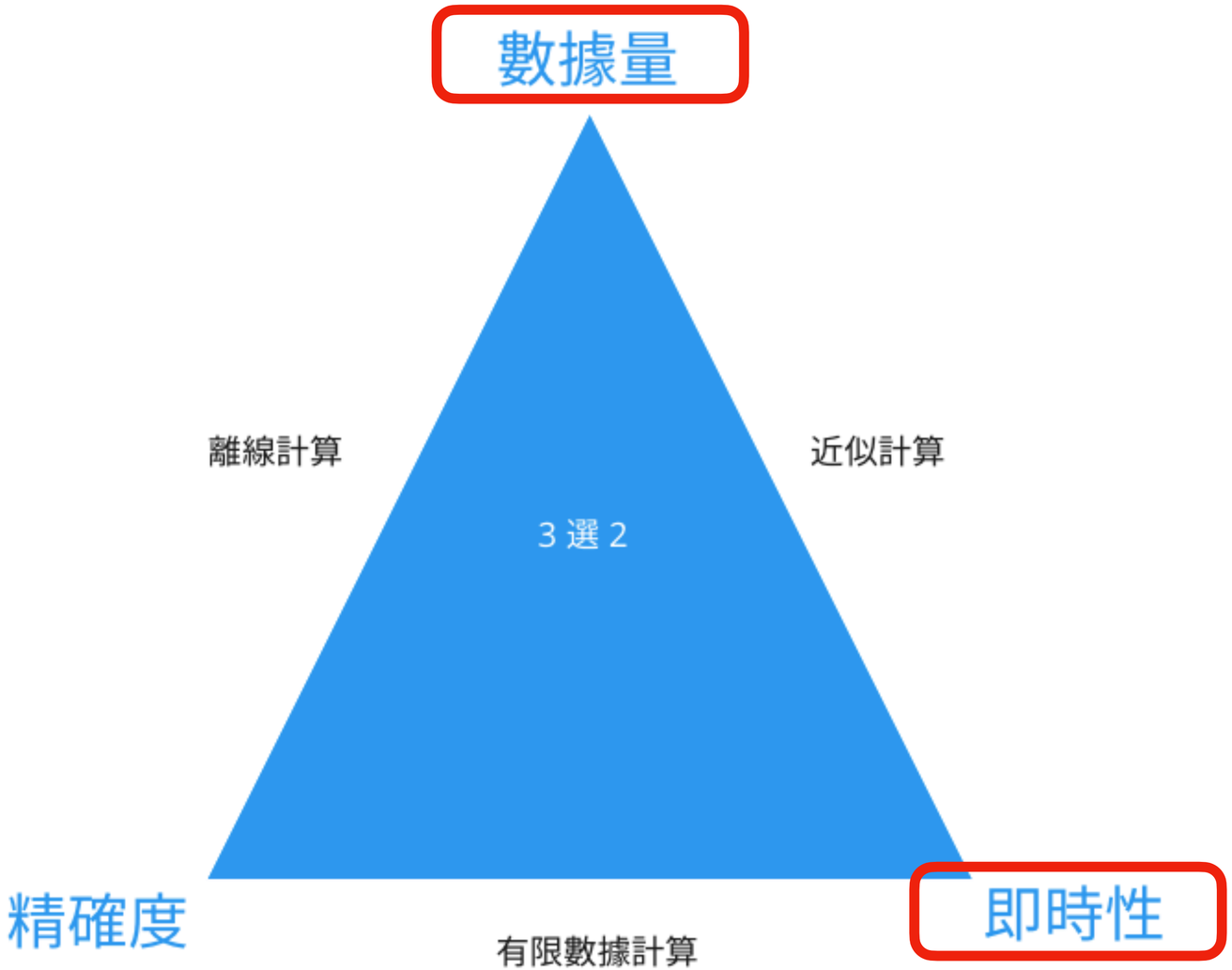
Probabilistic Data Structure

Data Structure For Big Data

Yuanchieh 2023/02/21

Why We Need Probabilistic DS - Trade Off



ref: https://cloud.tencent.com/developer/article/1815080

Agenda

- Topic 1: Membership
- Topic 2: Cardinality
- Demo
- Summary & Reference

Topic 1: Membership

- Akamai 在 2015 年的文章中提到，觀測兩天的線上流量，有 74% 的 Cache 只被 Access 一次，此情況稱為 “One Hit Wonders”

