Trabajo Práctico Especial 2
Hazelcast

Programación de Objetos Distribuidos

Grupo 6 - 2020Q2

Integrantes:

- *58639* Brula, Matías Alejandro
- *58546* de la Torre, Nicolás
- *59356* Tallar, Julián
- *59347* Vuoso, Julián Matías

Diseño y decisiones generales

Comencemos con la puesta en marcha del Server. Decidimos utilizar configuración por XML, donde seteamos el modo discovery a multicast. De esta manera, cada nodo debe conocer únicamente un grupo y puerto (usamos el default) en lugar de las direcciones IP de los otros miembros. Además, seteamos la interfaz de red dinámicamente por parámetro, pudiendo modificarla según el ambiente.

Con respecto a las queries, todas parten de un mismo código que ejecutará una u otra según el script. Las consultas requieren del archivo arbolesXXX, por lo que lo parseamos según la ciudad indicada, armando un List<Tree>. Cada Tree con nombre, barrio, diámetro y calle. La lista será el punto de partida de todas las queries. El otro archivo lo parsearán las queries que lo necesiten.

Una alternativa que evaluamos fue utilizar un Map<String, List<Tree>>, con el nombre de barrio como clave y filtrando con un KeyPredicate. Pero no nos resultó del todo correcto en ese caso que un Mapper reciba una lista en lugar de un único valor.

También intentamos utilizar MultiMap. Éste nos permitía tener un mapa con múltiples valores por clave y así tener los beneficios del mapa anterior pero recibiendo por cada Mapper un único Tree. Pero tuvimos conflictos para integrarlo y resultados incorrectos.

Por último, al finalizar la ejecución de la query, el cliente envía un shutdown a la instancia de HZ, terminando su ejecución.

Diseño y decisiones de trabajos MapReduce

Query 1. El Mapper recibe una colección con los barrios y emite [nombre_barrio; 1] si la colección contiene al barrio en cuestión. El reducer los acumula y suma, y el Collator divide los valores por la población y los guarda ordenados. Para esta query y la siguiente decidimos realizar el filtrado del archivo de barrios en el Mapper. Aunque el mapa es otra instancia a serializar, evitamos emitir valores innecesarios. También consideramos filtrar directo en el Collator, evitando la serialización del mapa, pero emitiendo de más. Al imprimir, de haber 2 truncados iguales, se ordena por valor real.

Query 2. El Mapper recibe una colección con los barrios y emite un [Street;1] con calle y barrio solamente si el barrio está en la colección. El Reducer los acumula y retorna la suma. En el Collator recibimos un valor mínimo y nos quedamos solamente con la calle que tenga más cantidad de árboles por cada barrio y, que en su totalidad la cantidad de árboles sea mayor al valor pasado por parámetro.

Query 3. Empleamos un MapReduce con Collator. El Mapper procede a emitir [nombre de árbol; diámetro]. Luego el Reducer acumula los diámetros para cada especie, sumarizando la cantidad de árboles tratados en cada caso, para finalmente retornar el diámetro promedio de la especie. Por último el Collator los ordena y luego retorna *N* especies de la colección, siendo el valor *N* pasado por parámetro.

Query 4. El Mapper recibe un nombre y emite [nombre del barrio;1] si el árbol tiene se nombre. El Reducer acumula y retorna la suma, originando pares [nombre de barrio; cant_X]. En el Collator nos quedamos con los barrios cuya cantidad supere un mínimo indicado por parámetro y armamos los pares de barrios con el orden requerido.

Query 5. Se utilizaron dos procesos MapReduce. El primero realiza la cuenta de los árboles por barrio de la misma manera que la Query 1. A diferencia de ésta, se retorna un mapa con el nombre de barrio como clave y la cantidad de árboles como valor (redondeado a 1000).

La segunda etapa toma dichos valores como entrada. El Mapper los invierte (emite value-key), eliminando los que tengan “0 miles”, y el Reducer agrega todos los nombres de los barrios bajo la misma cantidad de miles bajo un Set. Por último, el Collator genera los pares de barrios recursivamente y las inserta en un set ordenado.

Análisis de tiempos de resolución

Realizamos 3 pruebas en cada query con el dataset de Vancouver y un único nodo, y las promediamos. Adjuntamos las tablas con los tiempos obtenidos en el anexo.

Los tiempos de lectura del archivo de árboles son similares, como era de esperar, dado que armamos la misma colección para todas.

