POD - TPE2

Reclamos Urbanos

Grupo 2

Alumnos:

Liu, Jonathan Daniel 62533

Vilamowski, Abril 62495

Wischñevsky, David 62494

Profesores:

Turrin, Marcelo Emiliano

Meola, Franco Román

14 de Junio de 2025

Decisiones de diseño e implementación

En todas las *queries* se recibe como input un IMap de ids de reclamo y reclamo (*complaint*). Esto se hizo así para siempre proveer el *KeyValueSource* de la misma forma y de esta manera poder armar la lógica de los *jobs* de manera desacoplada del cliente. Esta abstracción se realizó en el método estático *run* de la clase *ClientUtils*. Esta se encarga de inicializar *HazelCast*, las conexiones y los mapas distribuidos para las *queries*. La función recibe como parámetros los nombres a utilizar para los recursos y un *QueryConsumer*, que es la función que realiza la *query*.

Para lograr la interoperabilidad con distintos formatos de archivo, se definió un archivo *ComplaintMappers*, en el que se establece cómo armar los reclamos de acuerdo al formato esperado por la ciudad indicada.

Para las *queries* que necesitan validar si el tipo de un reclamo es válido (está en el .csv de tipos) u obtener el nombre completo del tipo para mostrarlo, se proveyó el método estático *loadTypes* en la clase *ClientUtils*. Este agrega al contexto los tipos del csv de tipos, permitiendo que los *mappers* encargados de realizar este filtro puedan utilizarlos tanto como para filtrar como para traducir código (por ejemplo SR_SHORT_CODE en CHI) a tipo (SR_TYPE).
Para esto, se almacenaron los tipos en un diccionario distribuido *IMap<String, String>* que contiene como clave el campo que representa al tipo en los *.csv* de reclamos, y como valor la representación que se debe mostrar en el resultado final. Se consideró inicialmente utilizar *ISet* o *IList* para verificar la existencia de los tipos, pero debido a que los resultados sobre las queries de CHI en el enunciado muestran que se usa el SR_TYPE, se decidió trabajar con un mapa, que provee tanto la operación de contains como la de mapeo de código a descripción de manera eficiente. Este *"join"* (la sustitución del identificador de tipo en el *.csv* de reclamos por su versión representable) se hizo en los *mappers*.
Cabe destacar que el método *run* mencionado anteriormente se encarga también de destruir este mapa si fue inicializado en la *query*.

Query 1: Total de reclamos por tipo y agencia

Se solicita obtener, para cada tipo y agencia, la cantidad de reclamos.
Se definió el tipo serializable *TypeAgency* que funciona como tupla de tipo y agencia. Se implementó un *Job* con los siguientes pasos:

1. *TotalComplaintsByTypeAgencyMapper*: mapea los pares clave valor del id del complaint y el objeto complaint respectivamente, con un par clave valor del *TypeAgency* de ese reclamo con un *Long* (1L) para poder contarlo. Como se mencionó antes, ignora aquellos complaints cuyo tipo no esté en el mapa de tipos de reclamo permitidos y obtiene la representación completa del tipo.
2. *TotalComplaintsByTypeAgencyCombiner / Reducer*: agrupa por *TypeAgency* y suma de manera local/global los valores generados por el mapper.
3. *TotalComplaintsByTypeAgencyCollator*: ordena los resultados por los campos total descendentemente, tipo ascendentemente y agencia alfabéticamente.

Query 2: Tipo de reclamo más popular por barrio y cuadrante

Se solicita obtener, para un barrio y cuadrante, el tipo de reclamo más popular.
Se definió un tipo serializable *NeighborhoodQuad* que funciona como una tupla de barrio con su longitud, latitud y *NeighborhoodQuadType* que hereda de la anterior e incorpora el tipo de reclamo.
Se implementaron dos *Jobs* con los siguientes pasos:
***Job* 1**:

1. *NeighborhoodQuadTypeMapper*: mapea los pares clave valor del id del complaint y el objeto complaint respectivamente, con un par clave valor del *NeighbourhoodQuadType* de ese reclamo con un *Long* (1L) para poder contarlo.
   Como se mencionó antes, ignora aquellos reclamos cuyo tipo no esté en el mapa de tipos de reclamo permitidos y obtiene la representación completa del tipo.
2. *NeighborhoodQuadTypeCountCombiner/Reducer*: agrupa por *NeighbourhoodQuadType* y suma de manera local/global los valores generados por el *mapper*.

