Fast Pseudo-Random Fingerprints

Yoram Bachrach[1], Ely Porat[2]

[1] Microsoft Research, Cambridge, UK (yobach@micorosft.com)
[2] Bar-Ilan University, Ramat Gan, Israel (porately@cs.biu.ac.il)

Abstract. We propose a method to exponentially speed up computation of various fingerprints, such as the ones used to compute similarity and rarity in massive data sets. Rather then maintaining the full stream of b items of a universe $[u]$, such methods only maintain a concise fingerprint of the stream, and perform computations using the fingerprints. The computations are done approximately, and the required fingerprint size k depends on the desired accuracy ϵ and confidence δ. Our technique maintains a single bit per hash function, rather than a single integer, thus requiring a fingerprint of length $k = O(\frac{\ln \frac{1}{\delta}}{\epsilon^2})$ bits, rather than $O(\log u \cdot \frac{\ln \frac{1}{\delta}}{\epsilon^2})$ bits required by previous approaches. The main advantage of the fingerprints we propose is that rather than computing the fingerprint of a stream of b items in time of $O(b \cdot k)$, we can compute it in time $O(b \log k)$. Thus this allows an exponential speedup for the fingerprint construction, or alternatively allows achieving a much higher accuracy while preserving computation time. Our methods rely on a specific family of pseudo-random hashes for which we can quickly locate hashes resulting in small values.

1 Introduction

Hashing is a key tool in processing massive data sets. Many uses of hashing in various applications require computing many hash functions in parallel. In this paper we present a technique that "ties together" many hashes in a novel way, which enables us to speed up such algorithms by an *exponential factor*. Our method also works for some complicated hash function such as min-wise independent families of hashes. In this paper we focus on producing an optimal similarity fingerprint using this method, but our technique is *general*, as it is easy to use our approach to speed up other hash intensive computations. One easy example where our technique applies is approximating the number of distinct elements from [1]. A another example, which requires a slightly stronger analysis, is computing of L_p sketches [13] for $0 \leq p \leq 2$.

Min-wise independent families of hash functions, which we call *MWIFs* for short, were introduced in [16, 6]. Computations using MWIFs have been used in many algorithms for processing massive data streams. The properties of MWIFs allow maintaining concise descriptions of massive streams. These descriptions, called "fingerprints" or "sketches", allow computing properties of these streams and relations between them. Examples of such "fingerprint" computations include data summerization and subpopulation-size queries [9, 8], greedy list intersection [14], approximating rarity and similarity for data streams [10], collaborative filtering fingerprints [4, 3, 2] and estimating frequency moments [1]. Another motivation for studying MWIFs is reducing the amount of randomness used by algorithms [7, 16, 6].

Recent research reduced the amount of information stored, while accurately computing properties data streams. Such techniques improve the *space complexity*, but much less attention has been given to *computation complexity*. For example, many streaming algorithms compute huge amounts of hashes, as they apply *many* hashes to each element in a very long stream of elements. This leads to a high computation time, not always tractable for many applications.

Our main contribution is a method allowing an *exponential* speedup in *computation time* for constructing fingerprints of massive data streams. Our technique is *general*, and can speed up many processes that apply many random hashes. The heart of the method lies in using a specific family of pseudo-random hashes shown to be approximately-MWIF [12], and for which we can quickly locate the hashes resulting in a small value of an element under the hash. Similarly to [17] we use the fact that members of the family are pairwise independent between themselves. We also extend the technique and show one can maintain just a *single* bit rather than the full element IDs, thus improving the fingerprint size. Independently of us [15] also considered storing few bits per hash function, but focused only on minimizing storage rather than computation time.

1.1 Preliminaries

Let H be a family of functions over the same source X and target Y, so each $h \in H$ is a function $h : X \to Y$, where Y is a completely ordered set. We say that H is min-wise independent if, when randomly choosing a function $h \in H$, for any subset $C \subseteq X$, any $x \in C$ has an equal probability of being the minimal after applying h.

Definition 1. *H is min-wise independent (MWIF), if for all $C \subseteq X$, for any $x \in C$, $Pr_{h\in H}[h(x) = min_{a\in C}h(a)] = \frac{1}{|C|}$*

Definition 2. *H is a γ-approximately min-wise independent (γ-MWIF), if for all $C \subseteq X$, for any $x \in C$, $\left|Pr_{h\in H}[h(x) = min_{a\in C}h(a)] - \frac{1}{|C|}\right| \leq \frac{\gamma}{|C|}$*

Definition 3. *H is k-wise independent, if for all $x_1, x_2, \ldots, x_k, y_1, y_2, \ldots, y_k \subseteq X$, $Pr_{h\in H}[(h(x_1) = y_1) \wedge \ldots \wedge (h(x_k) = y_k)] = \frac{1}{|X|^k}$*

2 Pseudo-Random Family of Hashes

We describe the hashes we use.Given the universe of item IDs $[u]$, consider a big prime p, such that $p > u$. Consider taking random coefficients for a d-degree polynomial in $\mathbb{Z}_p$. Let $a_0, a_1, \ldots, a_d \in [p]$ be chosen uniformly at random from $[p]$, and the following polynomial in $\mathbb{Z}_p$: $f(x) = a_0 + a_1x + a_2x^2 + \ldots + a_dx^d$. We denote by F_d the family of all d-degree polynomials in $\mathbb{Z}_p$ with coefficients in $\mathbb{Z}_p$, and later choose members of this family uniformly at random. Indyk [12] shows that choosing a function f from F_d uniformly at random results in F_d being a γ-MWIF for $d = O(\log \frac{1}{\gamma})$.

Randomly choosing $a_0, \ldots, a_d$ is equivalent to choosing a member of F_d uniformly at random, so $f(x) = a_0 + a_1x + a_2x^2 + \ldots + a_dx^d$ is a hash chosen at random from the γ-MWIF F_d. Similarly, consider $b_0, b_1, \ldots, b_d \in [p]$ be chosen uniformly at random from $[p]$, and $g(x) = b_0 + b_1x + b_2x^2 + \ldots + b_dx^d$, which is also a hash chosen

at random from the γ-MWIF F_d. Now consider the hashes $h_0(x) = f(x), h_1(x) = f(x) + g(x), h_2(x) = f(x) + 2g(x), \ldots, h_i(x) = f(x) + ig(x), \ldots, h_{k-1}(x) = f(x) + (k-1)g(x)$. We call this random construction procedure for $f(x), g(x)$ the *base random construction*, and the construction of h_i the *composition construction*. We prove properties of such hashes. We denote the probability of an event E when the hash h is constructed by choosing f, g using the base random construction and composing $h(x) = f(x) + i \cdot g(x)$ (for some $i \in [p]$) as $Pr_h(E)$.