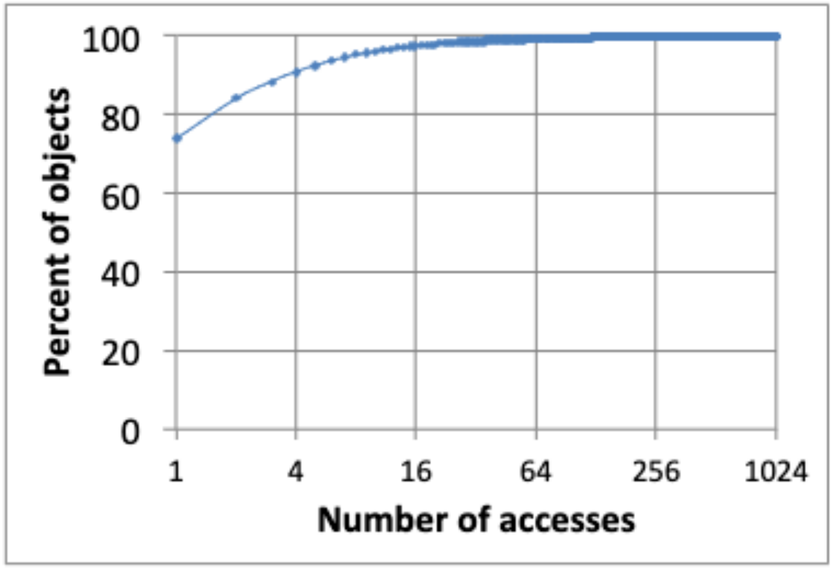


Figure 5: On a typical CDN server cluster serving web traffic over two days, 74% of the roughly 400 million objects in cache were accessed only once and 90% were accessed less than four times.

ref: https://www.sigcomm.org/sites/default/files/ccr/papers/2015/July/0000000-0000009.pdf

Topic 1: Membership

- 具體需求：「輸入網址，幫我存起來，之後會詢問某網址是否有出現過」
- 系統背景：
  - 網址長度假設 200 Bytes
  - 每日網址不重複數量約 400m 筆

Topic 1: Membership - Intuition

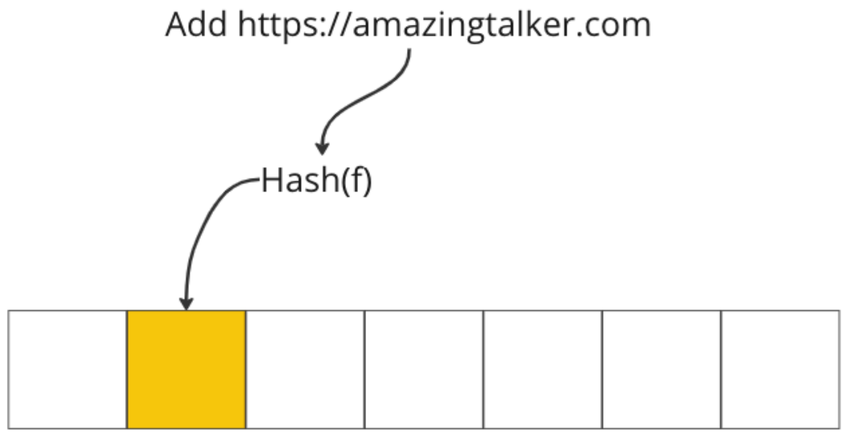
- Maybe HashMap ?
- {
  “https://amazingtalker.com”: 1,
  “https://tw.amazingtalker.com/?gclid=Cj0KCQiAi8KfBhCuARIsADp-A55tk4RbXTHBnV9ydle0ikHP9-m82-gypkrE0Io0GUFs5czVDmh7pS0aAh3PEALw_wcB”: 1,
  …..
  }
- Space: 200 Byte * 400 m = 80 GB!

Bloom Filter

- 原本的網址太長，那就用 Hash 轉換成 int，儲存到對應的 bit array 中
- 利用 Hash same input same output 特性，下次查找到對應的 bit array 位置
- 在 bit array 足夠大以及 Hash 產生的 value 足夠分散的情況，可以有足夠的正確性

