Alejandro Domínguez
Miguel Ángel Muñoz

Práctica de Algorítmica (Primavera 2017)

Para generar K funciones de hash, como pide el enunciado, decidimos buscar una estrategia que nos las permitiese generar “al vuelo” fuera cual fuese el valor de K. Nuestra solución fue utilizar el método de hashing por multiplicación del CLRS (pág 263, 11.3.2). Una función de hash con este método queda definida por una constante A entre 0 y 1 que multiplica a la clave y una constante M bastante grande que multiplica la parte faccionaria de la clave por A. Este resultado módulo el tamaño del filtro indica la posición en el filtro que ha de ponerse a 1.

El primer problema que nos encontramos es que las claves a hashear iban a ser alfanuméricas, y nuestro método necesitaba que la clase fuera un entero, con lo cual quisimos representar cualquier combinación de caracteres y números como un entero en base 10 (haciendo un cambio de base 62 a base 10), este entero pasó a ser la clave que pasaríamos al filtro de bloom. Los valores [0-9a-zA-Z] son los 62 símbolos de nuestra base. Obviamente esto puede convertirse rápidamente en un número muy grande, O(62^n) para una clave de tamaño n, así que el tamaño de las claves será limitado.

Este problema pudimos reducirlo en cierta medida haciendo nuestro programa “case insensitive”, considerando sólo 36 símbolos y reduciendo nuestra base a 36.

Aún así, para números que deberían ser fácilmente tratables con doubles, del orden de 10^25, teníamos que el producto de la clave por A nos estaba dando cero, cosa que no pasaba para la clave “MiguelAngel”, pero sí para “MiguelAngelMunoz”.

Por este camino teníamos muchos problemas de precisión, así que decidimos utilizar una librería de precisión arbitraria para poder manejar claves de cualquier tamaño sin problemas. Elegimos GNU MP Bignum Library[1]. Utilizando esta librería conseguimos solucionar los problemas y generar las funciones de hash que necesitábamos muy rápido.

Probando nuestro método, resultaba que todas las claves, por muy diferentes que fuesen, se estaban hasheando a las mismas casillas del filtro de bloom con demasiada frecuencia. Tras realizar muchísimas pruebas y comprobar que nuestro código funcionaba correctamente, concluimos que el problema estaba en nuestro método, y decidimos probar otro: usando el método de la división de CLRS(pág 269). Dado un número de funciones de hash N, escogíamos los primeros N números primos a partir de un número que experimentalmente hemos visto que daba buenas distribuciones.

Para utilizar la encriptación SHA-256 hemos buscado en internet una implementación y la hemos utilizado tras probar que funcionara.

Midiendo el tiempo que nuestro algoritmo tarda en añadir una clave dada al filtro de bloom, sea encriptada con SHA o no, es del orden de 10μs.

1 http://gmplib.org/

A priori, probando unas pocas combinaciones, no parecía que hubiese una diferencia significativa entre hashear las claves encriptadas con SHA-256 o sin, si usábamos palabras sin sentido con cierta aleatoriedad. Nuestro primer test fue coger una lista de nombres, en concreto los 200 nombres más populares en España, y probar varios filtros de bloom y búsquedas con estas claves. Realizamos una búsqueda de 40 claves sin sentido, y para nuestra sorpresa, todas dieron falsos positivos; el problema radicaba en nuestra selección del número primo, daba buenos resultados para filtros pequeños, pero muy malos para filtros grandes: habíamos cometido el error de no coger primos siempre más grandes que el tamaño del bloom filter, y por ello se estaban hasheando todas las claves a un subconjunto del filtro. La siguiente prueba no produjo ningún falso positivo.

Datos del programa:
Tiempo que tardamos en crear un bloom filter:
Las inicializaciones tardan O(1), pero calculamos tantos primos como funciones de hash K nos pidan, y lo guardamos en un vector. El coste de averiguar si un número U es primo es O(sqrt(U)), no sabemos cuantos números tendremos que tratar hasta conseguir K primos, pero al menos tardaremos Ω(K·sqrt(U)).

Tiempo de añadir una clave a un filtro de bloom de tamaño M y con K funciones de hash:
Tardamos O(1) en generar una función de hash porque hemos guardado los primos antes (los usamos como módulos en el método de la multiplicación) y O(1) en poner el bit correspondiente del bloom filter a 1. Hacemos esto K veces para cada clave, con lo que tardamos O(K).
De esto se deriva que el tiempo de introducir N claves es O(NK)

Tiempo en buscar si una clave C pertenece probablemente al conjunto:
Volvemos a generar las K funciones ya que es muy barato de hacer, y comprobamos si los bits correspondientes a hashear C con las K funciones están activados. En el peor caso tardamos O(K).