Lemma 1 (Uniform Minimal Values). *Let f, g be constructed using the base random construction, using $d = O(\log \frac{1}{\gamma})$. For any $z \in [u]$, any $X \subseteq [u]$ and any value i used to compose $h(x) = f(x) + i \cdot g(x)$: $Pr_h[h(z) < min_{y \in X}(h(y)] = (1 \pm \gamma)\frac{1}{|X|}$.*

Proof. Fix i, $z \in [u]$ and $X \subseteq [u]$, construct f, g using the base random construction, and compose $h(x) = f(x) + i \cdot g(x)$. Note in $i \cdot g(x) = i \cdot (b_0 + b_1 x + \ldots b_d x^d)$, the coefficient of x^j is $q = (i \cdot b_j) \mod p$. Given a value $s \in [p]$ There is exactly one value in $r \in [p]$ such that $(q + r) \mod p = s$. Thus, for any $s \in [p]$, the probability that the coefficient of x^j in $h(x)$ is s is $\frac{1}{p}$. Therefor $Pr_h[h(x) \equiv p(x)] = \frac{1}{p^{d+1}} = \frac{1}{F_d}$. We have: $Pr_h[h(z) < min_{y \in X} h(y)] = \sum_{p(x) \in F_d} Pr_h[h(z) < min_{y \in X} h(y) | h(x) \equiv f(x) + i \cdot g(x) \equiv p(x)] \cdot Pr_h[h(x) \equiv p(x)] = \sum_{p(x) \in F_d} \frac{Pr_h[h(z) < min_{y \in X} h(y) | h(x) \equiv p(x)]}{|F_d|} = \sum_{p(x) \in F_d} \frac{Pr[p(z) < min_{y \in X}(p(y))]}{|F_d|} = Pr_{p(x) \in F_d}[p(z) < min_{y \in X}(p(y))] = (1 \pm \gamma)\frac{1}{|X|}$. If $p(x)$ is a polynom such that for any $z \in \mathbb{Z}_p$ we have $p(z) < min_{y \in X}(p(y))$, then we have $Pr[p(z) < min_{y \in X}(p(y))] = 1$, and otherwise $Pr[p(z) < min_{y \in X}(p(y))] = 0$. Thus we get $\sum_{p(x) \in F_d} \frac{Pr[p(z) < min_{y \in X}(p(y))]}{|F_d|} = Pr_{p(x) \in F_d}[p(z) < min_{y \in X}(p(y))]$. The last transition uses the fact that F_d is an γ-MWIF, which requires $d = O(\log \frac{1}{\gamma})$.

Lemma 2 (Pairwise Interaction). *Let f, g be constructed using the base random construction, using $d = O(\log \frac{1}{\gamma})$. For all $x_1, x_2 \in [u]$ and all $X_1, X_2 \subseteq [u]$, and all $i \neq j$ used to compose $h_i(x) = f(x) + i \cdot g(x)$ and $h_j(x) = f(x) + j \cdot g(x)$:*

$$Pr_{f,g \in F_d}[(h_i(x_1) < min_{y \in X_1} h_i(y)) \wedge (h_j(x_2) < min_{y \in X_2} h_i(y))] = (1 \pm \gamma)^2 \frac{1}{|X_1| \cdot |X_2|}$$

Proof. Given $p_1(x) \in F_d = u_0 + u_1 x + \ldots + u_d x^d$ and $p_2(x) \in F_d = v_0 + v_1 x + \ldots + v_d x^d$, there is *exactly one* pair of polynoms $f(x), g(x) \in F_d$ such that both $f(x) + i \cdot g(x) = p_1(x)$ and $f(x) + j \cdot g(x) = p_2(x)$. Each coefficient location $l \in [d]$ results in two equations with two unknowns in $\mathbb{Z}_p$, with a single solution (a_l, b_l) (where a_l is the coefficient of x^l in $f(x)$, and b_l is the coefficient of x^l in $g(x)$.

Fix $i \neq j$, $x_1, x_2 \in [u]$ and $X_1, X_2 \subseteq [u]$, construct f, g using the base random construction, and compose $h_i(x) = f(x) + i \cdot g(x)$, $h_j(x) = f(x) + j \cdot g(x)$. For brevity, denote $m_1^i = \min_{y \in X_1} h_i(y)$. Similarly, denote $m_2^j = \min_{y \in X_2} h_j(y)$. We have: $Pr_{f,g \in F_d}[(h_i(x_1) < m_1^i) \wedge (h_j(x_2) < m_2^j)] = \sum_{p_1, p_2 \in F_d} Pr[(h_i(x_1) < m_1^i) \wedge (h_j(x_2) < m_2^j) | (h_i(x) \equiv p_1(x) \wedge h_j(x) \equiv p_2(x))] \cdot Pr[(h_i(x) \equiv p_1(x) \wedge h_j(x) \equiv p_2(x))] = \sum_{p_1, p_2 \in F_d} \frac{Pr[(h_i(x_1) < m_1^i) \wedge (h_j(x_2) < m_2^j) | (h_i(x) \equiv p_1(x) \wedge h_j(x) \equiv p_2(x))]}{|F_d|^2}$. Thus, $Pr_{f,g \in F_d}[(h_i(x_1) < m_1^i) \wedge (h_j(x_2) < m_2^j)] = \sum_{p_1, p_2 \in F_d} \frac{Pr[(p_1(x_1) < m_1^i) \wedge (p_2(x_2) < m_2^j)]}{|F_d|^2} =$

$\sum_{p_1,p_2 \in F_d} \frac{Pr[p_1(x_1)<m_1^i] \cdot Pr[p_2(x_2)<m_2^j]}{|F_d|^2} = \sum_{p_1 \in F_d} \sum_{p_2 \in F_d} \frac{Pr[p_1(x_1)<m_1^i]}{|F_d|} \cdot \frac{Pr[p_2(x_2)<m_2^j]}{|F_d|} = (1 \pm \gamma)^2 \frac{1}{|X_1| \cdot |X_2|}$

3 Fingerprinting Using Pseudo-Random Hashes

Several methods were suggested for building fingerprints for approximating relations between massive datasets, such as the Jackard similarity (see [6] for example). Given a universe U, where $|U| = u$, consider C_1, C_2, where each $C_i \subseteq U$ is described as a set $|C_i|$ integers in $[u]$ (we use $[u]$ to denote $\{1, 2, \ldots, u\}$). The Jackard similarity is $J_{1,2} = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|}$. Many fingerprints rely on applying many hashes to each elements in the long streams. We use a the hashes of Section 2 to exponentially speed up such computations. We use pseudo-random effects in this hash, so we must relax the MWIF requirement to a pairwise independence requirement (2-wise independence).

For completeness, we briefly consider previously suggested approaches for approximating Jackard similarity [6]. Let $h \in H$ be a randomly chosen function from a MWIF H. We can apply h on all elements C_1 and examine the minimal integer we get, $m_1^h = \arg\min_{x \in C_1} h(x)$. We can do the same to C_2 and examine $m_2^h = \arg\min_{x \in C_2} h(x)$. Fingerprints for estimating the Jackard similarity are based on computing the probability that $m_1 = m_2$: $Pr_{h \in H}[m_1^h = m_2^h] = Pr_{h \in H}[\arg\min_{x \in C_1} h(x) = \arg\min_{x \in C_2} h(x)]$.

