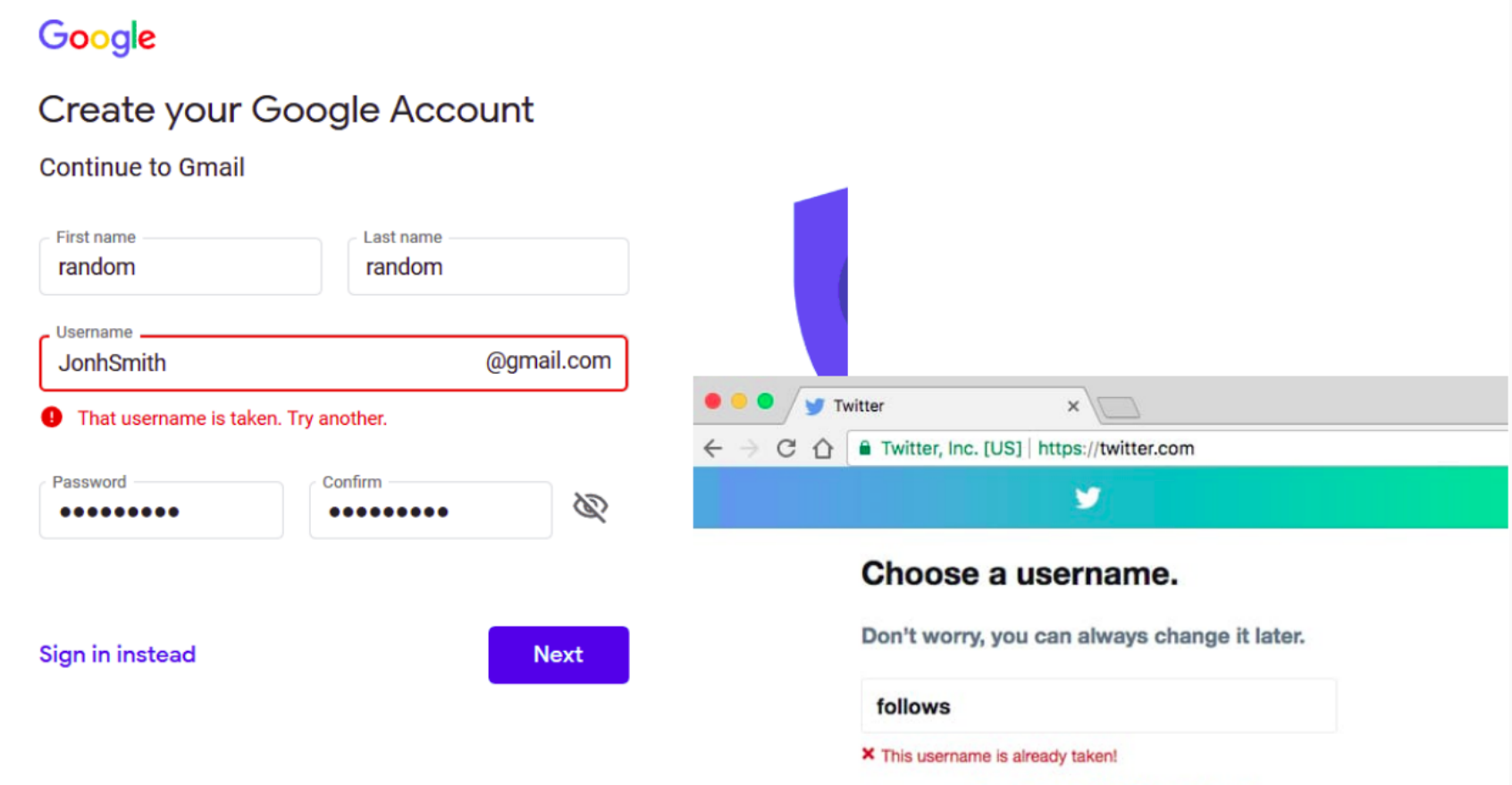
Unidad 4: Algoritmos para flujos de datos

Filtrado

Blanca Vázquez

Abril 2020

Introducción al filtrado



Introducción al filtrado

¿Cómo Google o Twitter checan ’rápidamente’ la disponibilidad de los nombres?
(dentro de los millones de nombres registrados)

Introducción al filtrado

Algunas ideas pueden ser (no necesariamente son las más óptimas):

