fecha y emite (<patente, barrio>, 1).

2. El combiner1 simplemente suma lo que el mapper le mandó por cada par.

3. El reducer1 hace exactamente lo mismo que el combiner1 pero a nivel global.

4. Esto devuelve la cantidad de veces que existen cada par <patente, barrio>. Como el orden de magnitud es patente · barrio (muy alto) se hace un segundo MapReduce.

5. El mapper2 emite (barrio, <patente, cantidad>).

6. El combiner2 emite (barrio, patenteConMasInfracciones).

7. El reducer2 hace exactamente lo mismo que el combiner2 pero a nivel global.

8. El collator entrega el resultado final en un TreeSet para ser iterado y escrito en el CSV de salida.

### 2.3.5. Query 5

La consulta 5 resuelve los pares de infracciones que tienen, en grupos de a cientos, el mismo promedio de monto de multa.

1. El mapper emite (definición, monto).

2. El combiner emite (definición, <suma, cantidad>). Se necesita la cantidad porque el promedio de los promedios no es el promedio usualmente (solo cuando si la cantidad de todos los combiners es igual, lo cual no es necesariamente cierto).

3. El reducer ya puede calcular el “grupo de cien” y emitir (definición, grupo).

4. El collator entrega un TreeSet iterable, con los pares de infracciones que son del mismo grupo, sin repetir y en orden alfabético.

# 3. Cuestiones particulares sobre la implementación

## 3.1. Parallel Read

Para abordar eficientemente la lectura de archivos grandes, optamos por un enfoque secuencial con procesamiento concurrente. Esto significa que utilizamos un solo hilo para leer el archivo CSV línea por línea, colocando cada línea en una cola. Luego, múltiples hilos consumen las líneas de esta cola, procesando y filtrando los campos de interés para luego ser almacenados en el IMap de Hazelcast.

Como resultado directo de esta implementación, logramos reducir significativamente el tiempo de procesamiento de archivos grandes, como por ejemplo, el archivo CSV de NYC con 15 millones de líneas que pasó de más de 1 hora a entre 3 y 4 minutos (que consideramos razonable).

## 3.2. Multas de cualquier otra ciudad

Para ser lo más permisivos posible al leer archivos CSV de diferentes formatos, diseñamos un sistema configurable que permite al usuario definir la estructura del CSV a través de un archivo JSON y manejar diferentes formatos de fecha.

El usuario debe subir un archivo JSON que especifica el índice de cada columna de interés entre los campos mencionados en el trabajo práctico. Esto permite una flexibilidad considerable, ya que el sistema puede adaptarse a diferentes estructuras de archivos CSV sin necesidad de modificar el código fuente. El usuario además puede subir archivos que tengan más columnas y el programa se encargará de tomar sólo las de interés.

Luego, la clase DateFormats permite agregar diferentes tipos de formatos de fecha, facilitando la inclusión de CSV con distintos formatos de fecha. Las fechas suelen variar más que otros campos en los datasets, por lo que esta flexibilidad es esencial.

Para los demás campos optamos por elegir tipos que engloben a lo que se encuentra usualmente. Por ejemplo, para el monto usamos Double, ya que si es entero o con decimales no estamos perdiendo información.

## 3.3. TopNSet

TopNSet es una colección que se encarga de mantener siempre N elementos ordenados para cada key. La idea era usarla para la query 2 y 3, pero al final sólo sirvió para la 2. De esta manera dejamos a la segunda consulta como genérica en vez de “top 3”. Igualmente para respetar la consulta se pasa como parámetro 3 por defecto.

# 4. Resultados

## 4.1. Tiempo para cada query según cantidad de nodos

El siguiente gráfico se generó a partir de los resultados analizados sobre el dataset de 15M de líneas de NYC (el que se encuentra en Pampero).