Theorem 1 (Jackard and MWIF Collision Probability). $Pr_{h \in H}[m_i^h = m_j^h] = J_{i,j}$. *The proof is given in [6], and in the appendix for completeness.*

Similarly, regarding a hash h from a γ-MWIF, [5, 6] shows that:

Theorem 2. $|Pr_{h \in H}[m_i^h = m_j^h] - J_{i,j}| \leq \gamma$.

Rather than maintaining the full C_i's, previous approaches [5, 6] suggest maintaining their fingerprints. Given k hashes $h_1, \ldots, h_k$ randomly chosen from an γ-MWIF, we can maintain $m_i^{h_1}, \ldots, m_i^{h_k}$. Given C_i, C_j, for any $x \in [k]$, the probability that $m_i^{h_x} = m_j^{h_x}$ is $J_{i,j} \pm \gamma$. A hash h_x where we have $m_i^{h_x} = m_j^{h_x}$ is called a hash collision. We can thus estimate J by counting the proportion of collision hashes out of all the chosen hashes. In this approach, the fingerprint contains k item identities in $[u]$, since for any x, $m_i^{h_x}$ is in $[u]$. Thus, such a fingerprint requires $k \log u$ bits. To achieve an accuracy ϵ and confidence δ, such approaches require $k = O(\frac{\ln \frac{1}{\delta}}{\epsilon^2})$. Our basis for the fingerprint is a "block fingerprint" which allows approximating $J_{i,j}$ with a given accuracy ϵ and a confidence of $\frac{7}{8}$. This block fingerprint maintains only a *single bit* per hash, as opposed to previous approaches which maintain $\log u$ bits per hash. Later we show how to achieve a given accuracy ϵ with a given confidence δ, by combining several block fingerprints, and creating a full fingerprint.

To shorten the fingerprints using a single bit per hash, we use a hash mapping elements in $[u]$ to a single bit — $\phi : [u] \rightarrow \{0, 1\}$, taken from a pairwise independent family (PWIF for short) of such hashes. Rather than defining $m_i^h = \arg\min_{x \in C_1} h(x)$ we define $m_i^{\phi,h} = \phi(\arg\min_{x \in C_1} h(x))$. Maintaining $m_i^{\phi,h}$ rather than m_i^{ϕ} shortens the fingerprint by a factor of $\log u$. We examine the resulting accuracy and confidence.

Theorem 3. $Pr_{h\in H}[m_i^{\phi,h} = m_j^{\phi,h}] = \frac{J_{i,j}}{2} + \frac{1}{2} \pm \frac{\gamma}{2}$.

Proof. $Pr_{h\in H,\phi\in H'}[m_i^{\phi,h} = m_j^{\phi,h}] = Pr[m_i^{\phi,h} = m_j^{\phi,h} | m_i^h = m_j^h] \cdot Pr_{h\in H}[m_i^h = m_j^h] + Pr[m_i^{\phi,h} = m_j^{\phi,h} | m_i^h \neq m_j^h] \cdot Pr_{h\in H}[m_i^h \neq m_j^h] = 1 \cdot Pr_{h\in H}[m_i^h = m_j^h] + \frac{1}{2} \cdot (1 - Pr_{h\in H}[m_i^h = m_j^h]) = \frac{1+J_{i,j}\pm\gamma}{2}$

The purpose of the fingerprint block is to provide an approximation of J with accuracy ϵ. We use k hashes, and choose $k = \frac{8.02}{\epsilon^2}$. Denote $\alpha = \frac{2^{10}-1}{2^{10}}$, and let $\gamma = (1-\alpha)\cdot\epsilon = \frac{1}{2^{10}}\epsilon$. We construct a γ-MWIF [1]. To construct the family, consider choosing $a_0, \ldots, a_d$ and $b_0, b_1, \ldots, b_d$ uniformly at random from $[p]$, constructing the polynomials $f(x) = a_0 + a_1x + a_2x^2 + \ldots + a_dx^d$, $g(x) = b_0 + b_1x + b_2x^2 + \ldots + b_dx^d$, and using the k hashes $h_i(x) = f(x) + ig(x)$, where $i \in \{0, 1, \ldots, k-1\}$. We also use a hash $\phi : [u] \to \{0, 1\}$ chosen from the PWIF of such hashes. We say there is a collision on h_l if $m_i^{\phi,h_l} = m_j^{\phi,h_l}$, and denote the random variable Z_l where $Z_l = 1$ if there is a collision on h_l for users i, j and $Z_l = 0$ if there is no such collision. $Z_l = 1$ with probability $\frac{1}{2} + \frac{J}{2} \pm \frac{\gamma}{2}$ and $Z_l = 0$ with probability $\frac{1}{2} - \frac{J}{2} \pm \frac{\gamma}{2}$. Thus $E(Z_l) = \frac{1}{2} + \frac{J}{2} \pm \frac{\gamma}{2}$. Denote $X_l = 2Z_l - 1$. $E(X_l) = 2E(Z_l) - 1 = J \pm \gamma$. X_l can take two values, -1 when $Z_l = 0$, and 1 when $Z_l = 1$. Thus X_l^2 always takes the value of 1, so $E(X_l^2) = 1$. Consider $X = \sum_{l=1}^{k} X_l$, and take $Y = \hat{J} = \frac{X}{k}$ as an estimator for J. We show that for the above choice of k, Y is accurate up to ϵ with probability of at least $\frac{7}{8}$.

Theorem 4 (Simple Estimator). $Pr(|Y - J| \leq \epsilon) \geq \frac{7}{8}$. *Proof given in appendix.*

Due to Theorem 4, we can approximate J with accuracy ϵ and confidence $\frac{7}{8}$ using a "block fingerprint" for C_i, composed of $m_i^{h_1,\phi_1}, \ldots, m_i^{h_k,\phi_k}$, where $h_1, \ldots, h_k$ are randomly constructed members of a γ-MWIF and $\phi_1, \ldots, \phi_k$ are chosen from the PWIF of hashes $\phi : [u] \to \{0, 1\}$. We shows that it suffices to take $k = O(\frac{1}{\epsilon^2})$ to achieve this. Constructing each h_i can be done by choosing f, g using the base random construction and composing $h_i(x) = f(x) + i \cdot g(x)$. The base random construction chooses f, g uniformly at random from F_d, the family of d-degree polynoms in $\mathbb{Z}_p$, where $d = O(\log \frac{1}{\epsilon})$. This achieves a γ-MWIF where $\gamma = (1-\alpha)\cdot\epsilon = \frac{1}{2^{10}}\epsilon$.

