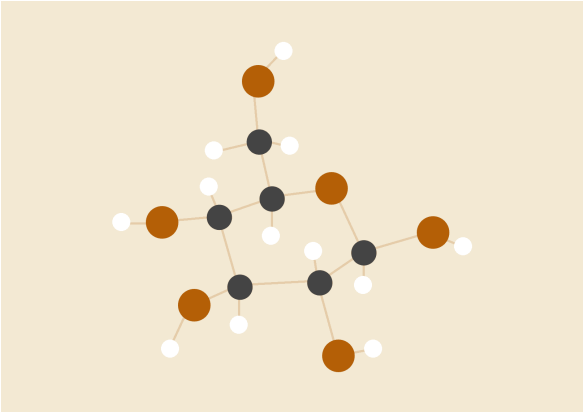
TPE 2: Peatones

Grupo 8

2022 2Q



Fecha de entrega: 12/10/2022

Docentes:

Turrin, Marcelo Emiliano (Responsable)
Meola, Franco Román (Adjunto)

Integrantes:

Gonzalo Rossin - 60.135
Jerónimo Brave - 61.053
Mateo Bartellini Huapalla - 61.438
Juan Ignacio Garcia Matweiszyn - 61.441
Juan Manuel Negro - 61.225
Mauro Daniel Sambartolomeo - 61.279

Índice

DECISIONES DE DISEÑO

En primer lugar, podemos definir la estructura principal del proyecto, que se separa en 3 secciones. La primera se debe al servidor que en nuestro caso son los nodos de *Hazelcast* y por lo tanto no es más que una clase que configura e inicializa un nodo, que puede ser llamada todas las veces requeridas para generar múltiples nodos.

La segunda es la *API* que contiene información compartida por las otras dos secciones. En la *API* definimos modelos, para conceptualizar las distintas entidades a administrar según el paradigma de la programación orientada a objetos, se decidió utilizar una lista para almacenar las lecturas de los sensores y un mapa para almacenar la información respectiva a cada sensor. También, todas las entidades participantes del proceso de *MapReduce* como *Mappers*, *Reducers*, *Collators*, etc, fueron guardados en esta sección ya que estos tienen que ser generados por el cliente pero utilizados por los nodos y por lo tanto todos necesitan acceso.

Por último también la sección del cliente que es donde definimos las diferentes consultas, leemos los archivos de entrada e invocamos el proceso *MapReduce* en los nodos *Hazelcast* con la información. Luego además creamos los archivos de salida antes de finalizar. Para la implementación de esto se creó una clase *Query* que engloba el comportamiento general de las consultas, como conexión al cluster, importación de archivos, creación de archivo de salida, etc. Luego cada consulta particular, se define el proceso MapReduce correspondiente y se imprime al archivo creado la salida.

Para la lectura e importación de archivos a Hazelcast se tomó la decisión de realizarlo de a batches de 500.000 entradas de csv por un tema de performance y de memoria del cliente. Esta decisión fue tomada ya que, por un lado, si se leyera e importara todo el csv de una, el cliente debería cargar en memoria todos los datos lo cual puede causar un que este se quede sin memoria y por el otro, si se leen e importan uno a uno es necesario crear una conexión con hazelcast por línea y hazelcast debe encargarse de replicar todo en cada inserción, lo cual es muy lento. Se tomó el número 500.000 en base a prueba y error de que número era el más performante.

DISEÑO DE COMPONENTES PARA *MAPREDUCE*

Query 1:

Para la etapa de *mapeo* se procedió a tomar cada lectura y evaluar si la lectura procedía de un sensor registrado al contrastar el *ID* del sensor en el *reading* con el mapa de sensores y posteriormente si el estatus del sensor en la lectura era activo. De no cumplirse estas dos condiciones se decidió no emitir. Al momento de emitir, se decidió emitir un par clave valor en el cual la clave fuera la descripción del sensor y el valor, el conteo de personas registrado por la lectura.

