Algoritmos de Streaming

Organización de Datos 75.06
Marzo 2014

Un "stream" es un flujo de datos infinito, los datos arriban a gran velocidad y debe ser posible procesarlos para extraer información y estadísticas a partir de los mismos sin poder almacenar todos los datos. La mayoría de los algoritmos de streaming trabajan en memoria usando una cantidad de espacio constante o acotada y realizan estimaciones de los datos que se quieren conocer a partir del stream.

Algunos ejemplos del mundo real de streams son:

- Las actualizaciones de status en Facebook.
- Los posteos en Twitter
- Las búsquedas en Google
- Los paquetes IP que arriban a un router
- Las compras realizadas en Amazon/Ebay
- Los clicks realizados por usuarios en páginas de un cierto sitio

Sampling

El primer método básico de streaming es almacenar un muestreo de los datos y luego a partir de ese muestreo realizar las consultas y estadísticas necesarias.

Hay dos formas posibles de mantener una muestra a partir de un stream:

- Que el tamaño de la muestra sea proporcional al stream
- Que el tamaño de la muestra sea fijo

Muestra proporcional al Stream

El principio aqui es muy simple: de todos los datos observados un cierto porcentaje se almacena y el resto se descarta. Luego se realizan las consultas necesarias sobre las muestras almacenadas. Las consultas pueden realizarse offline usando MapReduce.

El único detalle a considerar es de que forma decidir cuando un dato del stream pasa a ser parte del muestreo y cuando no. En algunos streams alcanza simplemente con almacenar o no el dato de acuerdo al porcentaje que queremos almacenar. Por ejemplo si queremos guardar el 1% de los datos vistos entonces por cada dato hacemos un random 0..99 y si sale 0 el dato es almacenado y sino es descartado.

Debe tenerse cuidado cuando se quiere guardar un porcentaje que no sea sobre el total de los datos. Por ejemplo si tenemos clicks realizados por usuarios de la forma (userid, timestamp, url) y queremos almacenar datos sobre el 10% de los usuarios debemos notar que:

Guardar el 10% de los datos no es igual a guardar los datos del 10% de los usuarios

Deberiamos entonces hacer la decisión de almacenar o no el dato en base al userid, una forma es mediante una funcion de hashing. En nuestro ejemplo si queremos guardar los datos del 10% de los usuarios podemos aplicar un hash 0..9 al userid y si es 0 almacenar el dato y sino descartarlo.

El muestreo proporcional a los datos permite todo tipo de consultas y estadísticas pero tiene como gran problema que la cantidad de datos almacenado crece indefinidamente. En streams para los cuales el volumen de información es realmente masivo es necesario acotar el tamaño del muestreo.

Muestra de tamaño fijo

Supongamos ahora que para un stream infinito queremos guardar una muestra de tamaño constante "s",si la cantidad de datos vistos hasta el momento es "n" entonces queremos que la probabilidad de que un dato esté en la muestra sea s/n.

El algoritmo que logra esto se llama "Reservoir Sampling"

La idea del algoritmo es muy simple: almacenar cada dato en la muestra con probabilidad s/n. Donde "s" es el tamaño de la muestra y "n" es la cantidad de datos procesados hasta el momento. Para los primeros "s" elementos todos se almacenan. Luego a partir del elemento s+1 se aplica una funcion de hashing 0..n y si el resultado es < s el dato se almacena y sino se descarta.

Si el dato se almacena es necesario que reemplace a algún otro elemento de la muestra. En cuyo caso se elige al azar un dato cualquiera y se lo reemplaza.

Teorema: La probabilidad de cualquier dato que esta en el muestreo es s/n

Demostración:

Vamos a probar que el algoritmo mantiene la propiedad que deseamos por inducción.

Luego de "n" elementos la probabilidad de cada elemento en la muestra es s/n
Queremos probar que luego de n+1 elementos la probabilidad de cada elemento en la muestra es s/n+1

Caso base: Luego de "s" elementos el muestreo cumple que cada elemento esta en el muestreo con probabilidad s/n (en este caso s/n=1)

Hipotesis Inductiva: Supongamos que luego de n elementos la probabilidad de cada elemento en la muestra es s/n

Queremos ver si para n+1 la probabilidad es entonces s/n+!