Este *job* genera como resultado un *IMap* intermedio con la suma de reclamos por barrio y lon/lat.

***Job* 2**:

1. *MostPopularTypeMapper*: una vez que ya se sabe la cantidad de reclamos de un tipo en un *NeighbourhoodQuadType,* se convierte el par clave valor en *NeighbourhoodQuad* junto a una tupla *TypeCount* que agrupa tipo y cantidad.
2. *MaxTypeCombiner/Reducer*: agrupa por *NeighbourhoodQuad* y hace un máximo de manera local/global sobre las tuplas retornadas por el *mapper*.
3. *NeighborhoodQuadTypeMaxCountCollator*: ordena los resultados alfabéticamente por agencia, ascendente por latitud y ascendente por latitud.

Query 3: Media móvil de reclamos abiertos por agencia, año y mes

Se solicita obtener, para cada agencia, año y mes, la media móvil dado una ventana de tiempo.
Se definió el tipo serializable *AgencyDate* que funciona como tupla de agencia, año y mes.
Se implementó un *Job* con los siguientes pasos:

1. *AgencyDateMovingAverageMapper*: Por cada par clave valor de id y complaint, y por cada mes que lo contiene en su ventana (los w-1 meses siguientes posibles), emite un par clave valor de *AgencyDate* de ese reclamo y ese mes, y un *Long* (1L) para poder contarlo. De esta manera, se pueden agrupar las ventanas en el siguiente paso.
2. *AgencyDateMovingAverageCombiner / Reducer*: agrupa por *AgencyDate* y suma de manera local/global los valores generados por el mapper. El *Reducer* además hace la división por la cantidad de meses de la ventana. Se decidió realizar el truncamiento a dos dígitos en este paso ya que se consideró parte del dominio y no simplemente una cuestión de representación. En otra implementación esto podría haberse hecho en el collator al generar el DTO o incluso en el cliente al momento de mostrarlo.

3. *AgencyDateMovingAverageCollator*: ordena los resultados por los campos agencia, año y mes ascendentemente.

Query 4: Porcentaje de tipos de reclamo por calle

Se solicita obtener el porcentaje de reclamos por calle. Para esto se calcula el cociente entre los tipos de reclamos distintos en una calle y el total de reclamos distintos de todas las calles de la ciudad.
Se definió el tipo serializable *TypeStreet* que funciona como tupla de tipo de reclamo y calle.

Tanto el *job* 1 como el *job* 2 reciben el *IMap* de ids de reclamo y reclamo mencionado anteriormente.

***Job* 1**:

1. *TotalTypeCountMapper*: por cada par clave valor de id y reclamo, se emite un 1 para cada tipo, en caso de que el tipo esté incluido dentro del mapa de tipos disponibles. A diferencia de otros mappers, se devuelve un *Integer* en vez de un *Long*, ya que no se espera que la cardinalidad de los tipos de reclamo sea tan elevada como la de los reclamos en sí.
2. *TotalTypeCountCombiner/Reducer*: agrupa por tipo, procurando que quede una sola tupla con 1 por cada tipo incluido.
3. *TotalTypeCountCollator*: cuenta las tuplas retornadas

Como resultado de este *job* se obtiene la cantidad total de tipos de reclamo reales globales (es decir, que no solo aparecen en los archivos de tipos, sino que efectivamente hay reclamos en la ciudad que los referencian).