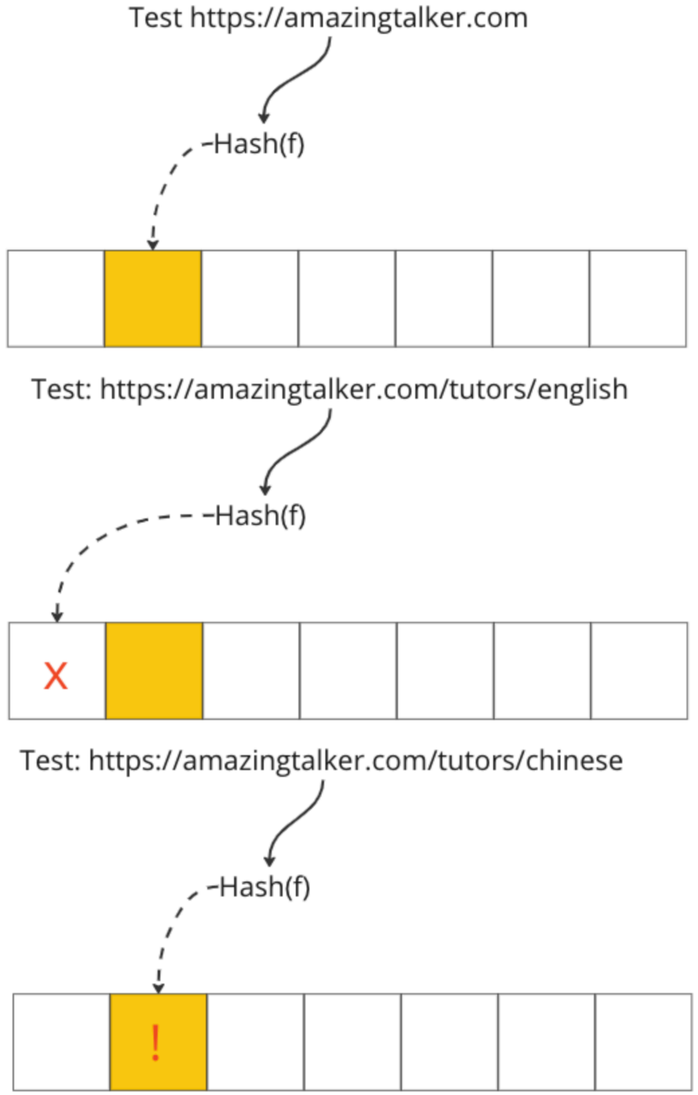
Bloom Filter - Adding

- Adding: 將 element 透過 Hash function 轉換成 int，把對應的 bit array index 設為 true



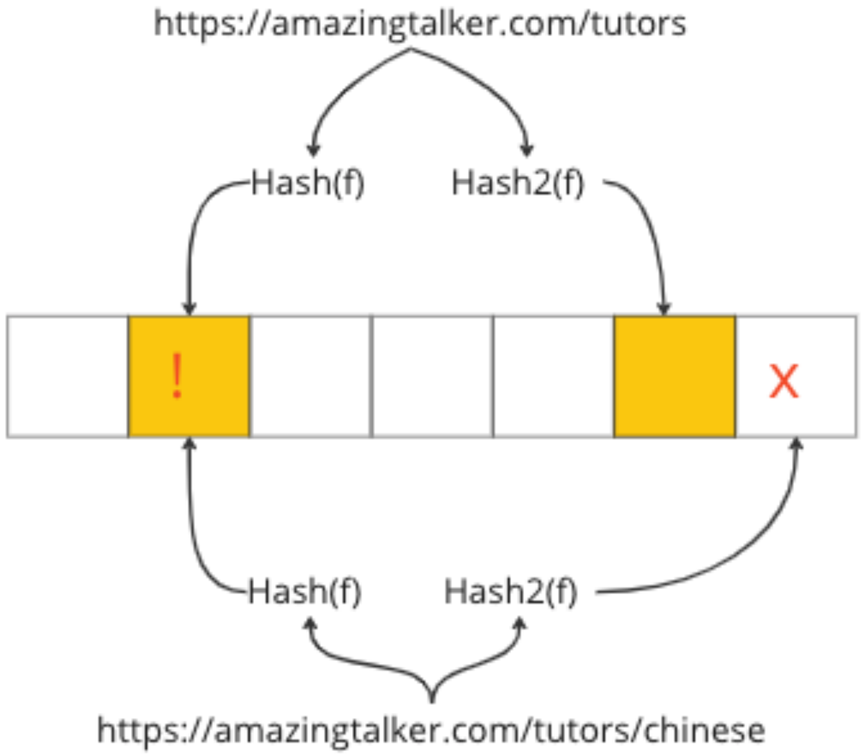
Bloom Filter - Testing

- Testing: 檢查 element 是否出現過
  - True => “可能”出現過 (False Positive)
  - False => 一定沒有出現過



Bloom Filter - Optimize

- Use K hash
- Use larger bit array



Bloom Filter - Algorithm

Algorithm 2.1: Adding element to the Bloom filter

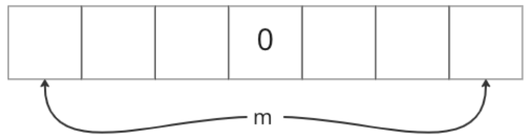
```
Input: Element x ∈ 𝔻
Input: Bloom filter with k hash functions {h_i}_{i=1}^k
for i ← 1 to k do
    j ← h_i(x)
    BloomFilter[j] ← 1
```

Algorithm 2.2: Testing element in the Bloom filter

```
Input: Element x ∈ 𝔻
Input: Bloom filter with k hash functions {h_i}_{i=1}^k
Output: False if element not found and True if element may exist
for i ← 1 to k do
    j ← h_i(x)
    if BloomFilter[j] ≠ 1 then
        return False
return True
```

Bloom Filter - False Positive Rate

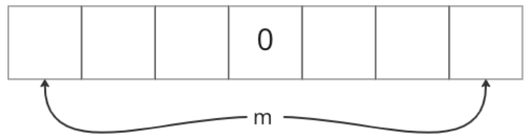
- 加入一個 element 後 bit array 中某一個 bit 還是 0 的機率



