# Aplicación y Análisis de Estructuras de Datos Probabilísticos.

Junior Micha[1], Brenner Bustillos[2]

***Abstract*—Éste artículo matemático-computacional presentará los resultados obtenidos, mediante un proceso de investigación conjunta, sobre tres estructuras de datos que pueden hacer el trabajo de análisis de datos sea una mejor alternativa ademas de ser muy potente; antes de usar técnicas como un clúster de cómputo para ejecutar herramientas distribuidas de cómputo paralelo como por ejemplo Hadoop y Spark que son herramientas correctas, pero muy costosas.Para entender mejor estas herramientas usaremos dos lenguajes de programación que nos ayudarán a visualizar mejor todo el análisis a realizar.**

## I. INTRODUCCIÓN

Éste artículo describe tres estructuras de datos probabilísticos: Bloom Fitler, Count-min sketch e HyperLogLog. para poder verificar la forma de solución al problema de cada uno de ellas usaremos la programación en lenguaje R y/o Python, como principal herramienta de solución del problema; que es en este caso el conteo y las consultas referidas a elementos dentro de un conjunto que veremos más adelante.

### A. Objetivos

- Aplicar la estructura bloom Fitler para hacer consultas sobre elementos que puedan pertenecer al conjunto analizado.
- Usar la estructura Count-min sketch como tabla de frecuencia de eventos en una secuencia de datos.
- Calcular el número aproximado de elementos en un multiconjunto usando la estructura HyperLogLog.

En esta oportunidad usaremos cada una de las tres estructuras aplicadas sobre un problema real. En el primer (Bloom Filter) generaremos un conjuntos de **n** elementos aleatorios, seguidamente utilizaremos las funciones añadir y query(consultar) para añadir elementos y consultar si algún elemento pertenece al conjunto que hemos generado en el caso de la estructura Count-min sketch, finalmente con la estructura HyperLogLog generaremos un multiconjunto grande el cual calcularemos al cardinalidad de un elemento específico con una excatitud del 2% usando 1,5kb de memoria.

Manuscrito creado en los primeros días de setiembre del 2018; cuya revisión final será el día 9 de Noviembre del 2018. Éste trabajo es compatible en formato IEEE y se distribuye bajo el nombre **Proyecto cm-274** .El manuscrito puede ser encontrado en las cuentas GitHub de los autores

[1] J. Micha es estudiante pregrado de Matemática, Universidad Nacional de Ingeniería, 2016-2022, Lima, Perú. https://github.com/JMicha23

[2] B. Bustillos es estudiante pregrado de Matemática, Universidad Nacional de Ingeniería, 2015-2021, Lima, Perú. https://github.com/brenner-08

## II. RESUMEN

### A. BLOOM FITLER

- Estructura de datos espacio eficiente (Borton Howard Bloom -1970).
- Usado para probar si un elemento es miembro de un conjunto.
- Una consulta arroja ”posiblemente en el conjunto” (falso positivo) o ”definitivamente no en el conjunto” (falso negativo).
- Se puede agregar elementos al conjunto pero no removerlos.
- Mayores elementos con añadidos, mayores probabilidad de falsos positivos .
- Generalmente, menos de 10 bits por elemento son requeridos para un 1% de probabilidad falso positivo, independientemente del tamaño o número de elementos del set.

1) **Descripción de un algoritmo**: Arreglo de **m** bits, todo en 0 . También debe haber **k** funciones hash, cada uno de los cuales mapea algún elemento a una de las m posiciones (distribución aleatoria uniforme).
2) **Para añadir un elemento**: Se alimenta a cada una de las **k** funciones para obtener **k** posiciones en el arreglo , Todas esas posiciones se van a 1.
3) **Para consultar(si es que el elemento está)**: Se alimenta a cada una de las **k** funciones para obtener **k** posiciones en el arreglo , si algunas de las posiciones son 0 el elemento definitivamente no se encuentra en el conjunto, si todos son 1, entonces o bien el elemento está o los bits han sidos puestos a 1 cuando fueron insertados.

### B. COUNT-MIN SKETCH

– Sirve como una tabla de frecuencia de eventos en una data stream . Usa funciones hash para mapear eventos a frecuencias, pero solo usa espacios sub-lineales a costa de ”sobrecontar algunos eventos a causa de colisiones (2003-Graham Cormode, S.Muthu Muthokrishnan).
– Esencialmente es lo mismo que Bloom fitler, pero son usados de manera diferente y se ponen el tamaño de manera diferente.

1) **Estructura**: El objetivo de count-min sketch es la de consumir un stream de eventos, uno a la vez, y contar la frecuencia de los diferentes tipos de eventos en el stream. En cualquier momento el sketch puede ser consultado para la frecuencia de

un particular tipo de evento y regresará un estimado de esta frecuencia dentro de una cierta distancia de la frecuencia real, con una cierta probabilidad.

### C. HYPERLOGLOG

– Algoritmo para el conteo distintivo, aproximando el número de distintos elementos en un multiconjunto. Es capaz de estimar cardinalidades mayor a $10^9$ con una certeza del 2% usando 1,5 kb de memoria.

