No Repetition: Fast and Reliable Sampling with Highly Concentrated Hashing

Anders Aamand
MIT
Massachusetts, USA
aamand@mit.edu

Debarati Das
Penn State University
Pennsylvania, USA
debaratix710@gmail.com

Evangelos Kipouridis
BARC, University of Copenhagen
Copenhagen, Denmark
kipouridis@di.ku.dk

Jakob B. T. Knudsen
BARC, University of Copenhagen
Copenhagen, Denmark
jakn@di.ku.dk

Peter M. R. Rasmussen
BARC, University of Copenhagen
Copenhagen, Denmark
pmrrasmussen@icloud.com

Mikkel Thorup
BARC, University of Copenhagen
Copenhagen, Denmark
mikkel2thorup@gmail.com

ABSTRACT

Stochastic sample-based estimators are among the most fundamental and universally applied tools in statistics. Such estimators are particularly important when processing huge amounts of data, where we need to be able to answer a wide range of statistical queries reliably, yet cannot afford to store the data in its full length.

In many applications we need the sampling to be coordinated which is typically attained using hashing. In previous work, a common strategy to obtain reliable sample-based estimators that work within certain error bounds with high probability has been to design one that works with constant probability, and then boost the probability by taking the median over r independent repetitions. Aamand et al. (STOC'20) recently proposed a fast and practical hashing scheme with *strong concentration bounds*, Tabulation-1Permutation, the first of its kind. In this paper, we demonstrate that using such a hash family for the sampling, we achieve the same high probability bounds without any need for repetitions. Using the same space, this saves a factor r in time, and simplifies the overall algorithms.

We validate our approach experimentally on both real and synthetic data. We compare Tabulation-1Permutation with other hash functions such as strongly universal hash functions and various other hash functions such as MurmurHash3 and BLAKE3, both with and without resorting to repetitions. We see that if we want reliability in terms of small error probabilities, then Tabulation-1Permutation is significantly faster.

PVLDB Reference Format:
Anders Aamand, Debarati Das, Evangelos Kipouridis, Jakob B. T. Knudsen, Peter M. R. Rasmussen, and Mikkel Thorup. No Repetition: Fast and Reliable Sampling with Highly Concentrated Hashing. PVLDB, 15(13): 3989 - 4001, 2022.
doi:10.14778/3565838.3565851

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/kipoujr/no_repetition.

All authors contributed equally.

Proceedings of the VLDB Endowment, Vol. 15, No. 13 ISSN 2150-8097.
doi:10.14778/3565838.3565851

1 INTRODUCTION

Recent years have brought with them a huge demand for algorithms that can process and reliably compute statistics of large sets of data. However, in many practical applications, the sheer volume of the data makes it impossible to store a complete copy and perform exact computations. We therefore have to resort to estimation algorithms. For instance, instead of precisely counting the number of unique visitors to a website, we may settle for a good estimate.

Bernoulli sampling is one of the most basic ways to reduce the size of sets of data, while still having access to such reliable estimators. Given some set X, we sample a subset S by independently including each element of X with probability p. The basic idea is then similar to the one behind election polls: In order to understand a given statistic of X, (e.g., the number of people of X who supports a certain political party), we calculate the same statistic for the sampled set S, scale it appropriately, and use that as our estimator. To be more precise, given any selection Y from X of size n, the random variable $|Y \cap S|$ is binomially distributed as $Bin(n, p)$, which is sharply concentrated around its mean $|Y|p$. Therefore, $|Y \cap S|/p$ will be an unbiased estimator for $|Y|$ lying within a small error bound with high probability.

There are many ways of using the above basic idea, e.g., in a streaming context where elements from X arrive as a stream of unknown length, $x_1, x_2, \ldots$. In this case, we may decrease p as X grows. In this setting, the idea is to generate a random r_i uniformly in $(0, 1]$ for each i, and select x_i if $r_i < p$. We refer to this as *threshold sampling* with threshold p. If the sampled set grows too large, we may decrease p (and discard elements from S having a too large r_i). The decrease could be by a constant factor, but alternatively we could use a bottom-k sample of the distinct keys seen so far, which corresponds to having p to be the $(k+1)$-th smallest value of the r_i. A beautiful result from order statistics is that with this choice of p, $1/p$ has expectation n/k, and further that for any selection Y from X, $\mathbb{E}\left[|Y \cap S|/p\right] = |Y|$.

Now, in many contexts, we need coordinated sampling that is determined by a hash function $h : U \to (0, 1]$. In the above, we would have $r_i = h(x_i)$. Below we discuss three examples.

(1) **Set Intersection.** Suppose we have samples S_X and S_Y from two different sets X and Y, and want to estimate their intersections. Then $X \cap Y$ is just a selection from $X \cup Y$, so we can estimate $|X \cap Y|$ as $|S_X \cap S_Y|/p$. This is the underlying idea of Broder's original min-hash algorithm [6].
(2) **Distinct elements.** In streaming, coordinated sampling is particularly important, as the same element x may appear many times. Using hashing for the sampling, if x is picked once, it is picked every time because the hash value of x does not depend on when x appears. We can therefore estimate the number of distinct elements by looking at the number of distinct elements in the sample. This is the underlying idea of the classic algorithm by Bar-Yossef *et al.* [3].
(3) **Trajectory sample [12].** Suppose we want to sample packet paths through a network, e.g., to determine the source of the traffic using a given link. If all routers sample with the same hash function, then if a packet is sampled once, it will be sampled by all the routers, so its whole path is collected.

More examples can be found in [8], including other network problems, and application domains such as document-features and market-basket datasets.

In the simple Bernoulli case where each key is sampled independently with probability p, the size of the sample S is distributed as $Bin(n, p)$. The strong concentration of the estimates in the examples above then hinge on the concentration of $Bin(n, p)$ around its mean. However, the size of the sample only has this distribution if each key is sampled independently, i.e., if all the hash values of the distinct keys are independent, but such highly independent hashing is impossible to implement in practice. Classic fast hash functions such as $(ax + b) \mod p$ are only 2-independent. On the positive side, the 2-independence ensures that the variance of $|Y \cap S|$ is the same as if h was fully random (namely $|Y|p(1 - p)$), and hence we get concentration according to Chebyshev's inequality. However, in the tail, Chebyshev is much weaker than the classic Chernoff bounds which we get with independent Bernoulli sampling.

To get estimators that provably work with high probability within a certain error bound, a common strategy is to design one that works with constant probability, and then boost the probability by taking the median over r independent repetitions. To explain this, consider a selection Y from X of size n. If we use a 2-independent hash family for sampling elements from X with probability, say, $p = 4/(\varepsilon^2 n)$, then Var $[|Y \cap S|] \leq 4/\varepsilon^2$ and it follows from Chebychev's inequality that the estimator $|Y \cap S|/p$ is within a $1 \pm \varepsilon$ factor of n with probability at least $3/4$. By performing r independent repetitions (e.g., with hash functions $h_i(x) = (a_i x + b_i) \mod p$, where the a_i's and b_i's are all mutually independent) and taking the median of the estimators, the error probability drops exponentially in r. However, running r independent experiments increases the processing time by a factor r.

Recently Aamand *et al.* [1] introduced a new practical hashing scheme, Tabulation-1Permutation, which provably satisfies so called *strong concentration bounds* (to be defined in Section 2.2) akin to those which hold for sums of Bernoulli variables, and even more recently Houen and Thorup [16] showed that also the Mixed-Tabulation scheme from [9] enjoys such bounds. The main theoretical contribution of this paper is demonstrating that using such a hashing scheme, we get the same high probability bounds on the errors of the estimators without any need for repetitions. Instead of making r independent samples, we sample a single set which is $\Theta(r)$ times larger. The total space usage is therefore essentially the same. However, we only apply a single hash function, processing each element in constant time regardless of r, with an altogether significantly simpler algorithm. The hashing schemes of [1, 16] provide concentration bounds for Bernoulli trials, where each key is sampled based on its own hash value. This leads to a variable number of sampled keys. However, if one wants a fixed number of samples, then one can use the union bound approach of [20] to obtain similar bounds for more convoluted sampling schemes like bottom-k sampling [8] and priority sampling [13]. We will see an example of this in Section 4.

As we will see examples of, using a fast hashing scheme with strong concentration bounds to remove the need for independent repetitions, can also help speed up streaming algorithms. To illustrate a streaming scenario where the processing time is critical, consider the Internet. Suppose we want to process packets passing through a high-end Internet router. Each application only gets very limited time to look at the packet before it is forwarded. If it is not done in time, the information is lost. Since processors and routers use some of the same technology, we never expect to have more than a few instructions available. Slowing down the Internet is typically not an option. The papers of Krishnamurthy *et al.* [18] and Thorup and Zhang [21] explain in more detail how high speed hashing is necessary for their Internet traffic analysis. Incidentally, the hash function we use from [1] is a bit faster than the ones from [18, 21], which do not provide strong concentration bounds.

1.1 Experiments

To demonstrate that our approach works well in practice, both in terms of speed and reliability, we perform experiments on the use of Tabulation-1Permutation and Mixed-Tabulation for sampling. We have chosen threshold sampling and bottom-k sampling for our algorithms, see Section 3 and 4 for details. As it was slightly faster, we here focus our discussion on Tabulation-1Permutation, but the two schemes had very similar experimental performances in terms of accuracy. We compare with the fastest known strongly universal hash functions and with other commonly used hash functions such as MurmurHash3 [2] and BLAKE3 [17]. Without the use of independent repetitions, we demonstrate that Tabulation-1Permutation provides more reliable estimates than the fast strongly universal hash functions. Moreover, the implementation with Tabulation-1Permutation is faster than when using MurmurHash3 and BLAKE3. In fact, BLAKE3 was approximately 150 times slower than Tabulation-1Permutation, so we disregard it in our experiments. On the other hand, the implementation with Tabulation-1Permutation is both faster and provides better estimates than when implementing the algorithm with the strongly universal hash functions and independent repetitions.