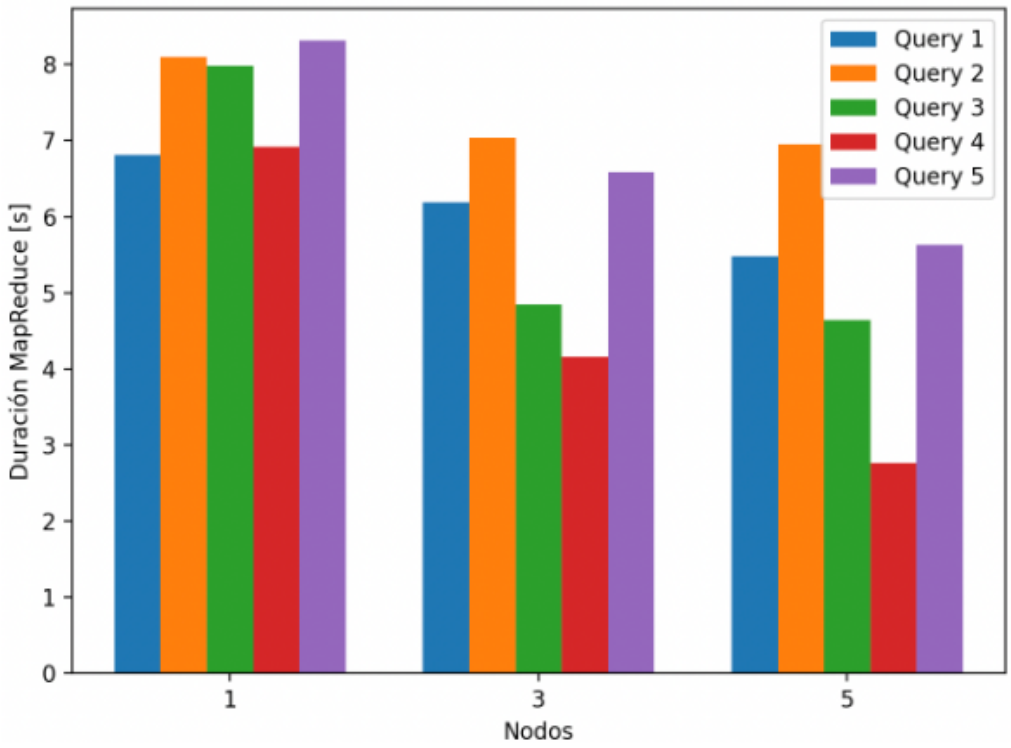


Figura 1: Tiempo de cada Query con diferente cantidad de nodos

Se destaca que:

1. La performance aumenta a medida que se agregan más nodos ya que se distribuye el cómputo de la tarea. Sin embargo (hay muchos "peros" en este trabajo práctico, pero queremos analizarlos o al menos entenderlos todos), este resultado es así ya que se utilizaron máquinas virtuales en Google Cloud Platform dentro de una misma red privada y muy rápida.

2. Si esto se hiciera con nuestras computadoras, abriendo puertos y pasando por internet, con ancho de banda muchos menor, al agregar más nodos podría ocurrir que el tiempo de envío de información exceda al tiempo de cómputo, resultando en mayores tiempos para mayor cantidad de nodos. No probamos esta configuración pero hablando con otros grupos notamos eso. En un ambiente productivo se deberían analizar estas cuestiones.

## 4.2. Con y sin Combiner

El siguiente gráfico se generó, también, a partir de los resultados analizados sobre el dataset de 15M de líneas de NYC. Pero este se redujo a diferentes tamaños para analizar el cambio de tiempo según se agregaban líneas.

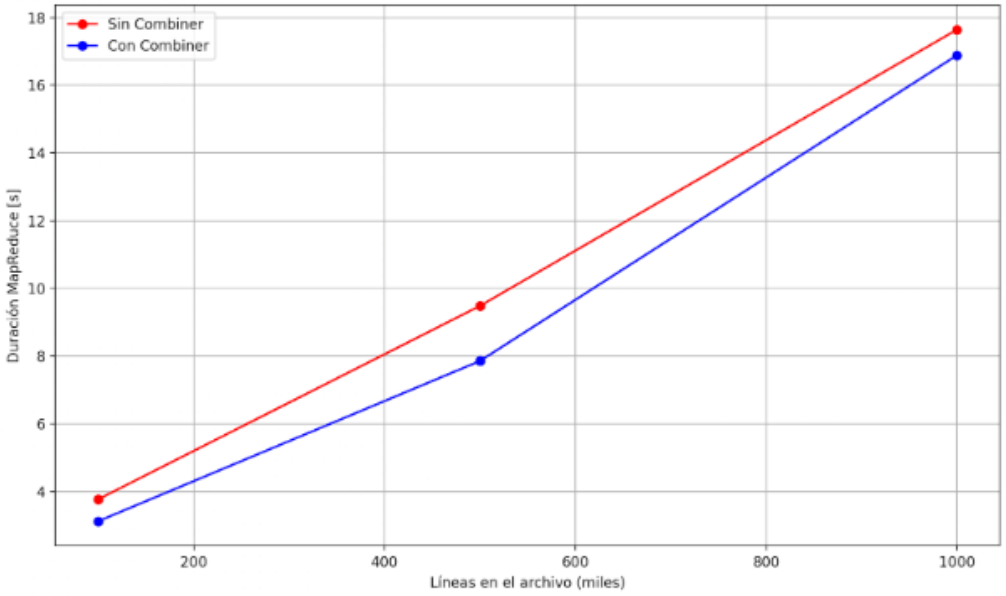


Figura 2: Duración del MapReduce para la Query2 con y sin combiner

Con esto podemos notar que claramente hay una diferencia entre tiempos, ya que la idea del Combiner es reducir la información que viaja por la red. Sin embargo, hay que volver a aclarar que esto puede no resultar así. Se destaca lo siguiente:

1. En un ambiente local no existe latencia entre el envío de información entre el Combiner y el Reducer, por lo que si estos hacen lo mismo (en diferentes niveles), este cambio en el tiempo de procesamiento no se nota. Casi lo mismo pasa en un ambiente con conexión cableada, que el traspaso de información se hace mucho más rápido que en un ambiente inalámbrico con ancho de banda reducido. Estos resultados nuevamente salen de la misma configuración en GCP.

