Análisis de la eficiencia de los filtros de Bloom

David Escobar Rupalí López Alfonso Valderrama

12 de mayo de 2021

Resumen

Un filtro de Bloom es una estructura de datos que permite reducir el tiempo asociado a una serie de consultas en disco, mediante el almacenamiento de las palabras más comúnmente consultadas en memoria principal. En este informe se presenta un análisis sobre su eficiencia temporal y espacial, mediante la simulación de una situación de la vida real. Los resultados muestran que la cantidad de falsos positivos generados por la estructura empleada disminuye mientras mayor es su vector de bits respecto de la cantidad de inserciones esperadas, de manera que su efectividad no está acotada. De esto se infiere que la posibilidad de utilizar un filtro de Bloom como método fiable de hashing está limitada por la memoria del computador.

1. Introducción

El presente estudio tiene como propósito analizar los filtros de Bloom. Dado un archivo en disco que contiene una lista de palabras L, un filtro de Bloom B es una estructura de datos que almacena en memoria principal las palabras que se consultan más a menudo. Su comportamiento consiste en procesar cada consulta por una palabra p, resultando en un valor $B(p)$ tal que:

$$B(p) = \begin{cases} 1 & si\ la\ palabra\ p\ quizás\ está\ en\ L \\ 0 & si\ la\ palabra\ p\ definitivamente\ no\ está\ en\ L \end{cases} \tag{1}$$

Lo descrito significa que se sabe con certeza si una palabra no está en el archivo, se evita buscarla y así se disminuyen los accesos a disco.

Se estudia mediante experimentación la eficiencia temporal y espacial de los filtros de Bloom, además de la probabilidad de falsos positivos, observando qué tanto se asemejan los resultados a lo esperado teóricamente.

2. Filtros de Bloom

Un filtro de Bloom está determinado por su tamaño m y k funciones de hash $h_1, ..., h_k$, las cuales retornan valores en $[0, m-1]$. Se tiene un vector de bits V de tamaño m, donde inicialmente cada posición parte en 0.

Estas estructuras de datos soportan dos operaciones:

- **Insertar(*p*)**: Por cada función de hash h_i, se hace $V[h_i(p)] = 1$.
- **Revisar(*p*)**: Si para alguna función de hash h_i se cumple que $V[h_i(p)] = 0$ entonces se retorna 0, de lo contrario se retorna 1.

A continuación se pretende demostrar que efectivamente los filtros de Bloom cumplen:

(i) si `Revisar` retorna 0 sobre un elemento, entonces éste jamás se ha insertado.

(ii) si un elemento se ha insertado en el filtro, entonces `Revisar` retorna 1 sobre él.

Entonces, sea B un filtro de Bloom de k funciones de hash $h_1, ..., h_k$ y con su vector de bits V de tamaño m y sea p^* una palabra:

(i) $B.Revisar(p^*) = 0$
$\Rightarrow$ para alguna función de hash h_i^*, $V[h_i^*(p^*)] = 0$
$\Rightarrow$ para alguna función de hash h_i^*, $V[h_i^*(p^*)] \neq 1$
$\Rightarrow \neg(\forall h_i, V[h_i(p^*)] = 1)$
$\Rightarrow \neg(B.Insertar(p^*))$
Luego, p^* nunca se ha insertado en B. ■

(ii) $B.Insertar(p^*)$
$\Rightarrow \forall h_i, V[h_i(p^*)] = 1$
$\Rightarrow \nexists h_i, V[h_i(p^*)] \neq 1$
$\Rightarrow \nexists h_i, V[h_i(p^*)] = 0$
$\Rightarrow B.Revisar(p^*) = 1$■

3. Probabilidad de falsos positivos en filtros de Bloom

A veces, los filtros de Bloom indican que un elemento p *puede* haber sido insertado, aun siendo que tal elemento jamás se insertó. Diremos que este tipo de resultados son falsos positivos. Lo que sigue busca acotar la probabilidad de obtener falsos positivos.

Se asume que las k funciones de hash son totalmente independientes, y que retornan valores entre 0 y $m - 1$ de manera aleatoria y con distribución uniforme.