Achieving a Desired Confidence We combine several *independent* fingerprints to increase the confidence to a desired level δ. Section 3 used a fingerprint of length k to achieve a confidence of $\frac{7}{8}$. Consider taking m fingerprints for each stream, each of length k. Given two streams, i, j, we have m pairs of fingerprints, each approximating J with accuracy ϵ, and confidence $\frac{7}{8}$. Denote the estimators we obtain as $\hat{J}_1, \hat{J}_2, \ldots, \hat{J}_m$, and denote the *median* of these values as $\hat{J}$. Consider using $m > \frac{32}{9} \ln \frac{1}{\delta}$ "blocks".

Theorem 5 (Median Estimator). $Pr(|\hat{J} - J| \leq \epsilon) \geq 1 - \delta$. *Proof given in appendix.*

Due to Theorem 5 to make sure that $|\hat{J} - J| \leq \epsilon$ it suffices to take $m > \frac{32}{9} \ln \frac{1}{\delta}$ fingerprints, each with $k = \frac{8.02}{\epsilon^2}$ hashes. In total, it is enough to take $\frac{32}{9} \ln \frac{1}{\delta} \cdot \frac{8.02}{\epsilon^2} \leq \frac{28.45 \ln \frac{1}{\delta}}{\epsilon^2}$ hashes. Thus, we use $O(\frac{\ln \frac{1}{\delta}}{\epsilon^2})$ hashes, storing a single bit per hash.

[1] The accuracy γ is much stronger than the overall accuracy ϵ required of the full fingerprint, for reasons to be later examined

4 Fast Method for Computing the Fingerprint

We discuss speeding up the fingerprint computation. Consider computing the fingerprint for a set of b items $X = \{x_1, \ldots, x_b\}$ where $x_i \in [u]$. The fingerprint is composed of m "block fingerprints", where block r is constructed using k hashes $h_1^r, \ldots, h_k^r$, built using $2 \cdot d$ random coefficients in $\mathbb{Z}_p$. The i'th location in the block is the minimal item in X under h_i: $m_i = \arg\min_{x \in X} h_i(x)$, which is then hashed through a hash ϕ mapping elements in $[u]$ to a single bit. We show how to quickly compute the block fingerprint $(m_1, \ldots, m_k)$. A naive way to do this is applying $k \cdot b$ hashes to compute $h_i(x_j)$ for $i \in [k], j \in [b]$. The values $h_i(x_i)$ where $i \in [k], j \in [b]$ form a matrix, where row i has the values $(h_i(x_1), \ldots, h_i(x_b))$, illustrated in Figure 1.

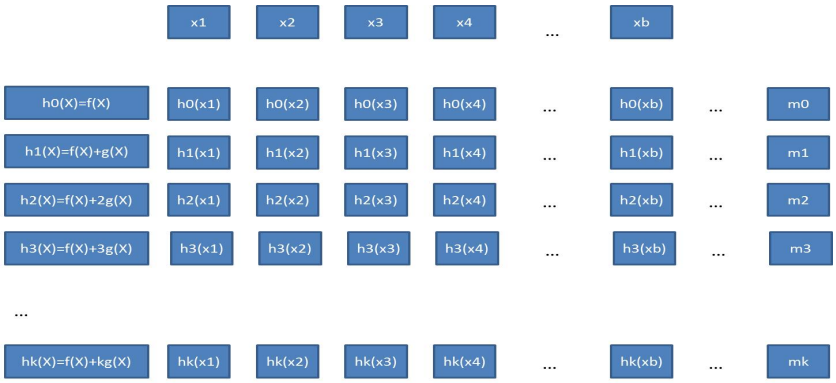

Fig. 1. A fingerprint "chunk" for a stream.

Once all $h_i(x_j)$ values are computed for $i \in [k], j \in [b]$, for each row i we check for which column j the row's minimal value occurs, and store $m_i = x_j$, as illustrated in the left of Figure 2. Thus, computing the fingerprint requires finding the minimal value across the rows (or more precisely, the value x_j for the column j where this minimal value occurs). To speed up the process, we use a method similar to the one discussed in [18] as a building block. Recall the hashes h_i were defined as $h_i(x) = f(x) + ig(x)$ where $f(x), g(x)$ are d-degree polynomials with random coefficients in $\mathbb{Z}_p$. Our algorithm is based on a procedure that gets a value $x \in [u]$ and a threshold t, and returns all elements in $(h_0(x), h_1(x), \ldots, h_{k-1}(x))$ which are smaller than t, as well as their locations. Formally, the method returns the index list $I_t = \{i | h_i(x) \leq t\}$ and the value list $V_t = \{h_i(x) | i \in I_t\}$ (note these are lists, so the j'th location in V_t, $V_t[j]$, contains $h_{I_t[j]}(x)$). We call this the *column procedure*, and denote by $pr - small - loc(f(x), g(x), k, x, t)$ the function that returns I_t, and by $pr - small - val(f(x), g(x), k, x, t)$ the function that returns V_t . We describe a certain implementation of these operations in Section 4.1. The running time of this implementation is $O(\log k + |I_t|)$, rather than the naive algorithm which evaluates $O(k)$ hashes. Thus, this procedure quickly finds small elements across columns (where by "small" we mean smaller than t). This is illustrated on the right of Figure 2.

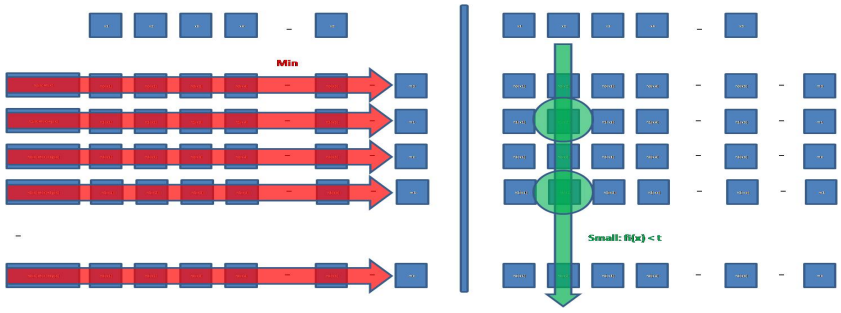

Fig. 2. Finding small elements across columns rather than minimal elements across rows

Roughly speaking, our algorithm maintains a bound for the minimal value for each row, and operates by going through the columns, finding the small values in each of them, and updating the bounds for the rows where these occur.

