



Bit-array-based alternatives to HyperLogLog

Svante Janson^a, Jérémie Lumbroso^b, Robert Sedgewick^{c,*}

^a Department of Mathematics, Uppsala University, Uppsala, Sweden

^b Computer and Information Science Department, University of Pennsylvania, Philadelphia, PA, USA

^c Department of Computer Science, Princeton University, Princeton, NJ, USA

ARTICLE INFO

Section Editor: Paul G. Spirakis

Handling Editor: Sebastian Wild

Keywords:

Cardinality estimation

Sketching

HyperLogLog

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 small 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. Performance hypotheses are validated through experiments using a realistic input stream, with the general conclusion that our algorithms are significantly more accurate than HyperLogLog when using the same amount of memory, and they use significantly less memory than HyperLogLog to achieve a given accuracy.

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 found in the stream is useful for many purposes.

One classic application is for 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 different visitors in a given time period might be an indication that the site is under a denial-of-service attack.

Another classic application is for 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 items in memory, which is far too high a cost to be useful in typical applications. In

* Corresponding author.

E-mail address: rs@cs.princeton.edu (R. Sedgewick).

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

s	x	k	r(x)	sketch[]
81.95.186.98.freenet.com.ua	111...1011110011101011	7	2	00000001
lsanica.dsl-w.verizon.net	011...0001111110100000	3	0	00000001
117.222.48.163	110...0000111011101101	6	1	00000001
1.23.193.58	100...1000011101100011	4	2	00001001
188.134.45.71	101...0101101000111001	5	1	00001001
gsearch.CS.Princeton.EDU	010...1000011010000100	2	0	00001001
81.95.186.98.freenet.com.ua	011...1110000111010000	3	0	00001001
81.95.186.98.freenet.com.ua	111...1011110011101011	7	2	00001001
1.23.193.58	000...1111101011100111	0	3	10001001
lnse3.cht.bigpond.net.au	111...1011011101001110	7	0	10001001
117.211.88.36	000...0100010010111010	0	0	10001001
1.23.193.58	000...1111101011100111	0	3	10001001
lsanica.dsl-w.verizon.net	011...0001111110100000	3	0	10001001
81.95.186.98.freenet.com.ua	111...1011110011101011	7	2	10001001
gsearch.seas.upenn.edu	000...1011010100001000	0	0	10001001
109.108.229.102	010...1010110110011111	2	5	10101001
msnbot.search.msn.com	001...0111001001101100	1	0	10101001

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 different 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 [1][2] it has been known that we actually can get by with a surprisingly small amount of processing time and memory space. To be more specific, 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 small constant multiple of its size.
- The space used is small, and independent of the length of the stream.
- The estimate is expected to be very close to 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. In the context of this paper, we refer to an algorithm satisfying the first three constraints as a “streaming algorithm.”

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 [3] [4] [5] [6]. 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 and averages the counts in the substreams.
- 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, an important and significant accomplishment [7][8][9]. However, these papers lack implementations, likely because the implied constants in the proofs are much too large for the methods to be viable in practice. Some also use theoretical devices that would never be used in practice, like reprocessing the stream in the (low probability) event of a bad estimate.

Papers in the theory literature also typically make the implicit assumption that strong assumptions on the hash functions are necessary. This dates back to the 1996 paper by Alon, Matias and Szegedy [7] where the work of Flajolet and Martin in [2] is dismissed with a mischaracterization that leaves the incorrect impression that the algorithms rely on the availability of hash functions with very strong random properties. To the contrary, Flajolet and Martin simply observed that using such hash functions would be prohibitively

expensive, so they invented stochastic averaging (which is extremely efficient), did a precise analysis to develop the hypothesis that using it would have similar performance to the idealized algorithm in practice, and ran experiments to validate that hypothesis. And this conclusion has been further validated for decades by countless implementations and applications (see, for example, [5]).

Using idealized models to state and prove theorems that are sufficiently precise that they can lead to practical algorithms and hypotheses about performance that can be validated through experimentation is the basis of algorithm science, dating back to Knuth's pioneering study of linear probing in 1963, which precisely characterizes the performance of the algorithm under the assumption that all hash sequences are equally likely (see [10]). It is interesting to note that not even Knuth explicitly states that his theorem is a reasonable basis for forming hypotheses about real-world performance. The fact remains that any practical application of hashing, however perfect in theory, must assume, at least, that random bits exist, and therefore rests on some hypothesis connecting the theory and the real world. See, for example, [11] for further discussion of this issue. Moreover, a precise mathematical model is a necessity in situations where multiple parameters are involved. Choosing the best values of parameters can be done within the model, while doing so via experimentation would be expensive or infeasible.

In this paper, we focus on algorithms with the potential to be useful in practice, (using hash functions that have been widely used for myriad applications for decades), develop mathematical models and use them to set parameter values, hypothesize that any differences from ideal models needed to prove theorems are relatively insignificant, and describe experiments that validate such hypotheses.

HyperLogLog uses 5 bits per substream— $5M$ bits for $N < 2^{32}$. Much higher values of N are typical in modern applications, but it is safe to assume that $N < 2^{64}$, so 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 [12] [13] [14] [15] [16] but still leave the following question: Is there a practical algorithm as simple as HyperLogLog with comparable accuracy that uses cM bits for some constant c that is significantly less than 6?

