Bit-array-based alternatives to HyperLogLog

Svante Janson[1]
Department of Mathematics, Uppsala University, Sweden

Jérémie Lumbroso
Department of Computer Science, University of Pennsylvania, USA

Robert Sedgewick[2]
Department of Computer Science, Princeton University, USA

Abstract

We present a family of algorithms for the problem of estimating the number of distinct items in an input stream that are simple to implement and are appropriate for practical applications. Our algorithms are a logical extension of the series of algorithms developed by Flajolet and his coauthors starting in 1983 that culminated in the widely used `HyperLogLog` algorithm. These algorithms divide the input stream into M substreams and lead to a time-accuracy tradeoff where a constant number of bits per substream are saved to achieve a relative accuracy proportional to $1/\sqrt{M}$. Our algorithms use just one or two bits per substream. Their effectiveness is demonstrated by a proof of approximate normality, with explicit expressions for standard errors that inform parameter settings and allow proper quantitative comparisons with other methods. Hypotheses about performance are validated through experiments using a realistic input stream, with the conclusion that our algorithms are more accurate than `HyperLogLog` when using the same amount of memory, and they use two-thirds as much memory as `HyperLogLog` to achieve a given accuracy.

2012 ACM Subject Classification Theory of computation → Sketching and sampling

Keywords and phrases Cardinality estimation, sketching, Hyperloglog

Digital Object Identifier 10.4230/LIPIcs.AofA.2024.00

Acknowledgements This work is dedicated to the memory of Philippe Flajolet

1 Introduction

Counting the number of distinct items in a data stream is a classic computational challenge with many applications. As an example, consider the stream of strings taken from a web log shown in the left column of Table 1 (we will use 1 million strings from this log of which $N = 368,217$ are distinct values as a running example in this paper). There is no bound on the length of the stream, but maintaining an estimate of the number of different strings is useful for many purposes.

One classic application is in computer networks. The ability to estimate the number of different visitors of a website is certainly of interest, and can be critical in maintaining the integrity of the site. For example, a significant drop in the percentage of distinct visitors in a given time period might be an indication that the site is under a denial-of-service attack.

Another classic application is found in database systems, where estimating the number of different strings having each attribute is a critical piece of knowledge in implementing certain common data base operations. In this case, the length of the streams is available, but may be very large, and a rough estimate suffices, so using a streaming algorithm is appropriate.

Elementary algorithms for solving the problem are standard in introductory computer science classes. Perhaps the simplest is to use a *hash table*, but that requires saving all the

[1] Supported by the Knut and Alice Wallenberg Foundation and the Swedish Research Council.
[2] Corresponding author

35th International Conference on Probabilistic, Combinatorial and Asymptotic Methods for the Analysis of Algorithms.
Editors: Cécile Mailler and Sebastian Wild; Article No. 00; pp. 00:1–00:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

s	x	k	r(x)	sketch[]
81.95.186.98.freenet.com.ua	111...1111101111011110011101011	7	2	00000001
lsanca.dsl-w.verizon.net	011...1010100010001111110100000	3	0	00000001
117.222.48.163	110...0111001100000111011101101	6	1	00000001
1.23.193.58	100...0100101001000011101100011	4	2	00001001
188.134.45.71	101...0101111000101101000111001	5	1	00001001
gsearch.CS.Princeton.EDU	010...1010011011000011010000100	2	0	00001001
81.95.186.98.freenet.com.ua	011...1011110001110000111010000	3	0	00001001
81.95.186.98.freenet.com.ua	111...1111101111011110011101011	7	2	00001001
1.23.193.58	000...0100111001111101011100111	0	3	10001001
lnse3.cht.bigpond.net.au	111...0110011001011011101001110	7	0	10001001
117.211.88.36	000...1001100010100010010111010	0	0	10001001
1.23.193.58	000...0100111001111101011100111	0	3	10001001
lsanca.dsl-w.verizon.net	011...1010100010001111110100000	3	0	10001001
81.95.186.98.freenet.com.ua	111...1111101111011110011101011	7	2	10001001
gsearch.seas.upenn.edu	000...1000100011011010100001000	0	0	10001001
109.108.229.102	010...1010111101010110110011111	2	5	10101001
msnbot.search.msn.com	001...1001110110111001001101100	1	0	10101001

Table 1 Computing a sketch for `HyperBitT` (with $M = 8$ and $T = 1$)

items in memory, which is far too high a cost to be useful in typical applications. In fact, *any* method for computing an exact count must save all the items in memory (trivial proof: any item not saved might or might not be distinct from all the others, and that fact cannot be known until the last item is seen).

Accordingly, we focus on *estimating* the count. In typical applications, exact counts are actually not needed—the estimates are being used to make relative decisions that do not require full accuracy.

Since the seminal research by Flajolet and Martin in the 1980s [5][6] it has been known that we actually can get by with a surprisingly small amount of memory. The *practical cardinality estimation problem* is to estimate the number of distinct items in a data stream under the following constraints:

- Each item is examined only once.
- The time to process each item is a very small constant multiple of the size of the item.
- The amount of memory used is very small, no matter how large the stream.
- The estimate is expected to be within a small percentage of the real count.

A solution to this problem typically is defined by an implementation that makes clear its time and space requirements and an analysis that provides a precise characterization of how the estimate compares to the actual value.

For many years, the state of the art in solving the practical cardinality estimation problem has been `HyperLogLog`, the last in the series of algorithms developed by Flajolet and colleagues from the 1980s through the 2000s [4] [7] [9] [14]. `HyperLogLog` is based on four main ideas: *Hashing* is used to convert each item in the stream into a fixed-length binary number; the position of the *rightmost zero* is computed, taking the maximum value found as an estimate of the binary logarithm of the count; a technique known as *stochastic averaging* splits the stream into M independent substreams so that an average of experimental results can be computed; and the *harmonic mean* is used to properly handle outlying values. One reason HyperLogLog is so widely used is that *precise analysis* of the bias in the estimate provides the basis for formulating hypotheses about how the algorithm will perform in practical situations, and the results of experiments that validate the hypotheses are presented. The analysis exposes a *space-accuracy tradeoff*, allowing practitioners to choose with confidence the amount of memory needed to achieve a given accuracy or the accuracy achieved for a given amount of memory use: For a stream with N distinct values and using M substreams, HyperLogLog uses $M \lg\lg N$ bits and typically produces an estimate with a relative standard error of $c/\sqrt{M}$ where $c \doteq 1.04$.

A series of theory papers have proven that $O(M)$ bits are necessary and sufficient to achieve estimates with asymptotic accuracy on the same order as HyperLogLog, but these papers lack implementations, likely because the implied constants in the proofs are much too large for the methods to be viable in practice [1][10][11]. They also make the implicit assumption that strong assumptions on the hash functions are necessary (even to the point of dismissing algorithms like HyperLogLog as illegitimate). Strong hash functions add to the expense of processing each item, and the idea that using one makes any difference at all in practice is tenuous at best (see, for example, [3] for a discussion of this issue). In this paper, we focus on algorithms with the potential to be useful in practice—we use hash functions that are widely used in practice and hypothesize that any differences from the ideal are relatively insignificant. *Any* practical application of hashing, however perfect in theory, must assume, at least, that random bits exist, and therefore requires such a hypothesis.

HyperLogLog uses $5M$ bits for $N < 2^{32}$, but much higher values are typical in modern applications. Since it is safe to assume that $N < 2^{64}$, HyperLogLog demonstrates that $6M$ bits suffice for the practical cardinality estimation problem. Some improvements to HyperLogLog and some interesting new approaches to the problem have been studied in recent years [16] [19] [15] [12] [17] but we are still left with the following question: can we find a practical algorithm as simple as HyperLogLog with comparable accuracy that uses cM bits for some constant c that is significantly less than 6?

