Instituto Tecnológico de Buenos Aires

Informe

72.42 - Programación de Objetos Distribuidos

Tepedino, Cristian - 62830
Mendonca, Juana - 61185
Bloise, Luca - 63004

Decisiones de diseño generales

Para leer los datasets, desarrollamos un conjunto de clases “csvParser”, que conocen toda la lógica sobre la disposición del csv y traducen cada registro en un objeto que contenga la información de forma agnóstica a la ciudad. Esto lleva a que agregar una nueva ciudad solo implique la creación de los csv parsers asociados y el agregar una entrada al enum de ciudades disponibles.

Para el desarrollo del sistema se tomaron decisiones de diseño que permitieran a las queries poder operar de forma eficiente ante grandes volúmenes de datos. Particularmente, se trató de optimizar el tráfico de red.

Trabajando bajo la asunción de que los datasets de infracciones y agencias siempre van a tener cantidades de registros similares a las de los archivos de NYC y CHI dados en el enunciado, decidimos que lo mejor es mantener las colecciones de agencias y de infracciones en la memoria del cliente, en vez de guardarlos en hazelcast. Esto permite evitar el tener que transferir esta información por red, además de volver ligeramente más rápida la lectura de la información. Aún así, en caso de que este supuesto sea invalido, y dado que mantener en memoria estas colecciones podría ser un problema si se tiene en algún momento un dataset con más infracciones y agencias, se dejó en el proyecto una versión alternativa de la query1, que sí guarda estos datos en hazelcast. Para las otras queries este mismo cambio podría hacerse fácilmente, pero por simplicidad no incluimos las versiones alternativas en el repositorio.

Se desarrollaron múltiples clases que encapsulan solo unos pocos campos del total de información, a modo de DTOs, para así evitar tener que pasar información de más entre los componentes de los jobs.

En todas las queries, la keyValueSource proviene del dataset de tickets, aunque nuevamente solo guardamos un DTO con la información esencial para la query. Decidimos usar una clave autoincremental para el IMap, dado que ninguna combinación de campos de los registros de tickets parecía garantizar unicidad.

Para el testeo de los correctos resultados de las queries, se decidió escribir scripts que ejecuten una sentencia equivalente a lo que busca la query en PostgreSQL y que comparen los resultados con los de las queries. Sin embargo, no pudimos hacerlos enteramente automáticos, ya que los archivos que generan tienen ligeras diferencias que no pudimos resolver (Por ejemplo, al ordenar alfabéticamente, Java considera a un espacio como un valor menor a una letra, mientras que PostgreSQL hace lo contrario, llevando a un orden ligeramente distinto de los resultados en los csv generados).

Algunas queries requerían mostrar la definición de una infracción en su salida, pero el csv de multas sólo contiene el id. En estos casos se realizó todo el job MapReduce usando el id y al final, en el collator, se realizó el cambio de información. Esto evita tener que modificar cada registro de multa en un punto anterior de la query, puesto que los resultados finales casi siempre tendrán un número menor de registros. Además, se espera que los id de las definiciones sean más cortos que sus descripciones, minimizando así la información que viaja por red.

Decisiones de diseño de los jobs MapReduce

Query1:

El mapper se encarga del filtrado de multas a solo aquellas cuya agencia e infracción se corresponda a valores que existan en el csv correspondiente. Emite como key al par (infracción, agencia).

El combiner y el reducer se encargan de contar la cantidad de ocurrencias de cada key distinta.

Finalmente, el collator guarda los campos de claves y valores en un set ordenado según el criterio pedido, mientras que tambien reemplaza los id de infracciones por sus descripciones.

Query2:

El mapper recibe la agencia, la fecha y el monto y filtra los resultados cuya agencia no este en el csv de agencias. Emite como key al par (agencia, año) y como valor al mes y monto de la multa.

El combiner se encarga de generar y mantener un array con los montos por mes, y luego enviarselo al reducer. Luego, el reducer suma todos los arrays que pertenezcan a la misma clave.

El collator nuevamente arma un set ordenado, pero esta vez por cada entrada que reciba del reducer, la “expandirá” en 12 entradas al set (1 por mes), cada una con la agencia, fecha y el YTD calculado a partir del array recibido del reducer.

Query3:

Dada la complejidad de esta query, se opto por usar 2 jobs MapReduce para resolverla.

La primera tiene como objetivo devolver, para cada barrio y patente, la cantidad de reincidencias de cada infracción.

El mapper filtra las multas que no pertenezcan al rango de fechas, y emite como key al par (patente, barrio) y como valor al id de la infracción.

Luego, el combiner emite un mapa con el id y la cantidad de ocurrencias de cada infracción para cada par patente-barrio, y luego el reducer recibe estos mapas del reducer y los combina. Ya que solo interesa la cantidad de veces que cada patente cometió una infracción (y no cual es la infracción en sí), lo que se emite es una lista formada a partir de los valores del mapa.

Este job no usa collator, sino que simplemente se usa la salida como entrada del segundo job, que se encarga de calcular los porcentajes de reincidencia por barrio.