This paper is an extended version of our conference paper [17] that provides answers to this question, with more thorough and detailed explanations and a more logical and complete narrative than is possible within the strict requirements for conference papers.

The algorithms we present have the same structure as HyperLogLog but use much less memory. In Section 2, instead of recording the maximum number of trailing ones, we focus on *one* bit per sub-stream indicating whether a threshold has been hit. Then we use a rough estimate of the cardinality as an input parameter in order to set the threshold to be the logarithm of the estimated number of distinct items per substream and then compute a precise estimate of the cardinality of the stream. While not a streaming algorithm, the resulting algorithm is useful in its own right, admits a precise analysis, and 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. Section 5 covers details of developing real-world implementations that keep costs low by using low-level machine operations (see also [18]). Section 6 is a discussion of how these algorithms match up against others in the literature, and Section 7 concludes the paper with a discussion of possible further improvements and directions for further research.

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. Still, as we will see, it can play an important role in some practical applications. More important, it admits a precise analysis that we can use to develop the streaming algorithms in Section 3 and Section 4. One of the primary contributions of this paper is this analysis, presented in this section.

We start with hashing and stochastic averaging with M substreams precisely in the same manner as 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 at least since the original work of Flajolet and Martin [1] that the maximum number of trailing 1s 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 (see [10]). 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 1s. 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 0s. 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 Section 5). We typically take M to be 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 1s in x . As described in Section 5, 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 $T = 1$.