***Job* 2**:

1. *TypeStreetMapper*: similar a *TotalTypeCountMapper*, pero además de filtrar por los tipos disponibles, también filtra por el argumento de barrio.
2. *TypeStreetUniqueCombiner/Reducer*: agrupa por *TypeStreet*, de manera similar al *TotalTypeCountCombiner/Reducer* del *Job* 1.

Como resultado del *job*, se obtienen los *TypeStreet* sin repeticiones.

Para el último *job*, se utiliza como *KeyValueSource* el output producido por el *job* 2
***Job* 3**:

1. *StreetMapper*: por cada par de *TypeStreet* y el *Integer* 1, devuelve el par clave valor formado por la calle del *TypeStreet* y el 1 casteado a *Long*, para poder hacer la suma.
2. *TypePercentageByStreetCombiner*: agrupa por calle y suma los valores parciales
3. *TypePercentageByStreetReducer*: agrupa por calle y calcula el promedio truncado haciendo la suma global y la división por el total (calculado en el *job* 1).
   Nuevamente, se podría haber hecho el redondeo en el collator, pero se decidió manejarlo como parte del dominio de los datos.
4. *TypePercentageByStreetCollator*: ordena los resultados de manera descendente por porcentaje, luego alfabético por calle

Análisis de tiempos

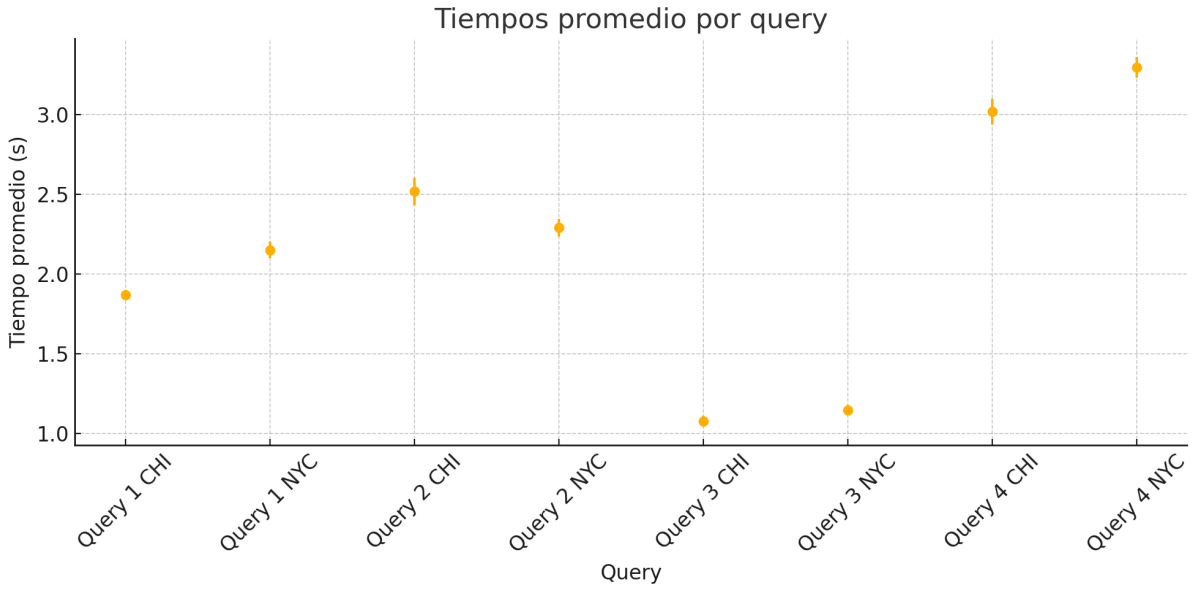
Un solo nodo

100.000 filas

Las pruebas se realizaron con la herramienta *hyperfine* que permite calcular promedios y desvíos estándar de múltiples ejecuciones de un comando. Para la comparación, considerar despreciables los tiempos de conexión inicial.