Cuando el elemento n+1 es observado podemos calcular la probabilidad de que un elemento del muestreo permanezca en la muestra como:

Probabilidad de que n+1 no vaya a la muestra + Probabilidad de que n+1 ingrese a la muestra * Probabilidad de que el elemento no sea eliminado de la muestra

Probabilidad de que n+1 no vaya a la muestra = 1 – (s/n+1)
Probabilidad de que n+1 ingrese a la muestra: s/n+1
Probabilidad de que un elemento de la muestra sea reemplazado: s-1/s

=> 1-(s/n+1) + (s/n+1) (s-1/s) = n/n+1

Entonces en el paso "n" el dato esta en la muestra con probabilidad s/n
y en el pao n+1 el dato se mantiene en la muestra con probabilidad n/n+1

La probabilidad de que un dato este en la muestra en el paso n+1 es entonces la probabilidad de que este en el paso n por la probablidad de que se mantenga en el paso n+1
(s/n) (n/n+1) = s/n+1

Que es lo que queríamos demostrar.

Calculo de Momentos

Sea un stream en el cual hemos observado n datos. Cada "dato" distinto aparece en el stream 1 o mas veces. Podemos decir que la frecuencia de cada dato "i" es Mi es decir la cantidad de veces que el iesimo dato se ha visto en el stream.

Definimos al "momento" de orden k del stream como

$$\sum Mi^k$$

Cuando k=0 lo que hacemos es sumar 1 por cada dato diferente en el stream es decir que el momento de orden cero de un stream es la cantidad de datos diferentes observados hasta el momento. Para calcular esto usaremos el algoritmo de Flajolet-Martin que describimos luego.

Cuando k = 1 lo que hacemos es simplemente sumar todas las frecuencias es decir contar la cantidad de datos que hemos observado en el stream. Esto es trivial pues solo necesitamos un contador e ir sumando 1 cada vez que vemos un dato. No hace falta ningún algoritmo para esto.

Cuando k=2 estamos calculando lo que se conoce como "Número sorpresa" de un stream, que es un indicador de si los datos en el stream se distribuyen en forma pareja o si algun dato aparece muchas mas veces que otro.

Por ejemplo supongamos que cada dato es un número de 1 a 9

Si tenemos 10 datos observados y son:

1,2,3,4,5,6,7,8,9,1

Tenemos a 1 con frecuencia 2 y a todos los demas con frecuencia 1.

El momento de orden 2 es: 2^2 + 8 * 1^2 = 12

Si en cambio los datos fueran:

1,1,2,3,1,1,4,5,1,7

Tenemos 5^2 + 5 = 30

Para calcular el momento de orden 2 de un stream usaremos el algorimo de Alon Matias y Szegedy

Algoritmo de Flajolet Martin

Este algoritmo calcula la cantidad de elementos distintos observados hasta el momento en un stream, es decir el momento de orden 0 del mismo.

Debemos notar que el problema no es trivial en absoluto ya que la única forma de resolverlo con exactitud es almacenando todos los elementos diferentes observados en algún tipo de estructura. En general realizar esto sobre datos infinitos no es posible.

El algoritmo de Flajolet y Martin es muy simple: A cada dato se le aplica una función de hashing que genera un número de m bits (m mayor a log2 n siendo n la cantidad de datos observados) y se observa con cuantos bits en cero comienza el resultado de la función. En memoria se mantiene la cantidad máxima de bits 0 observados que llamamos "r".

Ejemplo:

10001010 r= 0 (empieza con 1)
00101110 r= 2
01010101 r= 2 (no cambia)
00011010 r= 3

La cantidad de elementos diferentes en el stream puede estimarse como 2^r

Pseudo-Demostración:

Un 50% de los hashes empieza con 0
Un 25% de los hashes empieza con 00
Un 12.5% de los hashes empieze con 000
etc...

Por lo tanto si hemos visto un hash que comienza con tres ceros lo mas probable es que hayamos visto 8 elementos. (2^3 = 8)

Refinando el algoritmo:

Un problema evidente del algoritmo es que es extremadamente sensible a resultados del hash que por ejemplo tengan muchos ceros a izquierda o peor aun que sean todos ceros. Una vez que esto pase el algoritmo simplemente deja de funcionar.

Una primera aproximación es usar varias funciones de hashing y llevar varios contadores. Sobre esta cantidad de contadores, que puede ser grande por ejemplo del orden de miles podriamos calcular un promedio o la media.

