Counting distinct elements in a data stream

Ziv Bar-Yossef[1], T. S. Jayram[2], Ravi Kumar[2], D. Sivakumar[2], and Luca Trevisan[3]

[1] Computer Science Division, Univ. of California at Berkeley, Berkeley, CA 94720. *
zivi@cs.berkeley.edu
[2] IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120.
{jayram, ravi, siva}@almaden.ibm.com
[3] Computer Science Division, Univ. of California at Berkeley, Berkeley, CA 94720. **
luca@cs.berkeley.edu

Abstract. We present three algorithms to count the number of distinct elements in a data stream to within a factor of $1 \pm \epsilon$. Our algorithms improve upon known algorithms for this problem, and offer a spectrum of time/space tradeoffs.

1 Introduction

Let $\mathbf{a} = a_1, \ldots, a_n$ be a sequence of n elements from the domain $[m] = \{1, \ldots, m\}$. The *zeroth-frequency moment* of this sequence is the number of distinct elements that occur in the sequence and is denoted $F_0 = F_0(\mathbf{a})$. In this paper we present three space- and time-efficient algorithms for approximating F_0 in the data stream model.

In the data stream model, an algorithm is considered efficient if it makes one (or a small number of) passes over the input sequence, uses very little space, and processes each element of the input very quickly. In our context, a data stream algorithm to approximate F_0 is considered efficient if it uses only $\mathrm{poly}(1/\epsilon, \log n, \log m)$ bits of memory, where $1 \pm \epsilon$ is the factor within which F_0 must be approximated.

Let $\epsilon, \delta > 0$ be given. An algorithm $\mathcal{A}$ is said to (ϵ, δ)-*approximate* F_0 if for any sequence $\mathbf{a} = a_1, \ldots, a_n$, with each $a_i \in [m]$, it outputs a number $\tilde{F}_0$ such that $\Pr[\left|F_0 - \tilde{F}_0\right| \leq \epsilon F_0] \geq 1 - \delta$, where the probability is taken over the internal coin tosses of $\mathcal{A}$. Two main parameters of $\mathcal{A}$ are of interest: the workspace and the time to process each item. We study these quantities as functions of the domain size m, the number n of elements in the stream, the approximation parameter ϵ, and the confidence parameter δ.

---

* Part of this work was done while the author was visiting IBM Almaden Research Center. Supported by NSF Grant CCR-9820897.
** Work supported by a Sloan Research Fellowship and an NSF Career Award.

There are several reasons for designing algorithms for F_0 in the data stream model. Counting the number of distinct elements in a (column of a relational) table of data is a fairly fundamental problem in databases. This has applications to estimating the selectivity of queries, designing good plans for executing a query, etc.—see, for instance, [WVT90,HNSS96]. Another application of counting distinct elements is in routing of Internet traffic. The router usually has very limited memory, but it is desirable to have the router gather various statistical properties (say, the number of distinct destination addresses) of the traffic flow. The number of distinct elements is also a natural quantity of interest in several large data set applications (eg., the number of distinct queries made to a search engine over a week).

Flajolet and Martin [FM85] designed the first algorithm for approximating F_0 in the data stream (or what was then thought of as a one-pass) model. Unfortunately, their algorithm assumed the existence of hash functions with some ideal properties; it is not known how to construct such functions with limited space. Alon, Matias, and Szegedy [AMS99] built on these ideas, but used random pairwise independent hash functions [CW77,WC79] and gave an (ϵ, δ)-approximation algorithm for $\epsilon > 1$; their algorithm uses $O(\log m)$ space. For arbitrarily small ϵ, Gibbons and Tirthapura [GT01] gave an algorithm that used $S = O(1/\epsilon^2 \cdot \log m)$ space and $O(S)$ processing time per element; Bar-Yossef *et al.* [BKS02] gave an algorithm that used $O(1/\epsilon^3 \cdot \log m)$ space (and time per element) but that had some other nice property required for their application. Cohen [Coh97] considered this problem in the context of graph-theoretic applications; her algorithm is similar in spirit to that of [FM85,AMS99]; specifically, it has a high-level viewpoint similar to the first algorithm in this paper. However, the implementation is very different, and does not yield a $o(m)$ space algorithm.

