Why Go Logarithmic if We Can Go Linear? Towards Effective Distinct Counting of Search Traffic *

Ahmed Metwally Divyakant Agrawal
Ask.com
555 12th Street, Suite 500
Oakland, CA 94607
{Ahmed.Metwally, Divy.Agrawal}@ask.com

Amr El Abbadi
Department of Computer Science
University of California at Santa Barbara
Santa Barbara CA 93106
amr@cs.ucsb.edu

ABSTRACT

Estimating the number of distinct elements in a large multiset has several applications, and hence has attracted active research in the past two decades. Several sampling and sketching algorithms have been proposed to accurately solve this problem. The goal of the literature has always been to estimate the number of distinct elements while using minimal resources. However, in some modern applications, the accuracy of the estimate is of primal importance, and businesses are willing to trade more resources for better accuracy. Throughout our experience with building a distinct count system at a major search engine, Ask.com, we reviewed the literature of approximating distinct counts, and compared most algorithms in the literature. We deduced that Linear Counting, one of the least used algorithms, has unique and impressive advantages when the accuracy of the distinct count is critical to the business. For other estimators to attain comparable accuracy, they need more space than Linear Counting. We have supported our analytical results through comprehensive experiments. The experimental results highly favor Linear Counting when the number of distinct elements is large and the error tolerance is low.

1. INTRODUCTION

Modern Internet-based applications have imposed numerous constraints on data management research. For many such applications, when it comes to algorithm design, the rate of Internet traffic, and the business nature of the applications impose extremely small margins for inefficiency and inaccuracy. These challenges made the data streams model widely adopted in both the research and the industrial communities. This computational model assumes one-scan on the data and in-memory algorithms, and requires tight error guarantees on the results.

In the context of search engines, these challenges are especially manifested. The search engines' traffic rate is expected to be among the highest among all Internet-based businesses, since they are the gateway of the entire Internet. Thus, there is no wonder in them being among the most popular sites, which entails assiduous search for optimality to handle their high traffic. Such huge and continuous traffic requires more responsiveness than what is usually offered by exact off-line analysis algorithms. Several one-pass main-memory algorithms have been adopted to analyze traffic online for time-critical functionalities, such as detecting click fraud [35].

The accuracy constraints are not any less stringent. Very crucial to the business of search engines is to understand the high proportionality between the traffic utilization and the revenue. For instance, estimating the market share of the search engine by estimating its number of distinct surfers (visitors) does not only impact strategic planning, but also translates directly into the business value of the search engine as discussed in Section 2.1. This necessitates extremely accurate traffic analysis.

In the search context, this work deals with a challenging data management problem, which is counting distinct elements in data streams. We report our experience with building a distinct count system at a major search engine, Ask.com, which abides by strict accuracy and efficiency constraints. This problem has received considerable attention (e.g., [13, 14, 16, 21, 26]) that is reviewed in Section 3. Given the strict accuracy and efficiency constraints of search engines, we discovered much merit in Linear Counting [42], an algorithm that is rarely used. We show analytically that for some applications, Linear Counting is extremely accurate given its space consumption when compared to other algorithms. We have conducted comprehensive experiments on the number of unique surfers, and the results came to be in favor of Linear Counting when almost exact counting is required.

The rest of the paper is organized as follows. In Section 2, we motivate the need for highly accurate distinct counting in the context of search engines. Section 3 is a quick review and a high level classification of the algorithms that were proposed to solve the distinct count problem. The algorithms are discussed in more details in Sections 4 through 8. We designate Section 9 for concluding our analysis of the algorithms' performance. In Section 10, we report the results of our comprehensive set of experiments using real data. Finally, we conclude and report our future directions in Section 11.

*Supported by NSF under grants IIS 02-23022, and CNF 04-23336.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT'08, March 25–30, 2008, Nantes, France.
Copyright 2008 ACM 978-1-59593-926-5/08/0003 ...$5.00.

