Trabajo Práctico Especial 2

Multas de estacionamiento

72.42 - Programación de Objetos Distribuidos

Segundo cuatrimestre 2024

Grupo 2

Integrantes:

- Quian Blanco, Francisco (Legajo: 63006)
- Stanfield, Theo (Legajo: 63403)
- Ves Losada, Tobias (Legajo: 63342)

Profesores:

- Turrin, Marcelo Emiliano
- Meola, Franco Román

Diseño de componentes *MapReduce*

Query 1: Total de multas por infracción y agencia

El objetivo de esta consulta es obtener una lista de todas las infracciones, asociadas a las agencias correspondientes, y contar la cantidad total de multas emitidas por cada agencia para cada infracción. Luego, la lista debe ordenarse de forma descendente por la cantidad total de multas. En caso de empate en la cantidad, se debe desempatar alfabéticamente, primero por infracción y luego por agencia.

El *"KeyPredicate"* verifica la validez de la agencia como también verifica la validez del identificador de las infracciones viendo si están presentes en sus mapas correspondientes.

Dentro del *"Mapper"*, se obtiene un id de la agencia, el cual es asignado a la hora de construir el mapa, y se obtiene también el id de la infracción y se crea un par *"agencyID-infractionID"* para actuar de clave. Luego, emite un par clave-valor en el cual la clave es la combinación *"AgencyInfractionIdsPair"* y el valor es 1, indicando que una multa para el par agencia-infracción ha sido procesada.

El *"Reducer"* toma los valores emitidos por el *"Mapper"*, para cada par agencia-infracción, suma y almacena el total de multas que recibe. Al final, retorna el conteo total de multas por combinación de agencia e infracción.

El *"Combiner"* realiza un trabajo similar al del Reducer, pero lo hace a nivel local en cada nodo antes de enviar los resultados a la etapa final de reducción.

Por último, el *"Collator"* intercambia los identificadores de la agencia y de la infracción por los nombres de los mismos utilizando los mapas creados en *"Hazelcast"* y luego con un *"Comparator"* ordena los resultados de mayor cantidad de tickets a menor cantidad de tickets, desempatando alfabéticamente primero por infracción y luego por agencia.

Query 2: Recaudación *YTD* por agencia

El objetivo de esta consulta es calcular la recaudación total acumulada de cada agencia a lo largo del año, por mes, y mostrar el total *"YTD"* (Year To Date) de manera ordenada alfabéticamente por agencia y desempatando cronológicamente por año y mes. Para cada mes de un año determinado, el total *"YTD"* representa la suma de los montos de las multas emitidas por cada agencia desde el comienzo del año hasta ese mes inclusive.

El *"KeyPredicate"* verifica la validez de la agencia viendo si está presente en su mapa correspondiente.

El *"Mapper"* toma los datos filtrados por el *"KeyPredicate"* y extrae la agencia, el año y el monto de la multa, usando como referencia la fecha de emisión. Luego, emite un par clave-valor donde la clave es el objeto compuesto *"AgencyYearPair"* que representa la agencia y el año de emisión, mientras que el valor es otro objeto *"MonthAmountPair"* que contiene el monto de la multa y el mes en que se emitió.

El *"Reducer"* toma todos los valores emitidos por el *"Mapper"* para cada clave (par de agencia y año) y acumula los montos de todas las multas emitidas por esa agencia, agrupadas mes a mes. Cada valor representa el monto de una multa registrada en un mes específico, por lo que el resultado final de la reducción será un mapa ordenado que muestra la recaudación acumulada por cada mes (que tuvo recaudaciones) en orden cronológico. Así, al finalizar, cada mes incluye la suma total de la recaudación de las multas de dicho mes, lo cual es útil para calcular la recaudación acumulada del año (Year-to-Date, *"YTD"*).

El *"Combiner"* realiza un trabajo similar al del *"Reducer"*, pero lo hace a nivel local en cada nodo antes de enviar los resultados a la etapa final de reducción. Agrupa previamente por mes en los nodos retornando un mapa ordenado al *"Reducer"* y, de esta manera, minimiza las operaciones que debe realizar el *"Reducer"*. Por la implementación del *"Combiner"*, hubo que realizar la implementación de un segundo tipo de *"Reducer"* ya que cambiaron los tipos de datos que recibe el *"Reducer"*. Previamente el mismo recibía un par de mes-importe, pero la el *"Combiner"*, devuelve un mapa ordenado con los importes pre agrupados por mes. De esta manera, el nuevo *"Reducer"* recibe los mapas y combina los mapas para computar las sumas totales de importes en cada mes.

El *"Collator"* toma los resultados emitidos por el *"Reducer"* y organiza los datos en un conjunto ordenado. En este caso, los valores provienen de las recaudaciones acumuladas de las agencias durante cada mes del año. El *"Collator"* recibe del *"Reducer"*, por cada par agencia-año, un mapa ordenado que contiene la recaudación acumulada en cada mes. Entonces se toma dicho mapa y se recorre en orden cronológico, acumulando para cada mes progresivamente lo de los anteriores, calculando el *"YTD"*. Entonces organiza los datos en un *"SortedSet"* usando tres criterios de ordenación: primero por agencia alfabéticamente, luego por año en orden cronológico, y por último, dentro de cada año, por mes en orden cronológico. Para cada mes calculado, crea un objeto *"YTDCollectionResult"* con estos datos y lo agrega al *"SortedSet"*, el cual será finalmente devuelto.

Query 3: Porcentaje de patentes reincidentes por barrio en cierto rango

El objetivo de esta consulta es obtener el porcentaje de patentes reincidentes por barrio, donde una patente se considera reincidente en un barrio sí tiene al menos una cantidad especificada de multas de una misma infracción en el rango de fechas especificado. El porcentaje de reincidencia de patentes en un barrio se calcula dividiendo el número de patentes reincidentes únicas entre el total de patentes únicas en ese barrio dentro del rango de fechas.

Cómo utilizamos la fecha de emisión del ticket para clave del *"Job"*, el trabajo del *"KeyPredicate"* es filtrar todas aquellas claves que se encuentren por fuera del rango de fechas especificadas al momento de ejecutar. Previamente habíamos planteado una clave distinta pero, como el *"KeyPredicate"* debe filtrar por fecha, es correcto que esta misma sea la clave.

El *"Mapper"* toma como entrada el *"TicketRow"* (información de la multa) y la fecha de la infracción (*"LocalDate"*). Para cada registro de multa, emite un par clave-valor, donde la clave es el nombre del barrio (o *"county"*) donde ocurrió la infracción. El valor es un triplete que contiene el *"PlateId"* (número de patente del vehículo), el *"InfractionId"* (código de la infracción), y un valor constante (*"ONE"*), que indica que esta infracción representa una ocurrencia de multa. Este proceso permite organizar las infracciones por barrio y patente, lo que facilita luego el conteo de cuántas veces una patente ha cometido la misma infracción en un barrio, lo cual es clave para analizar las patentes reincidentes dentro de un período de tiempo específico.