One of the drawbacks of the algorithms of [GT01,BKS02] is that the space and time are the *product* of $\mathrm{poly}(1/\epsilon)$ and $\log m$. Even with modestly small constants in the O notation and $\epsilon = 0.01$, the space required might be prohibitive in certain applications (eg., a router with very little memory or a database application where the frequency estimation is required to be piggy-backed on some other computation). This leads to the question of whether it is possible to obtain space/time upper bounds that are $\mathrm{poly}(1/\epsilon) + \log m$. In this paper we achieve this, modulo factors of the form $\log \log m$ and $\log(1/\epsilon)$.

Results. We give three algorithms with different space-time tradeoffs for approximating F_0. Each of our algorithms is an improvement over any of the existing algorithms in either space or processing time or both.

We will state the bounds for our algorithms in terms of ϵ and $\log m$ (suppressing the dependence on n). This is without loss of generality: If indeed $m < n$, it is clearly advantageous to have an algorithm whose bounds depend on $\log m$ and not on $\log n$. If, on the other hand, $m > n$, we can employ a simple hashing trick (with $O(\log(m + n))$ space and time per element) that reduces the description of each stream element to $O(\log n)$ bits. Thus we will assume for the rest of the paper that $\log m = O(\log n)$. We will also assume that there exists $\epsilon_0 < 1$ such that the accuracy parameter ϵ given to the algorithms is at most ϵ_0. (Note that this is, in fact, the interesting case. We make this assumption explicit only so that we may abbreviate $\max\{1/\epsilon, \epsilon_0\}$ by $1/\epsilon$.)

The following table summarizes our results. The $\tilde{O}$ notation suppresses $\log(1/\epsilon)$ and $\log\log m$ factors. For simplicity, the dependence on δ, which is a multiplicative factor of $\log(1/\delta)$ for both space and time, is also dropped.

Algorithm	Space	Time/element
1, Thm. 1	$O(1/\epsilon^2 \cdot \log m)$	$\tilde{O}(\log m)$
2, Thm. 2	$\tilde{O}(1/\epsilon^2 + \log m)$	$\tilde{O}(1/\epsilon^2 \cdot \log m)$
3, Thm. 3	$\tilde{O}(1/\epsilon^2 + \log m)$	$\tilde{O}(\log m)$ [amortized]

A lower bound of $\Omega(\log m)$ was shown in [AMS99]. It is also easy to show an $\Omega(1/\epsilon)$ lower bound by a reduction from the "indexing" problem in the one-way communication complexity model. An interesting open question is to obtain an algorithm with space bound $(1/\epsilon) \cdot \text{polylog}\ (m)$, or a lower bound of $\Omega(1/\epsilon^2)$.

2 The first algorithm

Our first algorithm is a generalization of the algorithm of [FM85,AMS99] to work for any $\epsilon > 0$.

To make our exposition clearer, we first describe an intuitive way to look at the algorithm of [FM85,AMS99]. This algorithm first picks a random hash function $h : [m] \to [0, 1]$. It then applies $h(\cdot)$ to all the elements in $\mathbf{a}$ and maintains the value $v = \min_{j=1}^{n} h(a_j)$. In the end, the estimation is $\tilde{F}_0 = 1/v$. The algorithm has the right approximation (in the expectation sense) because if there are F_0 independent and uniform values in $[0, 1]$, then their expected minimum is around $1/F_0$. Of course, the technical argument in [AMS99] is to quantify this precisely, even when h is chosen from a pairwise independent family of hash functions.

In our algorithm, we also pick a hash function $h : [m] \to [0, 1]$, but we keep the $t = O(1/\epsilon^2)$ elements a_i on which h evaluates to the t smallest values. If we let v be the t-th smallest such value, we estimate $\tilde{F}_0 = t/v$. This is because when we look at F_0 uniformly distributed (and, say, pairwise independent) elements of $[0, 1]$, we expect about t of them to be smaller than t/F_0. The formal description is given below.

Theorem 1. *There is an algorithm that for any $\epsilon, \delta > 0$, (ϵ, δ)-approximates F_0 using $O(1/\epsilon^2 \cdot \log m \cdot \log(1/\delta))$ bits of memory and $O(\log(1/\epsilon) \cdot \log m \cdot \log(1/\delta))$ processing time per element.*

Proof. We pick at random a pairwise independent hash function $h : [m] \to [M]$, where $M = m^3$. Note that, with probability at least $1 - 1/m$, h is injective over the elements of $\mathbf{a}$.

