



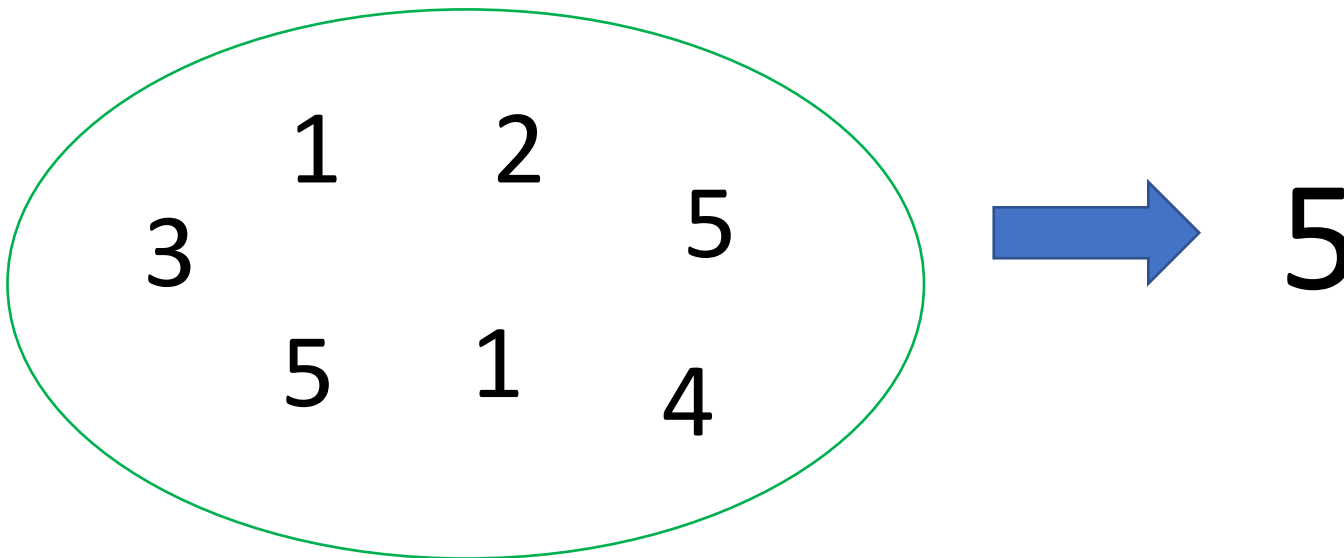
HyperLogLog

- **Integrantes:**
 - Ignacio Bascuñán
 - Benjamín Farías V.
 - Raimundo Martínez



Problema: Cardinalidad (Elementos Distintos)

- Base de datos representada por un conjunto (*multiset*)
- Buscamos la cantidad de elementos **distintos**
- **Sol:** Almacenar cada elemento en alguna estructura eficiente



Problema: Cardinalidad (Elementos Distintos)

- Queremos aplicarlo a bases de datos masivas (ej. redes sociales)
- La solución anterior es **$O(n)$** en espacio, **no es viable**
- Debemos **estimar!**



Algoritmo FM (Flajolet-Martin)

- Almacenar un *bitmap* de tamaño L
- **Hashing uniforme** de cada elemento a un *string* binario de L bits
- Encontrar la posición i del **bit menos significativo que tenga valor 1**
- Marcar la posición i del *bitmap* con **1**
- Repetir por cada elemento

hash(1) = 101

hash(2) = 010

hash(3) = 001

hash(4) = 111

hash(5) = 110




Bitmap = 011



Algoritmo FM (Flajolet-Martin)

- Dada R , la posición menos significativa del *bitmap* que **tenga valor 0**
- Se estima la cardinalidad como: $2^R / \phi$
- Factor de corrección de sesgo: $\phi \approx 0.77351$
- **Limitación:** Tiene una **alta varianza!**

