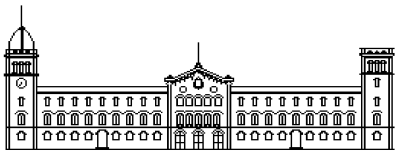

Trabajo final de grado

GRADO DE MATEMÁTICAS

Facultad de Matemáticas
Universidad de Barcelona

ESTIMACIÓN DEL TOP K

Carlos Martínez López

Director: Jaime Nebrera
Subdirector: Josep Vives

Abstract

The large amount of data stored in the last two decades makes us to create new algorithms that can treat this information and elicit the desired statistical analysis. In this paper we study the algorithms that find the most common elements k, top-k in a dataset or in a Data Stream. The algorithms must make a minimum memory usage, so the results will be an estimate, trying to minimize possible error. At the end of this work will be expose own implementation.

Resumen

Dada la gran cantidad de datos almacenados en las dos últimas décadas, nos hemos visto obligados a crear nuevos algoritmos que puedan tratar esta información y sonsacar los análisis estadísticos deseados. En este trabajo estudiaremos los algoritmos que nos encuentren los k elementos más frecuentes, top- k, en un conjunto de datos o en un $Stream$ de datos. Los algoritmos deberán hacer un uso mínimo de memoria computacionalmente hablando, así pues los resultados serán una estimación, intentando minimizar el posible error. Al final del trabajo se expondrá una implementación propia.

Índice

1 Introducción

Hace más de una década surgió la minería de datos como un campo de investigación muy activo que ofrecía técnicas de análisis para grandes volúmenes de información. Con el paso del tiempo, en algunos sectores se están generando datos a una velocidad lo suficientemente alta que provoca que no sea rentable su almacenamiento en términos de beneficios. Es por ello que la comunidad que estudia la minería datos se ha adaptado desarrollando nuevos algoritmos que analizan los datos según van llegando, estimando las consultas estadísticas pertinentes y almacenándolas para un futuro uso.

El presente trabajo se realiza en el marco del convenio de prácticas con la empresa Redborder y tiene como objetivo el estudio analítico de los distintos algoritmos existentes que tienen como finalidad encontrar una estimación de los k elementos más frecuentes, dentro de un conjunto de datos, y efectuar una comparación de ellos para hallar cuál es el algoritmo más eficaz. Para ello se dará al comportamiento de cada algoritmo dentro de un conjunto de datos con sesgo.

En este sentido los algoritmos analizados se dividen en dos tipos:

- Por un lado, los algoritmos basados en $sketch$ (coloquialmente se trataría de hacer un esbozo de todos los elementos del conjunto de datos) entre los que se analizarán los algoritmos *CountSketch[3]*, *CountMinSketch[7]*, *Grouptest[6]* y *hCount[5]*.

- Por el otro, los basados en contadores, es decir, que sólo almacenan un número determinado de elementos en una lista y cuando esta se llena se sustituyen por los nuevos elementos siguiendo un determinado patrón. Entre estos algoritmos se analizarán *Lossy Counting[1]*, *Frequent[14][4]* y *Space Saving[8]*.

El análisis matemático de estos algoritmos permitirá encontrar las carencias y las ventajas de cada uno y, partiendo de ello, proponer una mejora en el algoritmo que, a mi juicio, es el idóneo para nuestro objetivo planteado.

2 Estimación del top-*k*

Como ya hemos adelantado, el denominado top-k se define como los k elementos más frecuentes dentro de un conjunto de datos y su determinación, en la época actual, resulta esencial atendiendo a su potencial aplicación en los distintos sectores, ya sea desde el punto de vista comercial, ya sea desde el puno de vista estadístico, o bien de seguridad en la red (internet).

Un claro ejemplo de lo que acaba de señalarse puede verse en las tiendas on-line: el usuario adquiere un producto (al que llamaremos producto inicial) y normalmente se le propone la compra de otros productos que pueden estar relacionados con su adquisición o pueden resultarle de interés. La oferta de esos otros "productos relacionados" surge de la utilización de los algoritmos que se analizarán, ya que muestran el "top" de las compras adicionales que han realizado los distintos usuarios adquirentes de ese producto inicial o deseado.

Ilustración 1- Ejemplo de la aplicación comercial del top-k

Para la estimación del top-k ya se ha dicho que se utilizan distintos algoritmos divididos en dos clases. El primer grupo que va a tratarse es el que engloba los algoritmos basados en sketch, que utilizan las funciones denominadas hash, que no son más que funciones utilizadas en el campo de la criptografía, con el objetivo de reducir el rango del espacio de entrada. Sin embargo, el principal problema de los algoritmos sketch son las denominadas colisiones, ya que al aplicar la función hash y encriptar los datos, puede suceder que más de un elemento comparta el mismo contador. Es por ello que, normalmente, se utilizan varias funciones hash por algoritmo para reducir el error de las colisiones, lo que, a la postre, acaba por incrementar el uso de memoria en términos computacionales.

Este problema no se encuentra en el segundo grupo de algoritmos basados en contadores, en los que cada contador almacena un solo elemento, de modo que se elimina la probabilidad de colisiones. Sin embargo, estos algoritmos tienen como inconveniente que usan algún tipo de sistema para eliminar elementos monitorizados poco importantes en un principio, lo que supone la posibilidad de desechar elementos que, inicialmente, quizás no eran muy comunes pero que, con posterioridad, su frecuencia se ve incrementada y al volver a incluirse en la lista, estos entran con un error en su contador.

2.1 Algoritmos basados en sketch

2.1.1 CountSketch

Sean w y d parámetros con valores determinados. Sea $h_1, \dots, h_d$ funciones $hash$ con entrada los elementos del $Stream$, e, y salida $\{1, \dots, w\}$

$$h_1, \dots, h_d : \{e_1, \dots, e_n\} \rightarrow \{1, \dots, w\}$$

Y sean $s_1, \dots, s_d$ funciones $hash$ con entrada e y salida $\{+1, -1\}$.

$$s_1, \dots, s_d : \{e_1, \dots, e_n\} \rightarrow \{+1, -1\}$$

La estructura de datos *CountSketch* consiste en estas dos funciones $hash$ junto con una matriz $w \times d$ de contadores. Las funciones $hash$, h_i y s_i, son independientes dos a dos además de ser independientes entre ellas.

La estructura de datos soporta dos operaciones:

$Add\ (e)$: Para $i \in [1, d]$, $h_i[e] = h_i[e] + s_i[e]$.

$Estimate\ (e)$: Devuelve la $mediana_i\ \{h_i[e] \cdot s_i[e]\}$

Una vez definida la estructura de datos el algoritmo es sencillo. Para cada elemento, se usa la estructura de datos *CountSketch* para estimar su frecuencia, y guardar en una lista el top-k de los elementos vistos hasta ahora. El algoritmo de una manera formal seria:

Dado un $Stream\ e_1, \dots, e_n$ para cada $j = 1, \dots, n$:

1. $Add(e_j)$
2. Si e_j está en la lista se incrementa su contador de frecuencia. Sino, se añade e_j a la lista si $Estimate\ (C, e_j)$ es mayor que la menor frecuencia de los contadores de la lista. En este caso el valor con menos frecuencia será eliminado de la lista.

Propiedades CountSketch

Vamos a dar unas cotas para el cálculo de w y d. Para d vamos a usar $d = O(log \frac{n}{\delta})$, donde n es el total de elementos de nuestro conjunto de datos y δ es la probabilidad máxima que nuestro algoritmo falle. Para acotar w vamos a definir unos cuantos conceptos previos. Sea $f_e(l)$ la frecuencia de un elemento e, en el momento l de nuestro $Stream$, es decir, ya se han analizado l elementos. Sea $A_i[e]$ el conjunto de elementos que comparten el mismo valor de la función $hash\ h_i$, esto es:

$$A_i[e] = \{e' \mid e' \neq e, h_i[e'] = h_i[e]\}$$

Sea $A_i^{>k}[e]$ el subconjunto de elementos de $A_i[e]$ tales que no pertenecen a los k elementos más frecuentes. Para acabar sea:

$$v_i[e] = \sum_{e' \in A_i[e]} f_{e'}^2 \quad y \quad v_i^{>k}[e] = \sum_{e' \in A_i^{>k}[e]} f_{e'}^2$$

Con estas definiciones ya podemos introducir alguna de las propiedades.

Lema 1 La varianza de $h_i[e] \cdot s_i[e]$ está acotada por $v_i[e]$.

Lema 2

$$E\left[v_i{}^{>k}[e]\right] = \frac{\sum_{e' \in B} f_{e'}^2}{w}$$

Donde:

$B = \{Elementos\ del\ stream\ que\ tienen\ una\ frecuencia\ más\ pequeña\ que\ los\ del\ top - k\}$

Definimos:

Sea $SM_i[e]$ el evento que $v_i{}^{>k}[e] \leq 8 \cdot E\left[v_i{}^{>k}[e]\right]$ por la desigualdad de Marko:

$$\Pr\left[SM_i[e]\right] \geq 1 - \frac{1}{8}$$

Sea $NC_i[e]$ el evento que $A_i[e]$ no contenga ninguno de los elementos del top-k.

Si $w \geq 8k$ tenemos que:

$$\Pr[NC_i[e]] \geq 1 - \frac{1}{8}$$

Sea $SM_i[e](l)$ el evento tal que:

$$|h_i[e] \cdot s_i[e] - f_e(l)|^2 \leq 8 \cdot Var\left[h_i[e] \cdot s_i[e]\right].$$

Entonces,

$$\Pr[SM_i[e](l)] \geq 1 - \frac{1}{8}$$

y la cota de la probabilidad de la unión de estos tres eventos nos da,

$$\Pr\left[SM_i[e](l) \bigcup NC_i[e] \bigcup SM_i[e]\right] \geq \frac{5}{8}$$

A partir de ahora definiremos,

$$\gamma = \sqrt{\frac{\sum_{e' \in B} f_{e'}^2}{w}}$$

Con todo lo que tenemos definido hasta podemos dar una cota para el valor de w.

Lema 3 Con probabilidad $1 - \frac{\delta}{n}$, para todo $l \in [1, n]$,

$$|median\{h_i[e] \cdot s_i[e]\} - f_e(l)| \leq 8\gamma$$

Donde e es el elemento que ocurre en el momento l.

Demostración

Se probará que, con una alta probabilidad,

$$|\{h_i[e] \cdot s_i[e]\} - f_e(l)| \leq 8\gamma$$

Esto implicará que la mediana de $\{h_i[e] \cdot s_i[e]\}$ esta dentro del error acotado por el lema. Primero se observa como para un índice i, si todos los eventos anteriores ocurren, entonces:

$$|\{h_i[e] \cdot s_i[e]\} - f_e(l)| \leq 8\gamma$$

Dado que para un i fijado,

$$\Pr[|\{h_i[e] \cdot s_i[e]\} - f_e(l)| \leq 8\gamma] \geq \frac{5}{8}$$

Teorema 1 Si $w \geq 8 \cdot max\left(k, \frac{32 \cdot \sum_{e' \in B} f_{e'}^2}{(\epsilon f_e)^2}\right)$, entonces la estimación del top- k de los elementos ocurre como mínimo $(1-\epsilon)f_e$ veces y además todos los elementos con frecuencias al menos de $(1+\epsilon)f_e$ están entre la estimación del top- k.

Demostración

Por el lema 3 las frecuencias estimadas de todos los elementos se encuentran dentro de un factor aditivo 8γ del número verdadero de frecuencias. Así por dos elementos cuyo verdadero valor en las frecuencias difiere por más de 16γ, sus estimaciones identificarán correctamente cual es el de mayor frecuencia. Poniendo $16\gamma \leq \epsilon f_e$ se puede asegurar que solo los elementos que pueden reemplazar los verdaderos elementos más frecuentes en en la estimación del top-k son aquellos tales que su verdadera frecuencia es al menos $(1-\epsilon)f_e$.

$$16\gamma \leq \epsilon f_e \Leftrightarrow 16\sqrt{\frac{\sum_{e' \in B} f_{e'}^2}{w}} \leq \epsilon f_e \Leftrightarrow b \geq \frac{256 \cdot \sum_{e' \in B} f_{e'}^2}{(\epsilon f_e)^2}$$

Esto unido a la condición usada antes tal que $b \geq 8k$ prueba el lema. ■

Se ha explicado el algoritmo *CountSketch.* Hemos visto que los parámetros que hacen que el algoritmo use más o menos memoria son w y d. Así pues se ha dado una cota de estos dos según el error que se quiera llegar a cometer en la estimación del top- k.

2.1.2 CountMinSketch

La estructura de datos *CountMinSketch* con parámetros (ε, δ) es representada por una matriz de contadores de dimensiones $d \times w$: $count[1,1], \dots, count[d, w]$. Dado los parámetros (ε, δ). Definimos nuestra w y d como:

$$w = \left\lceil \frac{exp}{\varepsilon} \right\rceil \text{ y } d = \left\lceil ln \frac{1}{\delta} \right\rceil.$$

Cada entrada de la matriz es inicializada a cero. Además tendremos d funciones $hash$

$$h_1, \dots, h_d : \{e_1, \dots, e_n\} \rightarrow \{1, \dots, w\}$$

uniformes e independientes dos a dos donde el conjunto $\{e_1, \dots, e_n\}$ son los elementos del $Stream$.