Se hacen 10 pruebas con las primeras 100.000 filas de los archivos provistos por la cátedra. (99.999 reclamos)

	Promedio	Desvío estándar	Min	Max
Query 1 CHI	1.869 s	0.031 s	1.824 s	1.936 s
Query 1 NYC	2.151 s	0.052 s	2.072 s	2.215 s
Query 2 CHI	2.518 s	0.088 s	2.445 s	2.718 s
Query 2 NYC	2.291 s	0.055 s	2.199 s	2.377 s
Query 3 CHI	1.077 s	0.035 s	1.006 s	1.143 s
Query 3 NYC	1.146 s	0.032 s	1.096 s	1.205 s
Query 4 CHI	3.019 s	0.081 s	2.907 s	3.104 s
Query 4 NYC	3.299 s	0.065 s	3.212 s	3.395 s



Como se mencionó antes, las *queries* 1, 2 y 4 hacen uso del *IMap* de tipos para realizar el filtrado. Es posible que esta necesidad de mantener estructuras compartidas en un entorno distribuido introduzca *overhead* de red y sincronización, afectando directamente la performance.

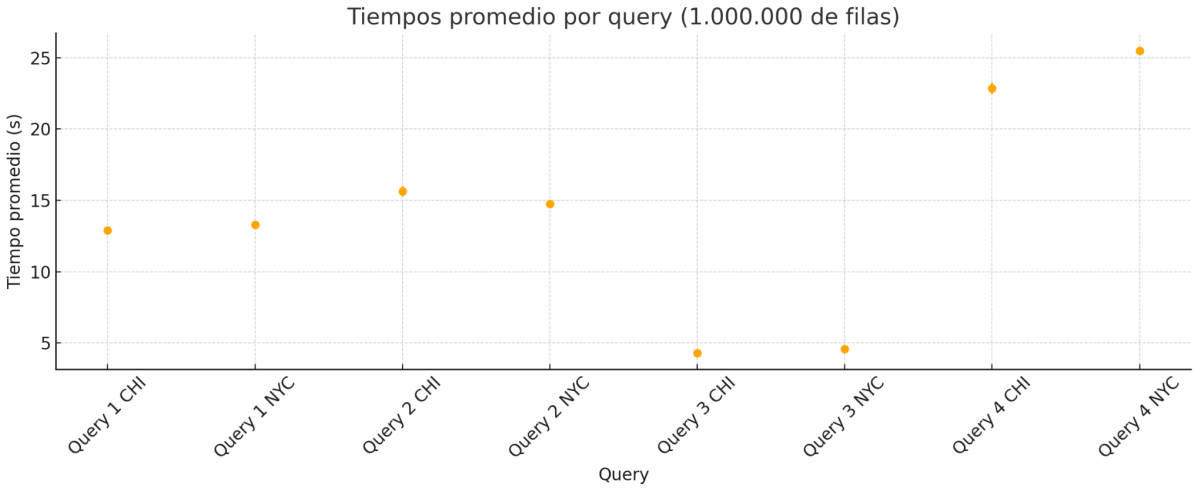
La *query* 3, en cambio, evita este *overhead* ya que no utiliza mapas distribuidos, y además aplica un filtro verificando si el estado del *complaint* está marcado como “Open”, lo que permite descartar una gran cantidad de registros antes del procesamiento y reduce así el volumen de datos que participa en el *MapReduce*.

Finalmente, las *queries* 2 y 4 son también más costosas porque requieren varias etapas de *MapReduce* consecutivas, lo que implica recorrer los datos más de una vez, generar estructuras intermedias y aumentar el uso de la red, intensificando el costo computacional general.

1.000.000 filas

Se hacen 10 pruebas con las primeras 1.000.000 filas de los archivos provistos por la cátedra (999.999 reclamos).
Para la *query* 2 se usó q=0.1, para la *query* 3 w=3, y para la *query* 4 neighbourhood=MANHATTAN (NYC) y LOOP (CHI).

