

A Tour of Understanding HyperBit*

Zhenghua Lyu
kainwen@gmail.com

Greenplum Developer @ Broadcom

Jan, 2025

Abstract

Hyperloglog is the state-of-the-art algorithm for cardinality estimation and are widely used in the world. HypertBit* is a family of algorithms for cardinality estimation that tries to use less memory however with the same accuracy as HyperLogLog. The idea is from Doctor Robert Sedgewick and two other authors work together for several years and recently they published the paper [3]. This article is my note on analysis of the algorithm.

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 2 |
| 2 | Mathematical Warm-ups | 2 |
| 2.1 | Introduction to Poisson distribution | 2 |
| 2.2 | The number of r -cycles in a permutation | 3 |
| 2.3 | Poisson Process | 4 |
| 2.4 | Some lemmas in the paper | 5 |
| 2.5 | Bits operations | 5 |
| 3 | Requirement of the algorithms for cardinality estimation | 6 |
| 4 | HyperBitT | 7 |
| 5 | HyperBitBit and HyperBitBitBit | 10 |
| 6 | HyperTwoBits | 12 |
| 7 | The end and start of the tour | 13 |

1 Introduction

My previous story with cardinality estimation can be found [here](#) back to 2020. In recent years I have read some more talks by Doctor Robert Sedgewick from his [personal page](#):

- [HyperBit Talk in 2022, Sedgewick](#)
- [Sedgewick's homepage, cardinality estimation part](#)

For years I am waiting for the paper ready and finally it seems the Doctor forget to update his homepage that I search other co-author's website and find it. Thus a new happy tour is starting.

2 Mathematical Warm-ups

2.1 Introduction to Poisson distribution

This part is based on Chapter 6.11 of [4]. Consider N nuclei that sending out particles. There is a parameter r which is the probability that any single nuclei will emit a particle in any one second. Given N, r the probability of exact n particles are sent out in one second is:

$$p(n|r, N) = \binom{N}{n} r^n (1-r)^{N-n} \quad (1)$$

If N is enormously large and r is enormously small, that is $N \rightarrow \infty, r \rightarrow 0$. We have to specify the limiting process to make the model work*: $N \rightarrow \infty, r \rightarrow 0$, with $Nr = s$, s is a constant, thus we have $r = \frac{s}{N}$.

$$\begin{aligned} p(n|r, N) &= \binom{N}{n} r^n (1-r)^{N-n} \\ &= \frac{N^n}{n!} \frac{s^n}{N^n} (1 - \frac{s}{N})^{N-n} \\ &= \frac{1}{n!} \frac{N(N-1)\dots(N-n+1)}{N^n} s^n (1 - \frac{s}{N})^{N-n} \\ &= \frac{s^n}{n!} (1 - \frac{1}{N})(1 - \frac{2}{N})\dots(1 - \frac{n-1}{N})(1 - \frac{s}{N})^{N-n} \\ \Rightarrow \lim_{N \rightarrow \infty} p(n|r, N) &= \frac{s^n}{n!} \left(\lim_{N \rightarrow \infty} (1 - \frac{1}{N})(1 - \frac{2}{N})\dots(1 - \frac{n-1}{N}) \right) \left(\lim_{N \rightarrow \infty} (1 - \frac{s}{N})^{N-n} \right) \\ &= \frac{s^n}{n!} \lim_{N \rightarrow \infty} \exp\{(N-n) \ln(1 - \frac{s}{N})\} \\ &= \frac{s^n}{n!} \lim_{N \rightarrow \infty} \exp\{(N-n)((-\frac{s}{N}) - \frac{1}{2}(-\frac{s}{N})^2 + \mathcal{O}((\frac{s}{N})^3))\} \\ &= \frac{s^n e^{-s}}{n!} \end{aligned} \quad (2)$$

*Refer to [4] to understand the infinite-set paradoxing: it is dangerous to jump directly into an infinite set without specifying any limiting process to define its properties.