En este caso hicimos dos *"Reducers"*, uno para utilizarse sin *"Combiner"* y otro con. El *"Reducer"* sin *"Combiner"* toma los valores emitidos por el mapper (tripletas de patente, infracción y recuento). Su función principal es contar las patentes reincidentes dentro de un barrio según una condición de mínimo de infracciones para una misma patente.

El *"Reducer"* organiza los datos de las patentes en un mapa, donde cada patente tiene un submapa que registra el conteo de infracciones por tipo de infracción. Luego, al final de la reducción, verifica si alguna patente tiene al menos *"n"* infracciones para una misma infracción, marcándose como reincidente.

Al finalizar, el *"Reducer"* calcula el porcentaje de patentes reincidentes dividiendo las patentes reincidentes entre el total de patentes en el barrio y lo multiplica por 100 para obtener un porcentaje.

En el otro caso, el *"Combiner"* realiza una agregación previa de los datos antes de pasarlos al *"Reducer"*. En lugar de enviar cada instancia de *"PlateNumberInfractionTriplet"* al

"Reducer", el combiner utiliza un mapa (*"Map<String, Map<String, Long>>"*) donde se mantiene el conteo acumulado de las infracciones para cada patente. Este mapa es intermedio, es decir, cada nodo procesa una parte de los datos antes de enviarlos al *"Reducer"*. Luego, el *"Reducer"* toma el mapa ya procesado y realiza la agregación final. Esto mejora el rendimiento al reducir la cantidad de datos que deben ser manejados en la fase final de la reducción.

El *"Collator"* recibe pares de barrio y porcentaje de patentes reincidentes, ordenándolos primero por porcentaje en orden descendente y luego alfabéticamente por barrio en caso de empate. Los barrios con un porcentaje de patentes reincidentes mayor a cero son los únicos incluidos en los resultados finales. Se utiliza un SortedSet para asegurar que los elementos estén siempre ordenados.

Query 4: Top N infracciones con mayor diferencia entre máximos y mínimos montos para una agencia

El objetivo de esta consulta es obtener, para cada infracción, el monto mínimo y máximo de las multas emitidas por una agencia específica, y calcular la diferencia entre estos montos. Los resultados se ordenan de forma descendente según la diferencia, y en caso de empate, alfabéticamente por el nombre de la infracción.

El *"KeyPredicate"*, elimina todas las claves que no sean equivalentes a la agencia especificada, pues la clave del *"Job"* es la agencia que emitió la infracción.

El *"Mapper"* toma cada registro de multa y emite dos valores para cada infracción: el monto de la multa y el ID de la infracción. Cada par emitido tiene como clave el *"infractionId"* (el identificador de la infracción) y como valor el monto de la multa (*"amount"*).

Al igual que en la query 3 en esta query también hicimos uso de dos *"Reducers"* diferentes para usar con y sin combiner. El *"Reducer"* sin combiner toma las multas de cada infracción y se va quedando con el máximo y mínimo de esas multas. Al finalizar, el *"Reducer"* devuelve un *"IntegerPair"* con los dos valores.

El *"Reducer"* con *"Combiner"* realiza una operación similar al *"Reducer"* sin *"Combiner"*, pero en este caso utiliza un *"IntegerPair"* para manejar los valores mínimos y máximos de manera más eficiente. En lugar de procesar un solo valor en cada paso de reducción, el combiner ya ha agrupado valores en pares dando el máximo y mínimo de un subconjunto de datos. Así, en la fase de reducción, el *"Reducer"* compara muchos menos valores.

El *"Collator"* organiza los resultados de las diferencias de montos de las multas por infracción, de mayor a menor diferencia. Primero, agrupa las infracciones con su monto máximo y mínimo, y calcula la diferencia. Luego, selecciona solo las *"n"* infracciones con la mayor diferencia, las ordena alfabéticamente en caso de empate y las retorna

Análisis de tiempos para resolución de queries

Los siguientes gráficos indican la relación entre los nodos de la red y el tiempo que tardaron las queries únicamente en realizar el funcionamiento del *"MapReduce"*, es decir, el tiempo de la carga del archivo no está tomada en cuenta. El archivo *tickets.csv* utilizado tiene un total de 3892 líneas, y los parámetros usados son los siguientes:

- City: NYC
- n: 2
- From: 01/01/1900
- To: 31/12/3000
- Agency: TRAFFIC

Todos los datos que corresponden a los nodos cuatro y cinco fueron precedidos y extrapolados a partir de los datos obtenidos (para las cantidades de nodos uno, dos y tres) y a partir de los resultados esperados (ver Anexo). A continuación se encuentran los gráficos para las distintas queries:

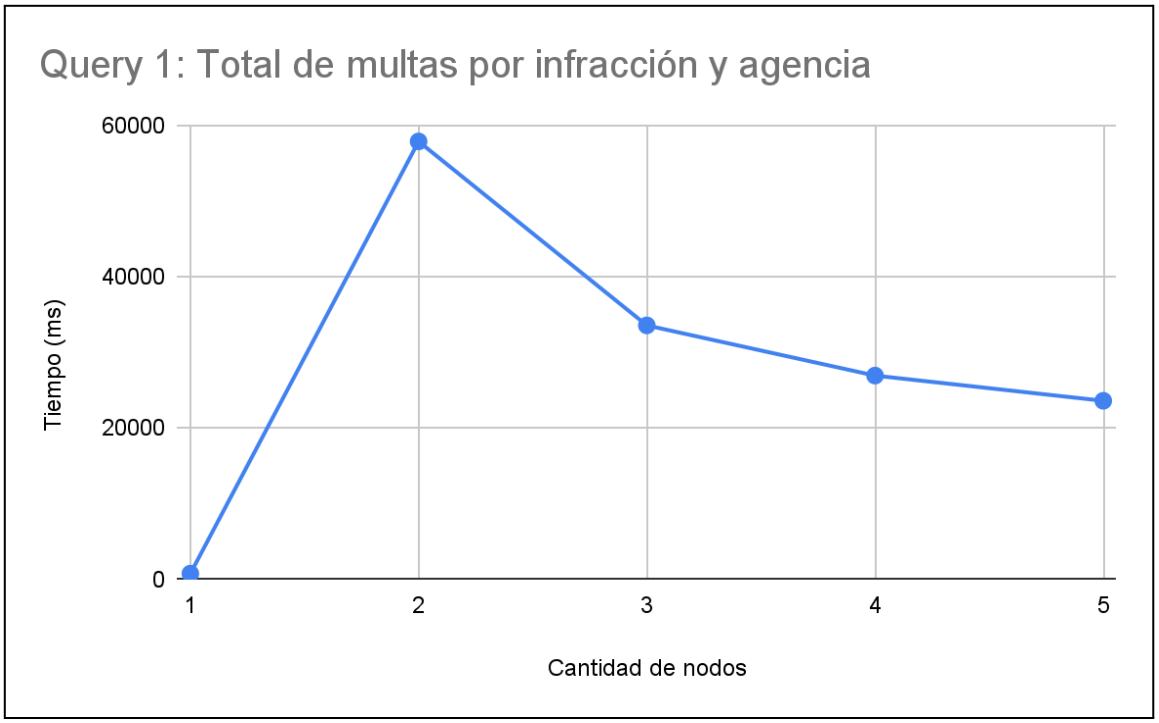

Gráfico 1: Tiempo transcurrido según cantidad de nodos para query 1

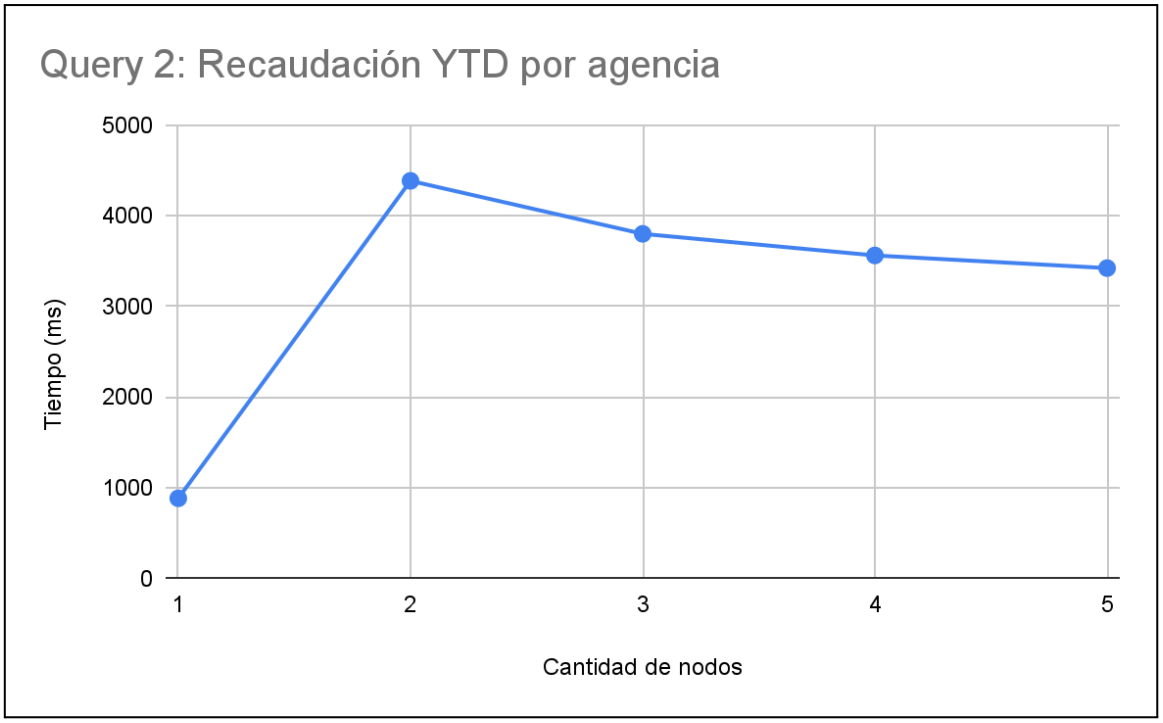

Gráfico 2: Tiempo transcurrido según cantidad de nodos para query 2

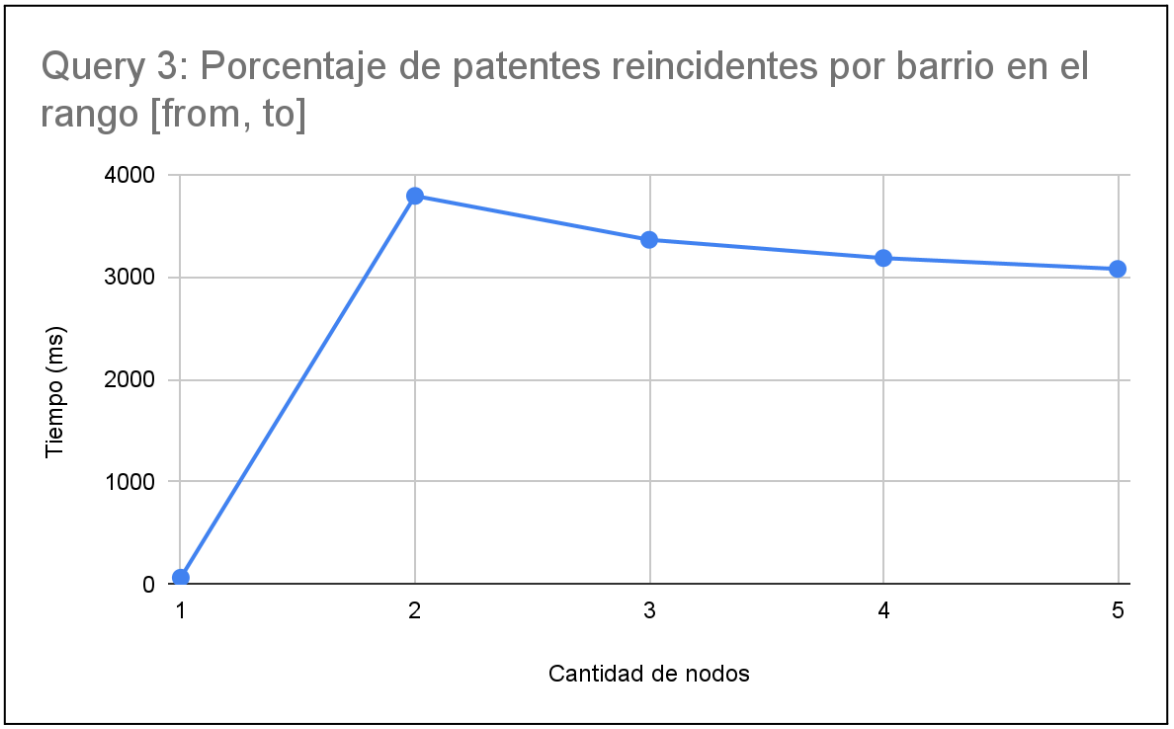

Gráfico 3: Tiempo transcurrido según cantidad de nodos para query 3

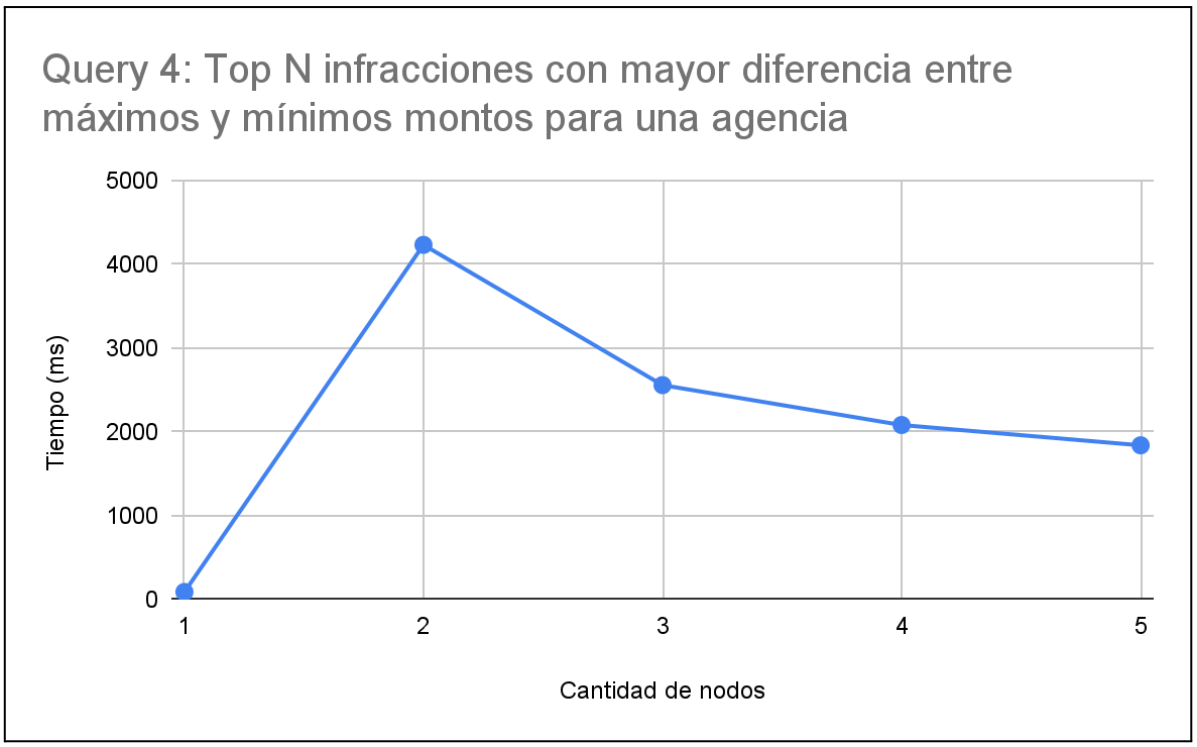

Gráfico 4: Tiempo transcurrido según cantidad de nodos para query 4

Como podemos observar, todas las queries siguen el mismo patrón. Para el primer nodo la velocidad de resolución es muy alta y luego, cuando se agrega el segundo nodo, su velocidad disminuye haciendo que el tiempo tardado para la resolución de las queries se incremente considerablemente. Ya al agregar los siguientes nodos, podemos observar un leve decremento en el tiempo transcurrido. Esto ocurre debido a que todos los procesos que se utilizan en el mapReduce al ser ejecutado en un solo nodo, corren en ese mismo nodo. Esto implica que no se tiene que utilizar la red para transferir datos. Al agregar el segundo nodo, estos nodos se deben comunicar y transferir datos a través de la red. Como sabemos, la red es muy lenta lo cual lleva a este gran incremento en el tiempo transcurrido. Al agregar más nodos, los datos van a seguir siendo transferidos por red pero tenemos una mayor cantidad de computo por lo que los tiempos tardados por las queries disminuyen.

Combiners

De igual manera que para los el ejecución de las queries sin *"Combiner"*, estas ejecuciones se realizan sobre las mismas bases. El archivo tickets.csv es el equivalente al cual se utilizó para las queries sin *"Combiner"* y tiene un total de 3892 líneas, los parámetros usados son los siguientes:

- City: NYC
- n: 2
- From: 01/01/1900
- To: 31/12/3000
- Agency: TRAFFIC

Los datos para la cantidades de nodos cuatro y cinco son datos predecidos utilizando el mismo método utilizado para ejecutar las queries sin *"Combiner"* (ver Anexo). Los siguientes gráficos indican la relación entre los nodos en la red y el tiempo que tardaron las queries en completar su funcionamiento:

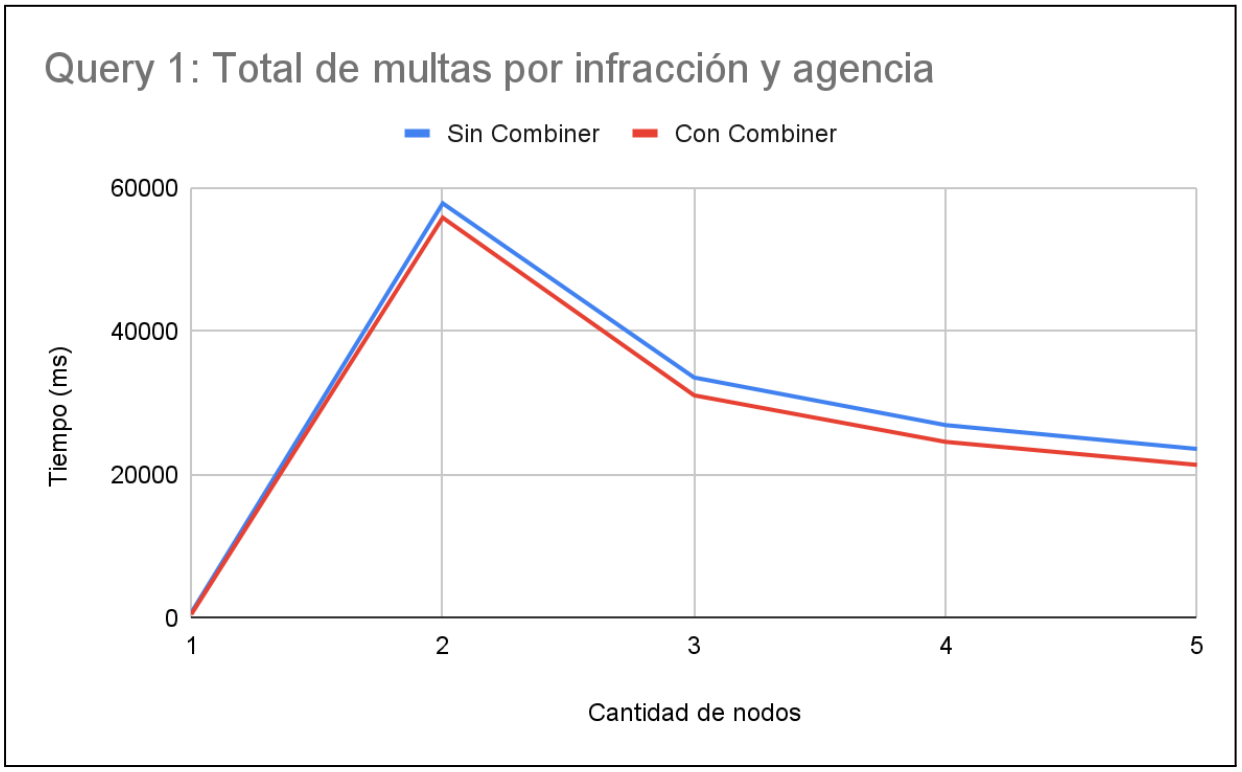

Gráfico 5: Comparación de tiempos con y sin combiner para query 1

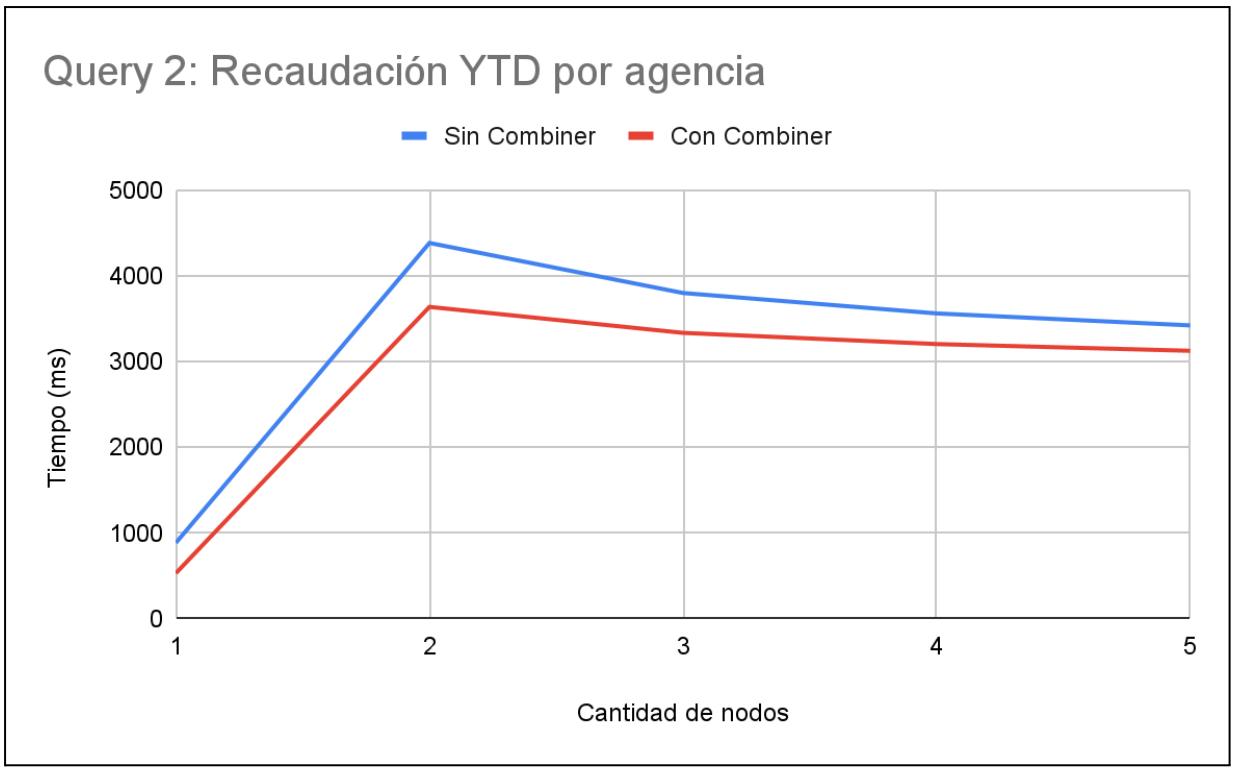

Gráfico 6: Comparación de tiempos con y sin combiner para query 2

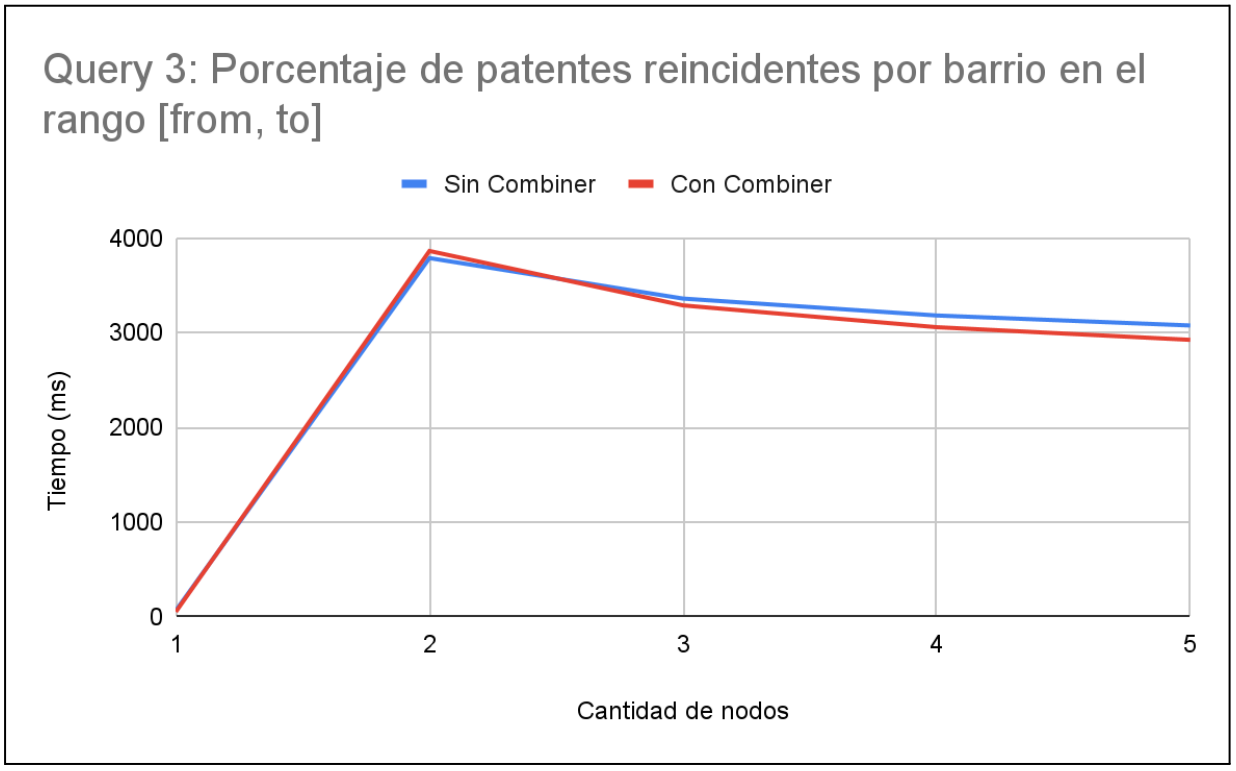

Gráfico 7: Comparación de tiempos con y sin combiner para query 3

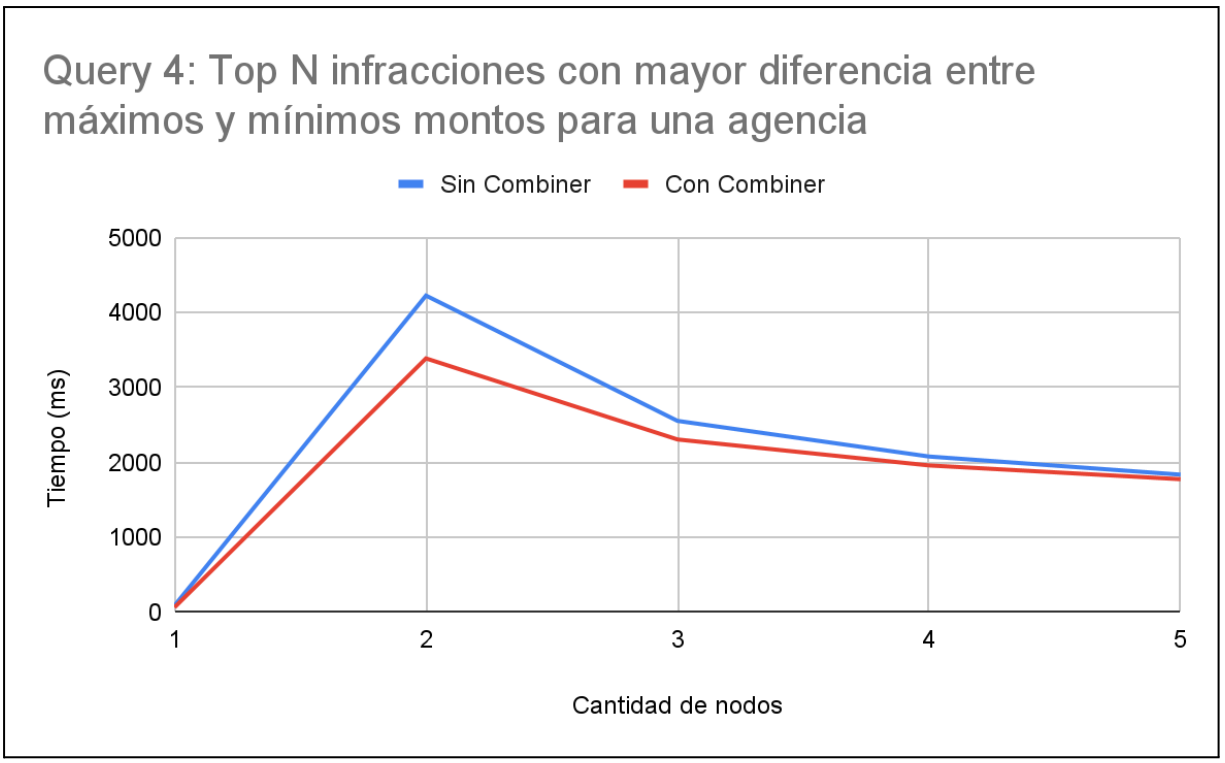

Gráfico 8: Comparación de tiempos con y sin combiner para query 4

En nuestras consultas, observamos que el "Combiner" contribuye a reducir el tiempo de ejecución, aunque el impacto puede parecer menor de lo esperado. Esto podría deberse al tamaño relativamente pequeño del archivo utilizado en nuestras pruebas (3892 líneas), en comparación con el archivo original de NYC, que cuenta con aproximadamente 15 millones de líneas.

Consideramos que, a medida que aumenta el tamaño del archivo de entrada, el efecto del "Combiner" en el tiempo de ejecución se vuelve más significativo, ya que permite agrupar los datos en cada nodo antes de enviarlos. Esto reduce la cantidad de datos que se transmiten a través de la red, disminuyendo significativamente el tráfico en comparación con el caso donde todos los datos se envían sin agrupar.

Potenciales puntos de mejora y/o expansión

Un potencial punto de mejora podría ser reemplazar todos los *"String"* los cuales son emitidos por los *"Mappers"* por un id (*"Integer"*) el cual los identifique. Como la mayor demora de tiempo se pierde cuando los datos viajan en la red, deberíamos focalizarnos en este aspecto para poder mejorar el rendimiento del servidor. Al reemplazar los *"String"* por un *"Integer"*, disminuimos la cantidad de datos que enviamos por red lo cual debería agilizar el proceso de *"MapReduce"* y concluir en un proceso más rápido. Esto