- Búsqueda lineal!!!
- Búsqueda binaria

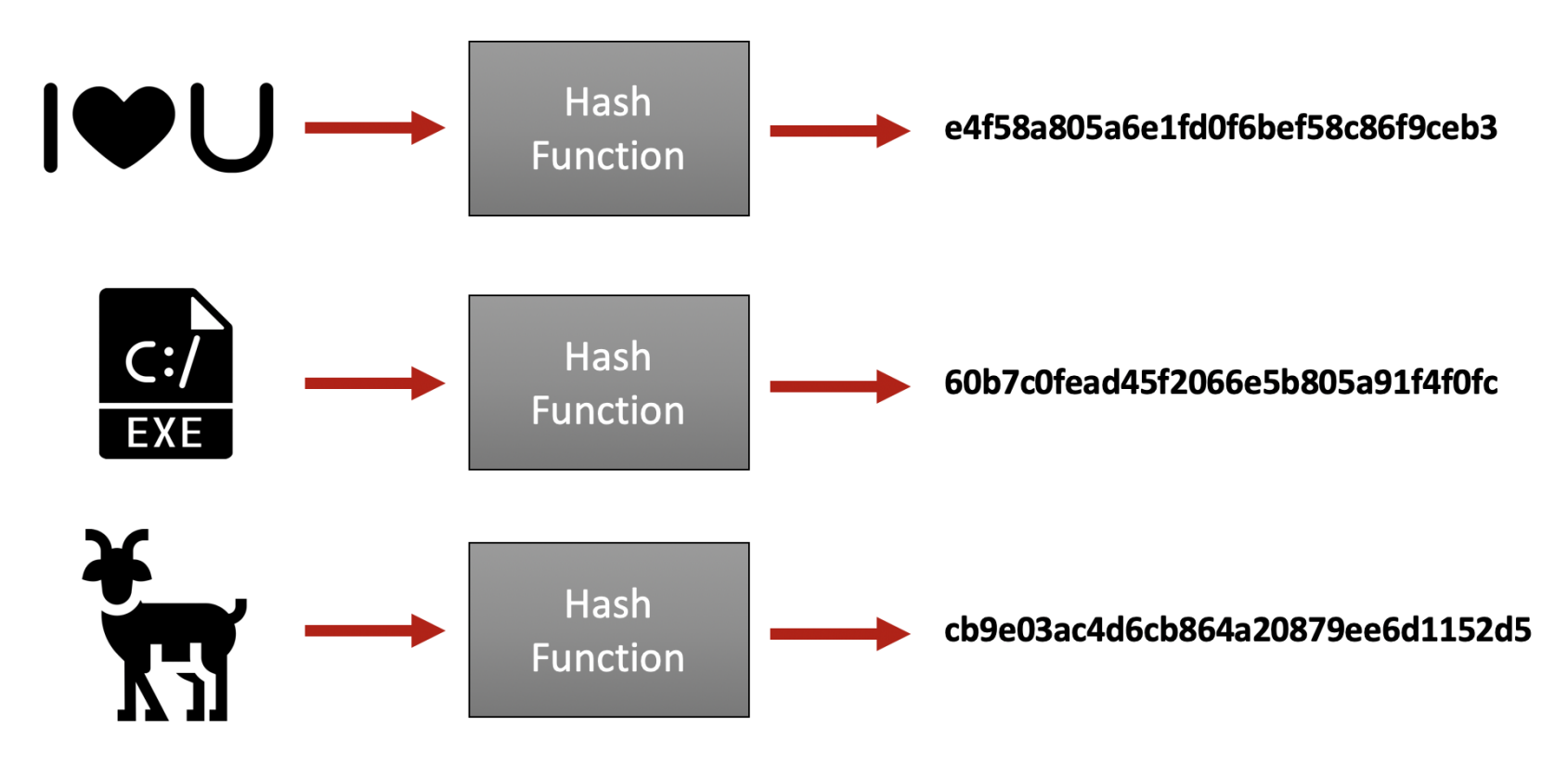
Introducción al filtrado

- Búsqueda binaria
  - Supongamos que almacenamos todos los nombres alfabéticamente y comparamos el nuevo nombre con el nombre que aparece a mitad de la lista
  - Si el nombre coincide, devuelve 'Try again'
  - 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.

Filtrado de Bloom

Una forma más óptima de realizar este proceso es usando el **filtrado de Bloom**.

Antes de entrar a detalles, vamos a repasar la función Hash.