Bitmap = 011  $E = 4 / 0.77 \approx 5.2$



LogLog

- Idea similar pero **agrupando** los valores de hash en **registros**
- Los primeros x bits indican el registro, y entre los restantes se busca el **primer bit 1**
- Cada registro almacena la **posición máxima** encontrada para este bit

hash(1) = 0101

hash(2) = 1010

hash(3) = 0001

hash(4) = 1001

hash(5) = 1101



reg 00 = 1

reg 01 = 1

reg 10 = 1

reg 11 = 1



LogLog

- Se define R como el promedio entre los **m registros**: $R = \frac{1}{m} \sum Reg$
- Estimación de cardinalidad: $E = \alpha_m * m * 2^R$
- El factor de corrección de sesgo α_m depende de la cantidad **m** de registros
- El error estándar es bajo: $stderr \approx \frac{1.3}{\sqrt{m}}$
- **Mejora:** El promedio sólo considera el 70% menor: $stderr \approx \frac{1.05}{\sqrt{m}}$

reg 00 = 1

reg 01 = 1

reg 10 = 1

reg 11 = 1



$$E = 0.63 * 4 * 2^{\frac{3}{2}} \approx 5$$



HyperLogLog

- Igual a *LogLog* pero cambiando el **promedio** entre registros por la **media armónica**
- Media armónica aplicada: $Z = (\sum_{j=1}^m 2^{-M[j]})^{-1}$
- Estimación de cardinalidad: $E = \alpha_m * m^2 * Z$
- El error estándar es aún mejor, ya que los *outliers* afectan menos: $stderr \approx \frac{1.04}{\sqrt{m}}$

reg 00 = 1

reg 01 = 1

reg 10 = 1

reg 11 = 1



$$E = 0.63 * 16 * 0.5 \approx 5$$



HyperLogLog

- Para cardinalidades pequeñas, la estimación de *HLL* presenta un **sesgo**
- Técnica alternativa para **cardinalidades bajas**, denominada *Linear Counting*
- *Linear Counting*, con **V** la cantidad de registros **con valor 0**:

$$E^* = m * \log(m/V)$$

- Si no hay registros con valor 0, se utiliza la estimación común de *HLL*
- Para **grandes cardinalidades** también existe una aproximación:

$$E^* = -2^{32} * \log(1 - \frac{E}{2^{32}})$$



HyperLogLog

- Error estándar: $stderr \approx \frac{1.04}{\sqrt{m}}$
- Ocupa espacio en **orden logarítmico** sobre los datos, muy eficiente en memoria!
- Operaciones de conteo y almacenamiento toman **O(1)** con registros fijos



HyperLogLog: Aplicaciones

- *Reddit*: Conteo de vistas totales de una publicación
- *BigQuery*: Conteo de elementos únicos en una base de datos masiva
- Análisis sobre *Big Data* (*Redis*, *Amazon Redshift*, *Facebook Presto*, *Apache Druid*)



