Informe TPE2 - Grupo 7
Programación de Objetos Distribuidos - 72.42

Comisión S

Tobias Perry (62064), ... tperry@itba.edu.ar
Manuel Esteban Dithurbide (62057), ... mdithurbide@itba.edu.ar
Christian Ijjas (63555), ... cijjas@itba.edu.ar
Luca Seggiaro (63855), ... lseggiaro@itba.edu.ar

28 de Abril de 2024

Contents

1 Introducción

Este informe tiene como objetivo exponer algunas de las decisiones a la hora de implementar el TPE2. Se intentará explicar de manera concisa las decisiones de diseño respecto al trabajo **MapReduce**, y los distintos aspectos de Hazelcast involucrados. Utilizaremos gráficos para demostrar el impacto de estas.

2 Decisiones de diseño y Análisis de tiempos

2.1 MapReduce

A continuación, se mencionarán las decisiones más relevantes a la hora de implementar los trabajos MapReduce. En líneas generales, se intentó mantener un enfoque simplista. Los Mappers deben definir qué dato se desea utilizar como valor, buscando poder representarlo como una variable primitiva, para que sea lo más liviano posible. Por otro lado, buscamos que el combiner y reducer tengan el mismo objetivo, que busquen hacer "lo mismo". Finalmente, el Collator se encargará de ordenar los datos, y aplicar alguna condición adicional, como establecer un límite para los datos. Esta condición **NO** debe ser de lógica que pueda ser calculada previamente.

Si uno mira el código, podrá observar que los Collators presentan la mayor cantidad de lógica. Esto se debe a que, las otras tres etapas realizan operaciones simples, mientras que esta será encargada de ordenar el mapa, o asociar la clave de la infracción con la descripción.

Si bien la mayoría de las queries se solucionan con un único *job*, la query 5 es la excepción. Dada la complejidad de la query fue necesario dividir en dos *jobs* distintos. Uno se encarga de calcular el promedio de las multas y otro las agrupa como es especificado en la consigna.

En el primer trabajo, nos encontramos con un desafío al calcular el promedio. Necesitábamos emitir un par clave-valor de código-monto para cada código de infracción. Sin embargo, esta implementación presentaba un problema: no podíamos calcular el promedio durante la etapa de combinación porque no teníamos la cantidad total de elementos. Para resolver esto, decidimos emitir un par concatenado de código-'monto,count'. De esta forma, al realizar un *split* con una coma, podemos recuperar en cualquier momento la cantidad de ítems sobre los que se calculó el promedio.

Finalmente un único collator se encarga de juntar los pares y presentarlos como fue indicado.

2.2 Combiner

A partir de las sugerencias de la cátedra, se decidió medir el impacto del Combiner a la hora de realizar un trabajo MapReduce.

Query 3 - Uso de Combiner		
	Presente (s)	Ausente (s)
1	6,1	7,5
2	5,5	5,8
3	6,3	5,8
4	6,5	5,7
5	6,2	6,3
promedio	6,12	6,22

Figure 1: Comparación en el uso del Combiner, query 3, 15 millones de datos, 1 nodo.

Observando los tiempos promedios, no podemos establecer que haya una diferencia significativa entre los valores, considerando el bajo tamaño de la muestra. Sin embargo, cabe destacar que se estaba trabajando sobre un único nodo, con 10 GB de RAM. Consideramos que el Combiner se destaca a la hora de tener múltiples nodos, ya que (teóricamente) agrupar los datos localmente reduce la carga de información que se deberá transferir entre estos. Nos hubiera gustado evaluar este caso con el objetivo de proveer datos más aplicables a la programación de objetos distribuidos.