La probabilidad de que el bit i esté en 1 tras n inserciones es la probabilidad de que, al hacer nk operaciones de hashing, al menos una de ellas coloque el bit i en 1. Esto se traduce en la siguiente expresión:

$$Pr(E_i) = Pr(todos\ eligen\ i) + Pr(todos\ menos\ uno\ eligen\ i) + ... + Pr(solo\ uno\ elige\ i)$$

Notando que cada una de estas probabilidades corresponde a la de "elegir j operaciones de entre las kn para que no elijan i", todas corresponden a una distribución binomial. De esta forma, la expresión de arriba es equivalente a:

$$Pr(E_i) = \sum_{j=0}^{nk-1} \binom{nk}{j} \left(1 - \frac{1}{m}\right)^j \left(\frac{1}{m}\right)^{kn-j}$$

Esta distribución equivale a una suma de términos de un binomio de Newton de base $(1/m + (1 - 1/m)) = 1$, excepto porque falta el último término, que es la probabilidad de que ninguna inserción modifique i. Por consiguiente, la expresión puede reducirse a:

$$Pr(E_i) = 1 - \left(1 - \frac{1}{m}\right)^{kn}$$

Tomando m suficientemente grande, se puede tomar el límite:

$$\left(1 - \frac{1}{m}\right)^{kn} = \left(1 - \frac{1}{m}\right)^{m \cdot \frac{kn}{m}} \to (e^{-1})^{\frac{kn}{m}} = e^{-\frac{kn}{m}}$$

Con lo cual la probabilidad de que un bit i esté en 1 se aproxima a:

$$Pr(E_i) \approx 1 - e^{-\frac{kn}{m}}$$

Si los eventos E_i fueran independientes, la probabilidad de que haya un falso positivo (que todas las funciones de hashing, al preguntar por un elemento no presente, respondan 1) correspondería al producto de k probabilidades de que un bit esté en 1 y sea accedido por una función de hashing, es decir,

$$Pr(FP) = Pr(E_i) \cdot Pr(revisión\ de\ elemento\ elija\ i) = \left(\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)\left(1 - \left(1 - \frac{1}{m}\right)^{k}\right)\right)^{m}$$

Sin embargo, las probabilidades de esto no son independientes, pues el hecho de que un bit esté en 1 significa que una o más inserciones eligieron ese bit y no otro. Luego, la probabilidad de no elegir un bit está acoplada con la probabilidad de elegir otro bit.

Para poder determinar la probabilidad de un falso positivo es necesario tener en consideración la cantidad de bits que hay en 1 al momento de hacer la revisión. Dada una cantidad de bits x en 1, la probabilidad de un falso positivo es igual a la probabilidad de elegir alguno de los x bits en 1 (probabilidad x/m) k veces, es decir, $(x/m)^k$.

Suponiendo que la función está bien concentrada, es decir, $Pr(X = \lfloor E(X) \rceil) = 1$, la probabilidad de un falso positivo se calcula como sigue:

$$Pr(FP) = \sum_{x=1}^{m} Pr(FP|X = x) \cdot Pr(X = x) = Pr(FP|X = \lfloor E(X) \rceil) \cdot Pr(X = \lfloor E(X) \rceil)$$

$$= \left(\frac{\lfloor E(X) \rceil}{m}\right)^{k} \cdot 1 \approx \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k}$$

Se necesita una cantidad de funciones de hashing tal que la probabilidad de falsos positivos sea mínima. Para determinarla, se minimizará la función $f(k) = ln(Pr(FP|k* = k))$, la cual es equivalente en cuanto a minimización. Tomando la primera derivada, se obtiene

$$\frac{\mathrm{d}}{\mathrm{d}k} k \log\left(1 - e^{-\frac{kn}{m}}\right) = \log\left(1 - e^{-\frac{kn}{m}}\right) + \frac{kn}{m} \frac{e^{-\frac{kn}{m}}}{1 - e^{-\frac{kn}{m}}}$$

Igualando esta expresión a cero y reordenando, se obtiene la expresión

$$x \log(x) = (1 - x) \log(1 - x)$$

donde $x = e^{-\frac{kn}{m}}$. Una posibilidad para que se cumpla esta igualdad es que $1 - x = x$, es decir, $x = 1/2$. Reemplazando x y despejando k se obtiene $k = m \log 2/n$.