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

Bloom Filter - False Positive Rate

- 加入一個 element 後 bit array 中某一個 bit 還是 0 的機率



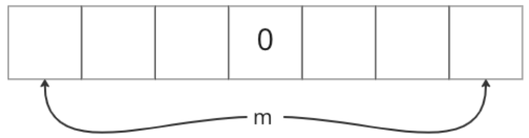
$$\left(1-\frac{1}{m}\right)^{k}$$

- 經過 n 輪後這個 bit 還是 0 的機率

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

Bloom Filter - False Positive Rate

- 加入一個 element 後 bit array 中某一個 bit 還是 0 的機率



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

- 經過 n 輪後這個 bit 還是 0 的機率

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

- 第 n+1輪在發生 False Positive 的機會，也就是該 bit 之前曾經被設為 1 ，機率剛好是排除 “保持為 0 的機率”

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

Bloom Filter - Result

- 計算所需的空間

n

Number of items in the filter (optionally with SI units: k, M, G, T, P, E, Z, Y)

4e8

p

Probability of false positives, fraction between 0 and 1 or a number indicating 1-in-p

1e-2

m

Number of bits in the filter (or a size with KB, KiB, MB, Mb, GiB, etc)

k

Number of hash functions

4 Submit

n = 400,000,000
p = 0.01 (1 in 100)
m = 4,209,081,847 (501.76MiB)
k = 4

80 GB => 500 MB!

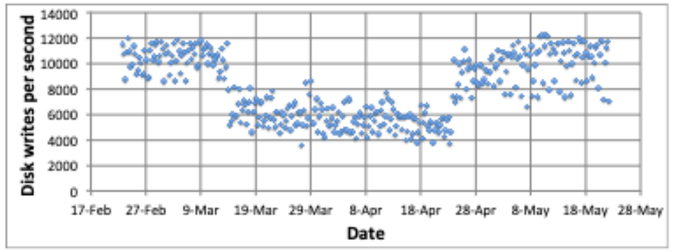
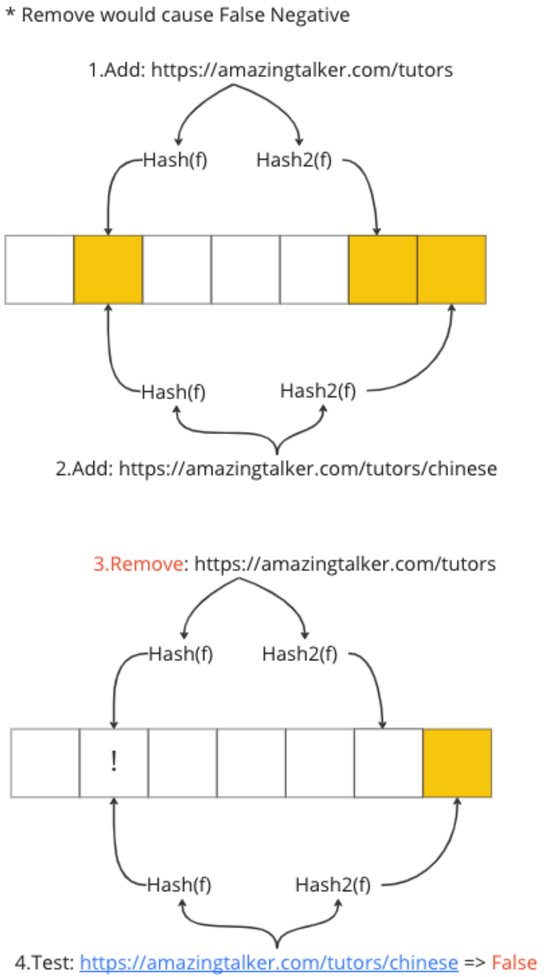


Figure 7: Turning on cache filtering decreases the rate of disk writes by nearly one half because objects accessed only once are not written to disk.

```
n = ceil(m / (-k / log(1 - exp(log(p) / k))))
p = pow(1 - exp(-k / (m / n)), k)
m = ceil((n * log(p)) / log(1 / pow(2, log(2))));
k = round((m / n) * log(2));
```

