

# ANÁLISIS DE DATOS MASIVOS

## MUESTREO Y FILTRADO

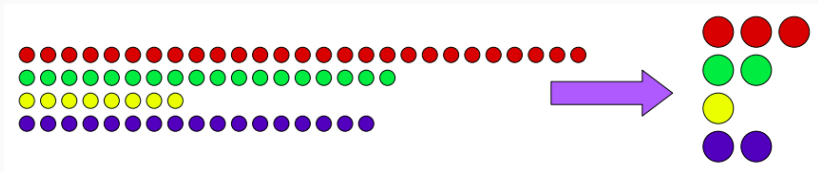
---

Blanca Vázquez

8 de octubre de 2024

# MUESTREO DE FLUJOS DE DATOS (1)

- En muchos casos no es posible almacenar todos los datos de un flujo, por lo que es necesario realizar muestreo
- Objetivo: seleccionar un subconjunto de datos del flujo de tal manera que se puedan realizar que sean representativas de todo el flujo de datos.



## MUESTREO EN FLUTOS DE DATOS (2)

- Ventaja
  - Costo computacional más bajo debido a que estamos usando solo una porción del flujo.
- Retos
  - ¿Cómo sabemos qué tan largo es el flujo de datos?
  - ¿Cada cuánto tiempo debemos muestrear?
  - ¿Cómo hacemos el muestreo?

- Ventanas deslizantes
- Muestreo aleatorio
- Muestreo de tamaño fijo

# VENTANAS DESLIZANTES

- Las consultas se realizan sobre una *ventana* de tamaño  $w$ .
- Si un elemento llega en el tiempo  $t$ , expira en el tiempo  $t + w$ .

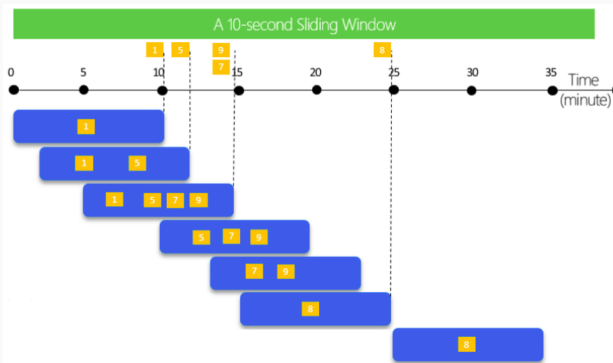


Imagen tomada de Azure Stream Analytics

## VENTANAS DESLIZANTES: EJEMPLO

- En este ejemplo el tamaño de la ventana deslizante es 6, observamos el traslape entre datos.

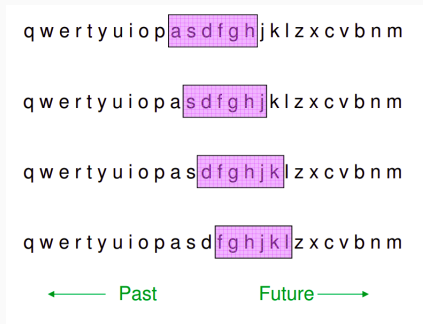


Imagen tomada de J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets, <http://www.mmds.org>

# PROMEDIO DE VENTANA DESLIZANTE (1)

Calificaciones

10.0	7.8	6.8	8.0	9.2	9.0
------	-----	-----	-----	-----	-----

**Definir:**

- Tamaño de la ventana: 3

## PROMEDIO DE VENTANA DESLIZANTE (2)

Calificaciones

Tamaño de la ventana: 3

10.0	7.8	6.8	8.0	9.2	9.0
------	-----	-----	-----	-----	-----

$\text{suma\_ventana} = (10 + 7.8 + 6.8) / 3$   
 $\text{resultado} = 8.2$

Secuencia resultante: [8.2]



## PROMEDIO DE VENTANA DESLIZANTE (3)

Calificaciones

Tamaño de la ventana: 3

10.0	7.8	6.8	8.0	9.2	9.0
------	-----	-----	-----	-----	-----

$\text{suma\_ventana} = (7.8 + 6.8 + 8.0) / 3$   
 $\text{resultado} = 7.53$

Secuencia resultante: [8.2, 7.53]

## PROMEDIO DE VENTANA DESLIZANTE (4)

Calificaciones

Tamaño de la ventana: 3

10.0	7.8	6.8	8.0	9.2	9.0
------	-----	-----	-----	-----	-----

$\text{suma\_ventana} = (6.8 + 8.0 + 9.2) / 3$   
 $\text{resultado} = 8$

Secuencia resultante: [8.2, 7.43, 8, ]