Google BigQuery



redis

presto



Experimentos

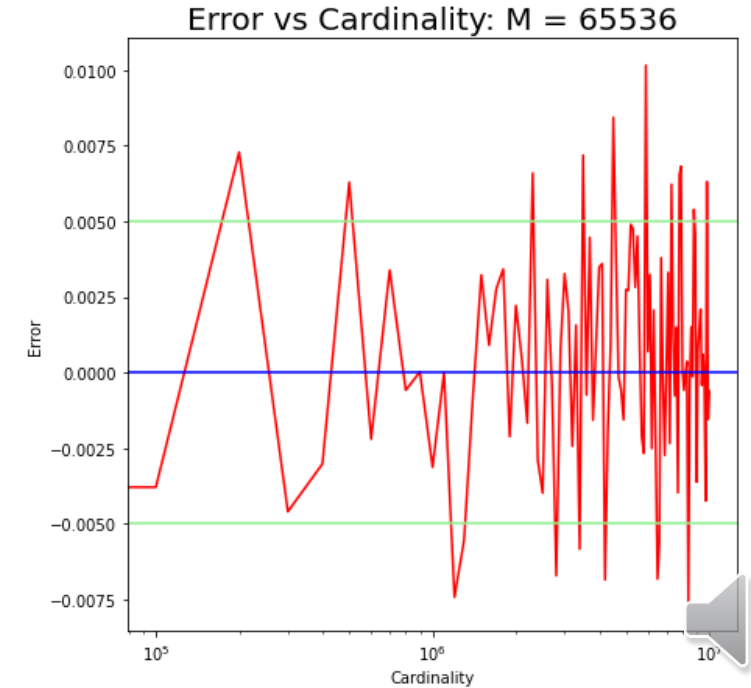
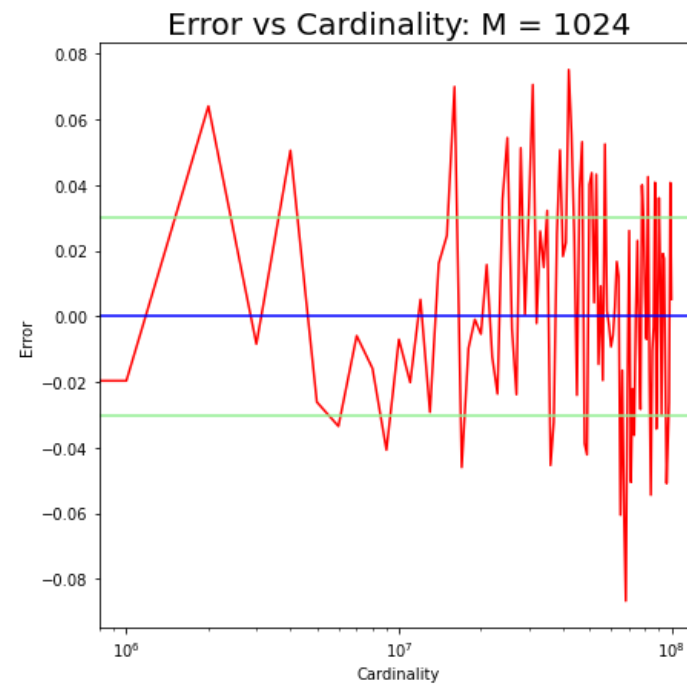