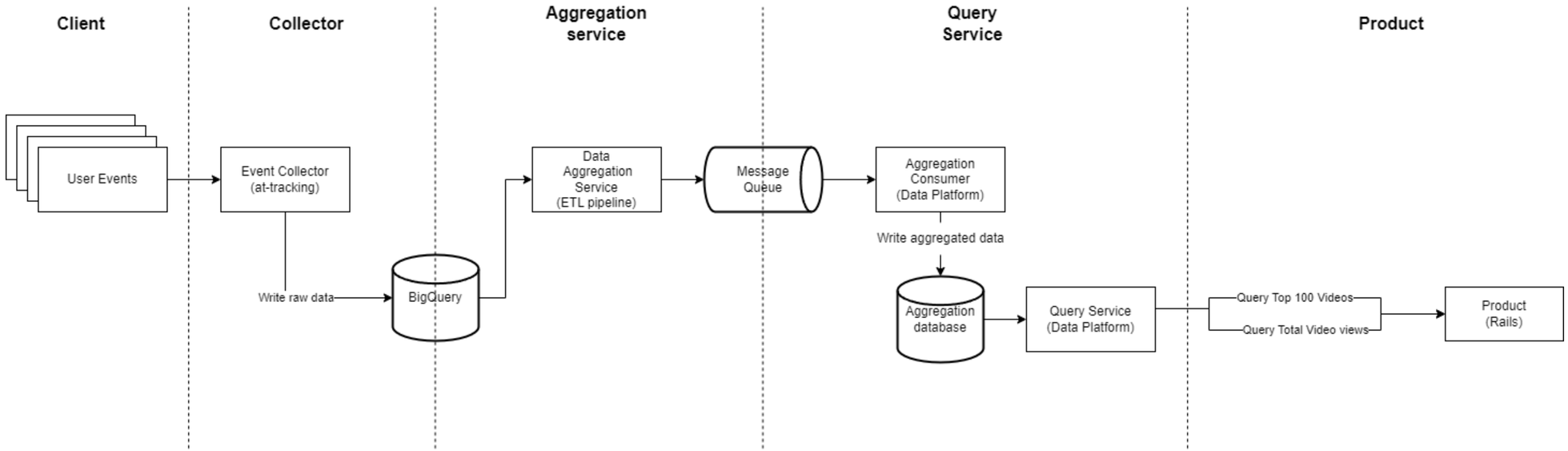
Bloom Filter - Limitation

- 加入不能刪除，否則會出現 False Negative
  - Counting Bloom Filter
- 如果 input dataset 很大無法塞進記憶體時，效能會大幅度衰減
- bit array 無法 resize
  - resize 後需要把所有 element 重新 hash 一次找到對應的位置
- Other Solution => Quotient Filter / Cuckoo Filter



Topic 2: Cardinality

- 短影片功能，PM 想知道每支影片的不重複瀏覽人數，目前約[redacted]影片，每部影片最高不[redacted]



Topic 2: Cardinality

- [redacted]
- [redacted] 500k 影片，每支影片不重複觀看人數暴增到 1m，想要 real time 看到不重複觀看人數的統計

Topic 2: Cardinality - Intuition

- HashMap ?
- {
  “video1”: {
  “user_1”: 1,
  “user_2”: 1,
  .....
  }
  .....
  }

Linear Counting

- 參考 Bloom Filter 作法，把 element hash 後寫入 bit array 對應位置
- 從 "某個 bit 經過 n 輪後還是 0" 的機率，數學推導等於 (陣列中為 0 的比率)，

$$\overline{\mathrm{V}} = \left(1 - \frac{1}{m}\right)^n \approx e^{-m \cdot n}. \qquad (3.1)$$

- 推導出 n (不重複個數) 的公式

$$n \approx -m \cdot \ln \mathrm{V}. \qquad (3.2)$$

Algorithm 3.2: Estimating cardinality with Linear Counting

```
Input: Dataset 𝔻
Output: Cardinality estimation
LinearCounter[i] ← 0, i = 0 ... m − 1
for x ∈ 𝔻 do
    Add(x, LinearCounter)
Z ← count_{i=0...m−1} (LinearCounter[i] = 0)
return ⌊−m · ln Z/m⌋
```

Linear Counting

- 特性
  - 適合小的資料集
  - 資料儲存空間跟輸入範圍成線性關係 (Linear)

LogLog 與他的家族

- LogLog
- SuperLogLog
- HyperLogLog
- HyperLogLog++

LogLog - 概念

- 玩一個擲公正硬幣的遊戲
  - 如果是 H => 繼續
  - 如果是 T => 結束
- 今天出現 “HHHT” 的機率是 1/16
  - 換句話說 “平均擲 16 次出現 1 次”
- simulate: https://gist.github.com/sj82516/41f2ef0cd235089993502c86b65d142b

LogLog - 概念

- 玩一個擲公正硬幣的遊戲
  - 如果是 H => 繼續
  - 如果是 T => 結束
- 今天出現 “HHHT” 的機率是 1/16
  - 換句話說 “平均擲 16 次出現 1 次”
  - 在換句話說 “出現 HHHT 前會出現 15 個不同的結果”
  - 公式 ： $2^{(\text{leading H count}+1)}$
- simulate: https://gist.github.com/sj82516/41f2ef0cd235089993502c86b65d142b