Let $t = \lceil 96/\epsilon^2 \rceil$. Our algorithm maintains the t smallest distinct values of $h(a_i)$ seen so far. The algorithm updates (if necessary) this list each time a new element arrives. Let v be the value of the t-th smallest such value when the entire sequence has been processed. The algorithm outputs the estimation $\tilde{F}_0 = tM/v$.

The algorithm can be implemented in $O(1/\epsilon^2 \cdot \log m)$ space, since the hash function h requires $O(\log m)$ space, and each of the $t = O(1/\epsilon^2)$ values to be stored requires $O(\log m)$ space. The t values could be stored in a balanced binary search tree, so that each step can be implemented in $O(\log(1/\epsilon) \cdot \log m)$ time (rather than $O(1/\epsilon^2 \cdot \log m)$ which would be necessary if the elements are stored as a list).

We can assume that $1/M < \epsilon t/(4F_0)$. Since $F_0 \leq m$, $M = m^3$, and $t \geq 96/\epsilon^2$, the condition is satisfied as long as $m \geq \sqrt{\epsilon/24}$. Let $b_1, \ldots, b_{F_0}$ be the distinct elements of $\mathbf{a}$.

Let us first consider the case $\tilde{F}_0 > (1+\epsilon)F_0$, i.e., the case when the algorithm outputs a value above $(1+\epsilon)F_0$. This means the sequence $h(b_1), \ldots, h(b_{F_0})$ contains at least t elements that are smaller than $tM/(F_0(1+\epsilon)) \leq (1-\epsilon/2)tM/F_0$ (using the fact $\epsilon \leq 1$). Each $h(b_i)$ has a probability at most $(1-\epsilon/2)t/F_0 + 1/M < (1-\epsilon/4)t/F_0$ (taking into account rounding errors) of being smaller than $(1-\epsilon/2)tM/F_0$. Thus, we are in a situation where we have F_0 pairwise independent events, each one occurring with probability at most $(1-\epsilon/4)t/F_0$, and at least t such events occur. Let $X_i, i = 1, \ldots, F_0$, be an indicator r.v. corresponding to the event "$h(b_i) < (1-\epsilon/2)tM/F_0$". Clearly, $\mathrm{E}[X_i] \leq (1-\epsilon/4)t/F_0$. Let $Y = \sum_{i=1}^{F_0} X_i$. It follows $\mathrm{E}[Y] \leq (1-\epsilon/4)t$ and by pairwise independence, $\mathrm{Var}(Y) \leq (1-\epsilon/4)t$. The event that the algorithm outputs a value above $(1+\epsilon)F_0$ occurs only if Y is more than t, and therefore the probability of error is:

$$\Pr[Y > t] \leq \Pr[|Y - \mathrm{E}[Y]| > \epsilon t/4] \leq 16 \cdot \mathrm{Var}[Y]/(\epsilon^2 t^2) \leq 16/(\epsilon^2 t) \leq 1/6,$$

using Chebyshev's inequality.

Let us consider now the case in which the algorithm outputs $\tilde{F}_0$ which is below $(1-\epsilon)F_0$. This means the sequence $h(b_1), \ldots, h(b_{F_0})$ contains less than t elements that are smaller than $tM/(F_0(1-\epsilon)) \leq (1+\epsilon)tM/F_0$. Let X_i be an indicator r.v. corresponding to the event "$h(b_i) \leq (1+\epsilon)tM/F_0$", and let $Y = \sum_{i=1}^{F_0} X_i$. Taking into account rounding errors, $(1+\epsilon/2)t/F_0 \leq \mathrm{E}[X_i] \leq (1+3\epsilon/2)t/F_0$, and therefore $\mathrm{E}[Y] \geq t(1+\epsilon/2)$ and $\mathrm{Var}[Y] \leq \mathrm{E}[Y] \leq t(1+3\epsilon/2)$, as before. Now, $\tilde{F}_0 < (1-\epsilon)F_0$ only if $Y < t$, and therefore the probability of error is:

$$\Pr[Y < t] \leq \Pr[|Y - \mathrm{E}[Y]| \geq \epsilon t/2] \leq 4 \cdot \mathrm{Var}[Y]/(\epsilon^2 t^2) \leq 12/(\epsilon^2 t) < 1/6$$

Thus, the probability that the algorithm outputs $\tilde{F}_0$ which is not within $(1 \pm \epsilon)$ factor of F_0 is at most $1/3 + 1/m$. As usual, this probability can be amplified to $1-\delta$ by running in parallel $O(\log(1/\delta))$ copies of the algorithm, and taking the median of the resulting approximations.