We get a new probability distribution with parameter s :

$$p(n|s) = \frac{s^n e^{-s}}{n!} \quad (3)$$

It is called Poisson distribution which we get from binomial distribution. Now lets see Poisson distribution's mean and variance.

$$\begin{aligned} E(n) &= \sum_{n=0}^{\infty} n \frac{s^n e^{-s}}{n!} = \sum_{n=1}^{\infty} n \frac{s^n e^{-s}}{n!} \\ &= \sum_{n=0}^{\infty} (n+1) \frac{s^{n+1} e^{-s}}{(n+1)!} \\ &= s e^{-s} \sum_{n=0}^{\infty} \frac{s^n}{n!} = s e^{-s} e^s = s \\ E(n^2 - n) &= \sum_{n=0}^{\infty} n(n-1) \frac{s^n e^{-s}}{n!} = \sum_{n=2}^{\infty} n(n-1) \frac{s^n e^{-s}}{n!} \\ &= \sum_{n=0}^{\infty} (n+2)(n+1) \frac{s^{n+2} e^{-s}}{(n+2)!} \\ &= s^2 e^{-s} \sum_{n=0}^{\infty} \frac{s^n}{n!} = s^2 \\ Var(n) &= E(n^2) - E(n)^2 = E(n^2 - n) + E(n) - E(n)^2 \\ &= s^2 + s - s^2 = s \end{aligned} \quad (4)$$

2.2 The number of r -cycles in a permutation

I have written a [post](#) (in Chinese) to show that: the average number of cards which still locate at their original position after a random shuffling is 1. The analysis tool is bivariate Generating Functions [6]. Lets use this tool to show the connection of the number of r -cycles in a permutation to poisson distribution [1].

We are studying all the permutations in the world: $\mathcal{P} = \{\text{all permutations}\}$, we define the size function to be $|p| = \text{the length of } p, \forall p \in \mathcal{P}$. Given we want to talk about the number of r -cycles, so we define the cost to be $cost(p) = \#CYC_r$.

$$P(z, u) = \sum_{p \in \mathcal{P}} z^{|p|} u^{cost(p)} = \sum_{n, k} p_{n, k} z^n u^k = \sum_n p_n(u) z^n = \sum_k q_k(z) u^k \quad (5)$$

Symbolic methods [1] [6] tells us:

$$\begin{aligned} \mathcal{P} &= \text{SET}(\text{CYC}_{\neq r}(\mathcal{Z}) + u \text{CYC}_r(\mathcal{Z})) \\ \mathcal{P} &= \text{SET}(\text{CYC}(\mathcal{Z}) + (u-1) \text{CYC}_r(\mathcal{Z})) \implies \\ P(z, u) &= \exp \left(\ln \frac{1}{1-z} + (u-1) \frac{z^r}{r} \right) = \frac{e^{\frac{(u-1)z^r}{r}}}{1-z} \end{aligned} \quad (6)$$

The Theorem 5.5 of [6] tells us that any function with the radius of convergence is strictly larger than 1 and $f(1) \neq 0$:

$$[z^n] \frac{f(z)}{(1-z)^\alpha} \sim \frac{f(1)}{\Gamma(\alpha)} n^{\alpha-1} \quad (7)$$

Using 7 in 6:

$$\begin{aligned} P(z, u) &\sim e^{\frac{u-1}{r}} = e^{-\frac{1}{r}} \sum_{k \geq 0} \frac{\left(\frac{u}{r}\right)^k}{k!} \implies \\ \text{Prob}(k) &\sim \frac{1}{p_n(1)} [u^k] \left(e^{-\frac{1}{r}} \sum_{k \geq 0} \frac{\left(\frac{u}{r}\right)^k}{k!} \right) = \frac{\left(\frac{1}{r}\right)^k e^{-\frac{1}{r}}}{k!} = \text{poisson}\left(\frac{1}{r}\right) \end{aligned} \quad (8)$$

We should that asymptotic law of the number of r -cycles is Poisson with rate $\frac{1}{r}$ [1].

2.3 Poisson Process

Poisson Process is stochastic process for modeling how an event comes to happen. Given an event X_i happens at time t_i (or this is the start time 0), the next event X_{i+1} happens at time t_{i+1} , the probability density function of the interval $x = t_{i+1} - t_i$ is: $f(x) = \lambda e^{-\lambda x}$, $x > 0$, the corresponding distribution function is $F(x) = \text{Prob}(u \leq x) = \int_0^x f(u) du = 1 - e^{-\lambda x}$.

