Informe TPE2

Programación de Objetos Distribuidos

Procesamiento de Reclamos Urbanos con Hazelcast MapReduce

I.T.B.A.

Integrantes:
José Burgos (61525)
Lautaro Gazzaneo (61484)
Pedro Curti (61616)

Índice

1. Introducción

Este proyecto tiene como objetivo procesar grandes volúmenes de datos de reclamos urbanos utilizando la herramienta Hazelcast MapReduce (versión 3.8.6) en Java. Los datasets corresponden a reclamos de las ciudades de Nueva York (NYC) y Chicago (CHI). El procesamiento incluye tareas como conteo de reclamos por tipo y agencia, identificación de los tipos más populares, cálculo de medias móviles y porcentajes de tipos de reclamo por calle. La elección de Hazelcast MapReduce permite distribuir la carga de procesamiento y manejar datasets de millones de reclamos de forma eficiente.

2. Queries: Descripción, Diseño y Decisiones Técnicas

2.1. Query 1: Cantidad de Reclamos por Tipo y Agencia

2.1.1. Descripción

Esta consulta calcula la cantidad total de reclamos para cada par de (tipo de reclamo, agencia). El resultado es un archivo CSV con columnas: tipo;agencia;cantidad. Se filtran únicamente los tipos de reclamos válidos presentes en el archivo de tipos.

2.1.2. Diseño

Tenemos un solo par Map/Reduce. El Mapper recibe objetos Complaint y emite un par clave-valor con la clave compuesta (tipo, agencia) y como valor 1. El Reducer suma los valores para obtener la cantidad total de reclamos. Luego el Collator lo ordena como es pedido.

2.1.3. Decisiones Técnicas

Esta query se pensó como un simple word-count, con el agregado de la clave compuesta. Se eligió una clave compuesta `Pair<String, String>` para poder agrupar por tipo y agencia.

2.1.4. Posibles Optimizaciones

No encontramos una optimización ideal, ya que es un word-count. Más allá del `Combiner` no hay mucho para hacer.

2.2. Query 2: Tipo más Popular por Barrio y Cuadrante

2.2.1. Descripción

Se obtiene, para cada barrio y cuadrante, el tipo de reclamo más popular. Para el cálculo del cuadrante, para un determinado valor de q se tiene:

$$Q_q(\text{lat}, \text{lon}) = \left(\left\lfloor \frac{\text{lat}}{q} \right\rfloor, \left\lfloor \frac{\text{lon}}{q} \right\rfloor \right), q \in (0, 1]$$

2.2.2. Diseño

El Mapper emite como clave el par (barrio, cuadrante) y como valor el tipo de reclamo. El Reducer cuenta las ocurrencias de cada tipo y selecciona el de mayor frecuencia. Luego el collator los ordena como es pedido.

2.2.3. Decisiones Técnicas

Se utiliza un `Pair<String, Pair<Integer, Integer>` para clave, reflejando barrio y cuadrante. Cuando fuimos a agregar el `Combiner`, nos encontramos con el problema de que nuestro `Reducer` esperaba `(key, tipo)`, pero el `Combiner` que pre agrega, no tiene sentido que devuelva `(key, Collection<tipo>)`. Entonces hicimos un nuevo `Reducer` que reciba `(key, int)` que son los tipos distintos que se preagregaron en el `Combiner`.

2.2.4. Posibles Optimizaciones

Ajustar el tamaño de cuadrante q según la densidad de datos.

2.3. Query 3: Media Móvil de Reclamos Abiertos por Agencia

2.3.1. Descripción

Calcula, para cada agencia y para cada mes de cada año, el promedio móvil de tamaño w meses de los reclamos abiertos en la ciudad correspondiente. En **NYC**, un reclamo está abierto si su estado es distinto a "Closed"; en **CHI**, si su estado es "Open". Para los primeros meses del año, la ventana se adapta al número de meses disponibles e incluye meses con cero reclamos para reflejar variaciones completas en la tendencia.