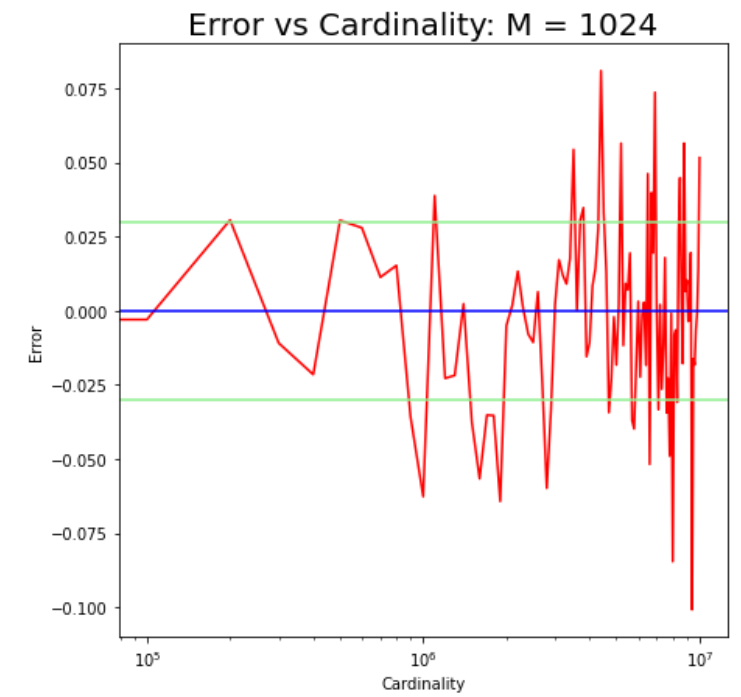
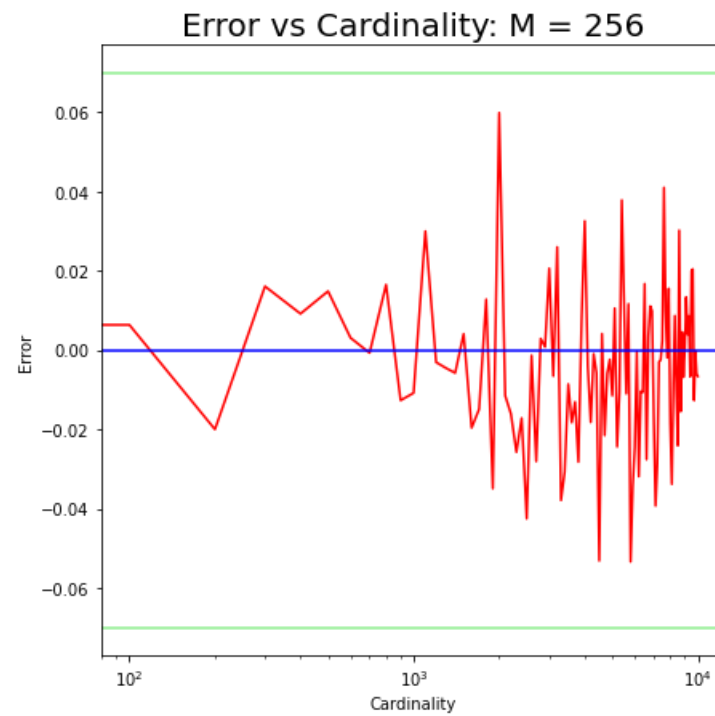
Implementación
en C