Por último, el *Reducer* se ocuparía de sumar los conteos de personas para cada sensor y se mandaría a un collator con el fin de realizar el ordenamiento final de valores.

Alternativas de diseño con las que se experimentaron fueron la de agregar un Combiner para reducir el tiempo de ejecución de la operación MapReduce pero fue descartado ya que no solo no redujo el tiempo de ejecución sino que lo aumentó.

Query 2:

Este trabajo es el único que no requiere de información de los sensores, por lo que sólo se enfoca en las mediciones de por sí. Con el objetivo de agrupar las mediciones por año, por cada medición se emite como llave el año de la medición en sí, y como valor se emite un par consistiendo del valor medido y un *boolean* indicando si el día de la medición fue día de semana laboral o del fin de semana.

Por otro lado, se decidió incluir un *Combiner* por la cantidad de pares *key-value* emitidos con misma llave, pues el año de la medición se repite muchas veces. Este *Combiner* evalúa si sumar la medición a un contador de día de semana laboral o de fin de semana, abarcando la lógica principal de la *Query*, y devolviendo un par de valores enteros que, finalmente, el *Reducer* tomará y sumará entre todos los valores devueltos por el *Combiner*.

Query 3:

Para esta *query* decidimos trabajar, además de con las funciones de *map* y *reduce*, con un *collator*. El objetivo de la *query* es encontrar la máxima medición de cada sensor que supere a un cierto valor de umbral, decidido por el cliente. Para esto, el *mapper* recibe por constructor dicho valor de umbral, y sólo emitirá en el método *map* aquellos valores que superen el umbral.

Una vez terminado el *map*, se pasa al *reduce* que lo que hace es, por cada clave (que a

partir del map es el nombre del sensor), y si el valor de dicha clave es mayor a los anteriores lo guarda como tal. Por último pasamos todo por un *collator* para poder ordenar los resultados en la forma pedida.

Cómo última consideración, no utilizamos para resolver esta query un *combiner*, ya que el *mapper* emite únicamente una o ninguna clave.

Query 4:

En esta query se debió utilizar una cota n para la cantidad de respuestas y un año específico del cual me pedían los máximos promedios de sensores por mes. Para esto, primero mapeamos los sensores del año "year" pedido, con el esquema clave: nombre o descripción de sensor y valor: mes y peatones. Luego, al reducir se calcula para cada sensor un mapa de los meses que tiene y sus promedios, para luego quedarnos con el máximo al finalizar.

Al momento de dar la respuesta, tomamos el valor "n" de la cota inicial para únicamente retornar los primeros "n" sensores de ese año que cumplen la condición de mes máximo.

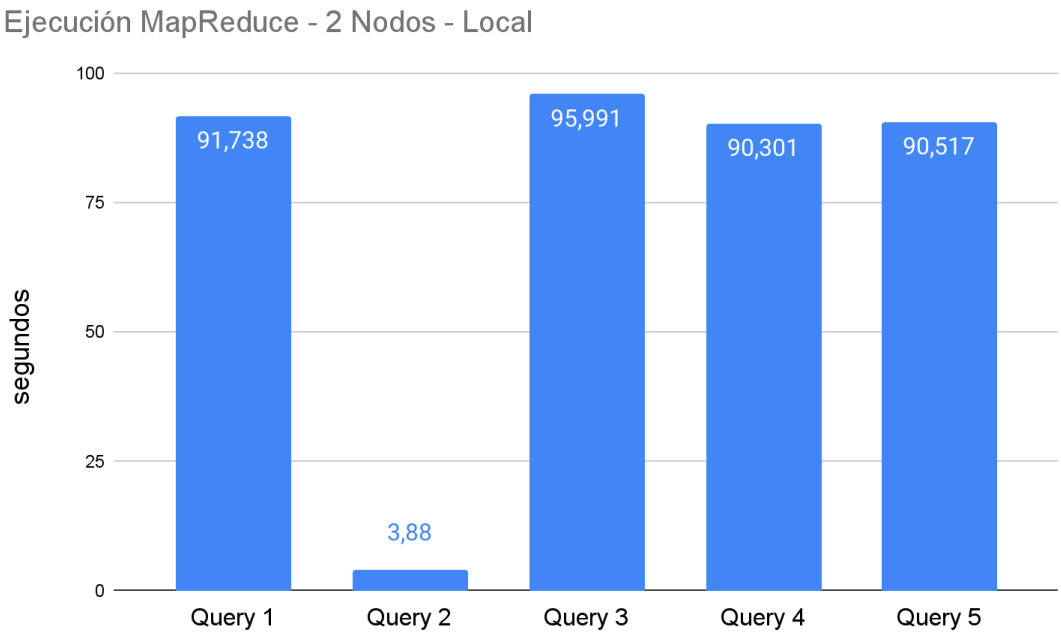
Se debe comentar que estamos usando un mapa para reducir correctamente y encontrar ese promedio máximo, este nuevo mapa está "repitiendo" información pero es necesario para calcular el promedio y al ser de 12 elementos como máximo, se permite.