In this paper, we provide answers to this question. The algorithms we present have the same structure as HyperLogLog but use much less memory—instead of recording the maximum number of trailing ones, we focus on *one* bit per sub-stream indicating whether a threshold has been hit. In Section 2, we use a rough estimate of the cardinality as an input parameter in order to set the threshold to be the logarithm of the extimated number of distinct items per substream. As such, the resulting algorithm is *not* a streaming algorithm, but it serves as a basis for the streaming algorithms in Section 3 and Section 4 that do solve the practical cardinality estimation problem, using just *two* bits per substream. In Section 5 we conclude by discussing how these algorithms match up against those in the literature.

2 HyperBitT

Our first algorithm uses the standard technique of starting with a rough estimate of the cardinality and is therefore not properly a streaming algorithm, as no fixed estimate can remain accurate as the cardinality grows without bound. We consider this algorithm because, as we will see, it is sometimes useful in its own right, and it admits a precise analysis that we can use to develop the streaming algorithms in Section 3 and Section 4.

We start with hashing and stochastic averaging with M substreams precisely as does HyperLogLog, but use just *one* bit per substream, as follows. Of course, we expect each substream to have about N/M distinct values, and it has been known since the original

Algorithm 1 `HyperBitT`.

```
public static int estimateHBT (Iterable<String> stream, int M, int T)
{
   bit[] sketch[M];
   for (String s : stream)
   {
      long x = hash1(s);                    // 64-bit hash
      int  k = hash2(s, M);                 // (lg M)-bit hash
      if (r(x) > T) sketch[k] = 1;          // more than T trailing 1s?
   }
   double beta = 1.0 - 1.0*p(sketch)/M;  // fraction of 0s in sketch
   return (int) (Math.pow(2, T)*M*Math.log(1.0/beta));
}
```

work of Flajolet and Martin [5] that the maximum number of trailing `1`s found among the items in a stream is a good estimator of the logarithm of the number of distinct items in the stream. (Indeed, this is the same as the length of the rightmost path in a random trie, a quantity that was studied in the 1970s.) In this spirit, we use a parameter T as an estimate of $\lg(N/M)$. That is, 2^T is an estimate of N/M, and $2^T M$ is an estimate of the cardinality N. Now, we maintain a *sketch* comprising an array of M bits, one per substream, and set the bit corresponding to a substream to `1` when an item from that substream has more than `T` trailing `1`s. When we want to estimate the number of distinct values in the stream, it turns out that we can use a simple function of the number of `0` bits in the sketch to improve our estimate. The algorithm may produce an inaccurate result or fail completely if the rough estimate `T` is poorly chosen, but, as we will see, it is remarkably forgiving.

Implementation

We start with a bit array `sketch[]` with one bit per substream, initialized to all `0`s. For clarity, we use a `bit[]` type to describe our algorithms—although few programming languages support an explicit `bit[]` type, the abstraction is easily implemented. For small M, we can use integer values; for large M, we can use shifting and masking on arrays of integers (see Appendix B). We typically use a power of two for convenience.

For each new item `s` in the stream, we compute a hash value `x` to represent it and a second hash value `k` to identify its substream (typically, one might compute a 64-bit hash and use the leading $\lg M$ bits for `k` and the rest for `x`). Then we compute `r(x)`, the number of trailing `1`s in `x`. As described in Appendix B, this operation can be implemented with only a few machine-language instructions. If `r(x)` is larger than `T`, we set `sketch[k]` to `1`. Table 1 is a trace of the process for a small sequence of hash values with $M = 8$ and $T = 1$.

When the stream is exhausted, we compute a correction to the rough estimate of $N = 2^T M$ that takes into account some bias, as a function of the bit values in the sketch. Specifically, we are interested in the parameter β, the proportion of `0`s in the sketch. As indicated by the analysis below, the appropriate correction factor is $\ln(1/\beta)$. If the sketch is small enough to fit in a computer word, computing the number of `1`s in the sketch is a classic machine-language programming exercise and is actually a single instruction in many modern machine architectures. For clarity, we use the function `p(sketch)`; for large M it is preferable to just increment a counter each time a sketch bit is changed from `0` to `1`, as described in Appendix B. The implementation in Algorithm 1 follows immediately and is easily translated to any programming language.

If T is too small or too large, the algorithm fails because the estimate cannot be reasonably corrected (when β is close to 0 or 1, the correction factor is too large or too small to be useful). But, as we shall see, the algorithm does produce accurate results for a remarkably large range of cardinality values, and we can precisely characterize that range and the accuracy.

Analysis.

As a basis for developing an intuition about the problem, we start with an approximate analysis for the mean value of the number of distinct values in the stream. After N distinct values have been processed from the input stream, we have seen an average of N/M distinct values in each substream. As an approximation, assume that *exactly* N/M values go to each substream. The probability that a given value has at least T trailing `1`s is $1/2^T$ so the probability that a given bit in `sketch[]` remains `0` after N/M values are processed in its corresponding substream is given by a Poisson approximation

$$\Big(1 - \frac{1}{2^T}\Big)^{N/M} \sim e^{-N/(M2^T)}$$

(see for example, [18]). The number of `0`s in `sketch[]` is a binomially distributed random variable, so this value is also (approximately) β, the expected proportion of `0`s in `sketch[]` after N values have been processed. Thus, $N/M \sim 2^T \ln(1/\beta)$ and the expected number of values processed is $N \sim M2^T \ln(1/\beta)$. In other words, we need to correct our rough estimate of the number of values per stream by the factor $\ln(1/\beta)$.

A full detailed analysis provides much more information, which is critical for studying the performance of the algorithm. Specifically, we are able to approximate the *distribution* of the reported cardinality, which gives us the information needed to estimate how accurate it will be for given values of M.

The proof is based on the idea of *Poissonization*—instead of assuming that we have a fixed given number N of distinct items, we assume that the number is random with a Poisson distribution. It uses two technical lemmas from probability theory:

▶ **Lemma 1.** *Suppose that $X_n \geq 0$ are random variables and a_n, b_n, and σ^2 numbers such that, as $n \to \infty$, we have $a_n \to a > 0$, $b_n \to 0$, and $(X_n - a_n)/b_n \xrightarrow{d} \mathbb{N}(0, \sigma^2)$. If f is a continuously differentiable function on $(0, \infty)$ with $f'(a) \neq 0$, then $(f(X_n) - f(a_n))/b_n \xrightarrow{d} \mathbb{N}(0, f'(a)^2\sigma^2)$.*

Proof. See Appendix A. ◀

▶ **Lemma 2.** *Let $X \sim \text{Binomial}(n, p)$ and let $Y \in \text{Poisson}(np)$ 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 $\mathbb{P}(X \neq Y) \leq p$.*

Proof. See Theorem 2.M and pages 1–8 in [2]. ◀

▶ **Theorem 3.** *Suppose that a stream 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 $c_\beta/\sqrt{M}$ where $c_\beta = \sqrt{1/\beta - 1}/\ln(1/\beta)$. Formally,*

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

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

Proof. Assume first that $N \sim aM2^T$ for some $a \in (0, \infty)$. Pretend that the distinct items in the stream arrive according to a Poisson process with rate 1. We then may consider the process at a given time $\tilde{N}$. If we keep $\tilde{N}$ fixed, then the number of distinct items seen so far is a random variable obeying a Poisson distribution Poisson($\tilde{N}$). We let $\tilde{N} \sim N \sim aM2^T$. For reference, we summarize here the notations used in this proof:

- $N \sim aM2^T$, the cardinality of the stream seen so far when Algorithm 1 terminates
- a, a positive number
- $\hat{N} = M2^T \ln(1/\beta)$, the reported estimate of N
- $\tilde{N} \sim aM2^T$, the Poisson parameter

Our goal is to approximate the distribution of $\hat{N}$.

