Informe TPE 2: Árboles

- Faustino Pannunzio, legajo: 59223
- Ignacio Sagüés, legajo: 59244
- Ignacio Vazquez, legajo: 59309
- Tobías Brandy, legajo: 59527

Decisiones de diseño e implementación

Para el trabajo se decidió implementar un cliente al cual se le podrán pedir cada una de las queries indicadas en el trabajo. Sin embargo, para cada una de ellas se diseñó una clase estática específica con la lógica pertinente. Finalmente, el cliente se encargará de correr el método correspondiente a la consulta que se haya pedido por parámetro en la invocación.

Dado que el servidor solo tiene el deber de iniciar el cluster de Hazelcast y mantenerlo vivo, la mayoría de la lógica implementada se encuentra en el módulo api. En este, existirá una carpeta con los componentes específicos de cada query. Sin embargo, a lo largo del trabajo se buscó construir componentes comunes que puedan ser utilizados por más de una query.

No utilizar el collator para hacer query 4 y 5 en un solo MR.

A la hora de realizar las queries 4 y 5, en las cuales se deben construir pares de respuestas parciales en base a algún criterio arbitrario, se pensó en primera instancia realizar este "armado" de la respuesta en el Collator. Sin embargo, se consideró que se estaba sobrecargando de lógica a este último y que esta no era su función, por lo que se implementó un segundo MapReduce para el armado de las parejas usando una respuesta intermedia como colección de entrada. En el caso de la query 4, la respuesta parcial será la respuesta final de la query 3 la cual en el segundo MapReduce será agrupada en base a los pares de barrios con igual cantidad de centenas de árboles diferentes.

Configuración de MultiMap

De forma nativa, Hazelcast implementa los MultiMaps utilizando un Set. Esta característica era incompatible con nuestro caso de uso deseado. Es por esto que se decidió modificar la configuración de Hazelcast para que, en vez de usar un Set como colección de los MultiMap, utilice una List.

Optimización en queries de conteo (1 y 5)

Para el caso de las queries en las cuales se debían realizar operaciones de conteo, utilizamos la estrategia de mapear los valores de cada tupla clave-valor a 1, y luego en el combiner/reducer, sumarlos y retornar el total. Los valores son provistos al MapReduce job mediante un multimapa.

Vimos dos estrategias para mapear los datos. La primera es al momento de cargar los CSV, mapear cada objeto (sea barrio o árbol) a una clave y el mismo objeto como valor. Luego, el mapper lo único que deberá hacer es emitir la misma clave y un 1 como valor, ignorando el valor que viene del multimapa. Esta situación es más realista, pues es probable que tengamos almacenados estos objetos en las estructuras de Hazelcast, pero es muy poco eficiente en el uso del ancho de banda, pues transmitimos objetos enteros para luego ignorarlos. La otra opción es directamente dejar en el multimapa el valor 1, y luego el mapper no tendrá que hacer nada. Esta estrategia es menos realista (quién guardará el valor constante 1?!), sin embargo mucho más eficiente, el valor 1 tiene un tamaño minúsculo en comparación con un objeto entero.

En pos de ser eficiente en el uso de la red, como indica la consigna, elegimos la segunda estrategia, a pesar de ser menos realista.

Abstracción de la lógica general de las queries.

El proceso vinculado a correr cada una de las queries es muy similar entre sí. Todas necesitan conectarse al cluster de Hazelcast que hará las operaciones, necesitan leer los archivos provistos y necesitan escribir la respuesta del procesamiento. Estas tareas son realizadas por el Client. Luego, cada query es responsable de cargar la información en Hazelcast de la forma que le sea más conveniente, determinar las implementaciones deseadas de Mapper, Reducer, etc. y limpiar los recursos utilizados una vez finalizada la ejecución. Si bien la serialización de la respuesta no está realizada de forma directa al correr una Query, cada una expone la lógica necesaria para poder dejarla en formato CSV.

En particular, se consideró importante abstraer la lógica de lectura de los archivos de entrada. Esto se debe a que se pedía trabajar con dos tipos de fuentes de datos distintas y que, en un futuro podría ser de interés utilizar alguna nueva fuente no presente en el diseño de este sistema. Para esto se implementó se definió un enum que representa las posibles fuentes de datos junto con la lógica asociada a obtener la información de los árboles y los barrios utilizada por el sistema.