$block - update((x_1, \ldots, x_b), f(x), g(x), k, t)$:

1. Let $m_i = \infty$ for $i \in [k]$
2. Let $p_i = 0$ for $i \in [k]$
3. For $j = 1$ to b:
   (a) Let $I_t = pr - small - val(f(x), g(x), k, x_j, t)$
   (b) Let $V_t = pr - small - loc(f(x), g(x), k, x_j, t)$
   (c) For $y \in I_t$: // Indices of the small elements
      i. If $m_{I_t[y]} > V_t[y]$ // Update to row x required
         A. $m_{I_t[y]} = V_t[y]$
         B. $p_{I_t[y]} = x_j$

If our method updates m_i, p_i for row i, once the procedure is done, m_i indeed contains the minimal value in that row, and p_i the column where this minimal value occurs, since if even a single update occurred then the row indeed contains an item that is smaller than t, so the minimal item in that row is smaller than t and an update would occur for that item. On the other hand, if all the items in a row are bigger than t, an update would not occur for that row. The running time of the column procedure is $O(\log k + |I_t|)$, which is a random variable, that depends on the number of elements returned for that column, $|I_t|$. Denote by L_j the number of elements returned for column j (i.e. $|I_t|$ for column j). Since we have b columns, the running time of the block update is $O(b \log k) + O(\sum_{j=1}^{b} L_j)$. The total number of returned elements is $\sum_{j=1}^{b} L_j$, which is the total number of elements that are smaller than t. We denote by $Y_t = \sum_{j=1}^{b} L_j$ the random variable which is the number of all elements in the block that are smaller than t. The running time of our block update is thus $O(b \log k + Y_t)$.

The random variable Y_t depends on t, since the smaller t is the less elements are returned and the faster the column procedure runs. On the other hand, we only update rows whose minimal value is below t, so if t is too low we have a high probability of having rows which are not updated correctly. We show that a certain compromise t value allows achieving both a good running time of the block update, with a good probability of correctly computing the values for all the rows.

Theorem 6. *Given the threshold $t = \frac{12 \cdot p \cdot l'}{b}$, where $l' = 80 + 2\log\frac{1}{\epsilon}$ (so $l' = O(\log\frac{1}{\epsilon})$), the runtime of the $block - update$ procedure is $O(b\log\frac{1}{\epsilon} + \frac{1}{\epsilon^2}\log\frac{1}{\epsilon})$.*

Proof. Recall that to get a γ-MWIF (for $\gamma = \frac{1}{2^{10}}\epsilon$) we used $d = O(\log\frac{1}{\gamma})$ as the degree of the random polynoms f, g in the base random construction, used to compose the $h_1, \ldots, h_k$ hashes. Examining the constant in the work of Indyk [12] shows that the requirement is $d > 80 + 2\log\frac{1}{\epsilon}$. Denote $l' = 80 + 2\log\frac{1}{\epsilon}$. Due to our choice of d we have $d > l'$, so the hashes $h_1, \ldots, h_k$ were effectively chosen at random from an l'-wise independent family. Let H be an $l' - wise$ independent family of hashes. Consider the following equation from [12], regarding E_t, the expected number of elements $x \in X$ such that $h(x) \leq t$ (i.e. elements that are smaller than t under h chosen at random from H): $Pr[min_{x\in X}h(x) > t] \leq 48\left(\frac{6\cdot l'}{E_t}\right)^{(l'-1)/2}$.

When computing the fingerprint for the elements in X, we know $|X|$[2] and denoted $|X| = b$. Each h_i is γ-MWIF, so $E_t = \frac{tb}{p}$. Now consider choosing $t = \frac{12\cdot p\cdot l'}{b}$. Under this choice[3] of $t = \frac{12l'\cdot p}{b}$ we have $E_t = \frac{tb}{p} = 12l'$ and using the fact that $l' = 80 + 2\log\frac{1}{\epsilon}$ the above lemma can be rewritten as: $Pr[min_{x\in X}h(x) > t] < 48\left(\frac{6l'}{E_t}\right)^{(l'-1)/2} = 48\cdot\left(\frac{1}{2}\right)^{\frac{79}{2}}\cdot\left(\frac{1}{2}\right)^{2\log\frac{1}{\epsilon}} < \frac{1}{2^{33}}\cdot\epsilon^2$. There are k rows, and by applying the union bound we obtain: $Pr[\exists i \in [k](min_{x\in X}h_i(x) > t] < \frac{k\cdot\epsilon}{2^{33}} = \frac{8.02\cdot\epsilon^2}{2^{8.9}\cdot\epsilon^2} < \frac{1}{2^{29}}$.

We prove our algorithm runs in time $O(b\log\frac{1}{\epsilon} + \frac{1}{\epsilon^2}\log\frac{1}{\epsilon})$ with high probability. We have kb random values, $h_1(x_1), \ldots, h_{k-1}(x_b)$, which are (at least) pairwise independent. Denote $Y_{i,j}$ the indicator variable of the event that $h_j(x_i) < t = \frac{12pl'}{b}$, and so $Pr[Y_{i,j} = 1] = \frac{12l'}{b}$ and $E[Y_{i,j} = 1] = \frac{12l'}{b}$. Then $Y = \sum_{i=0}^{b}\sum_{j=0}^{k-1} Y_{i,j}$. The running time of the algorithm is $O(b\log\frac{1}{\epsilon} + Y)$. We show that $Y = O(\frac{1}{\epsilon^2}\log\frac{1}{\epsilon})$ with high probability[4]. We obtain: $E[Y] = E[\sum_{i=0}^{b}\sum_{j=0}^{k-1} Y_{i,j}] = \sum_{i=0}^{b}\sum_{j=0}^{k-1} E[Y_{i,j}] = 12\cdot l'\cdot k$. We use the following lemma, proven in the appendix: $Var(Y) \leq E(Y)$, and using Chebychev's inequality obtain: $Pr[Y > 11E(Y)] \leq Pr[|Y - E(Y)| > 10Var(Y)] < \frac{1}{100}$. To guarantee the required run time in a worst case analysis, we can drop all the blocks which require too long to compute. This reduces our probability of success in each block from $\frac{7}{8}$ to at least $\frac{7}{8} - 2^{-29} - \frac{1}{100}$ (The 2^{-29} factor is due to the probability that there exists a hash that gets a minimum value higher than t). Taking $4\log\frac{1}{\delta}$ blocks still obtains this probability. Overall the algorithm runs in time $O(b\log\frac{1}{\epsilon} + \frac{1}{\epsilon^2}\log\frac{1}{\epsilon})$ per block, or $O(\log\frac{1}{\delta}(b\log\frac{1}{\epsilon} + \frac{1}{\epsilon^2}\log\frac{1}{\epsilon}))$ for all blocks.

---

[2] We use this assumption for simplicity. If we don't know $|X|$, we can update the threshold t online. We store all elements until we have $\frac{\log\frac{1}{\delta}}{\epsilon^2}$ elements. Then we set t according to $b = 2\frac{\log\frac{1}{\delta}}{\epsilon^2}$. We double b by 2 each time $|X| > b$ and update t according to the new b.

[3] Notice that this constant is only to bound the worst case usually in a block the maximum between the minimal values is about l' moreover we can improve the running time if we drop from the sketch all the hash functions which there minimal value is to big.