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; // > T trailing 1s?
    }
    double beta = 1.0 - 1.0*p(sketch)/M; // fraction of 0s
    return (int) (Math.pow(2, T)*M*Math.log(1.0/beta));
}

```

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 0s 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 1s 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 $p(\text{sketch})$; 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 Section 5. 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 1s 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

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

(see for example, [19]). The number of 0s in `sketch[]` is a binomially distributed random variable, so this value is also (approximately) β , the expected proportion of 0s 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 2.1. Suppose that $X_n \geq 0$ are random variables and a_n , b_n , and σ^2 numbers such that, as $n \rightarrow \infty$, we have $a_n \rightarrow a > 0$, $b_n \rightarrow 0$, 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,

$$\frac{f(X_n) - f(a_n)}{b_n} = f'(X_n^*) \frac{X_n - a_n}{b_n} \quad (1)$$

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 $b_n \rightarrow 0$, we have $X_n - a_n \xrightarrow{p} 0$. Furthermore, $a_n \rightarrow a$, 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 (1) and the assumption. \square

Lemma 2.2. 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 [20]. \square

Theorem 2.1. Suppose that a stream S has N distinct items and that `HyperBitT` processes S using M substreams with parameter T , terminates with βM 0s left in the sketch. If the hash functions return each possible value with equal probability, 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,

$$\frac{\sqrt{M}}{c(\beta)} \left(\frac{M2^T \ln(1/\beta)}{N} - 1 \right) \xrightarrow{d} \mathbb{N}(0, 1) \quad (2)$$

as $N, M, T \rightarrow \infty$ 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 $\text{Poisson}(\tilde{N})$. 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 by Algorithm 1,
- a , a positive number,
- $\tilde{N} = M2^T \ln(1/\beta)$, the reported estimate of N , and
- $\tilde{N} \sim aM2^T$, the Poisson parameter.

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

We begin by finding, in the Poisson model, the distribution of βM , the number of 0s 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 $\text{Poisson}(\tilde{N}/M)$. These random numbers are independent, and each item in the k th 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 $Y_k \in \text{Poisson}(2^{-T} \tilde{N}/M) = \text{Poisson}(\tilde{N}/(M2^T))$. Now, let q be the probability that `sketch[k] = 0` (which is the same for all k). Then

$$q = \mathbb{P}(Y_k = 0) = \exp\left(-\frac{\tilde{N}}{M2^T}\right) \rightarrow e^{-a}. \quad (3)$$

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

$$\beta M \in \text{Binomial}(M, q), \quad (4)$$

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

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

$$\sqrt{M}(\beta - q) = \frac{M\beta - Mq}{\sqrt{M}} \xrightarrow{d} \mathbb{N}(0, e^{-a}(1 - e^{-a})). \quad (5)$$

Now, applying Lemma 2.1 with the function $f(x) = \ln(1/x)$ gives

$$\sqrt{M}(\ln(1/\beta) - \ln(1/q)) \xrightarrow{d} \mathbb{N}(0, e^a - 1). \quad (6)$$

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

$$\sqrt{M}\left(\frac{\hat{N}}{\tilde{N}} - 1\right) = \sqrt{M} \frac{M2^T}{\tilde{N}} \left(\ln \frac{1}{\beta} - \ln \frac{1}{q}\right) \xrightarrow{d} \mathbb{N}(0, a^{-2}(e^a - 1)). \quad (7)$$

Furthermore, (6) implies $\ln(1/\beta) - \ln(1/q) \xrightarrow{p} 0$, and thus, using (3), $\ln(1/\beta) \xrightarrow{p} a$; hence (7) implies (2) (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 2.2. We may assume that we start by selecting all items with at least T trailing 1s. Since each item is selected with probability 2^{-T} , the number of selected items is $\text{Binomial}(N, 2^{-T})$. Similarly, if we consider the Poisson model with $\text{Poisson}(N)$ items (thus choosing $N = \tilde{N}$ above) then the number of selected items is $\text{Poisson}(N2^{-T})$. By Lemma 2.2. We may couple the two versions such that the number of selected items agree with probability no less than $1 - 2^{-T} \rightarrow 1$. Hence, (2) for a fixed N follows from the Poisson version.