M es la cantidad
de registros del
HLL

N es la
cardinalidad del
set de datos

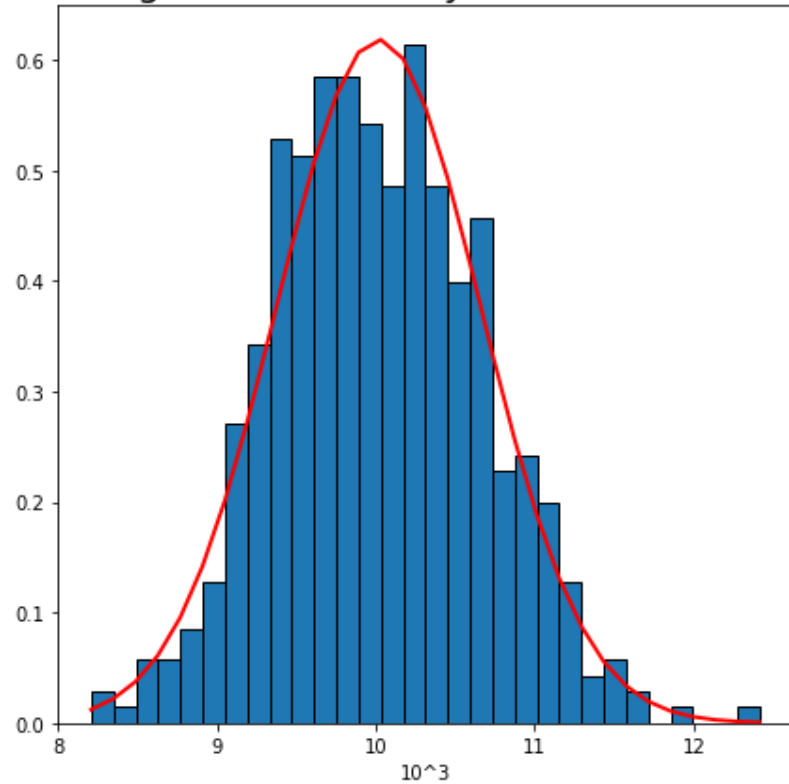


Experimentos

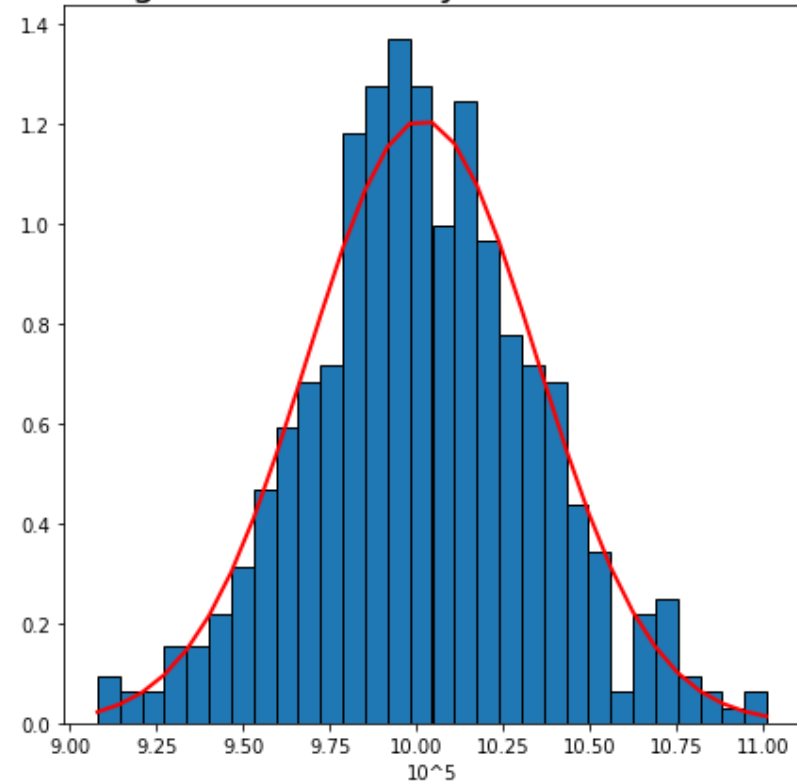


Experimentos

Histogram: Cardinality = 10^4 , $M = 256$



Histogram: Cardinality = 10^6 , $M = 1024$



Mejora: HyperLogLog++

- Propuesto en el año 2013 en el paper "*HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm*"
- Mejoras respecto al **uso de memoria y precisión** para ciertos rangos de cardinalidades:
 - **1.** Uso de una función de hash de 64 bits
 - **2.** Corrección de sesgo para cardinalidades bajas
 - **3.** Representación *sparse* de registros



1. Uso de una Función de Hash de 64 Bits

- *HyperLogLog* original: hash de 32 bits
- Colisiones más probables con cardinalidades cercanas a 2^{32}
- Bajo impacto en uso de memoria: depende de la posición del primer bit 1 y del número de registros

