MapReduce Fest - Desafío 4
Filtrado

Lorena Etcheverry
Instituto de Computación,
Facultad de Ingeniería,
lorenae@fing.edu.uy

27 de junio de 2013

Índice

1. Bloom Filters

Los *Bloom filters* son vectores de bits (secuencia de 0s y 1s) que permiten representar en forma compacta un conjunto de elementos. Estas estructuras de datos están diseñadas no sólo para minimizar el espacio que ocupan si no que también para hacer eficiente el chequeo de pertenencia o no de determinado elemento al conjunto.

Inicialmente el filtro tiene todos sus bits en 0. Durante la fase de entrenamiento una serie de funciones de hash es aplicada a cada uno de los miembros del conjunto que se desea representar, y esto determina qué posiciones o bits dentro del filtro deben valer 1. Si alguno de estos bits ya valía 1 su valor no cambia, si valía 0 pasa a valer 1.

Para determinar la pertenencia o no de un elemento al conjunto representado por el *Bloom filter* basta con aplicar las mismas funciones de hash al elemento y comparar si todos los bits que deberían estar en 1 en el filtro están efectivamente marcados. Basta con que uno de ellos esté en 0 para concluir que el elemento no está en el conjunto, pero existe la probabilidad de que ocurran falsos positivos, es decir que un elemento que no pertenezca al conjunto tenga la misma representación que uno que si pertenece. Cada *Bloom filter* posee cuatro parámetros: el largo del filtro (m), el tamaño del conjunto que representa

(n), la probabilidad de falsos positivos (p) y la cantidad de funciones de hash que utiliza (k).

La implementación de *Bloom filters* de Hadoop considera como parámetros del constructor m, k y el tipo de función de hash a utilizar. Para calcular los valores de estos parámetros se recurrió a las ecuaciones que se presentan en el Apéndice A de [1], las cuales se transcriben a continuación. El largo del filtro (m) puede calcularse a partir del tamaño del conjunto a representar (n) y la probabilidad de falsos positivos (p) que se desea para el filtro. La ecuación (1) presenta la expresión que permite determinar dicho valor de m, el cual conduce a su vez a un valor óptimo para la cantidad de funciones de hash a utilizar (k), el cual se calcula mediante la ecuación (2).

$$m = \frac{-n.ln(p)}{ln(p)^2} \tag{1}$$

$$k = \frac{m.ln(2)}{n} \tag{2}$$

2. Filtrado en Map-Reduce usando Bloom Filters

Como ya se mencionó en la Sección 1 Hadoop posee su propia implementación de *Bloom filters*. Éstos son utilizados para representar conjuntos de valores en forma compacta, y por lo tanto son útiles a la hora de filtrar valores. La figura 1 presenta el patrón de filtrado usando Bloom filters. Este consiste en una fase inicial de entrenamiento del filtro, para luego colocar el filtro en el sistemas de archivos de Hadoop (HDFS) y así poder compartirlo entre los diferentes *mappers*. Cada *mapper*, mediante el método `setup`, carga en memoria el filtro y lo utiliza para determinar o no la pertenencia de elementos de la entrada al filtro. Sólo los elementos que pertenezcan al filtro son volcados a la salida. Este patrón sólo implementa *mappers*, en la fase de reducción no se realiza procesamiento y tampoco tiene sentido utilizar *combiners* dado que no hay que realizar operaciones de agregación sobre las entradas, sólo filtrar las que no sirven.

2.1. Entrenamiento del Bloom Filter

Para entrenar el *Bloom filter* se toma como entrada el conjunto de palabras que se quiere representar y la probabilidad de falsos positivos que se desea tener, y se genera una instancia de la clase org.apache.hadoop.util.bloom.BloomFilter que luego es serializada al HDFS. Este código se basa en código de [1].