[4] We base our calculation on the pairwise independence of $Y_{i,j}$. Notice that $Y_{i,j}$ is more independent when running over i. Therefor in practice the constants are smaller.

4.1 Computing The Minimal Elements of the Pseudo-Random Series

We give a recursive implementation of $pr - small - loc(f(x), g(x), k, x, t)$ and $pr - small - val(f(x), g(x), k, x, t)$, the procedures for computing V_t and I_t. Recall the hashes h_i were defined as $h_i(x) = f(x) + ig(x)$ where $f(x), g(x)$ are d-degree polynomials with random coefficients in $\mathbb{Z}_p$. Consider a given element $x \in \mathbb{Z}_p$ for which we attempt to find all the values (and indices) in $(h_0(x), h_2(x), \ldots, h_{k-1}(x))$ smaller than t. Given x, we can evaluate $f(x), g(x)$ in time $O(d) = O(\log \frac{1}{\gamma})$[5], and denote $a = f(x) \in \mathbb{Z}_p$ and $b = g(x) \in \mathbb{Z}_p$. Thus, we are seek all values in $\{a \mod p, (a+b) \mod p, (a+2b) \mod p, \ldots, (a+(k-1)b) \mod p\}$ smaller than t, and the indices i where they occur. Consider the series $S = (s_1, \ldots, s_k)$ where $s_i = (a+ib) \mod p$ and $i = \{0, 1, \ldots, k-1\}$. We denote the arithmetic series $a + bi \mod p$ for $i \in \{0, 1, \ldots, k-1\}$ as $S(a, b, k, p)$, so under this notation $S = S(a, b, k, p)$.

Given a value we can find the index where it occurs, and vice versa. To compute the value for index i, we compute $(a+ib) \mod p$. To compute the index i where a value v occurs, we solve $v = a + ib$ in $\mathbb{Z}_p$ (i.e. $i = \frac{v-a}{b} \mod p$). This can be done in $O(\log p)$ time using Euclid's algorithm. Note we compute b^{-1} in $\mathbb{Z}_p$ only once to transform *all* values to generating indices[6]. We call a location i where $s_i < s_{i-1}$ a *flip location*. The first index is a flip location if $a - b \mod p > a$. First, consider the case $b < \frac{p}{2}$. If s_i is a flip location, we have $s_{i-1} < p$ but $s_{i-1} + b > p$, so $s_i < b$. Also, since $b < \frac{p}{2}$ there is at least one location which is *not* a flip location between any two flip locations. Given $S = S(a, b, k, p)$, denote by $f(S)$ the flip locations in S.

Lemma 3 (Flip Locations Are Small). *When $b < \frac{p}{2}$, at most $\frac{k}{2}$ elements are flip locations, and all elements that are smaller than b are flip locations.*

Proof. Note that the non-flip locations between any two flip locations are monotonically increasing. Any flip location has a value of at most b, since the element before a flip location is smaller than p (modulo p), and adding b to it exceeds p, but through this addition it is impossible to exceed p by more than b.

We denoted by $f(S)$ the flip locations of S. Denote $f_0(S) = f(S)$. Denote by $f_1(S)$ all elements that occur directly after a flip location, $f_2(S)$ all elements that occur exactly two places after the closest flip locations (i.e they cannot be flip locations) and by $f_i(S)$ all elements that occur i places after the closest flip location.

Lemma 4 (Element Comparison). *When $b < \frac{p}{2}$, if $x \in f_i(S)$ and $y \in f_j(S)$ where $i > j$, then $x > y$.*

---

[5] Using multipoint evaluation we can calculate it in amortized time $O(\log^2 \log \frac{1}{\gamma})$. Moreover we can use other constructions for d-wise independent which can be evaluate in $O(1)$ time in the cost of using more space.

[6] We can store a table of inverse to further reduce processing time. If the required memory for the table is unavailable, we can do the computation in F_{p^c} for smaller p and store table of size p and then calculating the inverse requires $O(c \log c)$ time. Notice that we can easily take $c < \log_{\frac{\log \frac{1}{\delta}}{\epsilon^2}} u$ which will probably be less then $\log \frac{1}{\epsilon}$

Proof. All flip locations have a value of at most b. Due to Lemma 3, a location directly after a flip location is not a flip location, and is thus bigger than the flip location before it by exactly b, and is thus greater than b. Thus any element in $f_1(S)$ must be greater than any element in $f_0(S)$. Using the same argument, we see that any element in $f_2(S)$ is greater than any element in $f_1(S)$ and so on. A simple induction completes the proof.

The first flip location is $\lceil \frac{p-a}{b} \rceil$, as to exceed p we add $b \lceil \frac{p-a}{b} \rceil$ times. Also, the number of flip locations is $\lfloor \frac{a+bk}{p} \rfloor$. Denote the first flip location as $j = \lceil \frac{p-a}{b} \rceil$, with value $a' = (a + jb) \mod p$. Denote $b' = (b - p) \mod b$ and the number of flip locations as $k' = \lfloor \frac{(a+bk)}{p} \rfloor$. The flip locations are known to also be an arithmetic progression [18] [7].

Lemma 5 (Flip Locations Arithmetic Progression). *The flip locations of $S = S(a, b, k, p)$ are also an arithmetic progression $S' = (a', b', k', b)$.*

Given the above lemmas, we can search for the elements smaller than t, by examining the flip locations series in recursion. If case $b < t$, given $q = \lceil t \rceil b$, due to Lemma 4 $f(S), f_1(S), \ldots f_{q-1}(S)$ are smaller then t, and all of their elements must be returned. We must also scan $f_q(S)$ and also return all the elements of $f_q(S)$ which are smaller then t. This additional scan requires $O(|f_q(S)|)$ time $|f_q(S)| \leq |f(S)|$. Thus this case of $b < t$ examines $O(|I_t|)$ elements. Due to Lemma 3, if $b > t$, all non-flip locations are bigger than b and thus bigger than t, and thus we must only consider the flip-locations as candidates. Using Lemma 5 we can scan the flip locations recursively by examining the arithmetic series of the flip locations. If at most half of the elements in each recursion are flip locations, this results in a logarithmic running time. However, if b is high more than half the elements are flip locations. For the case where $b > \frac{p}{2}$ we can examine the same flip-location series S', in reverse order. The first element in the reversed series would be the last element of the current series, and rather than progressing in steps of b, we progress in steps of $p - b$. This way we obtain exactly the same elements, but in reverse order. However, in this reversed series, at most half the elements are flip locations. The following procedure implements the above method. It finds elements smaller then t in time $O(\log k) = O(\log \frac{1}{\epsilon} + |I_t|)$ where $|I_t|$ is the number of such values. Given the returned indices, we get the values in them. We use the same b for all $|I_t|$, so this can be done in time $O(c \log c + |I_t|)$ (Usually c is a constant).
$ps - min(a, b, p, k, t)$:

1. if $b < t$:
   (a) $V_t = []$
   (b) if $a < t$ then $V_t = V_t + [a + ib$ for i in range $(\lceil \frac{t-a}{b} \rceil)]$
   (c) $j = \lceil \frac{p-a}{b} \rceil$ // First flip (excluding first location)
   (d) while $j < k$:
      i. $v = (a + jb) \mod p$
      ii. while $j < k$ and $v < t$:
         A. V_t.append(v)
         B. $j = j + 1$
         C. $v = v + b$

[7] See Lemma 2 page 11.

iii. $j = j + \lceil \frac{p-v}{b} \rceil$ //next flip location
iv. return list1

(e) if $b > \frac{p}{2}$ then return $f((a + (k-1) \cdot b) \mod p, p - b, p, k, t)$
(f) $j = \lceil \frac{p-a}{b} \rceil$
(g) $new_k = \lfloor \frac{a+bk}{p} \rfloor$
(h) if $a < b$ then $j = 0$ and $new_k = new_k + 1$// calculate the first flip location and the number of flip locations
(i) return $f((a + jb) \mod p, -p \mod b, b, new_k, t)$

5 Conclusions

We have presented a fast method for computing fingerprints of massive datasets, based on pseudo-random hashes. We note that although we have examined the Jackard similarity in detail, the exact same technique can be used for any fingerprint which is based on minimal elements under several hashes. Thus we have described a general technique for exponentially speeding up computation of such fingerprints. Our analysis has used fingerprints using a single bit per hash. We have shown that even for these small fingerprints which can be quickly computed, the required number of hashes is asymptotically similar to previously known methods, and is logarithmic in the required confidence and polynomial in the required accuracy. Several directions remain open for future research. Can we speed up the fingerprint computation even further? Can similar techniques be used for computing fingerprints that are not based on minimal elements under hashes?

References

1. 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.
2. Y. Bachrach and R. Herbrich. Fingerprinting Ratings For Collaborative FilteringTheoretical and Empirical Analysis. 2010.
3. Y. Bachrach, R. Herbrich, and E. Porat. Sketching Algorithms for Approximating Rank Correlations in Collaborative Filtering Systems. In *String Processing and Information Retrieval*, pages 344–352. Springer, 2009.
4. Y. Bachrach, E. Porat, and J.S. Rosenschein. Sketching techniques for collaborative filtering. *IJCAI 2009*.
5. A.Z. Broder. On the resemblance and containment of documents. *In Compression and Complexity of Sequences (SEQUENCES'97*, 1998.
6. A.Z. Broder, M. Charikar, A.M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60(3):630–659, 2000.
7. A.Z. Broder, M. Charikar, and M. Mitzenmacher. A derandomization using min-wise independent permutations. *Journal of Discrete Algorithms*, 1(1):11–20, 2003.
8. E. Cohen, N. Duffield, H. Kaplan, C. Lund, and M. Thorup. Sketching unaggregated data streams for subpopulation-size queries. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, page 262. ACM, 2007.
9. E. Cohen and H. Kaplan. Summarizing data using bottom-k sketches. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, page 234. ACM, 2007.

10. M. Datar and S. Muthukrishnan. Estimating rarity and similarity over data stream windows. *AlgorithmsESA 2002*, pages 323–335.
11. Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.
12. P. Indyk. A Small Approximately Min-Wise Independent Family of Hash Functions. *Journal of Algorithms*, 38(1):84–90, 2001.
13. P. Indyk. Stable distributions, pseudorandom generators, embeddings, and data stream computation. *Journal of the ACM (JACM)*, 53(3):323, 2006.
14. R. Krauthgamer, A. Mehta, V. Raman, and A. Rudra. Greedy list intersection. In *IEEE 24th International Conference on Data Engineering, 2008. ICDE 2008*, pages 1033–1042, 2008.
15. P. Li and C. Konig. b-Bit minwise hashing. In *Proceedings of the 19th international conference on World wide web*, pages 671–680. ACM, 2010.
16. K. Mulmuley. Randomized geometric algorithms and pseudorandom generators. *Algorithmica*, 16(4):450–463, 1996.
17. M. Patrascu and M. Thorup. On the k-Independence Required by Linear Probing and Minwise Independence.
18. A. Pavan and S. Tirthapura. Range-efficient counting of distinct elements in a massive data stream. *SIAM Journal on Computing*, 37(2):359–379, 2008.

6 Appendix: Proofs

The proof of Theorem 1: $Pr_{h\in H}[m_i^h = m_j^h] = J_{i,j}$.

Proof. Denote $x = J_{1,2}$. The set $C_i \cup C_j$ contains three types of items: items that appear *only* in C_i, items that appear *only* in C_j, and items that appear in $C_i \cap C_j$. When an item in $C_i \cap C_j$ is minimal under h, i.e., for some $a \in C_i \cap C_j$ we have $h(a) = min_{x\in C_1\cup C_2}h(x)$, we get that $min_{x\in C_i}h(x) = min_{x\in C_j}h(x)$. On the other hand, if for some $a \in C_i \cup C_j$ such that $a \notin C_i \cap C_j$ we have $h(a) = min_{x\in C_1\cup C_2}h(x)$, the probability that $min_{x\in C_i}h(x) = min_{x\in C_j}h(x)$ is negligible [8]. Since H is MWIF, any element in $C = C_i \cup C_j$ is equally likely to be minimal under h. However, only elements in $I = C_i \cap C_j$ would result in $m_i^h = m_j^h$. Thus $Pr_{h\in H}[m_i^h = m_j^h] = \frac{1}{|C_i\cup C_j|} \cdot |C_i \cap C_j| = \frac{|C_i\cap C_j|}{|C_i\cup C_j|} = J_{i,j}$.

The proof of Theorem 4 (Simple Estimator for Jackard With Single Bit Per Hash): $Pr(|Y - J| \leq \epsilon) \geq \frac{7}{8}$.

Proof. Our proof uses Chebychev's inequality:

$$Pr(|X - E(X)| \geq \epsilon) \leq \frac{Var(X)}{\epsilon^2}$$

We have:

$$E(X) = E(\sum_{l=1}^{k} X_l) = \sum_{l=1}^{k} E(X_l) = k \cdot (J \pm \gamma)$$

[8] Such an event requires that two *different* items, $x_i \in C_i$ and $x_j \in C_j$ would be mapped to the same value $h^* = h(x_i) = h(x_j)$, and that this value would also be the minimal value obtained when applying h to both all the items in C_i and in C_j. As discussed in [12], the probability for this is negligible when the range of h is large enough.