- Máxima posición del bit 1: $L + 1 - p$

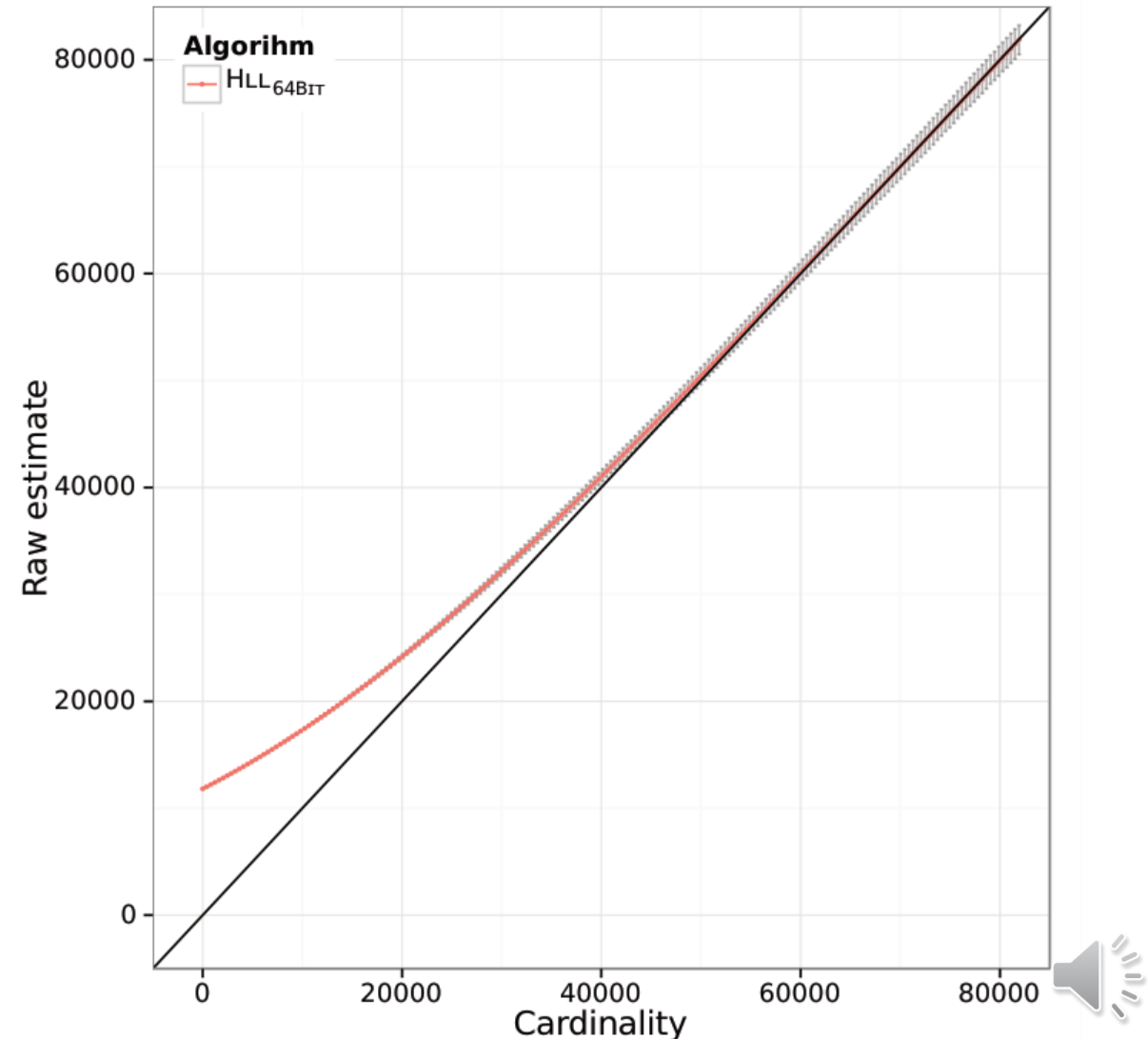
- Bits necesarios: $\lceil \log_2(L + 1 - p) \rceil \cdot 2^p$