3 The second algorithm

The second algorithm is based on recasting the F_0 problem as estimating the probability of an appropriately defined event. The main idea is to define a quantity that can be approximated in the data stream model and that, in turn, can be used to approximate F_0.

Theorem 2. *There is an algorithm that for any $\epsilon, \delta > 0$, (ϵ, δ)-approximates F_0 using $\widetilde{O}((1/\epsilon^2 + \log m) \cdot \log(1/\delta))$ bits of memory. The processing time per data item is $\widetilde{O}(1/\epsilon^2 \cdot \log m \cdot \log(1/\delta))$. (The $\widetilde{O}$ notation suppresses $\log(1/\epsilon)$ and $\log \log m$ factors).*

Proof. We take advantage of the fact that the algorithm of [AMS99] can be used to provide a rough estimate of F_0; namely, it is possible to obtain an estimate R such that $2F_0 \leq R \leq 2cF_0$, where $c = 25$, with probability at least $3/5$. In addition, R may be assumed to be a power of 2. Our algorithm will implement the AMS algorithm on one track of the computation, and keep track of some extra quantities on another. In the sequel, we will add $2/5$ to the error probability, and assume that we have an estimate R that meets this bound.

Let $b_1, \ldots, b_{F_0}$ denote the F_0 distinct elements in the stream $\mathbf{a}$, and let B denote the set $\{b_1, \ldots, b_{F_0}\}$. Consider a completely random map $h : [m] \to [R]$, and define $r = \Pr_h[h^{-1}(0) \cap B \neq \emptyset] = 1 - (1 - 1/R)^{F_0}$. We first show that if R and F_0 are within constant multiples of each other, then approximating r is a good way to approximate F_0.

Lemma 1. *Let $c = 25$ and let $\epsilon > 0$ be given. Let R and F_0 satisfy $1/(2c) \leq (F_0/R) \leq 1/2$. Then if $|r - \tilde{r}| \leq \gamma = \min\{1/e - 1/3, \epsilon/(6c)\}$, then $\tilde{F}_0$, defined by*

$$\tilde{F}_0 = \frac{\ln(1 - \tilde{r})}{\ln(1 - 1/R)}$$

satisfies $\left| F_0 - \tilde{F}_0 \right| \leq \epsilon F_0$.

Proof. Since $R \geq 2F_0$, we have $R \geq 2$, therefore $1/R \leq 1/2$, and so $1 - 1/R \geq e^{-2/R}$ (since $1 - x \geq e^{-2x}$ for $x \leq 1/2$). Hence $r = 1 - (1 - 1/R)^{F_0} \leq 1 - e^{-2F_0/R}$. Since $F_0/R \leq 1/2$, we have $r \leq 1 - 1/e$. By definition, $\gamma \leq 1/e - 1/3$, so we have $r + \gamma < 2/3$ and $1/(1 - (r + \gamma)) < 3$. Also for $R > 1$, we have $-1/\ln(1 - 1/R) \leq R$.

For a continuous function f, $|f(x) - f(x + \epsilon)| \leq \epsilon \left| \sup_{y \in (x, x+\epsilon)} f'(y) \right|$. Letting $f(x) = \ln(1 - x)$, we obtain $|f(x) - f(\tilde{x})| \leq |x - \tilde{x}| / (1 - \max\{x, \tilde{x}\})$. Therefore,

$$\left| F_0 - \tilde{F}_0 \right| = \frac{|\ln(1 - r) - \ln(1 - \tilde{r})|}{-\ln(1 - 1/R)} \leq \frac{R\,|r - \tilde{r}|}{1 - (r + \gamma)} \leq 3R\gamma \leq 3 \cdot (2cF_0) \cdot \frac{\epsilon}{6c} = \epsilon F_0.$$