Analizando los tiempos de MapReduce, vemos que la Q4 es la más rápida: razonable dado que emite menos que las demás e imprime pocos valores en el CSV (para el caso de VAN). La Q3, a pesar de tener que emitir para cada árbol, imprime únicamente un Top N (en la prueba, 3). La Q1 requiere la serialización del mapa de barrios e imprime varios valores en el CSV. La Q5, al requerir de dos MapReduce, requiere de algo más de tiempo (además de tener una salida larga). Por último, la más lenta es la Q2. Ésta requiere de la serialización del mapa de barrios y de los objetos Street. Además su collator requiere comparaciones con valores previos por cada barrio.

Con respecto al funcionamiento con varios nodos, intentamos realizar pruebas con 2 computadoras pero aunque los dos nodos se reconocían y armaban el clúster, al ejecutar una query, comenzamos a ver warnings de timeout y de migración de tareas. En algunos casos (y dependiendo de la PC) la consulta eventualmente terminaba con los resultados correctos, pero en tiempos cercanos a los 2 minutos.

Dados los problemas mencionados, pasamos a predecir qué pasaría si pudieran funcionar en clúster. Dado el dataset nuestro, creemos que con varios nodos los tiempos serían **mayores**, ya que los nodos deberán comunicarse para mantener la colección distribuida y para todo el procesamiento de MapReduce. Sin embargo, si tuviéramos un dataset más grande o un mayor costo de procesamiento, el funcionamiento distribuido comenzaría a reducir los tiempos.

Potenciales Mejoras

En primer lugar creemos que se podría elegir una mejor alternativa como colección inicial. Podría ser viendo cómo usar correctamente el MultiMap, sin tener los errores a los que llegamos.

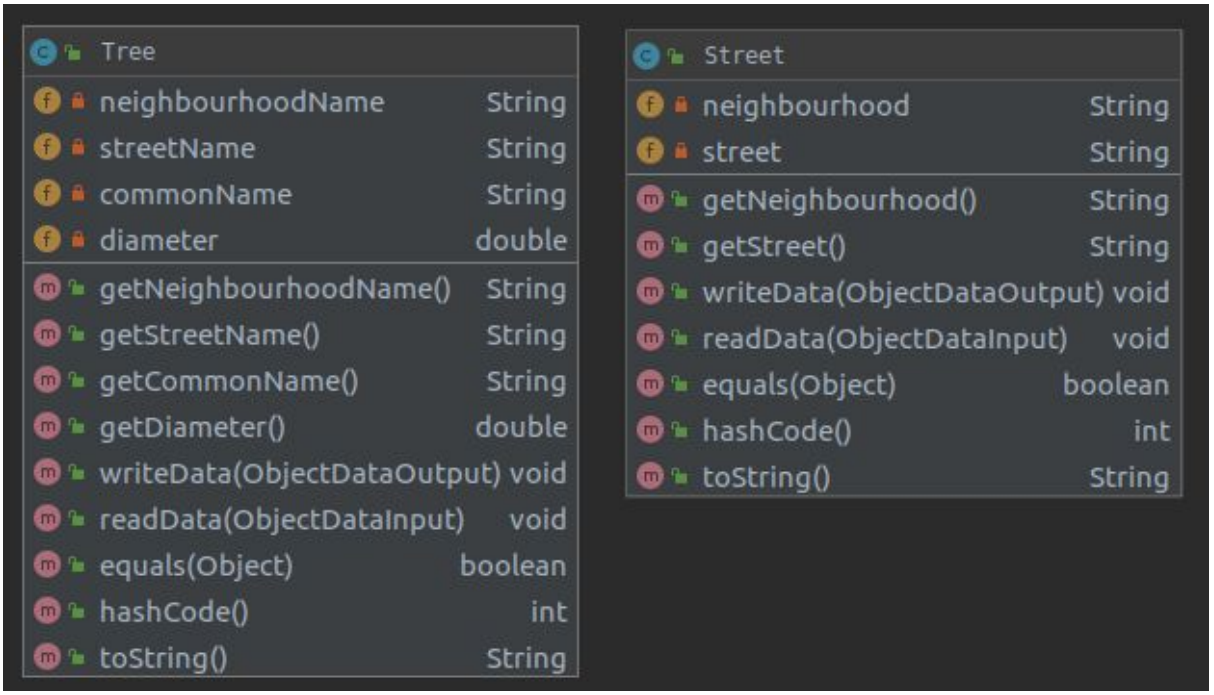
Por otra parte, podríamos adentrarnos en el archivo hazelcast.xml y buscar realizar modificaciones para lograr un mejor rendimiento de nuestro programa.