El algoritmo se realiza de la siguiente manera. Cuando un elemento (e_i, c_i) llega, donde c_i es su valor propio (Por ejemplo en una transacción el elemento Carlos, e_i, ha abonado 20 euros, c_i). Se calcula su función $hash$ para cada una de las d funciones $hash$ que tenemos y le añadimos su valor de c_i a su respectiva celda, es decir para todo $j \in \{1, \dots, d\}$

$$count\left[j, h_j(e_i)\right] = count\left[j, h_j(e_i)\right] + c_i$$

en la siguiente figura se da una visión de cómo funciona el algoritmo.

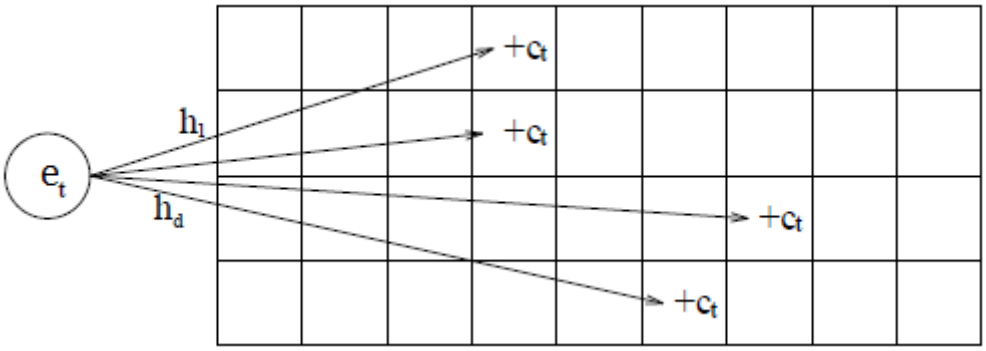

Ilustración 2 - Funcionamiento CountMinSketch

Definimos el estimador de la frecuencia de e_i como $\widehat{f_{e_i}} = min_j count\left[j, h_j(e_i)\right]$.

Propiedades CountMInSketch

Teorema 1 El estimador $\widehat{f_{e_i}}$ tiene las siguientes garantías:

$$f_{e_i} \leq \widehat{f_{e_i}}$$

además, con probabilidad al menos $1 - \delta$ se tiene

$$\widehat{f_{e_i}} \leq f_{e_i} + \varepsilon \cdot ||f||_1$$

donde f es el vector con todas las frecuencias reales de todos los elementos.

Demostración

Sea $I_{i,j,k}$ el indicador tal que:

$$I_{i,j,k} = \begin{cases} 1, & i \neq k \; y \; h_j(e_i) = h_j(e_k) \\ 0, & en \; los \; otros \; casos \end{cases}$$

Por la independencia dos a dos de las funciones $hash$ se tiene:

$$E\left[I_{i,j,k}\right] = Pr\left[h_j(i) = h_j(k)\right] \leq \frac{1}{w} = \frac{\varepsilon}{exp}.$$

Se define la variable $X_{i,j} = \sum_{k=1}^{n} I_{i,j,k} \cdot f_{e_k}$. Por construcción, $count\left[j, h_j(e_i)\right] = f_{e_i} + X_{i,j}$. Entonces, claramente, $\min count\left[j, h_j(e_i)\right] \geq f_{e_i}$. Por otro lado se observa que,

$$E\left[X_{i,j}\right] = E\left[\sum_{k=1}^{n} I_{i,j,k} \cdot f_{e_k}\right] \leq \sum_{k=1}^{n} f_{e_k} E\left[I_{i,j,k}\right] \leq \frac{\varepsilon}{exp} ||f||_1$$

y por la independencia de h_j y la desigualdad de Markov obtenemos:

$$\begin{aligned} Pr\left[\widehat{f_{e_i}} > f_{e_i} + \; \varepsilon \left|\left|f\right|\right|_1\right] &= \Pr\left[\text{ para todo } j, count\left[j, h_j(e_i)\right] > f_{e_i} + \; \varepsilon \left|\left|f\right|\right|_1\right] \\ &= \Pr\left[para \; todo \; j, f_{e_i} + X_{i,j} > f_{e_i} + \; \varepsilon \left|\left|f\right|\right|_1\right] \\ &= Pr\left[\text{para todo } j, X_{i,j} > exp \cdot E\left[X_{i,j}\right]\right] < \exp(-d) \leq \delta \end{aligned}$$

Así pues se ha explicado el algoritmo *CountMinSketch* y se ha dado una cota de los parámetros (ε, δ) en función del error que se quiera cometer. En todo los cálculos se ha supuesto que todo (e_i, c_i) tiene valor $c_i > 0$.

2.1.3 hCount

Dada una secuencia de transacciones en un $Stream$ de datos, donde una transacción puede ser insertar o eliminar un elemento k en el tiempo i, denotado como $t_i = eliminar(k)$ o $t_i = insertar(k)$, sin pérdida de generalidad vamos a suponer que los elementos k vienen dados en un intervalo de $[1, \dots, M]$. Denotamos por n_k al total de las ocurrencias del elemento k y N como la suma de todas las ocurrencias de los elementos en el $Stream$. Entonces la frecuencia de cualquier elemento e se denotara como $f_k = \frac{n_k}{N}$ que nos dará un porcentaje.

Cuando un elemento vaya a ser insertado actualizamos $n_k = n_k + 1$ y cuando vaya a ser eliminado actualizamos $n_k = n_k - 1$

Se definirán 3 parámetros: Un soporte $s \in (0,1)$ tal que todas las frecuencias reales que sean más grandes que este soporte serán mostradas, el error $\varepsilon \in (0,1)$ tal que ninguna frecuencia real cuyo valor sea menos que $s - \varepsilon$ será mostrada y el parámetro de probabilidad ρ cercano a 1 donde las frecuencias estimadas serán más grandes que las reales como máximo con un error ε con una alta probabilidad ρ.

En este algoritmo se usará un tabla $hash$, $S[m][h]$, con h funciones $hash$. Cada una de estas funciones $hash$ a partir de un espacio de $[0, \dots, M-1]$ nos da un dígito entre $[0, \dots, m-1]$ de manera uniforme e independiente. Para este propósito se utilizarán las siguientes funciones $hash$:

$$\mathcal{H}_i(k) = \big((a_i \cdot k + b_i)\ mod\ P\big)\ mod\ m,\ 1 \le i \le h \qquad (1)$$

Donde a_i y b_i son valores aleatorios y P es un número primo grande. Un elemento e tiene asociado un conjunto de contadores: $\langle S[\mathcal{H}_1(k)][1], \dots, S[\mathcal{H}_h(k)][h]\rangle$. Estos contadores asociados aumentan o disminuyen al mismo tiempo cuando se actualizan por un elemento k.

Al finalizar con todas las transacciones de nuestro $Stream$ tendremos que nuestra frecuencia estimada para cada elemento k será: $min_{1 \le j \le h}(S[\mathcal{H}_1(k)][j])$.

Para poder ver mejor el funcionamiento de este algoritmo se propone un ejemplo.

Ejemplo 1.

Se asume un $Stream$ de datos con 16 elementos diferentes y 38 transacciones. La tabla $hash$ creada es de dimensiones $m = 5$ y $h = 4$. La ecuación (1) viene dada por $\mathcal{H}_1(k)$, $\mathcal{H}_2(k)$, $\mathcal{H}_3(k)$ y $\mathcal{H}_4(k)$ con los siguiente pares de (a_i, b_i): $(a_1, b_1) = (7,13), (a_2, b_2) = (22,6), (a_3, b_3) = (24,11)$ y $(a_4, b_4) = (14,27)$. El valor de P es 31.

Las transacciones vienen dadas por la siguiente tabla:

t_i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
e	2	1	6	3	9	-6	16	1	13	2	4	3	-16	1	5	3	10	5	2

t_i	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38
e	11	-11	2	1	3	8	2	1	-4	11	3	7	5	1	1	9	2	2	13

Tabla 1 - Datos de entrada para el algoritmo hCount

Donde t_i indica i-ésima transación. Si k es un elemento positivo se está hablando de una inserción en cambio sí es negativo se habla de una eliminación. La primera transacción es positiva así que aumentará nuestro contador de cada una de las cuatro celdas adjudicadas, en nuestra tabla, según el valor de las funciones $hash$: $\mathcal{H}_1(2) = 2\,, \mathcal{H}_2(2) = 4, \mathcal{H}_3(2) = 3, \mathcal{H}_4(2) = 4$ tal como se muestra en la siguiente tabla.

0	0	1	0	0
0	0	0	0	1
0	0	0	1	0
0	0	0	0	1

Tabla 2 - Visualización del algoritmo en el momento t_1

Si seguimos calculando nuevas transacciones en el momento de la sexta transacción tendremos:

1	0	1	1	1
1	0	0	2	1
1	1	0	1	1
1	0	1	0	2

Tabla 3 - Visualización del algoritmo en el momento t_6

Vemos que en la sexta transacción el elemento es negativo y por lo tanto tendremos que restar uno a cada celda adjudicada según las funciones $hash$ de este elemento.

En la última transacción obtenemos la tabla definitiva que queda de la siguiente forma:

8	3	11	6	2
7	0	1	14	8
2	5	5	10	8
8	2	6	2	12

Tabla 4 - Visualización del algoritmo en el momento t_{38}

En este caso la frecuencia estimada del elemento 6 es $min_{1\leq j\leq h}(S[\mathcal{H}_1(6)][j]) = 2$

Propiedades hCount

Proposición. Para este algoritmo serán usados $\frac{e}{\varepsilon} \cdot ln\left(-\frac{M}{ln\rho}\right)$ contadores para estimar cada uno de los elementos con un error máximo de εN con probabilidad ρ, con $m = \frac{e}{\varepsilon}$, y $h = ln\left(-\frac{M}{ln\rho}\right)$.

Demostración

Dado un elemento arbitrario k, su asociado conjunto contadores viene dado por: $\langle S[\mathcal{H}_1(k)][1], \dots, S[\mathcal{H}_h(k)][h]\rangle$, donde cada contador asociado puede contener ocurrencias de otros elementos. Definimos $\langle \epsilon_1, \dots, \epsilon_h\rangle$, como los errores de los h contadores para un elemento k así pues el conjunto de contadores asociados al elemento k se puede reescribir como $\langle n_k + \epsilon_1, \dots, n_k + \epsilon_h\rangle$. Suponiendo que las funciones $hash$ son uniformes significa que hay una media de $\left[\frac{M}{m}\right]$ contadores de los elementos en cada celda. Sea Y la variable aleatoria que denota el error, entonces:

$$E[Y] \leq \frac{N}{m}$$

Además por la desigualdad de Markov se sabe que:

$$Pr[|Y| - \lambda E[|Y|] > 0] \leq \frac{1}{\lambda}$$

donde λ es un número positivo. Porque Y es siempre un número mayor que 0 y $E[Y] \leq \frac{N}{m}$ se puede reescribir la desigualad como:

$$Pr\left[Y - \lambda\frac{N}{m} > 0\right] \leq \frac{1}{\lambda}$$

Esta fórmula muestra que para una variable aleatoria Y, si se intenta una vez, el caso de que el valor de Y sea más grande que $\lambda\frac{N}{m}$ sucede con una probabilidad no más de $\frac{1}{\lambda}$. Esta probabilidad la denotaremos por q. Si lo intentamos h veces y todos los valores son mayores que $\lambda\frac{N}{m}$, la probabilidad es no más que q^h.

En otras palabras, el suceso que el valor de Y como mínimo una vez sea más pequeño que $\lambda\frac{N}{m}$ pasa con probabilidad $1 - q^h$. Sea Y_{min} el valor mínimo después de h intentos entonces:

$$Pr\left[Y_{min} - \lambda\frac{N}{m} > 0\right] \leq \frac{1}{\lambda^h}$$

$$Pr\left[Y_{min} - \lambda\frac{N}{m} < 0\right] \geq 1 - \frac{1}{\lambda^h}$$

Denotamos como ρ la probabilidad del suceso que todos los M elementos cumplan las 2 fórmulas de arriba y $\varepsilon = \frac{\lambda}{m}$. Entonces,

$$\rho = \left(1 - \frac{1}{\lambda^h}\right)^M \approx \exp\left(\frac{-M}{\lambda^h}\right)$$

Vamos a minimizar el tamaño de la tabla $hash$. Sea $V = m \cdot h$ lo podemos reescribir como:

$$V = \frac{1}{\varepsilon} \cdot ln\left(-\frac{M}{ln\rho}\right) \cdot \frac{\lambda}{ln\lambda}$$

λ es un número positivo y $min_{\lambda>0}\ \frac{\lambda}{ln\lambda} = e$. Entonces concluimos que:

$$V = \frac{e}{\varepsilon} \cdot ln\left(-\frac{M}{ln\rho}\right), \quad h = ln\left(-\frac{M}{ln\rho}\right)\ y\ \ m = \frac{e}{\varepsilon}$$

Pese al uso de distintas variables en los algoritmos para definir los mismo se puede apreciar la similitud de estos tres algoritmos vistos basados en *sketch*. Todos tienen una parecida estructura de datos con algunas diferencias en este caso para el hCount se nos define como tienen que ser las funciones $hash$.