We include two figures from Section 6 (which will be explained in more detail in that section). Figure 1a shows the relative error incurred by different hash functions over 5×10^4 experiments on a synthetic data set. We see that when implementing the algorithms with classic strongly universal hash functions like Multiply-Shift and Multiply-Mod-Prime, some of the estimates are significantly off. Figure 1b shows the running time per experiment for each of the hash functions tested. We see that Multiply-Shift is the only hashing scheme that outperforms Tabulation-1Permutation in regards to speed. However, in order to eliminate the outliers of Multiply-Shift seen in Figure 1a, we need to run independent repetitions, making Tabulation-1Permutation much faster. See Section 6 for details.

In summary, our experiments show that the fastest known strongly universal hashing schemes fail on simple bad instances. Remedying this with independent repetitions, the implementation with just a single Tabulation-1Permutation function becomes faster and gives more reliable estimates. Finally, one could hope that popular hash functions like MurmurHash3 or cryptographic hash functions, would perform well on most data[1]. However, Tabulation-1Permutation is simply faster and, as shown in [1], they *provably* performs well on any given data set with high probability. To the best of our knowledge, we are the first to provide an experimental understanding of the large outliers that occur with weaker hashing schemes (and their non-existence with hashing with strong concentration). Our extensive experimental evaluation of the reliability of the tabulation-based hashing schemes for sampling is the second main contribution of the paper.

Remark. An interesting statistical artifact appearing in our experiments is the following. Looking at the accuracy of the estimators when implemented with Multiply-Shift and Multiply-Mod-Prime, it appears that for certain structured data sets, they provide even better estimates than fully random hashing would. This could lead to the false impression that their performance is *always* better than implementing the estimators with e.g., Tabulation-1Permutation. Since the variance of the estimators are the same for all the seeded hash functions, it follows that if a hashing scheme yields estimators that are 'too good', it must inevitably fail occasionally and provide estimates that are far off. We will see examples of this in Section 6 and discuss it further in Section 6.5.

2 HASHING AND CONCENTRATION

In the present paper, the crucial component of our algorithms is a random hash function $h\colon U \to [0,1)$ mapping some key universe U, e.g. 64-bit keys, uniformly into $R = [0,1)$. The application is the following. Let $S \subset U$ be a set of keys and $p \in [0,1)$ a threshold value. Define $X = X^{<p} = |\{s \in S \mid h(s) < p\}|$, the number of keys from S that hash below p. We wish to estimate the size of S based on the size of X. Here p could be an unknown function of S, but p should be independent of the random hash function h. Since the mean μ of X is $\mathbb{E}[X] = |S|\,p$, we may estimate the size of S by X/p with precision increasing in the concentration of X around its mean. In particular, we are interested in the probability δ that X deviates from μ by more than a factor $\varepsilon > 0$, i.e., the probability $\delta = \Pr[|X - \mu| \geq \varepsilon\mu]$.

[1]However, it is important to note that no amount of experimenting can prove that a hashing scheme performs well on all possible data, and finding the problematic data sets for a given hash family is often a non-trivial task. For Multiply-Shift and Multiply-Mod-Prime, we have concrete examples of data sets on for which they fail, as is evident from Figure 1a. Moreover, in [5] the authors provide concrete bad data sets for MurmurHash3.

If the hash function h is fully random, we get the classic Chernoff concentration bounds on X (see, e.g, [19]):

$$\Pr[X \geq (1+\varepsilon)\mu] \leq \exp(-\varepsilon^2\mu/3) \text{ for } 0 \leq \varepsilon \leq 1, \tag{1}$$

$$\Pr[X \leq (1-\varepsilon)\mu] \leq \exp(-\varepsilon^2\mu/2) \text{ for } 0 \leq \varepsilon \leq 1. \tag{2}$$

Unfortunately, we cannot implement fully random hash functions as it requires space as big as the universe.

2.1 Strongly Universal Hashing

To get something implementable in practice, Wegman and Carter [22] proposed strongly universal hashing.

Definition 2.1 (Strongly Universal Hashing). A hash function $h\colon U \to R$ is *strongly universal* if for every pair of distinct keys $x, y \in U$, the distribution of $(h(x), h(y))$ is uniform on R^2.

Many common hash functions are strongly universal. Worth mentioning is the Multiply-Shift hash function [11]. The textbook example of a strongly universal hash function hashing into $[0,1)$ is the Multiply-Mod-Prime hash function [7]. The Multiply-Mod-Prime hash function picks a large prime $\wp$ and two uniformly random numbers $a, b \in \mathbb{Z}_\wp$. Then $h_{a,b}(x) = ((ax+b) \bmod \wp)/\wp$ is strongly universal from $U \subseteq \mathbb{Z}_\wp$ to $R = \{i/\wp | i \in Z_\wp\} \subset [0,1)$. Obviously this hash function is not uniform on $[0,1)$ as we considered above, but for any $p \in [0,1)$, we have $\Pr[h(x) < p] \approx p$ with equality if $p \in R$. Below we ignore this deviation from uniformity on $[0,1)$.

Assuming we have a strongly universal hash function $h : U \to [0,1)$, we again let X be the number of elements from S that hash below p. Then $\mu = \mathbb{E}[X] = |S|p$ and because the hash values are 2-independent, we have $\mathrm{Var}[X] \leq \mathbb{E}[X] = \mu$. Therefore, by Chebyshev's inequality,

$$\Pr[|X - \mu| \geq \varepsilon\mu] \leq 1/(\varepsilon^2\mu). \tag{3}$$

As $\varepsilon^2\mu$ gets large, the error probability of strongly universal hashing is much higher than the Chernoff bounds with fully random hashing. However, as described in Section 3.1, it is still possible to guarantee high concentration by aiming for a constant error probability like $\delta = 1/4$ and then using the median over independent repetitions of the estimation to reduce the error probability.

2.2 Strongly Concentrated Hashing

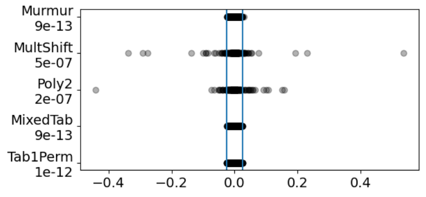
In this paper we discuss the benefits of hash functions with strong concentration akin to that of fully random hashing.

Definition 2.2. A hash function $h : U \to [0,1)$ is *strongly concentrated with added error probability* $\mathcal{E}$ if for any key set $S \subseteq U$, threshold $p \in [0,1)$, and $0 < \varepsilon \leq 1$, the number X of elements from S hashing below p satisfies

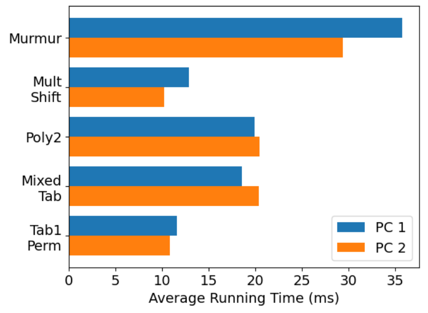
$$\Pr[|X - \mu| \geq \varepsilon\mu] = 2\exp(-\Omega(\varepsilon^2\mu)) + \mathcal{E},$$

where $\mu = p|S|$. If $\mathcal{E} = 0$, we say that h is *strongly concentrated.*

It is worth noting that a fully random hash function is strongly concentrated. Another way of viewing the added error probability $\mathcal{E}$ is as follows. We have strong concentration as long as we do not aim for error probabilities below $\mathcal{E}$, so if $\mathcal{E}$ is sufficiently low, we



(a) The relative error when estimating the size of a set based on its sample with various hash functions and no repetitions.



(b) The average running time per experiment with the different hash functions.

Figure 1: Relative error and timing when estimating the size of a set based on a sampled subset. These experiments ran on synthetic data and did not use independent repetitions.

can simply ignore it. In other words, except for an error term of $\mathcal{E}$, the hash function performs asymptotically as well as a random hash function.

What makes this definition interesting in practice is that Aamand et al. [1] recently presented a practical hash function with small constant running time that for a universe $U = [u] = \{0, \ldots, u-1\}$ is strongly concentrated with added error probability $u^{-\gamma}$ for any constant γ. For our applications, this term is so small that we can ignore it as the universe sets we consider are huge.

2.3 Tabulation-1Permutation

The hash function we consider in this paper is Tabulation-1Permutation, introduced by Aamand et al. [1]. For the applications of this paper it enjoys the benefits of strong concentration with a negligible added error probability, comparable in speed to the fastest hash functions on the market, and simple to implement[2].