	Promedio	Desvío estándar	Min	Max
Query 1 CHI	12.896 s	0.171 s	12.704 s	13.211 s
Query 1 NYC	13.289 s	0.148 s	13.035 s	13.510 s
Query 2 CHI	15.653 s	0.350 s	15.148 s	16.479 s
Query 2 NYC	14.747 s	0.221 s	14.518 s	15.307 s
Query 3 CHI	4.282 s	0.046 s	4.189 s	4.342 s
Query 3 NYC	4.567 s	0.093 s	4.471 s	4.691 s
Query 4 CHI	22.859 s	0.399 s	22.388 s	23.562 s
Query 4 NYC	25.503 s	0.172 s	25.278 s	25.723 s



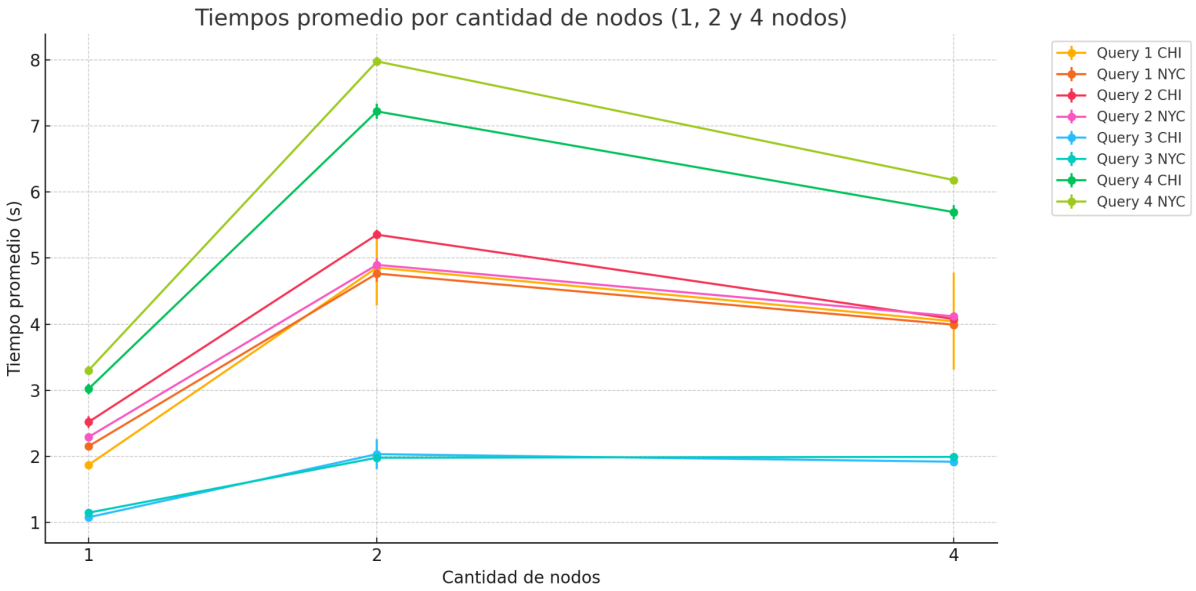
Al escalar a un volumen de 1.000.000 de registros, los tiempos de ejecución aumentan, como era esperable, aunque se conserva el comportamiento relativo entre las distintas *queries* ya observado con menor cantidad de datos.

Múltiples nodos (en containers de docker)

Para poder simular una red con múltiples nodos, se utilizó docker compose. Se armó un *virtual bridge* a través del cual los contenedores que corren instancias de server pueden comunicarse entre sí y con el *host*.
A continuación se presentan los tiempos tomados al correr 100.000 filas con 2 y 4 contenedores, con y sin combiners, en 10 corridas.