2. DISTINCT COUNTING AND SEARCH ENGINES

In this section, we motivate the usefulness of accurate distinct counting for search engines. In addition we touch on some of the challenges imposed by the nature of the data.

2.1 Motivating Distinct Counting for Search Engines

Conventional databases are well tailored for accessing indexed data through application-centered automated simple queries and updates. This OLTP operational mode is employed by store-based businesses. Estimating the number of distinct customers in a time period is straightforward through the `distinct` primitive of SQL. While this model holds for brick and mortar businesses, it does not scale up to handle the nine-digit number of daily searches, and the seven-digit number of distinct surfers visiting a search engine.

Once we realize the scale of the problem, it is understandable that conventional database and disk-based distinct counting techniques fail in such a scenario. Since the data entries collected for estimation could be in billions, which is too huge to fit in memory, the problem has to be solved approximately. This brings another crucial factor into play, the accuracy of the counting technique.

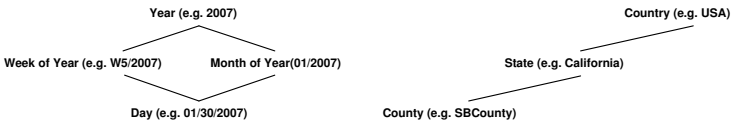
The number of distinct surfers visiting a search engine translates into the value of the business. For advertisers, this is a bound on the distinct Internet customers that will see their advertisements. For the search engine, estimating the number of distinct surfers accurately cannot be overemphasized when it comes to negotiating with advertisers about the prices of showing advertisements and of clicks. If the number of distinct surfers can be estimated within an error of $\pm 10\%$, then the advertisers can always bargain on the lower estimate. This loss in bargaining power is minimized as the estimation of the distinct surfer count becomes more accurate.

2.2 Multidimensional Distinct Counting

The problem of distinct counting is aggravated by the curse of dimensionality. The traffic of search engines can be segmented according to several dimensions. Advertisers are usually interested in a subset of values on each dimension. It is crucial to provide advertisers with estimates on the distinct surfers satisfying their interests.

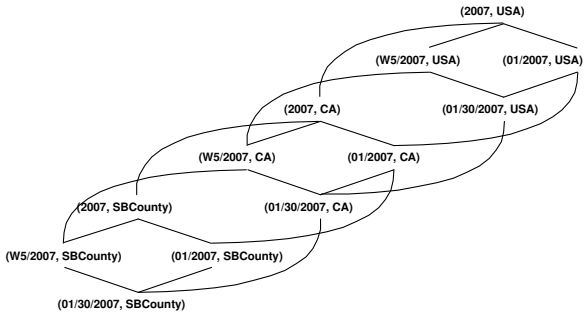
These dimensions are hierarchical in nature, and hence, their ranges are overlapping. For instance, a specific surfer querying the search engine on the 30th of January, 2007 should be counted for the day 01/30/2007, for the month 01/2007, and for the 5th week of 2007. Similarly, a search query that is coming from a UCSB IP address should be counted for California and for the County of Santa Barbara. We give an example of the temporal and geographical hierarchical traffic attributes in Figures 1(a) and 1(b), respectively.

From the literature on multidimensionality and data cubes [11, 24], the combination of hierarchical dimensions imposes a lattice structure on the data space. Each lattice node corresponds to a sub-space with each dimension aggregated at a specific level. For instance, the node $(01/30/2007, SBCounty)$ includes the traffic that arrived from UCSB on the 30th of January, 2007. Meanwhile, a traffic entry can belong to several nodes, since such traffic also belongs to the node $(2007, California)$. A *descendent* lattice node can be aggregated into another *ancestor* node if one dimension is aggregated along its hierarchy. This ancestor-descendent relationships correspond to the edges of the lattice graph. We give an example in Figure 1(c). $(01/30/2007, SBCounty)$ has the ancestor nodes $(01/2007, SBCounty)$, $(01/30/2007, California)$, and $(W5/2007, SBCounty)$. Then, the distinct surfers in the node $(01/30/2007, SBCounty)$ are included in the node $(01/2007, SBCounty)$.