2. La gráfica muestra (casi) una recta, por lo que a mismo aumento de tamaño del archivo, relativamente mismo aumento de tiempo de procesamiento.

## 4.3. Otra alternativa a la query 4

Como se mencionó, se usó una PK incremental como key en todas las queries, esto conllevó a que en la query 4 se requieran dos MapReduce debido al orden de magnitud de información que salía del primer Reducer.

Esto desembocó en probar otra configuración, donde el Mapper emite directamente (barrio, patente) en vez de emitir un (<barrio,patente>, 1) y reprocesar algunas cuestiones.

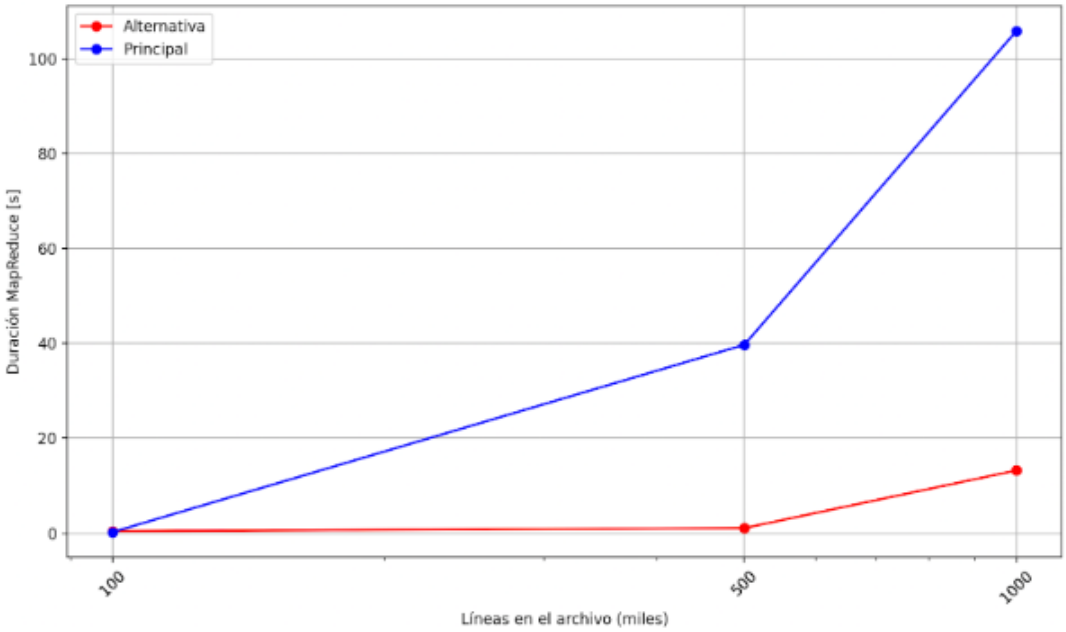


Figura 3: Comparación alternativa a Query4

Se puede ver que hay una mejora sustancial para el método alternativo. Cabe aclarar que estas corridas no se hicieron con la misma configuración de GCP sino que en una única computadora con menor procesamiento y RAM. Se siguió utilizando el dataset de NYC pero con menores cantidades de líneas.

La alternativa está en la branch main pero se tiene que cambiar a mano en el switch del cliente (ver sección “Mejoras a futuro”).

# 5. Mejoras a futuro

Se nombran algunos puntos de mejora y/o expansión, que se tuvieron en cuenta, pero debido al calendario apretado de junio y que no eran completamente necesarias para las condiciones se dejan como menciones.

1. Recibir por parámetro la estrategia a utilizar y crear diferentes estrategias para todas las queries, probando distintas key iniciales para los IMap.

2. Generar testeos automáticos y eficientes. Que no se vuelvan a cargar los IMap si no es necesario (imitando un ambiente productivo), ya que en nuestro código luego de cada query se hace un clear de los mapas utilizados.

3. Testear qué tanto cambia en un ambiente inalámbrico haber guardado las infracciones (código, definición) en un IMap. Ya que en realidad no tiene mucho sentido distribuir esa información porque no es tan pesada.

4. Analizar completamente la naturaleza del dataset. En realidad no es la idea, porque podrían ser infracciones como podrían ser palabras de un diccionario. Esto se nombró varias veces, por ejemplo, sabiendo los porcentajes de infracciones de cada barrio podríamos determinar si es buena idea utilizar "barrio" como key en algunas queries, o esto presentaría un desbalanceo.