# Bloom Filtering

Un Filtro de Bloom es una estructura de datos, muy eficiente en uso de memoria, que puede utilizarse para verificar (probabilísticamente) la presencia de un elemento en un conjunto. El concepto fue introducido por Burton Howard Bloom en 1970.

El objetivo de este ejercicio es la aplicación de esta técnica para el procesamiento y filtrado eficiente de comentarios en StackExchange, en particular:

- Dado un conjunto de comentarios, obtener aquellos que contienen al menos una de un conjunto de palabras clave.
- Dado un conjunto de comentarios, obtener aquellos que corresponden a usuarios con un cierto nivel de reputación.

Puesto que los Bloom Filters son probabilísticos, pueden ocurrir falsos positivos, no así falsos negativos. Por lo tanto, los resultados de su aplicación en estos problemas serán inexactos (con una probabilidad de falso positivo dada en la construcción del filtro).

El código entregado está basado (aunque en algunos casos con importantes modificaciones) en el propuesto en el libro *Map Reduce Design Patterns*.

## Datos de Entrada

Como datos de entrada, se usaron los mismos archivos que en el challenge anterior. Corresponden a comentarios en sitios de StackExchange (extraídos del *Stack Exchange Data Dump March 2013*, http://www.clearbits.net/creators/146-stack-exchange-data-dump/contents).

- meta.scifi.stackexchange.com (1.8 MB – 1850 usuarios – 6,227 comentarios).
- android.stackexchange.com (9.8 MB – 23,155 usuarios – 3,8857 comentarios).
- english.stackexchange.com (47 MB – 29,337 usuarios – 168,054 comentarios).
- superuser.stackexchange.com (131MB – 153,202 usuarios – 531,604 comentarios).

*Nota: Por su tamaño, estos archivos no se adjuntan en el paquete entregado. Pueden descargarse de Dropbox: https://www.dropbox.com/s/us6ui125fdg4d37/Challenge04-Input.rar*

## Hot list

Instrucciones de ejecución:

```
bin/hadoop jar bloom.jar HotwordsRunner <wordlist_file> <input_dir> <output_dir>
```

Este programa:

- Entrena un filtro de Bloom a partir del archivo con la lista de palabras (una por línea).
- Coloca el archivo del filtro en el cache distribuido de Hadoop.
- Ejecuta mappers que leen el filtro y procesan los archivos de comentarios de la entrada, aplicándolo para quedarse con aquellos en que haya ("probabilísticamente") al menos una palabra en la hot list.
- Produce la lista de comentarios como resultado en el output. No se emplea un reducer pues no nos interesa hacer agrupaciones, sino únicamente obtener un subconjunto de los registros originales.

(Nota: el paso de entrenamiento del filtro de Bloom normalmente sería un ejecutable que se ejecuta "offline" y por separado del Map/Reduce en sí; en este caso se decidió "embeberlo" únicamente de forma de simplificar la ejecución de las pruebas).

**Elección de la tasa de falsos positivos**

Uno de los parámetros a configurar para este problema era la tasa de falsos positivos admitidos para el filtro de Bloom (parámetro que se usa para determinar el número de bits necesarios).

El problema de filtrado de comentarios por palabras especiales tiene una característica particular: el registro es seleccionado si *cualquiera* de sus palabras es aceptada por el filtro. Por lo tanto una tasa de error relativamente pequeña ve "magnificado" su efecto por la cantidad de palabras contenidas en cada comentario.

Supongamos que se establece **0.01** como la tasa de falsos positivos. Entonces la probabilidad de rechazar un comentario de *n* palabras que no contenga ninguna de las interesantes es 0.99 elevado a la *n*. Por ejemplo, con un comentario de 40 palabras (algo nada exagerado) la probabilidad de un falso positivo sería aproximadamente de **0.33**. Si, además, la proporción de comentarios interesantes con respecto al total fuese baja, los resultados estarían "inundados" de falsos positivos. Claramente el filtro de Bloom necesita una menor tasa para ser útil en este problema.