Our idea is to approximate r by using hash functions h that are not totally random, but just random enough to yield the desired approximation. We will pick h from a family $\mathcal{H}$ of hash functions from $[m]$ into $[R]$, whose choice we will spell out shortly. Let $p = \Pr_{h\in\mathcal{H}}[h^{-1}(0) \cap B \neq \emptyset]$. For $i = 1, \ldots, F_0$, let $\mathcal{H}_i$ denote the set of hash functions in $\mathcal{H}$ that map the i-th distinct element b_i of B to 0. Note that $p = |\bigcup_i \mathcal{H}_i| \,/\, |\mathcal{H}|$, thus our goal is to estimate this union size. By inclusion–exclusion, we have

$$p = \left(\sum_i \Pr_{h\in\mathcal{H}}[h \in \mathcal{H}_i]\right) - \left(\sum_{i<j} \Pr_{h\in\mathcal{H}}[h \in \mathcal{H}_i \cap \mathcal{H}_j]\right) + \left(\sum_{i<j<k} \Pr_{h\in\mathcal{H}}[h \in \mathcal{H}_i \cap \mathcal{H}_j \cap \mathcal{H}_k]\right) - \cdots$$

Let P_ℓ denote the ℓ-th term in the above series. For any odd $t > 0$, we know that

$$\sum_{\ell=1}^{t-1} (-1)^{\ell+1} P_\ell \quad\leq\quad p \quad\leq \sum_{\ell=1}^{t} (-1)^{\ell+1} P_\ell.$$

Our key observation is that if picking $h \in \mathcal{H}$ yields a t-wise independent hash function, then we know precisely what each P_ℓ is, and we have

$$\sum_{\ell=1}^{t-1} (-1)^{\ell+1} \binom{F_0}{\ell} R^{-\ell} \leq p \leq \sum_{\ell=1}^{t} (-1)^{\ell+1} \binom{F_0}{\ell} R^{-\ell}. \tag{1}$$

On the other hand, via binomial expansion we know that

$$r = 1 - \left(1 - \frac{1}{R}\right)^{F_0} = \sum_{i=1}^{F_0} (-1)^{i+1} \binom{F_0}{i} R^{-i},$$

whence we have for odd t that

$$\sum_{\ell=1}^{t-1} (-1)^{\ell+1} \binom{F_0}{\ell} R^{-\ell} \leq r \leq \sum_{\ell=1}^{t} (-1)^{\ell+1} \binom{F_0}{\ell} R^{-\ell}. \tag{2}$$

From Equations (1) and (2), it follows that both p and r are sandwiched inside an interval of width $\binom{F_0}{t} R^{-t} \leq (eF_0/(tR))^t \leq (1/5)^t$, which, with a choice of $t = \lceil \lg(2/\gamma)/\lg 5 \rceil$ implies that $|p - r| \leq \gamma/2$. Recall that $\gamma = \min\{1/e - 1/3, \epsilon/(6c)\}$, where $c = 25$.

Finally, we will show how to produce an estimate $\tilde{p}$ of p such that $|p - \tilde{p}| \leq \gamma/2$, so that $|\tilde{p} - r| \leq \gamma$, and we can apply Lemma 1.

The idea is to pick several hash functions $h_1, \ldots, h_k$ from a family $\mathcal{H}$ of t-wise independent hash functions. For a sequence $H_R = (h_1, \ldots, h_k)$ of hash functions from $[m]$ into $[R]$, define the estimator

$$X(H_R) = \frac{1}{k} \left|\{j \mid h_j^{-1}(0) \cap B \neq \emptyset\}\right|.$$

Clearly, $\mathrm{E}[X(H_R)] = p$. If $k = O(1/\gamma^2) = O(1/\epsilon^2)$ is suitably large, then by Chebyshev's inequality, we can show that $\Pr[|X(H_R) - p| > \gamma/2] \leq 1/20$.

Each hash function can be described by $s = O(t \log m)$ bits. Instead of picking the k hash functions independently from $\mathcal{H}$, we will pick them pairwise independently; this requires only $2s$ bits (assuming $2^s \geq k$) that we dub the "master hash function." The idea is that we will keep only the master hash function; as each element a_i of the stream is processed, we will construct each of the k hash functions in turn and compute whether it maps a_i to 0. Extracting the description of each hash function h_j from the master hash function can be done easily in space $O(s)$ and time $O(s \log s)$. Alternatively, we could use a master hash function of $O(s \log k)$ bits with the ability to extract each hash function in space and time $O(s \log k)$.