I still remember that when I just joined Pivotal starting my career with Greenplum, I wrote a test framework using Poisson Process to simulate the SQL stream to enter the system. The core part is to write a random number generator of the the probability density function $f(x)$. Generally, most of programming languages provide a routine to generate a random number in the interval $[0, 1]$ uniformly, like Python's `random.random()`, Postgres' `random()`. We need to design algorithms to simulate random variables of some distribution functions given we have the routine for `unif(0, 1)`. Suppose a random variable $U \sim \text{unif}(0, 1)$, we have $\text{prob}(F^{-1}(U) \leq x) = \text{prob}(U \leq F(x)) = F(x)$, so $F^{-1}(U)$ is a random variable has distribution function $F(x)$. For Poisson Process, $F^{-1}(x) = \frac{1}{\lambda} \ln \frac{1}{1-x}$.

```
from random import random
from math import log
def interval(a):
    u = random()
    return -1/a*log(1-u)
```

Figure 1: Python Program simulating Poisson Process

Another definition of Poisson Process is: given any fixed time interval τ , the number N of events happening during this time interval is a random variable of Poisson distribution, $N \sim \text{Poisson}(\lambda\tau)$.

Let $\mathbf{N}(t)$ be a random Poisson Process with rate λ , if we uniformly split it into M independent Poisson Processes, then each sub-process' rate is $\frac{\lambda}{M}$.

More information can be found at [11.1.2 Basic Concepts of the Poisson Process](#) and [11.1.3 Merging and Splitting Poisson Processes](#).

2.4 Some lemmas in the paper

We prove some lemmas in the paper [3] as warm-up.

Lemma 1. *Given $\{X_n\}$ is a positive sequence of random variables and $\{a_n\}$, $\{b_n\}$ two sequences, and we have: $a_n \rightarrow a > 0$, $b_n \rightarrow 0$, $\frac{X_n - a_n}{b_n} \xrightarrow{d} \mathbb{N}(0, \sigma^2)$. f is a continuously differentiable function on $(0, +\infty)$ with $f'(a) \neq 0$, then $\frac{f(X_n) - f(a_n)}{b_n} \xrightarrow{d} \mathbb{N}(0, f'(a)^2 \sigma^2)$.*

Proof. Based on Lagrange mean value theorem, $\exists X_n^* \in [a_n, X_n]$ or $[X_n, a_n]$, $\frac{f(X_n) - f(a_n)}{b_n} = f'(X_n^*) \frac{X_n - a_n}{b_n}$. Since $\frac{X_n - a_n}{b_n} \xrightarrow{d} \mathbb{N}(0, \sigma^2)$, and $b_n \rightarrow 0$, then $X_n \xrightarrow{p} a_n$. With $a_n \rightarrow a$, we know $X_n \xrightarrow{p} a$. It is easy to know $X_n^* \xrightarrow{p} a$ because its upper and lower-bounds both $\xrightarrow{p} a$. f' is continuous, $f'(X_n^*) \xrightarrow{p} f'(a)$. \square

2.5 Bits operations

One important idea of the cardinality estimation algorithm family is transforming the original data into a binary string and then building a sketch based on simple features. Lets review some interesting bit operations for these features:

- $r(x)$: the number of trailing 1s in the binary string x , e.g. $r(0001111) = 4$
- $R(x) = 2^{r(x)}$, e.g. $R(0001111) = 16$
- $p(x)$: the number of all 1 bits in the binary string x , e.g. $p(1101101) = 5$

The famous book *The C Programming Language* [5] Ex2-9 is an exercise for $p(x)$:

In a two's complement number system, $x \&= (x-1)$ deletes the rightmost 1-bit in x . Explain why. Use this observation to write a faster version of `bitcount`.