2.3.1 Implementation. Tabulation-1Permutation obtains its power and speed using certain character tables in fast cache. The scheme views a keys from a universe $U = [u] = \{0, \ldots, u-1\}$ as consisting of a small, constant number c of characters from some alphabet Σ, that is, $U = \Sigma^c$. For 64-bit keys, this could be $c = 4$ characters of 16 bits each. For our applications, we only consider the case where the hash values belong to the same universe U, and interpret them as numbers in $[0, 1)$ by dividing with u. Tabulation-1Permutation needs $c + 1$ character tables mapping characters to hash values. To compute the hash value of a key, we make one lookup for each of the $c + 1$ tables and perform $O(c)$ fast AC^0 operation to extract the characters and XOR the hash values. The character tables can be populated with an $O(\log u)$-independent pseudo-random number generator, needing a random seed of $O(\log^2 u)$ bits. Tabulation-1Permutation is simple to implement and evaluate (see Figure 2). It takes up a dozen lines of code and is extremely fast. For details on the inner workings of Tabulation-1Permutation see [1]. For details regarding its running time in practice, see Section 6 as well as [1, Section 1.7].

[2]We focus our discussion on Tabulation-1Permutation, but recently Houen and Thorup [16] showed that also the Mixed-Tabulation from [9] (which is equally simple to implement and of similar speed) satisfies such bounds. We will also see experiments with Mixed-Tabulation.

```
uint64_t T[4][65536];
uint64_t P[65536];

uint64_t Tab1Perm(uint64_t x) {
    uint64_t y; int i;
    y = 0;
    for (i = 0; i < 4; ++i) {
        y ^= R[i][(uint16_t) x];
        x = x >> 16;
    }
    return y ^ P[(uint16_t) y];
}
```

Figure 2: The C-code for the evaluation of Tabulation-1Permutation with 4 characters for 64-bit keys.

2.3.2 Strong Concentration. The exact concentration results regarding Tabulation-1Permutation [1, Theorem 1.3] are far too general for our purposes. We instead state a version simplified for the purposes of this exposition. Here we identify a hash value from $[u]$ as a fraction in $[0, 1)$.

Theorem 2.3. *Let $h\colon [u] \longrightarrow [0, 1)$ be a Tabulation-1Permutation hash function with $[u] = \Sigma^c$, $c = O(1)$, and let $\gamma > 0$ be fixed. For any key set $S \subset [u]$, threshold $p \in [0, 1)$, and $0 < \varepsilon \leq 1$, the number X of elements from S hashing below p satisfies*

$$\Pr[|X - \mu| \geq \varepsilon\mu] = 2\exp(-\Omega(\mu\varepsilon^2)) + 1/u^{\gamma}.$$

Thus, for our applications, we are theoretically guaranteed that Tabulation-1Permutation will perform asymptotically as well as a random hash function up to an additive error term of $1/u^{\gamma}$. Importantly, in said applications the universe size is huge, so the error term is indeed negligible.

2.3.3 Computer Dependent Versus Problem Dependent Resources. We view the resources used for Tabulation-1Permutation as computer dependent rather than problem dependent. More precisely, you should pick the number of characters, c, and the size of the character alphabet, $|\Sigma|$, depending on the computer's architecture.

For hashing 64-bit keys one often picks either $c = 4$ or $c = 8$ which yields a space usage of $9 \times 2^8 \times 8$ bytes (less than 20 KB) and $5 \times 2^{16} \times 8$ bytes (less than 3 MB), respectively. An important property is that once the tables are populated they will never be overwritten. This means that the cache does not get dirty, that is, different computer cores can access the tables and not worry about consistency.

3 INDEPENDENT REPETITIONS VS STRONGLY CONCENTRATED HASHING

Suppose we are given a data set $X \subseteq U$ and a selection $Y \subseteq X$ of some unknown size $|Y| = n$. In this section, we discuss the theoretical guarantees that can be obtained when estimating the size of Y using hashing-based sampling. We are given parameters $0 < \varepsilon, \delta < 1$, and the goal is to create an estimator, $\hat{n}$ such that $|n - \hat{n}| \le \varepsilon n$ with probability at least $1 - \delta$. For instance, aiming to avoid big errors over many estimates, we could set $\varepsilon = 100\%$, and $\delta = 2^{-20}$. Given a hash function $h : U \longrightarrow [0, 1)$, and a $p \in [0, 1)$, we define the sample $S_Y = \{y \in Y \mid h(y) < p\} = \{x \in X \mid h(x) < p\} \cap Y$ consisting of the elements from Y that hash below p. Let further $Y^{<p} = |S_Y|$.

We will compare the three cases where (1) h is a strongly universal hash function and we use independent repetitions, (2) h is truly random, and (3) h satisfies strong concentration bounds. Our analysis demonstrates that strongly concentrated hash functions provide simpler algorithms and stronger theoretical guarantees (akin to those of truly random hash functions) than if we had used universal hashing and independent repetitions.

3.1 Strongly Universal Hashing and Independent Repetitions

This is perhaps the most common strategy for getting theoretical guarantees on the quality of the estimates. In this case, the hash values of any two distinct keys are independent, so as noted in Section 2.1, $\mathrm{Var}\left[Y^{<p}\right] \le \mathbb{E}\left[Y^{<p}\right] = np$ for every $0 < p < 1$. Thus, applying Chebyshev's inequality (as we did in (3)),

$$\Pr\left[|\hat{n} - n| \ge \varepsilon n\right] \le 1/(\varepsilon^2 pn)$$

To get the desired error probability δ, one option would be to set[3] $p = 1/(\delta\varepsilon^2 n)$. However, if δ is small, e.g., $\delta = 1/u$, then the sample S becomes much too large. To solve this issue, the standard approach is, as in [3], to apply classic *median trick*. Instead of aiming for a small error probability right away, we start by aiming for a constant error probability δ_0. Here we use $\delta_0 = 1/4$ for simplicity. Then it suffices to set $p = 4/(\varepsilon^2 n)$. With this choice of p and δ_0, we now sample r independent estimators for n, $\hat{n}_1, \dots, \hat{n}_r$, for some r to be determined later, by repeating the algorithm r times with fresh randomness. Our final estimator $\hat{n}$ is then the median of $\hat{n}_1, \dots, \hat{n}_r$.

Now, for each $1 \le i \le r$, $\Pr[|\hat{n}_i - n| \ge \varepsilon n] \le 1/4$ and these events are independent. If $|\hat{n} - n| \ge \varepsilon n$, then $|\hat{n}_i - n| \ge \varepsilon n$ for at least half of the $1 \le i \le r$. By the standard Chernoff bound (1), this probability can be bounded by

$$\Pr\left[|\hat{n} - n| \ge \varepsilon n\right] \le \exp(-(r/4)/3) = \exp(-r/12).$$

[3]As n is unknown to us, it is problematic to choose p depending on n when designing the estimator in practice. One way to handle this, is to through bottom-k sampling. We will discuss this in detail in Section 4.1 but for simplicity of the exhibition, we ignore this subtlety for now.

Setting $r = 12\ln(1/\delta)$, we get the desired error probability δ. The expected number of hash values stored is then

$$rpn = 48\ln(1/\delta)/\varepsilon^2 = \Theta(\ln(1/\delta)/\varepsilon^2).$$

3.2 Utopia: Fully Random Hashing

Suppose that we could implement a fully random hash function $h \colon U \longrightarrow [0, 1)$. In that case, sampling using h would yield excellent guarantees. Indeed, the Chernoff bounds (1) and (2) yield that

$$\Pr[|\hat{n} - n| \ge \varepsilon n] \le 2\exp(-\varepsilon^2 pn/3). \tag{4}$$

Thus, to get error probability δ, we just choose $p = 3\ln(2/\delta)/(\varepsilon^2 n)$, storing $3\ln(2/\delta)/\varepsilon^2$ hash values in expectation. There are several reasons why this is much better than the above approach using 2-independence and independent repetitions. It avoids the independent repetitions, so instead of applying $r = \Theta(\log(1/\delta))$ hash functions to each key we just need one. Moreover, with independent repetitions, we are tuning the algorithm depending on ε and δ, whereas with a fully-random hash function, we get the concentration from (4) for every $0 < \varepsilon \le 1$.

The only caveat is that fully-random hash functions cannot be implemented in practice. In some applications, cryptographic hash functions, or popular hash functions such as MurmurHash3 [2] have been applied. On datasets which have high entropy, such hash functions will perform well and appear "random", however, there is no guarantee that this will hold for every choice of data set. Importantly, it is impossible to predict how such a hash function will do on a particular structured data set.

3.3 Strongly Concentrated Hashing

Abandoning the infeasible fully random hashing, let instead $h \colon U \longrightarrow [0, 1)$ be a strongly concentrated hash function with error term $\mathcal{E}$. It then follows as above that for $0 < \varepsilon \le 1$,

$$\Pr\left[|\hat{n} - n| \ge \varepsilon n\right] = 2\exp\left(-\Omega\left(pn\varepsilon^2\right)\right) + O(\mathcal{E}). \tag{5}$$

To obtain the error probability $\delta = \omega(\mathcal{E})$, we again need to set $p = O(\log(1/\delta)/(\varepsilon^2 n))$, thus storing $O(\log(1/\delta)/\varepsilon^2)$ hash values in expectation. Within a constant factor this means that we use the same total number using 2-independence and independent repetitions, and we still retain the following advantages from the fully random case.

- By avoiding repetitions, we save a factor $\Theta(\log(1/\delta))$ in running time.
- We avoid tuning the algorithm for a given ε and δ. Instead we get the concentration from (5) for every $0 < \varepsilon \le 1$.