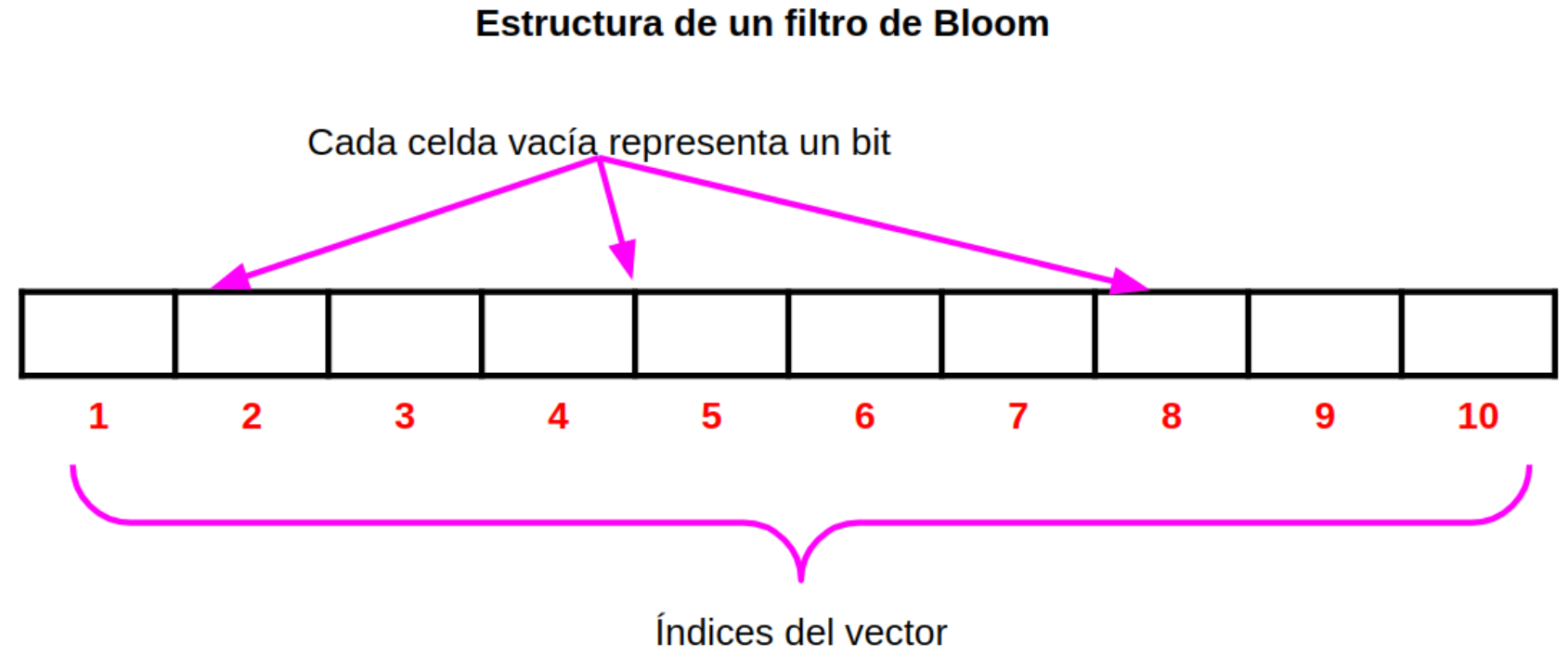


Toma una entrada y genera un identificador único de longitud fija

Filtrado de Bloom

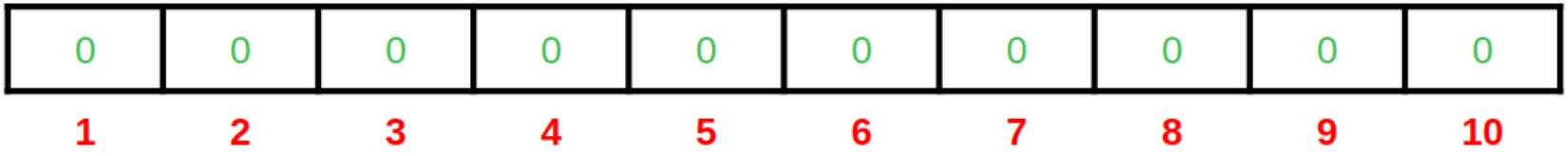
- Es una estructura de **datos probabilística y se emplea para evaluar si un elemento pertenece a un conjunto.**
- Fue desarrollado por Burton Howard Bloom en 1970.
- Ejemplo: verificar si un nuevo *'username'* pertenece a un conjunto. En este caso el conjunto es la lista de todos los nombres existentes*.
- Debilidades: obtención de falso positivos

Estructura


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.