2.2 Algoritmos basados en contadores

2.2.1 Frequent

Se trata de un algoritmo bastante sencillo la idea es de mantener una lista de m elementos. La lista se inicializa a cero y para cada elemento del $Stream$ se actualiza de la siguiente manera.

Si el elemento está siendo monitorizado en la lista se incrementa su contador si en cambio no está siendo monitorizado tenemos dos casos:

1. Si algún contador de la lista es cero entonces se define este nuevo contador como el nuevo contador para este elemento y se incrementa.
2. Si todos los contadores en la lista monitorizada son mayores que cero entonces se resta uno a cada contador.

Este fue uno de los primeros algoritmos basados en contadores que se diseñó. Es por ello que la notable falta de exactitud a la hora de estimar el top- k hace de él un simple recuerdo, como se podrá ver más adelante en nuestra comparación. Es por ello que no es necesario examinarlo con más detalle.

2.2.2 Lossy counting

Este algoritmo divide nuestro stream en buckets (cubos) de un tamaño predefinido cada uno:

$$w \in \mathbb{N}, \text{ tal que } \varepsilon = \frac{1}{w}$$

Estos cubos son etiquetados con identificadores comenzando por 1. Se denota al identificador del cubo que se está procesando como:

$$b_{current} = \left\lceil \frac{N}{w} \right\rceil$$

Para un elemento e, se denota su verdadera frecuencia vista hasta ahora en el $Stream$ como f_e. Los valores ε y w son fijados por el usuario mientras que N, el número actual de elementos procesados, $b_{current}$ y f_e, la frecuencia estimada del elemento e son variables que cambian en el transcurso de nuestro $Stream$.

La estructura de datos, $\mathcal{D}$, es un conjunto de entradas de la forma (e, f, Δ), donde e es un elemento del $Stream$, f representa su frecuencia estimada y Δ es el máximo error de f.

Inicialmente, $\mathcal{D}$ está vacío. Siempre que un elemento e llega, se mira que este exista en $\mathcal{D}$ si es así se incrementa su frecuencia f en uno por el contrario si no existe en $\mathcal{D}$ se crea una nueva entrada de la forma $(e, 1, b_{current} - 1)$.

Siempre que $N \equiv 0 \; mod \; w$ se hará una purga en $\mathcal{D}$ borrando algunos de sus elementos. La regla de eliminación es simple: una entrada en $\mathcal{D}$, (e, f, Δ) es borrada si $f + \Delta \leq b_{current}$. En la ilustración mostrada a continuación se nos presenta un claro ejemplo del funcionamiento del algoritmo.

Propiedades Lossy counting

Lema 1. Siempre que ocurre una eliminación, $b_{current} \leq \varepsilon N$ ■

Lema 2. Siempre que una entrada (e, f, Δ) se elimina $f_e \leq b_{current}$.

Demostración

Se tiene que para $b_{current} = 1$, una entrada (e, f, Δ) es borrada si $f = 1$ que en esta caso es igual que su verdadera frecuencia f_e. Así $f_e \leq b_{current}$.

Ahora se va a considerar una entrada (e, f, Δ) eliminada por un $b_{current} > 1$. Sabemos que $f_e \leq \Delta + f$ por definición del algoritmo si a esto le añadimos la regla de cuando se elimina una entrada nos queda que $f_e \leq \Delta + f \leq b_{current}$. Entonces se puede concluir que siempre que una entrada (e, f, Δ) se elimina $f_e \leq b_{current}$. ■

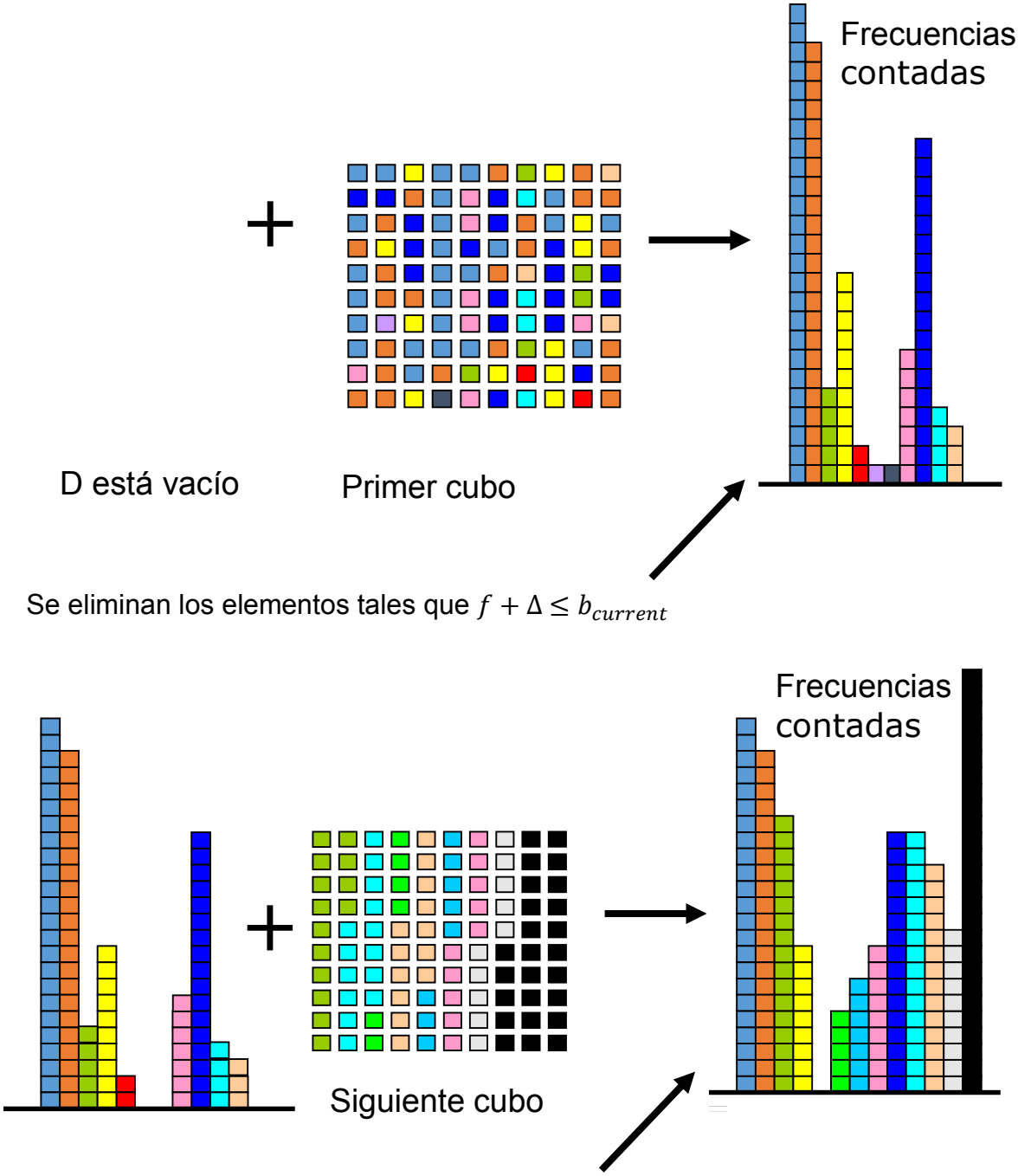

Ilustración 3 -Ejemplo del funcionamiento del Lossy Counting

Lema 3. Si e no aparece en $\mathcal{D}$, entonces $f_e \leq \varepsilon N$

Demostración

Si e no aparece en $\mathcal{D}$ es que en algún momento ha sido eliminado y por el lema 1 y 2 tenemos que $f_e \leq b_{current}$ y $b_{current} \leq \varepsilon N$ por lo tanto $f_e \leq \varepsilon N$. ■

Lema 4. Si $(e, f, \Delta) \in \mathcal{D}$, entonces $f \leq f_e \leq f + \varepsilon N$

Demostración

Si Δ= 0, entonces $f = f_e$. En cambio si $\Delta \neq 0$, e ha podido ser eliminado alguna vez en los anteriores Δ cubos. Del lema 2 se puede deducir que la frecuencia exacta, f_e, en su última eliminación es como máximo Δ. Sino ha sido eliminado ninguna vez tenemos por definición que $f_e \leq \Delta + f$. Como $\Delta \leq b_{current} - 1 \leq \varepsilon N$ se puede concluir que $f \leq f_e \leq f + \varepsilon N$. ■

Teorema 1. $Lossy\ Counting$ analiza los datos usando $\frac{1}{\varepsilon}\log(\varepsilon N)$ entradas.

Demostración

Sea $B = b_{current}$ el actual identificador del cubo. Para cada $i \in [1, B]$, sea d_i el número de entradas en $\mathcal{D}$ cuyo identificador del cubo es $B - i + 1$. El elemento correspondiente a dicha entrada debe ocurrir al menos i veces en los cubos desde $B - i + 1$ a B; de lo contrario tendría que ser eliminado. Dado que el tamaño de cada cubo es w se tiene las siguientes restricciones:

$$\sum_{i=1}^{j} i \cdot d_i \leq j \cdot w \ \ para\, j = 1,2, \dots, B \quad (1)$$

Se puede afirmar que:

$$\sum_{i=1}^{j} d_i \leq \sum_{i=1}^{j} \frac{w}{i} \ \ para\, j = 1,2, \dots, B \quad (2)$$

Para probar la inecuación (2) se usará inducción. Para $j = 1$ se obtiene de (1) directamente $d_i = w$. Se asume que (2) es cierto para $j = 1,2, \dots, p-1$ se va a probar que es cierto para $j = p$. Añadiendo (1) para $j = p$ a la suma de sumas (2) para $j = 1, \dots p-1$ se obtiene:

$$\sum_{i=1}^{p} i \cdot d_i + \sum_{i=1}^{1} d_i + \cdots + \sum_{i=1}^{p-1} d_i \leq p \cdot w + \sum_{i=1}^{1} \frac{w}{i} + \cdots + \sum_{i=1}^{p-1} \frac{w}{i}$$

Al reordenar los elementos se obtiene que:

$$p \cdot \sum_{i=1}^{p} d_i \leq p \cdot w + \sum_{i=1}^{p-1} \frac{(p-i)w}{i}$$

Que sustituyendo $p = j$ obtenemos (2).

Dado que $|\mathcal{D}| = \sum_{i=1}^{B} d_i$ de la inecuación (2) se deduce que $|\mathcal{D}| \leq \sum_{i=1}^{B} \frac{w}{i} \leq \frac{1}{\varepsilon}\log(B) = \frac{1}{\varepsilon}\log(\varepsilon N)$ ■

Este algoritmo a diferencia de los demás algoritmos basados en contadores no determina el número de elementos que va almacenar en su lista monitorizada sino que la va aumentando según vayan llegando los valores.

2.2.3 Space-saving

Para esta sección se propone el algoritmo Space-Saving y su asociada estructura de datos *Stream-Summary*. La idea subyacente es la de mantener información parcial de interés, es decir, solo m elementos serán monitorizados. Los contadores son actualizados de tal manera que se obtenga una estimación precisa de la frecuencia de los elementos más significativos, y estas estimadas frecuencias son guardadas en estructuras de datos ligeras (con un uso mínimo de memoria).

Cuando nos referimos al *Stream-Summary* hacemos alusión a su estrucutura de datos es decir que los elementos de la lista se guardan con cada uno de ellos almacenando 3 valores: $count_i$, e_i y ε_i, su error.

En una situación ideal cualquier elemento significante, E_i, con la i-ésima frecuencia más grande, F_i, debería estar en i-ésimo contador del *Stream-Summary*. Pero dado los errores estimando las frecuencias de los elementos, el orden de estos en el *Stream-Summary* puede que no refleje su clasificación exacta, es decir, el elemento que tiene mas frecuencia no tiene por qué estar en el primer contador. Por esta razón, se denota al contador en la i-ésima posición en la estructura de datos como $count_i$. El contador $count_i$ estima la frecuencia f_i. de algún elemento e_i.

El algoritmo es sencillo. Si un elemento monitorizado es observado, se incrementa el correspondiente contador. Si en cambio el elemento observado, e, no está monitorizado, se reemplaza e_m, el elemento que en ese momento tiene la menor frecuencia estimada, min, por e. El nuevo elemento, e, podría tener una frecuencia estimada entre $[1, min + 1]$. Se asigna $count_m$ el valor , $min + 1$, ja que el algoritmo esta designado para que el error que hagamos sea positivo, esto es para no dejarse nunca un elemento frecuente.