The C code shown in Figure 2 is Brian Kernighan's Algorithm.

```
unsigned int bitcount(unsigned int x)
{
    unsigned int c = 0;
    while (x)
    {
        x &= (x - 1);
        c++;
    }
    return c;
}
```

Figure 2: C Program of bitcount

There are other algorithms that using several instructions to implement this, like the SWAR Algorithm. Some platforms also have an instruction POPCNT for this job. Postgres

```

#pragma GCC target("sse4")

int popcount(int x) {
    return __builtin_popcount(x);
}

// ASM piece
//popcount:
//.LFB0:
//      .cfi_startproc
//      endbr64
//      xorl    %eax, %eax
//      popcntl %edi, %eax
//      ret
//      .cfi_endproc

```

Figure 3: C Program using `__builtin_popcount`

take the advantage of these instructions: *Using POPCNT and other advanced bit manipulation instructions*. Another interesting blog can be found at *__builtin_popcount and POPCNT*. Test result in my local VM is shown 3:

According to the slide *A Memory-Efficient Alternative to HyperLogLog*, for $R(x)$ it just needs 3 instructions: $(\sim x) \& (x+1)$.

3 Requirement of the algorithms for cardinality estimation

Cardinality estimation is for datasets or scenarios in which getting an accurate value is impossible due to limited resources (memory, time) or is unnecessary. Thus the natural requirements for such kinds of algorithms are: quick and accurate. Quick means only one pass scan of the whole data is allowed and the code handling one item should just use several machine instructions with limited memory. Accurate means the algorithm must be mathematically analyzed to proved to be close to the real value. These kinds of algorithm are very interesting: they are so simple to implement* however so hard to analyze.

Given hyperloglog has been the start-of-the-art algorithm for decades, any new algorithms must be:

- just use $\sim 20, 30$ machine instructions to handle each item in the dataset
- Given M , just use cM bits memory for some constant $c \sim 2, 3$ when $N < 2^{64}$ **

*In fact, to implement the kinds of algorithm for industrial-level applications is not an easy job, see Google's best practice hyperloglog paper [2].

** 2^{64} is super huge, from <https://explodingtopics.com/blog/google-searches-per-day>, we know that there are approximately 8,323,333,333 searches on Google every day, to reach 2^{64} , it takes about 6071969 years.

- Hyperloglog uses about 6144 bits to achieve 10% relative accuracy, so the new algorithm must use less bits (better just 2000 ~ 4000) for the same accuracy

The idea of these algorithms is:

- hash each item to a binary string
- model the binary string as random experiments
- maintain a sketch related to the binary string
- use the final sketch to estimate the cardinality

The following pseudocode in Figure 4 of Hyperloglog shows the framework:

```

for tuple in M:
    # x is the bit array of the hash value
    (x[0], x[1], x[2], ... x[b-1], x[b], ...) = hash(tuple)
    # the first b bit
    bucket_id = (x[0], x[1], x[2], ... x[b-1])
    pos = the_first_1_bit_pos(x[b], x[b+1], ...)
    reg[bucket_id] = max(reg[bucket_id], pos)
Z = 1/sum(2^(-r) for r in reg)
return fix_coef*Z

```

Figure 4: Pseudocode of Hyperloglog

The source code of the family of HyperBit* algorithms can be found at [HyperBitT](#), [HyperBitBit64](#) and [HyperTwoBits](#).

4 HyperBitT

The first algorithm is called HyperBitT, it has three input arguments. The first two arguments are the same as HyperLogLog: the data stream and the number of sub-streams: M . There is an extra parameter T , which estimates $\log(\frac{N}{M})$. The algorithm's pseudocode is shown in Algorithm 1:

Algorithm 1 HyperBitT: S, M, T

```

sketch ← bit[M]                                ▷ an array of bits with size M, all 0
for s : S do
    x ← hash1(s)                                ▷ use the first hash to get a "random" binary string of 64bit
    k ← hash2(s, M)                             ▷ use the second hash to get log(M) bit hash
    if r(x) ≥ T then
        sketch[k] ← 1
    end if
end for
β ← 1.0 - p(sketch)/M

return (int) 2T M log  $\frac{1}{\beta}$ 

```

The core part of the analysis of the algorithm **HyperBitT** is the following theorem 1, heavily using Poisson approximation for binomial distribution. Reading the code of **HyperBitT**, it turns out that if the parameter T is too huge or too small, which leads to sketch all 0 or all 1 (β is 0 or 1), under such cases, the algorithm fails due to impossible to correct the estimation (the sketch tells us nothing). For **HyperBitT** there is a constraint $N = \Theta(M2^T)$.