El mapper de este segundo trabajo toma la lista de repeticiones y emite un booleano según si existe algún valor de la lista que supere la cantidad mínima de repeticiones con la que una patente se considera reincidente. La llave emitida esta vez es solo el barrio, y ya se sabe que todo valor que llegue con un mismo barrio será una patente distinta.

Luego, el combiner y el reducer llevan cuenta del total de valores recibidos por barrio, y además cuentan el total de valores que se consideran reincidentes.

El collator se encarga de volcar estos datos al set ordenado, calculando el porcentaje a partir del valor del par (total, reincidentes) que emite el reducer.

Query4:

El mapper filtra para que solo queden los valores cuya agencia sea la pedida y cuya infracción exista en el csv de infracciones. Se usa como clave el id de la infracción y como valor, su monto.

El combiner recibe estos montos y mantiene un rango indicando el mayor y el menor, que después emite el reducer. El reducer por su parte compara y combina los extremos de los rangos que recibe para luego emitir al rango máximo.

Finalmente, el collator arma el set ordenado a partir de la infracción y el rango, y luego se devuelve una lista con los N valores más grandes del set.

Análisis de los tiempos de resolución de cada query

Para las pruebas se utilizó el dataset de 5M de registros de Chicago provisto en el enunciado. Además, para los nodos se usaron VMs simulando una red local de computadoras independientes.

Para los mapReduce jobs, los tiempos fueron:

Nodos	1	3	5
Query 1	2s 912ms	2s 505ms	2s 350ms
Query 2	5s 891ms	3s 501ms	3s 274ms
Query 3 (job 1)	8s 684ms	7s 402ms	6s 921ms
Query 3 (job 2)	2s 967ms	2s 391ms	2s 325ms
Query 4	2s 100ms	1s 688ms	1s 403ms

Entendemos que estos números no representan un caso realista, dado que no estamos contando la latencia que generaría transmitir datos en una red real, sin embargo, entre los datos obtenidos y la teoría vista en clase consideramos seguro asumir que el aumento de nodos tendrá a aumentar la velocidad de los jobs MapReduce

Comparando la query1 y la query1 alternative, los resultados fueron:

Nodos	1	3	5
Query 1 (lectura)	36s 204ms	97s 842ms	96s 734ms
Query 1a (lectura)	36s 516ms	96s 756ms	98s 922ms
Query 1 (job)	2s 912ms	2s 505ms	2s 350ms
Query 1a (job)	3s 315ms	2s 875ms	2s 612ms

La diferencia no es grande, pero demuestra que cargar las colecciones de menor tamaño en hazelcast disminuye la performance de las lecturas, y el tener que obtener los mapas parece ser más lento que enviarlo (aunque creemos que sería conveniente evaluar que tan cierto es esto en una red real)

En cuanto a los tiempos sin combiner, en algunas queries el combiner recibe un tipo de dato y emite otro tipo de dato combinando información, y luego el reducer ese tipo de dato. Entonces, realizar las query sin combiner implicaría programar una versión de reducer que tome el tipo de dato de entrada del combiner y emita el tipo de dato de salida del reducer, y la lógica sería prácticamente la misma que la del combiner en la versión actual de las queries. Sin embargo, por falta de tiempo no pudimos implementar estas versiones

alternativas. En todos los casos, sin embargo, lo que se espera que pase al no usar combiners es que los tiempos de las query aumentaran al ejecutar en múltiples nodos, en comparación a las versiones que lo usan (aunque en 1 nodo podrían ser prácticamente iguales).

Potenciales puntos de mejora y/o expansión

Mejoras en la eficiencia de la query 3: Nos gustaría analizar la performance de distintas estrategias para resolver esta query. Uno de los métodos posibles que consideramos, pero no pudimos probar por falta de tiempo, es que en el primer combiner y reducer se sepa el N que se toma para considerar a una placa como reincidente. Entonces, el reducer en lugar de emitir la lista simplemente emitirá el booleano que luego usa el segundo mapper, y el combiner podría emitir una instancia de un objeto con un campo booleano y un mapa. Si desde el combiner ya se detecta que un valor superó al N, el mapa se pone en null y solo se envía el booleano. El reducer, al detectar que un valor que recibe del combiner ya no contiene el mapa, puede simplemente ignorar todos los otros valores y emitir “true” al final sin realizar más cálculos.

Mejoras en la adaptabilidad a formatos de csv: Si bien consideramos que la solución actual aísla el código encargado de interpretar los csv de una forma suficiente, sigue requiriendo el modificar el código del programa. Una mejor alternativa podría ser el uso de archivos de configuración que guarden el orden de columnas y otra información relevante (formato de fechas, tipo de número) y que el programa pueda leer ese archivo de configuración para conocer la estructura del csv en runtime.

Mejorar las formas de testear el sistema: Para el scope del proyecto, no nos resulto un problema comparar los diff que generamos para revisar que las diferencias sean por problemas que no sean importantes, sin embargo, en un proyecto más grande el no poder automatizar los testeos es un problema. Por lo tanto, de tener más tiempo nos hubiera gustado poder cambiar la forma en que testeabamos el sistema (por ejemplo, mediante tests unitarios en los componentes del MapReduce o probando las queries con datasets pequeños cuyo resultado ya sea conocido)