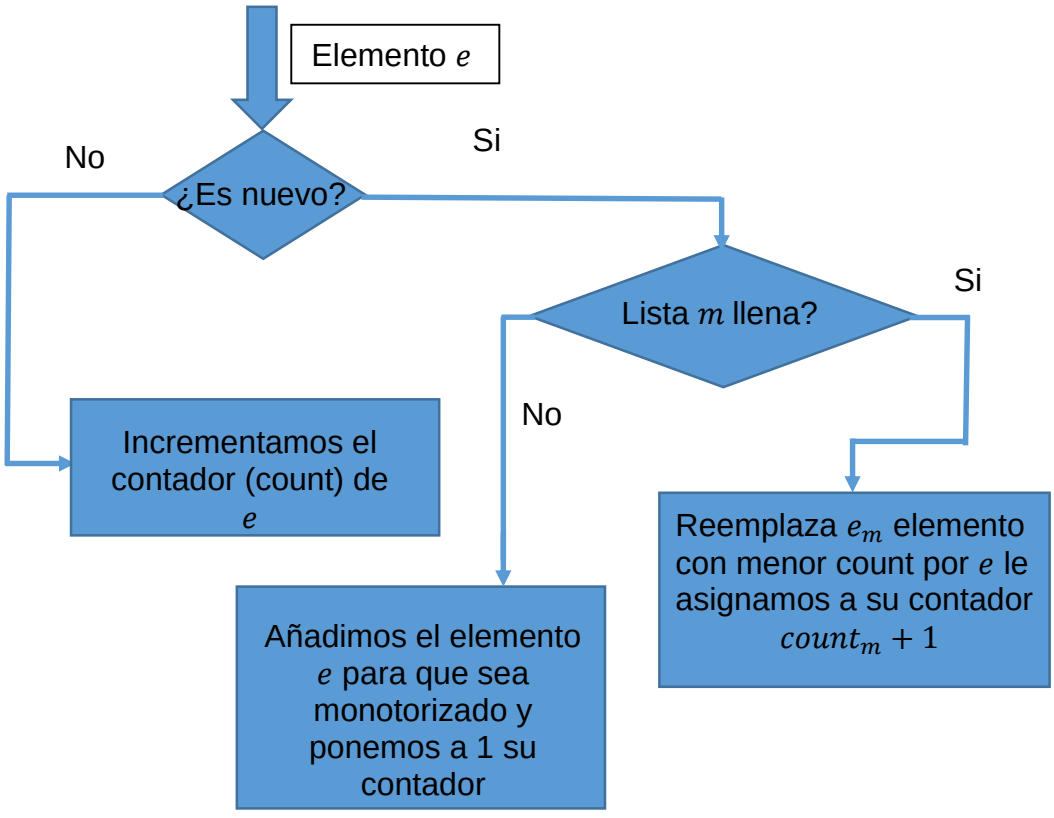

Ilustración 4 – Funcionamiento del Space Saving

Para cada elemento monitorizado e_i, hacemos un "seguimiento" de su máxima sobreestimación ,ε_i, como resultado de la inicialización de su contador cuando se inserta en la lista. Es decir, al comenzar a monitorizar e_i, se le da ε_i el valor del contador que se ha substituido.

En general, los mejores elementos entre los datos no asimétricos no son de gran importancia. Por lo tanto, nos concentramos en los conjuntos de datos sesgados, donde una minoría de los elementos, obtienen la mayoría de las frecuencias. La intuición básica es hacer uso de la propiedad sesgada de los datos mediante la asignación de contadores a elementos distintos, y seguir monitorizando los elementos de crecimiento rápido. Los elementos frecuentes se encontrarán en los contadores de los valores más grandes, y no serán distorsionados por los elementos irrelevantes poco frecuentes, y por lo tanto, nunca serán reemplazados de la lista de elementos monitorizados. Mientras tanto, los numerosos elementos poco frecuentes se esfuerzan para permanecer en los contadores más pequeños de la lista monitorizada, cuyos valores crecen más lentamente que las de los contadores más grandes.

Además, si el sesgo se mantiene, pero los elementos populares cambian con el tiempo, el algoritmo se adapta automáticamente. Los elementos más populares que están creciendo gradualmente serán empujados a la parte superior de la lista, ya que reciben aumentan su frecuencia. Si uno de los elementos anteriormente populares pierde su popularidad, dejarán de aumentar su frecuencia. Por lo tanto, su posición relativa decrecerá, en función de que otros contadores incrementen, y con el tiempo podría quedar fuera de la lista.

Incluso si los datos no están sesgados, los errores en los contadores son inversamente proporcionales al número de contadores, como se muestra más adelante. Mantener sólo un número moderado de contadores garantiza errores muy pequeños, ya que como se ha demostrado más tarde y se ilustra a través de experimentos, *Space Saving* es una de las técnicas más eficientes en términos de espacio. La razón es con más contadores es menos probable reemplazar elementos, y por lo tanto, menor es la sobre-estimación de errores en los valores de los contadores.

Para implementar este algoritmo, se necesita una estructura de datos que incremente los contadores sin alterar su orden, y que garantice un rápido tiempo de ejecución.

Propiedades Space Saving

Lema 1. La longitud, N, del stream es igual a la suma de todos los contadores en la estructura de datos Stream-Summary. Esto es, $N = \sum_{i \leq m}(count_i)$

Demostración

Cada elemento en S incrementa únicamente un contador de los m que hay. Esto es cierto incluso cuando ocurre una sustitución, es decir, cuando el elemento observado e no estaba previamente monitorizado, y este es reemplazado por algún elemento e_m. Esto es porque el $count_m$ se incrementa pase lo que pase. Entonces en cualquier momento del stream la suma de los contadores es igual al tamaño de la muestra analizada en el stream. ■

Tenemos que el valor de min es altamente dinámico ya que depende de la disposición de los elementos en nuestro stream S. Por ejemplo si $m = 2$, y $N = 4$. Entonces si $S = X, Z, Y, Y$ tenemos que $min = 1$ en cambio si $S = X, Y, Y, Z$ tenemos que $min = 2$.

El siguiente lema acota el valor de min asumiendo que todos los m contadores están ocupados.

Lema 2. El mínimo valor de nuestros contadores, min, no es más grande que $\left[\frac{N}{m}\right]$.

Demostración

El lema 1 se puede reescribir como:

$$min = \frac{N - \sum_{i \leq m}(count_i - min)}{m} \qquad (1)$$

Es directo ver que todos los términos del sumatorio son no negativos por lo tanto min alcanza su valor más alto cuando $\sum_{i \leq m}(count_i - min) = 0$ es decir que $min \leq \left[\frac{N}{m}\right]$ ■

El valor de min representa la cota superior de la sobreestimación para cada contador de nuestro $Stream - Summary$. En el siguiente Lema se establece esta relación.

Lema 3 Para cada elemento e_i en el *Stream-Summary*, tenemos que:

$0 \leq \varepsilon_i \leq min$, es decir, $f_i \leq (f_i + \varepsilon_i) = count_i \leq f_i + min$

Demostración

Del algoritmo podemos sacer que la sobreestimación del elemento e_i, ε_i, es no negativa, además siempre se le asigna el valor del mínimo contador, min, cuando este comienza a ser monitorizado. Como el valor de min crece de forma monótona en el tiempo hasta que alcanza el valor del min actual. Entonces para todo elemento monitorizado $\varepsilon_i \leq min$. ■

Por otra parte, todo elemento E_i, con frecuencia $F_i > min$ estará en el *Stream-Summary* como se muestra a continuación.

Teorema 1 Un elemento E_i con $F_i > min$, debe estar en el el *Stream-Summary*.

Demostración

Esta demostración es por contradicción. Supongamos que E_i no está en el *Stream-Summary*. Entonces, fue substituido en un momento anterior. Puesto que $F_i > min$, entonces F_i es más grande que el valor del mínimo contador en algún momento anterior, ya que este crece de forma monótona. Por lo tanto, del Lema 3, cuando E_i fue substituido por última vez, su frecuencia estimada era más grande que el valor del mínimo contador en ese instante. Esto contradice el algoritmo $Space - Saving$ que dice que el elemento substituido es siempre el que tiene menos valor en su contador. ■

Del Teorema 1 y el Lema 3, se puede obtener una regla de la sobreestimación de los contadores de los elementos. Para cada elemento E_i, el elemento con la i-ésima frecuencia más grande, $i \leq m$. La frecuencia de E_i, F_i, es no más de $count_i$, el contador que ocupa la i-ésima posición en el *Stream-Summary*. Es decir , $count_5$, el contador en la quinta posición

en el *Stream-Summary*, es una cota superior de F_5, incluso si la quinta posición en el *Stream-Summary* no está ocupada por E_5.

Teorema 2 Esté o no E_i ocupando la i-ésima posición en el *Stream-Summary*, $count_i$, el contador en la posición i, no es más pequeño que F_i, la frecuencia del i-ésimo elemento más grande, E_i.

Demostración

Hay cuatro posibilidades para la posición de E_i en el *Stream-Summary*.

- El elemento E_i no está en la lista de elementos monitorizados. Por el teorema 1, $F_i \leq min$. Entonces todo contador en el *Stream-Summary* no es más pequeño que F_i.

- El elemento E_i está en la posición j, con $j > i$. Por el Lema 3, la frecuencia estimada de E_i, $count_j$, no es más pequeña que F_i. Como j es más grande que i, entonces la frecuencia estimada de e_i no es más pequeña que $count_j$. Así que $count_i \geq F_i$.

- El elemento E_i está en la posición i. Por el Lema 3, $count_i \geq F_i$.

- El elemento E_i está en la posición j, con $j < i$. Al menos un elemento E_x con la x-ésima frecuencia más grande,$x < i$, está localizado en alguna posición y, $y \geq i$ en el *Stream-Summary*. Por el Lema 3 y $x < i$ la frecuencia estimada de E_x no es más pequeña que F_i. Como $y \geq i$, entonces $count_i \geq count_y$, que es igual a la frecuencia estimada de E_x. Y por lo tanto $count_i \geq F_i$.

Se tiene que en todos los casos $count_i \geq F_i$. ■

Todo elemento que tenga $(count_i - \varepsilon_i) \geq count_{k+1}$ debe de estar en el top-k.

En el siguiente teorema se va a calcular el valor de m que nos garantice un error máximo ϵ para el cálculo del top- k.

Teorema 3 Independientemente de la distribución de los datos, para encontrar una aproximación del top-k, $space - saving$ usa $m = \min\left(|A|, \frac{N}{\epsilon F_k}\right)$ contadores. Cualquier elemento con frecuencia mayor que $(1-\epsilon)F_k$ es garantizado para estar en la lista de elementos monitorizados. Donde $|A|$ es la cardinalidad de la muestra.

Demostración

Del teorema 1, cualquier elemento e_i tal que su frecuencia $f_i > min$ debe estar en el *Stream-Summary* Como min es una cota superior de ε_i por el Lema 2, entonces tenemos que $\varepsilon_i \leq min \leq \left\lceil \frac{N}{m} \right\rceil$. Ahora supongamos que $min = \epsilon F_k$ entonces $m \geq \frac{N}{\epsilon F_k}$ garantiza con un error ϵ encontrar los elementos tal que su frecuencia sea mayor que $(1-\epsilon)F_k$. ■

La sencillez de este algoritmo se verá ilustrada en su mínimo uso de memoria dando resultado muy buenos en datos sesgados. A continuación se va hacer un estudio comparativo de los algoritmos vistos hasta ahora en base de datos reales.

3 Elección del algoritmo óptimo

En estudios realizados por Nishad Manerikar y Themis Palpanas [12] nos proponen una comparación entre los diversos algoritmos, vistos hasta ahora, para el cálculo de los elementos más frecuentes.

Para cada experimento se tendrán en cuenta tres indicadores principales:

Precisión: La relación entre los elementos frecuentes encontrados por los algoritmos y los reales.

Uso de memoria: El total de memoria usada por los algoritmos para su estructura de datos interna.

Tiempo de actualización: El tiempo total que tarda el algoritmo en procesar todos los datos del $Stream$.

Se usarán tres base de datos reales distintas para ejecutar los experimentos con sus diversas características estadísticas detalladas en la tabla 5.

Retail: Contiene datos de la cesta de la compra al por menor de una tienda belga anónima.

Q148: Estos conjunto de datos derivan de los datos de KDD Copa 2000. Todos los valores que faltan (signos de interrogación) son reemplazados por 0.

Nasa: Para este conjunto de datos se ha utilizado los atributos "magnitud de campo" y "módulo de campo" de la nave espacial Voyager 2.

De ahora en adelante nos referiremos a los algoritmos como:

Frecuent → **Freq**

Lossy Counting → **LC**

Space-Saving → **SS**

Group Testing →**GT**

CountSketch→ **CS**

CountMinSketch→ **CMS**

hCount→ **hC**

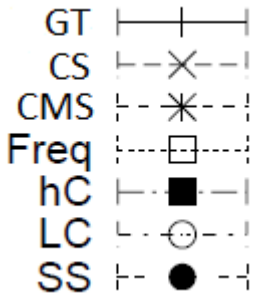

Tabla 5 - Leyenda para las siguientes gráficas

	Retail	Q148	Nasa
Nº Elementos	908576	234954	284170
Cardinalidad	16470	11824	2116
Min	0	0	0
Máx	16469	149464496	28474
Media	3264.7	3392.9	353.9
Mediana	1564	63	120
Desviación Std.	4093.2	309782.5	778.1
Sesgo	1.5	478.1	6.5

Tabla 6 - Estadísticos de las diferentes base de datos

1. Q148

En la siguiente ilustración se ve el rendimiento de los algoritmos bajo una restricción en el uso de memoria. Se puede observar como el Space-Saving, el hCount y el Lossy Counting obtienen una mayor precisión comparado con los otros algoritmos.