Derivando nuevamente la función a minimizar y reemplazando el k obtenido, se tiene que:

$f''(k = m \log 2/n) = 2 - 2 \log 2 > 0$

Con esto último, se prueba que el k encontrado minimiza f. Si reemplazamos este valor de k en la aproximación de la función de probabilidad de falsos positivos, se tiene que $P(FP) \approx 2^{-m \log 2/n}$ para valores grandes de m (es decir, se sigue una distribución descendiente exponencialmente), mientras que para valores pequeños de m, la probabilidad es muy cercana a 1.

La figura 1 muestra la probabilidad de un falso positivo según la proporción entre m y n.

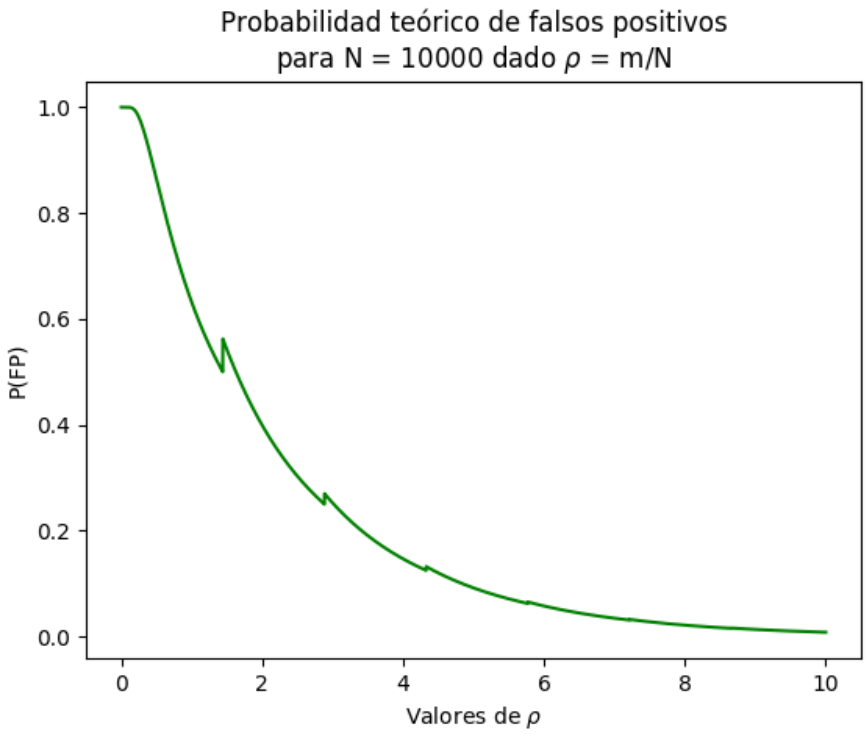


Figura 1: Distribución de probabilidad de falsos positivos para diferentes valores de $\rho = m/n$.

4. Hipótesis

De acuerdo con lo visto en la figura 1, si se utiliza k óptimo, la cantidad de falsos positivos por generarse debería disminuir conforme m aumenta con respecto a n. En particular, si $m << n$, debería haber casi tantos falsos positivos como palabras no insertadas en las consultas.

Para valores mayores, la cantidad de falsos positivos debería disminuir exponencialmente. Específicamente, una vez $m > n \cdot \log 2$, el total de falsos positivos debería ser aproximadamente la mitad del total de palabras buscadas no insertadas.

Generalizando, si $m > qn \cdot \log 2$ para algún $q \in \mathcal{N}$, el total de falsos positivos debería ser alrededor de 2^{-q} veces la cantidad de elementos buscados no insertados.

La distribución esperada sugiere que un filtro de Bloom solo resulta útil si el vector de bits es mayor al número de inserciones esperada, aunque no demasiado mayor, debido a que la disminución de la probabilidad de falsos positivos es exponencial conforme m aumenta.

La tabla 1 muestra la hipótesis para valores fijos de n, valores variables de m y cantidad de palabras ingresadas y preguntadas (estas últimas son el doble de las anteriores).

	Valor aproximado de $P(FP)$
$m << n$	1
$m < n/\log 2(x,\ k)$	$\in (1/2, 1)$
$m \in [n/\log 2, 2n/\log 2]$	$\in [1/4, 1/2]$
$m > 2n/\log 2$	$< 1/4$

Cuadro 1: Probabilidad aproximada de falsos positivos asociado al valor del tamaño de vector de bits m y el número de inserciones n.