Llenando el vector S

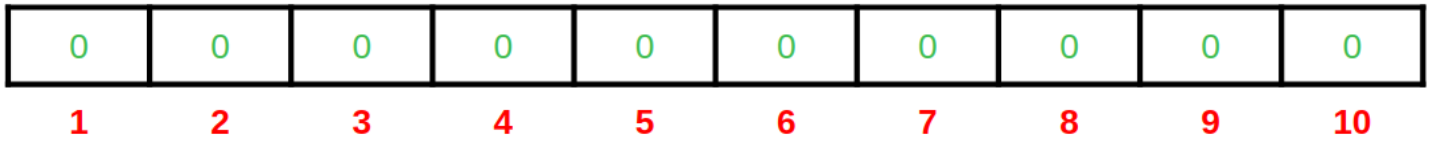
¿Cómo llenamos un filtro de Bloom?

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, que debe cambiarse de 0 a 1.

Definiendo m, n y k

Filtro de Bloom vacío



Antes de empezar debemos definir:

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

→

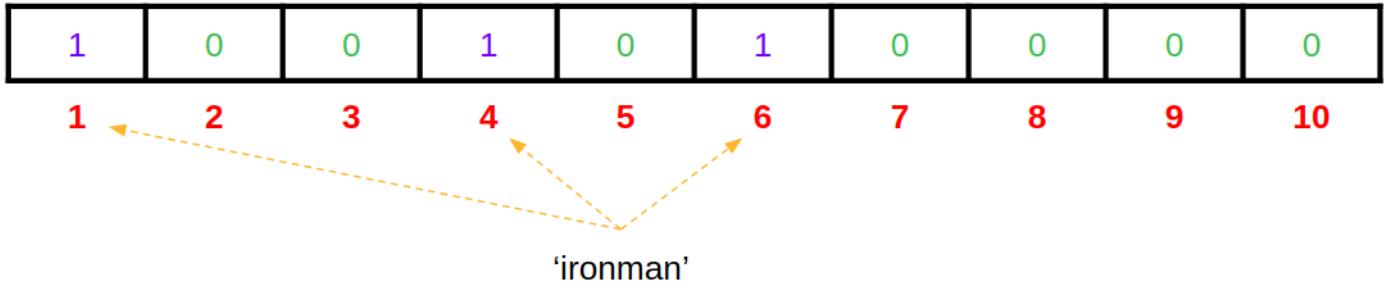
- ***m*** = 10
- ***n*** = 2
- ***k*** = 3

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.

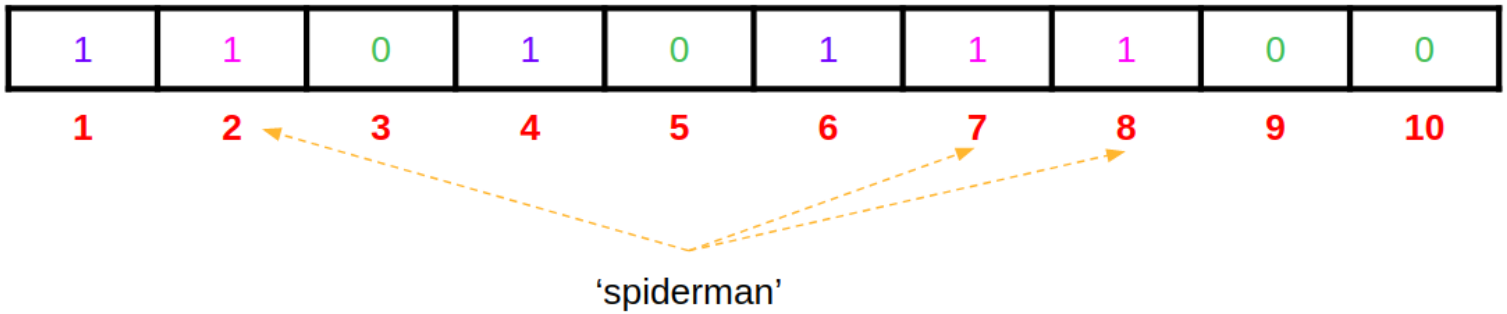


Spiderman!

- 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.

