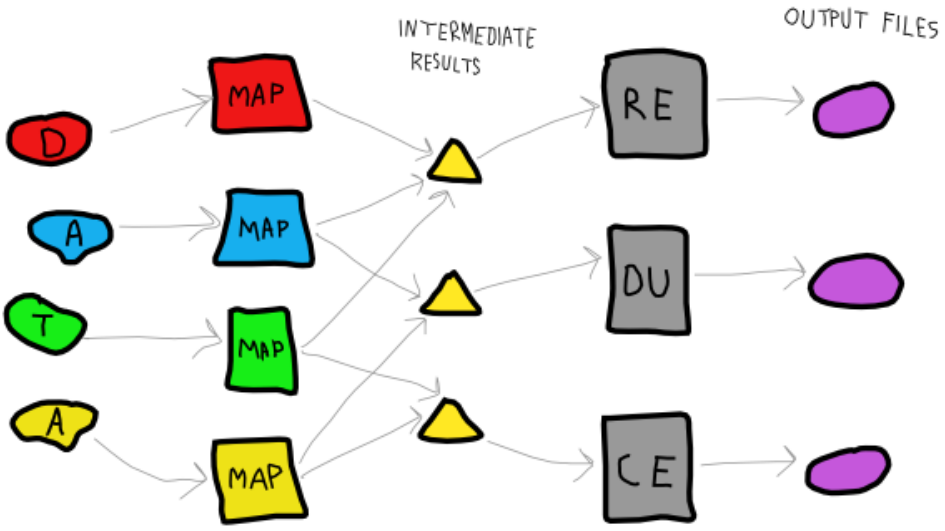
Trabajo Práctico 2 - Grupo 4

Mizrahi, Thomas (60154)
tmizrahi@itba.edu.ar

Chayer, Iván (61360)
ichayer@itba.edu.ar

Di Toro, Camila (62576)
cditoro@itba.edu.ar

Hazelcast MapReduce



Profesores:
Marcelo Emiliano Turrin
Franco Roman Meola

72.42 Programación de Objetos Distribuidos
Instituto Tecnológico de Buenos Aires
Segundo cuatrimestre 2023

Introducción

En este breve informe, vamos a comentar algunas de las decisiones de diseño que tomamos a lo largo de este trabajo práctico así como el análisis de los tiempos de cada query (con y sin optimizaciones) y por último algunas mejoras a futuro.

Diseño de las componentes

Respecto de la construcción de una query y sus estrategias:

Cada *query* se construye utilizando un `QueryBuilder`. Este constructor permite establecer los parámetros necesarios para, posteriormente, crear un `BaseQuery` que ejecutará, a través de una estrategia, la carga de los datos, la/las operación/es MapReduce y finalmente el procesamiento de los archivos *csv* y *txt* de salida.

El código fue implementado de tal manera que siempre se ejecuta una estrategia predeterminada por defecto. En caso de querer ejecutar otra estrategia, se puede indicar utilizando el parámetro `-Dstrategy` seguido de su nombre, el cual fue establecido en el método `setStrategy` de la query correspondiente.

Respecto de las operaciones MapReduce:

Lo primero que consideramos en una operación MapReduce es desde dónde se hará el *map*. Dado que nuestros *datasets* son *stations* y *trips*, estas son las opciones más naturales. Consideramos iniciar un MapReduce a partir de *stations*, pero esto significa que luego para cada estación, habría que buscar todos los viajes asociados. Preferimos entonces partir de *trips*, usar los IDs de estaciones durante *map* y *reduce*, y recién al final buscar las estaciones relacionadas.

Esto trae un problema, ya que solo se deben listar las estaciones presentes en `stations.csv`, ¿No sería mejor filtrar los viajes al principio para evitar procesarlos? Sin embargo, el inconveniente con esto es que habría que realizar este chequeo, que implica un acceso a un mapa distribuído, en la carga de viajes o en el mapper. Esto significa que podría tener que realizarse este acceso al mapa varios millones de veces. Asumiendo que la mayoría de los viajes referencian estaciones válidas, es preferible procesar unos registros de más y realizar este chequeo al final, dónde se hará probablemente menos de mil veces.