La función de añadir que usa el método de la división tarda también O(NK), pero nos da malos resultados en el bloom filter, así que fue descartada temprano. Está comentada dentro del programa, sin embargo.

Tests

Los siguientes tests se realizan sobre un conjunto de claves compuesta por los 200 nombres más populares en España.

Test0

El test 0 consiste en buscar todas las claves para ver si son encontradas sin problemas en ambos filtros: el que utiliza sha-256 y el que no.
Suministramos *claus* al programa cuando nos pide claves, y suministramos *claus* cuando nos pide qué buscar. Cualquier combinación de funciones de hash y tamaño de bloomfilter consigue encontrar todas las claves correctamente.

Test 1

El test 1 contiene búsquedas de 40 cadenas de caracteres de longitud y contenido arbitrarias. Lo siguiente es la tabla de resultados donde la columna representa el número de funciones de hash y la fila el tamaño del bloom filter.

Bloom filter sin usar SHA-256

Tamaño del filtro	1 Hash	5 Hashes	10 Hashes	20 Hashes
512	37.5%	47.5%	70%	95%
1024	12.5%	10%	22.5%	57.5%
2048	2.5%	0%	0%	5%
4096	12.5%	0%	0%	0%

Bloom filter usando SHA-256

Tamaño del filtro	1 Hash	5 Hashes	10 Hashes	20 Hashes
512	37.5%	37.5%	82.5%	95%
1024	2.5%	2.5%	22.5%	55%
2048	5%	0%	0%	10%
4096	0%	0%	0%	0%

Test 2

En el test 2 se han generado 1000 enteros pseudo-aleatorios (Usando la el programa randomgen.cc que hemos escrito)

Bloom filter sin usar SHA-256

Tamaño del filtro	1 Hash	5 Hashes	10 Hashes	20 Hashes
512	29.5%	46.7%	74.2%	97.2%
1024	16.5%	9.6%	17.9%	63.1%
2048	7.6%	0.8%	0.8%	4.8%
4096	4.2%	0%	0%	0%

Bloom filter usando SHA-256

Tamaño del filtro	1 Hash	5 Hashes	10 Hashes	20 Hashes
512	31.4%	44.4%	82.7%	96.1%
1024	17.1%	12.1%	19.5%	61.5%
2048	8.5%	0.9%	0.9%	7.1%
4096	3.9%	0%	0%	0%

Test 3

El test 3 trata de 50 claves de un tamaño bastante reducido (entre dos y cuatro caracteres).

Bloom filter sin usar SHA-256

Tamaño del filtro	1 Hash	5 Hashes	10 Hashes	20 Hashes
512	26%	74%	76%	98%
1024	22%	40%	60%	72%
2048	8%	10%	30%	44%
4096	12%	16%	28%	44%