Para la tarea 1.2, que requería filtrar los comentarios en los cuales se usara al menos una palabra de una lista de palabras claves, previo al entrenamiento del filtro se generó esta lista de palabras. La lista de palabras se obtuvo seleccionando un subconjunto de las palabras más frecuentes dentro del archivo *comments.xml*. Se utilizó este archivo como entrada para el proceso de Word Count implementado previamente en el curso, y se obtuvo como salida una lista de palabras y su cantidad de ocurrencias, ordenadas alfabéticamente. Dado que se deseaba una lista de palabras más frecuentes se procedió a ordenar esta salida por cantidad de ocurrencias, para luego descartar las primeras 100 palabras por tratarse de palabras frecuentes del idioma inglés y no palabras

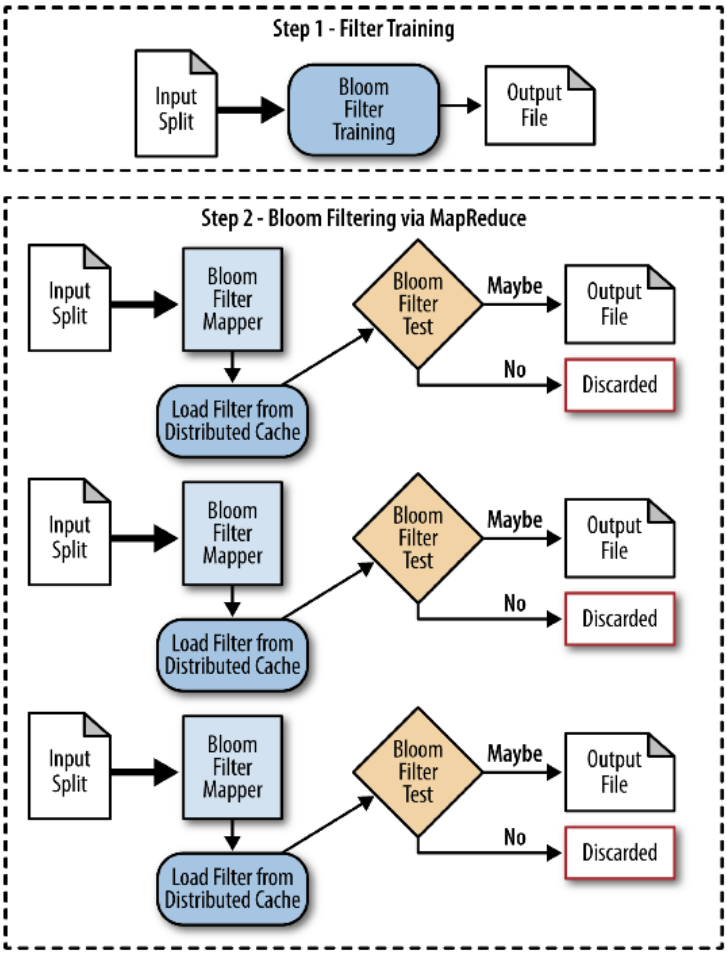

Figura 1: Patrón de diseño filtrado usando *Bloom filters*[1]

clave de la temática. La figura 2 presenta los comandos ejecutados, a partir de la salida de Word Count, para obtener la lista de palabras claves.

```
1 cat part-r-00000 | sort -k2gr |head -n800| tail -n700|cut -f 1 > hot_words.txt
```

Figura 2: Comandos ejecutados para preparar los datos de entrada.

En la tarea 1.3 se requería entrenar el filtro con los identificadores de usuarios con reputación mayor o igual a 1500 puntos. Para obtener esta lista se modificó el código utilizado para generar un índice inverso para generar una lista de usuarios, ordenados por reputación. Luego, se extrajo de esa lista a los usuarios con reputación mayor a 1500. El archivo `users.xml` utilizado tiene 379050 usuarios, de los cuales 7629 cumplen con la condición buscada.

3. Uso de Bloom Filters para filtrar consultas

Los *Bloom filters* suelen usarse para reducir la cantidad de consultas que se realizan sobre bases de datos externas, por ejemplo HBase. Para implementar esta tarea se procedió a entrenar un filtro con los identificadores de usuarios con reputación mayor o igual a 1500. El objetivo de la tarea es reducir la cantidad de consultas que se realizan sobre una base HBase, filtrando las mismas previamente.