5. Diseño experimental

Imagine que desea guardar n pares *(nombre de usuario, e-mail)* en disco. Cuando un individuo olvida la clave que le permite acceder a un sitio en particular, puede ingresar su nombre de usuario y así recibir un correo electrónico con instrucciones a seguir para recuperar su contraseña. Sin embargo, es recurrente que usuarios que nunca se han registrado realicen la acción descrita, lo que ocasiona que sus nombres de usuario se busquen infructuosamente en el disco, lo que desperdicia tiempo de computo.

Para analizar los filtros de Bloom desde un punto de vista práctico, se ha diseñado un experimento que pretende simular esta situación. Para dicho fin, se implementaron varios módulos, cuyo funcionamiento se explica como sigue.

5.1. Módulo generador de palabras aleatorias

Se programó un módulo que recibe dos enteros: $bigN$ y $smallN$, que corresponden al tamaño del universo de palabras y al tamaño del archivo a crear, que contendrá pares *(nombre de usuario, e-mail)*, respectivamente.

Se generan $bigN$ strings aleatorios que se almacenan en un archivo $universeFile$. Además, $smallN$ de ellos se concatenan con e-mails aleatorios, se insertan en un filtro de Bloom y en un archivo llamado $Lfile$.

Así, se obtiene un archivo sobre el cuál se podrá experimentar, y también se tiene un conjunto de consultas generadas al azar. Se sabe de antemano que de ellas habrá una fracción que efectivamente estará en $Lfile$ y otra que no, lo que garantiza que existe una adecuada aproximación a la realidad.

En el Anexo, sección B.1 se exhibe el pseudo-código correspondiente al módulo recién descrito.

5.2. Módulo buscador de palabras en archivo

Se implementó un buscador de palabras en un archivo que hace uso de la herramienta *grep*, cuyo pseudo-código se muestra en el Anexo, sección B.2.

5.3. Módulo de experimentación

Se creó un módulo para experimentar, el cual utiliza lo descrito en las secciones 5.1 y 5.2 para producir un archivo $Lfile$ de tamaño $smallN$, y para generar $bigN$ consultas al azar. Para cada nombre de usuario consultado, se revisa primero si el filtro de Bloom dice que éste está en $Lfile$. Solo en el caso afirmativo se procede a buscar.

El programa diseñado itera sobre un rango de valores de $bigN$ y sobre un rango de probabilidad de falsos positivos. Realiza una serie de iteraciones, para luego calcular la media de los resultados. Guarda información concerniente a los valores de parámetros como m y k, además de datos estadísticos relevantes, en un diccionario, para finalmente retornarlo.

En la sección B.3 del Anexo se observa el pseudo-código que representa el comportamiento recién explicado.

Adicionalmente, las especificaciones de la máquina con la que se corrieron los experimentos se encuentra en la sección A del Anexo.

5.4. Cálculo del tamaño de objetos

Para determinar el tamaño del filtro de Bloom se utilizó la biblioteca `objsize`, con la cual se calcula el tamaño en memoria de un objeto determinado. Se prefirió debido a que determina el tamaño de manera sencilla y encapsulada, eliminando capturas de tamaño dependientes de funciones aplicadas.

5.5. Parámetros utilizados en los experimentos

Se procede a declarar los valores de los parámetros que se usaron para llevar a cabo los experimentos, además de explicitar el razonamiento que llevó a escogerlos.

- Se tiene que para $bigN$, la cantidad de consultas generadas al azar, se emplearon los siguientes valores: $[2000, 5000, 10000, 15000, 20000]$. Se tomó esta decisión con el propósito de lograr que el tiempo de una búsqueda en disco fuera significativo.

- En cuanto a $smallN$ (el tamaño en filas del archivo que contiene los nombres de usuario e e-mails o, si se prefiere, el número de inserciones hechas en el archivo), se optó por usar $bigN/2$, es decir, $[1000, 2500, 5000, 7500, 10000]$, dado que de esta forma se garantiza que las consultas hechas sean tanto con nombres de usuario que efectivamente están en $Lfile$ como que no, teniendo así una aproximación a la realidad apropiada.