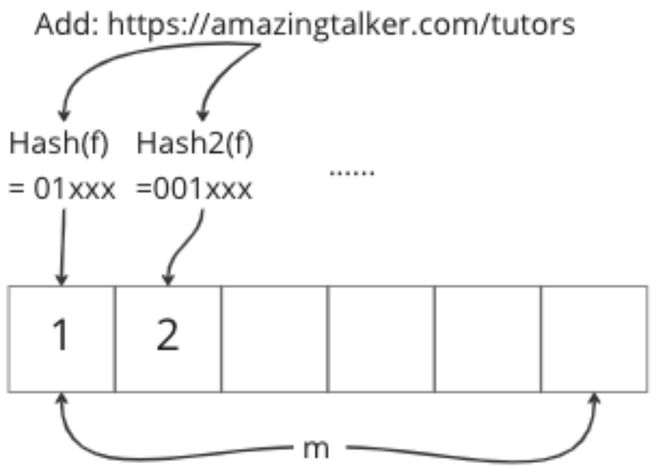
LogLog - 概念

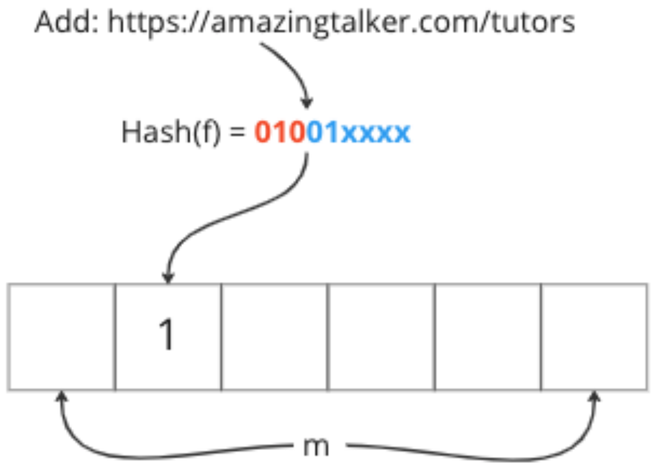
- element 經過 Hash function 輸出 int，轉成 binary format，把 0 = H / 1 = T，用 Leading 0 count 來推算 cardinality
  - ex. 0010101……
- 問題
  - 太過於運氣性質，被偏差值所影響
  - 預估都是 2 的次方倍，到後面每一層都差距很大

$$2^{(\text{leading H count}+1)}$$

LogLog - 再分散

- 解法
  - Hash m 次在更近一步分散，最後取平均
  - 但取 Hash 會需要耗費 CPU 資源，改用前幾個 bit 決定對應的 counter，剩餘 bit 才用來預估





LogLog - 演算法

- 演算法

Algorithm 3.7: Estimating cardinality with LogLog

Input: Dataset $\mathbb{D}$

Input: Array of m LogLog counters with hash function h

Output: Cardinality estimation

$\text{COUNTER}[j] \leftarrow 0, j = 0 \dots m-1$

for $x \in \mathbb{D}$ **do**

  $i \leftarrow h(x) := (i_0 i_1 \dots i_{M-1})_2 \,, i_k \in \{0,1\}$

  $j \leftarrow (i_0 i_1 \dots i_{p-1})_2$

  $r \leftarrow \texttt{rank}((i_p i_{p+1} \dots i_{M-1})_2)$

  $\text{COUNTER}[j] \leftarrow \max(\text{COUNTER}[j], r)$

$\mathrm{R} \leftarrow \frac{1}{m} \sum_{k=0}^{m-1} \text{COUNTER}[j]$

return $\alpha_m \cdot m \cdot 2^{\mathrm{R}}$

- 誤差

$$\delta \approx \frac{1.3}{\sqrt{m}}. \qquad (3.12)$$

LogLog - 實際所需空間

- Counter 只需要儲存 Leading 0 的 max count
  - 假設輸入範圍是 2^64，轉成 binary format 10010…0 (64 bit)
  - 儲存 64 只需要 7 bit!

LogLog - 實際所需空間

- Counter 只需要儲存 Leading 0 的 max count
  - 假設輸入範圍是 2^64，轉成 binary format 10010…0 (64 bit)
  - 儲存 64 只需要 7 bit!
  - 計算 leading 0 count ，數值範圍 0~64

LogLog - 實際所需空間

- Counter 只需要儲存 Leading 0 的 max count
  - 假設輸入範圍是 2^64，轉成 binary format 10010…0 (64 bit)
  - 計算 leading 0 count ，數值範圍 0~64
  - 輸入範圍與儲存空間的關係就是 Log(Log(2^64)) = 7

- 總共有 m 個 counter，total size = 6 * m bit!