Query 5:

Al ser necesario calcular la cantidad total de mediciones para luego agrupar por millón, se optó por utilizar dos etapas de *mapreduce*. Reutilizando la primera *query*, pudimos obtener las cantidades requeridas para luego poder realizar el siguiente cambio de par clave-valor: de sensor-cantidad a millón-grupoDeSensores. Al momento de implementar el *reducer*, resultaba natural agrupar en una lista los sensores, postergando la carga del listado de las parejas a un *collator*.

ANÁLISIS DE TIEMPOS

Ejecución MapReduce - 2 Nodos - Local				
Query 1	Query 2	Query 3	Query 4	Query 5
91,738	3,88	95,991	90,301	90,517



En los gráficos se puede ver que el tiempo de ejecución es similar entre todas las queries exceptuando a la query 2. Esto nos lleva a pensar que el uso del mapa que contiene los sensores juega un papel importante en el tiempo de ejecución de la query ya todas las querys la utilizan a excepción de la 2. Por último, cabe destacar que que los tiempos fueron tomados leyendo la totalidad del archivo *readings.csv* que contiene aproximadamente 4.5 millones de registros y esto directamente relacionado con el modulo del tiempo de ejecución en todas las queries.

PUNTOS DE MEJORA Y/O EXPANSIÓN

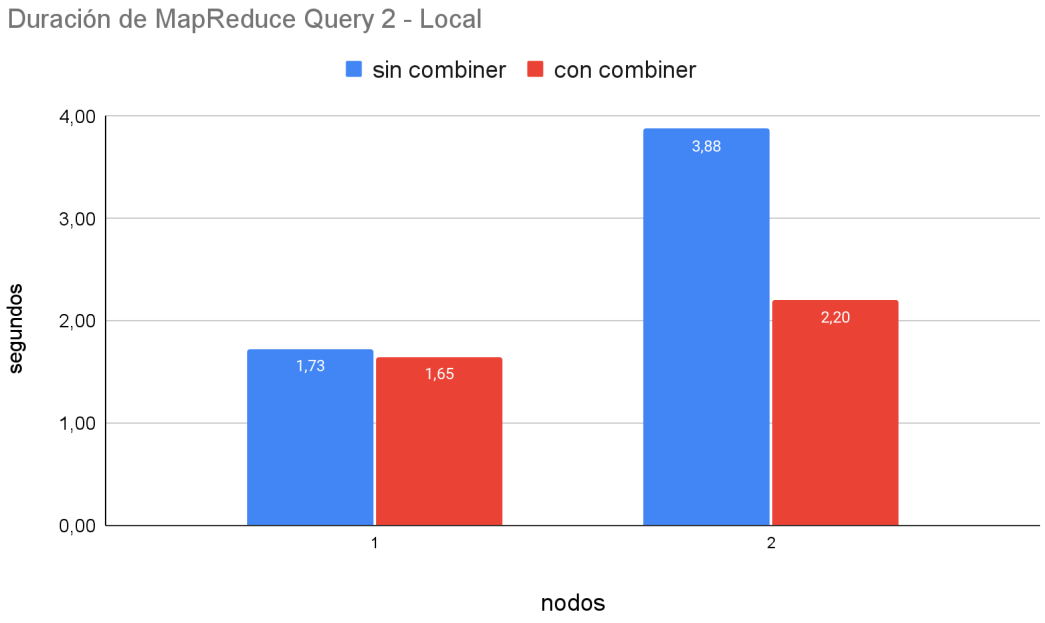
Un punto principal de mejora sería la utilización de pruebas unitarias para garantizar la correcta funcionalidad del código en sus componentes, como evaluar que un *Mapper* funcione correctamente por ejemplo. Esto sería importante particularmente en un *MapReduce* pues los datos sufren varias transformaciones y cálculos de a partes, por lo que los errores se arrastran en los componentes siguientes, hasta el resultado final.