- Para m, se utilizó el siguiente rango:

  $[smallN/1000, smallN/100, smallN/50, smallN/10, smallN, smallN \cdot 2, smallN \cdot 3, smallN \cdot 4,$

  $smallN \cdot 5, smallN \cdot 6, smallN \cdot 7, smallN \cdot 8, smallN \cdot 9, smallN \cdot 10]$

  Los valores utilizados permiten, para cada valor de $smallN$, describir con suficiente precisión las secciones críticas de la distribución de falsos positivos esperada. Es por ello que los valores pequeños de m -respecto de $smallN$- están más agrupados, mientras los valores grandes están más extendidos y crecen linealmente.

  Dado que los m utilizados dependen de $smallN$, se define la variable $\rho := m/n$ para analizar las cantidades de falsos positivos e inserciones en disco. Usando ρ, los datos variables quedan normalizados, y es posible realizar una comparación más estandarizada de los resultados. Puesto que se espera que las distribuciones de falsos positivos sean equivalentes para cada conjunto de datos y lo que se busca es comprobar dicha distribución, la normalización del input no resulta en pérdida de datos.

- El parámetro k se obtuvo aplicando la fórmula $k = \lceil m/smallN \cdot log(2) \rceil$, la cuál asegura que se minimice la probabilidad de falsos positivos.

6. Resultados

A continuación se presentan los resultados de los experimentos.

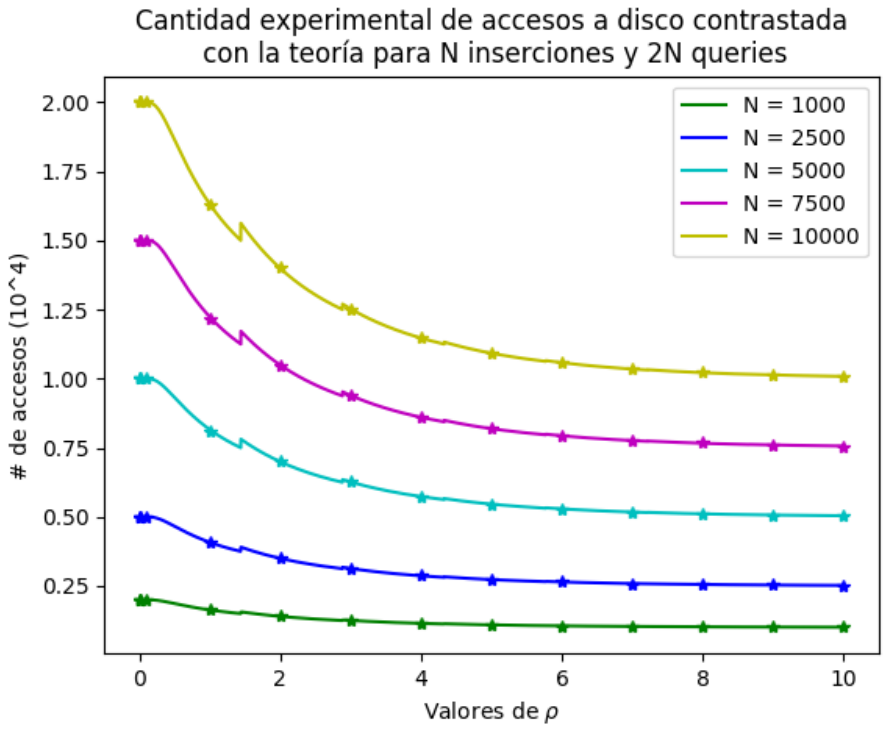


Figura 2: Se contrasta la cantidad de accesos a discos realizadas en el experimento contra la cantidad de accesos esperada por la teoría.

La figura 2 exhibe la cantidad de accesos a disco realizados. En las abscisas se observa la variable ρ, mientras que en las ordenadas la cantidad de accesos realizados. Los datos experimentales (los asteriscos) están comparados con su respectiva curva teórica.

Se observa un comportamiento similar para todas las curvas, las cuales muestran que a mayor valor de ρ, menor es la cantidad de accesos a disco que se harán.