Si bien esto mejora el algoritmo tanto el promedio como la media tienen problemas.

El promedio es, nuevamente, sensible a un valor muy extremo. Si alguna de las funciones de hashing da todos ceros, esa estimacion sera 2^r = maximo e influirá enormemente en el promedio con el cual estimamos la cantidad de elementos diferentes del stream.

La media es menos sensible a valores extremos pero tiene como problema que la media siempre será una potencia de 2 ya que cada contador estima 2^r. Esto tampoco es un valor que aproxime a la realidad.

La solución pasa por combinar ambas cosas, dividiendo los contadores en grupos podemos tomar la media de cada grupo, evitando de esta forma la influencia de valores extremos y luego hacer un promedio de todas las medias. Esta variante junto con un número alto de muestras es la mas eficiente para el algoritmo de Flajolet-Martin.

Algoritmo de Alon-Matias-Szegedy (AMS)

Este algoritmo se usa para calcular los momentos de orden 2, es decir el número sorpresa de un stream, el algoritmo es lo suficientemente amplio como para poder usarse para otros ordenes tema que no incluimos en el apunte.

Empezamos con la versión básica del algoritmo:

Elegir un número "k" de variables que vamos a mantener en memoria.
Cada variable k tiene 2 campos: valor y cantidad.

Por cada elemento del stream si el elemento está en alguna de esas k variables entonces incrementar esa variable en 1.

El momento de orden 2 se estima como el promedio de n(2ci-1) donde n es la cantidad de datos vistos hasta el momento y ci es la cantidad de cada variable ki.

Para mantener las "k" variables usaremos el algoritmo de reservoir sampling. Es decir que cada vez que procesamos un dato del stream si el mismo esta en la lista de variables aumentamos su frecuencia, si el dato no esta entonces con probabilidad k/n reemplazamos al azar alguna de las variables por el dato nuevo inicializado en cantidad = 1.

Cada una de las k variables es una estimación del momento de orden 2 del stream, para la estimación final dividimos las k variables en grupos y hacemos el promedio de cada grupo tomando luego la media de estos promedios. (Al reves que en Flajolet Martin donde tomabamos el promedio de las medias ya que aqui los valores no son potencias de 2)

Ejemplo con 3 variables (k=3)
Marcamos con * si el elemento nuevo ingresa a las variables (es decir que salio favorecido con probabilidad 3/n)

Dato	K1.elem	K1.val	K2.elem	K2.val	K3.elem	K4.val
1	1	1				
2	1	1	2	1		
3	1	1	2	1	3	1
2	1	1	2	2	3	1
4	1	1	2	2	3	1
2	1	1	2	3	3	1
5*	1	1	2	3	5	1

Dato	K1.elem	K1.val	K2.elem	K2.val	K3.elem	K4.val
3	1	1	2	3	5	1
4*	4	1	2	3	5	1
4	4	2	2	3	5	1
3	4	2	2	3	5	1
1	4	2	2	3	5	1

En total vimos 12 elementos

1 = 2 veces
2 = 3 veces
3 = 3 veces
4 = 3 veces
5 = 1 vez

Momento de orden 2 (real) = 4 + 9 + 9 + 9 + 1 = 32

La estimacion para k1 es 12(2*2-1) = 36
La estimacion para k2 es 12(2*3-1) = 60
La estimacion para k3 es 12(2*1-1) = 12

El promedio es (36+60+12) / 3 = 36 lo cual esta bastante cerca del resultado real (32)

Estadísticas sobre Ventanas Fijas

Para el siguiente problema vamos a considerar Streams binarios, es decir que cada dato puede ser 1 o 0. Tomamos una ventana fija de "m" bits con m realmente muy grande y queremos calcular cuántos bits en 1 vimos en los últimos k bits con k<=m.

Este modelo es muy flexible y puede usarse en muchísimas situaciones. Por ejemplo podemos tener un stream por producto y agregar un bit por cada venta realizada indicando con 1 si el vendimos ese producto y con 0 si fue otro. De esta forma podríamos saber cuantas veces se vendió un cierto producto en las últimas 100.000 ventas o cuántas veces se uso un cierto término en Google en las últimas n búsquedas. Etc.

Debemos notar dos cosas para entender la dificultad del algoritmo:

- n es un número muy grande por lo que no podemos simplemente almacenar los últimos n bits
- Para un n fijo debemos poder calcular la cantidad de unos para cualquier k <= n, no solo para la ventana completa.