Bloom filter usando SHA-256

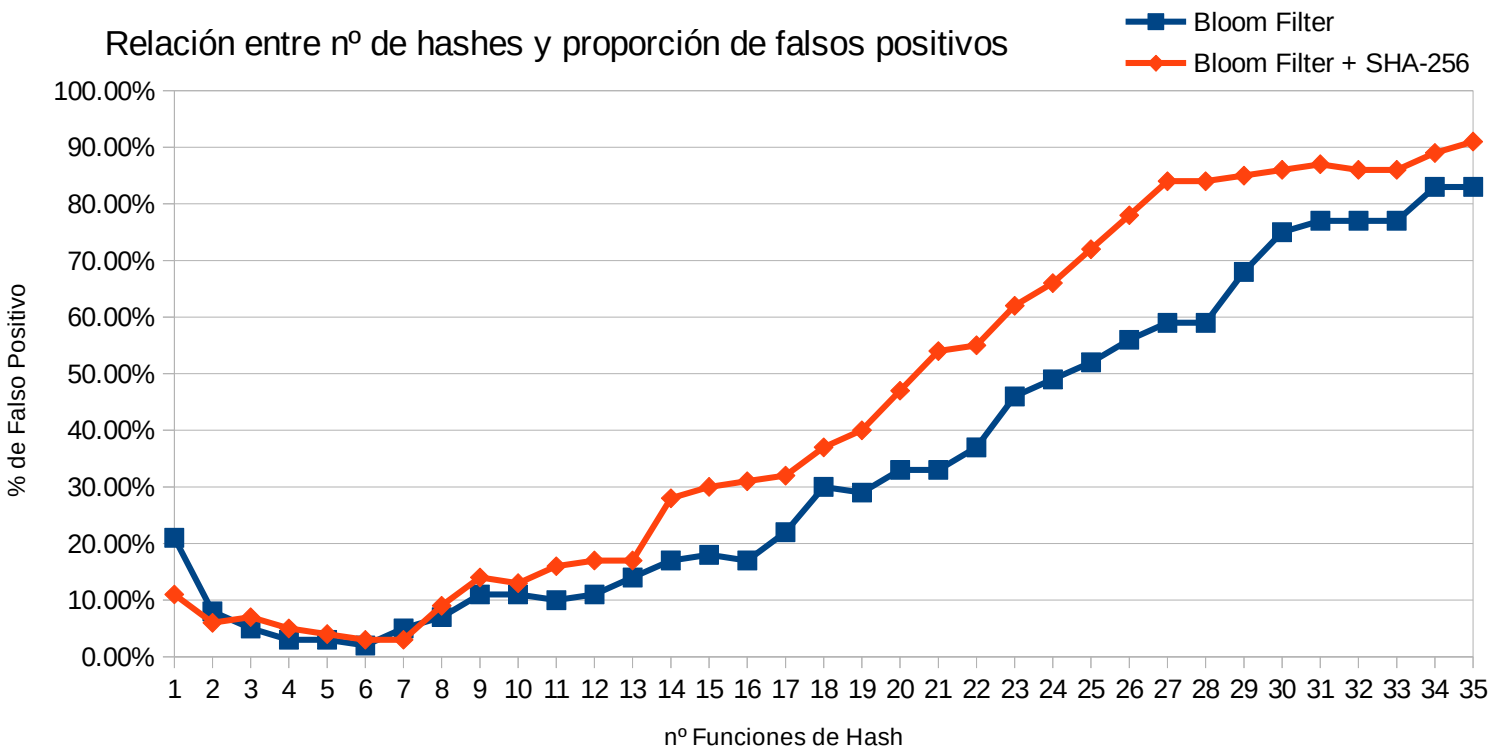
Tamaño del filtro	1 Hash	5 Hashes	10 Hashes	20 Hashes
512	34%	48%	78%	96%
1024	16%	8%	10%	56%
2048	12%	0%	0%	4%
4096	6%	0%	0%	0%

La diferencia en el tercer test es considerable. Esto se debe a nuestra codificación de las claves y nuestro método de generación de funciones: mientras que SHA-256 siempre genera claves largas independientemente del tamaño de la entrada, nosotros simplemente convertimos de base 36 a base 10. Cuando la clave es corta y el filtro es grande, los números primos que utilizamos para dividir son más grandes que las claves, haciendo que una clave pequeña hashee repetidamente al mismo sitio. En todos los tests parece ganar la variante que usa SHA-256, aun que en los primeros hay poca diferencia.

Algo que observamos de estas pruebas es que cuando el número de hashes es 1 hay más falsos positivos que cuando usamos mas hashes, pero si seguimos aumentando el número de hashes, vuelven a aumentar los falsos positivos. Este hecho no se ve en la tabla cuando el bloom filter es muy grande porque necesitaríamos todavía mas funciones de hash.
Para comprobar esto, realizamos el *Test 4.*

Test 4

Elegimos un filtro de bloom razonablemente grande, para este test no es importante el valor exacto, cogemos 1234. Variamos el número de funciones de hash para ver como afecta a ambas técnicas. Para las búsquedas usamos 100 enteros aleatorios.



Se demuestra lo antes dicho, pero dados los resultados de los tests anteriores esperábamos que los resultados de la versión con SHA-256 fueran mejores.
Vemos que al aumentar el número de funciones de hash se reducen los falsos positivos hasta llegar a un óptimo, en este caso el 6, y luego suben de nuevo: cuanto más saturado de unos está el bloom filter, más probables se vuelven los falsos positivos.

Conclusión

No tenemos resultados concluyentes para decidir si encriptar las claves con sha-256 antes de insertarlas en el filtro da mejores resultados en la probabilidad de falsos positivos, pero sabemos que es menos vulnerable al no tener el mal comportamiento cuando las claves son muy pequeñas.

NB: para detalles sobre el uso del programa refiérase a “LEEME.txt” en “/Sources”