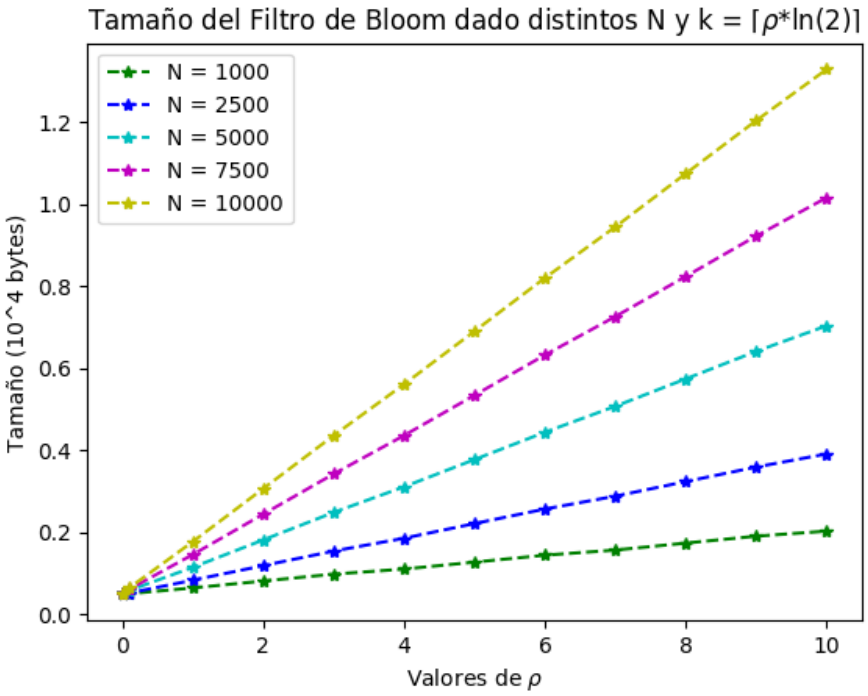


Figura 3: Se muestra el tamaño de los Filtros de Bloom para distintas cantidades de inserciones

La figura 3 muestra el tamaño en 10^4 $bytes$ de los filtros de Bloom en función del número de inserciones N y de ρ, de donde es trivial obtener k.

Se observa que el tamaño de los filtros crece proporcionalmente a como lo hace ρ. También, se advierte que existe una pendiente proporcional a N.