3.3.1 Instantiating with Tabulation-1Permutation. Let us relate the above discussion to the Tabulation-1Permutation hash function by Aamand et al. [1]. It follows by Theorem 2.3 that sampling using Tabulation-1Permutation would imply that for any $\gamma > 0$,

$$\Pr\left[|\hat{n} - n| \ge \varepsilon n\right] = 2\exp\left(-\Omega\left(pn\varepsilon^2\right)\right) + O(1/u^\gamma).$$

Since the universe size u is usually huge, the error term $O(1/u^\gamma)$ is negligible. Hence, from a practical perspective, the concentration is as good as fully random up to constant factors in the exponent of the exponential tail.

4 BOTTOM-k SAMPLING

In the previous section, we have chosen p depending on n in order to obtain the desired (ε, δ)-type bound. This may seem problematic as n is the quantity we wish to estimate. In practice, p is often just a fixed number (which could be determined by the allotted resources), and then the quality of the error of course depends on n. However, if we insist on getting estimators such that $\Pr[|\hat{n} - n| > \varepsilon n] \le \delta$, for some specific $0, \varepsilon, \delta \le 1$, then we can use the idea of a bottom-k sample, i.e., a sample that consists of the k elements from a given set X with the smallest hash values. Using such a sample requires a more careful analysis. We will analyse algorithms for the two problems of estimating the number of distinct elements in a stream and for estimating set intersection, when using strongly concentrated hashing to create the bottom-k sample.

For the distinct elements problem, several other very efficient algorithms and estimators are known, e.g., the HyperLogLog Algorithm [14]. For more details, a thorough survey by Harmouch and Naumann [15] provides experimental data on the choice of algorithm. The analysis we provide here, merely serves as a proof of concept and a warm up for our analysis of Broder's algorithm for set intersection [6] with strongly concentrated hashing.

4.1 Counting Distinct Elements with Strongly Concentrated Hashing

Consider a sequence (or stream) of keys $x_1, \ldots, x_s \in U$ from some universe U, where each element may appear multiple times. The problem of *counting distinct elements* in such a stream is the following. Using only little space, we wish to estimate the number n of distinct keys in the stream. We are given parameters $0 < \varepsilon, \delta < 1$, and the goal is to create an estimator, $\hat{n}$ such that $(1-\varepsilon)n \le \hat{n} \le (1+\varepsilon)n$ with probability at least $1 - \delta$.

In our estimation, we follow the classic approach of Bar-Yossef et al. [3], which was later revised by Beyer et al. [4] to introduce their unbiased version of the estimator. We proceed as follows. Let $h\colon U \to [0,1)$ be a random hash function, and assume for simplicity that h is collision-free over U. For some $k > 1$, assumed to be significantly smaller than n, we process each element x_i in order, maintaining the k smallest distinct hash values $h(x_i)$ of the stream. Let $x_{(k)}$ be the key with the kth smallest hash value under h, and let $h_{(k)} = h\left(x_{(k)}\right)$. As in [3], our estimator for n is then $\hat{n} = k/h_{(k)}$. It is worth noting that [3] suggests several other estimators, but the points we will make below apply to all of them. We call this the bottom-k estimator for counting distinct elements.

The point of using a hash function h is that all occurrences of a given key x in the stream get the same hash value. Thus, if S is the set of distinct keys, $h_{(k)}$ is the kth smallest hash value from S. In particular, $\hat{n}$ depends only on S, not on the frequencies of the elements of the stream.

We would like $\hat{n} = k/h_{(k)}$ to be concentrated around n. For any probability $p \in [0,1]$, let $X^{<p}$ denote the number of elements from S that hash below p. Let $p_- = k/((1+\varepsilon)n)$ and $p_+ = k/((1-\varepsilon)n)$. Note that both p_- and p_+ are independent of the random hash function h. Now,

$$\hat{n} = k/h_{(k)} \le (1-\varepsilon)n \iff X^{<p_+} < k = (1-\varepsilon)\mathbb{E}\left[X^{<p_+}\right], \quad (6)$$

$$\hat{n} = k/h_{(k)} > (1+\varepsilon)n \iff X^{<p_-} \ge k = (1+\varepsilon)\mathbb{E}\left[X^{<p_-}\right]. \quad (7)$$

These observations form a good starting point for applying probabilistic tail bounds. Indeed, if $h\colon U \to [0,1)$ is a strongly concentrated hash function with error term $\mathcal{E}$, then instantiating the bottom-k estimator with h, it follows from (6) and (7) that for $0 < \varepsilon \le 1$,

$$\Pr\left[\hat{n} \le (1-\varepsilon)n\right] = \exp\left(-\Omega\left(\frac{k\varepsilon^2}{1-\varepsilon}\right)\right) + \mathcal{E},$$

$$\Pr\left[\hat{n} \ge (1+\varepsilon)n\right] = \exp\left(-\Omega\left(\frac{k\varepsilon^2}{1+\varepsilon}\right)\right) + \mathcal{E},$$

so

$$\Pr\left[|\hat{n} - n| \ge \varepsilon n\right] = 2\exp\left(-\Omega\left(k\varepsilon^2\right)\right) + O(\mathcal{E}). \quad (8)$$

To obtain the error probability $\delta = \omega(\mathcal{E})$, we pick $k = O(\log(1/\delta)/\varepsilon^2)$. We thus store $k = O(\log(1/\delta)/\varepsilon^2)$ hash values. In the alternative implementation with 2-independence, we would instead apply Chebychev, pick $k = O(1/\varepsilon^2)$ for a constant error probability, and then boost the error probability to δ using $O(\log(1/\delta))$ independent repetitions and the median trick. Within a constant factor this means that we store the same total number, and we still retain the advantages from the fully random, namely avoiding the independent repetitions and tuning the algorithm to ε and δ.

4.2 Implementing Bottom-k and an Alternative

Implementing the bottom-k estimator in practice requires maintaining the k smallest hash values. The most obvious and widely used approach is to use a priority queue. For instance, this is used in the survey of Harmouch and Naumann [15]. If the input arrives in random order, we need to update the priority queue $O(k \log n)$ times, so the total running time is $O(n + k \log k \log n)$. With the reasonable assumption that $k = O(n/\log^2 n)$, this is $O(n)$.

Unfortunately, it is not reasonable to assume that the data arrive in random order and thus, for hash functions that are not strongly concentrated we might need to update the priority queue much more. But for strongly concentrated hash functions, a straightforward argument shows that the priority queue need only be updated $O(k \log n)$ times no matter the input.

Threshold Estimator. Alternatively, we could use a related, but different sketch of Bar-Yossef et al. [3], which is more efficient. The algorithm identifies the smallest b such that the number $X^{<1/2^b}$ of keys hashing below $1/2^b$ is at most k. For the online processing of the stream, this means that we increment b whenever $X^{<1/2^b} > k$. At the end, we return $2^b X^{<1/2^b}$. The analysis of this estimator is similar to the analysis of the bottom-k estimator. Using strongly concentrated hashing, we get the same advantage of avoiding independent repetitions: In [3] they achieve error probability δ by running $\Theta(\log(1/\delta))$ independent experiments, each storing up to $k = \Theta(1/\varepsilon^2)$ hash values, whereas we achieve the same error probability by running only a single experiment with a strongly concentrated hash function storing $k = O(\log(1/\delta)/\varepsilon^2)$ hash values. The total number of hash values stored is the same, but asymptotically, we save a factor $\Theta(\log(1/\delta))$ in time.

5 SET SIMILARITY WITH STRONGLY CONCENTRATED HASHING

Another setting where the k smallest hash values of a key set is often used is for the problem of *set similarity*. Consider two subsets $A, B \subset U$ of a universe $U = [u] = \{0, 1, \dots, u-1\}$. A common metric for similarity between two such sets is the *Jaccard similarity*, $J(A, B) = |A \cap B| / |A \cup B|$. In this section, we estimate the Jaccard similarity using strongly concentrated hashing, applying similar techniques to the previous section.

We consider Broder's [6] original algorithm for set similarity which we now sketch. As in the previous section, let $h\colon [u] \to [0,1)$ be a hash function assumed to be collision free. We define the bottom-k sample $\mathrm{MIN}_k(S)$ of a set $S \subseteq [u]$ to be the k elements of S with the smallest hash values under h. Assume that $k \le n = |S|$. If h is fully random, then $\mathrm{MIN}_k(S)$ is a uniformly random subset of S of size k, and if $T \subseteq S$, we may estimate the frequency $f = |T|/|S|$ as $|\mathrm{MIN}_k(T) \cap \mathrm{MIN}_k(S)|/k$. In [6], Broder uses this observation to estimate the Jaccard similarity between two sets $A, B \subset [u]$ as follows. Given the bottom-k samples from A and B, the bottom-k sample of their union may be constructed as $\mathrm{MIN}_k(A \cup B) = \mathrm{MIN}_k(\mathrm{MIN}_k(A) \cup \mathrm{MIN}_k(B))$. Then the Jaccard similarity is estimated as $\hat{J}(A, B) = |\mathrm{MIN}_k(A \cup B) \cap \mathrm{MIN}_k(A) \cap \mathrm{MIN}_k(B)|/k$.