We have proved that (2) 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 (2) holds for the subsubsequence. As is well known, this implies that the full sequence converges (see Section 5.7 in [21]). \square

In summary, 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

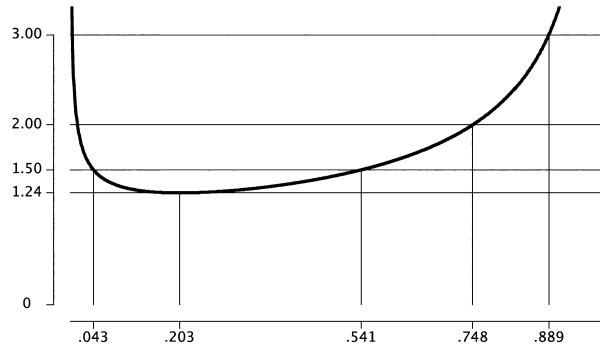


Fig. 1. This plot shows the coefficient of $1/\sqrt{M}$ in the relative standard error $c(\beta) = \sqrt{1/\beta - 1} / \ln(1/\beta)$ (y-coordinate) for β (fraction of 0s in the sketch) between 0 and 1 (x-coordinate). The value of $c(\beta)$ goes to infinity as β approaches 0 or 1, but it is relatively 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 $.0035 < \beta < .888$.

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

Theorem 2.1 provides formulas for two important pieces of information, as functions of β :

- the bias correction $\ln(1/\beta)$ that gives the estimate $2^T M \ln(1/\beta)$ for N ,
- the coefficient $c(\beta) = \sqrt{1/\beta - 1} / \ln(1/\beta)$ that gives the relative standard error $c(\beta)/\sqrt{M}$.

This is the information that we need to properly choose the value of T . Of most interest is the fact that $c(\beta)$ is relatively small and is large only when β is close to 0 or 1 (see Fig. 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 predominately 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 Fig. 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	$M 2^T \ln(1/\beta)$ for $\beta = .541$	$M 2^T \ln(1/\beta)$ for $\beta = .043$
6	40,261	206,212
7	80,522	412,425
8	161,044	824,850
9	322,088	1,649,701
10	644,177	3,299,402
11	1,288,355	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 2.1 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.)

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 Fig. 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 $c(\beta)$ 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 $M 2^T \ln(1/\beta)$ is between 161,044 and 824,851 (and $c(\beta) < 1.5$) when β is between .541 and .043.

Table 2 shows experimental results that constitute a quick validation check. Fig. 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 consist

Table 2

Since it is based on hash values, HyperBitT produces a different result every time it is run. This table shows the result of five consecutive runs of HyperBitT for our sample web log with $M = 1024$ and $T = 8$. The first column is the observed number of 0s in `sketch[]` when the run terminates. Dividing by M gives the values of β in the second column. The formula $2^T M \ln(1/\beta)$ gives the estimate of the cardinality in the third column, and the formula $c(\beta)/\sqrt{M}$ where $c(\beta) = \sqrt{1/\beta - 1}/\ln(1/\beta)$ gives the estimated relative standard error for each run in the fourth column. The actual error (comparing the estimated cardinality with the actual value 368,217) is in the fifth column. Since our estimate of the standard error is conservative ($c(\beta)$ 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.

observed 0s	observed β	estimated N	estimated error	actual error
228	.2226	393,773	3.9%	6.9%
253	.2470	366,498	3.9%	0.4%
257	.2509	362,386	3.9%	1.6%
261	.2548	358,338	3.9%	2.7%
265	.2587	354,351	3.9%	3.8%

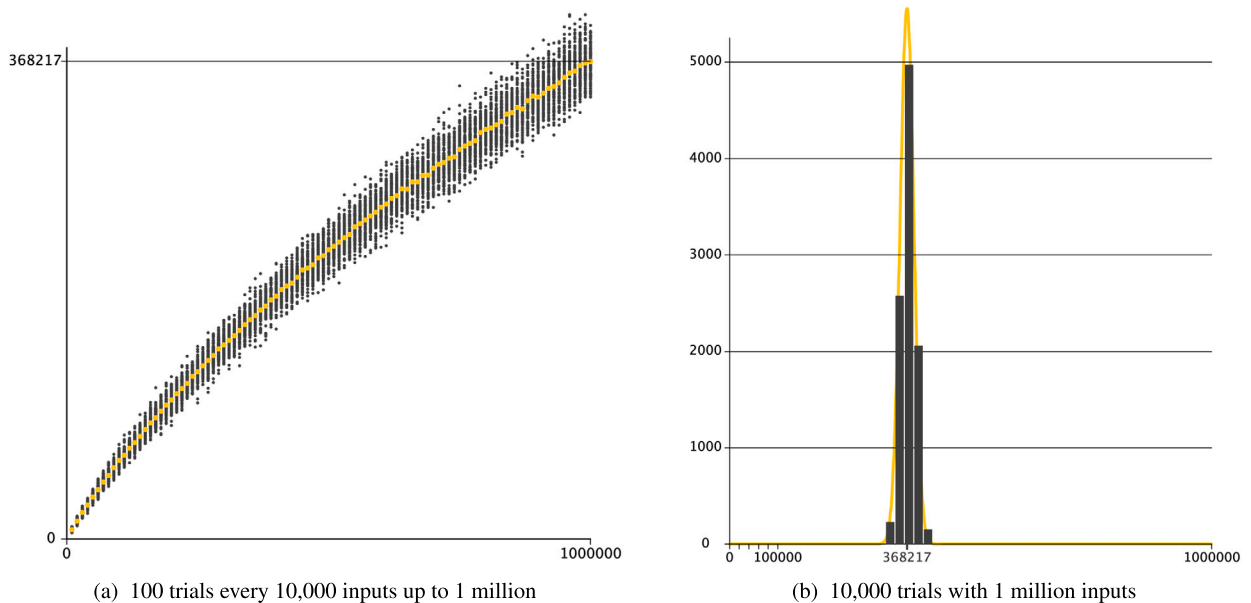


Fig. 2. Results of estimating cardinalities in a web log, each with 10,000 trials, taking $M = 1024$ and $T = 8$ as discussed in the text. In Fig. 2(a) HyperBitT was run 100 times for the first 10,000, 20,000, 30,000, ... items in the log, up to 1 million. Each grey dot shows the result of one experiment 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 Fig. 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 2.1 and our hypothesis that the estimated cardinality is likely to be within 5% of the true value. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

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.

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. We are faced with two choices: (i) when do we increase T , and (ii) by how much? Theorem 2.1 gives us precisely the information we need to make intelligent choices.

As the sketch becomes full, the percentage of 0s decreases and the standard error increases. Consider these values for $M = 64$:

# 1s in sketch	56	57	58	59	60	61	62	63
β	.125	.109	.093	.078	.062	.047	.031	.016
$c(\beta)$	1.27	1.29	1.31	1.35	1.40	1.47	1.60	1.90