Ejecutando el programa sobre el conjunto de datos más pequeño (6,227 registros), un archivo de hotlist de 50 elementos y diferentes tasas de falsos positivos, se obtuvieron los siguientes resultados:

| Tasa | Bloom bits | Registros output |
|---:|---:|---:|
| 0.1 | 239 | 5,508 |
| 0.01 | 479 | 1,142 |
| **0.001** | **718** | **648** |
| 0.0001 | 958 | 563 |
| 0.000001 | 1198 | 559 |

Como era previsible, el número de registros seleccionados parece ir convergiendo hacia el valor real a medida que el filtro se hace más estricto. Se consideró aceptable a los efectos de ejecutar las demás pruebas el valor de **0.001** como tasa de falsos positivos.

**Resultados obtenidos**

Para el archivo de hot-words construido de ejemplo y la tasa de falsos positivos mencionada, los resultados obtenidos fueron:

| | meta.scifi | android | english | superuser |
|---|---:|---:|---:|---:|
| Map input records | 6,227 | 38,857 | 168,054 | 531,604 |
| Map output records | 648 | 7,326 | 12,295 | 121,754 |
| **Tiempo insumido** | **7s** | **8s** | **16s** | **32s** |

La relación entre la cantidad total de comentarios y aquellos que el filtro selecciona no es constante para todos los conjuntos de datos. Esto es perfectamente comprensible si se considera que el archivo de palabras “interesantes” que se utilizó tenía una gran proporción de palabras del entorno computacional (debug, disk, download, html, reboot, windows…) y de Android en particular (eclair, froyo, gingerbread, honeycomb…).

**Propósito del método setup() en el Mapper**

Este método ejecuta una vez por cada instancia de la clase Mapper que el framework cree, antes de cualquier invocación al método map(). En este caso se utiliza para deserializar el Bloom filter desde disco (más precisamente, desde el cache distribuido de Hadoop) de forma de dejarlo listo para la ejecución de los maps.

**Uso del Hadoop Distributed Cache**

Normalmente, los mappers acceden a los datos a procesar por medio del HDFS. Sin embargo, en este caso también se precisa del filtro de Bloom que se calculó antes de iniciar el proceso de Map/Reduce, serializado en un archivo. Este archivo es necesario para la ejecución de cualquiera de los mappers, y no es salvado por ninguno de ellos.

El mecanismo de Distributed Cache se aplica perfectamente a este escenario. Al agregar el archivo con el Bloom filter a este cache, el framework se encargará de copiarlo, a cualquier nodo que vaya a ejecutar alguna task del job. Y esta copia se realiza una sola vez, ya que se sabe que el archivo no será modificado durante la ejecución de todo el job.

El cache distribuido se manipula mediante los métodos addCacheFile() y getCacheFiles() de la clase DistributedCache. Cuando el mapper invoca este último, se le retornan las *copias locales* de los archivos del cache, *que ya fueron colocadas en el file system de ese nodo*, y que por lo tanto pueden ser accedidas con cualquier mecanismo estándar de I/O de Java (tal como FileInputStream).

**Uso de Combiner**

En este problema no tiene sentido usar un combiner exactamente por el mismo motivo que no se usa un reducer: no hay ningún procesamiento de agrupación a realizar. De hecho, la clave del output es lo único que interesa, por lo que se usa NullWritable como valor de todos los registros.

## Bloom Filter y HBase

Instrucciones de ejecución:

```
bin/hadoop jar bloom.jar ReputationRunner <input_dir> <output_dir> <users_file>
```

**Importante:**

- La dirección de la máquina local (en etc/hosts) debe ser 127.0.0.1 (esto no ocurre por defecto en Ubuntu, por ejemplo). Ver https://hbase.apache.org/book/quickstart.html
- HBase debe estar ejecutando (en localhost y en el puerto por defecto) *antes* de lanzar el proceso de hadoop.
- Puesto que este programa usa clases del cliente HBase además de las propias de Hadoop, los jars de HBase deben estar en el classpath de Java al momento de lanzar el job. La forma más sencilla de lograr esto es modificando el archivo hadoop-env.sh y agregándolos al HADOOP_CLASSPATH.[1]
- Cada ejecución de este proceso borra y recrea la tabla user_table, reinsertando los datos a partir del archivo de entrada.