	Cantidad de nodos	Promedio (combiner)	Promedio (sin combiner)
Query 1 CHI	2	4.648 s ± 0.166 s	4.856 s ± 0.576 s
Query 1 CHI	4	4.059 s ± 0.088 s	4.044 s ± 0.733 s
Query 1 NYC	2	4.772 s ± 0.080 s	4.766 s ± 0.120 s
Query 1 NYC	4	3.778 s ± 0.030 s	3.993 s ± 0.044 s
Query 2 CHI	2	5.321 s ± 0.105 s	5.355 s ± 0.070 s
Query 2 CHI	4	4.356 s ± 0.073 s	4.078 s ± 0.041 s
Query 2 NYC	2	5.063 s ± 0.073 s	4.898 s ± 0.097 s
Query 2 NYC	4	3.901 s ± 0.037 s	4.119 s ± 0.052 s
Query 3 CHI	2	1.976 s ± 0.049 s	2.033 s ± 0.228 s
Query 3 CHI	4	1.944 s ± 0.021 s	1.918 s ± 0.031 s
Query 3 NYC	2	2.022 s ± 0.026 s	1.979 s ± 0.026 s
Query 3 NYC	4	1.995 s ± 0.026 s	1.992 s ± 0.035 s

Query 4 CHI	2	7.608 s ± 0.135 s	7.221 s ± 0.115 s
Query 4 CHI	4	5.396 s ± 0.032 s	5.696 s ± 0.109 s
Query 4 NYC	2	8.085 s ± 0.092 s	7.978 s ± 0.066 s
Query 4 NYC	4	5.982 s ± 0.055 s	6.181 s ± 0.053 s



Se puede observar que contra lo esperado, aumentar la cantidad de nodos a 2 o 4 en ningún caso resulta mejor que trabajar con un solo nodo.

Sin embargo, al comparar las de 2 nodos con las de 4, se nota que se logran reducir los tiempos de ejecución a medida que se incrementa la cantidad de nodos, manteniéndose el comportamiento relativo entre *queries* ya observado previamente.

La *query* 3 continúa siendo la más rápida, con diferencias mínimas al escalar entre 2 y 4 nodos, lo cual confirma su bajo costo computacional.

Con relación al uso de *combiners*, según lo visto en la teórica, estos deberían mejorar la performance, ya que realizan agregaciones parciales de los datos en los nodos, disminuyendo el tráfico sobre la red. Sin embargo, los datos medidos no logran dejar en evidencia que provoquen una mejora significativa de los tiempos. Si bien en algunos casos, los tiempos fueron mejores al usar *combiners*, en muchos otros no usarlos resultó igual o más rápido.

Múltiples nodos (en hosts físicos)

Se intentó realizar pruebas con hasta 3 computadoras en una misma red. Para esto fue necesario configurar en el servidor el discovery TCP/IP, además del multicast que se venía usando.
Dado que la comunicación se hizo por Wifi (que es un medio con mucha pérdida de paquetes y latencia), los tiempos en general no fueron buenos.

De todas formas, realizar la experiencia con computadoras reales podría considerarse equivalente a un entorno dockerizado como el de la sección anterior, siempre y cuando:

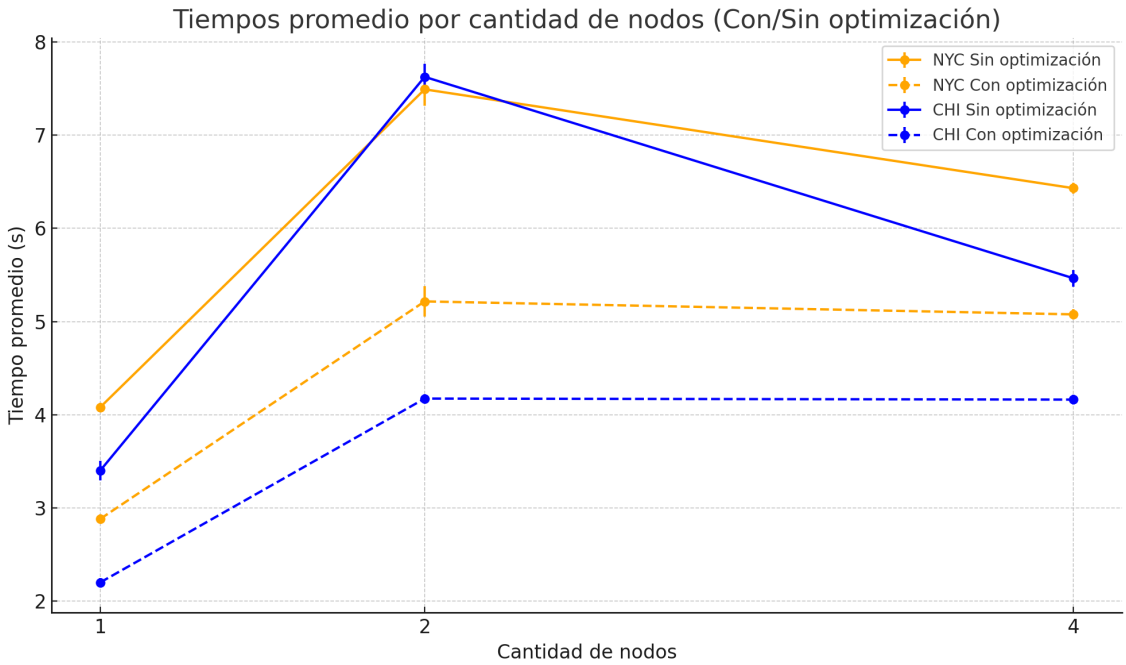
- En el entorno dockerizado, la computadora host tenga suficientes recursos de memoria y CPU.
- En el entorno real se utilice algún medio más estable como puede ser un cable ethernet.

Optimización de la query 4

Dado que los jobs 1 y 2 de la query 4 son independientes, estos pueden ser ejecutados en paralelo. De esta manera, si se hace el *get* bloqueante del *CompletableFuture* asociado al primer job después de iniciar el segundo, se puede lograr un aumento de performance, ya que se evita tener que esperar a que termine el job 1 para empezar el 2.

Estas pruebas se ejecutaron sobre 100.000 filas, con 10 corridas, utilizando docker compose sobre una computadora distinta a las anteriores.

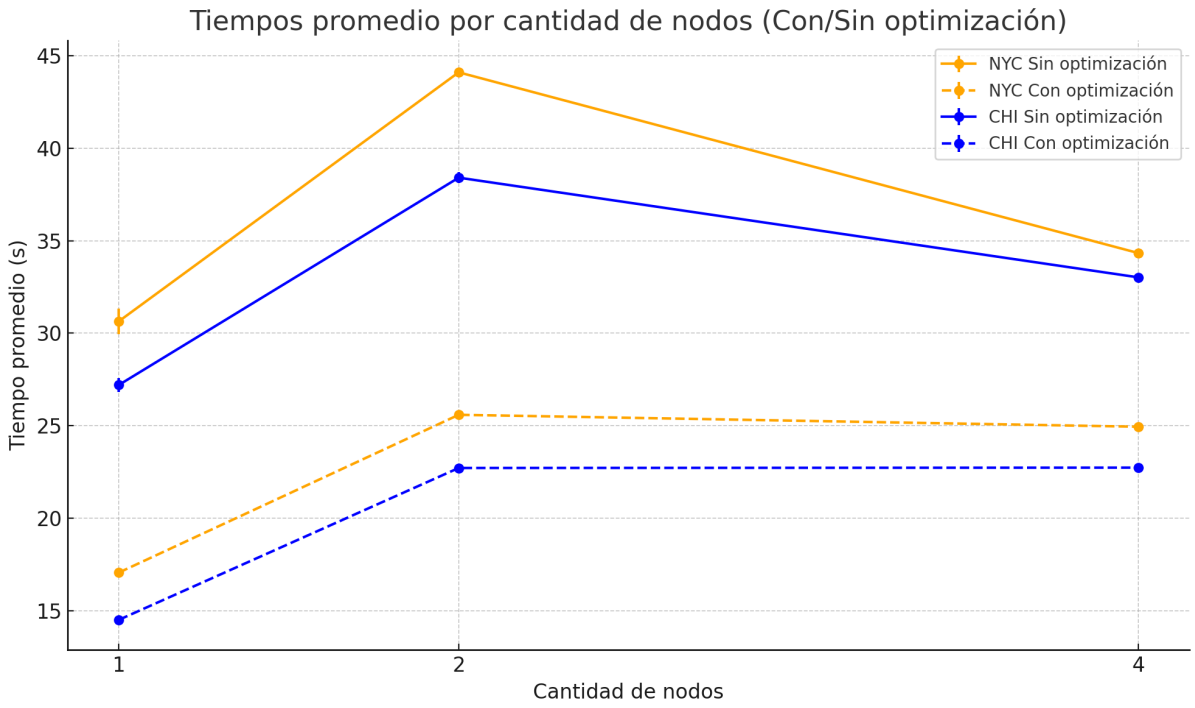
	Cantidad de nodos	Sin optimización	Con optimización
NYC	1	4.082 s ± 0.047 s	2.884 s ± 0.049 s
NYC	2	7.490 s ± 0.178 s	5.216 s ± 0.164 s
NYC	4	6.430 s ± 0.058 s	5.077 s ± 0.053 s
CHI	1	3.403 s ± 0.103 s	2.203 s ± 0.044 s
CHI	2	7.624 s ± 0.139 s	4.174 s ± 0.048 s
CHI	4	5.464 s ± 0.088 s	4.163 s ± 0.053 s