$$(J-\gamma) \leq E(Y) \leq (J+\gamma)$$

We now bound $Var(X)$:

$$\begin{aligned}
Var(X) &= E(X^2) - E^2(X) \\
&= E((\sum_{l=1}^{k} X_l)^2) - E^2(\sum_{l=1}^{k} X_l) \\
&= E(\sum_{l=1}^{k} X_l^2 + 2\sum_{i\neq j} X_i X_j) - (E(\sum_{l=1}^{k} X_l))^2 \\
&= \sum_{l=1}^{k} E(X_l^2) + 2\sum_{i\neq j} E(X_i X_j) - (\sum_{l=1}^{k} E(X_l))^2 \\
&= \sum_{l=1}^{k} E(X_l^2) + 2\sum_{i\neq j} E(X_i X_j) - (\sum_{l=1}^{k} E(X_l)^2 + 2\sum_{i\neq j} E(X_i)E(X_j)) \\
&= \sum_{l=1}^{k} E(X_l^2) + 2\sum_{i\neq j} E(X_i)E(X_j) - (\sum_{l=1}^{k} E(X_l)^2 + 2\sum_{i\neq j} E(X_i)(X_j)) \\
&= \sum_{l=1}^{k} E(X_l^2) - \sum_{l=1}^{k} E(X_l)^2 \leq k
\end{aligned} \tag{1}$$

We use this to bound $Var(Y)$:

$$Var(Y) = Var(\frac{1}{k}\cdot X) = \frac{1}{k^2} Var(X) \leq \frac{1}{k^2}\cdot k \leq= \frac{1}{k}$$

Using Chebychev's inequality we get that:

$$Pr(|Y - E(Y)| > \beta) \leq \frac{Var(Y)}{\beta^2} \leq \frac{1}{k\cdot\beta^2}$$

Denote $\alpha = \frac{2^{10}-1}{2^{10}}$. Let $\beta = \alpha\cdot\epsilon$, so we obtain:
Thus using our choice of $k = \frac{8.02}{\epsilon^2}$ and $\beta = \alpha\cdot\epsilon$ (and noting that $J \leq 1, \epsilon \leq 1$) we have:

$$Pr(|Y - E(Y)| > \beta) \leq \frac{1}{k\beta^2} = \frac{1}{k\cdot\alpha^2\cdot\epsilon^2} = \frac{1}{8.0001} \leq \frac{1}{8}$$

Proof of Theorem 5 (Median Estimator for Jackard): $Pr(|\hat{J} - J| \leq \epsilon) \geq 1-\delta$.

Proof. We use Hoeffding's inequality [11]. Let $X_1, \ldots, X_n$ be independent random variables, where all X_i are bounded so that $X_i \in [a_i, b_i]$, and let $X = \sum_{i=1}^{n} X_i$. Hoeffding's inequality states that:

$$\Pr(X - \mathrm{E}[X] \geq n\epsilon) \leq \exp\left(-\frac{2\,n^2\,\epsilon^2}{\sum_{i=1}^{n}(b_i - a_i)^2}\right)$$

We say that the estimator $\hat{J}_l$ is *good* if $|\hat{J}_l - J| \leq \epsilon$ and that $\hat{J}_l$ is *bad* if $|\hat{J}_l - J| > \epsilon$. Each estimator $\hat{J}_l$ is bad with probability of $p \leq \frac{1}{8}$. Consider the random variable X_l where $X_l = 1$ if $\hat{J}_l$ is bad, and $X_l = 0$ if $\hat{J}_l$ is good. We have $Pr(X_l = 1) = p \leq \frac{1}{8}$, so $E(X_l) = p \leq \frac{1}{8}$. Denote $X = \sum_{l=1}^{m} X_l$, so $E(X) = m \cdot p \leq \cdot m \cdot \frac{1}{8}$. We now note that the $\hat{J}$ can be bad only if at least half the estimators $\hat{J}_1, \ldots, \hat{J}_m$ are bad, or in other words, when $X \geq \frac{m}{2}$.

The X_l's are independent, since for any x, y the hashes used to obtain the $\hat{J}_x$ are independent of the hashes used to obtain the $\hat{J}_y$. Since $p \leq \frac{1}{8}$ we have:

$$\Pr(X \geq \frac{m}{2}) \leq \Pr(X \geq (\frac{3}{8} + p) \cdot m) = \Pr(X - mp \geq \frac{3}{8}m)$$

However, $E(X) = mp$, so using Hoeffding's inequality, we require that $\Pr(X \geq \frac{m}{2}) \leq \delta$:

$$\Pr(X \geq \frac{m}{2}) \leq \Pr(X - mp \geq \frac{3}{8}m) \leq \exp(-2m \cdot \frac{9}{64}) \leq \delta$$

Extracting m we obtain that we require:

$$m > \frac{32}{9} \ln \frac{1}{\delta}$$

Proof of the lemma in Theorem 6:

Lemma 6. *Let $Y = \sum_{i=0}^{b} \sum_{j=0}^{k-1} Y_{i,j}$ in Theorem 6. Then $Var[Y] \leq E[Y]$.*

Proof.

$$\begin{aligned}
Var[Y] &= E[Y^2] - E^2[Y] = E[(\sum_{i=0}^{b}\sum_{j=0}^{k-1} Y_{i,j})^2] - E^2[\sum_{i=0}^{b}\sum_{j=0}^{k-1} Y_{i,j}] \\
&= E[\sum_{i=0}^{b}\sum_{j=0}^{k-1} Y_{i,j}^2 + 2\sum_{i'\neq i}\sum_{j'\neq j} Y_{i,j}Y_{i',j'}] - (\sum_{i=0}^{b}\sum_{j=0}^{k-1} E[Y_{i,j}])^2 \\
&= \sum_{i=0}^{b}\sum_{j=0}^{k-1} E[Y_{i,j}^2] + 2\sum_{i'\neq i}\sum_{j'\neq j} E[Y_{i,j}Y_{i',j'}] - (\sum_{i=0}^{b}\sum_{j=0}^{k-1} E[Y_{i,j}]^2 + 2\sum_{i'\neq i}\sum_{j'\neq j} E[Y_{i,j}][Y_{i',j'}]) \\
&= \sum_{i=0}^{b}\sum_{j=0}^{k-1} E[Y_{i,j}^2] + 2\sum_{i'\neq i}\sum_{j'\neq j} E[Y_{i,j}][Y_{i',j'}] - (\sum_{i=0}^{b}\sum_{j=0}^{k-1} E[Y_{i,j}]^2 + 2\sum_{i'\neq i}\sum_{j'\neq j} E[Y_{i,j}][Y_{i',j'}]) \\
&= \sum_{i=0}^{b}\sum_{j=0}^{k-1} E[Y_{i,j}^2] - \sum_{i=0}^{b}\sum_{j=0}^{k-1} E[Y_{i,j}]^2 = \sum_{i=0}^{b}\sum_{j=0}^{k-1} E[Y_{i,j}] - \sum_{i=0}^{b}\sum_{j=0}^{k-1} E[Y_{i,j}]^2 \\
&= E[Y] - \sum_{i=0}^{b}\sum_{j=0}^{k-1} E[Y_{i,j}]^2 \leq E[Y]
\end{aligned} \tag{2}$$