For example, we can ensure that the relative standard error stays below $1.60/\sqrt{M}$ if we take action before the sketch has 63 1 bits.

Obviously, T needs to increase at that point. 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 1s 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? We are faced with a delicate balance because a small value leads to too many sketches to maintain and a large increment corresponds to too many 0s in the first sketch (which implies too large a value of $c(\beta)$).

Continuing our specific example, take $M = 64$ and suppose that we limit the sketch to at most 62 1s. 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 63rd 1 bit arrives to the sketch for T . At that point the value of β is $2/64 \doteq .031$. Our goal is to choose i such that we do not need to maintain a third sketch.

Let β_i be the fraction of 0s in the sketch for $T+i$. Because the estimated value of N is the same, we have $\ln(1/\beta_0) \approx \ln(1/\beta_i)/2^i$. Solving for β_i gives $\beta_i \approx \exp(-\ln(1/\beta_0)/2^i)$. For i from 0 to 8 we have these values:

β_0	β_1	β_2	β_3	β_4	β_5	β_6	β_7	β_8
.031	.177	.420	.648	.805	.898	.947	.973	.987

These values tell us that if we plan to increment T by 2, we can expect the sketch for $T+2$ to have about 27 0s ($64 \times \beta_2$) and the sketch for $T+4$ to have about 52 0s ($64 \times \beta_4$); if we plan to increment T by 3, we can expect the sketch for $T+3$ to have about 41 0s ($64 \times \beta_3$) and the sketch for $T+6$ to have about 61 0s ($64 \times \beta_6$); and if we plan to increment T by 4, we can expect the sketch for $T+4$ to have about 12 0s ($64 \times \beta_4$) and the sketch for $T+8$ to have about 63 0s ($64 \times \beta_8$); Specifically, these calculations tell 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*. If we choose to increment T by 4 when the sketch has 2 0s, we can do so, update our sketch for T using the sketch for $T+4$, and update the sketch for $T+4$ to all 0s (the implicit value of the sketch for $T+8$).

We may possibly be ignoring a few 1s that would be in the sketch for $T+8$ had we maintained it, but this is unlikely to noticeably affect the final estimate. Even in the worst case (all items distinct), the discrepancy can be bounded by a small constant factor; on typical data it is likely to be zero or exceedingly small. If one wants to be very conservative, another alternative is to maintain the indices of these 1s, at a very small (if not negligible) extra cost, but few practitioners would bother doing this.

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 63rd 1 bit arrives to the first sketch. 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 Section 5. (Note: A preliminary version of this algorithm found in earlier talks that uses heuristic bias correction has occasionally been referenced as ‘HyperBitBit.’ The fully developed version in Algorithm 2 appears only in this paper and its conference predecessor.)

Estimating the error 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 2.1 applies throughout. As we have discussed, just before incrementing T , *sketch0* has about 97% 1s and *sketch1* has about 20% 1s. Thus, the fraction of 0s in the sketches stays in the range $.03 < \beta < .80$ and the value of $c(\beta)$ is in the flat part of its curve (see Fig. 1). Just after T is incremented the value of c_β starts at $c(.80) \doteq 1.27$, then decreases to the flat part of the curve (moving from right to left in the figure), staying there until increasing to $c(.03) \doteq 1.61$ just before the next increment—the value is always less than 2.27 with average value about $\frac{1}{.77} \int_{.03}^{.80} c(\beta) d\beta \doteq 1.48$. This is conservative—for 80% of the range the average value is smaller, about $\frac{1}{.69} \int_{.11}^{.72} c(\beta) d\beta \doteq 1.42$.

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 6, 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 an example, see [22]).

Larger values of M For $M = 128$ or $M = 256$ we can use two or four 64-bit integers, respectively, to implement `HyperBitBit` using the following strategy: *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

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;
        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;
    return (int) (Math.pow(2, T)*M*Math.log(1.0/beta));
}