2.3 IMap vs IList

En cuanto a la estructura que se usa para cargar los datos a Hazelcast, inicialmente nos pareció que usar un `IList` era lo correcto ya que hay tickets repetidos, y no encontramos ninguna justificación para usar un mapa sobre ello. Posteriormente, leyendo la documentación, nos dimos cuenta que `IList` no es *partition-tolerant* incumpliendo con nuestro objetivo de instanciar múltiples nodos. Por esta razón, decidimos utilizar `IMap` donde le asignamos una clave generada por nosotros y el valor es el tícket. Si bien esto es lo que ocurrió a nivel superficial, nos llevó mucho tiempo tomar la decisión de qué estructura usar ya que, al utilizar un único nodo, la implementación de `IList` es ampliamente más rápida que un `IMap`.

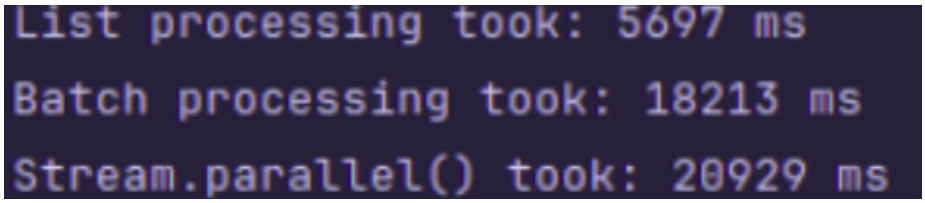

Figure 2: IList vs IMap en lotes vs IMap en stremas para la carga de 1 millón de tickets en un nodo

También se experimentó con `MultiMap`, quien presentó resultados similares a `IMap`. Con el objetivo de realizar una implementación eficiente en cualquier caso de uso, intentamos considerar cuántos nodos están corriendo a la hora de correr el trabajo MapReduce para definir qué estructura utilizar. Si bien sabemos cómo verificar dicho valor, debido a nuestra decisión inicial de estructura de clases, no era posible determinar la estructura sin hacer un replanteamiento del flujo del cliente.

2.4 Lectura del CSV

Para la lectura del archivo de tickets se hace uso del patrón adapter (un adapter para cada caso de archivo tickets) con el objetivo de garantizar que futuras implementaciones se manejen con facilidad a la hora de integrar su archivo específico de tickets. Con el archivo de infracciones fue innecesario para el caso particular pero se haría de forma semejante en caso de necesitarlo.

Se buscó analizar el parser CSV con el objetivo de reducir la carga de datos dentro de lo posible. Se probaron varias técnicas y se terminó optando por usar una librería (Univocity [1]) que según nuestras pruebas resultó en los mejores tiempos (ver https://github.com/uniVocity/csv-parsers-comparison). También se aplicaron técnicas de concurrencia utilizando `ConcurrentHashMap` y threads. De esto surgió un análisis de tamaños de lote donde analizamos el tamaño adecuado.