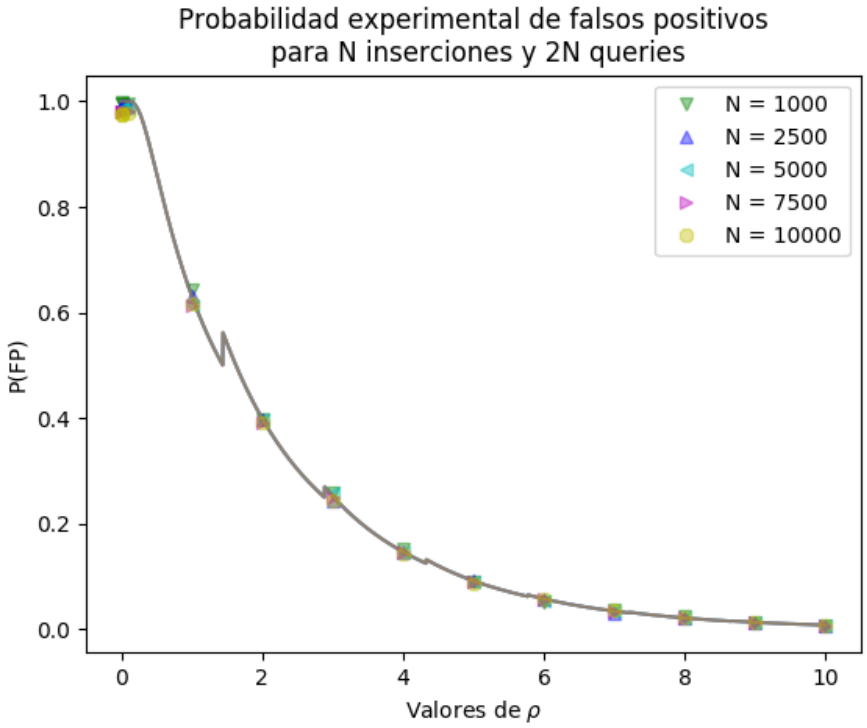
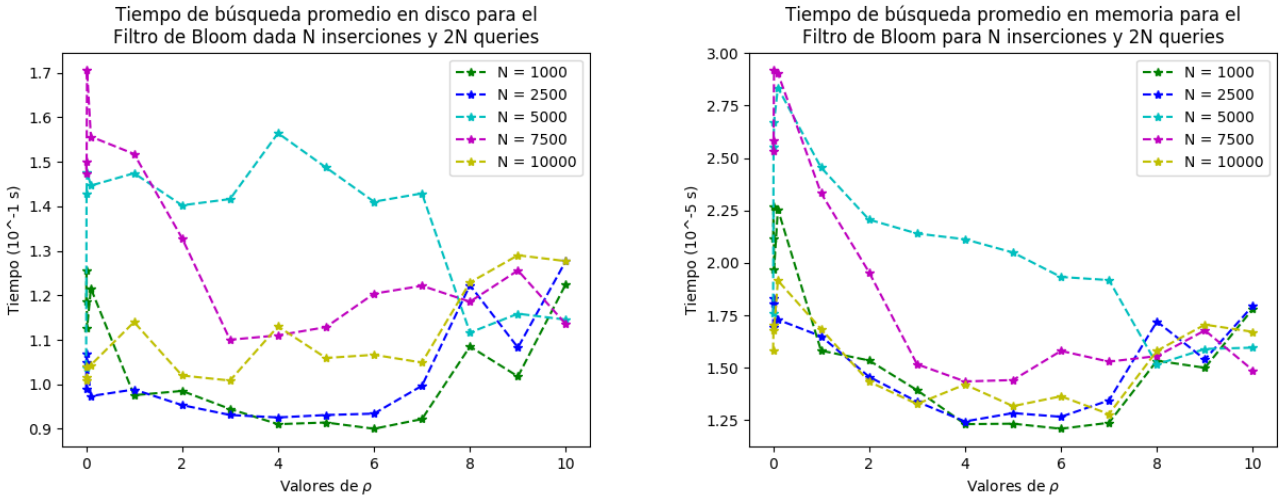


Figura 4: Probabilidades teórica y experimental dado el número de inserciones.

En la figura 4 se aprecian las probabilidades tanto teóricas como experimentales de falsos positivos para los filtros de Bloom, dados varios valores de N. Se observa que la posición de los puntos es prácticamente indistinguible de la curva teórica.



(a) Tiempo de búsqueda promedio en disco

(b) Tiempo de búsqueda promedio en RAM

Figura 5: Se muestran los tiempos de búsqueda promedio tanto para disco como para memoria usando los filtros de Bloom

Ambas sub-figuras de la figura 5 tienen como variable dependiente el tiempo. En el caso de la sub-figura

(a) se tiene el tiempo promedio de lectura en disco medido en 10^{-1} (s), mientras que en la figura (b) se mide el tiempo promedio de lectura en memoria principal de una palabra en el filtro de Bloom, esta vez medido en 10^{-5} (s).

No es posible destacar alguna correlación interesante, sin embargo, lo que capta la atención es el rango del tiempo promedio de búsqueda en RAM del filtro de Bloom, el cual es ínfimo, al punto de ser prácticamente despreciable.

7. Discusión

Los resultados obtenidos evidencian una alta correlación con el modelo propuesto para determinar la probabilidad de falsos positivos. La cantidad esperada de falsos positivos en función de los parámetros posee una distribución que se asemeja a la vista en los experimentos. La correlación muestra que una aproximación de la esperanza de los datos considerando resultado de hashing muy concentrado es un modelo fiable para determinar la distribución de probabilidad de falsos positivos.

Se aprecia que la probabilidad de falsos positivos disminuye exponencialmente en función de la razón entre la longitud del vector de bits utilizado en el filtro de Bloom y la cantidad de palabras insertadas. Esto se cumple exceptuando las instancias en que el vector de bits es considerablemente más pequeño que la cantidad de inserciones, en cuyo caso la probabilidad es altamente cercana a 1.