puede ser aplicado específicamente a el mapa creado a partir de los *infractions.csv* los cuales contienen el identificador de la infracción junto con su descripción o su nombre. El problema aquí es el siguiente, como el identificador puede tener letras debe ser almacenado como *"String"* y este es el valor el cual es enviado por red. La solución a esto podría ser almacenar estos datos en un mapa con un identificador asignado por nosotros, el cual sea *"Integer"* como clave y como valor la dupla del identificador y su descripción. A la hora de asignar el identificador se debe tener cuidado pues debe ser *"thread safe"* ya que podría ser accedido al mismo tiempo, por lo que se debería aplicar el principio de atomicidad a la hora de ir asignando estos identificadores.

Otro análisis de tiempos de ejecución

A la hora de cargar los datos en los mapas de *"Hazelcast"*, sin importar la query, cargamos todos los datos provenientes del *tickets.csv*. Esto podría ser mejorado porque las queries no suelen utilizar todos los datos, sino una simple porción de ellos. La optimización se basa en cargar solo los datos necesarios para cada query. Por ejemplo, para el caso de la query 1, nuestra implementación estaba utilizando únicamente los datos de la clave del mapa que utilizamos para agrupar de manera conveniente para la implementación del *"KeyPredicate"*, pero igualmente estábamos cargando como valor para dicha clave, un objeto de *"TicketRow"* conteniendo la información de toda la fila del *CSV*, lo que aumentaba los tiempos de distribución de datos por haber que serializar y deserializar una mayor cantidad de atributos. Lo lógico, ya que no se utilizaban dichos datos, era optimizar la implementación para únicamente cargar los datos útiles en cada query.