Theorem 1. *Suppose that a string S has N distinct items and that **HyperBitT** processes S using M substreams with parameter T and terminates with βM 0s left in the sketch. Then the statistic $M2^T \ln(1/\beta)$ is approximately Gaussian with mean N and relative standard error c_β/\sqrt{M} where $c_\beta = \sqrt{1/\beta - 1/\ln(1/\beta)}$. Formally,*

$$\frac{\sqrt{M}}{c_\beta} \left(\frac{M2^T \ln(1/\beta)}{N} - 1 \right) \xrightarrow{d} \mathbb{N}(0, 1) \quad (9)$$

as $N, M, T \rightarrow \infty$ with $N = \Theta(M2^T)$.

Lets first do some experiments to have a glance at the results. The test data set contains 13874 items with 10000 distinct items (can be downloaded [here](#)). Test result is shown in Figure 5.

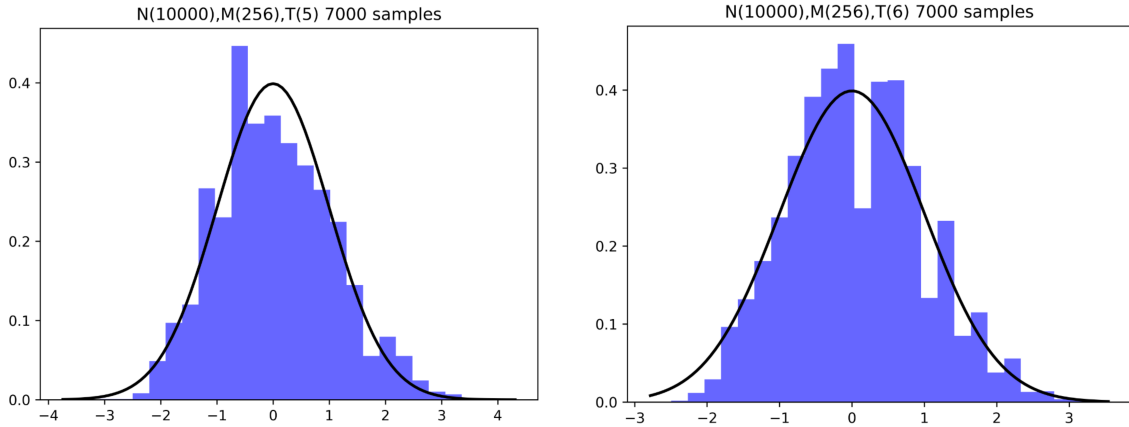


Figure 5: Distribution of the statistic in theorem 1

Now we need to do mathematic analysis to prove the theorem 1. All items in the input stream are dispatched to M substream uniformly for each distinct item. Given N is the real cardinality of the whole stream, for a specific substream i the cardinality is $\frac{N}{M}$ as an approximation. The event `sketch[i] == 0` means `r(x) >= T` never happens for all $\frac{N}{M}$ distinct items in substream i .

$$\begin{aligned}
\text{Prob}(\text{sketch}[i] == 0) &= \bigcap_{\text{each distinct } x} \text{Prob}(r(x) < T) \\
&= \prod_{\text{each distinct } x} (1 - \text{Prob}(r(x) \geq T)) \\
&= \prod_{\text{each distinct } x} (1 - \text{Prob}(\text{more than } T \text{ consistent trailing 1 bits})) \\
&= \left(1 - \frac{1}{2^T}\right)^{\frac{N}{M}} \sim e^{-\frac{N}{M2^T}} = \beta \text{ (last step is Poisson Approximation)}
\end{aligned} \tag{10}$$

The number of 0s in the final sketch bit array is a binomially distributed random variable: $\#0s \text{ in sketch} \sim b(M, \beta)$. Using the last step Approximation in 10, we have $e^{-\frac{N}{M2^T}} \sim \beta \implies \frac{N}{M} \sim 2^T \ln\left(\frac{1}{\beta}\right) \implies N \sim M2^T \ln\left(\frac{1}{\beta}\right)$. That means we need to correct the estimation using T by $\ln\left(\frac{1}{\beta}\right)$.

The following lemma 2 is the key bridge that allow us to prove theorem 1 using Poisson model. The proof of this lemma is beyond the scope of this note, readers please refer to the original paper. The idea here is we can replace X with Y or replace Y with X when p is very small.