(a) The temporal hierarchy of the traffic.

(b) The geographical hierarchy of the traffic.



(c) The lattice induced by the temporal and geographical hierarchies on the traffic $(01/30/2007, SBCounty)$.

Figure 1: The temporal and geographical hierarchies and an example on their Lattice.

While small advertisers can be interested in showing their advertisements at a specific geographical locality, or a special day of the year, like the node $(01/30/2007, SBCounty)$, bigger advertisers might be interested in larger locality, or a special season, like the node $(01/2007, California)$. Therefore, the distinct count estimation has to work at all the lattice nodes. To address this problem there are two approaches.

- **Simultaneously estimating distinct counts on the fly at all nodes of the lattice.** A distinct counts estimation system is maintained for every combination of dimensions' values at all combinations of dimensions' aggregations. When a new traffic entry arrives, all the distinct counts systems that this entry belongs to are updated.

- **Estimating distinct counts on the fly at the lowest lattice granularity only.** A distinct counts estimation system is maintained for each lattice nodes with no descendants. When a tuple with this combination of dimensions' values is encountered its corresponding counting system is updated. When distinct

counts at granularities higher in the hierarchies are desired, the low-granularity estimates are combined to obtain them.

The first approach calls for materializing a complete data cube for distinct counts on all the dimensions. While this problem has been studied in the context of stored data (e.g. [5, 15, 28, 32, 33, 36, 40]), it is very costly and slow, and hence does not fit in the streaming environment of search engines. In practice, the number of lattice nodes for a moderate number of dimensions and dimension cardinalities is huge. In addition, not all the lattice nodes are used in reality [34]. On the other hand, the second approach is very rudimentary. Although we finds it wise to calculate the distinct count at any lattice node from the distinct counts at its descendants on demand, calculating nodes from grand-grand-descendants, e.g., years from hours, could be very slow.

We follow a pragmatic solution on top of the second approach. We estimate the distinct counts at the lowest lattice granularity. To answer queries on the distinct counts of nodes higher in the hierarchy, aggregations are made using a union-like predicate on the descendants. All the distinct counts calculated along the path to the queried node are cached for later aggregations. A similar approach has been presented before by Han *et al.* [27].

Therefore, in addition to the high accuracy and efficiency constraints, the lattice structure imposes another challenge to distinct counting. The results of the distinct counts estimation system must be amenable to *merrge*. By analogy to constructing data cubes, a traffic entry should be counted only once for any of the nodes containing it. For instance, to estimate the number of distinct surfers in the node $(01/30/2007, California)$, the estimates of all the counties of California on (01/30/2007) should be merged even though some traveling surfers might have queried the search engine from several counties. Such traveling surfers should be included in the distinct count for $(01/30/2007, California)$ once.

The data management literature is rich with several algorithms that can be employed in the search context to estimate the number of distinct elements in a multiset and whose estimates are mergeable to estimate unions. We are mainly concerned with the append only model of the data streams, and hence we do not consider deletions of stream entries. We review the literature in the Section 3.

3. DISTINCT COUNTING PRELIMINARIES

Having motivated the need for distinct counting in the context of traffic analysis, we give a very high level classification of the published algorithms. We follow by the mathematical framework that we use in the sequel to contrast the algorithms in order to select the most appropriate algorithm for our context of traffic analysis.

3.1 High level Algorithm Classification

As shown in Figure 2, we classify the work on estimating the distinct count into two broad categories[1]. The first category [8, 25, 26, 29, 30, 31, 37, 38, 39] uses sampling for estimation. Hence, the main advantage of this category is not scanning the entire traffic dataset to estimate the distinct count, in addition to generating a sample of the traffic as a byproduct. However, several negative results, including [8, 26], were proved showing that almost all the dataset needs to be sampled to bound the estimation error of the distinct count to within a small constant.

[1]The work done in [10, 18, 19] estimates distinct counts on streams with element insertions and deletions. We do not consider this work any further since the extra complexity of accommodating deletions does not benefit our append-only traffic analysis application.