Para probar la optimización decidimos crear una rama aparte en el repositorio de *GitHub*, y realizamos los cambios única y exclusivamente para probarlos frente a los archivos de la ciudad *NYC ("New York City")* como prueba de concepto.

Los cambios consistieron en, para la query 1 como ya se mencionó, simplemente enviar un *"AtomicInteger"* incrementando en cada iteración su valor, para diferenciar cada registro y poder acumular los valores, y, para las queries 2 a 4, en crear nuevas estructuras que permitan almacenar, serializar y deserializar los valores requeridos para cada query. Las estructuras propuestas son las siguientes:

- Para la query 2 requerimos únicamente del nombre de la agencia como clave y de los campos de fecha de emisión e importe como valor. Por lo que creamos una clase *"IssueDateAmountPair"* que nos permita almacenar dichos campos. El resultado de la carga de datos en el cliente se ve de esta forma:

```
try (Stream<String> lines = Files.lines(Paths.get(inPath,  ...more: "tickets" + city + ".csv"), StandardCharsets.UTF_8)) {
    AtomicInteger id = new AtomicInteger();
    lines.skip( n: 1).forEach(line -> {
        String[] split = line.split( regex: ";");
        LocalDate issueDate = LocalDate.parse(split[4], DateTimeFormatter.ofPattern("yyyy-MM-dd"));
        double amount = Double.parseDouble(split[2]);
        ticketsMultiMap.put(split[3], new IssueDateAmountPair(id.getAndIncrement(), issueDate, amount));
    });
}
```

- Para la query 3 requerimos únicamente de la fecha de emisión como clave y de los campos de barrio, patente y código de infracción como valor. Por lo que creamos una

clase *"CountyPlateInfractionTriplet"* que nos permita almacenar dichos campos. El resultado de la carga de datos en el cliente se ve de esta forma:

```
try (Stream<String> lines = Files.lines(Paths.get(inPath, ...more: "tickets" + city + ".csv"), StandardCharsets.UTF_8)) {
    AtomicInteger id = new AtomicInteger();
    lines.skip( n: 1).forEach(line -> {
        String[] split = line.split( regex: ";");
        LocalDate issueDate = LocalDate.parse(split[4], DateTimeFormatter.ofPattern("yyyy-MM-dd"));
        ticketsMultiMap.put(issueDate, new CountyPlateInfractionTriplet(id.getAndIncrement(), split[5], split[0], split[1]));
    });
}
```

- Para la query 4 requerimos únicamente del nombre de la agencia como clave y de los campos de código de infracción e importe como valor. Por lo que creamos una clase *"InfractionAmountPair"* que nos permite almacenar dichos campos. El resultado de la carga de datos en el cliente se ve de esta forma:

```
try (Stream<String> lines = Files.lines(Paths.get(inPath, ...more: "tickets" + city + ".csv"), StandardCharsets.UTF_8)) {
    AtomicInteger id = new AtomicInteger();
    lines.skip( n: 1).forEach(line -> {
        String[] split = line.split( regex: ";");
        double amount = Double.parseDouble(split[2]);
        ticketsMultiMap.put(split[3], new InfractionAmountPair(id.getAndIncrement(), split[1], (int) amount));
    });
}
```

Una vez hechos los cambios tuvimos que analizar el tiempo que transcurre desde el inicio de la query hasta el final para cada una de ellas. Se utilizó los archivos de la ciudad de Nueva York, ya que la implementación fue únicamente para esta ciudad, con el archivo tickets.csv con una longitud de 3892 líneas y los siguientes parámetros:

- City: NYC
- n: 2
- From: 01/01/1900
- To: 31/12/3000
- Agency: TRAFFIC

Se debe decir que para estas ejecuciones se utilizó simplemente un nodo en el cluster y no se utilizaron los *"Combiner"*, con el objetivo de mantener estas variables constantes y que no afecten a los resultados. Los resultados fueron los siguientes:

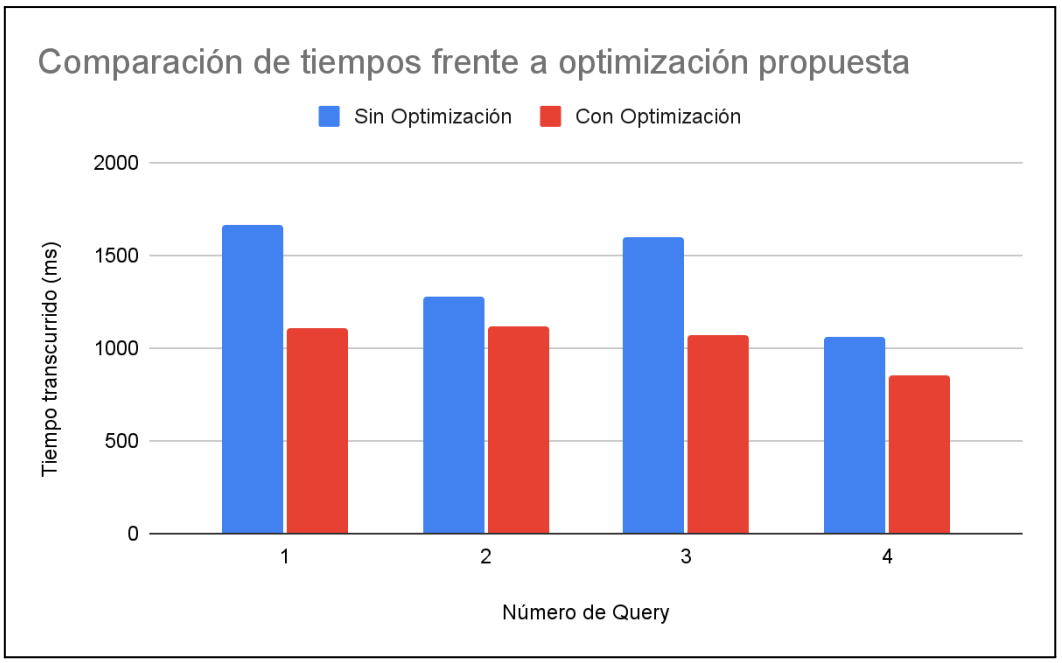

Gráfico 9: Comparación de tiempos con y sin optimización propuesta

Analizando los resultados podemos decir que la optimización propuesta fue satisfactoria. Los resultados obtenidos son promedios de varias ejecuciones y se entiende que en todas las queries la optimización reduce los tiempos. Esto es debido a la hora de serializar los datos para enviar a través de la red, si no incluimos datos innecesarios, se disminuye el tiempo de esta serialización sin perder funcionalidad ya que estos datos nunca iban a ser usados.