El algoritmo que usaremos fue desarrollado por Datar, Gionis, Indyk y Motwani. A partir de ahora lo llamaremos DGIM. Es un algoritmo cuyo funcionamiento es realmente muy extraño y de todos los que presentaremos en el apunte es el mas complejo de entender sin embargo una vez entendido es un algoritmo bastante simple. Podriamos decir que es algo asi como andar en bicicleta, aprender parece difícil pero una vez que lo hacemos no nos olvidamos mas.

Sobre la ventana de m bits el algoritmo va a mantener k sub-ventanas que no pueden superponerse. Por cada sub-ventana va a llevar 2 valores: la posición en la ventana donde comienza la sub-ventana y la cantidad de unos que hay en la misma. Por cada bit procesado a la posición de cada sub-ventana le sumamos uno (para mantener el offset)

Se sabe donde termina cada sub-ventana porque se sabe donde comienza la anterior y la última sub-ventana esta acotada por el fin de la ventana grande: "m". La posición donde comienza la ventana se calcula modulo "m". En el apunte no lo haremos para no confundir.

Para ver como se construyen estas sub-ventanas veamos como funciona el algoritmo.
Procesamos bit por bit el stream, si el bit es 0 lo ignoramos completamente. Si el bit es 1 entonces creamos una ventana de tamaño 1 para ese bit con contador = 1.

Luego debemos analizar cuantas sub-ventanas hay con 1 bit en total.
Si hay 1 o 2 no hacemos nada.
Si hay 3 entonces combinamos las dos sub-ventanas mas viejas en una nueva sub-ventana de 2 bits.
Y luego vemos cuantas sub-ventanas de 2 bits hay
Si hay 1 o 2 no hacemos nada.
Si hay 3 entonces combinamos las dos sub-ventanas mas viejas en una nueva sub-ventana de 4 bits.
Y asi sucesivamente...

Ejemplo, tomando un m que suponemos grande para que no influya.
Notar que el offset de las sub-ventanas siempre apunta al bit mas reciente de las mismas y que ese bit siempre es 1.
Recordar que siempre se combinan las sub-ventanas mas viejas.

```
Stream: [1]
[pos=0 bits=1]
Stream: [1]0
[pos=1 bits=1]
Stream: [1]0[1]
[pos=2 bits=1] [pos=0 bits=1]
Stream: [1]0[1]
[pos=3 bits=1] [pos=1 bits=1]
Stream: [1]0[1]00
[pos=4 bits=1] [pos=2 bits=1]
Stream: [1]0[1]00[1]
[pos=5 bits=1] [pos=3 bits=1] [pos=0 bits=1]

Aqui tenemos 3 sub-ventas con 1 bit, combinamos las dos mas viejas

Stream: [101]00[1]
[pos=3 bits=2] [pos=0 bits=1]
Stream: [101]00[1][1]
[pos=4 bits=2] [pos=1 bits=1] [pos=0 bits=1]
Stream: [101]00[1][1]0
[pos=5 bits=2] [pos=2 bits=1] [pos=1 bits=1]
Stream: [101]00[1][1]0[1] (combinar)
Stream: [101]00[11]0[1]
[pos=6 bits=2] [pos=2 bits=2] [pos=0 bits=1]
Stream: [101]00[11]0[1][1]
[pos=7 bits=2] [pos=3 bits=2] [pos=1 bits=1] [pos=0 bits=1]
Stream: [101]00[11]0[1][1]0
```

```
[pos=8 bits=2] [pos=4 bits=2] [pos=2 bits=1] [pos=1 bits=1]
Stream: [101]00[11]0[1][1]0[1] (combinar)
Stream: [101]00[11]0[11]0[1] (combinar, hay 3 de longitud 2)
Stream: [1010011]0[11]0[1]
[pos=5 bits=4] [pos=2 bits=2] [pos=0 bits=1]
```

Con la estructura de sub-ventanas para calcular cuantos 1s hay en una cierta sub-ventana de longitud k lo que hacemos es sumar la cantidad de bits de todas las sub-ventanas comprendidas y la mitad de la ultima.

Ejemplo si tenemos la salida de nuestro seguimiento:

```
[pos=5 bits=4] [pos=2 bits=2] [pos=0 bits=1]
```

Y queremos saber cuantos 1s hay en los ultimos 6 bits del stream notamos que las ventanas de pos=3 y pos=0 quedan dentro de estos 6 bits y la ventana de pos=11 parcialmente.