## PROMEDIO DE VENTANA DESLIZANTE (5)

Calificaciones

Tamaño de la ventana: 3

10.0	7.8	6.8	8.0	9.2	9.0
------	-----	-----	-----	-----	-----

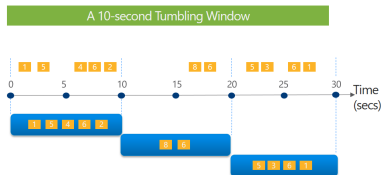
$\text{suma\_ventana} = (8.0 + 9.2 + 9.0) / 3$   
 $\text{resultado} = 8.73$

Secuencia resultante: [8.2, 7.43, 7.96, 8.73]

# VENTANAS DE SALTOS DE TAMAÑO CONSTANTE

- Se divide el flujo de datos en segmentos de tiempo sin traslape.

Tell me the count of tweets per time zone every 10 seconds



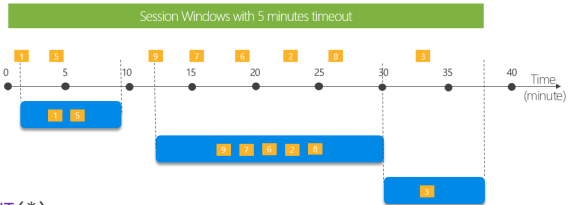
```
SELECT TimeZone, COUNT(*) AS Count
FROM TwitterStream TIMESTAMP BY CreatedAt
GROUP BY TimeZone, TumblingWindow(second,10)
```

Imagen tomada de Azure Stream Analytics

# VENTANAS DE SESIÓN

- Agrupan eventos que llegan en tiempos similares, filtrando los periodos en los que no se recibe ningún dato.
- Los parámetros de este tipo de ventana son el tiempo de espera y duración máxima.

Tell me the count of tweets that occur within 5 minutes to each other.



```
SELECT Topic, COUNT(*)  
FROM TwitterStream TIMESTAMP BY CreatedAt  
GROUP BY Topic, SessionWindow(minute, 5, 10)
```

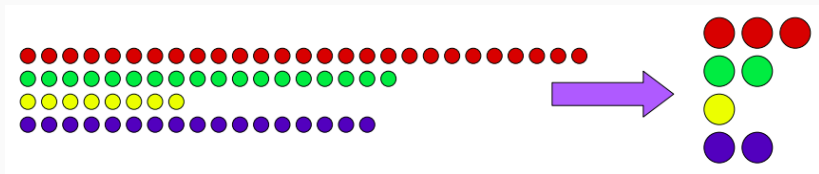
Imagen tomada de Azure Stream Analytics

- Los elementos tienen la misma probabilidad de ser seleccionados.
- Ejemplo: Cobertura de la vacuna anti-sarampión entre 1,200 niños de la escuela 'Juan Escutia':
  - Muestra: 60 niños
  - Hacer una lista de todos los niños
  - Numerarlos del 1 al 1,200
  - Selección aleatoria de 60 números (probabilidad igual)

- Los elementos se seleccionan en base a criterios o reglas específicas.
- Ejemplo: únicamente se registraran a los pacientes que acudan a la clínica en cierto día u horario particular.
  - Es posible que los elementos seleccionados sean poco representativos de todos los conjuntos generados

## MUESTREO DE TAMAÑO FIJO

Consiste en muestrear una porción fija de los elementos recibidos (digamos 1 de cada 10 recibidos)





## MUESTREO DE TAMAÑO FIJO: EJEMPLO (1)

- Flujo de datos: consultas de usuarios
- Entrada: flujos de datos en forma de tupla

User (IP)	Query	Time
-----------	-------	------

- ¿Qué fracción de las consultas de un usuario son duplicadas? (operaciones sobre el tiempo)

## MUESTREO DE TAMAÑO FIJO: EJEMPLO (2)

- Supongamos que cada usuario realiza  $x$  número de consultas únicas y  $d$  número de consultas repetidas
- El total de consultas que el usuario hace es  $x + 2d$
- Si realizamos un muestreo fijo con 1 de 10, mantendríamos  $\frac{1}{10}$  de todas las consultas
  - $\frac{x}{10}$  (de todas las consultas únicas)
  - $\frac{2d}{10}$  (de las consultas duplicadas)