Lemma 2. *Let $X \sim \text{Binomial}(n, p)$ and let $Y \sim \text{Poisson}(np)$ * where $n > 0$ and $p \in [0, 1]$. Then the total variation distance between them $d_{TV}(X, Y)$ is no greater than p ; in other words there exists a coupling of X and Y such that $\mathbb{P}(X \neq Y) \leq p$.*

Scanning the whole stream, the probability of selecting out an item whose hash value has more than T trailing 1s is 2^{-T} . Given there are N distinct items in the whole stream, the number of the selected items is $\text{Binomial}(N, 2^{-T})$. We can use a Poisson model to simulate this using Lemma 2 (because $2^{-T} \rightarrow 0$ when $T \rightarrow \infty$), that is, the number of the selected items is $\text{Poisson}(N2^{-T})$. If we consider any single sub-stream, the number of the selected items in that sub-stream is $\text{Poisson}(N2^{-T}/M)$, thus the probability of the corresponding sketch bit keeps 0 is $q = \text{Prob}_{\text{poisson}}(0) = e^{-\frac{2^{-T}N}{M}}$. Each sub-stream is independent with each other, then the total number of 0s in the final sketch bit array is: $\beta M \sim \text{Binomial}(M, q)$, with mean Mq and variance $Mq(1 - q)$.

When M is large enough, based on *Central Limit Theorem*, suppose $q = e^{-\frac{2^{-T}N}{M}} \rightarrow e^{-a}$, we have:

$$\frac{\beta M - Mq}{\sqrt{Mq(1 - q)}} \xrightarrow{d} \mathbb{N}(0, 1) \implies \frac{M\beta - Mq}{\sqrt{M}} \xrightarrow{d} \mathbb{N}(0, e^{-a}(1 - e^{-a})) \tag{11}$$

In the previous rough analysis, things seem related to $\ln\left(\frac{1}{\beta}\right)$. Recall the Lemma 1 we prove in warm-ups, and in the above 11 we have something of β , it seems that we should consider building a function $f(x) = \ln\left(\frac{1}{x}\right)$ to link them together. Replace β, q with $\ln\frac{1}{\beta}, \ln\frac{1}{q}$ in 11, we have:

*In the original paper, the authors wrote \in , I believe it is a typo.

$$\begin{aligned}
f'(x) &= -\frac{1}{x} \\
\sqrt{M}(\ln \frac{1}{\beta} - \ln \frac{1}{q}) &\stackrel{d}{\rightarrow} \mathbb{N}(0, e^a - 1) \\
\implies \ln \frac{1}{\beta} - \ln \frac{1}{q} &\stackrel{p}{\rightarrow} 0 \implies \ln \frac{1}{\beta} \stackrel{p}{\rightarrow} \ln \frac{1}{q} \implies \ln \frac{1}{\beta} \stackrel{p}{\rightarrow} a
\end{aligned} \tag{12}$$

The main part of theorem 1 is proved. The following Figure 6 (by wolframalpha) shows the relationship of variance (accuracy) and β . We find that when β close to 1, the accuracy decreases when β increases.