We begin by finding, in the Poisson model, the distribution of βM, the number of `0`s in the sketch. Since a randomly thinned Poisson process is a new Poisson process, it follows that each of the M substreams is a Poisson process with rate $1/M$, and thus the number of distinct items in each of them is Poisson($\tilde{N}/M$). These random numbers are independent, and each item in the kth substream has probability 2^{-T} to set `sketch[k]` to `1`. It follows that if the number of such items is Y_k, then Y_k is also Poisson, with $Y_k \in \text{Poisson}(2^{-T}\tilde{N}/M) = \text{Poisson}\big(\tilde{N}/(M2^T)\big)$. Now, let q be the probability that `sketch[k]=0` (which is the same for all k). Then

$$q = \mathbb{P}(Y_k = 0) = \exp\Big(-\frac{\tilde{N}}{M2^T}\Big) \to e^{-a}. \tag{2}$$

Since the numbers Y_k are independent, the total number of `0`s in the sketch is

$$\beta M \in \text{Binomial}(M, q). \tag{3}$$

with mean Mq and variance $Mq(1-q)$.

As $M \to \infty$, we have the normal approximation to the binomial:

$$\sqrt{M}(\beta - q) = \frac{M\beta - Mq}{\sqrt{M}} \xrightarrow{d} \mathbb{N}\big(0, e^{-a}(1 - e^{-a})\big). \tag{4}$$

Now, applying Lemma 1 with the function $f(x) = \ln(1/x)$ gives

$$\sqrt{M}\big(\ln(1/\beta) - \ln(1/q)\big) \xrightarrow{d} \mathbb{N}(0, e^a - 1). \tag{5}$$

Consequently, since $\hat{N} = M2^T \ln(1/\beta)$, $M2^T/\tilde{N} \to 1/a$, and $\ln(1/q) = \tilde{N}/M2^T$, we have

$$\sqrt{M}\Big(\frac{\hat{N}}{\tilde{N}} - 1\Big) = \sqrt{M}\frac{M2^T}{\tilde{N}}\Big(\ln\frac{1}{\beta} - \ln\frac{1}{q}\Big) \xrightarrow{d} \mathbb{N}\big(0, a^{-2}(e^a - 1)\big). \tag{6}$$

Furthermore, (5) implies $\ln(1/\beta) - \ln(1/q) \xrightarrow{p} 0$, and thus, using (2), $\ln(1/\beta) \xrightarrow{p} a$; hence (6) implies (1) (with $\tilde{N}$ instead of N).

This is the desired result for the Poisson model. To prove the result for a given number N of items, we use Lemma 2. We may assume that we start by selecting all items with at least `T` trailing `1`s. Since each item is selected with probability 2^{-T}, the number of selected items is Binomial($N, 2^{-T}$). Similarly, if we consider the Poisson model with Poisson(N) items (thus choosing $N = \tilde{N}$ above) then the number of selected items is Poisson($N2^{-T}$). By Lemma 2. we may couple the two versions such that the number of selected items agree with probability no less than $1 - 2^{-T} \to 1$. Hence, (1) for a fixed N follows from the Poisson version.

We have proved that (1) holds when $N/(M2^T)$ converges to a limit in $(0, \infty)$. The more general assumption $N = \Theta(M2^T)$ implies that every subsequence has a subsubsequence such that $N/(M2^T)$ converges, and thus (1) holds for the subsubsequence. As is well known, this implies that the full sequence converges (see Section 5.7 in [8]).

◀

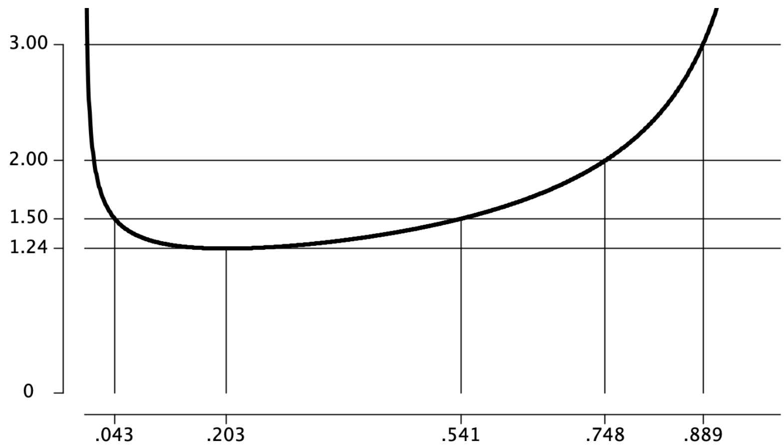


Figure 1 This plot shows the coefficient of $1/\sqrt{M}$ in the relative standard error $c_\beta = \sqrt{1/\beta - 1}\big/\ln(1/\beta)$ (y-coordinate) for β (fraction of 0s in the sketch) between 0 and 1 (x-coordinate). The value of c_β goes to infinity as β approaches 0 or 1, but it is relatively small when β is not close to these extremes. For example, $c_\beta < 1.5$ when $.043 < \beta < .541$, $c_\beta < 2$ when $.014 < \beta < .748$, and $c_\beta < 3$ when $.0035 < \beta < .888$.

To summarize, the goal of `HyperBitT` is to compute an estimate of N, the cardinality of the input stream. To do so, it takes two parameters

- M, the number of substreams (and the number of bits used)
- T, a rough estimate of $\lg(N/M)$

and, using an M-bit sketch, computes a value

- β, the fraction of 0s in the sketch.

Theorem 3 provides formulas for two important pieces of information, as functions of β:

- the correction factor $\ln(1/\beta)$, leading to the estimate $2^T M \ln(1/\beta)$ for N
- the coefficient of $1/\sqrt{M}$ in the relative standard error $c_\beta = \sqrt{1/\beta - 1}\big/\ln(1/\beta)$

This is the information that we need to properly choose the value of T. Of most interest is the fact that c_β is relatively small and is large only when β is close to 0 or 1 (see Figure 1). If T is too small, then the sketch will be predominately 1s, and β will be close to 0; if T is too large, the sketch will be predominantly 0s and β will be close to 1.

As an example, suppose that we take $M = 1024$ and aim to keep $c_\beta < 1.5$, which is the case when $.043 < \beta < .541$ (see Figure 1). As indicated in this table, each value of T leads to an accurate answer for a rather large range of values of N.

T	6	7	8	9	10	11
$M2^T \ln(1/\beta)$ for $\beta = .541$	40,261	80,522	161,044	322,089	644,177	1,288,356
$M2^T \ln(1/\beta)$ for $\beta = .043$	206,212	412,425	824,850	1,649,701	3,299,402	6,598,804

Validation

The purpose of our analysis is to enable us to hypothesize that the cardinality returned by `HyperBitT` behaves as described by Theorem 3 and to set parameter values that keep the error low. As with any scientific study, our confidence in the result grows with the number of experiments that validate it, so we can only give an initial indication. (For example, practitioners have confidence in a similar hypothesis for `HyperLogLog` because it has been used in a wide variety of practical situations for years.)