## MUESTREO DE TAMAÑO FIJO: EJEMPLO (3)

- ¿Qué fracción de las consultas de un usuario son duplicadas?
  - De las  $d$  consultas duplicadas, en la muestra solo tendríamos  $\frac{d}{100}$ 
    - $\frac{d}{100} = \frac{1}{10} \cdot \frac{1}{10} \cdot d$
  - De todo el conjunto de preguntas repetidas,  $\frac{18 \cdot d}{100}$  aparecerían realmente una vez.
    - $\frac{18}{100}$  = es la probabilidad de que una de las repeticiones esté en el  $\frac{1}{10}$  seleccionado y el otro en el  $\frac{9}{10}$  no seleccionado
    - $\frac{18 \cdot d}{100} = (\frac{1}{10} \cdot \frac{9}{10} + \frac{9}{10} \cdot \frac{1}{10}) \cdot d$
  - La fracción de consultas sería

$$\frac{\frac{d}{100}}{\frac{x}{10} + \frac{d}{100} + \frac{18d}{100}} = \frac{d}{10x + 19d}$$

## OBTENCIÓN DE MUESTRAS REPRESENTATIVAS

- Como observamos hacer el muestreo tomando una muestra de cada usuario, puede arrojar resultados poco confiables.
- ¿Y si en lugar de tomar  $\frac{1}{10}$  de las búsquedas de cada usuario, tomamos  $\frac{1}{10}$  de todos los usuarios?
  - Es decir, vamos a almacenar todas las búsquedas, descartando las consultas del resto de usuarios
  - Tomando la IP del usuario como ID, muestreamos  $\frac{a}{b}$  usuarios usando una función *hash* que almacena los IPs en  $b$  cubetas, agregando las consultas del usuario si su valor *hash* es menor a  $a$
  - Como resultado, tendríamos una muestra más representativa.

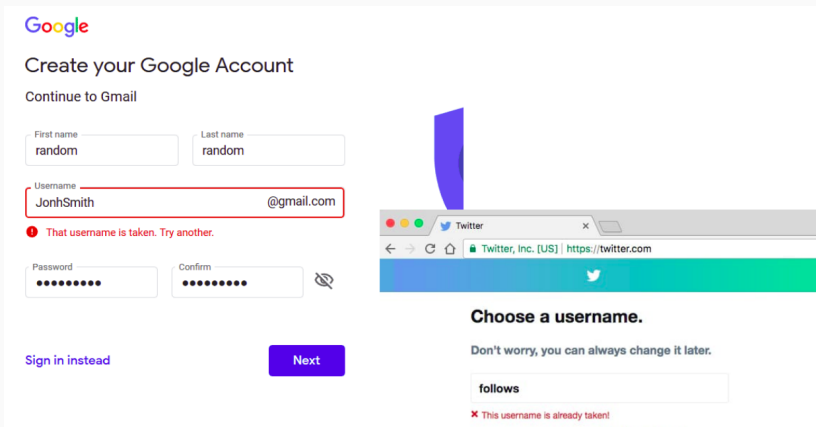
## MUESTREO DE PRESA (RESERVOIR SAMPLING)

- Consiste en muestrear los primeros  $k$  datos recibidos y los mantiene en memoria (presa)
- Cada nuevo elemento recibido tiene una probabilidad de  $\frac{k}{n}$  de reemplazar un elemento actual
- Procedimiento general
  1. Toma los primeros  $k$  elementos del flujo como muestra
  2. Supongamos que hemos visto  $n - 1$  elementos, y ahora recibimos el  $n$ -ésimo elemento ( $n > k$ )
  3. Con probabilidad  $\frac{k}{n}$ , mantenemos el elemento  $n$ -ésimo, reemplazando uno de los  $k$  elementos en la muestra

- Seleccionar elementos del flujo que cumplan cierto criterio y descartar el resto.
- Ejemplo: dado un flujo de números reales, filtrar los que sean mayores a 50.
  - Flujo: 33, 71, 58, 12, 41, 56, 3, 89
  - Elementos seleccionados: 71, 58, 56, 89
- Esta tarea se vuelve más difícil cuando el criterio requiere verificar si el elemento pertenece a un conjunto dado, especialmente si este conjunto es tan grande que no cabe en memoria

# PERTENENCIA A UN CONJUNTO

- Ejemplo: ¿cómo podemos revisar rápidamente la disponibilidad de un nombre dentro de cientos de millones existentes?



Google

## Create your Google Account

Continue to Gmail

First name  
random

Last name  
random

Username  
JonhSmith@gmail.com

That username is taken. Try another.

Password

Confirm

Sign in instead

Next

Twitter

Twitter, Inc. [US] | https://twitter.com

### Choose a username.

Don't worry, you can always change it later.

follows

This username is already taken!