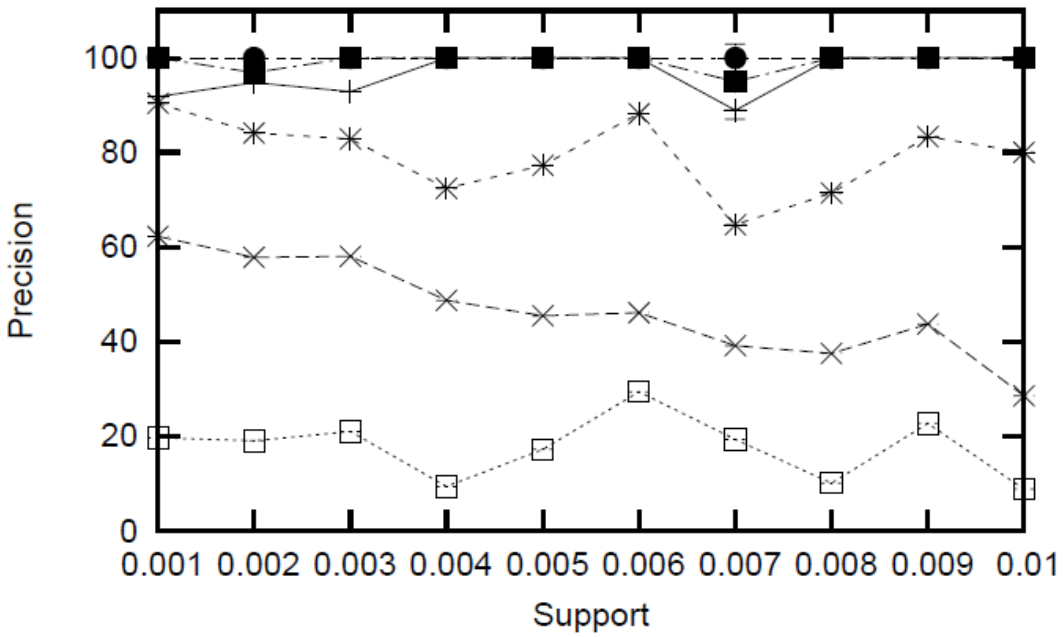

Ilustración 5 - Comparativa en la base de datos Q148

2. Retail

En la siguiente ilustración se observa una precisión muy baja de los algoritmos CountSketch, CountMinSketch y Frequent mientras tanto los Lossy Counting, hCount y Space-Saving obtiene como en la base de datos anterior una precisón muy alta.

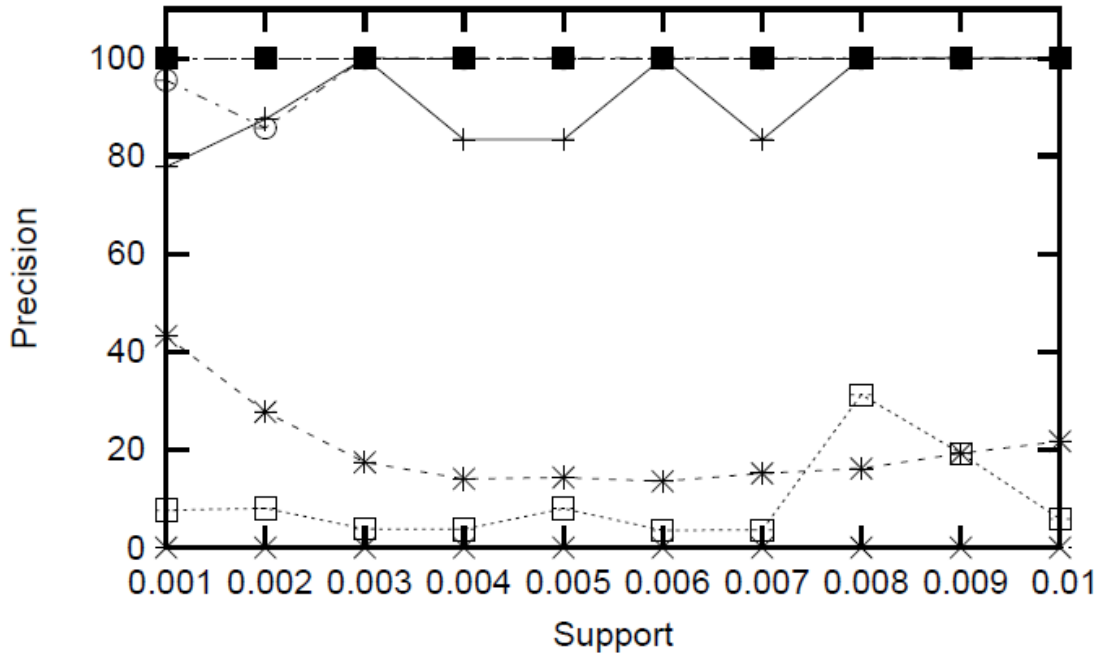

Ilustración 6 - Comparativa en la base de datos Retail

3. Nasa

En la siguiente ilustración los algoritmos hCount y Space-Saving siguen dando altos índices de precisión mientras que en esta caso el algoritmo Lossy Counting baja. Los demás algoritmos siguen dando malos resultados en comparación a los dos primeros.

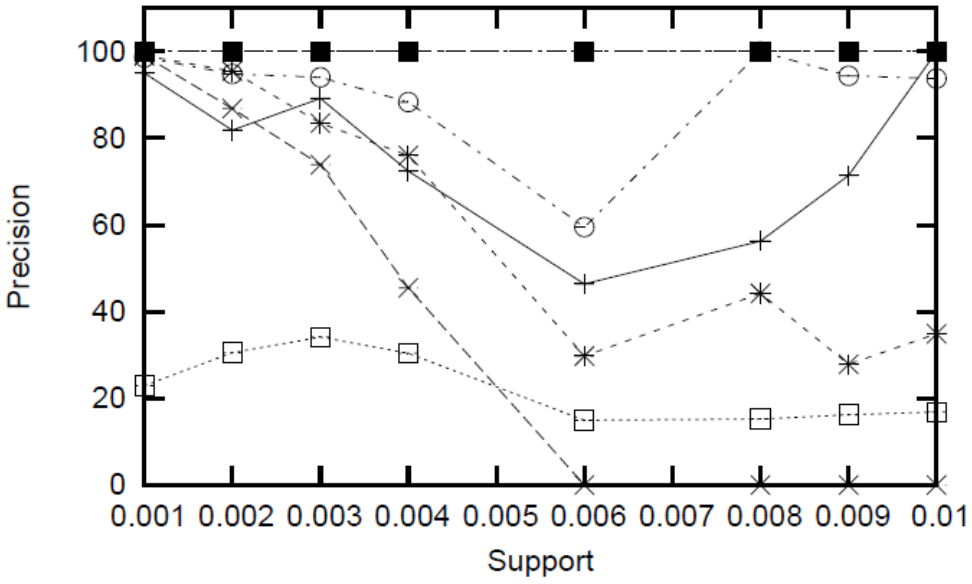

Ilustración 7 - Comparativa en la base de datos Nasa

Conclusiones

Se ha visto que que tanto Lossy Counting, hCount y Space-saving rinden bien en bases de datos reales. Es por ello que se ha querido estresarlos un poco más limitándoles la memoria. Para ello se ha comprobado la precisión para cada limitación en el uso de la memoria. Los algoritmos se han ejecutado 5 veces para comprobar su validez y los resultados vienen dados por la siguiente ilustración.

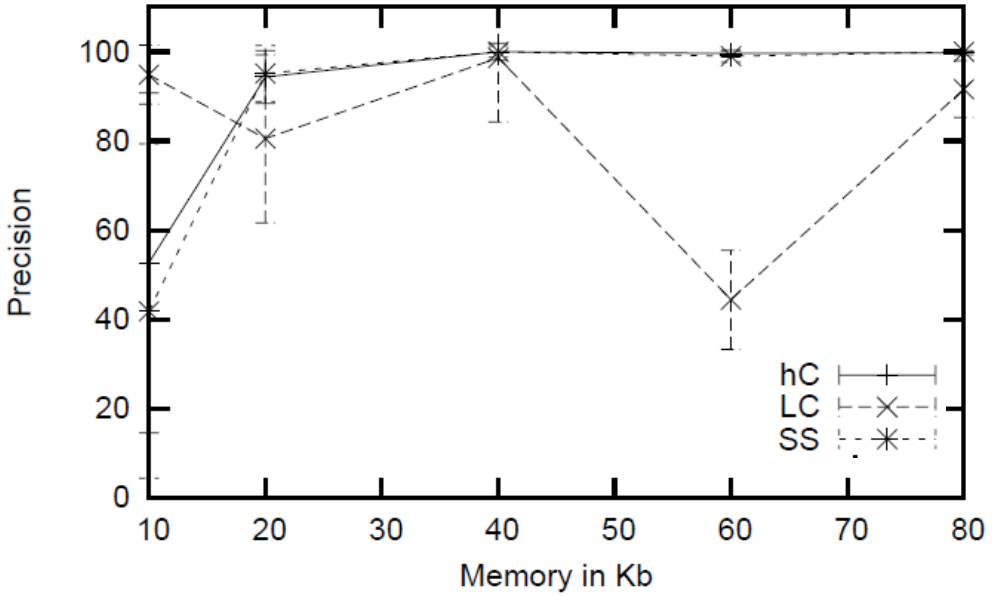

Ilustración 8 - Eficiencia respecto a un uso mínimo de memória

Se puede observar que un uso de memoria bastante bajo los algoritmos hCount y space-saving rinden mejor que el Lossy Counting, el cual baja su precisión de manera considerable cuando se le adjudica un uso limitado de la memoria.

Hemos mirado el rendimiento de los algoritmos a la hora de calcular los elementos más frecuentes a partir de un parámetro soporte,θ, de manera que se calculaban todos los elementos tales que su frecuencia era más grande que $\theta \cdot N$. Dado que la naturaleza del hCount es la de tener en cuenta todos los elementos y hacer una estimación de cada uno de ellos. El uso de este algoritmo para el cálculo del top-k, es decir, la búsqueda de los k elementos más frecuentes tendría un rendimiento menor que la del algoritmo space-saving dado que su principal uso es el de conseguir estimar los k elementos más frecuentes.

Es por ello que se ha seleccionado el space-saving como el algoritmo adecuado y en la siguiente sección se va comentar una mejora de este que nos mezclara características de los algoritmos basado en contadores y de los basados en sketch. Dando lugar a un nuevo algoritmo que obtendrá un rendimiento mejor que el propio space-saving.

4 Filtered Space Saving (FSS)

Este algoritmo es una extensión del *Space Saving*. FSS usa un contador *bitmap* para filtrar i minimizar las actualizaciones en los elementos monitorizados de la lista y también tener una mejor estimación del error asociado a cada uno de ellos. En vez de usar un único valor del error estimado, este usa un error estimado que depende de su contador hash. Esto permite una mejor estimación utilizando el máximo posible error para este valor de hash en particular, en lugar de un valor global. Aunque ha de mantenerse un *bitmap* adicional, se reduce el número de elementos de más en la lista para asegurar la calidad del top-k. La unión del bitmap con la lista de elementos minimiza el problema de la colisión de la mayoría de los casos de algoritmos basados en $Sketch$. El tamaño del contador del bitmap depende del número de k elementos que queremos para el top-k y no del número de elementos distintos del stream, que suele ser mucho más grande.

FSS utiliza un contador *bitmap* de h celdas, cada una con dos valores, α_i y c_i, para el error y el número de elementos monitorizados en la celda respectivamente. La función *hash* tiene que ser capaz de transformar los datos del $Stream\ S$ en un rango de enteros distribuidos uniformemente. El valor de la función *hash* $h(x)$ es utilizado para acceder al contador correspondiente. Inicialmente todos los valores de α_i y c_i se inicializan a 0. El segundo almacenamiento de elementos es una lista de los elementos monitorizados con tamaño m. La lista está vacía inicialmente. Cada elemento consiste de 3 partes: el propio elemento e_j, su valor de la frecuencia estimada f_j y su error asociado ε_i.

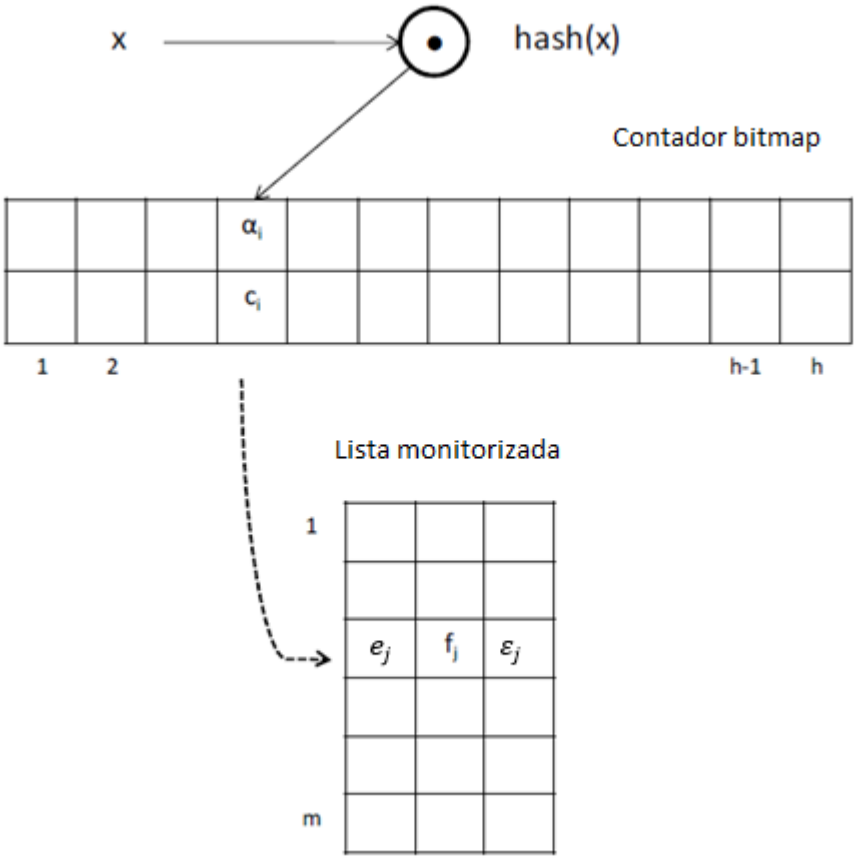

Ilustración 9 - Visión global del FSS

El valor α_i nos da una indicación del máximo error cometido de un elemento no monitorizado en la estimación de su frecuencia. Para que un elemento se incluya en la lista monitorizada, este tiene que tener un valor de α_i como el mínimo de las estimaciones de los elementos de la lista monitorizada, $\mu = \min\{f_j\}$. Mientras la lista tenga elementos libres, α_i y μ se ponen a cero.