Conclusión

En este trabajo práctico, se diseñaron y evaluaron múltiples componentes del *"MapReduce"* para procesar datos de multas de estacionamiento en dos ciudades, Nueva York y Chicago, usando *"Hazelcast"*. Las decisiones de diseño se enfocaron en la eficiencia de procesamiento en un entorno distribuido, priorizando la reducción del tráfico de red y optimización de los tiempos de ejecución. Optamos por implementar *"Combiners"* que agrupa datos en los nodos, reduciendo significativamente el tráfico de red en consultas con archivos grandes, y mejorando el rendimiento general de las consultas en comparación con ejecuciones sin *"Combiner"*.

Anexo

El método que se utilizó para predecir los datos es el siguiente, veamos con un ejemplo. Tomemos los datos de la query 1 sin combiner:

Cantidad de nodos	Tiempo (ms)
1	762
2	57847
3	33549

Como los datos a calcular son los de los nodos cuatro y cinco, los cuales utilizan la red, podemos obviar el dato obtenido con un simple nodo, ya que este no utiliza la red, por lo que no es correcto utilizarlo para predecir el comportamiento.

Utilizando los datos obtenidos con dos y tres nodos, podemos hacer los siguientes cálculos para obtener el decremento porcentual del tiempo tardado y el decremento de los datos que van a parar a cada nodo (suponiendo que los datos se distribuyen equitativamente):

$$Decremento\ del\ tiempo: \frac{57847-33549}{57847} \cdot 100 = 42\%$$

$$Decremento\ de\ los\ datos: 50\% - 33\% = 17\ puntos$$

Cuando habían dos nodos se distribuían el 50% de los datos a cada nodo, y para tres nodos se distribuye un 33% a cada nodo (siempre asumiendo que los datos se distribuyen equitativamente entre todos los nodos). A la hora de agregar el cuarto nodo, los datos se distribuyen en un 25% para cada nodo, y cuando sean cinco nodos se distribuyen en un 20% para uno.

Pasar de tres nodos a cuatro es equivalente a ver qué el esparcimiento de los datos pasa de un 33% para cada nodo a un 25% para cada nodo, un decremento de los datos de 8 puntos. Por esto haciendo regla de tres simple podemos ver qué decremento de tiempo debe ser el relacionado de pasar desde tres nodos a cuatro nodos, pasar de un esparcimiento de 33% a 25%.

$$Decremento\ del\ tiempo: \frac{42\% \cdot 8}{17} = 19,767\%$$

Al haber encontrado el decremento del tiempo al pasar de tres nodos a cuatro nodos, podemos encontrar el tiempo que debería tardar la ejecución de la query para cuatro nodos:

$$Tiempo\ cuatro\ nodos: 33549\ ms\ -\ 33549\ ms\ \cdot\ 0,19767\ \approx\ 26917,5$$

Utilizando la misma estrategia para cinco nodos, decremento desde 25% a 20% de los datos:

$$Decremento\ del\ tiempo: \frac{19{,}767\% \cdot 5}{8} = 12,354\%$$

$$Tiempo\ cinco\ nodos: 26917,5\ ms\ -\ 26917,5\ ms\ \cdot\ 0,12354\ \approx\ 23592,1$$

El resultado de la tabla queda de la siguiente manera:

Cantidad de nodos	Tiempo (ms)
1	762
2	57847
3	33549
4	26917.5
5	23592.1

De esta manera fuimos capaces de predecir el comportamiento con cuatro y cinco nodos a partir de los datos obtenidos para todas las quieres, incluyendo las queries utilizando los "Combiner". Estamos conformes con nuestro método para predecir el comportamiento ya que revisitando el gráfico 1:

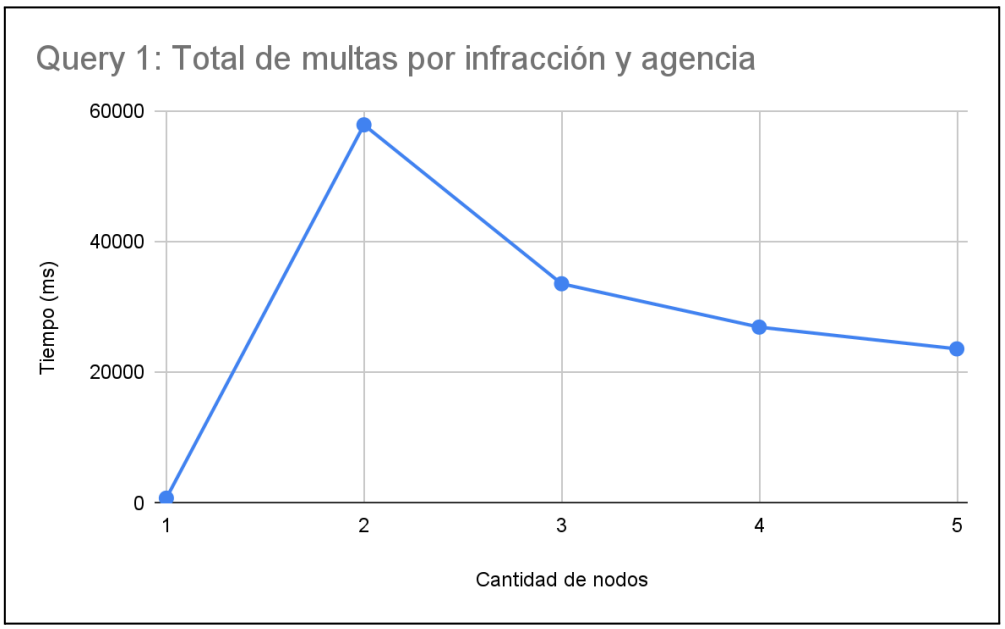

Gráfico 1: Tiempo transcurrido según cantidad de nodos para query 1

Podemos claramente ver cómo al seguir agregando nodos, el tiempo transcurrido es cada vez menor, lo cual tiene sentido ya que cuanto más nodos, menor cantidad de información tendrá que procesar cada nodo. También están conformes pues el decremento es cada vez menor, esto es debido a que pasar de utilizar dos a tres nodos puede ser un gran cambio ya que se está aumentando la capacidad de cómputo en un 50%, pero ya al seguir agregando nodos el aumento en la capacidad de de cómputo es cada vez menor, por lo que los tiempos transcurrido para que se ejecuten la queries deben seguir decrementando pero cada vez a un ritmo menor, que es precisamente lo que indica los tiempos predichos.