Se puede observar que para ambas ciudades la versión optimizada es más rápida.

Se realizaron las mismas mediciones para los archivos con 1.000.000 filas:

	Cantidad de nodos	Sin optimización	Con optimización
NYC	1	30.644 s ± 0.693 s	17.059 s ± 0.106 s
NYC	2	44.103 s ± 0.234 s	25.589 s ± 0.203 s
NYC	4	34.330 s ± 0.216 s	24.944 s ± 0.120 s
CHI	1	27.202 s ± 0.384 s	14.504 s ± 0.127 s
CHI	2	38.408 s ± 0.293 s	22.714 s ± 0.185 s
CHI	4	33.018 s ± 0.225 s	22.733 s ± 0.121 s



Las mediciones resultaron en un patrón muy similar, donde la versión optimizada es más rápida que la no optimizada.

Puntos de mejora

Un posible punto de mejora es realizar una abstracción para algunos *reducers* o *combiners* similares, en versiones genéricas. Por ejemplo, *NeighborhoodQuadTypeCountReducer* y *TotalComplaintsByTypeAgencyReducer*, que lo que hacen es agrupar y hacer la suma de sus valores, lo que los distingue son el tipo de clave con el que agrupan. Se podría definir un *SumReducer<T>* que agrupa por el tipo *T* y hace la suma. Esto se puede aplicar también para los respectivos *combiners*.
También se puede hacer una abstracción para *TypeStreetUniqueReducer*,

TotalTypeCountReduce, definiendo un *UniqueReducer<T>* que asegura que quede una sola tupla por cada clave. Esto también se puede aplicar con sus *combiners* respectivos.

Otro posible punto de mejora se puede hacer en el proceso de lectura de los archivos. En la clase CsvManager, tanto el método *readFile* como *writeLines* utilizan *streams* secuenciales para procesar los datos. Dado que cada línea puede ser leída, transformada y consumida de forma independiente, podría explorarse el uso de parallel() para acelerar la lectura, especialmente en archivos grandes. Esto permitiría distribuir el trabajo de parseo y consumo.
En un principio se intentó hacer una implementación de este estilo, pero debido a que el tiempo de ejecución de la implementación realizada no daba muestras de ser superior a la versión secuencial (que sube los reclamos de a batches), se decidió mantener la implementación actual.
Del mismo modo, en la escritura si bien se requiere más precaución debido al acceso concurrente al *BufferedWriter*, podría evaluarse una estrategia en paralelo si se implementa un mecanismo de sincronización adecuado.