- Supongamos que almacenamos todos los nombres alfabéticamente y comparamos el nuevo nombre con el que aparece a mitad de la lista
- Si el nombre coincide, devuelve *intentar nuevamente*
- En caso contrario, busca nuevamente en la mitad de los nombres restantes (arriba - abajo)
- Se repite el proceso, hasta que encuentre una coincidencia o hasta que termina la búsqueda y no encuentre nada.



- Es una estructura de datos probabilista y se emplea para evaluar si un elemento pertenece a un conjunto<sup>1</sup>.
- Elimina la mayoría de los elementos que no pertenecen al conjunto.
- Es muy eficiente en memoria, ya que no requiere mantener el conjunto en memoria
- Tiene falso positivos

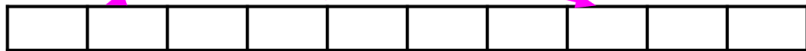
---

<sup>1</sup>Fue desarrollado por Burton Howard Bloom en 1970.

- Consiste en un arreglo de  $m$  bits inicializados con 0
- Construcción
  1. Para cada elemento  $s$  del conjunto, se calculan los valores *hash* con  $k$  funciones distintas  $h_1(s), h_2(s), \dots, h_k(s)$ .
  2. Los  $k$  bits en las posiciones correspondientes a los  $k$  valores *hash* se ponen a 1.
- Verificación de pertenencia de un nuevo elemento  $\tilde{s}$ 
  1. Calcula los valores *hash* para  $\tilde{s}$ :  $h_1(\tilde{s}), h_2(\tilde{s}), \dots, h_k(\tilde{s})$
  2. Si todos los bits en las posiciones correspondientes a los  $k$  valores *hash* son 1, entonces es probable que el elemento  $\tilde{s}$  pertenezca al conjunto, en caso contrario definitivamente no pertenece

## Estructura de un filtro de Bloom

Cada celda vacía representa un bit



1

2

3

4

5

6

7

8

9

10

Índices del vector

# FILTRO DE BLOOM VACÍO

**Filtro de Bloom vacío:** es un vector de  $m$  bits donde todos los valores son ceros.

0	0	0	0	0	0	0	0	0	0
1	2	3	4	5	6	7	8	9	10

### ¿Cómo construimos un filtro de Bloom?

Para añadir un elemento  $x$  al filtro  $S$ :

- $x$  debe transformarse a un conjunto de bits a través de  $k$  funciones hash.
- El resultado de cada función indica el índice dentro del filtro (el valor 0 debe cambiarse a 1).

# DEFINIENDO M, N Y K

Filtro de  
Bloom vacío

0	0	0	0	0	0	0	0	0	0
1	2	3	4	5	6	7	8	9	10

Antes de empezar debemos definir:

- $m$  = tamaño del vector (número de bits)
- $n$  = número de elementos agregar
- $k$  = número de funciones hash



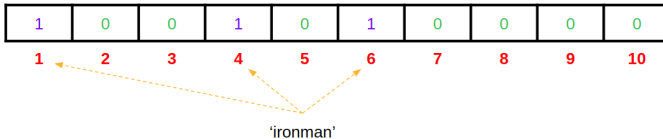
- $m = 10$
- $n = 2$
- $k = 3$

# IRONMAN!

- Añadir al filtro el nombre de usuario: *'ironman'*
  1. Calculamos las funciones hash, la salida será el índice que debemos cambiar a 1.

```
(mat_datmas) blanca@blanca-G7-7588:~/Desktop/virtualEnvironment/mat_datmas$  
Python 3.6.9 (default, Nov 7 2019, 10:44:02)  
[GCC 8.3.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> import mmh3  
>>> mmh3.hash('ironman',1) % 10  
4  
>>> mmh3.hash('ironman',2) % 10  
1  
>>> mmh3.hash('ironman',3) % 10  
6
```

2. Colocamos '1' en cada bit, usando el resultado de la función hash.

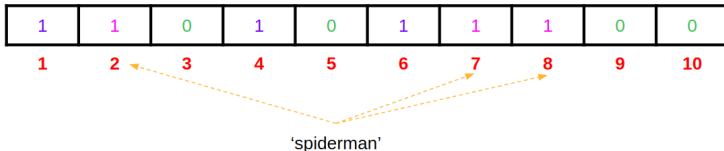


# SPIDERMAN!

- Añadir al filtro el nombre de usuario: *'spiderman'*
  1. Calculamos las funciones hash, la salida será el índice que debemos cambiar a 1.

```
>>> mmh3.hash('spiderman',1) % 10  
2  
>>> mmh3.hash('spiderman',2) % 10  
8  
>>> mmh3.hash('spiderman',3) % 10  
7  
>>> □
```