# of 0s in `sketch[]`	$M\beta$	228	253	257	261	265
estimated cardinality	$2^T M \ln(1/\beta)$	393,773	366,498	362,386	358,338	354,351
estimated relative accuracy	$c_\beta/\sqrt{M}$	3.9%	3.9%	3.9%	3.9%	3.9%
actual relative accuracy		6.9%	0.5%	1.6%	2.7%	3.8%

Table 2 Since it is based on hash values, `HyperBitT` produces a different result every time it is run. The following table shows the result of five consecutive runs of `HyperBitT` for our sample web log with these parameter values. The last line compares the estimated cardinality with the actual value 368,217. Since our estimate of the standard error is conservative (c_β is usually smaller than 1.5), four of the five runs produced estimates well within the desired 5%. Since the distribution is Gaussian, the outlier in the first experiment is not unexpected.

The hypothesis rests on three main assumptions. First, we assume that the data we have and that the hash functions we use have the idealized properties stipulated in the analysis, or that deviations from this ideal are relatively insignificant. Second, we assume that the second hash function splits the stream into each substream with equal probability, or that deviations from this ideal are relatively insignificant. Third, we assume that deviations from approximations in the analysis are relatively insignificant.

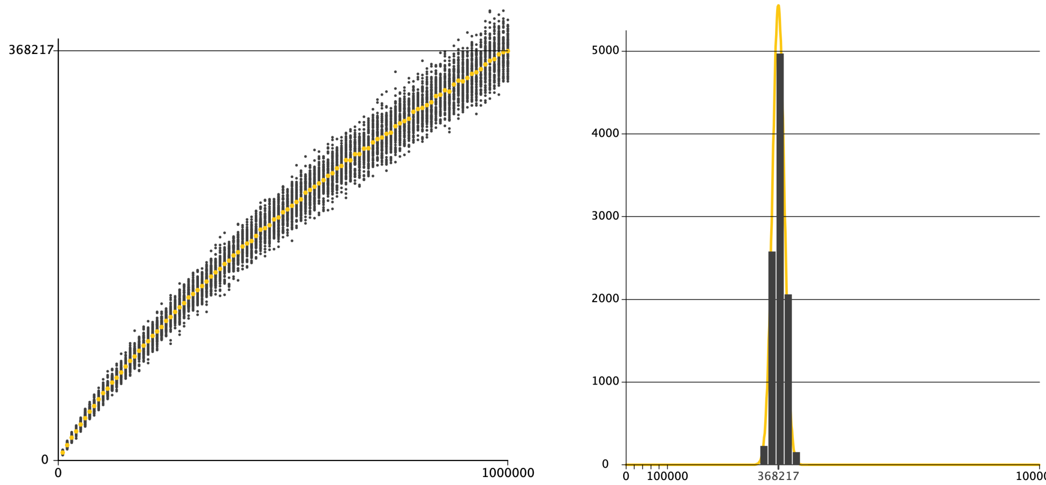
For example, suppose that we wish to use `HyperBitT` to estimate the number of distinct strings in the web log described in Section 1. To do so, we need to specify the values of the two parameters: M (the number of bits of memory we need to use to achieve the accuracy that we want) and T (where $2^T M$ is our rough guess of the cardinality).

First, we choose the value of M. As an example, suppose that we are looking for an accurate answer, say with 5% relative error. Referring to Figure 1, if β is in the range $(.043, .541)$, then $c_\beta < 1.5$ and $M = 1024$ will do the trick, because $1.5/\sqrt{1024} \doteq .0469$. This is a conservative choice because c_β is usually much smaller than 1.5 in that range.

Next, we choose the value of T. Suppose we decide that it is a reasonable guess that the unique values comprise somewhere between 20% and 80% of the stream (a rather wide range). This leads to the choice $T = 8$ because $M2^T \ln(1/\beta)$ is between 161,044 and 824,851 (and $c_\beta < 1.5$) when β is between .541 and .043.

Table 2 shows the experimental results that constitute a quick validation check. Figure 2 describes two experiments that each run it *10 thousand* times, which both are strong evidence of the validity of our analysis and our hypotheses about the performance of `HyperBitT`.

It is important to reiterate that `HyperBitT` is *not* a streaming algorithm. For example, it could not be used without some periodic adjustments for our web log example, where the log may be monitored for weeks, months, or even years, and therefore could consists of billions or trillions of strings or more. But there are many situations where `HyperBitT` may be useful because the estimate need not be very accurate and there are reasonable approaches to coming up with one. In a database or similar application, one might take a random sample. In a web log or similar application, one might take a small sample from initial values, or run multiple offsetting streams, using the estimate from one as the rough guess for another. For example, in protecting against a denial-of-service attack, the whole point might be to just set off an alarm when the cardinality deviates significantly from an expected range.



(a) 100 trials every 10,000 inputs up to 1 million **(b)** 10,000 trials with 1 million inputs

Figure 2 Results of estimating cardinalities in a web log, each with 10,000 trials. In Figure 2(a) `HyperBitT` was run 100 times for the first 10,000, 20,000, 30,000, ... items in the log, up to 1 million. Each grey dot shows the result of one experiment and the colored dots are the average of the values for each set of 100 experiments. A black line that shows the actual number of distinct items in the stream is completely hidden by the colored dots. The histogram in Figure 2(b) plots the estimates returned by `HyperBitT` for 10,000 runs on the first 1 million strings in the web log. The distribution matches a Gaussian, centered on the true number of distinct values, with relative standard deviation about $1.25/\sqrt{M} \doteq 0.039$ (plotted in color), thus validating Theorem 3 and our hypothesis that the estimated cardinality is likely to be within within 5% of the true value.

3 HyperBitBit and HyperBitBitBit

In this section, we describe variants of the algorithm that can *adapt* as the number of unique values grows, by making `T` a *variable* and then increasing it as needed.

Obviously, `T` needs to increase when the sketch becomes nearly full of `1`s. The first approach that comes to mind is to plan to increase `T` by one when the sketch becomes nearly full and to maintain a second sketch with `1` bits corresponding to whether or not an item with at least `T+1` trailing `1`s has been seen. Then, when the sketch is nearly full, we can increment `T` and replace the first sketch with the second one. But then we need to replace the second sketch. We could use a third sketch (and we will, when M is not small), but then do we need a fourth sketch? Moreover, when the sketch for `T` is nearly full of 1s, so is the sketch for `T+1`, so incrementing `T` by 1 does not help much.

So we want to increment `T` by *more* than one. But by how much? Recall that our analysis indicates that the accuracy degrades as the number of `0`s in the sketch grows, and incrementing `T` corresponds to increasing the number of `0`s. Eventually we can stop when we encounter sketches that are all `0`s, but we are faced with a delicate balance between the amount by which we increment `T` and the number of sketches we might need. Theorem 3 gives us precisely the information we need to make an intelligent choice.

To fix ideas, take $M = 64$ and suppose that we consider the sketch to be "nearly full" when 62 of its bits are 1 (and therefore $\beta = 2/62 \doteq 0.032$). Now, we want to choose an increment `i` for `T`—we will maintain a second sketch for `T+i` and increment `T` by `i` when the sketch for `T` is 97% full of `1`s. Our goal is to choose `i` such that we do not need to maintain a third sketch.

i	0	1	2	3	4	5	6	7	8
$\beta_i = \exp(-\ln(1/\beta)/2^i)$	.03	.17	.42	.64	**.80**	.90	.95	.97	**.99**

Table 3 Fraction of zeros in sketches for `T+i` when the sketch for `T` is 97% full. The sketch for `T` is 3% 0s, the sketch for `T+4` is 80% 0s and the sketch for `T+8` is 99% 0s.