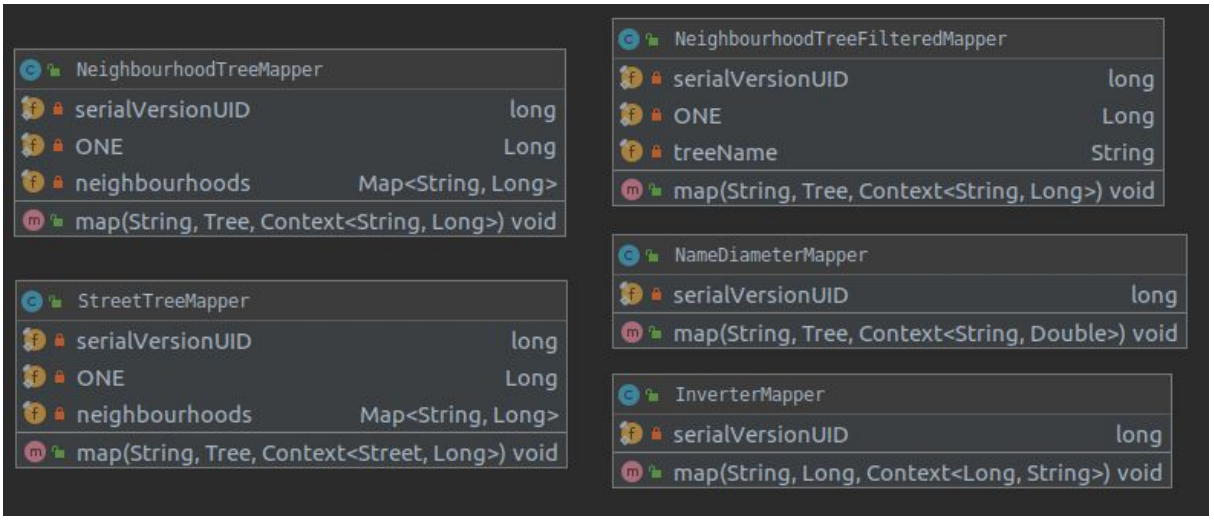
Por último, sería interesante utilizar Combiners en cada proceso MapReduce para disminuir el tráfico por la red, acumulando los valores que salen del Mapper antes de emitirlos.