- Hash de 32 bits: $5 \cdot 2^p$ Hash de 64 bits: $6 \cdot 2^p$



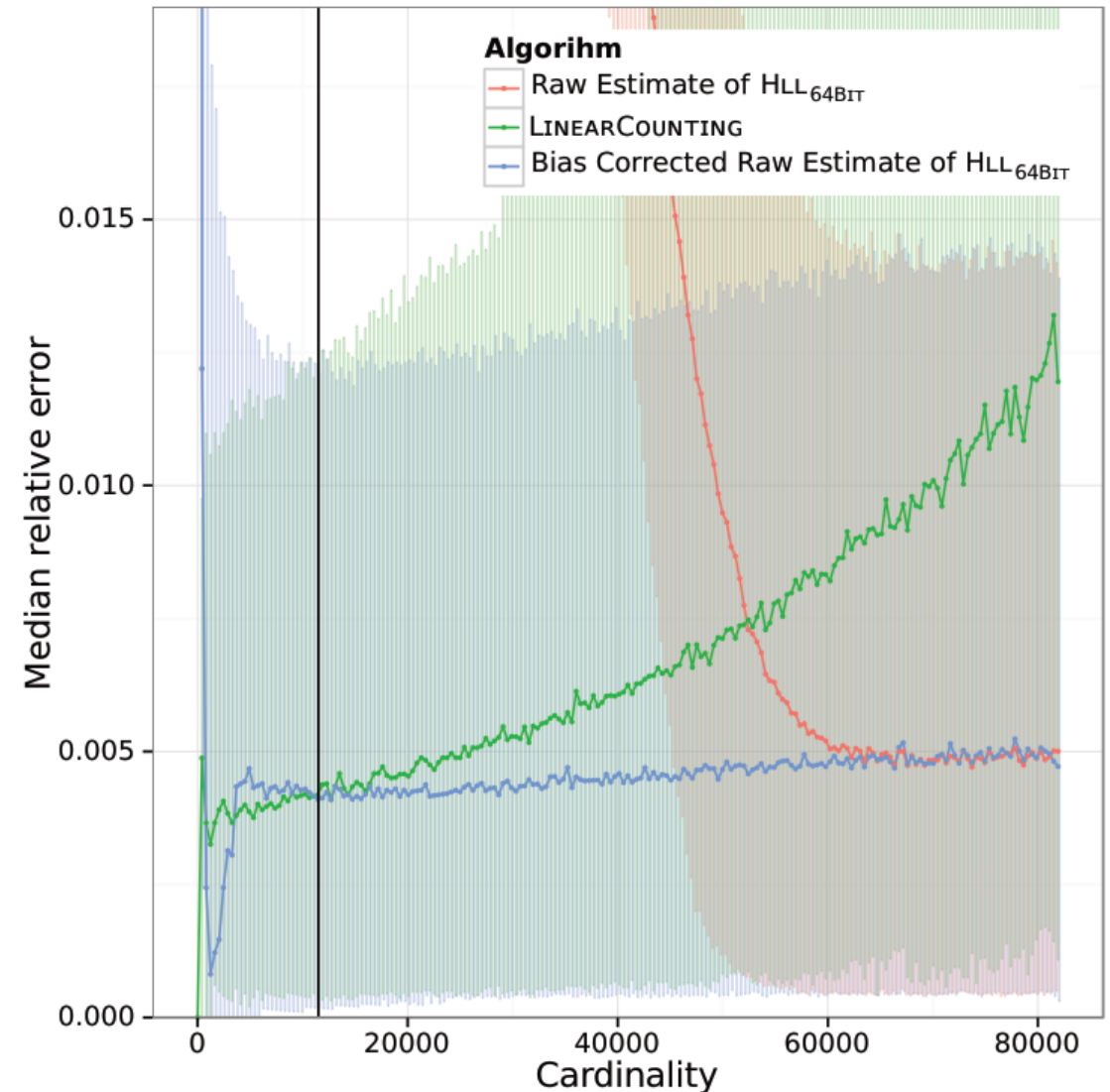
2. Corrección de Sesgo para Cardinalidades Bajas

- *HyperLogLog* sobreestima la cardinalidad si ésta es baja
 - Cardinalidad = 0 \rightarrow E = 0.7m
- *HyperLogLog* usa *LinearCounting* para mejorar estos resultados, pero el error persiste para un rango importante de cardinalidades



2. Corrección de Sesgo para Cardinalidades Bajas

- **Sol:** Corrección empírica con 200 sesgos precalculados e interpolación con *KNN*
- Dependiendo de la estimación se decide entre esta solución o *LinearCounting*
- Reducción de error para un rango considerable de cardinalidades



3. Representación *Sparse* de Registros

- *HyperLogLog* usa una cantidad de memoria constante para los registros, independiente de la cardinalidad
- Con cardinalidades pequeñas, muchos registros **no son usados**
- **Sol:** Guardar pares (índice de registro, posición del primer bit 1) como un entero en una lista ordenada
- Además, se mantiene un set con nuevos enteros en base a nuevos datos
- **Merge** entre el set y la lista para actualizar esta última



3. Representación *Sparse* de Registros

- Representación *sparse* con mayor precisión: $p' > p$, $(idx, pos1) \rightarrow (idx', pos1')$
- Regreso a precisión original si el uso de memoria crece demasiado ($> 6m$ bits):
 - idx' contiene los p' bits más significativos
 - Como $p' > p$, para obtener idx se obtienen los p bits más significativos de idx'
 - Para obtener $pos1$ se revisan los bits $63 - p$ a $64 - p'$ contenidos en idx' :
 - Si alguno de esos bits es 1, podemos obtener $pos1$ a partir de idx'
 - Si todos son 0, entonces $pos1 = pos1' + (p' - p)$