1) **Algoritmo**: Se basa en la observación de que las cardinalidades de un multiconjunto de números aleatorios uniformemente distribuidos pueden ser calculados estimando el máximo número de ceros en la representación binaria de cada número en el conjunto . Si el máximo número de ceros observados es $n$, un estimado para el número de elementos en el conjunto es $2^n$ .
2) **Operaciones**
   Add: Añade un elenento al conjunto.
   Count: Nos arroja la cardinalidad el conjunto.
   Merge: Para obtener la unión de dos conjuntos.

## III. ESTADO DEL ARTE

- **"Theory and Practice of Bloom Filters for Distributed Systems"**
  Este articulo presenta una serie de técnicas probabilísticas como los Bloom-filters y sus variantes como stable Bloom Filter, Adaptative Bloom Filters, Filter Banks, etc. que se utilizan para reducir el procedimiento de la información y los costos de esta información.

- **"An Improved Data Stream Summary: The Count-Min Sketch and its Applications"**
  Este articulo presenta otras aplicaciones de la estructura de datos Count-min Sketch para problemas como encontrar cuartiles, elementos frecuentes, etc.

- **"HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm"**
  En este articulo se presentan mejoras al algoritmo HyperLogLog reduciendo sus requisitos de memoria y aumentar su precisión para un rango importante de cardinalidades.

## IV. DISEÑO DEL EXPERIMENTO

### A. Implementación del Bloom Filter

1) Crear una estructura de datos Bloom Filter.
2) Generar un arreglo de elementos con la operación Add.
3) Realizar consultas(operación Query) para saber si un elemento en particular se encuentra en nuestro arreglo.

### B. Implementación del Count-min Sketch

1) Crear una estructura de datos Count-min Sketch.
2) Generar un arreglo bidimensional con eventos como elementos.
3) Realizar una consulta puntual (point query) que nos permite saber la frecuencia de un evento en particular.

### C. Implementación del HyperLogLog

1) Crear una estructura de datos Hyperloglog
2) Generación de un multiconjunto(arreglo) aleatorio con números como elementos.
3) Calculamos la cardinalidad del conjunto de números creados.

## REFERENCES


[1] P.Flajolet,E.Fusy,O.Gandouet,F.Meunier,(2007),*HiperLogLog: the analysis of a near-optimal cardinality estimation algorithm*.[Online].Available: http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf.

[2] S.Heule,M.Nunkesser,A.Hall,(2013),*Hyper LogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation. Algorithm*.[Online].Available: https://storage.googleapis.com/pub-tools-public-publication-data/pdf/40671.pdf

[3] G.Cormode,S.Muthukrishnan(2003),*An Improved Data Stream Summary: The Count-Min Sketch and its Applications*.[Online].Available: http://dimacs.rutgers.edu/ graham/pubs/papers/cm-full.pdf.

Bloomfilter

```
library(digest)
library(magrittr)
library(gmp)
library(Rmpfr)

vbits <- function(x,m=1000,k=7){
  vec <- rep(FALSE,m)
  for (i in 1:k) {
    for (j in 1:length(x)) {
      hash <- digest(x[j], algo = "murmur32", serialize = FALSE, seed = i) %>%
        Rmpfr::mpfr(base = 16) %%
        m %>%
        as.integer()

      vec[hash + 1] <- TRUE
    }
  }
  attributes(vec) <- list(m = m, k= k)

  return(vec)
}

vect <- vbits(c("Brenner", "jesus", "lisi", "Brian", "raul","juan", "rober",
                "jose","Maria", "marco", "susana","jordi"), m = 1000, k = 7)

is_vect <- function(x, vbits){
  k <- attr(vbits,"k")
  m <- attr(vbits,"m")

  for (i in 1:k) {
    hash <- digest(x,algo="murmur32",serialize = FALSE, seed=i) %>%
      Rmpfr::mpfr(base = 16)%%
      m %>%
      as.integer()
    if(!vbits[hash + 1]){
      return(FALSE)
    }
  }
  return(TRUE)
}

is_vect("Julio",vect)
is_vect("Bruno",vect)
is_vect("jose",vect)
is_vect("Luis",vect)
is_vect("pepe",vect)
is_vect("juan",vect)
```

```
library(digest)
a=10
conteo <- setRefClass('conteo',
                      fields = list(p = 'numeric', m = 'numeric', alpha = 'numeric', M = 'numeric
                      , methods = list(
                        initialize = function() {
                          .self$p <- 4
                          .self$m <- 2 ^ .self$p
                          .self$alpha <- 0.673
                          .self$M <- rep(0, .self$m)
                          .self$bitArray <- 2 ^ c(0 : (32 - .self$p))
                        }
                        , add = function(value) {
                          x <- as.numeric(paste('0x', digest(value, algo=c('murmur32')), sep=''))

                          j <- x %% .self$m
# equivalent to getting p-1 LSB bits
                         w <- x / .self$m
# equivalent to bitwise right shift of p bits

                          a <- which(.self$bitArray <= w)
                          idx <- a[length(a)]

                          rho <- length(.self$bitArray) - idx

                          if (rho == 0)
                            stop("Overflow error")

                          j <- j + 1 # since R uses indexing starting from 1 we need to offset j
                          .self$M[j] = max(.self$M[j], rho)
                        }
                        , len = function() {
                          .self$alpha * (.self$m ^ 2) / sum(2 ^ (-1 * .self$M))
                        }
                      )


)
input <- sample(0:100,a )
input
```