Lastly, we spell out how we handle the issue that we don't know R to begin with. Recall that R will be available to us through the AMS algorithm only after the stream has been processed. Thus, at the end of the stream, we need the ability to compute the estimator $X(H_R)$ for each $R = 1, \ldots, \log m$. Here we use the fact that for standard hash functions where the size of the range $[m]$ is a power of 2, extracting the least significant z bits, $1 \leq z \leq \log m$, gives a hash function with range $[2^z]$. Thus, for each hash function h_j, we will maintain not just one bit indicating whether $h_j^{-1}(0) \cap B \neq \emptyset$, but we will keep track of the largest z such that for some element b in the stream, $h_j(b)$ had z least significant bits equal to zero.

To complete the correctness argument, note that the error probability is bounded by the error probability of the application of the AMS algorithm, which is $2/5$, plus the error probability of the estimation, which is $1/20$. Thus, with probability at least $11/20$, the algorithm produces an estimate $\tilde{F}_0$ of F_0 such that $\left|F_0 - \tilde{F}_0\right| \leq \epsilon F_0$. Repeating this $O(\log 1/\delta)$ times and taking the median reduces the error probability to δ.

Let us summarize the space and time requirements:

1. Storing the master hash requires space either $O(s \log s) = O(\log(1/\epsilon) \cdot \log m \cdot (\log\log(1/\epsilon) + \log\log m))$ or $O(s \log k) = O(\log^2(1/\epsilon) \cdot \log m)$.
2. Storing the number of trailing zeros for each hash function needs $O(k \log\log m)$ space, which is $O(1/\epsilon^2 \cdot \log\log m)$ bits.
3. To process each item of the stream, the time required is dominated by accessing the master hash function k times, and is therefore $\widetilde{O}(1/\epsilon^2 \cdot \log m)$, suppressing $\log\log m$ and $\log(1/\epsilon)$ factors.

4 The third algorithm

The algorithm in this section is a unified and improved version of two previous algorithms: one due to Bar-Yossef *et al.* [BKS02] and one due to Gibbons and Tirthapura [GT01].

Theorem 3. *There is an algorithm that for any $\epsilon, \delta > 0$, (ϵ, δ)-approximates F_0 using $S = \widetilde{O}((1/\epsilon^2 + \log m)\log(1/\delta))$ bits of memory (suppressing $\log(1/\epsilon)$ and $\log\log m$ factors). The processing time per data item is $O(S)$ in the worst-case and $O((\log m + \log(1/\epsilon))\log(1/\delta))$ amortized.*

Proof. For a bit string $s \in \{0,1\}^*$, we denote by $\textsc{trail}(s)$ the number of trailing 0's in s.

Let $b_1, \ldots, b_{F_0}$ the F_0 distinct elements in the input stream $a_1, \ldots, a_n$. Let $B = \{b_1, \ldots, b_{F_0}\}$ and for each $i \in [n]$, let $B_i = B \cap \{a_1, \ldots, a_i\}$.

The algorithm picks a random pairwise independent hash function $h : [m] \to [m]$; we will assume that m is a power of 2. For $t = 0, \ldots, \log m$, define $h_t : [m] \to [2^t]$ to be the projection of h on its last t bits. The algorithm finds the minimum t for which $r = \left|h_t^{-1}(0) \cap B\right| \le c/\epsilon^2$, where $c = 576$. It then outputs $r \cdot 2^t$.

In order to find this t, the algorithm initially assumes $t = 0$, and while scanning the input stream, stores in a buffer all the elements b_j from the stream for which $h_t(b_j) = 0$. When the size of the buffer exceeds c/ϵ^2 (say, after reading a_i) the algorithm increases t by one. Note that $h_{t+1}(b_j) = 0$ implies $h_t(b_j) = 0$. Therefore, the algorithm does not have to rescan $a_1, \ldots, a_i$ in order to obtain $h_{t+1}^{-1}(0) \cap B_i$; rather, since $h_{t+1}^{-1}(0) \subseteq h_t^{-1}(0)$, the algorithm will simply extract it from the buffer, which contains $h_t^{-1}(0) \cap B_i$. At the end of the execution we are left with the minimum t for which the buffer size (i.e., $\left|h_t^{-1}(0) \cap B\right|$) does not exceed c/ϵ^2.