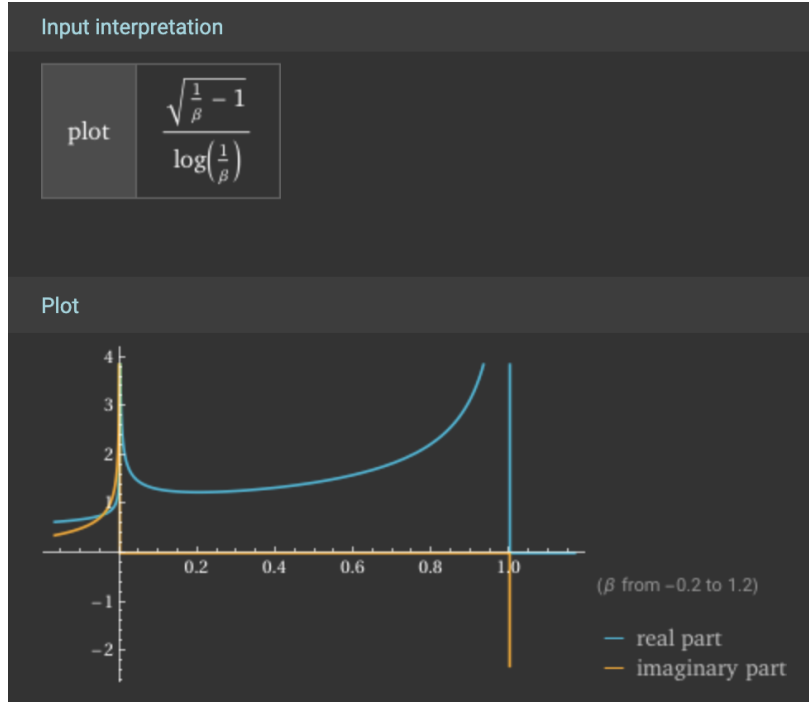


Figure 6: Graph of $\frac{\sqrt{\frac{1}{\beta} - 1}}{\ln \frac{1}{\beta}}$

5 HyperBitBit and HyperBitBitBit

The algorithm of the previous section, **HyperBitT** needs an extra parameter as input, which is not very good. This section we talk about two variants that maintain T as an algorithm internal variable. **HyperBitT** fails when the sketch bit array is almost full with 1s, that hints us T might be under-estimated. A natural idea is to increase T when needed (when the sketch bit array is almost all 1s). After increasing T by 1, we need a second sketch reflecting now it is $T + 1$. But, what if the second sketch becomes almost full with 1 again? To continue the increasing steps means we need more and more and more new

sketch bit arrays which is impossible. What about choosing a step length more than 1 for T ? Increasing T will lead to more 0s in the sketch bit array, and the accuracy decreases. Then could we find something like *Nash equilibrium* for this kind of algorithm?

We use a practical case with $M = 64$ to check if we can just use two bit-arrays as sketch. If we define almost full to be 97%, then it means we need $64 * 97\% \approx 62$ 1-bits and $\beta = 2/64 \approx 0.31$. When the first sketch becomes almost full, we increase T by i , and we maintain the second sketch for $T + i$. We have to find the best i that can save the third sketch. We should still use the algorithm like **HyperBitT** to estimate N so that $\ln \frac{1}{\beta} = \frac{1}{2^i} \ln \frac{1}{\beta_i} \implies \beta_i = e^{-\frac{1}{2^i} \ln \frac{1}{\beta}}$, here β_i is the β for $T + i$. We plot β_i for $i = 0, 1, \dots, 9$ in Figure 7.

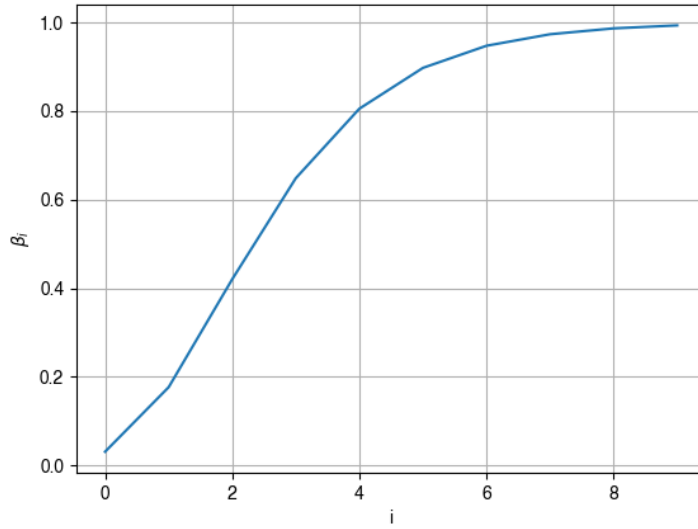


Figure 7: β_i for different i

If we choose step length as $i = 4$, then $\beta_4 > 0.8$ tells us that more than 80% bits in the sketch will be 0s. This might be a good point to get rid of the third sketch. Then the new algorithm **HyperBitBit64** is shown in 2. Surely, when setting the first sketch using the second and resetting the second to 0, we will lose some 1 bits in the second sketch, this might only have limited impact on the final estimation. **HyperBitBit** is a true streaming algorithm which does not need an extra parameter.