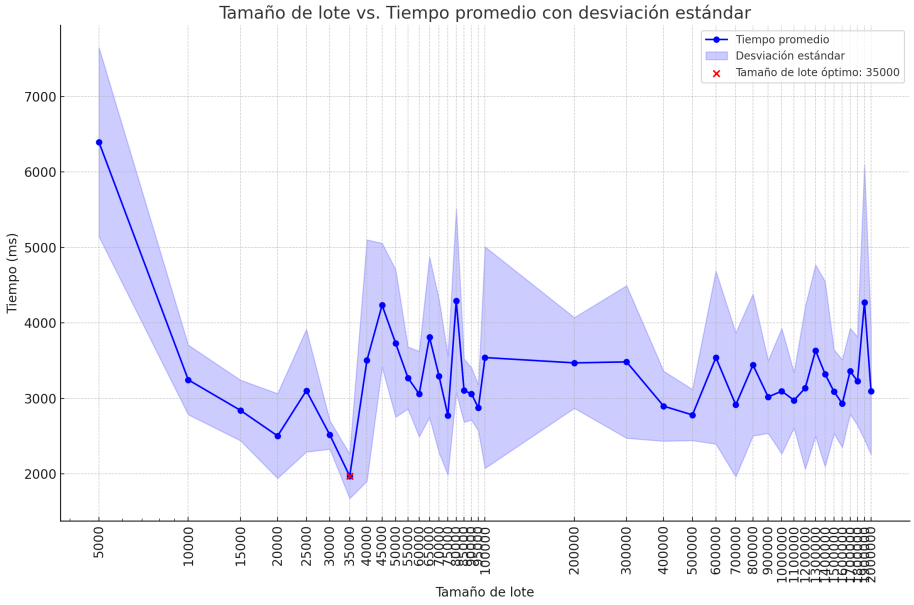

Figure 3: Tamaño de lote vs tiempo promedio para la lectura y carga a Hazelcast de 1 millón de registros de tickets en 1 nodo. (5 iteraciones por tamaño de lote)

Cabe destacar que este valor puede variar entre dispositivos, sería óptimo hacer una prueba si se desea implementar un sistema que haga uso constante de este proceso.

2.5 Serialización de los datos

Respecto a la serialización de los datos, inicialmente arrancamos con la interfaz `Serialized` de java la cual no presentaba muchos defectos hasta que (gracias a los comentarios en [2]) probamos con `DataSerialized` de Hazelcast y nos dimos cuenta de lo mucho más rápido que era. Probamos usar `Portable` en la misma manera (siendo más rápido que `Serialized`) pero igual no lograba alcanzar la velocidad de DataSerialized.

También optamos por algunas pequeñas optimizaciones locales del objeto serializado como usar primitivas en vez de objetos tipo `LocalDate` para el proceso de escritura y lectura.

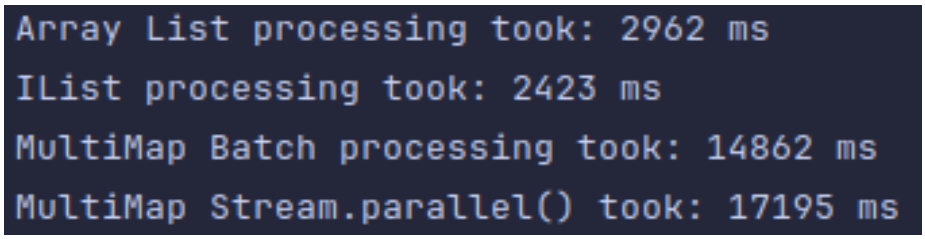

Figure 4: Tiempos de carga sobre listas y mapas haciendo uso de DataSerialized para 1 millón de registros

Se puede observar, comparando con la figura 2 en 2.3, la disminución de tiempo significativa para la carga de las estructuras a Hazelcast.

2.6 Otras

Con el objetivo de reducir el tiempo de generación de resultados para las queries, se optó por realizar el filtrado de datos desde el cliente. Si bien consideramos que sería mejor realizar el filtrado de manera distribuída, consideramos que dichos filtros son operaciones poco costosas, y justificamos que el cliente tendrá que cargar los datos igual, y puede decidir qué datos cargar. Dichos filtros son comparar la fecha con los valores introducidos por argumento, o verificar que exista la infracción, recorriendo un mapa de 60 valores, con complejidad O(1) (si está bien hasheado).

Acompañando esta decisión, también delegamos al cliente guardarse dichas infracciones, y enviarselas al collator para que este mapee el código a la descripción.

Para el caso de la query 3, se optó por calcular el total en el collator. Si bien sabemos que este paso ocurre en el lado del cliente, consideramos que la lógica adicional de incluirlo en el mapa y después filtrar sería más costoso.

Query 3 - Total en el mapper vs calc en el collator		
	Collator (s)	Mapper (s)
1	6,6	5,7
2	6,2	6,2
3	6,1	7,1
4	6,2	5,9
5	5,4	5,9
promedio	6,1	6,16

Figure 5: Cálculo del total en el collator vs el mapper. Query 3, 15 millones de datos, 1 nodo, 10 GB de RAM.