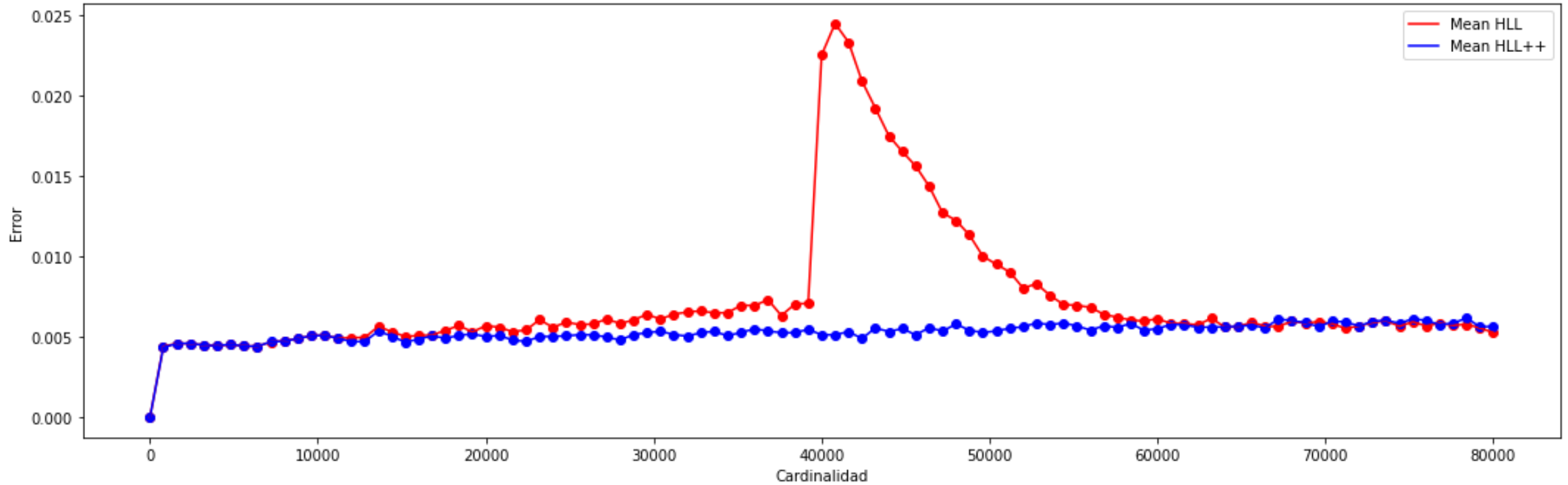


Mejoras Adicionales de Memoria

- Codificación de largo variable de enteros, en vez de usar un número fijo de bits
- Codificación de diferencias entre elementos de la lista:
 - **Lista Original:** a_1, a_2, a_3, \dots
 - **Lista con Codificación de Diferencias:** $a_1, a_2 - a_1, a_3 - a_2, \dots$



Experimentos: HyperLogLog++



Conclusiones

- El algoritmo *HyperLogLog* permite estimar la cardinalidad de forma muy precisa para grandes cantidades de datos
- Ocupa muy poco espacio en su estructura de datos, siendo efectivo en la práctica
- Es un algoritmo flexible, permitiendo agregar constantes mejoras a partir de la versión inicial, con el fin de aumentar su precisión y eficiencia



Bibliografía y Referencias

- [1] Flajolet, P., Fusy, É., Gandouet, O., Meunier, F. (2007). *HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm.*
- [2] Flajolet, P., Martin G. N. (1985). *Probabilistic Counting Algorithms for Data Base Applications.*
- [3] Durand, M., Flajolet, P. (2003). *LogLog Counting of Large Cardinalities.*
- [4] Hall, A., Heule, S., Nunkesser, M. (2013). *HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm.*
- [5] Dial, T. (2022). C/C++ Implementation of the HyperLogLog++ cardinality estimation algorithm [Source Code]. <http://dialtr.github.io/libcount/>