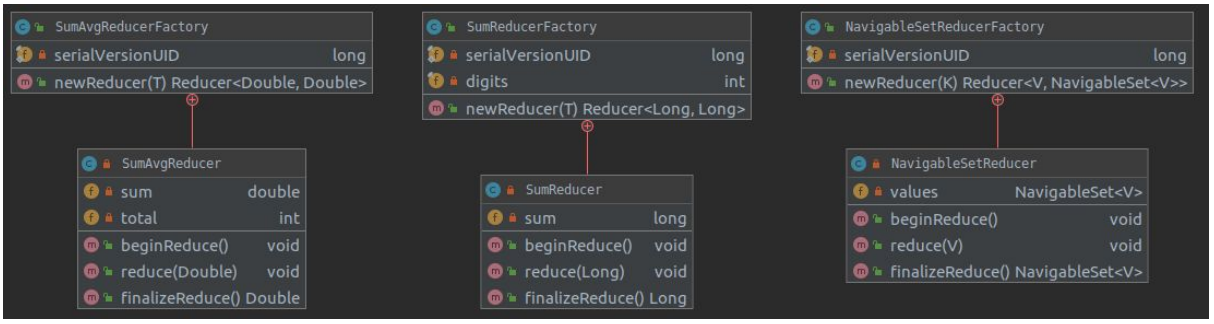
Tiempos obtenidos con VAN

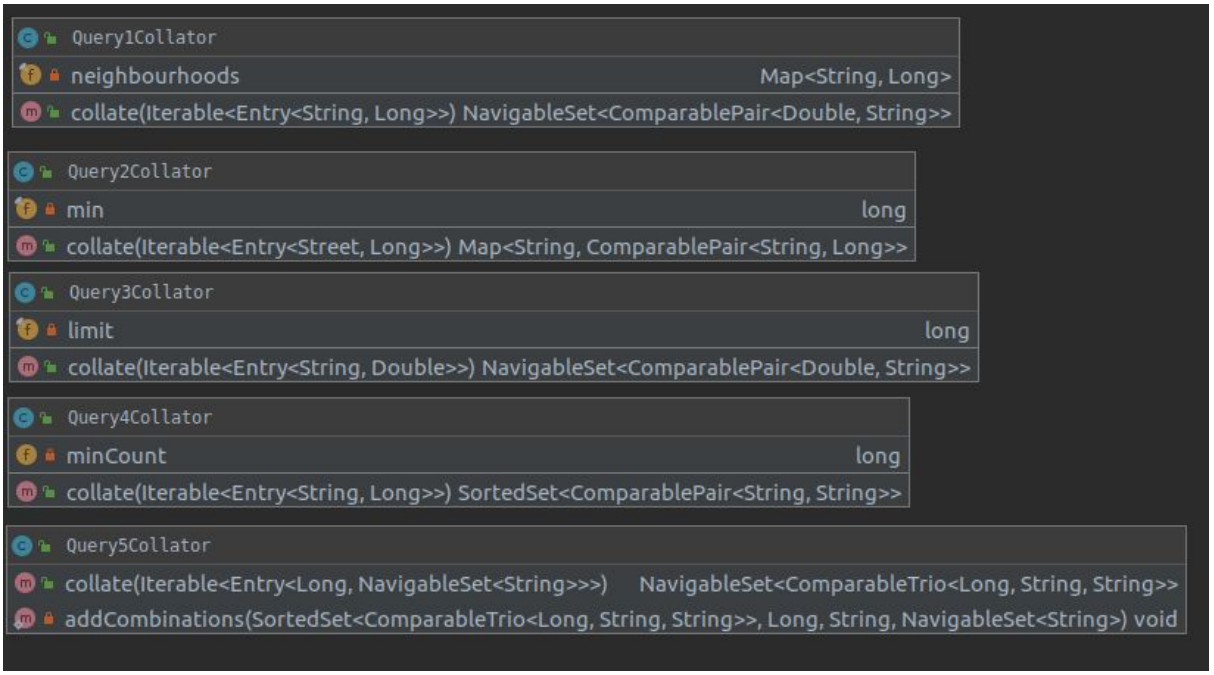
Query 1	T1	T2	T3	AVG
Lectura Arboles	0.718	0.662	0.837	0.739
MapReduce	0.297	0.106	0.114	0.172
Total	1.015	0.768	0.951	0.911
Query 2	T1	T2	T3	AVG
Lectura Arboles	0.922	0.658	0.922	0.834
MapReduce	0.505	0.337	0.313	0.385
Total	1.427	0.995	1.235	1.219
min=300				
Query 3	T1	T2	T3	AVG
Lectura Arboles	0.725	0.695	0.654	0.691
MapReduce	0.182	0.151	0.137	0.157
Total	0.907	0.846	0.791	0.848
n=3				
Query 4	T1	T2	T3	AVG
Lectura Arboles	0.791	0.854	0.953	0.866
MapReduce	0.090	0.093	0.178	0.120
Total	0.881	0.947	1.131	0.986
min=1, name='JAPANESE WALNUT'				
Query 5	T1	T2	T3	AVG
Lectura Arboles	0.734	0.791	0.675	0.733
MapReduce	0.256	0.180	0.134	0.190
Total	0.990	0.971	0.809	0.923