El algoritmo es simple. Para cada elemento del $Stream$ se aplica la función $hash$ y se mira si hay algún elemento monitorizado con el mismo $hash$, es decir, $c_i > 0$, si es así comprobamos si este es el que tenemos monitorizado, si lo es le añadimos su nueva frecuencia. En caso contrario si el elemento no está en la lista de elementos monitorizados lo sustituiremos por el elemento que tenga la frecuencia $\mu = \min\{f_j\}$ si y solo si $\alpha_i + 1 \geq \mu$ y si no cumple esta condición no se añadirá a la lista y aumentará α_i. De hecho α_i nos dará una cota del máximo número de veces que un elemento con $hash(x) = i$ no ha sido incluido en la lista.

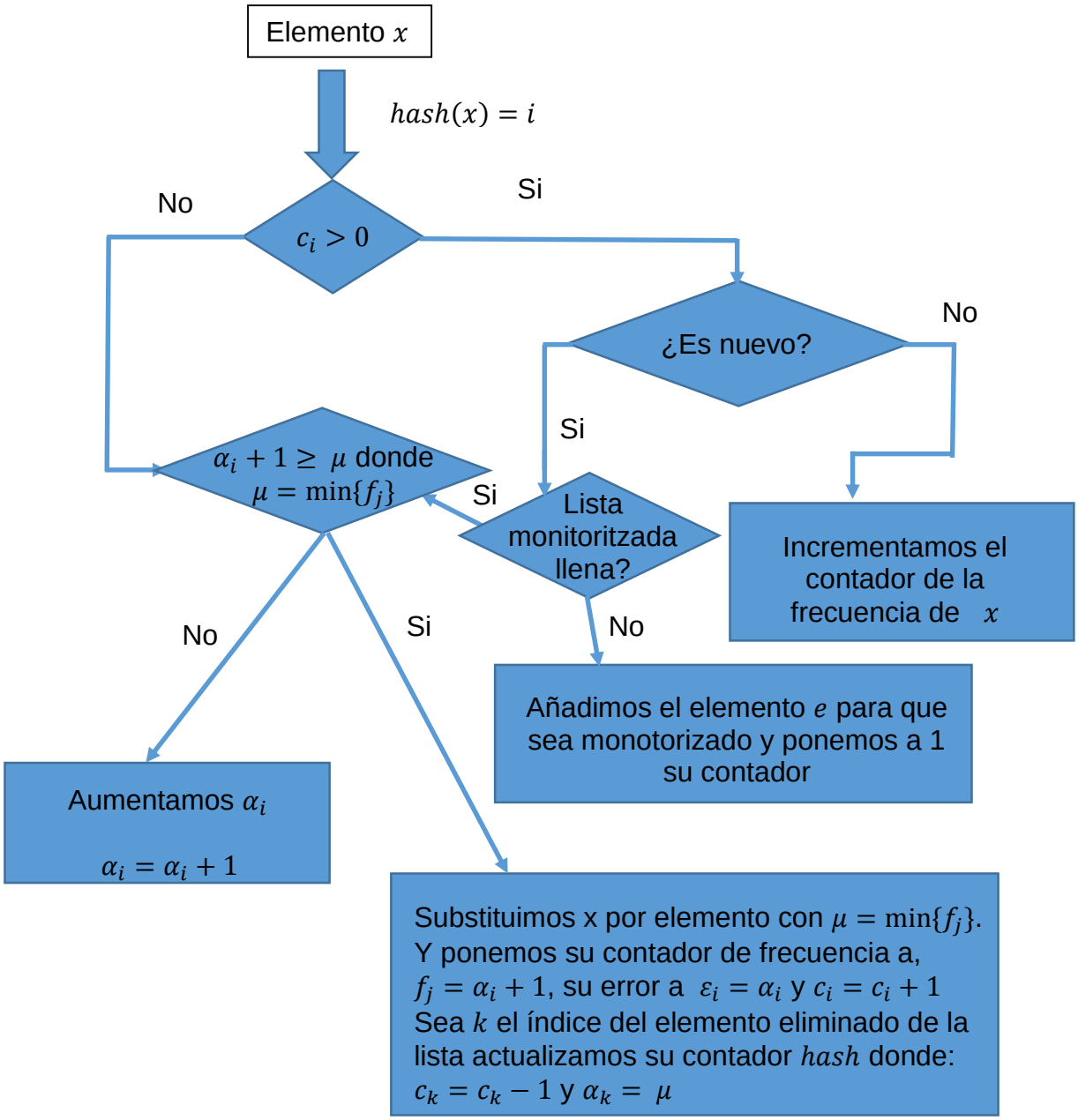

Ilustración 10 - Funcionamiento del FSS

A la hora de substituir un elemento se mira el que tenga una frecuencia estimada $\mu = \min\{f_j\}$. Si dos o más elementos cumplen está condición se mira el que tenga el error, más alto para ser substituido.

Cuando un elemento es substituido su correspondiente contador de la celda es actualizado, es decir, c_j decrece y $\alpha_j = \mu$. El nuevo elemento que se añadirá a la lista monitorizada sustituyendo al antiguo incrementará en su contador $hash\ c_i$ y pondrá $f_j = \alpha_i + 1$ y $\varepsilon_i = \alpha_i$.

Propiedades Filtered Space Saving

Lema 1 El valor de μ y el de cada α_i crece con el tiempo. Para cada t, t_0 después de que el algoritmo haya comenzado:

$$t \geq t_0 \Rightarrow \alpha_i(t) \geq \alpha_i(t_0)$$
$$t \geq t_0 \Rightarrow \mu(t) \geq \mu(t_0)$$

Demostración

Como α_i y μ son enteros:

$$\alpha_i + 1 > \mu \iff \alpha_i \geq \mu$$

Por construcción cuando $\alpha_i \geq \mu$ hay una sustitución en la lista monitorizada y un nuevo elemento será insertado donde $f_j = \alpha_i + 1$. En este instante μ será actualizado a $\min\{f_j\}$, entonces:

$$\alpha_i(t) \leq \max\{\alpha_i(t)\} \leq \mu(t) = \min\{f_j(t)\} \leq f_j(t)$$

Considera t_1 como el momento antes de la sustitución y t_2 el momento después de la sustitución, $t_2 > t_1$. Si el elemento en la lista a sustituir fuera el único que tiene $f_j(t_1) = \mu(t_1)$ entonces:

$$\mu(t_2) = \alpha_i(t_1) + 1 > \mu(t_1)$$

En cambio si mas de un elemento en la lista tiene $f_j(t_1) = \mu(t_1)$ entonces:

$$\mu(t_2) = \mu(t_1)$$

En cualquier caso:

$$\mu(t_2) \geq \mu(t_1)$$

En el proceso de sustitución, la celda del bitmap correspondiente al elemento reemplazado será actualizada con el valor de este elemento:

$$\alpha_i(t_2) = f_k(t_1) = \mu(t_1) \geq \max\{\alpha_i(t_1)\} \geq \alpha_i(t_1)$$

Cada vez que entra un nuevo elemento y este no requiere ser reemplazado por uno de la lista, un contador de entre los m contadores será incrementado para f_j o un contador de los h contadores del bitmap para α_i será incrementado. Por lo tanto en todo momento:

$$t \geq t_0 \Rightarrow \alpha_i(t) \geq \alpha_i(t_0)$$
$$t \geq t_0 \Rightarrow \mu(t) \geq \mu(t_0)$$ ■

Lema 2 El número total de elementos N en el $Stream\ S$ es mayor o igual que la suma de las frecuencias estimadas en la lista monitorizada.

$$N \geq \sum_j f_j$$

Demostración

Cada entrada de S incrementa como mucho un contador de los m contadores para f_j cuando el elemento está siendo monitorizado. Esto es cierto incluso cuando un nuevo elemento sustituye uno antiguo ya que este nuevo elemento obtiene una f_i igual al antiguo valor más el valor del nuevo elemento observado. ■

Teorema 1 Entre todos los contadores, el contador con valor mínimo, μ, no es más grande que $\frac{N}{m}$.

Demostración

Por el Lema 2 tenemos que:

$$N \geq \sum_j f_j = \sum_j (f_j - \mu) + \sum_j \mu = \sum_j (f_j - \mu) + m\mu$$

Y como $f_j - \mu \geq 0$, entonces:

$$\mu \leq \left[\frac{N - \sum_j (f_j - \mu)}{m}\right] \leq \frac{N}{m}$$ ■

Teorema 2 Un elemento x_i con $F_i > \mu$, donde F_i es la frecuencia real de x_i en el $Stream\ S$. Tiene que estar en la lista monitorizada.

Demostración

La prueba es por contradicción. Supongamos que x_i no está en la lista monitorizada. Entonces todas las entradas de x_i deberían estar contadas en algún α_i y esto quiere decir que α_i no tendría que ser más pequeño que F_i. Por lo tanto tenemos $\alpha_i \geq F_i > \mu$ que no puede ser ya que $\alpha_i \leq \mu$. ■

Lema 3 El valor $f_i - \varepsilon_i$, nos da el número de veces que un elemento ha sido observado desde que fue incluido a la lista monitorizada. Sea F_i su frecuencia real:

$$f_i - \varepsilon_i \leq F_i \leq f_i$$

Demostración

La primera desigualdad es obvia ya que $f_i - \varepsilon_i$ por definición es el número de veces que el elemento ha sido observado desde que fue incluido en la lista así que como máximo será igual a la frecuencia verdadera. Para la segunda desigualdad vamos asumir lo opuesto, es decir, $F_i > f_i$. En este caso $F_i > f_i = f_i - \varepsilon_i + \varepsilon_i = f_i - \varepsilon_i + \mu(t_0)$, donde t_0 es el momento inmediatamente antes de que x_i vaya a ser monitorizado en la lista. Pero esto significa que en ese instante $F_i > \mu(t_0)$, cosa que contradice el Teorema 2 ya que debería estar en la lista monitorizada. Entonces $F_i \leq f_i$. ■

Teorema 3 Esté o no x_i ocupando la i-ésima posición en la lista monitorizada, f_i, el contador en la posición i, no es más pequeño que F_i, la frecuencia del i-ésimo elemento más grande, x_i.

Demostración

La demostración sigue el mismo argumento que el Teorema 2 del $Space - saving$ ■

Teorema 4 Para cualquier celda que tenga elementos monitorizados:

$$\mu \leq \max\{N_i\} = N_{max}$$
$$E(\mu) \leq E(N_{max})$$

Donde N_i son las frecuencias de los elementos con $hash$ igual a i del $Stream\ S$.

Demostración

El hecho que las funciones $hash$ den valores en un rango de manera uniforme, da al algoritmo un compartimiento probabilístico. Sea A los elementos de la lista monitorizada y sea A_i el conjunto de elementos de A tal que $hash(e_j) = i$ y t_{j0} el momento que el elemento e_j fue insertado en la lista:

$$hash(e_j) = i \Rightarrow \ \varepsilon_j = \alpha_i(t_{j0}) \leq \alpha_i(t). \quad t > t_{j0}$$

El número total de frecuencias del $Stream$, N puede ser reescrito en función de sumas de N_i:

$$N = \sum_i N_i$$

De hecho N_i es más grande que α_i más el número frecuencias de los elementos en la lista monitorizada desde que se insertaron en la lista.

$$N_i \geq \alpha_i + \sum_{j\ |e_j \in A_i} (f_j - \varepsilon_j) \geq\ \alpha_i + \sum_{j\ |e_j \in A_i} (f_j - \alpha_i) \geq \alpha_i - c_i\alpha_i + \sum_{j\ |e_j \in A_i} f_j$$

$$N_i \geq \alpha_i - c_i\alpha_i(t_0) + \sum_{j\ |e_j \in A_i} f_j \geq \alpha_i - c_i\alpha_i +\ c_i\mu$$

$$c_i\mu \leq N_i + \alpha_i \cdot (c_i - 1) \leq c_i N_i$$

Para todo $c_i > 0$:

$$\mu \leq \max\{N_i\} = N_{max}$$

Y como consecuencia inmediata obtenemos que aplicando la esperanza a estos valores obtenemos:

$$E(\mu) \leq E(N_{max}).$$ ■

El *Filtered Space Saving* es una mezcla de algoritmos basados en contadores y los basados en *Sketch* ya que tiene la base del *Space Saving* con el añadido de un contador bitmap, que utiliza funciones hash para obtener información de las ocurrencias de los elementos en esa celda que no han sido contados por la lista monitorizada.