```

is $a = \ln(1/\beta) \doteq 4.41$ so $\beta = e^{-a} = .012$. That is, we should increment T by 4 and update the sketches when `sketch0` has $.988M$ 1 bits (about 126 for $M = 128$ and 253 for $M = 256$). At that point, the proportion of 0s in the sketch for $T+4$ will be about $e^{-a/2^4} \doteq .759$. The proportion of 0s in the sketch for $T+8$ would be about $e^{-a/2^8} \doteq .983$, so we are ignoring $(2, 4)$ 1 bits for $(128, 256)$ respectively, which is likely tolerable. The fraction of 0s in the sketches stays in the range $.012 < \beta < .759$, so the value of $c(\beta)$ is always less than 2.05 with average value about $\frac{1}{.747} \int_{.012}^{.759} c(\beta) d\beta \doteq 1.46$. For $M = 128$ the relative standard error is approximately $1.46/\sqrt{128} \doteq 13\%$; for $M = 256$ it is approximately $1.46/\sqrt{256} \doteq 9\%$.

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 1s 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 $e^{-a/2^{12}} \doteq .9989$, so we are ignoring (1, 1, 2, 4) 1 bits for (512, 1024, 2048, and 4096) respectively, again likely tolerable.

As just noted for HyperBitBit, the fraction of 0s in the sketches stays in the range $.012 < \beta < .759$, so the value $c(\beta)$ is always less than 2.05 with average value about $\frac{1}{.747} \int_{.012}^{.759} c(\beta) d\beta \doteq 1.46$. 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. That is, there are only four possibilities for the values in the corresponding position in the sketches. This observation means that we can represent the three sketches with an array of two-bit values that encode in binary the number of 1s in each position in the three sketches in `HyperBitBitBit`, as illustrated in the example below. Maintaining the array of two-bit values 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*.

before resetting sketches

<i>sketch for T</i>	111111111110111011111111111111111111011101111111001111
<i>sketch for T+4</i>	000100111010000000000001000011001011000000111100001000
<i>sketch for T+8</i>	00000001000000000000000000000000000001000000001100001000
<i>two-bit values</i>	111211232120111011111112111122112123101110223311003111

after resetting sketches and incrementing

<i>sketch for T</i>	000100111010000000000001000011001011000000111100001000
<i>sketch for $T+4$</i>	00000001000000000000000000000000000001000000001100001000
<i>sketch for $T+8$</i>	000
<i>two-bit values</i>	000100121010000000000001010011001012000000112200002000

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 Section 5). 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 Section 5).

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));
}

```

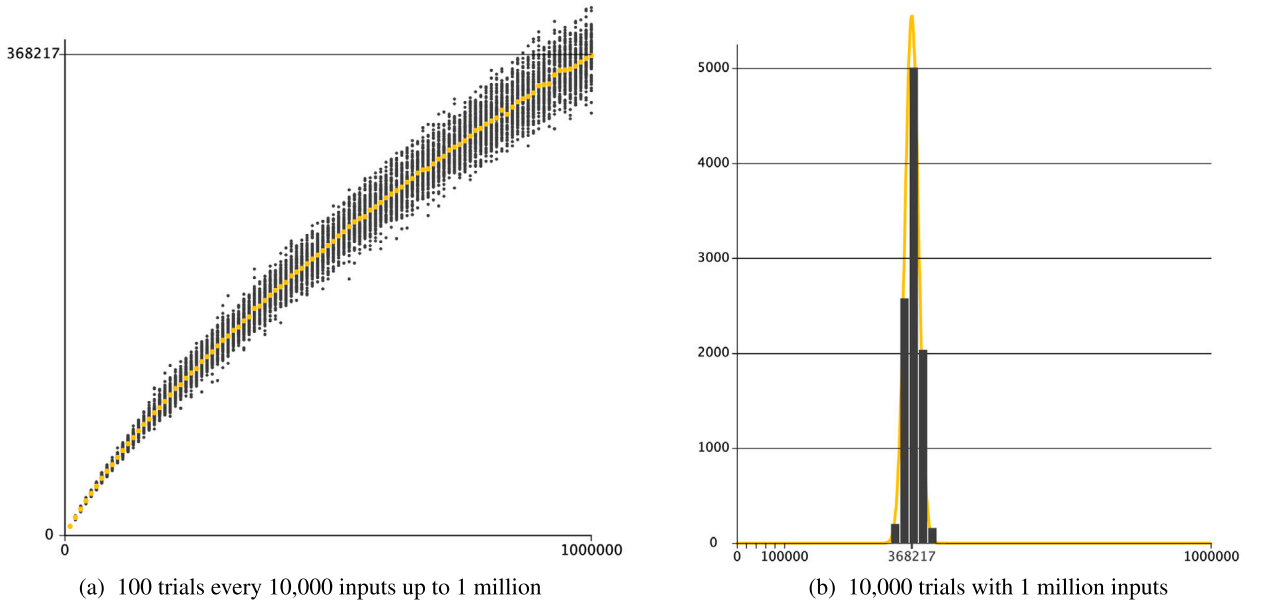


Fig. 3. Results of estimating cardinalities in a web log using Algorithm 3 with $M = 1024$, for comparison with Fig. 2 (where the details of the experiments are described). Given the same inputs (and the same random numbers), the figures for HyperBitBitBit would be identical.

In summary, HyperTwoBits is a true streaming algorithm, effective for M up to at least 4096, that uses $2M$ bits (plus six bits for T) and achieves relative standard error of about $1.46/\sqrt{M}$. As described in Section 5, it can be implemented such that processing each item in a stream requires only a few machine-language operations.

Fig. 3 presents the results of two experiments for Algorithm 3 corresponding to those presented for Algorithm 1 in Fig. 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.

5. 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 \ll and \gg and bitwise logical operations ($\&$, $|$, and \sim) for bitwise (AND, OR, and NOT) respectively.

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 s_1 for the more significant bit and a second `long` array s_0 for the less significant bit. To make the code more readable, we define the following methods to get and set the bit corresponding to bit k :

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

public static void set(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 tables, which give all possibilities for a given bit position, where s_1s_0 is the value before incrementing and t_1t_0 is the value after decrementing.

	before			after	
value	s_1	s_0	value	t_1	t_0
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 $t_1 = s_1 \text{ AND } s_0$ and $t_0 = s_1 \text{ AND NOT } s_0$. Furthermore, we can eliminate the temporary variables by doing the operations in the order $s_0 = s_1 \text{ AND NOT } s_0$ and then $s_1 = s_1 \text{ AND NOT } s_0$. Implementing these operations with bitwise operations on our arrays of `long` values is straightforward.

Algorithm 4 is a full low-level implementation of `HyperTwoBits` that uses these techniques to solve the practical cardinality estimation problem.

6. 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.

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