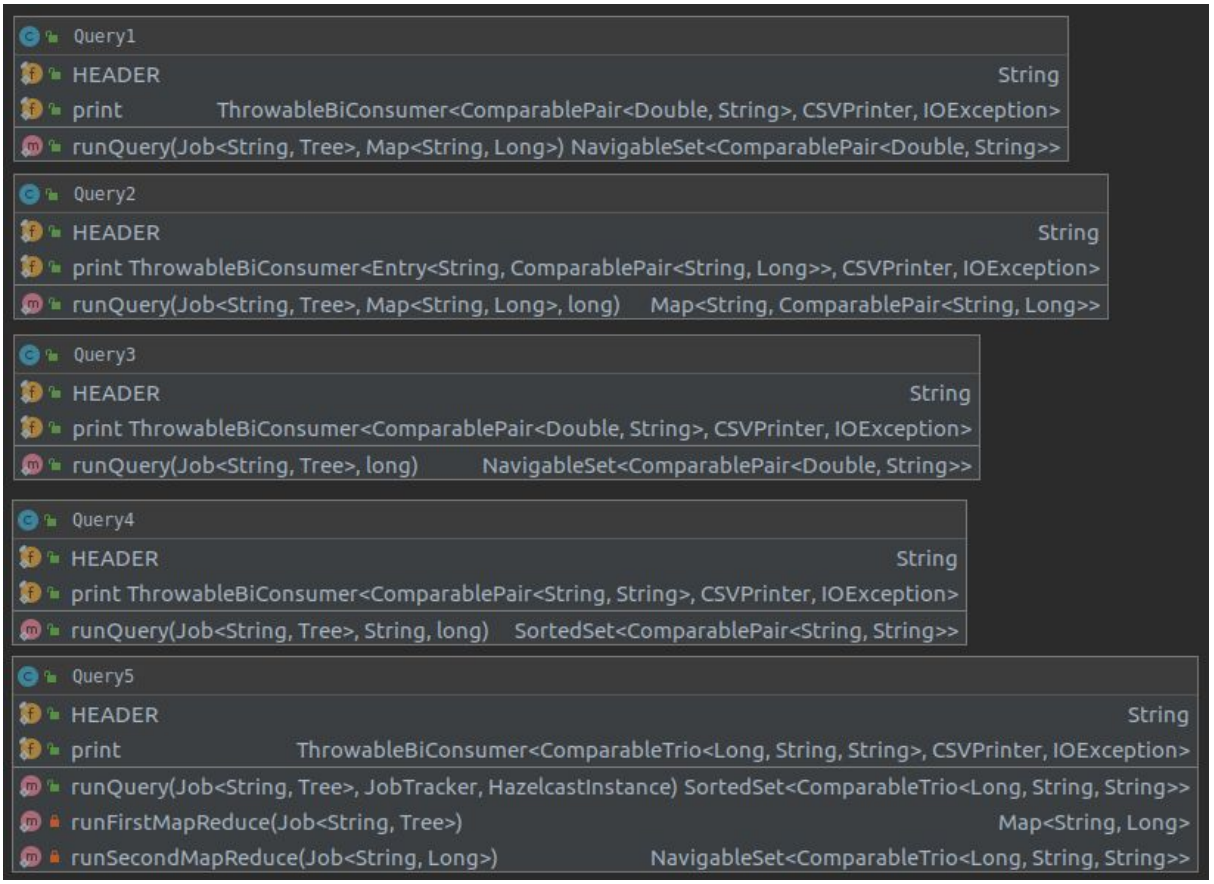
Diagramas UML








Query1Collator
neighbourhoods Map<String, Long>
collate(Iterable<Entry<String, Long>>) NavigableSet<ComparablePair<Double, String>>
Query2Collator
min long
collate(Iterable<Entry<Street, Long>>) Map<String, ComparablePair<String, Long>>
Query3Collator
limit long
collate(Iterable<Entry<String, Double>>) NavigableSet<ComparablePair<Double, String>>
Query4Collator
minCount long
collate(Iterable<Entry<String, Long>>) SortedSet<ComparablePair<String, String>>
Query5Collator
collate(Iterable<Entry<Long, NavigableSet<String>>>) NavigableSet<ComparableTrio<Long, String, String>>
addCombinations(SortedSet<ComparableTrio<Long, String, String>>, Long, String, NavigableSet<String>) void


Query1
HEADER String
print ThrowableBiConsumer<ComparablePair<Double, String>, CSVPrinter, IOException>
runQuery(Job<String, Tree>, Map<String, Long>) NavigableSet<ComparablePair<Double, String>>
Query2
HEADER String
print ThrowableBiConsumer<Entry<String, ComparablePair<String, Long>>, CSVPrinter, IOException>
runQuery(Job<String, Tree>, Map<String, Long>, long) Map<String, ComparablePair<String, Long>>
Query3
HEADER String
print ThrowableBiConsumer<ComparablePair<Double, String>, CSVPrinter, IOException>
runQuery(Job<String, Tree>, long) NavigableSet<ComparablePair<Double, String>>
Query4
HEADER String
print ThrowableBiConsumer<ComparablePair<String, String>, CSVPrinter, IOException>
runQuery(Job<String, Tree>, String, long) SortedSet<ComparablePair<String, String>>
Query5
HEADER String
print ThrowableBiConsumer<ComparableTrio<Long, String, String>, CSVPrinter, IOException>
runQuery(Job<String, Tree>, JobTracker, HazelcastInstance) SortedSet<ComparableTrio<Long, String, String>>
runFirstMapReduce(Job<String, Tree>) Map<String, Long>
runSecondMapReduce(Job<String, Long>) NavigableSet<ComparableTrio<Long, String, String>>