2.3.2. Diseño

Se emplea un patrón producer–consumer donde un hilo lee líneas del CSV y las inserta en una cola bloqueante, mientras varios hilos consumidores paralelos extraen esas líneas, las convierten en objetos `Complaint` y los agrupan en lotes. Cuando cada lote alcanza el tamaño fijo (`BATCH_SIZE`), se envía un único `putAll()` a Hazelcast para minimizar operaciones remotas. Al terminar la lectura, el productor introduce "poison pills" para cada consumidor, señalando el fin de datos y permitiendo vaciar el lote restante. Esta arquitectura maximiza la superposición entre lectura, parsing y envío a la vez que aprovecha todos los núcleos disponibles. De este modo se evitan cuellos de botella de I/O y se incrementa la eficiencia general.

2.3.3. Decisiones Técnicas

Se optó por `LinkedBlockingQueue` con capacidad limitada (`BATCH_SIZE*2`) para balancear la memoria y el throughput entre productores y consumidores. `AtomicInteger` asegura la generación de IDs únicos sin bloqueos adicionales. El número de hilos consumidores (`NUM_WORKERS`) se deriva de los núcleos de CPU disponibles para escalar automáticamente. El uso de "poison pills" garantiza un cierre ordenado de los hilos consumidores.

2.3.4. Posibles Optimizaciones

Si la API de Hazelcast lo permite, calcular la media móvil directamente en el Reducer reducirá la lógica de post-procesamiento en el cliente y disminuirá el tráfico de datos entre nodos; además, mover parte de la lógica de ventana al Collator permitirá paralelizar únicamente la agregación inicial y limitará el volumen de datos que deben moverse durante el shuffle.

2.4. Query 4: Porcentaje de Tipos de Reclamo por Calle en un Barrio

2.4.1. Descripción

Calcula el porcentaje de tipos de reclamo distintos por calle dentro de un barrio específico. El resultado tiene las columnas: calle;porcentaje. Se filtra únicamente el barrio indicado por el parámetro `-Dneighbourhood`.

2.4.2. Diseño

En este caso desarrollamos dos versiones de este código. Uno con un Map/Reduce y otro con dos Map/Reduce. Para el caso de único Map/Reduce el `Mapper` emite `((calle, tipo), 1)` luego el `Reducer` cuenta los tipos los acumula en un Set y devuelve el Set para que el `Collator` finalmente haga la lógica de sacar el porcentaje para cada valor. Luego con Combiner es similar, solo que el `Combiner` finalmente descarta la cantidad de valores emitidos para la misma key y lo mismo hace el `Reducer`. El `Collator` nuevamente es quien implementa la lógica para sacar el porcentaje.

2.4.3. Decisiones Técnicas

Se emplea un Combiner en ambas fases para sumar localmente los contadores y reducir el tráfico de red antes del shuffle. Las claves se formatean como cadenas separadas por ";" para asegurar una partición correcta en Hazelcast y reiniciar los totales al cambiar de mes o agencia. Se utiliza `BigDecimal` con `RoundingMode.DOWN` para truncar los porcentajes a dos decimales sin redondeo. Un `IMap` intermedio almacena resultados de la primera fase, facilitando la separación de responsabilidades y la reutilización de datos. El Collator final construye la salida en CSV y garantiza el orden alfabético por agencia y cronológico por año/mes.

2.4.4. Posibles Optimizaciones

Incorporar un `Combiner` adicional en la segunda fase para realizar un pre-cálculo parcial de los porcentajes mejora el uso de ancho de banda en el shuffle.

3. Métricas de Rendimiento y Escalabilidad

3.1. Metodologia de las pruebas

Se realizaron las pruebas con el siguiente sistema. Se tenían 4 nodos, un par corriendo en la máquina A y otro par corriendo en la máquina B. Luego, se ejecutaron las Queries con y sin Combiner para ambos datasets con 3 tamaños distintos: 100 mil de entradas, 1 millón de entradas y 5 millones de entradas. A continuación contamos con la descripción de las máquinas A y B. En las siguientes subsecciones veremos los resultados.