```

public static int estimateHTB(Iterable<String> stream, int M)
{
    int U = 2; // 2^T
    long[] s0 = new long [M/64];
    long[] s1 = new long [M/64];
    int count = 0;
    for (String s : stream)
    {
        long x = hash1(s); // 64-bit hash
        int k = hash2(s, M); // (lg M)-bit hash
        if ((x & (U-1)) == (U-1)) // > T trailing 1s?
            if (get(s1, s0, k) == 0)
            { count++; set(s1, s0, k, 1); }
        if ((x & (16*U-1)) == (16*U-1)) // > T+4?
            if (get(s1, s0, k) < 2) set(s1, s0, k, 2);
        if ((x & (256*U-1)) == (256*U-1)) // > T+8?
            if (get(s1, s0, k) < 3) set(s1, s0, k, 3);
        if (count >= .988*M)
        {
            for (int j = 0; j < M/64; j++) // Decrement counts
            { s0[j] = s1[j] & ~s0[j]; s1[j] = s1[j] & ~s0[j]; }
            count = 0;
            for (int j = 0; j < M; j++) // Count the non-0s
            { if (get(s1, s0, j) > 0) count++; }
            U = 16*U; // T+=4
        }
    }
    double beta = 1.0 - 1.0*count/M; // fraction of 0s
    return (int) (U*M*Math.log(1.0/beta));
}

```

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:

- (i) 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.
- (ii) The analyses generally define complexity results that use O -notation and are not sufficiently precise to enable comparison of the relative accuracy with other methods.

Generally, such methods are not potentially feasible for practical applications.

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. And the time required to process each item is a critical factor in many typical applications. 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, and ignore constant factors when citing accuracy results. Proper comparisons require taking constant factors into account and reckoning with the *total number of bits* of memory in all cases (see [23] for a discussion of this point with regard to another streaming problem). This reflects the broader reality that much of the literature focuses on scaling in terms of M , the number of substreams, rather than the total bit budget, which is what ultimately governs practical space usage. One of the contributions of this paper is to highlight and explicitly control for this dimension, allowing fair comparisons and more precise memory-performance tradeoffs. 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^* , the total number of bits used, or $c\sqrt{b}/\sqrt{M^*}$. Inverting this equation gives the number of bits needed to achieve a given accuracy x : $M^* = b(c/x)^2$.

Fourth, few papers actually *prove* anything about the distribution of the reported values, with the notable exception of [24]. 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 $c(\beta)$ of $1/\sqrt{M}$ in the relative standard error, which varies. We use the average value of $c(\beta)$ 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 shape of the curve in Fig. 1 tells us that it is likely that the value encountered in practice is smaller than the value cited.

Table 3

Performance of cardinality estimation algorithms: Adaptive sampling [1] (AS), Probabilistic counting [2] (PC), LogLog[3] (LL), HyperLogLog 8-bit and 6-bit versions [4] (HLL8 and HLL), Extended HyperLogLog[12] (EHLL), HyperBitT (HBT), HyperBitBit (HBB), HyperBitBitBit (HBBB), and HyperTwoBits (HTB). We emphasize normalization by *total bits used* (M^*), not just the number of substreams (M), to ensure fair comparisons.

algorithm	b	c	$c\sqrt{b}$	$M^* = b(c/x)^2$ bits needed for		$c\sqrt{b}/\sqrt{M^*}$ accuracy with	
				2%	20%	128 bits	8K bits
AS	64	1.20	9.60	230400	2304	85%	10.6%
PC	64	0.78	6.24	97344	973	55%	6.9%
LL	6	1.05	2.57	16538	165	23%	3.5%
HLL8	8	1.04	2.94	21632	216	26%	3.3%
HLL	6	1.02	2.55	16224	162	23%	2.8%
EHLL	7	0.88	2.33	13552	136	21%	2.6%
HBT	1	1.32	1.32	4356	44	12%	1.5%
HBB	2	1.46	2.06	—	128	18%	—
HBBB	3	1.46	2.53	15987	128	22%	2.8%
HTB	2	1.46	2.06	10658	128	18%	2.3%

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.

Seventh, we ignore relatively inconsequential small fixed costs such as the memory required to store the value of T . In theory, our algorithms all require $\lg \lg N$ bits to represent T . This is inconsequential in practice because six bits suffice to represent $\lg \lg N$ in any conceivable application, but it is worthwhile noting that we are *not* claiming that the algorithms use $O(1)$ memory for fixed M .

With all these caveats, Table 3 presents a comparison of the algorithms we have discussed. HyperBitT is the best by far when starting with a rough estimate is feasible. Among the streaming algorithms, our simplest and perhaps most useful implementation is HyperBitBit64, which achieves 18.5% accuracy on a stream on any length with just 128 bits (two 64-bit words) and can be implemented with a few dozen machine instructions. The 256-bit and 512-bit versions referred to in Section 3 are also simple and likely useful, as they achieve 13% and 9% accuracy, respectively. For better accuracy, a larger value of M (and more memory) is needed. 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 better than the 6-bit version of HyperLogLog. Generally, our algorithms provide much better accuracy for the same number of bits as HyperLogLog and use significantly fewer bits to achieve the same accuracy. These results underscore the importance of evaluating accuracy with respect to total bit usage (not just register count) as we have systematically done throughout this paper.

Further validation may be found in the work by Geis [25], a thorough implementation derived from the conference version of this paper [17].

7. 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.

- *Small cardinalities.* In many practical applications, the cardinality is often likely to be small, in which case users prefer exact values. For this reason, using a simpler algorithm that returns exact values for small cardinalities is typical [5]. Our accuracy estimates hold even for small cardinalities, so the switch from exact to approximate is an application-dependent decision.
- *Sparse arrays.* Precise characterization of the situation just after incrementing T (when the sketches are mostly 0s) may lead to slightly better accuracy estimates.
- *Use both sketches.* The second sketch contains information that may lead to a better accuracy estimate. Analyzing this effect is tractable, but not likely to improve the estimate 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.
- *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 $(T, \text{sketch}_0, \text{sketch}_1)$. To merge the two sketches $(T_A, \text{sketch}_{0A}, \text{sketch}_{1A})$ and $(T_B, \text{sketch}_{0B}, \text{sketch}_{1B})$ consider the following four cases:
 - If $T_A = T_B = T$ use $(T, \text{sketch}_{0A} \mid \text{sketch}_{0B}, \text{sketch}_{1A} \mid \text{sketch}_{1B})$.

(ii) If the values of T differ by 8 or more, use the larger value and its sketches.

(iii) If $T_A = T_B + 4$, use $(T_A, \text{sketch}_{0_A} | \text{sketch}_{1_B}, \text{sketch}_{1_A})$.

(iv) If $T_B = T_A + 4$, use $(T_B, \text{sketch}_{0_B} | \text{sketch}_{1_A}, \text{sketch}_{1_B})$.