For the hash function, h, Broder [6] first considers fully random hashing and this particular case is very well understood. In this case $\mathrm{MIN}_k(S)$ is a fully random sample of k distinct elements from S. However, fully random hashing is not implementable in practice. Broder also sketches some alternatives with realistic hash functions. Continuing this line of work, Thorup [20] showed that using strongly universal hashing, we get the same expected error as with fully random hashing. Writing $f = J(A, B)$ and $\hat{f} = \hat{J}(A, B)$, his precise result is that $\mathbb{E}\left[|f - \hat{f}|\right] = O(1/\sqrt{fk})$. Similarly to the problem of counting distinct elements, in order to obtain an estimator satisfying a similar bound with some high probability at least $1 - \delta$, we have to perform $O(\log(1/\delta))$ independent repetitions and use the median of the estimators as the final estimate. Our next result shows that when implementing the hashing using a scheme with strong concentration bounds, like Tabulation-1Permutation, we obtain a similar error bound by running the algorithm just a single time, but with a sketch which is $O(\log(1/\delta))$ times larger. Again, the total space usage is the same, but we eliminate the need for independent repetitions and save a factor of $O(\log(1/\delta))$ in time.

Our analysis follows the simple union-bound approach from [20]. It is simpler to study the case where we are sampling from a set S and want to estimate the frequency $f = |T|/|S|$ of a subset $T \subseteq S$. Let $h_{(k)}$ be the kth smallest hash value from S as in the above algorithm for estimating distinct elements. For any p let $Y^{\le p}$ be the number of elements from T with hash value at most p. Then $|\mathrm{MIN}_k(T) \cap \mathrm{MIN}_k(S)| = |T \cap \mathrm{MIN}_k(S)| = Y^{\le h_{(k)}}$ which is our estimator for fk.

Theorem 5.1. *For $\varepsilon \le 1$, if h is strongly concentrated with added error probability $\mathcal{E}$, then*

$$\Pr\left[|Y^{\le h_{(k)}} - fk| > \varepsilon fk\right] = 2\exp(-\Omega(fk\varepsilon^2)) + O(\mathcal{E}). \quad (9)$$

Proof. Let $n = |S|$. We already saw in (8) that for any $\varepsilon_S \le 1$, $P_S = \Pr\left[|1/h_{(k)} - n/k| \ge \varepsilon_S n/k\right] \le 2\exp(-\Omega(k\varepsilon_S^2)) + O(\mathcal{E})$. Thus, with $p_- = k/((1+\varepsilon_S)n)$ and $p_+ = k/((1-\varepsilon_S)n)$, we have $h_{(k)} \in [p_-, p_+]$ with probability $1 - P_S$, and in that case, $Y^{\le p_-} \le Y^{\le h_{(k)}} \le Y^{\le p_+}$.

Let $\mu^- = \mathbb{E}\left[Y^{\le p_-}\right] = fk/(1+\varepsilon_S) \ge fk/2$. By strong concentration, for any $\varepsilon_T \le 1$, we get that

$$\begin{aligned} P_T^- = \Pr\left[Y^{\le p_-} \le (1-\varepsilon_T)\mu_-\right] &\le 2\exp(-\Omega(\mu_-\varepsilon_T^2)) + \mathcal{E} \\ &= 2\exp(-\Omega(fk\varepsilon_T^2)) + \mathcal{E}. \end{aligned}$$

Thus

$$\Pr\left[Y^{\le h_{(k)}} \le \frac{1-\varepsilon_T}{1+\varepsilon_S} fk\right] \le P_T^- + P_S.$$

Likewise, with $\mu^+ = \mathbb{E}\left[Y^{\le p_+}\right] = fk/(1-\varepsilon_S)$, for any ε_T, we get

$$\begin{aligned} P_T^+ = \Pr\left[Y^{\le p_+} \ge (1+\varepsilon_T)\mu_+\right] &\le 2\exp(-\Omega(\mu_+\varepsilon_T^2)) + \mathcal{E} \\ &= 2\exp(-\Omega(fk\varepsilon_T^2)) + \mathcal{E}. \end{aligned}$$

and

$$\Pr\left[Y^{\le h_{(k)}} \ge \frac{1+\varepsilon_T}{1-\varepsilon_S} fk\right] \le P_T^+ + P_S.$$

To prove the theorem for $\varepsilon \le 1$, we set $\varepsilon_S = \varepsilon_T = \varepsilon/3$. Then $\frac{1+\varepsilon_T}{1-\varepsilon_S} \le 1 + \varepsilon$ and $\frac{1-\varepsilon_T}{1+\varepsilon_S} \ge 1 - \varepsilon$. Therefore,

$$\begin{aligned} \Pr\left[|Y^{\le h_{(k)}} - fk| \ge \varepsilon fk\right] &\le P_T^- + P_T^+ + 2P_S \\ &\le 8\exp(-\Omega(fk\varepsilon_T^2)) + O(\mathcal{E}) \\ &= 2\exp(-\Omega(fk\varepsilon_T^2)) + O(\mathcal{E}). \end{aligned}$$

This completes the proof of (9). □

As for the problem of counting distinct elements in a stream, in the online setting we may again modify the algorithm above to obtain a more efficient sketch. Assuming that the elements from S arrive in a stream, we again identify the smallest b such that the number of keys from S hashing below $1/2^b$, $X^{\le 1/2^b}$, is at most k. We increment b by one whenever $X^{\le 1/2^b} > k$ and in the end we return $Y^{\le 1/2^b}/X^{\le 1/2^b}$ as an estimator for f. The analysis of this modified algorithm is similar to the analysis provided above.

Remark. The case of set similarity illustrates the crucial importance of using a common hash function h as a source of randomness. In a distributed setting, different entities may generate the samples $\mathrm{MIN}_k(A)$ and $\mathrm{MIN}_k(B)$. As long as they agree on h, they only need to communicate the samples to estimate the Jaccard similarity of A and B. As noted before, for Tabulation-1Permutation h can be shared by exchanging a random seed of $O((\log u)^2)$ bits.

Remark. In the above, we have specifically analysed an estimator for the frequency $f = |T|/|S|$. On a high level, the analysis used strong concentration to (1) bound the sample threshold $p = h_{(k)}$ and (2) bound the number of keys from T hashing below p. A union bound then led to the final result. This argument is very robust: Letting the sample $X = \mathrm{MIN}_k(S)$, we could equally well use the estimate $|S \cap X|/p$ for $|S|$, $|T \cap X|/p$ for $|T|$, and $|S \cap X|/|T \cap X|$ for $|S|/|T|$ an obtain analogue bounds with a similar analysis. In fact, the estimator $|S \cap X|/p = k/h_{(k)}$ for $|S|$ is exactly the estimator for the number of distinct elements that we encountered in Section 4.1.

Table 1: Overview of our experiments. All experiments are designed to compare the tabulation-based hash functions with various other hashing schemes when estimating cardinality. The type of the experiment indicates whether the other hashing schemes use a single sketch, or 5 independent repetitions and the median trick. The tabulation-based hash functions always use a single sketch. When using the median trick, the size of each sketch is reduced by a factor of 5, so the total space usage is always the same.

Dataset	Algorithm	Measurement	Cardinality	Sketch-size	Experiments	Type	Plots
Synthetic	Bottom-k	Error	10^6	24,500	2×10^3	Single Rep.	3a
			5×10^5	24,500	5×10^4	Single Rep.	3b, 3c
			5×10^5	24,500	3×10^4	Median-Trick	5a
		Time	5×10^7	3500	10	Single Rep.	4a
			5×10^7	3500	10	Median-Trick	4b
Synthetic	Threshold sampling	Error	5×10^6	8×10^5	5×10^4	Single Rep.	6a, 6b
			2.5×10^5	1.6×10^5	5×10^4	Single Rep.	6c
			2.5×10^5	1.6×10^5	3×10^4	Median-Trick	5b
		Time	10^9	8×10^5	10	Single Rep.	7a
			5×10^6	8×10^5	10	Median-Trick	7b
Real-world Geometric	Threshold	Error	6.5×10^5	28,000	3×10^4	Both	8a, 8d
Real-world Demographic	Threshold	Error	12,775	7000	5×10^4	Both	8b, 8e
Real-world Atmospheric	Threshold	Error	20,000	7000	5×10^4	Both	8c, 8f

6 EXPERIMENTAL EVALUATION

In this section we experimentally evaluate hash functions with strong concentration bounds in the context of cardinality estimation. In particular, we use the recently announced Tabulation-1Permutation [1] and Mixed-Tabulation [16]. Our experiments address the following questions:

- Is it possible, in practice, to avoid independent repetitions by using hash functions with strong concentration bounds and still get reliable results?
- What are the implications on the running time of cardinality estimation algorithms, when using Tabulation-1Permutation?

In our experiments, we restrict the space usage of our algorithms to some parameter k. When using the fast Multiply-Mod-Prime and Multiply-Shift hash functions, we report both the results of performing a single repetition per experiment with a sample of size k, and the results of performing r independent repetitions per experiment each with a sample of size k/r and outputting the median of the r estimations[4]. For all experiments with repetitions we choose[5] $r = 5$.

We expect the first type of experiment to have about r times better running time, and the second type of experiment to produce more reliable results. On the other hand, when using Tabulation-1Permutation we only perform 1 repetition per experiment, effectively demonstrating that the estimation is still reliable even without the independent repetitions.

[4]In practical applications (e.g., streaming) it is rarely possible to "reuse" the space k which explains why we allocate space k/r for each repetition.

[5]This was sufficient to remove the outliers with these schemes and we would not expect more repetitions to improve the estimators, just slow down the running time.

A brief synopsis of our experiments is presented in Table 1. In what follows, we first discuss our experimental setup and the implementation details. Second, we present our results for synthetic data and the bottom-k algorithm discussed in previous sections. Third, we present similar results for synthetic data and a standard threshold-sampling cardinality estimation algorithm. Finally, we present our results on real-world datasets.