Varias de las consultas realizadas necesitan utilizar campos del mapa de sensores en los *mappers*, y nosotros solucionamos esto implementando la interfaz *HazelcastInstanceAware* mostrada en clase. El problema que esto nos trae es que al correr distribuido en varias computadoras es necesario acceder a todo el cluster para conseguir los datos por cada operación

de *Map*. Esto causa que las consultas tarden más tiempo en completarse ya que tienen que conectarse al *Mapper* por cada *map*. Pensamos en dos alternativas para mitigar esto. La primera es tener el mapa de sensores en memoria y pasarlo por constructor a los *mappers* así no es necesario acceder al *cluster* para buscar la información, pero optamos por no hacerlo ya que si tuviéramos una cantidad muy elevada de datos de sensores podríamos querer que esta información esté distribuida para no tener que tenerla toda en memoria. La segunda, y más viable, sería dejar de acceder al *cluster* para conseguir datos del sensor y pasar solamente el *id* conseguido en el *reading*, luego en el *Collator* hacer por cada id hacer el mapeo del sensor pero por un *constraint* de tiempo no pudimos explorar esta alternativa.

Por último al correr el proyecto en Pampero notamos que los nodos no son capaces de guardar toda la información de readings del *csv* provisto por la Cátedra por un tema de *RAM*. Para mitigar esto probamos con una cantidad de datos reducida que *Pampero* sea capaz de manejar, pero no es una solución práctica. Una alternativa evaluada sería tener un modelo específico para cada *query* solamente con los datos necesarios para resolverla, en lugar de tener todo. Esto si bien podría resolver el problema para la cantidad de datos provista, igual no es una solución a largo plazo ya que con una cantidad de datos más grande igualmente se quedaría sin memoria.

ANÁLISIS DE USO DE COMBINER



Se eligió usar la segunda *query* para probar la mejora de eficiencia usando combiner, debido a que por el reducido rango de valores para la clave (los datos finales estaban agrupados por año) y el número de registros, cada instancia del *cluster* emitiria muchas veces parejas con la misma clave.

En un principio, en base al gráfico anterior podríamos argumentar que la inclusión del combiner mejoró la *performance* de la query en todos los casos. Pero entendiendo el funcionamiento del *combiner*, podemos descartar la diferencia en el primer caso por meras variaciones en las ejecuciones. Pero al momento de correr con dos nodos, se ve una mejora notoria.

CONCLUSIÓN

A lo largo de este proyecto, pudimos trabajar con Hazelcast tanto como *storage* como procesador distribuido, pudiendo poner en práctica los conocimientos adquiridos acerca de *mapreduce* y persistencia distribuida. Pero a pesar del enfoque constante que le dimos a la eficiencia, no siempre un incremento en la cantidad de nodos implicaba un incremento en esta última. De ahí que fue necesario considerar tiempos de transmisión por red, creación de instancias auxiliares, serialización y similares para poder determinar posibles factores a la hora de configurar y desarrollar el programa.