De manera intuitiva se puede observar que este algoritmo presenta mejores resultados que el *Space-Saving.* En la siguiente sección propondremos un análisis comparativo de los dos algoritmos demostrando la eficacia del *Filtered Space Saving* respecto a su antecesor.

5 Comparación FSS y Space Saving

Para el siguiente Teorema supondremos que el tamaño de la lista que monitorizará los elementos en los dos algoritmos es del mismo tamaño.

Teorema de comparación. Cualquier elemento de la lista monitorizada en Space Saving tendrá una frecuencia estimada, $count_j$, mayor o igual que la frecuencia estimada, f_i, de este mismo elemento en *FSS.* Además sea F_t la frecuencia real de este elemento tenemos que:

$$F_t \leq f_i \leq count_j$$

Demostración

Nos basta con demostrar que $f_i \leq count_j$ ya que por el Lema 3 del *FSS* tenemos que $F_t \leq f_i$.

Para demostrar la segunda desigualdad vamos suponer diversos casos:

- El elemento nunca ha sido expulsado de la lista monitorizada de *Space Saving* y del *FSS* entonces tenemos que $f_i = count_i$.
- Un elemento ha sido expulsado alguna vez de la lista monitorizada de *Space Saving* Sea t_0 el momento antes en que un elemento es expulsado de la lista de *Space Saving* y se va a presuponer que es el único elemento que cumple que su contador $count_i = \min\{count_k\}$ entonces el nuevo elemento que va incorporarse entra con un contador $count_j = count_i + 1$ y al ser el único que había entonces el valor $\min\{count_k\} = count_j$ si acto seguido vuelve a aparecer el elemento que fue sustituido entraría con un error. Todo esto en el *FSS* se optimiza ya que existe la posibilidad de que este elemento no cumpla la condición para ser sustituido (se considera irrelevante) y por la tanto la lista monitorizada se deja intacta. Es decir en cualquier momento t:

$$\mu = \min\{f_j\} \leq \min\{count_k\}$$

Por lo tanto cuando un elemento entra en cualquiera de las dos listas monitorizadas tendrá un error mayor en *Space Saving* y como consecuencia una mayor sobreestimación de su frecuencia. Así pues $F_t \leq f_i \leq count_j$. ■

Analisis en distribuciones Zipfian

Vamos a trabajar con datos creados a partir de una distribución. Para este propósito se va utilizar la distribución Zipfian con un parámetro de $\alpha \geq 1$ de esta manera se generan datos sesgados que son los que se necesitan para el estudio de estos algoritmos. Un conjunto de datos Zipfian, con parámetro α, tiene la frecuencia, F_i, de un elemento, E_i, siendo este el i-ésimo elemento mas frecuente, tal que:

$$F_i = \frac{N}{i^{\alpha} \cdot \mathfrak{H}(\alpha)}$$

donde $\mathfrak{H}(\alpha) = \sum_{i=1}^{|A|} \frac{1}{i^{\alpha}}$ converge a una constante pequeña inversamente proporcional a $\alpha > 1$ y $|A|$ es la cardinalidad del conjunto de datos.

Para $\alpha = 1$ se tiene $\mathfrak{H}(\alpha) \approx \ln(1.78|A|)$.

Vamos a ver cómo se comporta el *Space Saving* en este tipo datos ya que luego lo podremos aplicar al *FSS*.

Entre todas las posibles permutaciones de A, el posible valor máximo de min se denotará como min_{max} y entre todos los elementos que tengan un valor más grande que min_{max} el elemento con menos frecuencia de estos se denotará como E_r.

Lema 1 Una elemento E_i, tiene $F_i > min_{max}$, está garantizado para estar en la lista garantizada si y solo si $i \leq r$.

Teorema 1 Asumiendo que se tiene un conjunto de datos Zipfian con parámetro $\alpha > 1$, para calcular el top-k exacto, el algoritmo *Space Saving* usa tantos contadores como $\min\left(|A|, O\left(\frac{k}{\alpha}\right)^{\frac{1}{\alpha}} \cdot k\right)$ y cuando $\alpha = 1$, $\min\left(|A|, O(k^2 \cdot \ln(|A|))\right)$.

Demostración

Por el Lema anterior y por el Lema 2 y Lema 3 del *Space Saving* se puede deducir del posible valor máximo de min, min_{max} y del elemento con menos frecuencia, E_r, que $min_{max} \leq \frac{N - \sum_{i \leq r}(F_i - min_{max})}{m}$. Del Lema anterior se obtiene $F_{r+1} \leq min_{max}$ y por lo tanto tenemos que $F_{r+1} \leq \frac{N - \sum_{i \leq r}(F_i - F_{r+1})}{m} = \frac{N - (\sum_{i \leq r} F_i) + rF_{r+1}}{m}$ y pasando F_{r+1} a la otra banda: $F_{r+1} \leq \frac{N - \sum_{i \leq r} F_i}{m - r}$.Reescribiendo las frecuencias para una base de datos con distribución Zipfian obtenemos:

$$m - r \leq (r+1)^{\alpha} \sum_{i=r+1}^{|A|} \frac{1}{i^{\alpha}}$$

Esta implicación puede ser aproximada por:

$$m - r < (r+1) * \sum_{i=1}^{|A|/(r+1)} \frac{1}{i^{\alpha}}$$

$$\frac{1}{r} < \frac{\mathfrak{H}(\alpha) + 1}{m - \mathfrak{H}(\alpha)}$$

Para garantizar que los primeros k elementos de la lista monitorizada pertenecen al top-k se debe cumplir que $F_{k+1} - F_k > min_{max}$ ya que por el Lema 3 de *Space Saving* $0 \leq \varepsilon_i \leq min_{max}$, para todos elementos monitorizados en la lista. Así pues tenemos que $min_{max} < \frac{N}{\mathfrak{H}(\alpha)} * \frac{(k+1)^{\alpha} - k^{\alpha}}{(k+1)^{\alpha} k}$ y por lo tanto podemos acotar $F_r < \frac{N}{\mathfrak{H}(\alpha)} * \frac{\alpha}{(k+1)^{\alpha} k}$ que combinándolo con las inecuaciones donde aparecen m y r se tiene que cumplir que:

$$\frac{N}{\mathfrak{H}(\alpha)} * \left(\frac{\mathfrak{H}(\alpha)+1}{m-\mathfrak{H}(\alpha)}\right)^{\alpha} < \frac{N}{\mathfrak{H}(\alpha)} * \frac{\alpha}{(k+1)^{\alpha}k}$$

Después de alguna manipulación se puede obtener una cota de m tal que:

$$m > \left[(\mathfrak{H}(\alpha)+1)\left(\frac{k}{\alpha}\right)^{\frac{1}{\alpha}}(k+1)\right] + \mathfrak{H}(\alpha)$$ ■

Análisis experimental

Para realizar los test se ha usado un ordenador Hp Pavilion con procesador i7 y Windows 7 y se va a crear una base de datos a partir de una distribución Zipfian con parámetro $\alpha > 1$ para garantizar que los datos sean sesgados. A continuación se van a mostrar una serie de tablas donde se darán los detalles de los resultados obtenidos se va a definir como N al número de elementos totales de dicha base.

	FSS		*SS*	
PRUEBA	1	2	1	2
TOP-K	20	20	20	20
M	30	30	60	60
H	120	120		
SUSTITUCIONES	17793	18147	134119	133488
TIEMPO DE EJECUCIÓN	71 ms	95 ms	170 ms	143 ms
PRECISIÓN	100%	100%	85%	90%

Tabla 7 - Prueba 1 y 2 con $\alpha = 1.01$, 600.000 y |N|=1000

Vemos que claramente el *Filtered Space Saving* Tiene una mejor precisión que su adversario pero no solo eso sino que además el elemento 20 de nuestro top-*k* coincide con el real y su estimación de la frecuencia solo tiene un 2% de error.

	FSS		*SS*	
PRUEBA	1	2	1	2
TOP-K	20	10	20	10
M	30	15	60	30
H	120	60		
SUSTITUCIONES	8162	23421	101604	133961
TIEMPO DE EJECUCIÓN	72 ms	55 ms	148 ms	122 ms
PRECISIÓN	100%	100%	90%	95%

Tabla 8 - Prueba 1 y 2 con $\alpha = 1.1$, 600.000 y |N|=700

Los algoritmos dan buenas buenos resultados del top-*k* pero el *FSS* a parte de darlos con el 100% de precisión también tiene un tiempo de ejecución menor y esto es debido al filtro que tiene añadido que provoca que no se hagan tantas sustituciones y por lo tanto no se tenga que reordenar la lista.

	FSS		*SS*	
PRUEBA	1	2	1	2
TOP-K	20	10	20	10
M	30	15	100	30
H	150	60		
SUSTITUCIONES	23715	62245	295539	474410
TIEMPO DE EJECUCIÓN	79 ms	71 ms	250 ms	229 ms
PRECISIÓN	100%	60%	90%	40%

Tabla 9 - Prueba usando como base datos todas las palabras de la biblia

Esta base de datos corresponde a todas las palabras biblia y los resultados obtenidos, es decir su top-*k*, son las palabras más frecuentes. Aunque se puede volver a ver que el *FSS* tiene un mejor rendimiento en este caso los elementos reportados no están en el orden en el que deberían estar. Además el elemento en la posición 20 tiene una frecuencia estimada con un error del 9.1% respecto al a la frecuencia real de su so dicho elemento.

En la siguiente sección se va a plantear una extensión del *Filtered Space Saving* que como se verá reducirá el error cometido al estimar las frecuencias dando como resultado estimaciones muy precisas.

6 Multihash FSS

Esta extensión del *FSS* ha sido fruto de la investigación de los otros algoritmos vistos en este trabajo ya que tiene un cierto parecido por ejemplo al *CountMinSketch*.

Para este algoritmo las entradas serán del tipo (e, c) donde c nos indicará el peso de este elemento y e es el elemento en cuestión.

Vamos a suponer una familia de funciones *hash* independientes y uniformes definidas de la siguiente manera:

$$h_1, \dots, h_d : \{e_1, \dots, e_n\} \rightarrow \{1, \dots, w\}$$

donde $\{e_1, \dots, e_n\}$ son elementos del *Stream*.

Multihash FSS usa una tabla *hash* de dimensiones $w \cdot d$ de la misma manera que *FSS,* es decir, para filtrar i minimizar las actualizaciones en los elementos monitorizados de la lista y tener una mejor estimación del error asociado a cada uno de ellos pero está vez se hará servir varias funciones *hash* para este propósito.

El algoritmo utiliza d contadores bitmap de w celdas cada uno, cada una de estas celdas guarda un valor $count[j, h_j(e_i)]$ que corresponde al α_i del *FSS*. El valor de las funciones *hash* servirán para acceder al contador correspondiente. Como en *FSS* incialmente todos los valores $count[j, h_j(e_i)]$ se inicializan a cero con $j \in \{1, \dots, d\}$.

Por otra parte se tiene una lista, de tamaño m, donde se almacenan los elementos monitorizados. Esta lista guardará 3 valores: el propio elemento v_j, su valor de la frecuencia estimada f_j y su asociado error ε_i.

El valor $min_j count[j, h_j(e_i)]$ nos da una indicación del máximo error cometido por un elemento e_i no monitorizado en la estimación de su frecuencia. Para que un elemento se incluya en la lista monitorizada, este tiene que tener un valor de $min_j count[j, h_j(e_i)] + c_i$ como el mínimo de las estimaciones de los elementos de la lista monitorizada, $\mu = \min\{f_j\}$.Mientras la lista tenga elementos libres, $count[j, k]$ y μ se ponen a cero.

El Funcionamiento del algoritmo es de la siguiente manera. Para cada elemento del $Stream$ se mira si está siendo monitorizado, si es así se aumenta la frecuencia, $f_j = f_j + c_i$, . En caso contrario si el elemento no está en la lista de elementos monitorizados lo sustituiremos por el elemento que tenga la frecuencia $\mu = \min\{f_j\}$ si y solo si $min_j count[j, h_j(e_i)] + c_i \geq \mu$. Si no cumple esta condición no se añadirá a la lista y actualizaremos nuestra tabla de la siguiente manera:

Sea k el índice tal que $min_j count[j, h_j(e_i)] = count[k, h_k(e_i)]$ entonces incrementamos este contador $count[k, h_k(e_i)] = count[k, h_k(e_i)] + c_i$ luego para todo $j \neq k$:

Si $count[j, h_j(e_i)] < count[k, h_k(e_i)]$ entonces $count[j, h_j(e_i)] = count[k, h_k(e_i)]$.

Es decir solo incrementaremos los contadores tales que el posible error máximo ocurrido sea menor que $count[k, h_k(e_i)]$ ya que este es el que marcará el posible error máximo ocurrido en el elemento e_i.

A la hora de substituir un elemento se mira el que tenga una frecuencia estimada $\mu = \min\{f_j\}$ si dos o más elementos cumplen está condición se mira el que tenga el error, más alto para ser substituido.