Up to this point, this is a simpler exposition of the Gibbons–Tirthapura algorithm. We further improve the algorithm by storing the elements in the buffer more efficiently. Instead of keeping the actual names of the elements, we keep their hash values, using a second hash function. Specifically, let $g : [m] \to [3 \cdot ((\log m + 1) \cdot c/\epsilon^2)^2]$ be a randomly chosen pairwise independent hash function. Note that since we apply g on at most $(\log m + 1) \cdot (c/\epsilon^2)$ distinct elements, g is injective on these elements with probability at least $5/6$. For each element b_j stored in the buffer, we need to store also the largest number t for which $h_t(b_j) = 0$ (which is basically $\textsc{trail}(h(b_j))$); we use this value during the extraction of $h_{t+1}^{-1}(0) \cap B_i$ from $h_t^{-1}(0) \cap B_i$. In order to do this succinctly, we keep an array of balanced binary search trees T of size $\log m + 1$. The t-th entry in this array is to contain all the elements b_j in the buffer, for which $\textsc{trail}(h(b_j)) = t$.

We start by analyzing the space and time requirements of the algorithm. The space used by the algorithm breaks down as follows:

1. Hash function h: $O(\log m)$ bits.
2. Hash function g: $O(\log m + \log(1/\epsilon))$ bits.
3. Buffer T: $O(\log m)$ bits for the array itself, and $O(1/\epsilon^2 \cdot (\log(1/\epsilon) + \log \log m))$ for the elements stored in its binary search trees (because we always store at most $O(1/\epsilon^2)$ elements, and each one requires $O(\log(1/\epsilon) + \log \log m)$ bits).

The total space is, thus, $O(\log m + 1/\epsilon^2 \cdot (\log(1/\epsilon) + \log \log m))$.

The worst-case running time per item is $O(\log m + (1/\epsilon^2) \cdot (\log(1/\epsilon) + \log \log m))$, because this is the number of steps required to empty the buffer T. The amortized running time per item is, however, only $O(\log m + \log(1/\epsilon))$ because each element is inserted at most once and removed at most once from the buffer T.

We next prove the algorithm indeed produces a $1 \pm \epsilon$ relative approximation of F_0 with probability at least $2/3$.

One source of error in the algorithm is g having collisions. Note that for a given stream $a_1, \ldots, a_n$ and a given choice of h, the at most $(\log m + 1) \cdot (c/\epsilon^2)$ elements on which we apply g are totally fixed. Since the size of the range of g is thrice the square of the number of elements on which we apply it, and since g is pairwise independent, the probability that g has collisions on these elements is at most $1/6$. We thus assume from now on that g has no collisions, and will add $1/6$ to the final error.

For each $t = 0, 1, \ldots, \log m$, we define $X_t = \left|h_t^{-1}(0) \cap B\right|$; i.e., the number of distinct elements in the stream that h_t maps to 0. Further define for each such t and for each $j = 1, \ldots, F_0$, $X_{t,j}$ to be an indicator random variable, indicating whether $h_t(b_j) = 0$ or not. Note that $\mathrm{E}(X_{t,j}) = \Pr[h_t(b_j) = 0] = 1/2^t$ and $\mathrm{Var}[X_{t,j}] \leq \mathrm{E}[X_{t,j}]$. Therefore, $\mathrm{E}[X_t] = \sum_j \mathrm{E}[X_{j,t}] = F_0/2^t$ and $\mathrm{Var}[X_t] = \sum_j \mathrm{Var}[X_{j,t}] \leq \mathrm{E}[X_t]$ (the latter follows from the pairwise independence of $\{X_{t,j}\}_j$).

Let t^* be the final value of t produced by the algorithm; that is, t^* is the smallest t for which $X_t \leq c/\epsilon^2$. Note that the algorithm's output is $X_{t^*} \cdot 2^{t^*}$.