Let β_i be the fraction of 0s in the sketch for `T+i`. Because the estimated value of N does not change, we must have $\ln(1/\beta) = \ln(1/\beta_i)/2^i$. Solving for β_i gives $\beta_i = \exp(-\ln(1/\beta)/2^i)$. Table 3 shows these values for possible increments up to 8 (after that point, the sketches are increasingly likely to be all 0s).

Specifically, Table 3 tells us something very important: for increments 4 or greater, there is no need to maintain a third sketch, *because it would be nearly all zeros*. With our choice to increment `T` by 4 when the sketch is 97% `0`s, we know that at that time the sketch for `T+4` is about 80% 0s and the sketch for `T+8` would be about 99% 0s, so we can increment `T`, update our sketch for `T` using the sketch for `T+4`, and set the sketch for `T+4` to all 0s. We may be ignoring a few `1`s that would be in the sketch for `T+8` had we maintained it, but the likelihood that ignoring them would noticeably affect the final estimate is very small. If we want to be very conservative, we could maintain the indices of these `1`s, at a very small (if not negligible) extra cost, but few practitioners would bother.

This discussion brings us to `HyperBitBit64` (Algorithm 2). It uses $M = 64$, maintains two sketches, increments `T` by 4, and updates the sketches when the first sketch becomes 97% full of `1`s. The implementation also illustrates how to use 64-bit words for the sketches, which eliminates the overhead of maintaining bit arrays and leads to very simple and efficient code in typical programming environments, even machine language. For clarity, Algorithm 2 uses the call `p(sketch)` to count the number of 1s in the sketch. If this is not available as an atomic operation, one might choose the alternative of counting as the bits are set, as described in Appendix B and illustrated in the code at `https://github.com/robert-sedgewick/hyperbitbit`.

From the above discussion, it is reasonable to hypothesize that when Algorithm 2 terminates, `sketch0` *is the same as the sketch when Algorithm 1 is used with the current value of* `T`. In other words, Theorem 3 applies throughout. As we saw in Table 3, just before incrementing `T`, `sketch0` has about 97% `1`s and `sketch1` has about 20% `1`s. Thus, the fraction of `0`s in the sketches stays in the range $.03 < \beta < .80$, so the value of c_β is in the flat part of its curve (see Figure 1)—it is always less than 2.25 with average value about $\frac{1}{.77}\int_{.03}^{.80} c_\beta \mathrm{d}\beta \doteq 1.48$. This is conservative—the number of `0`s quickly increases when it is small, so c_β is more often than not less than this average.

The end result is that `HyperBitBit64` is a true streaming algorithm that uses just 128 bits (plus six bits for `T`) to achieve an expected standard error which is usually lower than $1.48/\sqrt{64} \doteq 18.5\%$ even for streams having billions or trillions or more distinct items. As we will see in Section 5, this accuracy is substantially better than that achieved by `HyperLogLog` for the same number of bits. The cost of processing each element is the cost of hashing plus a few machine-language instructions. In applications where 18.5% accuracy suffices (and developing a rough guess that would enable use of `HyperBitT` is infeasible), `HyperBitBit64` is likely to be the method of choice because of these low costs. For example, it would be quite useful in an application where maintaining large number of different cardinality counters are needed, each responding to some different filter of the input stream.

For larger values of M (say 128 or 256) we can implement `HyperBitBit` with a bit array (perhaps implemented with an array of 64-bit integers as described in Appendix B) and do

Algorithm 2 `HyperBitBit64`.

```
public static int estimateHBB64(Iterable<String> stream)
{
   int T = 1;
   int M = 64;
   long sketch0;
   long sketch1;
   for (String s : stream)
   {
      long x = hash1(s);                              // 64-bit hash
      int  k = hash2(s, M);                           // 6-bit hash
      if (r(x) > T)   sketch0 = sketch0 | 1L << k;  // >T trailing 1s?
      if (r(x) > T+4) sketch1 = sketch1 | 1L << k;
      if (p(sketch0) > .97*M)                         // >62 1s?
      { sketch0 = sketch1; sketch1 = 0; T += 4; }
    }
    double beta = 1.0 - 1.0*p(sketch0)/M;             // fraction of 0s
    return (int) (Math.pow(2, T)*M*Math.log(1.0/beta));
}
```

even better. Specifically, it makes sense to *set the cutoff to increment `T` when the relative standard error for the new value is equal to the current relative standard error.* That is, with $a = \ln(1/\beta)$ and $c(a) = \sqrt{e^a - 1}/a$, we increment `T` by 4 when $c(a) = c(a/16)$. The solution to this equation is $a = \ln(1/\beta) \doteq 4.41$ so $\beta = e^{-a} \doteq .012$ That is, we should increment T by 4 and update the sketches when `sketch0` has $.988M$ `1` bits. At that point, the proportion of 0s in the sketch for `T+4` will be about $e^{-a/2^4} \doteq .75912$. The proportion of 0s in the sketch for `T+8` would be about $e^{-a/2^8} \doteq .983$, so we are ignoring (2, 4, 9) 1 bits for (128, 256, 512) respectively, which is likely tolerable. The fraction of `0`s in the sketches stays in the range $.012 < \beta < .759$, so the value of c_β is always less than 2.05 with average value about $\frac{1}{.747}\int_{.012}^{.759} c_\beta \mathrm{d}\beta \doteq 1.46$.

HyperBitBitBit

For even larger values of M, we can go to a third sketch, marking the subarrays with at least `T`, `T+4`, and `T+8` trailing `1`s and define `HyperBitBitBit` in a straightforward manner. The implementation is omitted because we present a significant improvement in Section 4. The proportion of 0s in the sketch for `T+12` would be about $e^{-a/2^{12}} \doteq .996$, so we are ignoring (1, 2, 4) 1 bits for (1024, 2048, and 4096) respectively, again likely tolerable.

As just noted for `HyperBitBit`, the fraction of `0`s in the sketches stays in the range $.012 < \beta < .759$, so the value c_β is always less than 2.05 with average value about $\frac{1}{.747}\int_{.012}^{.759} c_\beta \mathrm{d}\beta \doteq 1.46$. In summary, `HyperBitBitBit` is a true streaming algorithm, effective for M up to at least 4096, that uses $3M$ bits and achieves relative standard error of about $1.46/\sqrt{M}$.

4 HyperTwoBits

Remarkably, we can produce the same result as `HyperBitBitBit` but using just $2M$ bits. The trick is to note that if a bit is set in the sketch for `T+4`, the bit in the corresponding position in the sketch for `T` must be set, and if a bit is set in the sketch for `T+8`, the bits in the corresponding positions in the sketches for both `T+4` and `T` must be set. This observation means that we can represent the three sketches with an array of two-bit values that encode

Algorithm 3 `HyperTwoBits`.

```
public static int estimateHTB(Iterable<String> stream, int M)
{  // for M = 1024, 2048, or 4096
   int T = 1;
   twobit[] sketch = new twobit[M];
   for (String s : stream)
   {
      long x = hash1(s);       // 64-bit hash
      int  k = hash2(s, M);    // (lg M)-bit hash
      if (r(x) >= T  ) if (sketch[k] < 1) sketch[k] = 1;
      if (r(x) >= T+4) if (sketch[k] < 2) sketch[k] = 2;
      if (r(x) >= T+8) if (sketch[k] < 3) sketch[k] = 3;
      if (pnz(sketch) > .988*M)
      {
         T = T+4;
         for (int i = 0; i < M; i++)
            if (sketch[i] > 0) sketch[i]--;
      }
   }
   double beta = 1.0 - 1.0*pnz(sketch)/M;
   return (long) (Math.pow(2, T)*M*Math.log(1/beta));
}
```

in binary the number of `1`s in each position in the three sketches in `HyperBitBitBit`, as shown in this example:

	sketch for `T`	`1111111111101110111111111111111111111110111011111110011111111 1`
before	*sketch for* `T+4`	`0001001110100000000000000100001100101100000001111000010000000000`
	sketch for `T+8`	`0000000100000000000000000000000000010000000011000010000000000`
	two-bit values	`1112112321201110111111121111221121231011102233110031111111`