6.1 Experimental Setup

6.1.1 Hardware. We performed our experiments on two computers. The first has an Intel(R) Core(TM) i7-8665U CPU @ 1.90GHz, 16 GB RAM, and is running on 64bit Windows 10. The second has an Intel(R) Core(TM) i5-8350U CPU @ 1.7 GHz, 8 GB RAM, and is running on 64bit Ubuntu 18.04.4 LTS.

6.1.2 Datasets. We use both synthetic and real-world datasets. Notice that theoretically we guarantee that, when using hash functions with strong concentration bounds, the probability of a big error is negligible ("big" is quantified in relation to the sample size). In order to test such a claim, one needs to run many experiments and show that the result is always reliable. Therefore, the quality of the results is tested against small datasets, so that we can repeat the same experiment sufficiently many times.

Our synthetic datasets for testing accuracy are simply consecutive integers. Their cardinalities range between 2.5×10^5 and 5×10^6. The synthetic datasets for testing running times have cardinalities ranging between 5×10^6 and 10^9. There are good reasons for not running the accuracy and time experiments on the same data sets. First, we can afford to run the timing experiments on larger data sets as we do not need that many experiments to test the speed. Second, the data for which the strongly universal hash functions give big outliers is consecutive integers, and it is likely that on such data we would not get an accurate evaluation of their speed. This issue is removed by using random data, but on random data there will be no big outliers.

Concerning our real-world datasets, we have:

- geometric datasets; we are using the Openadresses dataset, which is a public database connecting the geographical coordinates with their postal addresses[6]. The cardinality of this dataset is about 6.5×10^5.
- datasets with demographic characteristics of people[7], extracted from the census bureau database[8]. Its cardinality is 12,775.
- high entropy datasets extracted from atmospheric noise measurements[9], with cardinality 20,000.

6.1.3 Algorithms. We use two different algorithms for cardinality estimation. First, we have the bottom-k algorithm [4] discussed in previous sections; in this case, we expect that the k-th smallest hash value is approximately $\frac{k}{n}$. The second algorithm, threshold-sampling, can be seen as the dual approach; that is, we keep all elements with hash values at most $p = \frac{k}{n}$, expecting the number of such elements to be$k$. In our experiments we use the proper value of p which depends on n, but generally we do not know n in advance.

6.1.4 Hash-Functions. For each algorithm, we test it using some of the most popular and fast hash functions employed in practice. Their output is always 64-bit unsigned integers. In particular, we use Tabulation-1Permutation [1], Mixed-Tabulation [16], Multiply-Mod-Prime [7], Multiply-Shift [10], and MurmurHash3 [2]. We use Multiply-Mod-Prime and Multiply-Shift as they are very fast strongly universal hash functions. MurmurHash3 is also used as it is known to perform well in practice, even though it does not provide theoretical guarantees (e.g. in [5] the authors show how to break MurmurHash3).

We did not experiment with random polynomials of degree 100, or the cryptographic hash function BLAKE3 because they proved to be more than 60 and 155 times slower than any other method, respectively.

6.1.5 Number of experiments. We demonstrate the need for *provably* reliable hash functions by performing a different number of experiments per dataset. When performing more than 10^4 experiments, the reliability of Tabulation-1Permutation becomes clear, while with fewer experiments it is possible to erroneously consider other hash functions reliable as well.

6.1.6 Randomness. The random seed needed for all hash functions was drawn from https://www.random.org/. For tabulation based methods, the seed was filled using a random degree 100 polynomial.

6.1.7 Implementation. We have used C++ 11 for implementation.

6.1.8 Evaluation Metrics. As the measure of estimation accuracy, we report the relative error. The relative error of an estimate $\hat{n}$ of a quantity n is defined as $\frac{n-\hat{n}}{n}$. We also report the 6th central moment of the relative errors of different experiments, as a measure of dispersion that is more sensitive to big errors, compared to the variance (2nd central moment). For measuring running time, we report the average time per experiment.

6.2 The Bottom-k Algorithm on Synthetic Data

In this subsection, we present our results on applying the bottom-k algorithm to synthetic data. These experiments are of two different kinds. The first uses one repetition per experiment for all hash functions, and the second uses the median trick with Multiply-Shift and Multiply-Mod-Prime.

6.2.1 One Repetition Per Experiment. We ran experiments on implementations of the Bottom-k algorithm, only using a single sketch for each hash function. The experiments are listed as rows 1, 2, and 4 of Table 1. All experiments had $k = 24{,}500$. The datasets used are structured datasets containing consecutive integers.

Testing for accuracy, for the dataset with cardinality 10^6 (row 1 of Table 1) all experiments were within a 3% relative error (Figure 3a). In fact, Multiply-Mod-Prime and Multiply-Shift gave the best estimates, with the 6th central moment of the relative errors of their respective experiments being considerably smaller compared to the other hash functions. It is to be expected that not too many outliers were observed since 2×10^3 experiments per hash function is not sufficient to detect the outliers. On the other hand, for the dataset with cardinality 5×10^5 (row 2 of Table 1) both Multiply-Mod-Prime and Multiply-Shift have some experiments with huge errors (about 45% and 55%, respectively). See Figure 3b.

Testing running times, we tested on a dataset of cardinality 5×10^7 (row 4 of Table 1) with $k = 3500$ (Figure 4a). Tabulation-1Permutation (11.2ms on average) and Multiply-Shift (11.5ms) had very similar running times, with Mixed-Tabulation (19.4ms) and Multiply-Mod-Prime (20.1ms) being slower, and MurmurHash3(32.5ms) being significantly slower.

6.2.2 Applying the Median Trick. We performed experiments where independent repetitions and the median trick were used for the instantiations with Multiply-Shift, Multiply-Mod-Prime, and MurmurHash3, while the implementation with the Tabulation-based methods still used just a single sketch.

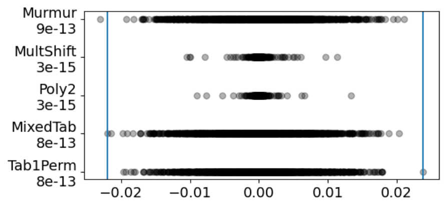
For accuracy, we performed 3×10^4 experiments on a dataset of cardinality 5×10^5 (row 3 of Table 1). The results are shown in Figure 5a. We see that the estimates, when using Tabulation methods are generally more concentrated around the actual cardinality.

For speed, we ran the algorithms on datasets of cardinality 5×10^7 with $k = 3500$ (row 5 of Table 1). As seen in Figure 4b, we obtain a significant improvement of Tabulation-1Permutation (11ms on average) and Mixed-Tabulation (22ms) over Multiply-Mod-Prime (109ms), Multiply-Shift (48ms) and Murmurhash3 (182ms), whose proportionate running times remain largely unchanged.

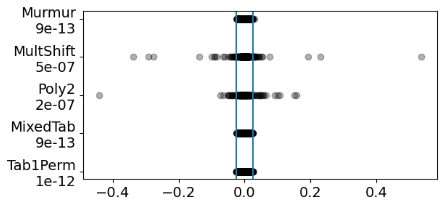
6.3 Threshold-Sampling on Synthetic data

In this subsection, we present our experiments on the threshold-sampling algorithm when using synthetic data. We first consider an implementation with one repetition per experiment for all hash functions. Secondly, we consider using the median trick with the non-Tabulation-based hash functions.

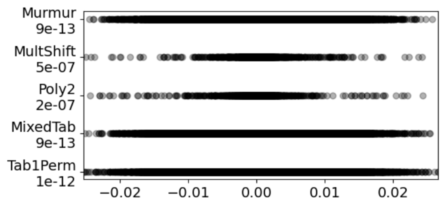
[6]https://openaddresses.io, Last accessed 16/9/2022
[7]https://archive.ics.uci.edu/ml/datasets/Census+Income, Last accessed 16/9/2022
[8]https://www.census.gov/data.html, Last accessed 16/9/2022
[9]https://www.random.org/, Last accessed 16/9/2022



(a) Cardinality: 10^6
Experiments per hash-function: 2×10^3

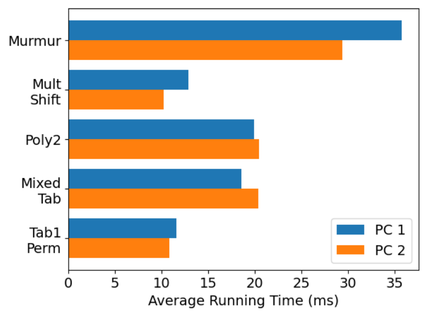


(b) Cardinality: 5×10^5
Experiments per hash-function: 5×10^4

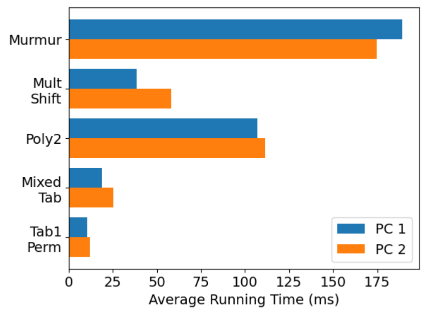


(c) Cardinality: 5×10^5
Experiments per hash-function: 5×10^4

Figure 3: Relative error of the single-repetition experiments using the bottom-k algorithm (k = 24,500). Each dot represents an experiment, and the x-coordinate is the relative error. The more opaque the dots are, the more experiments had the corresponding relative error. The vertical blue lines limit the results of the Tabulation methods. On the y-axis we also report the 6th central moment of the relative errors for each hash function.