Entonces estimamos 1+2+4/2 = 5 bits

Si recordamos el stream: `101001`**`101101`**

Vemos que la cantidad real de 1s en los ultimos 6 bits era 4, la estimación es cercana.
Sin embargo para este caso particular como sabemos que buscamos 6 bits y la tercer sub-ventana justo comienza en el sexto bit podemos saber que ese bit es un uno pues todas las ventanas comienzan con 1. Por lo tanto podemos calcular 1+2+1 = 4 bits. (es un caso particular)

DGIM necesita en promedio log2M sub-ventanas y por cada sub-ventana necesita logN bits para la posicion y loglogN bits para la cantidad de 1s. Es decir que el espacio en memoria siempre esta acotado al logarimo del tamaño de la ventana y esto es perfectamente manejable.

El error de DGIM es de aproximadamente un 50%, esto es bastante alto. Para reducirlo lo que hacemos es en lugar de admitir 2 sub-ventanas como máximo de cada tamaño admitir "r". Haciendo esto el error de DGIM es de 1/r. Es decir que si queremos un error del 10% admitimos hasta 10 ventanas de cada longitud pero necesitamos almacenar mayor cantidad de sub-ventanas con mayor consumo de memoria.

Extension para valores enteros

Supongamos que el stream en lugar de ser binario es de valores enteros y queremos la suma de los últimos k elementos con k<n siendo n un número muy grande.

Se puede usar DGIM facilmente para resolver este problema si suponemos que los enteros son números de "z" bits cada uno. Podemos entonces usar un DGIM por cada bit del número.

Luego tenemos la cantidad 1s en promedio en los últimos k elementos por cada bit del número y podemos estimar el número como

$$\sum Ci * 2^i$$

Decaying Windows

Otra forma de analizar el problema de contar cuantos 1s aparecen en los últimos bits de un stream consiste en aplicar una función que vaya decayendo el peso de cada bit en 1 ya observado a medida que procesamos el stream.

Si nos interesan los últimos "n" bits del stream entonces usamos una constante c = 10^-n, llevamos un contador que comienza en 0 y por cada bit del stream multiplicamos el contador por 1-c y luego sumamos el bit (1 o 0).

Ejemplo:

Sea c=0.1 (1-c=0.9)
1 (1)
10 (0.9)
100 (0.81)
1001 (1.729)
10011 (2.5561)
100110 (2.30049)
1001100 (2.070441)
10011001 (2.8633969)
etc..

En heneral se usan valores del orden de 10^-n con n grande.

Supongamos ahora que tenemos un stream de URLs clickeadas por usuarios y queremos saber cuales son las URLs mas populares en los ultimos "n" clicks. Podemos implementar esto con el algoritmo que presentamos si asociamos a cada URL un stream binario en donde 0 indica que la URL no fue clickeada y 1 que si.

La cantidad de URLs es muy grande por lo que queremos llevar solo una cantidad fija en memoria. El algoritmo entonces seria algo asi:

Por cada URL del stream

Multiplicar todos los valores en memoria por (1-c)
Eliminar de memoria todas las URLs por debajo de un cierto umbral (ej 0.5)
Si la URL que estamos procesando esta en memoria sumarle 1 sino agregarla con valor 1.

Por ejemplo sea el stream

ABCABBABCD

Y sea c=0.1 con umbral 0.6

	A	B	C	D
A	1			
B	0.9	1		
C	0.81	0.9	1	

A	1.73	0.81	0.9	
B	1.56	1.73	0.81	
B	1.4	2.56	0.73	
A	2.26	2.3	0.66	
B	2.03	3.07		
C	1.83	2.76	1	
D	1.65	2.48	0.9	1

Filtrando Streams

Dado un stream queremos saber si los elementos que observamos en el mismo pertenecen o no a un cierto conjunto de elementos predefinidos. Por ejemplo podemos tener una base de direcciones de email confiables y queremos analizar si un email puede ser spam o no verificando si su emisor está o no en esta lista de direcciones confiables.

La "base" de elementos contra los cuáles verificamos es siempre muy grande, no sirve mantenerlas en memoria en un hash o estructura similar.

