MAP REDUCE FEST

Challenge 4

José Marcos Barreto
mbarreto@fing.edu.uy

Bloom Filter

Bloom Filter es un algoritmo que permite, con poca utilización de espacio filtar información. En el ejemplo propuesto, se fitran todos los comentarios de stackoverflow que contienen determinadas palabras.

Agrego hotlist al DFS: bin/hadoop dfs -put hotlist hotlist_dir
Ejecuta el driver que lee el hostlist y crea un BloomFilter a partir de los parámetros de entrada:
Numero de elementos ***n***
Factor de error ***f***

bin/hadoop jar BloomFilterDriver.jar BloomFilterDriver hotlist_dir 100 0.1 hotlist

"Reading line: scala
Reading line: akka
Reading line: play!
Reading line: android
Reading line: Android Studio
Reading line: IntelliJ
Reading line: groovy
Reading line: grails
Training Bloom filter of size 479 with 3 hash functions, 100 approximate number of records, and 0.1 false positive rate"

Los valores de "numero de elementos" y "factor de error" son primordiales. El BloomFilter se crea a partir de ellos, creando un vector de bits. Estos bits se prenderán dependiendo de las funciones de hash y de la aplicaciones de dichas funciones de hash sobre las keys.

Se utilizaron los comentarios de stackOverflow de Noviembre del 2011 (aproximadamente 2 GB), los cuales se los partio en archivos de 500MB con la función split en Linux. Posteriormente, se ejecutó el BloomFilterDriver para una selección particular de palabras y luego se ejecutaron varias corridas del BloomFilter aumentando en cada una 500MB la información a analizar.

El resultado de las ejecuciones está ligado a la elección de los parámetros de entrada. Se realizaron pruebas con "numero de elementos" = 2 – 4, esto genera un vector con pocos elementos y pocas funciones de hash, provocando una enormidad de falsos positivos haciendo de que el filtro no sea aplicado. Utilizando valores mas grandes, el comportamiento del Bloom Filter es correcto pero utiliza más espacio y es más lento.

La utilización del cache distribuido es necesaria para que el BloomFilter este presente en todas las tasks que van a utilizar dicho filtro. La utilidad de DistributedCache asegura esto. En Driver crea el BloomFilter y lo guarda y luego, en le método setup del Mapper se levanta el BloomFilter a aplicar en los comentarios de stackOverflow. El rol de la implementación del método setup es cargar el BloomFilter entrenado desde el cache distribuido. En este caso, los keys de output es la misma entrada, no tiene sentido usar un combiner ni un reducer si no se va a agrupar resultados, únicamente a filtrar parte de los datos de entrada. Por dicha razón en el main se setea el valor *"setNumReduceTasks(0)"*.

Resultados de ejecución:

Ejecucion 500MB:
Tiempo de ejecución: 56 segundos.
Map input records=529650

Ejecucion: 1GB
Tiempo de ejecucion: 93 segundos.
Map input records: Map input records=1020246

Ejecucion 1.5GB
Tiempo de ejecución: 145 segundos.
Map input records: 1493871

Ejecucion 2GB:
Tiempo de ejecución: 215 segundos.
Map input records=1962783

Distributed Grep

A diferencia de la propuesta del libro, se decidió utilizar las clases de java Pattern y Matcher para implementar el grep, pues me pareció que la implementación por defecto era más similar a la por defecto en sistemas Linux. La expresión regular a aplicar se obtiene como un parámetro de entrada y se guarda como configuración. Posteriormente, en le setup del mapper, se crea la clase Pattern con dicha expresión regular.
El método map luego únicamente verifica si dicha expresión aparece en el comentario. De aparecer se la agrega como output.
En el distributed Grep tampoco se utilza reducers.

El SRS se utiliza para obtener un subset de datos de forma aleatoria, donde cada línea tenga la misma probabilidad de pertenecer al conjunto solución. Para ello, se utiliza un parámetro de configuración "*filter_percentage*" y un random, si el random es menor al porcentaje, se retorna dicha línea como solución. El SRS es una solución simple para obtener datasets menores aleatoriamente, sin afectar la calidad de los datos.

Comando de ejecución:
bin/hadoop jar DistributedGrep.jar DistributedGrepMain input output scala

Resultado de ejecución:

Ejecucion 500GB:
Tiempo de ejecución: 40 segundos.
Map input records=529650

Ejecucion 1Gb:
Tiempo de ejecución: 105 segundos.
Map input records=1020246

Ejecucion 1.5Gb
Tiempo de ejecución: 158 segundos.
Map input records=1493871