Cuando un elemento es substituido, e_p por ejemplo, su correspondientes contadores son actualizados,$count[j, h_j(e_p)] = \mu$ para todo $j \in \{1, .., d\}$. El nuevo elemento que se añadirá a la lista monitorizada sustituyendo al antiguo incrementará $f_j = min_j count[j, h_j(e_i)] + c_i$ y $\varepsilon_i = min_j count[j, h_j(e_i)]$.

Las propiedades de este algoritmo son las mismas que las del *FSS.*

Ahora vamos a ver un Teorema que nos relaciona las estimaciones de las frecuencias de estos dos algoritmos

Teorema de comparación Cualquier elemento de la lista monitorizada en FSS tendrá una frecuencia estimada, f_j', mayor o igual que la frecuencia estimada, f_i, de este mismo elemento en *Multihash FSS.* Además sea F_t la frecuencia real de este elemento tenemos que:

$$F_t \leq f_i \leq f_j'$$

Demostración

Como en el Teorema de comparación de la sección anterior nos basta con demostrar que $f_i \leq f_j'$ ya que por el Lema 3 del *FSS* que comparte *Multihash FSS* tenemos que $F_t \leq f_i$.

Se va usar el hash $h_1(e_i)$ del *Multihash FSS* como la función *hash* del *FSS.*

Para demostrar la segunda desigualdad vamos suponer diversos casos:

- El elemento nunca ha sido expulsado de la lista monitorizada de *FSS* y del *FSS* entonces tenemos que $f_i = f_j'$.
- El elemento ha sido expulsado en algún momento de la lista monitorizada. Sea t_0 el momento antes en que el primer elemento es expulsado de la lista de *FSS* entonces un nuevo elemento e_p va incorporarse y esto significa que $\alpha_{h_1(e_p)} + c_i \geq \mu$ y se actualiza su $f_i = \alpha_{h_1(e_p)} + c_i$ y actualizando el $\alpha_j = \mu$ respectivo al elemento expulado pudiendo así aumentar el error de α_j, μ o f_i. En cambio en el *Multihash FSS* puede pasar dos cosas una que $min_j count[j, h_j(e_i)] + c_i \geq \mu$ que se efectuaría de manera similar a lo anterior o que $min_j count[j, h_j(e_i)] + c_i < \mu$ entonces actualizaríamos de la siguiente manera:
  Sea k el índice tal que $min_j count[j, h_j(e_i)] = count[k, h_k(e_i)]$ entonces incrementamos este contador $count[k, h_k(e_i)] = count[k, h_k(e_i)] + c_i$ luego para todo $j \neq k$:

  Si $count[j, h_j(e_i)] < count[k, h_k(e_i)]$ entonces $count[j, h_j(e_i)] = count[k, h_k(e_i)]$.

A rangos generales esto implica que la lista monitorizada no se actualizado mientras que el *FSS* si lo ha hecho.

Por lo tanto cuando un elemento entra en cualquiera de las dos listas monitorizadas en *Multihash FSS* tendrá un error no mayor que en *FSS* y como consecuencia $F_t \leq f_i \leq f_j'$.

A continuación se van a realizar un análisis experimental comparando estos dos algoritmos está claro que si el *FSS* funciona bien con datos sesgados el algoritmo propuesto también lo hará es por eso que vamos a intentar estresarlos para llevarlos al límite y ver allí sus debilidades.

Análisis experimental

Para la implementación del *FSS* se ha usado la misma que en la anterior sección y para el *Multihash FSS* se han usado 3 funciones *hash* de las cuales 1 es la misma que la del *FSS.*

	FSS		*MULTIHASH FSS*	
PRUEBA	1	2	1	2
TOP-K	20	10	20	10
M	30	15	30	15
H	150	75	150	75
SUSTITUCIONES	23715	53475	602	366
TIEMPO DE EJECUCIÓN	81 ms	62 ms	101 ms	86 ms
PRECISIÓN	100%	70%	100%	100%

Tabla 10 - Resultados estadísticos de los dos algoritmos variando su h y m

En la prueba 1 pese que parece que el *FSS* sea mejor ya que tiene menor tiempo de ejecución la verdad es que los elementos del top-*k* devueltos aparecen en el orden incorrecto al real. Como se muestra en la siguiente tabla la precisión del *Multihash FSS* es de envidiar ya que calcula todos los elementos del top-*k* ordenados y con una precisión muy exacta.

Exacto	Multihash FSS	FSS
the / 64203	the / 64203	the / 64203
and / 51764	and / 51764	and / 51764
of / 34789	of / 34789	of / 34789
to / 13660	to / 13660	to / 13660
that / 12927	that / 12927	that / 12928
in / 12725	in / 12725	in / 12725
he / 10421	he / 10421	he / 10421
shall / 9840	shall / 9840	shall / 9925
unto / 8997	unto / 8997	i / 9037
for / 8996	for / 8996	for / 9024
i / 8854	i / 8854	unto / 8999
his / 8473	his / 8474	his / 8476
a / 8234	a / 8237	a / 8244
lord / 7830	lord / 7830	lord / 7916
they / 7379	they / 7380	is / 7867
be / 7032	be / 7032	they / 7559
is / 7014	is / 7014	be / 7285
him / 6659	him / 6660	not / 7220
not / 6617	not / 6617	him / 7210
them / 6430	them / 6430	them / 6817

Tabla 11 - Resultados exactos de la prueba 1

Exacto	Multihash FSS	FSS
the / 64203	the / 64203	the / 64203
and / 51764	and / 51764	and / 51764
of / 34789	of / 34789	of / 34789
to / 13660	to / 13660	to / 13951
that / 12927	that / 12927	that / 13282
in / 12725	in / 12725	in / 12944
he / 10421	he / 10421	for / 12212
shall / 9840	shall / 9842	or / 12160
unto / 8997	unto / 9001	a / 12157
for / 8996	for / 8998	you / 12150

Tabla 12 - Resultados exactos de la prueba 2

En la segunda prueba hemos hecho servir valores más pequeños para los parámetros *m* y *h*. Se puede volver a ver la precisión al calcular los elementos del top-*k* y su respectiva estimación de la frecuencia. *FSS* en cambio reporta valores erróneos y con una precisión no muy buena.

Ya por acabar vamos a estresar aún más el algoritmo así que esta vez haremos servir la misma base de datos pero esta vez cada letra tendrá un peso igual a la potencia cúbica de su número total de letras, de esta manera daremos más énfasis a las letras largas que son las que usualmente aparecen poco

	FSS		*MULTIHASH FSS*	
PRUEBA	1	2	1	2
TOP-K	20	10	20	10
M	30	15	30	15
H	150	75	150	75
SUSTITUCIONES	21067	33799	422	338
TIEMPO DE EJECUCIÓN	52 ms	47 ms	83 ms	74 ms
PRECISIÓN	50%	40%	95%	90%

Tabla 13 - Resultados estadísticos de los dos algoritmos variando su h y m

En la primera prueba vemos que el rendimiento de *FSS* se ha desplumado dando resultados poco prácticos. A todo esto *Multihash FSS* a pesar de bajar su exactitud sigue dando resultados muy buenos y con buena estimación de sus frecuencias. En la segunda prueba vemos la continuación de la decadencia del *FSS.*

Exacto	Multihash FSS	FSS
the / 1733481	the / 1733481	the / 1733481
and / 1397628	and / 1397628	and / 1397628
shall / 1230000	shall / 1230125	shall / 1234328
children / 922624	children / 923244	righteousness / 967943
therefore / 901773	therefore / 902023	therefore / 936484
that / 827328	that / 827328	children / 928652
righteousness / 672282	righteousness / 745944	that / 832237
congregation / 628992	jerusalem / 643519	jerusalem / 778162
jerusalem / 591219	congregation / 637503	which / 740403
according / 581013	according / 583599	christ / 727381
unto / 575808	against / 577818	httppglaforgfundraising / 704870
against / 572124	unto / 575835	brethren / 703620
israel / 554040	israel / 564345	information / 702218
which / 552500	which / 552500	according / 701694
lord / 501120	lord / 501120	httpwwwgutenbergorg110 / 701508
their / 491500	their / 493118	httpwwwgutenbergorg / 698969
they / 472256	they / 472741	businesspglaforg / 697060
people / 462672	people / 469753	distribute / 696798
because / 415030	because / 415621	redistribution / 696546
them / 411520	themselves / 414970	httppglaforgdonate / 695764

Tabla 14 - Resultados exactos de la prueba 1

Con todos estos datos se puede asegurar que el *Multihash FSS* es un algoritmo mejor en relación al *FSS.* Su único inconveniente es el tiempo de ejecución. Dado la falta de conocimientos informáticos se han implementado unas funciones *hash* que podrían ser sustituida por otras con menor tiempo de ejecución y dando así mejores resultados.

Respecto al espacio en memoria aunque es un poco más que el del *FSS* sigue siendo mínimo. En todos estos datos que hemos podido observar hay uno que no se ha comentado y es que en todos las pruebas el *Multihash FSS* sustituye un número mucho menor de elementos en la lista monitorizada respecto al *FSS* y esto provoca más errores en el *FSS*. Además el hecho de trabajar con valores con peso provoca que haya una probabilidad mayor que un elemento sea sustituido en el *FSS* y no en el *Multihash FSS* y como consecuencia aumentar el error de estimación del *FSS* respecto al *Multihash FSS.*

7 Conclusiones

En este trabajo se ha dado una vuelta por los distintos algoritmos desarrollados para el cálculo del top-*k*. Se han analizado, comparado y a partir de allí se han sacado conclusiones para ver cuál podría ser el más óptimo. Una vez llegado a este punto se han propuesto mejoras algunas ya implementadas y otras como el *Multihash FSS* fruto del trabajo previo.

La realización de este trabajo ha necesitado de conocimientos de programación, en este caso Java, para elaborar los diferentes tipos de experimentos. La programación de los algoritmos esta adjunta al trabajo como un proyecto de Java.

Como futura líneas de investigación se propone encontrar la relación entre el número de funciones $hash$ usadas en el algoritmo *Multihash FSS* y su disminución en el error. En los resultados y experimentos obtenidos se puede apreciar una disminución del error de forma exponencial, es decir con dos funciones $hash$ el error disminuye bastante que con uno pero si son tres funciones $hash$ las que usamos vemos que el error se ha estabilizado de manera notoria.

Referencias

[1] G. Manku y R. Motwani. Approximate Frequency Counts over Data Streams en *Proceedings of the 28th ACM VLDB International Conference on Very Large Data Bases*, páginas 346-374, 2002.

[2] J. Misra y D. Gries. Finding Repeated Elements. *Science of Computer Programming*, 2:143-152, 1982.

[3] M. Charikar, K. Chen y M. Farach-Colton. Finding Frequent Items in Data Streams en *Proceedings of the 29th ICALP International Colloquium on Automata, Languages and Programming*, páginas 379-390, 2000.

[4] E. Demaine, A. López-Ortiz y J. Munro. Frequency Estimation of Internet Packet Streams with Limited Space en *Proceedings of the 10th ESA Annual European Symposium on Algorithms*, páginas 348-360, 2002.

[5] C. Jin, W. Qian, C. Sha, J. Yu y A. Zhou. Dynamicalli Mantaining Frequent Items over a Data Stream en *Proceddings of the 12th ACM CIKM International Conference on Information and Knowledge Management*, páginas 287-294, 2003.

[6] G. Cormode y S. Muthukrishnan. What's Hot and What's Not: Tracking Most Frequent Items Dynamically. *PODS 2003*, Junio 9-12, 2003, Sand Diego, CA.

[7] G. Cormode y S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. Elsevier Science, diciembre 2003.

[8] A. Metwally, D. Agrawal y A. El Abbadi. Efficient Computation of Frequent and Top-k Elements in Data Streams en *ICDT'05 Proceedings of the 10th international conference on Database Theory*, Páginas 398-412, 2005.

[9] B. Ellis. *Real-Time Analytics: Techniques to Analyze and Visualize Streaming Data*. Chapter 10: Approximating Streaming Data with Sketching, páginas 331 a 364. John Wiley & Sons, 2014.

[10] P. Flajolet, E. Fusy, O. Gandouet y F. Meunier. HyperLogLog: the analysis of a near-optional cardinality estimation algorithm en *Conference on Analysis of Algorithms, Discrete Mathematics and Theoretical Computer Science (DMTCS)*, France, 2007.

[11] H. Liu, Y. Lu, J. Han y J. He. Error Adaptative and Time-Aware Maintenance of Frequency Counts over Data Streams en *Advances in Web-Age Information Management, 7th International Conference, WAIM 2006*, Hong Kong, China, Junio 17-19, 2006.

[12] N. Manerikar y T. Palpanas. Frequent Items in Streaming Data: An Experimental Evaluation of the State-of-the-Art en Data & Knowledge Engineering**,** Tomo 68, Páginas 415-430, Ed. Elsevier, Abril 2009.

[13] N. F. Lopes Homen. *Behavioral Pattern Detection using Compact and Fast Methods*. Universidade Técnica de Lisboa, Instituto Superior Técnico. Noviembre 2011.

[14] R. M. Karp, S. Shenker, and C. H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. ACM Trans. Database Syst., 28(1):51–55, 2003.