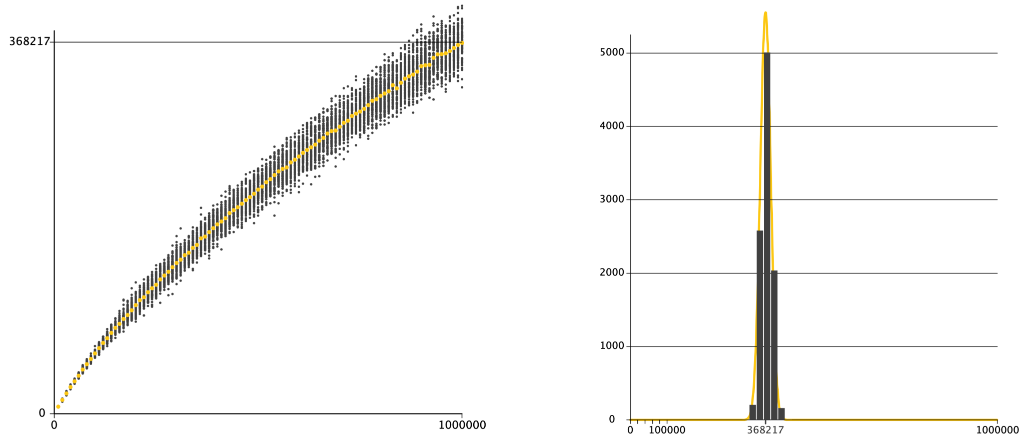
	sketch for `T`	`0001001110100000000000000100001100101100000001111000010000000000`
after	*sketch for* `T+4`	`0000000100000000000000000000000000010000000011000010000000000`
`T+=4`	*sketch for* `T+8`	`0000000000000000000000000000000000000000000000000000000000000`
	two-bit values	`0001001210100000000000000101001100101200000001122000020000000000`

Maintaining this array while streaming is simple: for each data item, we identify its stream and set its value as appropriate. Then when the number of nonzero values reaches the threshold, we increment `T` by 4 and simply *decrement the nonzero values in the array.*

From this description, the implementation in Algorithm 3 is immediate. For clarity, we use a `twobit[]` type to describe the algorithm—although no programming languages support an explicit `twobit[]` type, the abstraction can be implemented with shifting and masking on arrays of integers, an amusing exercise in bit logic (see Appendix B). For clarity, we use a method `pnz()` to count the nonzero entries in the array—its implementation is omitted because it is better to maintain the count dynamically (also see Appendix B).

In summary, `HyperTwoBits` is a true streaming algorithm, effective for M up to at least 4096, that uses $2M$ bits and achieves relative standard error of about $1.46/\sqrt{M}$. As described in Appendix B, it can be implemented such that processing each item in a stream requires only a few machine-language operations.

Figure 3 presents the results of two experiments for Algorithm 3 corresponding to those presented for Algorithm 1 in Figure 2, which validate our hypothesis that the relative accuracies of the algorithms are comparable and are strong evidence of the utility of the algorithm in practice.



(a) 100 trials every 10,000 inputs up to 1 million **(b)** 10,000 trials with 1 million inputs

Figure 3 Results of estimating cardinalities in a web log using Algorithm 3 with $M = 1000$, for comparison with Figure 2 (where the details of the experiments are described). Given the same inputs (and the same random numbers), the figures for `HyperBitBitBit` would be identical.

5 Performance comparisons

Comparing the performance of our algorithms with each other and with cardinality estimation algorithms in the literature needs to be done carefully for several reasons.

First, many papers from the theoretical computer science literature study algorithms implemented in pseudocode (or just described in English). While these papers often introduce interesting ideas, they cannot be evaluated as solutions to the practical cardinality estimation problem for two reasons. First, the methods described have never been implemented (and are sufficiently complicated that implementing them is not likely to be worthwhile) so the time required to process each item while streaming cannot be determined. Second, the analyses generally define complexity results that use O-notation and are not sufficiently precise to compare the relative accuracy with other methods.

Second, even among methods that have been implemented and tested, practitioners might prefer algorithms that are much simpler to implement and maintain over more complicated methods that perform slightly better. Some methods are sufficiently complicated to implement that practitioners might shy away from (or may not be able to afford) actually doing so. For example, `HyperLogLog` is easy to implement with 8-bit bytes, but 6-bit bytes are sufficient. Implementing a 6-bit byte array with arrays of 64-bit words is not difficult, but may be too cumbersome from the point of view of some practitioners.

Third, many papers use the parameter M to count the number of bytes or words (of varying length) of memory used. Proper comparisons necessitate counting *total number of bits* of memory in all cases. As an extreme example, suppose that two algorithms achieve standard error $2/\sqrt{M}$ but one uses M bits and the other uses M 64-bit words. The first is *eight times* more accurate for a given number of bits of memory. In general, if we know that the accuracy of an algorithm is $c/\sqrt{M}$ and that it stores Mb bits, we express the accuracy in terms of $M^\star$, the total number of bits used, or $c\sqrt{b}/\sqrt{M^\star}$. Inverting this equation gives the number of bits needed to achieve a given accuracy $x : M^\star = b(c/x)^2$. We ignore relatively inconsequential small fixed costs such as the six bits required to store the value of `T` in our adaptive algorithms.

Fourth, few papers actually *prove* anything about the distribution of the reported values, with the notable exception of [13]. Typically, normality is instead presented as a reasonable hypothesis, which may often be the case, but our proof of asymptotic normality of the reported cardinalities is significant.

Fifth, the accuracy of our algorithms depend on the coefficient c_β of $1/\sqrt{M}$ in the relative standard error, which varies. We use the average value of c_β over the interval of values β might take on during the execution of the algorithm. For `HyperBitT` we (somewhat arbitrarily) use the interval where $c_\beta < 1.5$; our other algorithms calculate an appropriate interval. As we have noted, the curve in Figure 1 is quite flat, so it is likely that the value encountered in practice is smaller than the value cited.

Sixth, it is important to remember that we are dealing with random fluctuations and approximate analyses. It may be tempting to use more precision, but any differences indicated would not be noticed in practice. For example, one might conclude that `HyperLogLog` with 6-bit bytes should be very slightly better than `LogLog` with 6-bit bytes because its standard error of $1.02/\sqrt{M}$ is very slightly better than $1.05/\sqrt{M}$, but it would be extremely challenging to develop experimental validation of that hypothesis.