Esta parte del desafío no se completó, debido a que no se consiguió cargar a partir del archivo XML una tabla en HBase para realizar las pruebas.

4. Top-ten

El objetivo de este parte consiste en obtener los 10 usuarios de mejor reputación (*top-ten*) a partir de una lista de usuarios. Este problema responde al problema general *top-k* que consiste en obtener a los k primeros elementos de un conjunto ordenado según algún criterio. La estrategia directa para resolver este problema consiste en ordenar el conjunto para luego quedarse con los k mejores, pero ordenar por completo un dataset grande usando Map Reduce es muy costoso. En un contexto de programación paralela resulta más eficiente ordenar, en cada *mapper*, la porción de entrada que le es asignada para luego enviar a la fase de *reduce* sólo los k mejores de cada *mapper*. Luego, en el reducer, se ordenan los registros y se envían a la salida sólo los k mejores. Es importante tener en cuenta de que esta estrategia funciona sólo si se usa un solo *reducer*. La figura 3 presenta gráficamente lo antes descripto para el caso k=10 [1].

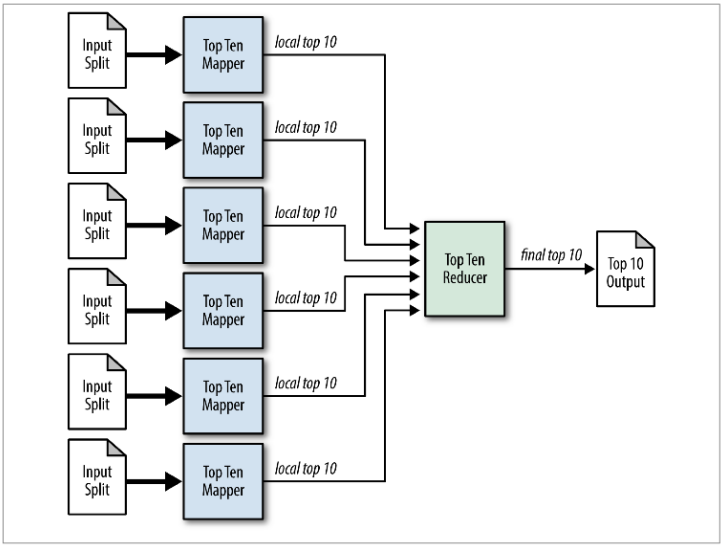

Figura 3: Patrón de diseño para obtener los top-k elementos de un conjunto[1]

4.1. Análisis de la solución propuesta en[1]

En la solución propuesta en cada *mapper* se utiliza un `java.util.TreeMap` para almacenar los k usuarios de mejor reputación. Esta clase implementa un árbol *red-black*, una variante de árboles binarios de búsqueda que garantiza O(log(n)) para las operaciones *containsKey*, *get*, *put* y *remove* [1]. Cada usuario que es procesado desde la entrada es almacenado en el `TreeMap`, considerando su reputación como clave de ordenación dentro de la estructura. Por otro lado, como sólo interesa almacenar los k mejores, si al insertar un elemento se llega a k+1 se borra del TreeMap el elemento con menor reputación. Este elemento, de acuerdo con la documentación, es accesible usando la operación `firstKey()`.

Una vez procesada toda la entrada el método `cleanup` se encarga de emitir los contenidos del `TreeMap` a la fase de *reduce*. El *reducer* propuesto usa la misma estrategia que el *mapper* para ordenar los usuarios por reputación y guardar sólo los k usuarios, almacenando las entradas en un `TreeMap`. Dado que sabemos que hay un sólo *reducer* en ejecución y un sólo grupo de entrada es posible generar la salida en el *reducer*, sin necesidad de hacerlo en el procedimiento *cleanup* del mismo.

Referencias

[1] Donald Miner and Adam Shook. *MapReduce Design Patterns Building Effective Algorithms and Analytics for Hadoop and Other Systems*. O'Reilly Media, Inc., 2012.

[1] `http://docs.oracle.com/javase/6/docs/api/java/util/TreeMap.html`