Algorithm 2 HyperBitBit64: S


```
int  $T \leftarrow 1$ 
int  $M \leftarrow 64$ 
long  $sketch0 \leftarrow 0$ 
long  $sketch1 \leftarrow 0$ 
for  $s : S$  do
    long  $x \leftarrow hash1(s)$   $\triangleright$  use the first hash to get a "random" binary string of 64bit
    int  $k \leftarrow hash2(s, M)$   $\triangleright$  use the second hash to get  $\log(M) = 6$  bit hash
    if  $r(x) \geq T$  then
         $sketch0 \leftarrow sketch0 \mid 1L \ll k$ 
    end if
    if  $r(x) \geq T + 4$  then
         $sketch1 \leftarrow sketch1 \mid 1L \ll k$ 
    end if
    if  $p(sketch0) > 0.97M$  then
         $sketch0 \leftarrow sketch1$ 
         $sketch1 \leftarrow 0$ 
         $T \leftarrow T + 4$ 
    end if
end for
 $\beta \leftarrow 1.0 - p(sketch0)/M$ 

return (int)  $2^T M \log \frac{1}{\beta}$ 
```

If we continue to enlarge M , we can use the same skill to choose step length and even introduce the third sketch to get an algorithm **HyperBitBitBit** which uses exact 3 bit-array sketches.

Lets do a quick analysis on the algorithm **HyperBitBit**. We expect when the algorithm 2 terminates, $sketch0$ is the same as the final sketch in algorithm 1. Thus we can still use theorem 1 to analyze the accuracy. A rough guess of the variance can be got to compute the average value of the variance in theorem 1 in a reasonable interval of β . **HyperBitBitBit** for M at least 4096, it uses $3M$ bits and achieves relative standard error of about $\frac{1.46}{\sqrt{M}}$.

6 HyperTwoBits

This section talks about a new algorithm that produce the same result as the algorithm **HyperBitBitBit** however just uses $2M$ bits. The trick is to note a simple fact: if a bit is set 1 in a sketch of $T + 4$, then the corresponding bit of the sketch of T must also be set; if a bit is set 1 in a sketch of $T + 8$, then the corresponding bit of the sketch of $T + 4$ and the sketch of T must also be set. Suppose b_0, b_4, b_8 are 3 bits of the k th position of the $sketch_T, sketch_{T+4}, sketch_{T+8}$. From Table 1 there are in total 8 choices for them, but after applying the simple fact, only 4 remains (with  symbols in the last column of the table). We can just use 2 bits to encode this 3 bits.

| b_0 | b_4 | b_8 | OK? |
|-------|-------|-------|-----|
| 0 | 0 | 0 | ✓ |
| 0 | 0 | 1 | ✗ |
| 0 | 1 | 0 | ✗ |
| 0 | 1 | 1 | ✗ |
| 1 | 0 | 0 | ✓ |
| 1 | 0 | 1 | ✗ |
| 1 | 1 | 0 | ✓ |
| 1 | 1 | 1 | ✓ |

Table 1: Possible values for bits of HyperBitBitBit

7 The end and start of the tour

This note quickly goes through the latest research of Dr. Sedgewick, I feel we can use the similar analytical combinatorics method of HyperLogLog paper for HyperBit* here (I haven't yet tried, maybe sometime later). As a software engineer, I should keep starting new tours, keep thinking and keep self-improvement. Use the final picture 7 to end this tour.

自强不息，厚德载物

References

- [1] P. Flajolet and R. Sedgewick. *Analytic combinatorics*. cambridge University press, 2009.
- [2] S. Heule, M. Nunkesser, and A. Hall. Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 683–692, 2013.
- [3] S. Janson, J. Lumbroso, and R. Sedgewick. Bit-array-based alternatives to hyperloglog. In *35th International Conference on Probabilistic, Combinatorial and Asymptotic Methods for the Analysis of Algorithms (AofA 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.
- [4] E. T. Jaynes. *Probability theory: The logic of science*. Cambridge university press, 2003.
- [5] B. W. Kernighan and D. M. Ritchie. *The C programming language*. prentice-Hall, 1988.
- [6] R. Sedgewick and P. Flajolet. *An introduction to the analysis of algorithms*. Pearson Education India, 2013.