					$M^\star = b(c/x)^2$ bits needed for		$c\sqrt{b}/M^\star$ accuracy with	
algorithm	*range for M*	*b*	*c*	$c\sqrt{b}$	*2%*	*20%*	*128 bits*	*8K bits*
Adaptive sampling[5]		64	1.20	9.60	230400	2304	85%	10.6%
Prob. counting[6]		64	0.78	6.24	97344	973	55%	6.9%
`LogLog`[4]		6	1.05	2.57	16538	165	23%	3.5%
`HyperLogLog8`[7]		8	1.04	2.94	21632	216	26%	3.3%
`HyperLogLog`[7]		6	1.02	2.55	16224	162	23%	2.8%
`ExtHyperLogLog`[16]		7	0.88	2.33	13552	136	21%	2.6%
`HyperBitT`		1	1.32	1.32	4356	44	12%	1.5%
`HyperBitBit64`	64	2	1.48	2.09	—	128	19%	—
`HyperBitBit`	64–512	2	1.46	2.06	—	128	18%	—
`HyperBitBitBit`	128–4096	3	1.46	2.53	15987	128	22%	2.8%
`HyperTwoBits`	128–4096	2	1.46	2.06	10658	128	18%	2.3%

Table 4 Performance of cardinality estimation algorithms

With all these caveats, Table 4 presents a comparison of the algorithms we have discussed. Our simplest and perhaps most useful implementation is `HyperBitBit64`, which achieves 18.5% accuracy on a stream on any length with just two 64-bit words and can be implemented with a few dozen machine instructions. `HyperBitT` is the best by far when starting with a rough estimate is feasible. More generally, if a straightforward and easy to maintain implementation is desired, `HyperBitBit` and `HyperBitBitBit` are arguably simpler than the 8-bit version of `HyperLogLog` and substantially more efficient. If a careful implementation with improved efficiency is desired, `HyperTwoBits` is substantially more efficient than the 6-bit version of `HyperLogLog`. In both cases our algorithms provide much better accuracy for the same number of bits and use two-thirds as many bits to achieve the same accuracy.

6 Further Improvements

We conclude by briefly mentioning some opportunities that may lead to variants of our algorithms that may be worthy of study in various particular situations.

- *Sparse arrays.* Precise characterization of the "transition cost" just after incrementing `T` (when the sketches are mostly `0`s) may lead to slight performance improvements.
- *Use two sketches.* The second sketch contains information that may lead to a more accurate estimate. Analyzing this effect is tractable, but not likely to improve the accuracy by more than a percentage point or two.
- `HyperThreeBits`. Using 3-bit counters instead of the 2-bit counters in `HyperTwoBits` allows implementation of seven layers of bit arrays and may be useful for specialized applications needing very high accuracy (requiring huge values of M) for the kinds of truly huge streams seen in modern computing.
- `HyperBit`. We have studied many approaches to modifying `HyperBitT` to just increment `T`, reset the sketch to `0`s, and then characterizing the error due to the "transition cost". Despite some promising empirical results, the problem of developing a mathematical model admitting proper comparison of such an algorithm with the ones described here remains open.
- *Mergeability.* Many applications can benefit from being able to merge sketches built from two different streams. Our sketches are not difficult to merge, as indicated by the following argument for `HyperBitBit`. A sketch is a triple (`T`, `sketch0`, `sketch1`). To merge $(\mathtt{T}_A, \mathtt{sketch0}_A, \mathtt{sketch1}_A)$ with $(\mathtt{T}_B, \mathtt{sketch0}_B, \mathtt{sketch1}_B)$ consider the following three cases:
  - If $\mathtt{T}_A = \mathtt{T}_B = \mathtt{T}$ use $(\mathtt{T}, \mathtt{sketch0}_A|\mathtt{sketch0}_B, \mathtt{sketch1}_A|\mathtt{sketch1}_B)$.
  - If the values of `T` differ by 8 or more, use the larger value and its sketches.
  - Otherwise, suppose wlog that $T_A = T_B + 4$. Use $(\mathtt{T}_A, \mathtt{sketch0}_A|\mathtt{sketch1}_B, \mathtt{sketch1}_A)$.

  In the first and third cases, check whether the first sketch is nearly full. If so, increment `T` (by 4) and update the sketches as usual. This result is not precisely the same as if the two streams had actually been merged, but the difference is likely acceptably small in many practical situations. The argument for `HyperBitT` is similar, but simpler; the argument for `HyperBitBitBit` is similar, but more complicated.

7 Acknowledgements

We would like to thank Martin Pépin and two anonymous reviewers for their helpful comments on our initial submission. We would also like to thank our colleagues, Conrado Martínez, Sampath Kannan, and Val Tannen, for their interest and feedback; and our students, Alex Iriza and Alex Baroody for their discussions and implementation work on earlier versions of these algorithms. Finally, we would like to thank the editors, Cécile Mailler and Sebastian Wild, for their service to the community.

References

1. Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 20–29. ACM, 1996. doi:10.1145/237814.237823.
2. A. D. Barbour, Lars Holst, and Svante Janson. *Poisson Approximation*. Oxford University Press, 1992.
3. Kai-Min Chung, Michael Mitzenmacher, and Salil P. Vadhan. Why simple hash functions work: Exploiting the entropy in a data stream. *Theory Comput.*, 9:897–945, 2013. URL: https://doi.org/10.4086/toc.2013.v009a030, doi:10.4086/TOC.2013.V009A030.
4. Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities (extended abstract). In Giuseppe Di Battista and Uri Zwick, editors, *Algorithms - ESA 2003, 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003, Proceedings*, volume 2832 of *Lecture Notes in Computer Science*, pages 605–617. Springer, 2003. doi:10.1007/978-3-540-39658-1_55.
5. Philippe Flajolet and G. Nigel Martin. Probabilistic counting. In *24th Annual Symposium on Foundations of Computer Science, Tucson, Arizona, USA, 7-9 November 1983*, pages 76–82. IEEE Computer Society, 1983. doi:10.1109/SFCS.1983.46.
6. Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985. doi:10.1016/0022-0000(85)90041-8.
7. Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In Philippe Jacquet, editor, *AofA 07— Proceedings of the 2007 Conference on Analysis of Algorithms, Juan-les-pins, France, June 18-22, 2007, DMTCS Proceedings volume AH*, pages 127–146. DMTCS, 2007. URL: https://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf.
8. Allan Gut. *Probability: A Graduate Course (2nd edition)*. Springer Texts in Statistics, 75, 2013.
9. Stefan Heule, Marc Nunkesser, and Alexander Hall. Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In Giovanna Guerrini and Norman W. Paton, editors, *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pages 683–692. ACM, 2013. doi:10.1145/2452376.2452456.
10. Piotr Indyk and David P. Woodruff. Tight lower bounds for the distinct elements problem. In *44th Symposium on Foundations of Computer Science (FOCS 2003), 11-14 October 2003, Cambridge, MA, USA, Proceedings*, pages 283–288. IEEE Computer Society, 2003. doi:10.1109/SFCS.2003.1238202.
11. Daniel M. Kane, Jelani Nelson, and David P. Woodruff. An optimal algorithm for the distinct elements problem. In Jan Paredaens and Dirk Van Gucht, editors, *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, June 6-11, 2010, Indianapolis, Indiana, USA*, pages 41–52. ACM, 2010. doi:10.1145/1807085.1807094.
12. Matti Karppa and Rasmus Pagh. Hyperlogloglog: cardinality estimation with one log more. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 753–761, 2022.
13. Jérémie Lumbroso. An optimal cardinality estimation algorithm based on order statistics and its full analysis. *Discrete Mathematics & Theoretical Computer Science*, AM, 2010. URL: https://doi.org/10.46298/dmtcs.2780.
14. Jérémie Lumbroso. How Flajolet processed streams with coin flips. *CoRR*, abs/1805.00612, 2018. URL: http://arxiv.org/abs/1805.00612, arXiv:1805.00612.
15. Jérémie Lumbroso and Conrado Martínez. Affirmative Sampling: Theory and Applications. In Mark Daniel Ward, editor, *33rd International Conference on Probabilistic, Combinatorial and Asymptotic Methods for the Analysis of Algorithms (AofA 2022)*, volume 225 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:17, Dagstuhl, Germany, 2022.

Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.AofA.2022.12`, `doi:10.4230/LIPIcs.AofA.2022.12`.

16 Tal Ohayon. Extendedhyperloglog: Analysis of a new cardinality estimator. *CoRR*, abs/2106.06525, 2021. URL: `https://arxiv.org/abs/2106.06525`, `arXiv:2106.06525`.

17 Seth Pettie and Dingyu Wang. Information theoretic limits of cardinality estimation: Fisher meets shannon. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 556–569, 2021.

18 Robert Sedgewick and Philippe Flajolet. *An introduction to the analysis of algorithms, second edition*. Addison-Wesley-Longman, 2013.

19 Dingyu Wang and Seth Pettie. Better cardinality estimators for hyperloglog, pcsa, and beyond. In Floris Geerts, Hung Q. Ngo, and Stavros Sintos, editors, *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2023, Seattle, WA, USA, June 18-23, 2023*, pages 317–327. ACM, 2023. `doi:10.1145/3584372.3588680`.

A Proof of Lemma 1

Suppose that $X_n \geq 0$ are random variables and a_n, b_n, and σ^2 numbers such that, as $n \to \infty$, we have $a_n \to a > 0$, $b_n \to 0$, and $(X_n - a_n)/b_n \xrightarrow{d} \mathbb{N}(0, \sigma^2)$. If f is a continuously differentiable function on $(0, \infty)$ with $f'(a) \neq 0$, then $(f(X_n) - f(a_n))/b_n \xrightarrow{d} \mathbb{N}(0, f'(a)^2\sigma^2)$

Proof. This is well known, but we include this proof for completeness.

By the mean value theorem,

$$\frac{f(X_n) - f(a_n)}{b_n} = f'(X_n^*)\frac{X_n - a_n}{b_n} \tag{7}$$

for some X_n^* with $X_n \leq X_n^* \leq a_n$ or $a_n \leq X_n^* \leq X_n$. Since $(X_n - a_n)/b_n \xrightarrow{d} \mathbb{N}(0, \sigma^2)$ and $b_n \to 0$, we have $X_n - a_n \xrightarrow{p} 0$. Furthermore, $a_n \to a$, and hence $X_n \xrightarrow{p} a$. Consequently, also $X_n^* \xrightarrow{p} a$. Thus, since f' is continuous, $f'(X_n^*) \xrightarrow{p} f'(a)$. The result follows from (7) and the assumption. ◀

B Implementation details

The abstract operations we have used in expressing our algorithms can be implemented efficiently on most computers, as described in the following paragraphs. Our code makes liberal use of Java's left and right shift operators `>>` and `>>` and bitwise logical operations (`&`, `|`, and `~`) for bitwise (*AND*, *OR*, and *NOT*) respectively. Algorithm 4 is a full low-level implementation of `HyperBitBit64` that solves the practical cardinality estimation problem.

Sketches

As we have noted, few programming languages support an efficient `bit[]` type (even Java does not guarantee that boolean arrays use one bit per entry). As we saw in `HyperBitBit64` (Algorithm 2), shifting and masking on 64-bit long values is an easy way to implement the abstraction. For larger values of `M`, we use arrays of 64-bit values. In Java, for example, we maintain the sketch as an array of `long` values:

```
long[] sketch = new long [M/64];
```

Then the Java code

```
if ((sketch[k/64] & (1L << (k % 64))) != 0)
```

tests whether the `k`th bit in the sketch is `1` and the Java code

```
sketch[k/64] = sketch[k/64] | (1L << (k % 64));
```

sets the `k`th bit in the sketch to `1`.

Trailing 1s

The key abstract operation in our implementations involves computing the function `r(x)`, so that we can test whether a 64-bit value `x` has at least `T` trailing 1s. Rather than maintaining the parameter T, we maintain $U = 2^T$. The reason for doing so is that the value `U-1` has `T` trailing 1s, which enables us to test whether a value `x` has at least `T` trailing ones with the bitwise logical operation `(x & (U-1)) == (U-1)`, which is easy to implement with a few machine-language instructions.

Population count

The second abstract operation in our implementations is the function `p(x)`, the so-called "population count"—the number of `1` bits in a binary value. This function has a long and interesting history, but, for our purposes, it is easy to avoid, by maintaining a count of the number of `1` bits in the sketches, incrementing when each bit is set.

Two-bit counters

Again, we use shifting and masking on arrays of 64-bit `long` values. We keep one `long` array `s1` for the more significant bit and a second `long` array `s0` for the less significant bit. To make the code more readable, we define the following methods to test and set the bit corresponding to bit `k`:

```
public static long val(long[] s1, long[] s0, int k)
{ return 2*((s1[k/64] >> (k % 64)) & 1L)+((s0[k/64] >> (k % 64)) & 1L); }
```

```
public static void setval(long[] s1, long[] s0, int k, long v)
{
    s1[k/64] = (s1[k/64] & ~(1L << (k % 64))) | ((v/2) & 1L) << (k % 64);
    s0[k/64] = (s0[k/64] & ~(1L << (k % 64))) | (v & 1L) << (k % 64);
}
```

In a tightly efficient or machine-code version, this code would be used inline.

The final abstract operation to consider is to decrement all the non-zero counters. Consider the following table, which gives all possibilities for a given bit position, where $s_1 s_0$ is the value before incrementing and $t_1 t_0$ is the value after decrementing.

before			*after*		
value	`s1`	`s0`	*value*	`t1`	`t0`
0	0	0	0	0	0
1	0	1	0	0	0
2	1	0	1	0	1
3	1	1	2	1	0

Considering these as truth tables on boolean values, it is easy to check that `t1 = s1` *AND* `s0` and `t0 = s1` *AND NOT* `s0`. Furthermore, we can eliminate the temporary variables by doing the operations in the order `s0 = s1` *AND NOT* `s0` and then `s1 = s1` *AND NOT* `s0`. Implementing these operations with bitwise operations on our arrays of `long` values is straightforward.

Algorithm 4 `HyperTwoBits` (full low-level implementation).

```
public static int estimateHTB(String[] stream, int N, int M)
{
   int U = 2;
   double alpha = .988;
   long[] s0 = new long [M/64];
   long[] s1 = new long [M/64];
   int count = 0;
   for (int i = 0; i < N; i++)
   {
      long x = hash1(s);        // 64-bit hash
      int  k = hash2(s, M);     // (lg M)-bit hash
      if ((x & (U-1)) == (U-1)) count++;
      if ((x & (U-1)) == (U-1))
         if (val(s1, s0, k) < 1) setval(s1, s0, k, 1);
      if ((x & (16*U-1)) == (16*U-1))
         if (val(s1, s0, k) < 2) setval(s1, s0, k, 2);
      if ((x & (256*U-1)) == (256*U-1))
         if (val(s1, s0, k) < 3) setval(s1, s0, k, 3);
      if (count >= alpha*M)
      {
         for (int j = 0; j < M/64; j++)
         { s0[j] = s1[j] & ~s0[j]; s1[j] = s1[j] & ~s0[j]; }
         count = 0;
         for ( int j = 0; j < M; j++)
           if (val(s1, s0, j) > 0) count++;
         U = 16*U;
      }
   }
   double beta = 1.0 - 1.0*count/M;
   double bias = Math.log(1.0/beta);
   return (int) (U*M*bias);
}
```