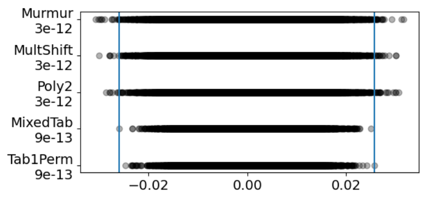


(a) Single-repetition experiments with structured synthetic datasets (consecutive integers). Cardinality = 5×10^7, k = 3500.

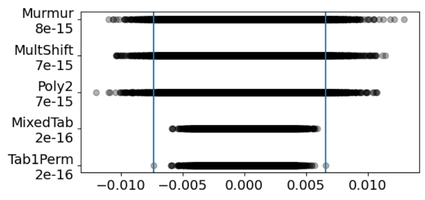


(b) Experiments on random synthetic datasets. With Multiply-Mod-Prime and Multiply-Shift, $(r, k) = (5, 700)$**. With Tabulation-1Permutation,** $(r, k) = (1, 3500)$**. Cardinality** = 5×10^7

Figure 4: Timing of bottom-k experiments (synthetic data).



(a) Bottom-k: Cardinality = 5×10^5
For Multiply-Mod-Prime and Multiply-Shift we have (r, k) = $(5, 4900)$**. For Tabulation-1Permutation** $(r, k) = (1, 24{,}500)$**.**

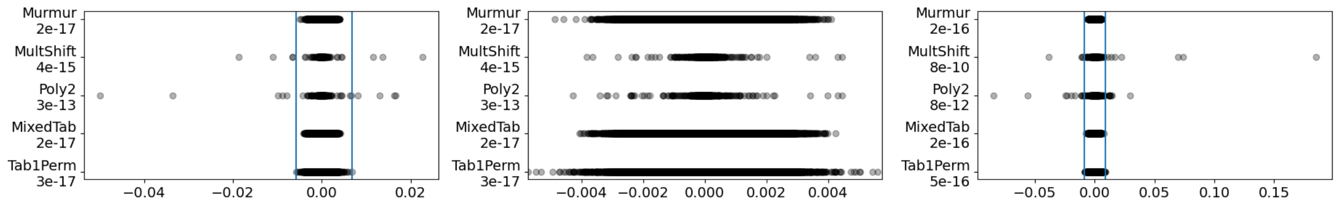


(b) Threshold-sampling: Cardinality = 2.5×10^5
For Multiply-Mod-Prime and Multiply-Shift we have (r, p) = $(5, 0.128)$**. For Tabulation-1Permutation** $(r, p) = (1, 0.64)$**.**

Figure 5: Relative error of 3×10^5 experiments for each hash-function, using the bottom-k and the threshold-sampling algorithm on random synthetic data. Multiply-Mod-Prime and Multiply-Shift used 5 repetitions per experiment, whereas the rest ran just once per experiment with a 5 times larger sketch. The plots should be interpreted as those in Figure 3.

6.3.1 One Repetition Per Experiment. We performed experiments on implementations of the threshold-sampling algorithm using only a single sketch for each hash functions. Details of the experiments are listed as rows 6, 7, and 9 of Table 1.

For accuracy, we performed experiments on a datasets with cardinality 5×10^6 (row 6 of Table 1). The results are plotted in Figure 6a. Only Multiply-Mod-Prime and Multiply-Shift gave results with large outlier errors (larger than 5% and 2%, respectively, compared to errors significantly less than 1% for the rest of the hash functions). If we zoom in (Figure 6b, effectively ignoring the outliers) these two hash functions are far more accurate than all others. This is actually not at all surprising, as we remark in Section 6.5. Rather it is a direct consequence of the estimators having the same variance. We also performed experiments for datasets with cardinality 2.5×10^5 (row



(a) Cardinality: 5×10^6
Experiments per hash-function: 5×10^4

(b) Cardinality: 5×10^6
Experiments per hash-function: 5×10^4

(c) Cardinality: 2.5×10^5
Experiments per hash-function: 5×10^4

Figure 6: Relative error of the single-repetition experiments using the threshold-sampling algorithm, using structured synthetic datasets (consecutive integers). The plots should be interpreted as those in Figure 3.

7 of Table 1). The results are plotted in (Figure 6c). Here, we see a similar picture, but the errors are much higher (about 10% relative error for Multiply-Mod-Prime, and 20% for Multiply-Shift).

Concerning the running time, we run datasets of cardinality 10^9 (row 9 of Table 1). The results can be seen in Figure 7a: from fastest to slowest we have Multiply-Shift (132ms on average), Tabulation-1Permutation (187ms), Multiply-Mod-Prime (278ms), Mixed-Tabulation (314ms), and MurmurHash3 (531ms).

6.3.2 Applying the Median Trick. We performed experiments using independent repetitions and the median trick with the non-Tabulation-based hash functions, while the Tabulation-based ones still used a single sketch.

For accuracy, we performed 3×10^4 experiments with a dataset of cardinality 2.5×10^5 (row 8 of Table 1). We used the threshold $p = 0.128$ for Multiply-Mod-Prime and Multiply-Shift and $p = 0.64$ for Tabulation-1Permutation. The results are shown in Figure 5b. Note that the threshold p when performing 5 repetitions is 5 times smaller. This ensures that the elements of each of the 5 independent sketches only use a fifth of the total allowed space k. We observe that the estimates are better concentrated with tabulation methods.

Testing for speed, we ran experiments on datasets of cardinality 5×10^6 (row 10 of Table 1). Here, Tabulation-1Permutation (1.9ms on average) and Mixed-Tabulation (2.5ms) are significantly faster than Multiply-Shift (3.5ms), Multiply-Mod-Prime (6.8ms), and MurmurHash3 (9.6ms). The results can be seen in Figure 7b.

6.4 Experiments on Real-World Data

In this subsection, we present our experiments with real-world data. The experiments are listed as rows 11 through 16 of Table 1. We performed experiments using both the bottom-k sampling and threshold sampling. As we have already seen several estimates of the running times (and they are predictable from experiment to experiment since the different hash functions use the same computations regardless of the keys), we here focus on accuracy.

6.4.1 One Repetition Per Experiment. In the setting with a single sketch for all the hash functions, the results were as follows.

For accuracy, when running on both the geometric dataset and the demographic dataset, Tabulation-1Permutation, Mixed-Tabulation and MurmurHash3 were giving similar results, without any large outliers. On the other hand, Multiply-Mod-Prime and Multiply-Shift did have outliers similar to what we saw for the synthetic data sets. See Figure 8a and Figure 8b, respectively. In particular, for the demographics dataset some of these deviations were as big as 10% for both Multiply-Mod-Prime and Multiply-Shift.

On the other hand, when we experiment on the atmospheric dataset (Figure 8c), which has high entropy, the results are different. For this data all the estimators behave similarly and there are no large outliers. E.g., we see that (without repetitions) the 6th central moments are almost equal. As noted, this is not unexpected assuming the atmospheric data is indeed almost fully random.

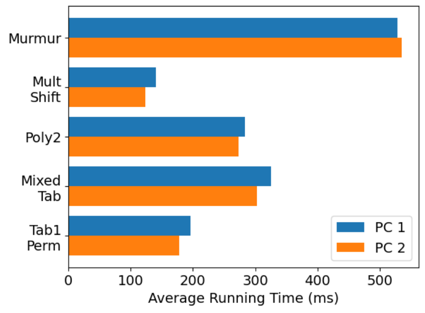
6.4.2 Applying the Median Trick. When using the median trick for the estimator of the non-Tabulation based methods, (while still using a single repetition for the Tabulation based methods), we see that the big outliers with Multiply-Mod-Prime and Multiply-Shift disappeared. However, neither of these hash functions nor MurmurHash3 give as reliable estimates as the Tabulation methods with a single repetition. See Figure 8d, 8e and 8f for the results.

6.5 Remark on Concentration

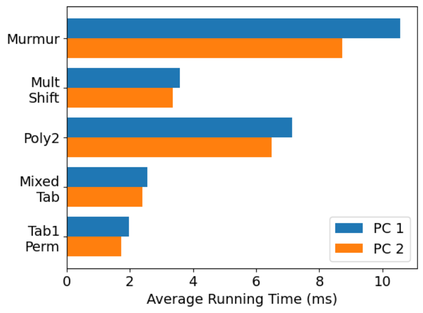
An interesting observation related to the errors when running only a single sketch for all hash functions is the following: If we only focus our attention on the successful experiments, i.e., the outcomes with relative error less than ε, then on some data sets, Multiply-Mod-Prime and Multiply-Shift appear more accurate than Tabulation-1Permutation (see for instance Figure 3a and Figure 3c). Running only a few experiments, this could lead to the false impression that these hash functions are *always* better than hash functions like Tabulation-1Permutations. However, the variance of the estimators are the same for all the seeded hash functions. This means that if we obtain these 'to good to be true' estimates most of the time, we must inevitably have some cases where the estimates are far off. This is precisely the behaviour that we see with Multiply-Mod-Prime and Multiply-Shift. On the other hand, as seen in Figure 3a and Figure 3c, the Tabulation-based hash functions provide estimates that reliably lie within an acceptable error margin, not extremely close to the precise cardinality, yet with no wild outliers.