Este programa:

- Crea una table en HBase (user_table).
- Lee el archive de usuarios suministrados, y carga su id, nombre y reputación en esta tabla.
- Entrena un filtro de Bloom a partir de los id de usuario con reputación mayor o igual a 1500.
- Coloca el archivo del filtro en el cache distribuido de Hadoop.
- Ejecuta mappers que leen los archivos de comentarios de la entrada, y seleccionan aquellos que corresponden a usuarios con reputación mayor o igual a 1500. Para esto, primero se utiliza el Bloom filter y recién si éste responde afirmativamente se consulta la tabla de HBase.

(Al igual que en el ejercicio anterior, los pasos de cargar la base de datos y entrenamiento del filtro están lógicamente separados de la ejecución del Map/Reduce en sí. Se ejecutan en forma encadenada de forma de simplificar la prueba).

[1] Por ejemplo: export HADOOP_CLASSPATH=<dir_hbase>/*:<dir_hbase>/lib/*

**Propósito del Bloom Filter**

A diferencia del ejercicio anterior, en el que el filtro de Bloom determina la salida del mapper, en este caso se trata únicamente de un mecanismo de optimización: realizar menos consultas a la base de datos HBase, excluyendo aquellos registros que, se sabe, no corresponden a los usuarios que interesan. Como un filtro de Bloom *nunca produce falsos negativos*, los usuarios que no pasan el filtro pueden descartarse sin temor.

**Overhead de HBase y optimización mediante el Bloom Filter**

De hecho, se obtendría exactamente el mismo resultado prescindiendo del filtro: para cada comentario, se consulta la tabla HBase de usuarios para obtener su reputación, y si es mayor al valor de umbral, se selecciona el registro.

Sin embargo, las consultas a la base de datos son costosas (especialmente en esta implementación "sencilla" que realiza una consulta individual por cada registro) por lo que descartar previamente muchos usuarios que, se sabe, no tienen la reputación necesaria, debería ser notoriamente distinto en performance.

En el caso del juego de datos de "English", por ejemplo, el procesamiento con la optimización demora **31 segundos**, mientras que sin la misma insume **45 segundos**.

## Distinct Users

Obtener el conjunto (sin repetidos) de todos los ids de usuarios en el archivo de comentarios. Se utilizan los mismos juegos de datos del ejercicio anterior.

Instrucciones de ejecución:

```
bin/hadoop jar distinct.jar Runner <input_dir> <output_dir> [withCombiner]
```

La implementación es extremadamente sencilla:

- El mapper procesa registro a registro el archivo de entrada, produciendo como clave de la salida el id de usuario de cada uno, y sin valor asociado.
- Puesto que el objetivo es obtener el conjunto de usuarios, y éste corresponde al conjunto de claves, el reducer simplemente emite la clave recibida, ignorando los valores (nulos).

**Uso de valores nulos**

Si bien Map/Reduce está basado en ítems clave/valor (agrupando por clave y siendo estos valores procesados por un mapper) a los efectos de este problema la clave es suficiente. En lugar de los valores se usa el tipo de datos NullWritable, una clase especial de Writable con serialización vacía (0 bytes – no se escribe ni lee nada del stream) de forma de no crear bytes inútiles a ser transferidos entre mappers y reducers.

**Uso de combiner**

Se puede apreciar a simple vista que este problema es ideal para el uso de combiners, y que la misma clase de reducer puede servir como combiner. La cantidad de reduce input groups (así como los bytes transmitidos) es mucho menor usando combiners.

| | **meta.scifi** | | **android** | | **english** | | **superuser** | |
|---|---|---|---|---|---|---|---|---|
| | s/cmb | c/cmb | s/cmb | c/cmb | s/cmb | c/cmb | s/cmb | c/cmb |
| Map output records | 6,070 | 6,070 | 38,028 | 38,028 | 165,156 | 165,156 | 514,543 | 514,543 |
| Reduce input records | 6,070 | **312** | 38,028 | **4,592** | 165,156 | **9,365** | 514,543 | **66,496** |