Después de comparar resultados, podemos concluir que no presenta diferencias significativas en el entorno experimentado. Se espera que la diferencia sea más clara en un cliente con menor poder de cómputo. De todas formas, creemos que seguirá siendo irrelevante ya que la cantidad de elementos de salida es baja, como máximo la cantidad de agencias presentes.

3 Métodos de conexión y múltiples nodos

Con el objetivo de probar múltiples nodos en una red, experimentamos distintas alternativas para ver cómo estas impactaban el rendimiento de la ejecución de las queries. En primer lugar, utilizamos Hamachi, una herramienta que utiliza un formato VPN para simular que los usuarios estaban en una red local. Posteriormente, inicializamos 3 nodos distintos en una misma red, vía Wi-Fi. Los resultados dejaron **mucho** por desear:

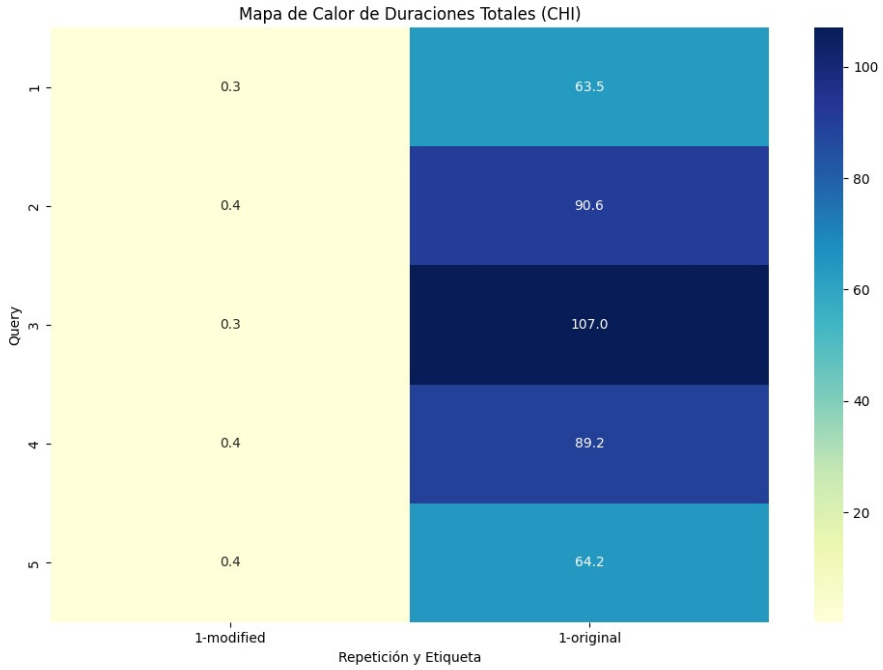

Figure 6: Mapa de calor sobre los tiempos de ejecución, local con 1 nodo (modified) vs wifi con 2 nodos (original). 10,000 datos.

Como se podrá observar, la red por la que transitaban los datos trabó por completo su ejecución, resultando en un 30000% de pérdida de rendimiento para el peor caso. Existen algunos factores a considerar, como reducir el tamaño del objeto a serializar, y mandar solo los campos relevantes para cada query, pero no consideramos que sean la raíz de este problema. Un aspecto a destacar fue que la carga sobre la RAM de las computadoras fue mejor distribuida. Finalmente, notamos que la latencia en red era muy alta, lo que se puede atribuir a factores como que Hazelcast intenta mantener consistencia y sincronización entre los nodos en un ambiente poco favorable.

Sin tener acceso a una red cableada, nos pusimos a analizar lo sucedido. Las conclusiones fueron que:

1. Para baja cantidad de registros no se vería una mejora significativa en aumentar los nodos, contrario a lo que sí ocurriría si la cantidad de registros fueran muchos (a partir de los primero millones). Con estas cantidades las computadoras podrían distribuir equitativamente la carga con menor probabilidad de cuellos de botella en un solo recurso y procesar mejor la información. Obteniéndose resultados de manera más rápida (más aún con un Combiner previo al Reducer).