QueryAnswers

Para cada una de las queries, se creó también una clase específica representando la respuesta de la misma. La idea detrás de esto fue que la respuesta final sea construida en los Reducers siendo agnóstico a lo que se realizará con esta luego del proceso de MapReduce. Las Queries recibirán un callback por lo que se podrá decidir que se realizará con la respuesta. Esto ayudó mucho a la hora de realizar los test ya que permitió manipular las respuestas de manera sencilla y poder así realizar los chequeos de manera directa.

Reutilización de KeyPredicate y Sort Collator

Dado que, para las consultas del trabajo se deben filtrar las entradas en base a una colección, se diseñó un KeyPredicate agnóstico al tipo de colección usada para el filtrado, siempre que Hazelcast la soporte, creando la instancia con el HazelcastCollectionExtractor correspondiente. Este tuvo que implementar HazelcastInstanceAware dado que la colección por la cual se filtra estará guardada en este. A su vez, como todas las consultas pedidas por el trabajo contarán con algún orden en particular, se construyó el Sort Collator que se encargará de ordenar la lista de respuestas obtenidas en el MapReduce recibiendo un Comparador. A su vez, esta clase recibirá un Callback que, como ya fue explicado anteriormente, permitirá decidir al usuario que desea realizar con la colección ya ordenada.

Ordenamiento parcial query 4 y 5

En las query 4 y 5, decidimos aprovechar el cómputo distribuido del Map Reduce ordenando parcialmente la lista de valores de salida del Reducer (por barrio/calle), y luego en el Collator solo tener que colocar dichas listas en el orden que indica la key (por grupo). Entendemos que ante volúmenes grandes de datos, esta estrategia debería mostrarse más performante.

Futuras mejoras

Se podría mejorar la manera de leer los campos específicos de los CSV's dado que en este momento, la posición de los parámetros elegidos está hardcodeada.

No es ideal que los métodos de ejecución de las queries obtengan sus parámetros específicos directamente de los System Properties. Para evitar esto, una opción es que dejen

de ser métodos estáticos para pasar a ser clases de instancia, y poder configurarles los parámetros mediante el constructor.

Rendimiento de Cómputo Distribuido

Dado que no teníamos acceso a múltiples computadoras dentro de la misma red local, decidimos comparar el rendimiento de las queries variando la cantidad de nodos corriéndose todos en la misma computadora.

Si bien esto no es un escenario realista, cada nodo podrá aprovechar un núcleo distinto del procesador, y deberán ponerse de acuerdo con el mismo protocolo que usarían en un caso real.

Nuestra hipótesis es que solo tiene sentido realizar el cálculo de manera distribuida si el tiempo de procesamiento que realiza cada nodo en el Mapper y el Reducer justifica el overhead que representa serializar y deserializar los datos, y que los nodos se pongan de acuerdo y distribuyen los datos. Por supuesto, cuanto mayor el tamaño de los datos, mayor debe ser el tiempo de procesamiento que justifique la distribución del mismo. Es decir, el cociente:

(tamaño promedio clave valor) / (tiempo de procesamiento clave valor)

debe ser menor a cierta constante C, para que distribuir el procesamiento disminuya el tiempo de cómputo, y no lo aumente.

Nuestra predicción para este caso es que no tiene sentido distribuir, pues el trabajo que debe realizar cada nodo es muy simple y rápido como para justificar el overhead.

Todos los análisis fueron realizados con el dataset de Vancouver, sólo variando la cantidad de nodos, y solo considerando los tiempos de trabajo del Map Reduce, no del cargado de la información de los datasets.

nodos / query	Query 1	Query 2	Query 3	Query 4	Query 5
1	1,510 s	3,145 s	1,537 s	1,630 s	1,064 s
3	6,169 s	3,872 s	3,989 s	4,905 s	3,015 s
5	4,856 s	3,253 s	4,479 s	4,675 s	2,974 s

Lo primero interesante que se confirma es que el procesamiento distribuido en general empeora la performance, como esperábamos. También vemos que pasar de 1 a 3 nodos es mucho peor que pasar de 3 a 5, por lo que el degradamiento de performance es lejos de lineal. Sería interesante probar qué pasaría con procesos más intensos de CPU, a ver si en ese caso si compensa distribuir el trabajo.

Una cosa que no esperábamos es la diferencia de performance de la Query 2, respecto de la demás. Más aún, que distribuir el trabajo casi no haya impactado el tiempo de ejecución. A simple vista, sin un análisis más profundo, no pudimos distinguir que causa este comportamiento.

Es importante aclarar que este análisis está lejos de ser conclusivo o académico. Si bien intentamos mostrar valores promedios, las pruebas se realizaron en una CPU que ejecutaba otras tareas paralelamente, ni tampoco realizamos análisis del error ni desviación de los resultados.