If $t^* = 0$, then it means that $F_0 \leq c/\epsilon^2$, in which case our algorithm computes F_0 exactly. Assume, then, that $t^* \geq 1$. We write the algorithm's error probability as follows: (in the derivation we define $\bar{t}$ to be the t for which $12/\epsilon^2 \leq F_0/2^{\bar{t}} < 24/\epsilon^2$; note that such a $\bar{t}$ always exists).

$$
\begin{aligned}
&\Pr\left[\left|X_{t^*} \cdot 2^{t^*} - F_0\right| > \epsilon F_0\right] \;=\; \Pr\left[\left|X_{t^*} - \frac{F_0}{2^{t^*}}\right| > \epsilon \frac{F_0}{2^{t^*}}\right] = \\
&= \sum_{t=1}^{\log m} \Pr\left[\left|X_t - \frac{F_0}{2^t}\right| > \epsilon \frac{F_0}{2^t} \;\mid\; t^* = t\right] \cdot \Pr[t^* = t] \\
&= \sum_{t=1}^{\log m} \Pr\left[|X_t - \mathrm{E}[X_t]| > \epsilon \mathrm{E}[X_t] \mid X_t \leq \frac{c}{\epsilon^2}, X_{t-1} > \frac{c}{\epsilon^2}\right] \cdot \Pr\left[X_t \leq \frac{c}{\epsilon^2}, X_{t-1} > \frac{c}{\epsilon^2}\right] \\
&\leq \sum_{t=1}^{\bar{t}-1} \Pr\left[|X_t - \mathrm{E}[X_t]| > \epsilon \mathrm{E}[X_t]\right] + \sum_{t=\bar{t}}^{\log m} \Pr\left[X_t \leq \frac{c}{\epsilon^2}, X_{t-1} > \frac{c}{\epsilon^2}\right] \\
&\leq \sum_{t=1}^{\bar{t}-1} \frac{\mathrm{Var}[X_t]}{\epsilon^2 \mathrm{E}^2[X_t]} + \Pr\left[X_{\bar{t}-1} > \frac{c}{\epsilon^2}\right] \;\leq\; \sum_{t=1}^{\bar{t}-1} \frac{1}{\epsilon^2 \mathrm{E}[X_t]} + \mathrm{E}[X_{\bar{t}-1}] \cdot \frac{\epsilon^2}{c} \\
&= \sum_{t=1}^{\bar{t}-1} \frac{2^t}{\epsilon^2 F_0} + \frac{F_0}{2^{\bar{t}-1}} \cdot \frac{\epsilon^2}{c} \;\leq\; \frac{2^{\bar{t}}}{\epsilon^2 F_0} + \frac{F_0}{2^{\bar{t}}} \cdot \frac{2\epsilon^2}{c} \;\leq\; \frac{1}{\epsilon^2} \cdot \frac{\epsilon^2}{12} + \frac{24}{\epsilon^2} \cdot \frac{2\epsilon^2}{576} \;=\; \frac{1}{6}.
\end{aligned}
$$

Thus, the total error probability is $1/3$, as required. As before, this probability can be amplified to $1 - \delta$ by running in parallel $O(\log(1/\delta))$ copies of the algorithm, and outputting the median of the resulting approximations.

References

[AMS99] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

[BKS02] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 623–632, 2002.

[Coh97] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *Journal of Computer and System Sciences*, 55(3):441–453, 1997.

[CW77] L. Carter and M. Wegman. Universal classes of hash functions. In *Proceedings of the 9th ACM Annual Symposium on Theory of Computing*, pages 106–112, 1977. Journal version in *Journal of Computer and System Sciences*, 18(2) 143–154, 1979.

[FM85] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31:182–209, 1985.

[GT01] P. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *Proceedings of the 13th ACM Symposium on Parallel Algorithms and Architectures*, pages 281–291, 2001.

[HNSS96] P.J. Haas, J.F. Naughton, S. Seshadri, and A.N. Swami. Selectivity and cost estimation for joins based on random sampling. *Journal of Computer and System Sciences*, 52(3), 1996.

[WC79] M. Wegman and L. Carter. New classes and applications of hash functions. In *Proceedings of the 20th IEEE Annual Symposium on Foundations of Computer Science*, pages 175–182, 1979. Journal version titled “New Hash Functions and Their Use in Authentication and Set Equality” in *Journal of Computer and System Sciences*, 22(3): 265-279, 1981.

[WVT90] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems*, 15(2):208–229, 1990.