- **Maquina A**:
- **OS**: Debian GNU/Linux bookworm 12.10 x86_64
- **Kernel**: Linux 6.1.0-32-amd64
- **CPU**: AMD Athlon 3000G with Radeon Vega Graphics (4) @ 3.50 GHz
- **RAM**: 17.51 GiB
- **Maquina B**:
- **OS**: Arch Linux x86_64
- **Kernel**: Linux 6.15.2-arch1-1
- **CPU**: AMD Ryzen 3 1300X (4) @ 3.50 GHz
- **RAM**: 15.55 GiB

3.2. Análisis de Tiempos

3.2.1. Query 1

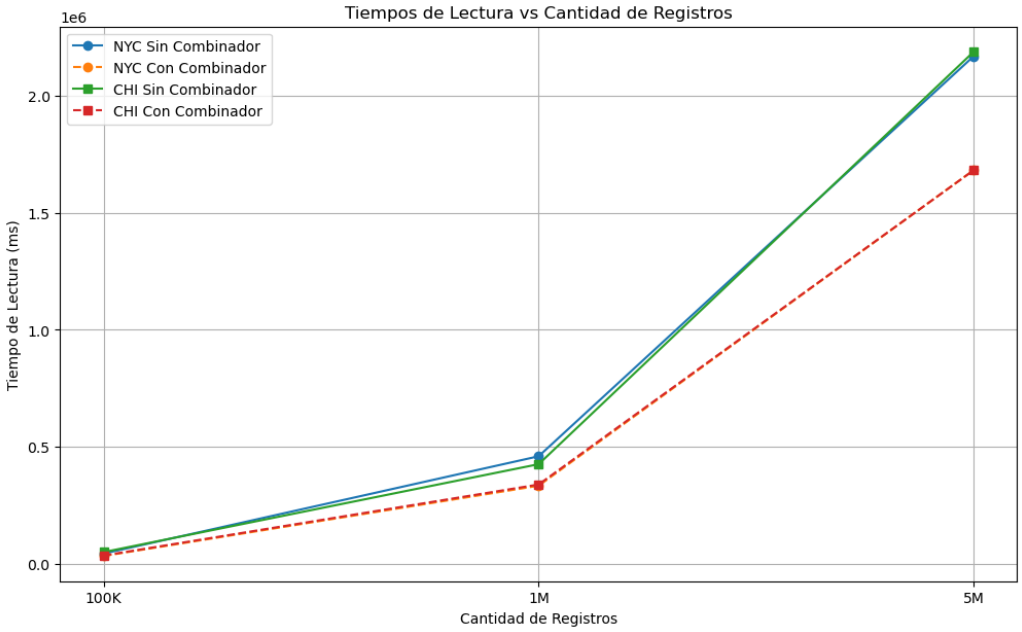

Figura 1: Tiempo de lectura en función de la cantidad de registros.