LogLog - 實際所需空間

SuperLogLog

- 在 m 個 bucket 中，排序只取前 70% 後平均
- 誤差

$$\varepsilon = \frac{1.05}{\sqrt{m}}$$

HyperLogLog

- 改良方向
  - 當輸入很小時，改用 Linear Counting
  - 超過一個閥值，改用 LogLog
  - 逼近 Hash value 上限 (32 bit)，改用另一個公式
  - 不用平均，改用其他的公式彙整
- 誤差

$$\frac{1.04}{\sqrt{m}}$$

Algorithm 3.8: Estimating cardinality with HyperLogLog

```
Input: Dataset 𝔻
Input: Array of m LogLog counters with hash function h
Output: Cardinality estimation
COUNTER[j] ← 0, j = 0 ... m − 1
for x ∈ 𝔻 do
    i ← h(x) := (i_0 i_1 ... i_31)_2 , i_k ∈ {0, 1}
    j ← (i_0 i_1 ... i_{p−1})_2
    r ← rank((i_p i_{p+1} ... i_31)_2)
    COUNTER[j] ← max (COUNTER[j], r)
R ← Σ_{k=0}^{m−1} 2^{−COUNTER[j]}
n̂ = α_m · m^2 · 1/R
n ← n̂
if n̂ ≤ 5/2 m then
    Z ← count_{j=0...m−1} (COUNTER[j] = 0)
    if Z ≠ 0 then
        n ← m · log(m/Z)
else if n̂ > 1/30 2^32 then
    n ← −2^32 · log(1 − n/2^32)
return n
```

HyperLogLog++

- 改良方向
  - 把 Hash 範圍從 32 bit 改成 64 bit
  - 基於經驗調整參數與修正偏差

Algorithm 3.10: Estimating cardinality with HyperLogLog++

Input: Dataset $\mathbb{D}$

Input: Array of m LogLog counters with hash function h

Output: Cardinality estimation

$\text{COUNTER}[j] \leftarrow 0, j = 0 \dots m-1$

for $x \in \mathbb{D}$ **do**

  $i \leftarrow h(x) := (i_0 i_1 \dots i_{63})_2 \,, i_k \in \{0,1\}$

  $j \leftarrow (i_0 i_1 \dots i_{p-1})_2$

  $r \leftarrow \text{rank}((i_p i_{p+1} \dots i_{63})_2)$

  $\text{COUNTER}[j] \leftarrow \max(\text{COUNTER}[j], r)$

$\text{R} \leftarrow \sum_{k=0}^{m-1} 2^{-\text{COUNTER}[j]}$

$\hat{n} = \alpha_m \cdot m^2 \cdot \frac{1}{\text{R}}$

$n \leftarrow \hat{n}$

if $\hat{n} \leq 5m$ **then**

  $n \leftarrow \textbf{CorrectBias}(\hat{n})$

$\text{Z} \leftarrow \underset{j=0\dots m-1}{\text{count}} (\text{COUNTER}[j] = 0)$