El método de filtrado mas popular consiste en usar los llamados "Filtros de Bloom". Un filtro de Bloom es un vector binario de "B" bits y k funciónes de hashing 0..B. Para "agregar" un elemento al filtro le aplicamos las funciones de hashing y luego encendemos en 1 los bits apuntados por las funciónes. Se prenden k o menos bits según haya o no colisiones.

Para verificar si un dato del stream pertenece a nuestro conjunto le aplicamos las funciones de hashing y verificamos si los bits están en todos en 1. Si alguno está en 0 entonces el elemento no pertenece al conjunto. Si todos los bits están en 1 entonces el elemento pertenece al mismo con una cierta probabilidad ya que podría haber un falso positivo si alguno de esos bits fue encendido por algún otro elemento.

Si tenemos k funciones de hashing, n elementos a hashear y m bits. Luego de hashear todos los elementos. ¿Cuántos bits quedan en 1 y cuántos en 0 en el vector? ¿Cuál es la probablidad de un falso positivo?

Comenzamos con algunos calculos basicos:

La probablidad de que una función encienda un cierto bit es 1/m
La probablidad de que una función no encienda un cierto bit es 1-(1/m)
Si tenemos k funciones de hashing la probabilidad de que ninguna encienda un cierto bit es:
$[1-(1/m)]^k$
Luego de insertar "n" elementos la probabilidad de que un cierto bit siga siendo 0 es
$[1-(1/m)]^{kn}$
La probablidad de que un cierto bit sea 1 es entonces
$1-([1-(1/m)]^{kn})$
La probabilidad de que las k posiciones a testear sean 1 entonces es:
$[1-([1-(1/m)]^{kn})]^k$

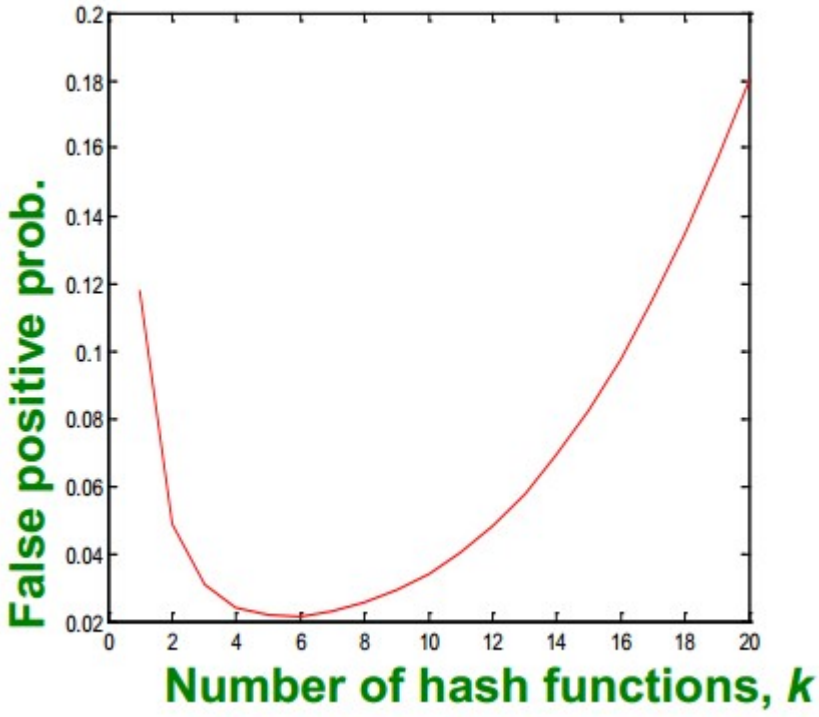
Y una estimación para esta fórmula es: $(1-e^{(-kn/m)})^k$

Esta funcion se puede graficar para probar valores de n,m y k que funcionen correctamente.

Para valores de n y m fijos el valor óptimo de k se puede calcular de la forma:

$$k = (m/n) log2$$

Para m y n fijos el grafico de k segun la probabilidad de falsos positivos es:



Como podemos ver la probabilidad de un falso positivo va bajando hasta llegar al óptimo y luego sube ya que hay demasiados bits en 1 en el filtro.

Si usamos el k óptimo entonces la cantidad de bits a usar puede calcularse como:

$$m = -[(n \ln p)]/[(ln2)^2]$$

siendo p la probabilidad que queremos para un falso positivo.