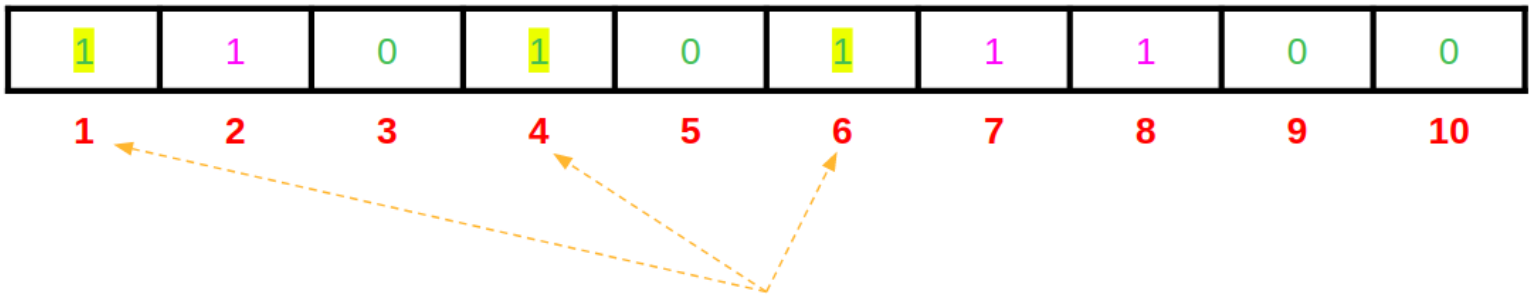


Buscando en el filtro

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
```



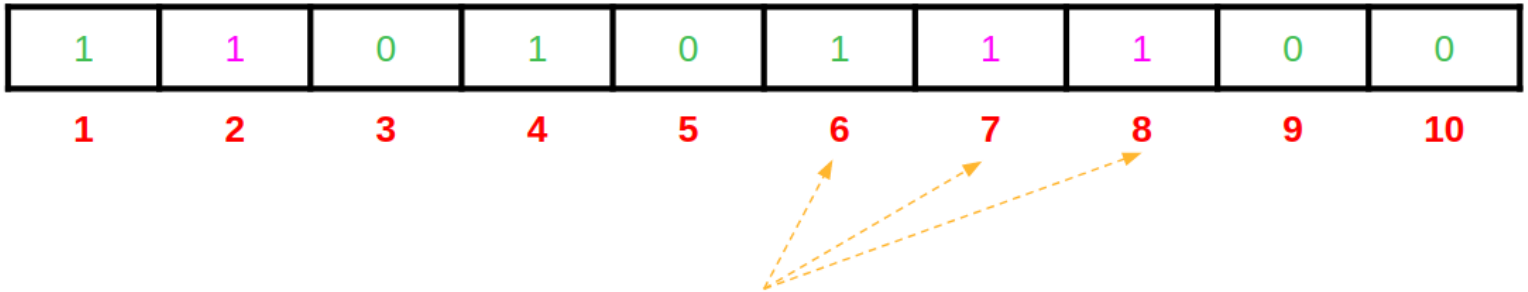
Si todos los índices (arrojados de la función hash) son 1s, entonces decimos que: 'ironman' *probablemente* están presente en S

Thanos!

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
```



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

Falsos positivos

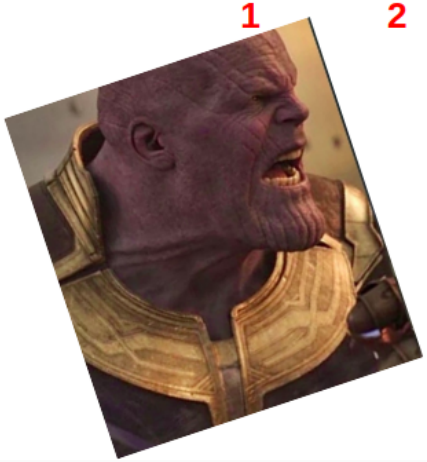
Lo más importante de un filtro de Bloom es buscar un elemento dentro del vector S