if $\text{Z} \neq 0$ **then**

  $n_{\text{lin}} \leftarrow m \cdot \log \frac{m}{\text{Z}}$

  if $n_{\text{lin}} \leq \varkappa_m$ **then**

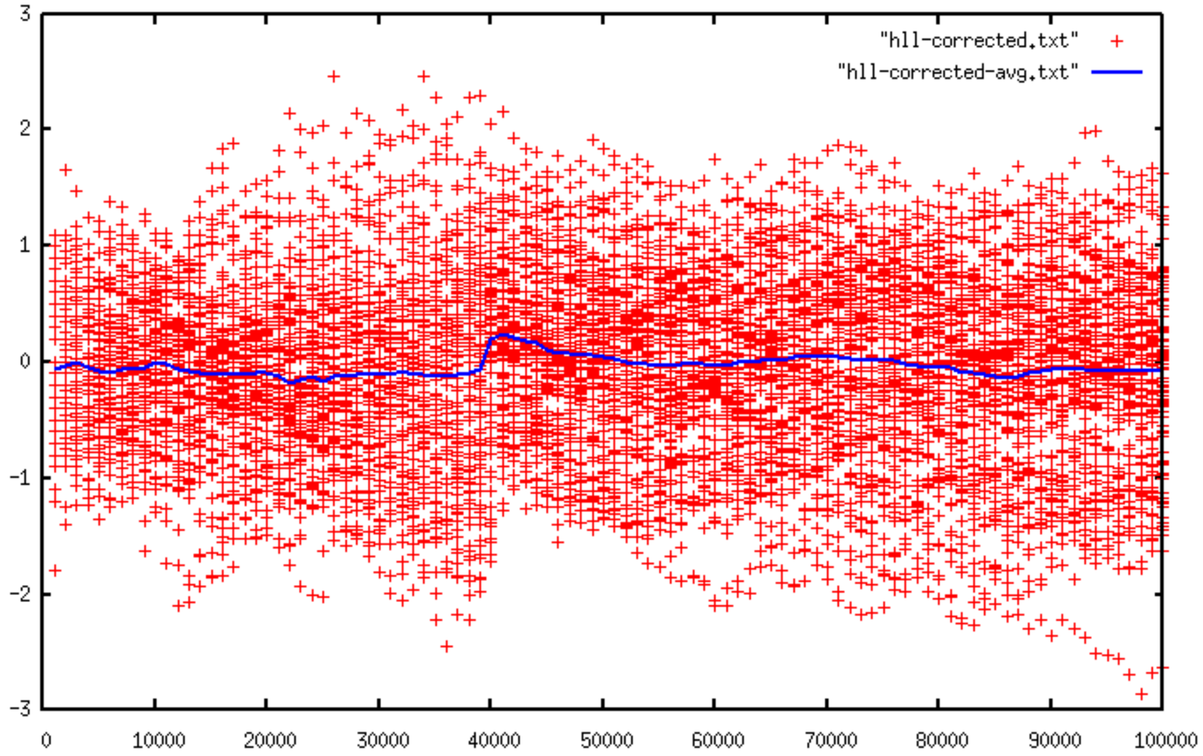
    $n \leftarrow n_{\text{lin}}$

return n

HyperLogLog - in Redis

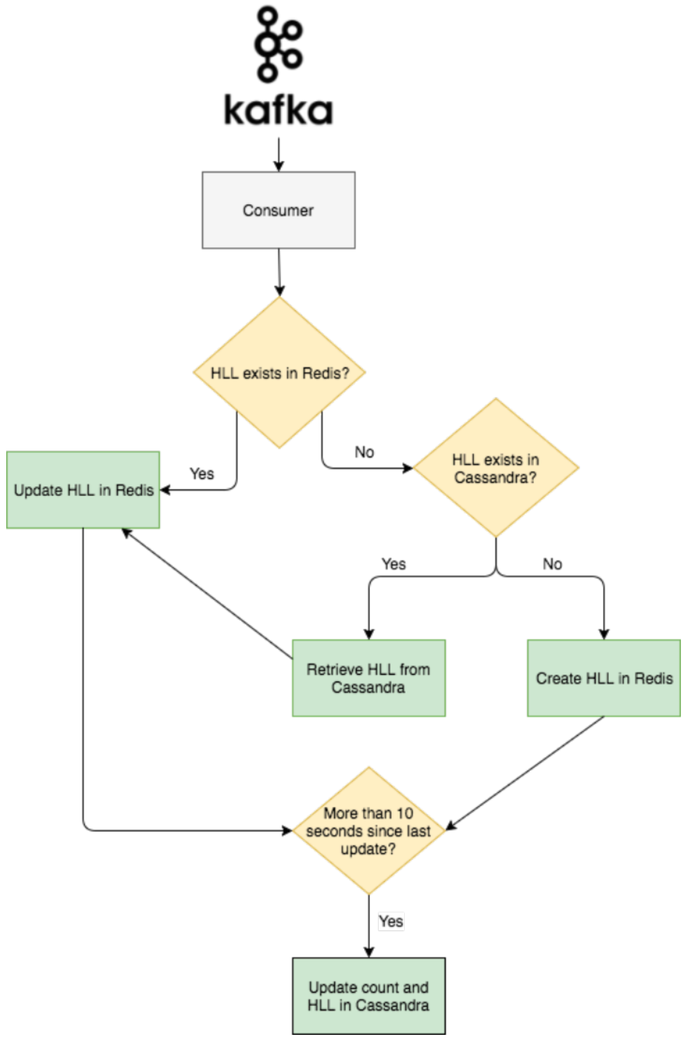
- Hash 產生 64 bit int
- 16384 (14 bit) bucket, so the standard error is 0.81%
- 50 bit 用來 counting => 6 bit
- 總共需要 12296 (12k) bytes

- Result
  - 500k * 12 kB = 6 GB
  - response in real time
  - 0.81 % error rate



HyperLogLog 實際應用

- Reddit 在計算 Page View 採用 Redis HLL
  - 熱門頁面會有破百萬的不重複用戶瀏覽



ref: https://www.redditinc.com/blog/view-counting-at-reddit/

Demo: BigQuery

782131

783512 (誤差 0.17 %)

https://cloud.google.com/bigquery/docs/reference/standard-sql/hll_functions#hyperloglog_functions

Demo: Redis

- RedisBloom
  - docker run -p 6380:6379 -p 8001:8001 redis/redis-stack
  - BloomFilter / Cuckoo Filter / t-digest / Count-min Sketch
- HyperLogLog (v2.8.9)
  - PFADD
  - PFCOUNT
- Bloom Filter
  - BF.ADD
  - BF.EXISTS

Summary

- Probabilistic Data Structure 降低精準度換取 CPU/Memory 資源上的節省
- Hash!
- Membership
  - element 是否出現在 dataset
  - Bloom Filter / Quotient Filter / Cuckoo Filter
- Cardinality
  - dataset 中不重複個數
  - Linear Count / HyperLogLog