Ejemplo:
Si tenemos mil millones de direcciones de email y queremos que la probabilidad de un falso positivo sea: p=0.01
entonces

m = - (1000000000 ln 0.01) / (ln2)^2
m = 9585 millones de bits

y k = 2.88 es decir 3 funciones de hashing

Algo interesante sobre los filtros de Bloom es que puede usarse el filtro para estimar la cantidad de elementos que han sido insertados en el mismo (Swamidass & Baldi, 2007)

$$E = -(M \ln[1 - (X/M)])/k$$

M = cantidad de bits del filtro.
X = cantidad de bits en 1

Los filtros de Bloom son muy sencillos en su concepto pero aplicarlos correctamente requiere el uso de estas fórmulas y no todo el mundo las conoce!

Extensión para admitir borrado:

Si queremos poder insertar y eliminar elementos del filtro entonces la estructura normal no sirve porque no podemos al borrar apagar los bits en 1 indicados por las k funcionesd de hashing pues estos bits podrían corresponder a otros elementos ya insertados en el filtro y luego al buscarlos el filtro nos diría que no están.

La solución es usar un "counting filter" que en lugar de 1 bit por cada posicion tiene un entero de f bits, de esta forma cada vez que se inserta se incrementan los enteros apuntados por las funciones de hashing y cuando se elimina se decrementa. La pertenencia es simplemente verificar que las posiciones sean todas distintas de cero.

La desventaja, evidente, es la mayor cantidad de espacio necesaria.

Elementos Mas Frecuentes en un Stream

Nuestro próximo tema es estimar cuáles son los elementos mas frecuentes de un stream. Esto serviría para analizar las búsquedas mas populares en Google, cuáles son los trending topics en Twitter, etc.

Es evidente que mantener en memoria un contador por cada elemento del Stream no es posible, necesitamos una solución que utilice una cantidad de memoria acotada.

Vamos a ver dos algoritmos para resolver este problema.

Algoritmo de Misra-Gries

Este algoritmo es muy simple, mantiene en memoria k variables y cada variable contiene el elemento y su frecuencia.

Los primeros k elementos del stream van todos a la muestra.

Por cada elemento del stream si el mismo esta en la muestra aumentamos su frecuencia en 1. Si el elemento no esta en la muestra entonces restamos 1 a todos los elementos en la muestra y descartamos los que queden en cero.

Notemos que por definición la estimación que hace este algoritmo puede ser igual a la frecuencia real del item si nunca salió de la muestra. La estimación puede ser menor a la real si el item salió alguna vez de memoria. La estimación nunca puede ser mayor a la frecuencia real del item.

Se demuestra que:

freal – (n/k) <= festimada <= freal (n= cantidad de elementos observados)

Como es evidente la estimación es mejor cuanto mayor sea la cantidad de variables que mantenemos en memoria.

Algoritmo Count-Min

Este algoritmo está basado en filtros de Bloom de tipo "counting" es decir que tienen en cada posición un número entero.El algoritmo fue desarrollado por Cormode y Muthukrishnan en 2003.

Se usan "d" filtros en total, cada uno de "w" bits y asociado a UNA función de hashing. Hay entonces "d" funciones de hashing en total.
Cuando se observa un elemento del stream se le aplican las "d" funciones de hashing y en cada uno de los "d" counting filters se incrementa la posición indicada por la función de hashing.

Para estimar la frecuencia de un cierto elemento lo que se hace es hashearlo con las "d" funciones de hashing, recuperar los valores de los filtros y tomar el MINIMO como el valor de su frecuencia. Este valor es una estimación, nunca puede ser menor a la frecuencia real del item, puede ser igual o mayor.

Ejemplo:

Si tenemos los filtros:

[4,3,2,6,3,2,1,3]
[5,3,4,3,9,4,9,8]
[3,6,7,4,9,9,5,3]

y para un cierto elemento las funciones de hashing son:

h1 = 3 f1(3) = 6 (contando desde 0)
h2 = 1 f2(1) = 3
h3 = 0 f3(0) = 3

Min(6,3,3) = 3

Entonces estimamos la frecuencia del item como 3.

Para llevar registro de los "i" elementos mas frecuentes en un stream podemos simplemente reservar memoria para "i" elementos y a medida que procesamos el stream actualizamos los filtros y calculamos el count-min del elemento, si el mismo supera al menor de la lista de i elementos mas frecuentes simplemente lo reemplazamos.