2. Colocamos '1' en cada bit, usando el resultado de la función hash.





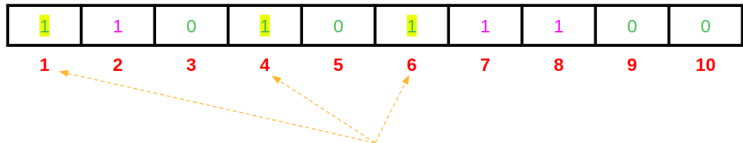
# BUSCANDO EN EL FILTRO

Lo más importante de un filtro de Bloom es buscar un elemento dentro del vector S

- Buscar si en S existe: *'ironman'*

1. Calculamos las funciones hash, la salida será el índice que debemos cambiar a 1.

```
>>> mmh3.hash('ironman',1) % 10
4
>>> mmh3.hash('ironman',2) % 10
1
>>> mmh3.hash('ironman',3) % 10
6
```



Si todos los índices (arrojados de la función hash) son 1s, entonces decimos que: *'ironman'* *probablemente* están presente en S

Lo más importante de un filtro de Bloom es buscar un elemento dentro del vector S

- Buscar si en S existe: *'thanos'*

1. Calculamos las funciones hash, la salida será el índice que debemos cambiar a 1.

```
>>> mmh3.hash('thanos',1) % 10
7
>>> mmh3.hash('thanos',2) % 10
8
>>> mmh3.hash('thanos',3) % 10
6
```

1	1	0	1	0	1	1	1	0	0
1	2	3	4	5	6	7	8	9	10

Si todos los índices (arrojados de la función hash) son 1s, entonces decimos que: *'thanos' probablemente están presente en S*

# FALSOS POSITIVOS

Lo más importante de un filtro de Bloom es buscar un elemento dentro del vector S

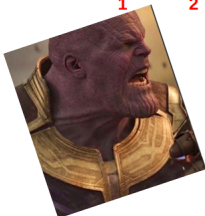
- Buscar si en S existe: *'thanos'*

1. Calculamos las funciones hash, la salida será el índice que debemos cambiar a 1.

```
>>> mmh3.hash('thanos',1) % 10  
7  
>>> mmh3.hash('thanos',2) % 10  
8  
>>> mmh3.hash('thanos',3) % 10  
6
```



**Falso positivo**



Si todos los índices (arrojados de la función hash) son 1s, entonces decimos que: *'thanos' probablemente están presente en S*

Dependiendo de la aplicación, un falso positivo puede representar un gran problema o simplemente mantenerse.

¿Cómo reducir los falsos positivos?

- Más espacio (incrementar el tamaño del arreglo)
- Incrementar el número de  $k$  (funciones hash)

## FALSOS POSITIVOS (1)

- La probabilidad de que 1 bit no sea puesto a 1 durante el registro de un elemento es

$$1 - \frac{1}{m}$$

- Para  $k$  funciones *hash* esto es

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

- Usando la identidad de  $e^{-1}$

$$\left(1 - \frac{1}{m}\right)^{kn} = \left(\left[1 - \frac{1}{m}\right]^m\right)^{\frac{kn}{m}} \approx e^{-\frac{kn}{m}}$$

- La probabilidad de que 1 bit sí sea puesto a 1 para m bits, n elementos y k funciones hash

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

- La probabilidad de que todos los bits sean 1 y no pertenezcan al conjunto es

$$\left[1 - \left(1 - \frac{1}{m}\right)^{k \cdot n}\right]^k$$

Propiedades de las funciones hash usadas en el filtrado de Bloom:

- Independientes
- Uniformemente distribuidas (dado un conjunto de valores, cada valor tiene la misma probabilidad de suceder).
- Deben ser rápidas (para su cálculo)
- No criptográficas (las funciones criptográficas son más estables, pero costosas).

Cuando el número de funciones hash incrementa, el filtrado se vuelve lento.

Ejemplos de funciones con bajas tasas de colisiones y no criptográficas:

- MURMUR: multiplicar (MU), rotar(R),multiplicar (MU), rotar(R).
- FNV: Fowler, Noll y V es un indexador rápido
- Jenkins o HashMix



- **Medium** usa filtros de Bloom para recomendar publicaciones a los usuarios, filtrando las publicaciones que ya ha visto el usuario
- **Quora** filtra historias no vistas.
- **Google Chrome** usó filtros de Bloom para detectar URLs maliciosas.