7 CONCLUSION

In this paper we showed that the use of hash functions with strong concentration bounds, like Tabulation-1Permutation and Mixed-Tabulation, can speed up sampling based algorithms by avoiding time consuming independent repetitions, and still provide accurate statistical estimates with high probability. Our results are backed

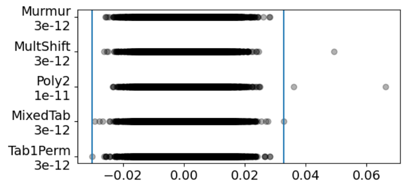


(a) Single repetition experiments on structured synthetic data (consecutive integers). Cardinality $= 10^9$, $p = 8 \times 10^{-4}$.

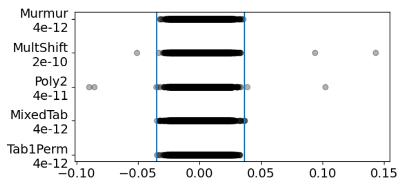


(b) Experiments on random synthetic datasets. With Multiply-Mod-Prime and Multiply-Shift, $(r, p) = (5, 0.032)$. **With Tabulation-1Permutation** $(r, p) = (1, 0.16)$. **Cardinality**$=5 \times 10^6$.

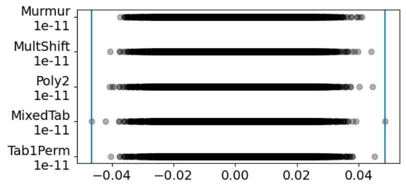
Figure 7: Timing of threshold-sampling experiments (synthetic data).



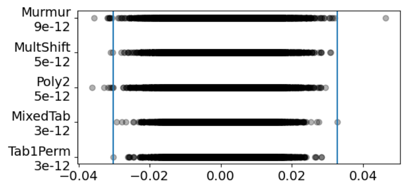
(a) Accuracy on geometric dataset (single experiment).
Cardinality: 6.5×10^5
Experiments: 3×10^4



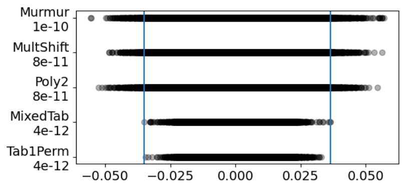
(b) Accuracy on demographics dataset (single sketch).
Cardinality: 12,775
Experiments: 5×10^4



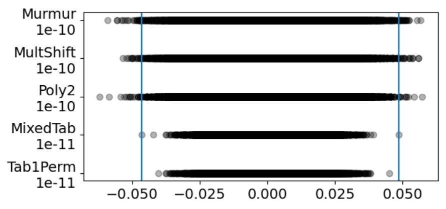
(c) Accuracy on atmospheric dataset (single experiment).
Cardinality: 20,000
Experiments: 5×10^4



(d) Accuracy on geometric dataset (median trick).
Cardinality: 6.5×10^5
Experiments: 3×10^4



(e) Accuracy on demographics dataset (median trick).
Cardinality: 12,775
Experiments: 5×10^4



(f) Accuracy on atmospheric dataset (median trick).
Cardinality: 20,000
Experiments: 5×10^4

Figure 8: Accuracy experiments on the real-world datasets. We used threshold sampling for the geometric and atmospheric data sets and bottom-k sampling for the demographic data set. The plots are interpreted as Figure 3.

up by experiments showing that it is faster and more reliable to sample with these hashing schemes compared to using classic 2-independent hash functions like Multiply-Shift and Multiply-Mod-Prime and independent repetitions. It is also significantly faster than using commonly used hash functions like MurmurHash3, which provided reliable estimates in our experiments (even without independent repetitions) but have no similar theoretical guarantees. We further saw that without independent repetitions, the simple 2-independent hash functions failed to provide reliable sampling on several real world data sets. These bad real-word instances motivate an interesting direction for future work, namely to explore whether there are other applications of hashing in industrial databases that are not as reliable as they should be because they employ too weak hash functions. As no amount of experimenting can prove that a hashing scheme works well for all possible input, they also suggest the importance of a continued *theoretical* study to find fast hash functions that are powerful enough for important applications.

ACKNOWLEDGMENTS

All authors partly supported by Thorup's Investigator Grant 16582, Basic Algorithms Research Copenhagen (BARC), from the VILLUM Foundation. Anders Aamand is funded by the DFF-International Postdoc Grant 0164-00022B from the Independent Research Fund Denmark.

REFERENCES

[1] Anders Aamand, Jakob Bæk Tejs Knudsen, Mathias Bæk Tejs Knudsen, Peter Michael Reichstein Rasmussen, and Mikkel Thorup. 2020. Fast hashing with strong concentration bounds. In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020*. ACM, 1265–1278.

[2] Austin Appleby. 2016. *Murmurhash3*. Available at https://github.com/aappleby/smhasher/wiki/MurmurHash3, Last accessed 16/9/2022.

[3] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. 2002. Counting Distinct Elements in a Data Stream. In *International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM)*. 1–10.

[4] Kevin S. Beyer, Peter J. Haas, Berthold Reinwald, Yannis Sismanis, and Rainer Gemulla. 2007. On synopses for distinct-value estimation under multiset operations. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*. 199–210. https://doi.org/10.1145/1247480.1247504

[5] Martin Boßlet. 2012. *Breaking murmur: Hash-flooding dos reloaded*. Available at https://emboss.github.io/blog/2012/12/14/breaking-murmur-hash-flooding-dos-reloaded/, Last accessed 16/9/2022.

[6] Andrei Z. Broder. 1997. On the resemblance and containment of documents. In Proc. Compression and Complexity of Sequences (SEQUENCES). 21–29.

[7] Larry Carter and Mark N. Wegman. 1979. Universal classes of hash functions. *J. Comput. System Sci.* 18, 2 (1979), 143–154. Announced at STOC'77.

[8] Edith Cohen and Haim Kaplan. 2008. Tighter estimation using bottom k sketches. *Proc. VLDB Endow.* 1, 1 (2008), 213–224. https://doi.org/10.14778/1453856.1453884

[9] Søren Dahlgaard, Mathias Bæk Tejs Knudsen, Eva Rotenberg, and Mikkel Thorup. 2015. Hashing for Statistics over K-Partitions. In *Proceedings of the 56th IEEE Symposium on Foundations of Computer Science (FOCS)*. 1292–1310. https://doi.org/10.1109/FOCS.2015.83

[10] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. 1997. A Reliable Randomized Algorithm for the Closest-Pair Problem. *Journal of Algorithms* 25, 1 (1997), 19–51.

[11] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. 1992. Dynamic Hashing in Real Time. In *Informatik, Festschrift zum 60. Geburtstag von Günter Hotz*, Johannes Buchmann, Harald Ganzinger, and Wolfgang J. Paul (Eds.). Teubner-Texte zur Informatik, Vol. 1. Teubner / Springer, 95–119. https://doi.org/10.1007/978-3-322-95233-2_7

[12] Nick G. Duffield and Matthias Grossglauser. 2000. Trajectory sampling for direct traffic observation. In *Proceedings of the ACM SIGCOMM 2000 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 28 - September 1, 2000, Stockholm, Sweden*, Craig Partridge (Ed.). ACM, 271–282. https://doi.org/10.1145/347059.347555

[13] Nick G. Duffield, Carsten Lund, and Mikkel Thorup. 2007. Priority sampling for estimation of arbitrary subset sums. *J. ACM* 54, 6 (2007), 32. https://doi.org/10.1145/1314690.1314696

[14] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frederic Meunier. 2007. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *In AOFA '07: Proceedings of the 2007 International Conference on Analysis of Algorithms*. 127–146.

[15] Hazar Harmouch and Felix Naumann. 2017. Cardinality Estimation: An Experimental Survey. *Proceedings of the VLDB Endowment* 11 (12 2017), 499–512. https://doi.org/10.1145/3164135.3164145

[16] Jakob Bæk Tejs Houen and Mikkel Thorup. 2022. Understanding the Moments of Tabulation Hashing via Chaoses. In *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France (LIPIcs)*, Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff (Eds.), Vol. 229. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 74:1–74:19. https://doi.org/10.4230/LIPIcs.ICALP.2022.74

[17] Samuel Neves Jack O'Connor, Jean-Philippe Aumasson and Zooko Wilcox-O'Hearn. 2020. *BLAKE3*. Available at https://github.com/BLAKE3-team/BLAKE3, Last accessed 16/9/2022.

[18] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. 2003. Sketch-based change detection: methods, evaluation, and applications. In *Proceedings of the 3rd ACM SIGCOMM Internet Measurement Conference, IMC 2003, Miami Beach, FL, USA, October 27-29, 2003*. 234–247. https://doi.org/10.1145/948205.948236

[19] Rajeev Motwani and Prabhakar Raghavan. 1995. *Randomized Algorithms*. Cambridge University Press.

[20] Mikkel Thorup. 2013. Bottom-k and Priority Sampling, Set Similarity and Subset Sums with Minimal Independence *(STOC '13)*. Association for Computing Machinery, New York, NY, USA, 371–380. https://doi.org/10.1145/2488608.2488655

[21] Mikkel Thorup and Yin Zhang. 2012. Tabulation-Based 5-Independent Hashing with Applications to Linear Probing and Second Moment Estimation. *SIAM J. Comput.* 41, 2 (2012), 293–331. Announced at SODA'04 and ALENEX'10.

[22] Mark N. Wegman and Larry Carter. 1981. New Classes and Applications of Hash Functions. *J. Comput. System Sci.* 22, 3 (1981), 265–279. Announced at FOCS'79.