La distribución de probabilidad indica que el buen funcionamiento de un filtro de Bloom no es acotado, pues mientras más inserciones fueron hechas, más aumenta la probabilidad de obtener falsos positivos. El uso de estas estructuras, entonces, tiene una efectividad restringida por el tamaño de la memoria del computador a utilizar, y por ello no es posible aplicar un filtro de Bloom universal. Las buenas noticias es que el filtro de Bloom funciona con escaso margen de error para un vector de bits solo 7 veces más grande que la cantidad de inserciones esperada y dado que agotar la memoria supondría un número de inserciones cercano a 10^9 para las memorias actuales, es improbable que sea necesario.

Los tiempos de las distintas operaciones se muestran relativamente estables, teniendo los accesos a disco una ligera tendencia a aumentar conforme aumenta el tamaño del vector de bits respecto del número de inserciones en la mayoría de los casos. Las búsquedas en memoria duran un tiempo despreciable. Debido a esto, es pertinente concluir que el filtro de Bloom es efectivamente útil, pues significa que la duración de una serie de búsquedas en disco es proporcional a su total.

8. Conclusiones

En este informe se ha analizado la validez de un filtro de Bloom como estrategia de hashing. El estudio muestra que ésta es alta para casos en que se realiza un número acotado de inserciones.

Si se debe realizar una cantidad de inserciones pequeña respecto de la memoria disponible, el filtro de Bloom es una buena alternativa, dada la sencillez de su implementación. En caso de ejecutar un número de inserciones cercana o mayor al tamaño de la memoria de que se dispone (lo cual supone números extremadamente grandes), se recomienda optar por otra estrategia.

El filtro de Bloom se muestra como una opción atractiva para evitar accesos no deseados a servidores, discos duros o para evitar cualquier tipo de operación que sea costosa, debido a la escasa cantidad de memoria que consume al no guardar la información como tal. Incluso si la memoria fuese un problema, existe un margen bajo el cual la probabilidad de obtener un falso positivo es baja, de manera que se puede seguir ahorrando un número importante de operaciones costosas.

9. Anexo

A. Detalles técnicos del equipo usado para los experimentos

El computador utilizado para llevar a cabo los experimentos corresponde a un Lenovo 330S 14-IKB con las siguientes especificaciones técnicas:

- Sistema Operativo: Manjaro Gnome x64 20.1
- CPU: Intel Core i7 8550U @1.7 - 4 GHz.
- RAM: 12GB SODIMM-DDR4 2400MHz
- Almacenamiento: Crucial P1 500GB - 1900 MB/s Lectura, 950 MB/s Escritura.

B. Módulos usados para llevar a cabo los experimentos

B.1. Pseudo-código módulo generador de palabras aleatorias

Algoritmo 1 Pseudo-código módulo generador de palabras aleatorias

```
1:  procedure generateFiles(bloomFilter, bigN, smallN)
2:      for index ∈ [0...bigN] do
3:          randomUsername ← generateRandomString(randomLength)
4:          universeFile.write(randomUsername)
5:          if index < smallN then
6:              bloomFilter.insert(randomUsername)
7:              randomEmail ← generateRandomString(randomLength) + emailDomain
8:              line ← randomUsername + ' ' + randomEmail
9:              Lfile.write(line)
10: return bloomFilter, Lfile, universeFile
```

B.2. Pseudo-código módulo buscador de palabras en archivo

Algoritmo 2 Pseudo-código módulo buscador de palabras en archivo

```
1:  procedure searchUsernameInFile(username, LfileName)
2:      command ← f''grep'username''LfileName''
3:  return os.system(command)
```

B.3. Pseudo-código módulo de experimentación

Algoritmo 3 Pseudo-código módulo de experimentación

```
1:  procedure EXPERIMENT()
2:      output ← {}
3:      for bigN ∈ Nrange do
4:          smallN ← bigN/2
5:          currentOutput ← {}
6:          for m ∈ Mrange do
7:              k ← getK(m, smallN)
8:              for index ∈ iterations do
9:                  bloomFilter, Lfile, universeFile ← generateFiles(bloomFilter, bigN, smallN)
10:                 for usernameQuery ∈ totalUsernamesList do
11:                     if bloomFilter.check(usernameQuery) then
12:                         searchUsernameInFile(usernameQuery, LfileName)
13:             currentOutput[m] ← {k, m, timeBF, timeFile, falsePositives, diskAccess, sizeBF}
14:         output[bigN] ← currentOutput
15: return output
```