- 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
```

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

Falso positivo



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

Falsos positivos

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

- ¿Cómo evitar / reducir los falsos positivos?
  - Más espacio (incrementar el tamaño del vector)
  - Incrementar el número de k (funciones hash)

Reducción de falsos positivos

Cálculo	Fórmula
Probabilidad de falsos positivos	$$P = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k$$ Dónde: m = tamaño del vector k = número de funciones hash n = número de elementos esperados
Tamaño del vector (número de bits)	$$m = -\frac{n \ln P}{(ln2)^2}$$ Si conocemos el número de elementos a añadir (n) y la probabilidad deseada de falsos positivos (p), entonces el número de bits (m) se calcula usando la fórmula de arriba.
Número óptimo de funciones hash	$$k = \frac{m}{n} ln2$$ El número de funciones hash debe ser un número entero positivo.

Funciones Hash

Selección de funciones hash

Las funciones hash usadas en el filtrado de Bloom deben ser:

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

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) (2011)
- **FNV**: Fowler/ Noll/ Vo, es un indexador rápido (1991)
- **Jenkins** o **HashMix** (1997)

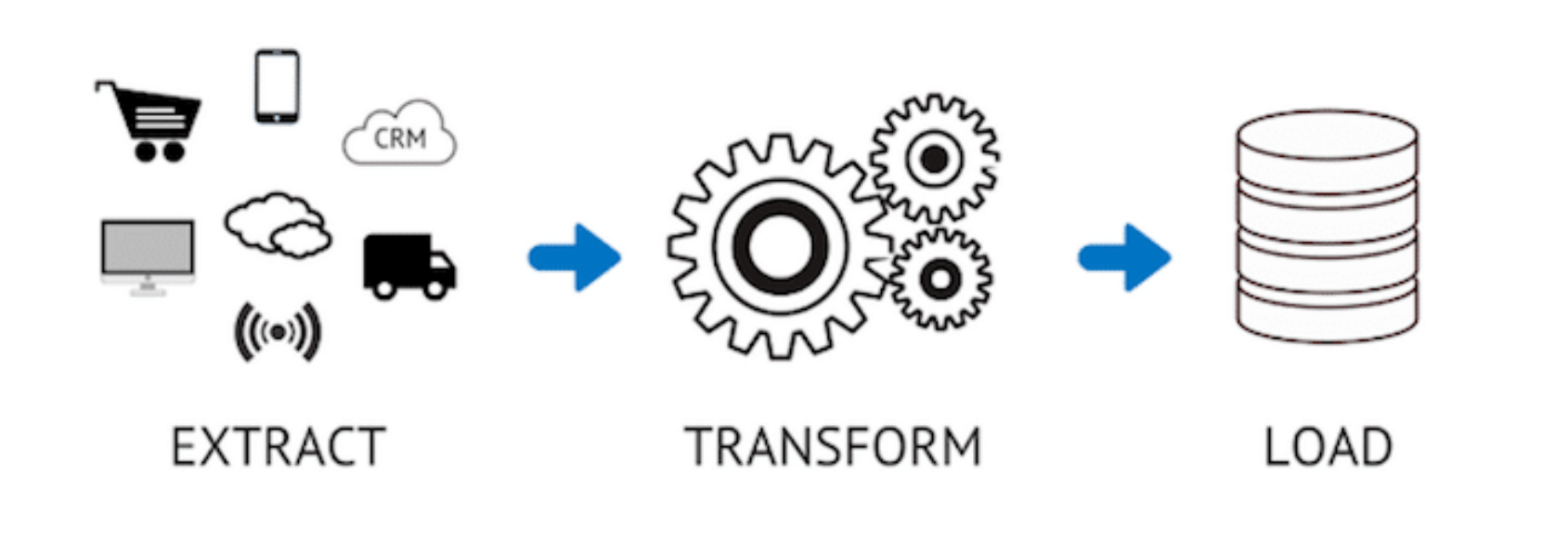
Aplicaciones del filtrado de Bloom

- **Medium.com** 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.

Procesamiento ETL

Extraer - Transformar - Cargar

Introducción ETL



Flujo del proceso ETL

Imagen tomada de Shana Pearlman, 2019

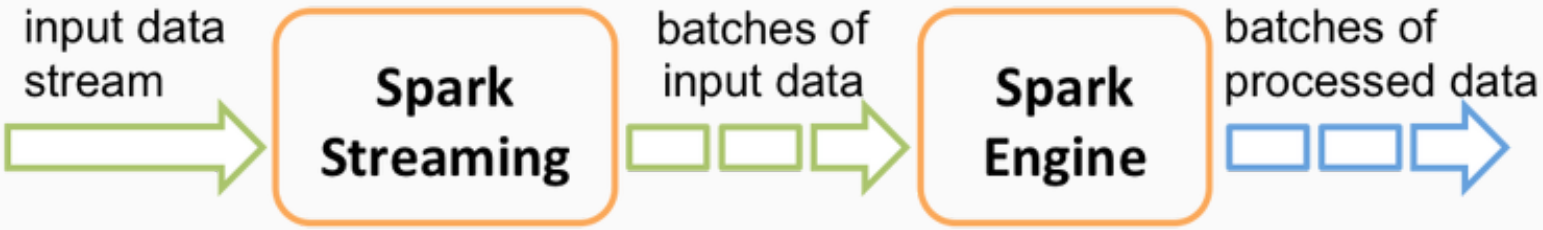
Introducción ETL

Extraer	Transformar	Cargar
Analiza los datos	Pre-procesamiento de los datos	Los datos se envían a su destino
Detección de anomalías	Verificación de los datos	Carga completa
Registro de actividad	Eliminación de datos no útiles	Carga incremental

Spark Streaming

Introducción a Spark Streaming

- Fue añadido en Apache Spark en 2013
- Es un API que permite el procesamiento de datos en streaming de manera escalable, alto rendimiento y tolerancia a fallos.
- Los datos pueden ser cargados desde Kafka, Flume, Kinesis o sockets TCP.



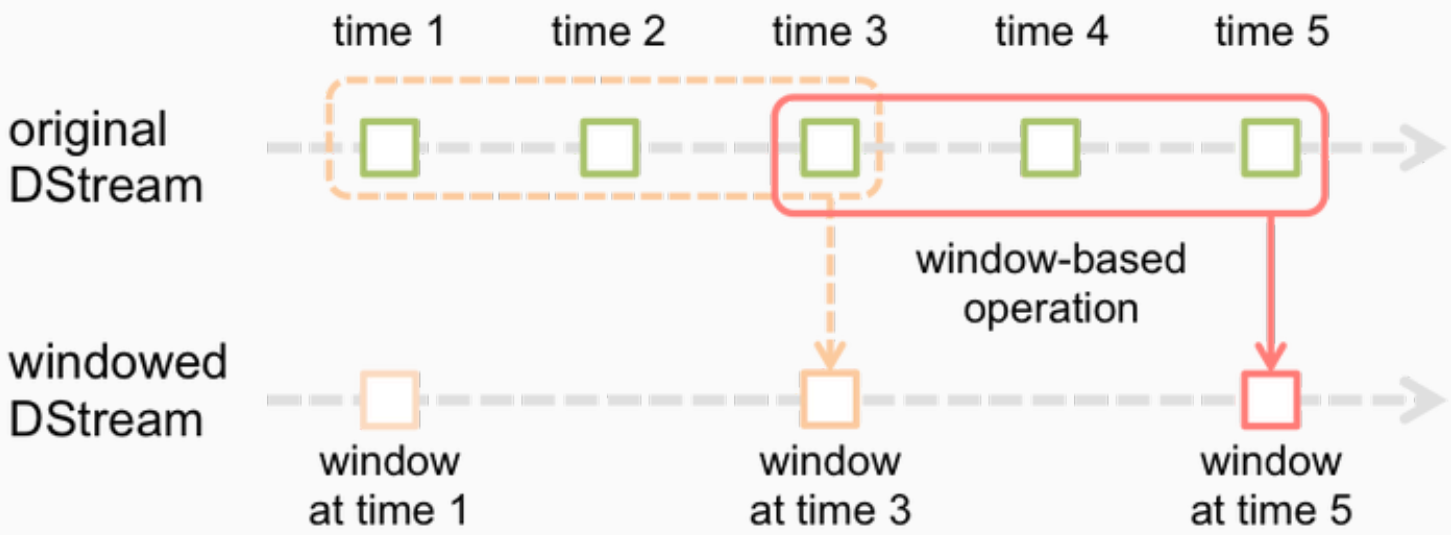
Flujo Spark streaming

Imagen tomada de spark.apache.org

DStream

- Es una abstracción de alto nivel generada por Spark Streaming
- Representan un flujo contínuo de datos
- Internamente cada DStream es una secuencia de RDDs.