Descripción breve del diseño del MapReduce de cada estrategia de cada query:

Query 1

Query1Default:

La carga de datos no se ve afectada por ninguna condición.

Luego, por cada `Trip`, el *map* emite un par (`{originStationId, destinationStationId}, 1`). El *reducer* toma todos los pares que comienzan con dicha clave compuesta y suma los valores asociados para obtener la cantidad total de viajes iniciados en la estación `originStationId` y finalizados en la estación `destinationStationId`.

Por último, un *submitter* transforma estos IDs de estaciones en nombres, descartando los registros que contengan un ID de estación inválido.

Query1FromStations:

La carga de datos solamente filtra los trips que tengan el mismo origen y destino.

Luego, se cargan los datos de los *trips* en un multimapa. A partir de una estación el mapper emite un par (`{originStationId, destinationStationId}, countOfEntriesWithSameDestination`). El *reducer* simplemente tiene que sumar todos los valores asociados a dicha clave compuesta.

Query 2

Query2Default:

La carga de datos es análoga a `Query2Default`, la única diferencia es que se utiliza un MultiMap para los viajes. Dicho MultiMap contiene como clave el `originStationId` y como valores todos los *trips* que comienzan en dicha estación.

Luego, por cada `Station`, el *map* emite un par (`"result", ArrayList<StationAndDistance>`). Este ArrayList consiste de una lista ordenada con N o menos elementos, siendo N parámetro del query. El *reducer* entonces consiste en tomar todas las listas que reciba y quedarse con los N más altos, descartando todos los demás. La operación produce un mapa con una clave `"result"` que contiene la lista final resultante.

Query2FromTrips:

En la carga de datos, particularmente de los *trips*, se impone la condición de que *member* sea *true* y que la estación destino no sea la misma que origen.

Luego, por cada `Trip`, el *map* emite un par (`originStation`, `distanceToDestination`). Para lograrlo, el *mapper* implementa *HazelcastInstanceAware* para obtener la instancia de Hazelcast, utilizarla para obtener del mapa de estaciones, que puede usar posteriormente para encontrar las coordenadas de cada estación.

Notar que el *reducer* no puede simplemente promediar estos valores, pues debe realizar el promedio de todos, entonces maneja dos valores; la suma total y la cantidad total. Esto permite finalmente obtener el promedio de distancia aproximada, con el collator al final de la operación tomando solo los N mayores.

Query 3

Query3Default:

En la carga de datos, particularmente de los *trips*, se impone la condición de que el destino no sea igual al origen.

Luego, por cada `Trip`, el *map* emite un par (`originStationId`, `Trip`). El *reducer* entonces toma todos los viajes que comienzan en una estación y toma el de mayor duración.

Query 4

Query4Default:

Consiste de dos operaciones MapReduce:

- La primera mapea cada `Trip` a dos pares (`StationIdAndDate, Integer`), el primero consistiendo del ID de la estación origen y el tiempo de salida, con valor -1. El segundo par es lo mismo pero con el destino y valor +1. Cualquiera de los dos es descartado si cae fuera del rango de fechas pedido. Luego el reduce consiste de la suma de todos estos valores. Esto produce un mapa que para cada estación-día, nos dice cuántas bicicletas totales netas entraron para dicha estación en dicho día.
- Esto se usa como entrada a otro MapReduce, con el *mapper* produciendo `AffluxCount`, que simplemente cuenta la cantidad de días con flujo positivo y negativo. El *reducer* suma todos estos `AffluxCount` para finalmente saber cuántos días hubo flujo positivo, negativo, y el neutro se calcula restando esos valores a la cantidad total de días en el rango. Este cálculo se hace en un *collator*, que también se encarga de encontrar el nombre de la estación (ya que hasta ahora, todo el procedimiento operó sólamente con los viajes)

Query4FromStations:

Consiste en cargar los viajes en dos multimapas, uno por estación origen y otro por estación destino (entonces se duplicarán los viajes). El MapReduce se inicia a partir de las

estaciones, el mapper mapea cada estación a un `AffluxCount`, usando los multimapas para computar los días con afluencia positiva y negativa (esta no es una utilización muy eficiente de MapReduce).