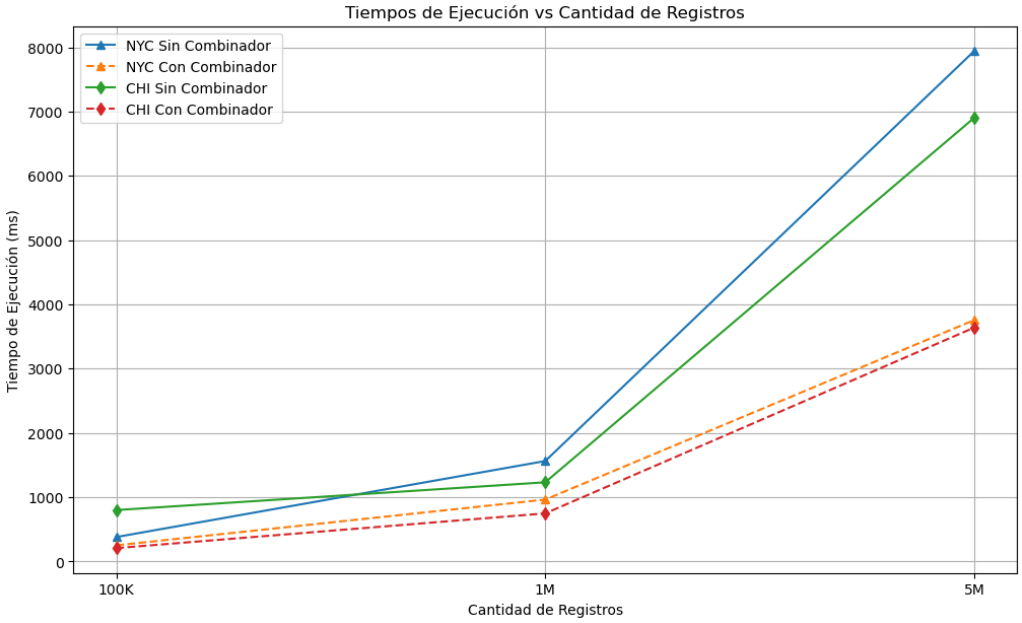

Figura 2: Tiempo de ejecución en función de la cantidad de registros.

3.2.2. Query 2

Parámetros:

- $q = 0{,}1$

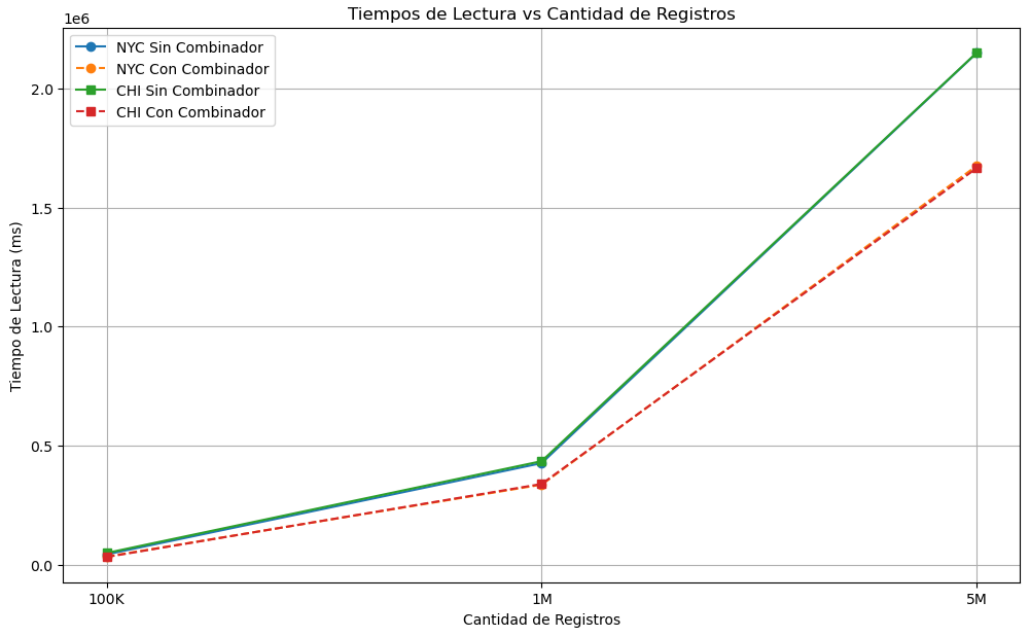

Figura 3: Tiempo de lectura en función de la cantidad de registros.