In all 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. All of these approaches require alignment of the hash functions, which may present practical challenges.

As a final remark, we note that we have studied many approaches to developing a true `HyperBit` algorithm, by modifying `HyperBitT` to just increment T , reset the sketch to 0s, and then characterizing the error due to the transition. Despite some promising empirical results with heuristic bias corrections, the problem of developing a mathematical model admitting proper comparison of such an algorithm with the ones described here remains open.

CRediT authorship contribution statement

Svante Janson: Writing – original draft, Validation, Investigation, Formal analysis, Conceptualization. **Jérémy Lumbroso:** Writing – review & editing, Writing – original draft, Validation, Methodology, Investigation, Formal analysis, Conceptualization. **Robert Sedgewick:** Writing – review & editing, Writing – original draft, Supervision, Methodology, Investigation, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Svante Janson reports financial support was provided by Knut and Alice Wallenberg Foundation. Svante Janson reports financial support was provided by Swedish Research Council. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work is dedicated to the memory of Philippe Flajolet.

These results were developed over a period of several years through talks and discussions at AofA (International Meetings on Probabilistic, Combinatorial and Asymptotic Methods for the Analysis of Algorithms), Schloss Dagstuhl – Leibniz Center for Informatics, and Banff International Research Station conferences, seminars, and workshops. We are indebted to these institutions and the organizers of these meetings for their work in supporting this research.

We would like to thank Martin Pépin and two anonymous reviewers for their helpful comments on our initial submission; and Seth Pettie and Jelani Nelson for feedback on this paper. Thanks are due to Heinz Geis for pointing out a bug in `HyperBitT` in the conference version of the paper. We would also like to thank our colleagues, Conrado Martínez, Sampath Kannan, Val Tannen, and Pedro Paredes 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.

References

- [1] P. Flajolet, G.N. Martin, Probabilistic counting, in: 24th Annual Symposium on Foundations of Computer Science, Tucson, Arizona, USA, IEEE Computer Society, 1983, pp. 76–82.
- [2] P. Flajolet, G.N. Martin, Probabilistic counting algorithms for data base applications, *J. Comput. Syst. Sci.* 31 (2) (1985) 182–209, [https://doi.org/10.1016/0022-0000\(85\)90041-8](https://doi.org/10.1016/0022-0000(85)90041-8).
- [3] M. Durand, P. Flajolet, Loglog counting of large cardinalities (extended abstract), in: G.D. Battista, U. Zwick (Eds.), Algorithms - ESA 2003, 11th Annual European Symposium, Budapest, Hungary, September 16–19, 2003, Proceedings, in: Lecture Notes in Computer Science, vol. 2832, Springer, 2003, pp. 605–617.
- [4] P. Flajolet, Éric Fusy, O. Gandouet, F. Meunier, Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm, in: P. Jacquet (Ed.), AofA 07—Proceedings of the 2007 Conference on Analysis of Algorithms, Juan-les-pins, France, June 18–22, 2007, DMTCS Proceedings volume AH, DMTCS, 2007, pp. 127–146.
- [5] S. Heule, M. Nunkesser, A. Hall, Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm, in: G. Guerrini, N.W. Paton (Eds.), Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18–22, 2013, ACM, 2013, pp. 683–692.
- [6] J. Lumbroso, How Flajolet processed streams with coin flips, *CoRR*, arXiv:1805.00612, 2018, arXiv:1805.00612.
- [7] N. Alon, Y. Matias, M. Szegedy, The space complexity of approximating the frequency moments, in: G.L. Miller (Ed.), Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22–24, 1996, ACM, 1996, pp. 20–29.
- [8] P. Indyk, D.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, IEEE Computer Society, 2003, pp. 283–288.
- [9] D.M. Kane, J. Nelson, D.P. Woodruff, An optimal algorithm for the distinct elements problem, in: J. Paredaens, D.V. Gucht (Eds.), Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, June 6–11, 2010, Indianapolis, Indiana, USA, ACM, 2010, pp. 41–52.
- [10] D.E. Knuth, The Art of Computer Programming, Volume III, 2nd edition, Addison-Wesley, 1998.
- [11] K. Chung, M. Mitzenmacher, S.P. Vadhan, Why simple hash functions work: exploiting the entropy in a data stream, *Theory Comput.* 9 (2013) 897–945, <https://doi.org/10.4086/TOC.2013.V009A030>.
- [12] T. Ohayon, Extendedhyperloglog: analysis of a new cardinality estimator, *CoRR*, arXiv:2106.06525, 2021, arXiv:2106.06525.
- [13] D. Wang, S. Pettie, Better cardinality estimators for hyperloglog, pcsa, and beyond, in: F. Geerts, H.Q. Ngo, S. Sintos (Eds.), Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2023, Seattle, WA, USA, June 18–23, 2023, ACM, 2023, pp. 317–327.

- [14] J. Lumbroso, C. Martínez, Affirmative sampling: theory and applications, in: M.D. Ward (Ed.), 33rd International Conference on Probabilistic, Combinatorial and Asymptotic Methods for the Analysis of Algorithms (AofA 2022), in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 225, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2022, pp. 12:1–12:17.
- [15] M. Karpapa, R. Pagh, Hyperlogloglog: cardinality estimation with one log more, in: Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, 2022, pp. 753–761.
- [16] S. Pettie, D. Wang, Information theoretic limits of cardinality estimation: Fisher meets Shannon, in: Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing, 2021, pp. 556–569.
- [17] S. Janson, J. Lumbroso, R. Sedgewick, Bit-array-based alternatives to HyperLogLog, in: C. Mailler, S. Wild (Eds.), 35th International Conference on Probabilistic, Combinatorial and Asymptotic Methods for the Analysis of Algorithms (AofA 2024), in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 302, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2024, pp. 5:1–5:19.
- [18] S. Janson, J. Lumbroso, R. Sedgewick, Hyperbitt, hyperbitbit64 and hypertwobits, <https://github.com/robert-sedgewick/hyperbitbit>, 2024.
- [19] R. Sedgewick, P. Flajolet, An Introduction to the Analysis of Algorithms, second edition, Addison-Wesley-Longman, 2013.
- [20] A.D. Barbour, L. Holst, S. Janson, Poisson Approximation, Oxford University Press, 1992.
- [21] A. Gut, Probability: A Graduate Course, 2nd edition, Springer Texts in Statistics, vol. 75, 2013.
- [22] P. Boldi, M. Rosa, S. Vigna, Hyperanf: approximating the neighbourhood function of very large graphs on a budget, in: Proceedings of the 20th International Conference on World Wide Web, WWW '11, Association for Computing Machinery, New York, NY, USA, 2011, pp. 625–634.
- [23] Q. Ma, S. Muthukrishnan, M. Sandler, Frugal streaming for estimating quantiles: one (or two) memory suffices, arXiv:1407.1121, 2014.
- [24] J. Lumbroso, An optimal cardinality estimation algorithm based on order statistics and its full analysis, Discrete Math. Theor. Comput. Sci. AM (2010).
- [25] H. Geis, Hypertwobits implementation, <https://github.com/axiomhq/hypertwobits>, 2024.