2. También se vería afectada la performance por la introducción de proce-

samiento verdaderamente paralelo.

3. Se cree que no hay mucho beneficio en escalar a más de 5 nodos ya que se tornaría en una alta complejidad de configuración y sincronización entre nodos.

4. Un beneficio adicional de aumentar el número de nodos es una mayor redundancia y tolerancia a fallos. Esto significa que mientras más nodos tengas, mejor podrá manejar el sistema la pérdida de uno o más nodos sin afectar significativamente el rendimiento global.

4 Puntos de mejora y/o expansión

4.1 Estructuras condicionales

Como fue mencionado previamente en 2.3, deseamos poder utilizar un `IMap` o un `IList` dependiendo de la cantidad de nodos a la hora de correr un MapReduce. Para lograr esto, deberíamos reestructurar nuestros clientes nuevamente, ya que el uso de *generics* y *wildcards* no alcanzan para implementar esta funcionalidad.

4.2 *Preload* y *Caching*

Otro potencial punto de expansión es considerar los clientes que desean correr múltiples queries sobre el mismo set de datos. El tiempo de carga del csv con 15 millones de datos a los nodos es significativo, por lo que nos gustaría poder correr todas las queries sobre un mismo grupo de datos, en vez de cargarlos y limpiarlos cada vez que se hace una consulta. Esto afectaría nuestra decisión de que el cliente filtre los datos que carga, el servidor debería encargarse de este paso posteriormente.

4.3 Testeo

Relacionado al punto anterior, nos gustaría generar una estructura de testeo más organizada. Escribimos unos scripts en python y bash que nos permitieron correr continuamente distintos casos de uso, para generar resultados más consistentes, pero cada uno revisó lo que le parecía pertinente, y nos podemos haber perdido algún caso. El punto anterior ayudaría mucho este proceso, ya que se podría cargar una única vez los datos y enfocarse en el MapReduce.

4.4 Serialización

Finalmente, consideramos optimizar el método de serialización (discutido en 2.5) de cada query, teniendo en cuenta que estas suelen pedir sólo algunos de los datos. No decidimos hacer esto porque consideramos que sería ajustar mucho las cosas al funcionamiento exacto de una query en particular y no de un sistema más interoperable.

5 Bibliografia

References

[1] Comparación de parsers CSV, https://github.com/uniVocity/csv-parsers-comparison

[2] Serialización Hazelcast, https://stackoverflow.com/questions/37432721/best-way-to-bulk-load-data-in-hazelcast

[3] Collections de Hazelcast, https://docs.hazelcast.org/docs/latest/javadoc/com/hazelcast/collection/package-summary.html

[4] Combiner de Hazelcast, https://docs.hazelcast.org/docs/3.8.6/javadoc/com/hazelcast/mapreduce/Combiner.html

[5] MapReduce de Hazelcast, https://docs.hazelcast.org/docs/3.8.6/manual/html-single/index.html#mapreduce

[6] Jobtracker de Hazelcast, https://docs.hazelcast.org/docs/3.8.6/javadoc/com/hazelcast/mapreduce/JobTracker.html

[7] Jobtracker de Hazelcast, https://docs.hazelcast.org/docs/3.8.6/javadoc/com/hazelcast/mapreduce/JobTracker.html

[8] Mapper de Hazelcst, https://docs.hazelcast.org/docs/3.8.6/javadoc/com/hazelcast/mapreduce/Mapper.html

[9] Reducer Factory de Hazelcast, https://docs.hazelcast.org/docs/3.8.6/javadoc/com/hazelcast/mapreduce/ReducerFactory.html

[10] Reducer de Hazelcast, https://docs.hazelcast.org/docs/3.8.6/javadoc/com/hazelcast/mapreduce/Reducer.html

[11] Collator de Hazelcast, https://docs.hazelcast.org/docs/3.8.6/javadoc/com/hazelcast/mapreduce/Collator.html