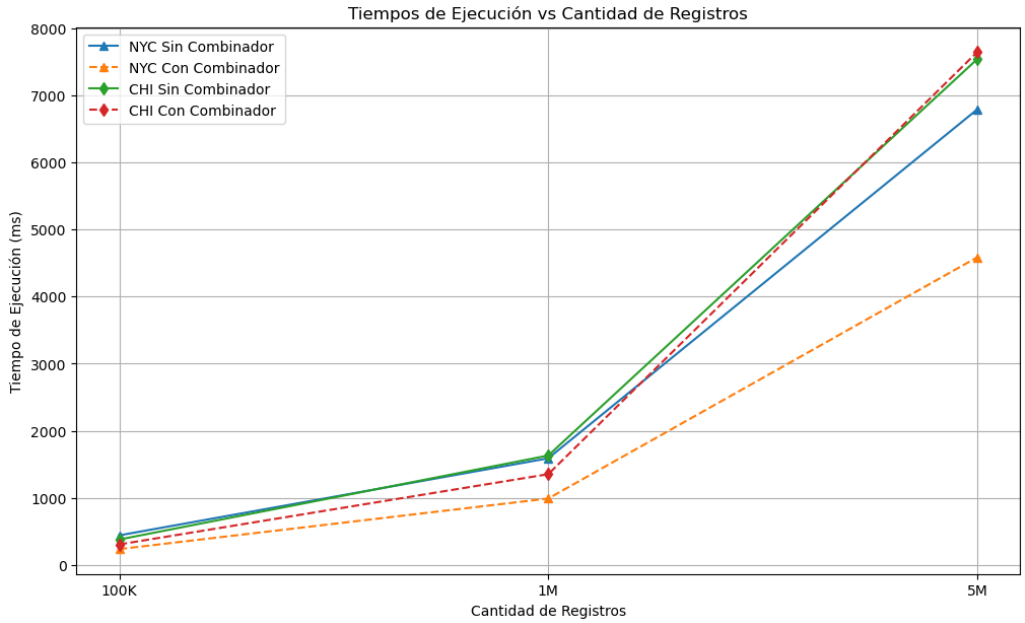

Figura 4: Tiempo de ejecución en función de la cantidad de registros.

3.2.3. Query 4

Parámetros:

- neighbourhood
- CHI: WEST_TOWN
- NYC: BROOKLYN

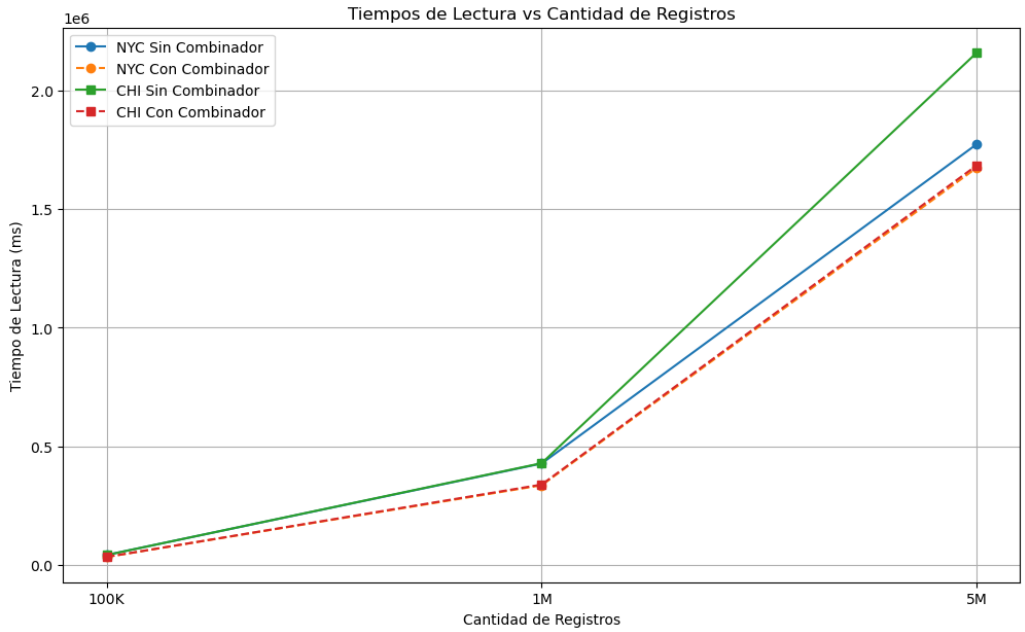

Figura 5: Tiempo de lectura en función de la cantidad de registros.

