Instituto Tecnológico de Buenos Aires

MapReducing Parking Tickets

72.42 Programación Orientada a Objetos - 2024Q1

Alumnos:

Tomás Santiago Marengo, 61587

Abril Occhipinti, 61159

Santino Ranucci, 62092

Agustin Zakalik, 62068

Profesores:

Ing. Marcelo Turrín

Ing. Franco Román Meola

Índice

1. Introducción

En este trabajo, se explora el diseño y la implementación de un sistema distribuido utilizando Hazelcast y el paradigma MapReduce para procesar grandes volúmenes de datos de infracciones de tránsito. El objetivo principal es desarrollar consultas eficientes y escalables, optimizando el rendimiento y evaluando estrategias de procesamiento distribuido. Se tomaron decisiones críticas en la arquitectura del sistema para maximizar la eficiencia y minimizar la latencia, evaluando alternativas de diseño con base en la escalabilidad y robustez.

Se analizó el impacto de utilizar combiners y se realizaron pruebas con conjuntos de datos variados, incluyendo distribuciones desbalanceadas para probar cómo ciertas elecciones de clave (key) afectan el balanceo de carga y el rendimiento. Estos experimentos permiten obtener una comprensión más profunda de cómo diferentes factores influyen en la eficiencia del procesamiento distribuido.

2. Diseño de los componentes de cada MapReduce

2.1. Elección de la Key del Mapa Inicial

En el diseño de un sistema MapReduce, **la elección de la clave (key) para el mapeo inicial es fundamental para asegurar una distribución equilibrada de la carga entre los nodos**. La clave debe ser seleccionada teniendo en cuenta la naturaleza del dataset para evitar sobrecargar un solo nodo. En un escenario ideal con un número considerable de nodos, la distribución equitativa de los datos es crucial.

Como ejemplo, para la consulta de las top 3 infracciones por barrio, inicialmente podríamos pensar en utilizar el barrio como clave. Sin embargo, esta elección puede llevar a una distribución desigual si ciertos barrios concentran un alto porcentaje de las infracciones. Por ejemplo, en un dataset de infracciones de tránsito en una gran estado, algunas ciudades de mayor concentración demográfica o industrial, pueden tener una mayor densidad de población y, por lo tanto, más infracciones. Si el barrio se utiliza como clave, los nodos que procesan esos barrios podrían sobrecargarse. El tiempo en un proceso en paralelo es el tiempo del que más tarda, por lo que en el peor caso podría ser como no distribuir la data.

Para evitar este problema, **se decidió utilizar una clave primaria (PK) autoincremental** como clave para la distribución de los datos. Esto asegura que los tickets se distribuyan aleatoriamente entre los nodos, equilibrando la carga de trabajo. Sin embargo, dependiendo de la naturaleza de la agrupación requerida por la consulta, puede requerir una etapa adicional de procesamiento para agrupar los datos de interés.

2.2. Componentes

Primero se detalla la idea general, que en principio valdría para cualquier consulta, salvo excepciones. En tal caso se nombran en una sección aparte.

Los **mappers** toman cada ticket con un ID particular y emiten pares clave-valor en la forma (clave de interés, 1). Cada ticket se sube a la red con la información pertinente para cada consulta. Esto es simplemente para agilizar los testeos y no sobrecargar la red, aunque se entiende que en un ambiente operativo y del día a día estos datos ya están cargados.

Los **combiners** realizan una operación similar a la de los reducers pero a nivel local en cada nodo. Agrupan los valores (1s) localmente para reducir la cantidad de datos que necesitan ser transferidos a través de la red en la fase de reducción.

Los **reducers** agrupan los valores recibidos de los mappers y combiners, sumando los 1s para cada clave de interés. Este proceso consolida los datos a nivel global.

Los **collators** trabajan con un orden de magnitud mucho menor después de la fase de reducción, realizando operaciones finales como el ordenamiento y selección de los top N elementos que son poco costosas dada la magnitud de la información que le llega, pero no es siempre así...

2.2.1. ¿Y si el collator trabaja con mucha información?

En el caso de la Query 4, que requiere encontrar las placas de peor comportamiento por barrio, el orden de magnitud de los datos que llegan al collator es considerablemente alto. Los datos están compuestos por pares (barrio, placa), donde la cantidad de barrios es relativamente pequeña pero la cantidad de placas puede ser muy grande.

En ese caso hay dos opciones:

1. Para manejar este gran volumen de datos, se puede realizar un MapReduce adicional. Si el collator recibiera un dataset con un orden de magnitud muy alto, el procesamiento se volvería ineficiente, por ejemplo un ordenamiento sería muy costoso. Por lo tanto, se optó por dividir el trabajo en dos fases de MapReduce.

2. Se puede pensar desde cero el MapReduce, utilizando otra key que no sea el entero incremental, y que permita reducir la magnitud de la información en pasos anteriores como el mapper o el reducer.

3. Cuestiones particulares sobre la implementación

3.1. Parallel Read

Para abordar eficientemente la lectura de archivos grandes, optamos por un enfoque secuencial con procesamiento concurrente. Esto significa que utilizamos un solo hilo para leer el archivo CSV línea por línea, colocando cada línea en una cola. Luego, múltiples hilos consumen las líneas de esta cola, procesando y filtrando los campos de interés para luego ser almacenados en el IMap de Hazelcast.

Como resultado directo de esta implementación, logramos reducir significativamente el tiempo de procesamiento de archivos grandes, como por ejemplo, el archivo CSV de NYC con 15 millones de líneas que pasó de más de 1 hora a entre 3 y 4 minutos (que consideramos razonable).

3.2. Multas de cualquier otra ciudad

Para ser lo más permisivos posible al leer archivos CSV de diferentes formatos, diseñamos un sistema configurable que permite al usuario definir la estructura del CSV a través de un archivo JSON y manejar diferentes formatos de fecha.

El usuario debe subir un archivo JSON que especifica el índice de cada columna de interés entre los campos mencionados en el trabajo práctico. Esto permite una flexibilidad considerable, ya que el sistema puede adaptarse a diferentes estructuras de archivos CSV sin necesidad de modificar el código fuente. El usuario además puede subir archivos que tengan más columnas y el programa se encargará de tomar sólo las de interés.

Luego, la clase DateFormats permite agregar diferentes tipos de formatos de fecha, facilitando la inclusión de CSV con distintos formatos de fecha. Las fechas suelen variar más que otros campos en los datasets, por lo que esta flexibilidad es esencial.

Para los demás campos optamos por elegir tipos que engloben a lo que se encuentra usualmente. Por ejemplo, para el monto usamos Double, ya que si es entero o con decimales no estamos perdiendo información.

3.3. TopNSet

TopNSet es una colección que se encarga de mantener siempre N elementos ordenados para cada key. La idea era usarla para la query 2 y 3, pero al final sólo sirvió para la 2. De esta manera dejamos a la segunda consulta como genérica en vez de “top 3”. Igualmente para respetar la consulta se pasa como parámetro 3 por defecto.

4. Resultados

En todos estos resultados se utilizó la misma configuración del sistema, salvo que se diga la contrario: 3 nodos, y 5 datasets con diferentes cantidades de líneas: 100k, 500k, 1M, 5M y 15M.

4.1. Tiempo para cada query según cantidad de nodos

4.2. Con y sin Combiner

4.3. Dataset aleatorio y desbalanceado