Consideraciones adicionales de esta sección

- Se omiten detalles de optimizaciones y post-procesamiento ya que el grupo entiende que el objetivo es demostrar el entendimiento de la estrategia MapReduce.
- Todos los queries default hacen uso de *combiners* para optimizar la distribución y el tráfico de red.

Análisis de tiempos por estrategia

Para todas las estrategias MENOS la Query 1 (999.999 de registros) se cargaron todos los registros de los archivos CSV provistos por la cátedra.

Query Strategy	Tiempo carga de datos	Tiempo ejecución del query	Cantidad de nodos	Uso de Combiner
Query1Default	~ 10 segs	~ 6 mins	1	✅
Query1Default	~ 20 segs	~ 11 mins	2	✅
Query1Default	~ 10 segs	~ 13 mins	2	❌
Query1Default	~ 15 segs	~ 13 mins	3	✅
Query1Default	~ 20 segs	~ 14 mins	3	❌
Query1FromStations	~ 10 segs	~ 6 mins	1	✅
Query1FromStations	~ 30 segs	~ 2 mins	2	✅
Query1FromStations	~ 20 segs	~ 3 mins	2	❌
Query1FromStations	~ 1 mins	~ 4 mins	3	✅
Query1FromStations	~ 30 segs	~ 5 mins	3	❌
Query2Default	~ 2 mins	~ 9 mins	1	✅
Query2Default	~ 2 mins	~ 16 mins	2	✅
Query2Default	~ 3 mins	~ 16 mins	2	❌
Query2Default	~ 6 mins	~ 6 mins	**3**	✅
Query2Default	~3 mins	~ 20 mins	**3**	❌
Query2FromStations	~ 2 mins	~ 4 mins	1	✅
Query2FromStations	~ 3 mins	~10 mins	2	❌
Query2FromStations	~ 3 mins	~ 4 mins	2	✅
Query2FromStations	~ 3 mins	~ 6 mins	**3**	❌
Query2FromStations	~ 3 mins	~ 4 mins	**3**	✅
Query3Default	~ 2 mins	~ 1 min	1	✅
Query3Default	~ 3 mins	~50 segs	2	✅
Query3Default	~ 5 mins	~ 2 mins	2	❌
Query3Default	~ 3 mins	~ 50 segs	**3**	✅
Query3Default	~ 4 mins	~ 90 segs	**3**	❌
Query4Default	~ 1,2 mins	~ 15 mins	1	✅
Query4Default	~ 2 mins	> 60 mins	**3**	✅

Query4Default	~ 2 mins	~ 60 mins	2	❌
Query4FromStations	~ 2,4 mins	~ 44 segundos	1	no tiene
Query4FromStations	~ 6 mins	~ 17 segundos	**3**	no tiene

Análisis de tiempos con y sin Combiner

Con un *combiner*, se podría reducir significativamente la cantidad de datos que se envían a través de la red, lo que mejoraría drásticamente el rendimiento de la operación del *reduce*.

Por ejemplo, se puede ver como el *combiner* mejora la performance para `Query3Default` y para la `Query2Default`.

Mejoras a futuro

En algunos de los *queries*, el tiempo de carga de datos resulta una cantidad significativa del tiempo total de ejecución. En casos reales, los datos podrían ya estar precargados en el *cluster* y se podría realizar la operación de MapReduce directamente sobre ellos. Esto requeriría que los queries usen los datos en el mismo formato, lo que limitaría las estrategias que se pueden utilizar.