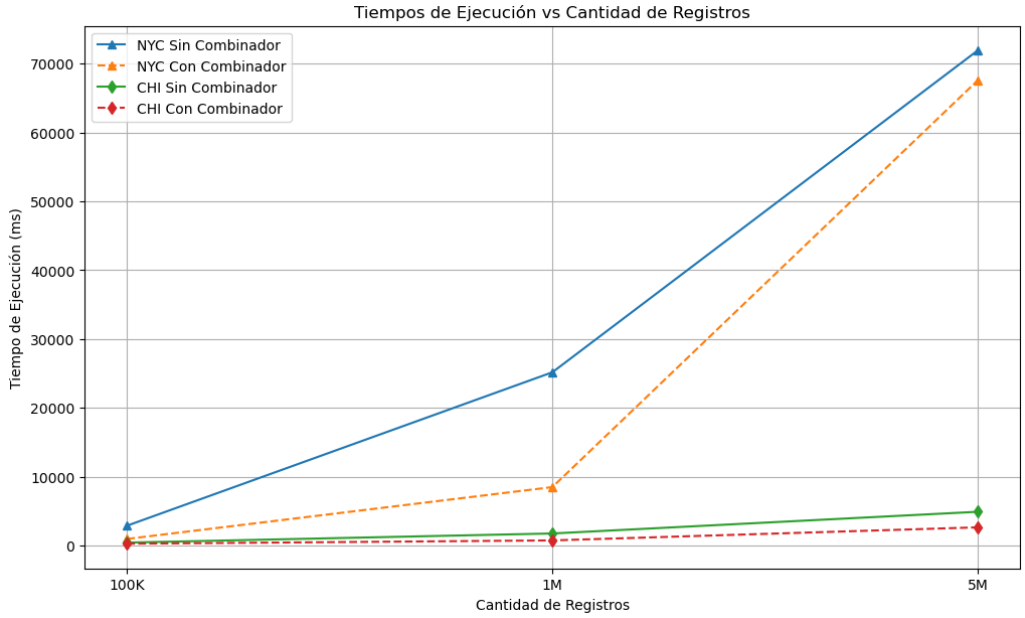

Figura 6: Tiempo de ejecución en función de la cantidad de registros.

3.3. Comparativa de Combiner

Como podemos ver en las figuras, la ejecución de las consultas con combiners, muestran, en general, una mejoría en los tiempos de ejecución, sobre todo en la primer query que es un word-count con una clave compuesta, como particularidad.

3.4. Otras Optimizaciones

Como podemos ver en los puntos anteriores y en los gráficos. Actualmente el mayor problema que encontramos es la lectura. Esto implica, leer la línea del archivo, parsearla, crear el objeto Complaint y subirlo al mapa de Hazelcast. En nuestro caso particular se hace todo una línea a la vez. O sea, por cada línea, primero la leemos, luego la parseamos, creamos el objeto y últimamente, la subimos al mapa de complaints. Esta forma de llevarlo a cabo tiene mucho tiempo perdido en operaciones de

I/O. Tanto la lectura del CSV como el tiempo que demora en conectarse con los nodos. Idealmente, los nodos no están todos en la misma máquina que el cliente, o en la misma red local, provocando todavía más latencia. Por lo tanto, consideramos que la mejora de estos tiempos debería ser la prioridad, por sobre ver si hay alguna manera de conseguir realizar las queries más eficientemente, sobre todo la 3 y 4, donde el cliente también trabaja.