- Se puede ejecutar cualquier tipo de operaciones al igual que en los RDDs.

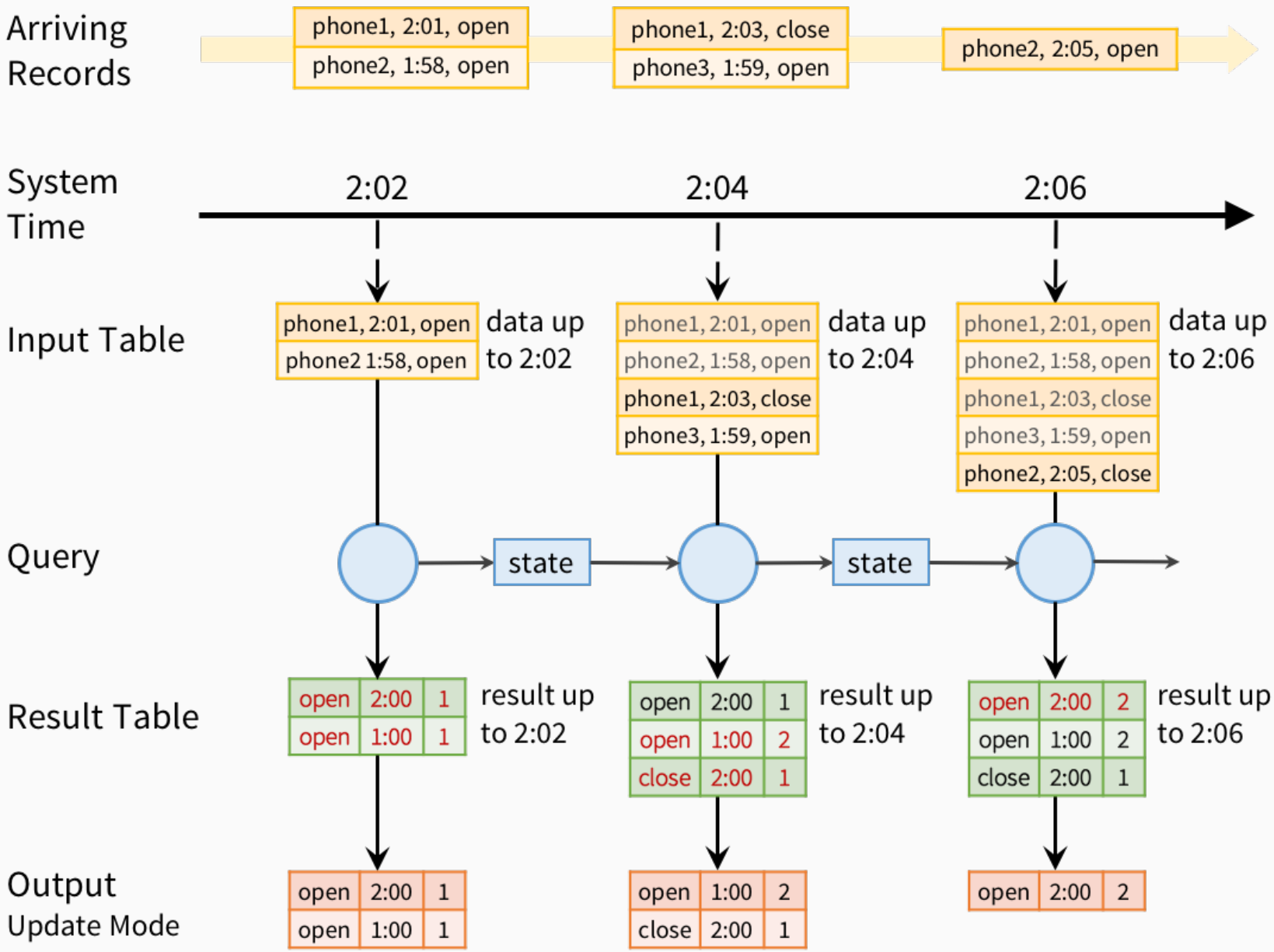
Ventanas en Spark Streaming



Operaciones de ventanas deslizantes en Spark streaming

Imagen tomada de spark.apache.org

Ejemplo


Arriving Records
phone1, 2:01, open
phone2, 1:58, open
phone1, 2:03, close
phone3, 1:59, open
phone2, 2:05, open
System Time
2:02
2:04
2:06
Input Table
phone1, 2:01, open
phone2 1:58, open
data up to 2:02
phone1, 2:01, open
phone2, 1:58, open
phone1, 2:03, close
phone3, 1:59, open
data up to 2:04
phone1, 2:01, open
phone2, 1:58, open
phone1, 2:03, close
phone3, 1:59, open
phone2, 2:05, close
data up to 2:06
Query
state
state
Result Table
open 2:00 1
open 1:00 1
result up to 2:02
open 2:00 1
open 1:00 2
close 2:00 1
result up to 2:04
open 2:00 2
open 1:00 2
close 2:00 1
result up to 2:06
Output
Update Mode
open 2:00 1
open 1:00 1
open 1:00 2
close 2:00 1
open 2:00 2

Referencias

- Bloom Filters
  https://www.geeksforgeeks.org/bloom-filters-introduction-and-python-implementation
- Bloom Filters by Example
  https://llimllib.github.io/bloomfilter-tutorial/
- Spark Structured Streaming: A new high-level API for streaming
  https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html