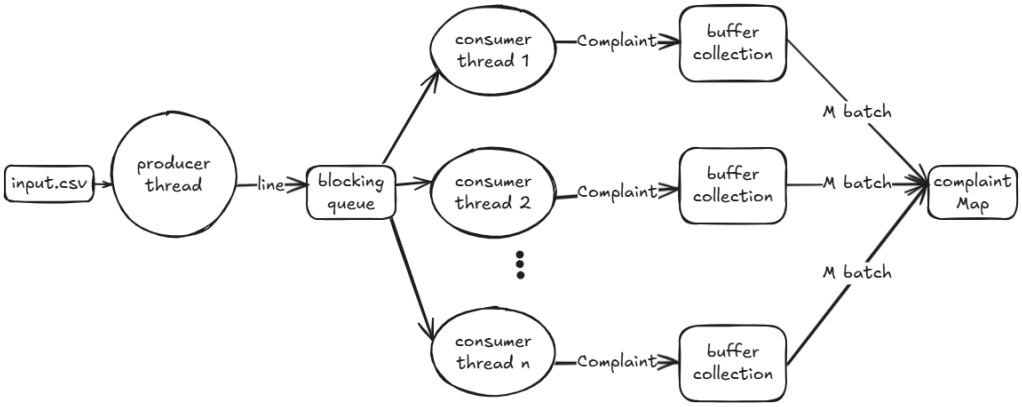

Figura 7: Esquema de la propuesta de optimización

La propuesta es utilizar un esquema de producer-consumer. Nuestro thread *producer* lee el CSV y deja las lineas en una cola bloqueante. Luego cuando termina de leer, agrega a la queue, tantas poison pills como consumers haya. Nuestros consumers, toman una linea la parsean, crean la Complaint y las suben a los nodos. Si bien esto ya implicaría una mejora sobre el proceso original, aun así, agregamos otra estrategia sobre esta, para reducir el trafico de red. Cada *consumer* cuenta con un colección, donde ira acumulando las Complaints que fue creando. Cada vez que esta colección alcanza un tamaño de batch predeterminado, realiza una operación de putAll al mapa de Hazelcast, reduciendo así el tiempo que se perdería si tiene que establecer la conexión por cada una de las complaints. A continuación un ejemplo de la velocidad que se gana, procesando el CSV original de esta manera.

El siguiente es el output del archivo de tiempos, corriendo con un único nodo en la misma maquina que el cliente, de la query 1, con el dataset original de CHI que es de 11M de lineas, aproximadamente.

```
14/06/2025 19:23:48:4529 INFO [main] Started reading complaints
14/06/2025 19:23:48:4539 INFO [main] Finished reading complaints. Duration: 13150 ms
14/06/2025 19:23:48:4541 INFO [main] Started MapReduce job
14/06/2025 19:23:48:4541 INFO [main] Finished MapReduce job. Duration: 7124 ms
```

Como podemos ver tardo tan solo 13 segundos en cargar todo el archivo, comparado con nuestros casos de ejecución con los 4 nodos en dos computadoras en una red local, donde llego a tardar mas de 30 minutos la lectura.

3.5. Posibles Mejoras

Si bien estamos conformes con la optimizacion lograda, creemos que hay mas espacio para mejoras. Las siguientes son algunas ideas.

- Separar los clientes de ejecución y lectura, permitiendo así subir una sola vez los archivos y poder correr todas las queries con una sola misma lectura. Esto es claramente como se hace en la realidad, y muchas veces el que se encarga de las queries no tiene que subir los datos sobre los cuales realizara consultas.

- Las queries 3 y 4 actualmente utilizan al cliente sobre el final. Es claro que probablemente se vean beneficiadas de usar mas de un Map-Reduce sobre todo con data sets aun mas grandes.

4. Conclusión

El uso de Hazelcast MapReduce permitió procesar millones de reclamos urbanos de forma distribuida y eficiente. Las decisiones técnicas, como el diseño de las claves, la normalización y el manejo cuidadoso de fechas y porcentajes, garantizan resultados correctos y robustos. Este proyecto demuestra la aplicabilidad de Java y Hazelcast para resolver problemas reales de Big Data en el contexto urbano.