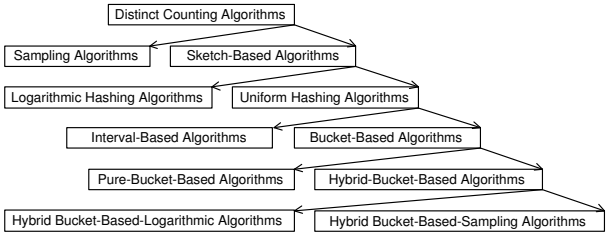


Figure 2: A classification of the distinct counting algorithm.

The second category [1, 3, 9, 10, 13, 14, 16, 18, 19, 21, 23, 42] is a sketch-based approach that scans the entire traffic and hashes each element into a sketch. The sketch is used at query time to estimate the distinct count. Usually, independent sketches and hash functions are employed in parallel to reduce the estimation error.

The sketch-based algorithms can be classified further into logarithmic hashing [1, 13, 16], and uniform hashing algorithms. The uniform hashing algorithms comprise interval-based [3, 9, 23] and bucket-based algorithms. The bucket-based algorithms are either pure-bucket-based [3, 42] or hybrid-bucket-based algorithms. Hybrid-bucket algorithms can be further classified into hybrid-bucket-sampling algorithms [21, 4], and hybrid-bucket-logarithmic algorithms [14].

We discuss the sketch-based category in detail, in Sections 4 through 8, due to the fast per element processing, and the tight error guarantees of these algorithms. Both factors are crucial for our traffic analysis application. However, we first introduce the mathematical framework which we subsequently use to analyze and contrast the algorithms.

3.2 The Mathematical Framework

Throughout the sequel and for any algorithm we discuss, we denote the estimator of n, the distinct count, as $\hat{n}$. We are specifically interested in the random variable $\frac{\hat{n}}{n}-1$, which represent the accuracy of the estimator. The bias of $\hat{n}$ is given by $bias(\hat{n}) = E\left(\frac{\hat{n}}{n} - 1\right)$, and the standard error of $\hat{n}$ is given by the $stderr(\hat{n}) = \sigma\left(\frac{\hat{n}}{n} - 1\right) = \sqrt{E\left(\left(\frac{\hat{n}}{n}\right)^2\right) - \left(E\left(\frac{\hat{n}}{n}\right)\right)^2}$.

We disregard the bias of the estimators since all the algorithms discussed are asymptotically unbiased. Given $stderr(\hat{n})$ in terms of the space usage of the algorithm, we can compare algorithms by comparing their relative space that would yield the same standard error, or by comparing their standard error given a specific space usage. We are assuming that for any algorithm the estimation error decreases monotonically with the space usage. That is, an algorithm never improves its accuracy by using less space. In addition, we compare the algorithms based on the time to process one traffic element.

For some of the algorithms that we discuss in the sequel, the authors did not provide a closed form on $stderr(\hat{n})$. Instead, the guarantees of some algorithms are given in the

form $\mathbf{Pr}\left[f_1(n) \geq \frac{\hat{n}}{n} \geq f_2(n)\right] \leq const$. We try to transform this error guarantee to the Chebyshev's inequality form $\mathbf{Pr}\left[\left|\frac{\hat{n}}{n} - 1\right| \geq f_3(n)\right] \leq const'$. We use Chebyshev's inequality and work backward to estimate a lower bound on the standard error for such algorithms. We can safely derive that $stderr(\hat{n}) \geq f_3(n) \times \sqrt{const'}$, if we assume the analysis of the authors provides a tight bound on the failure probability. The reason is that Chebyshev's inequality holds for all random variables and is tight only under some conditions. The one sided Chebyshev's inequality can also be used in a similar manner.

After introducing the mathematical framework of our study, we start by discussing the sketch-based distinct counting algorithms. We overview the logarithmic hashing algorithms, the interval-based algorithms, the pure-bucket algorithms, the hybrid-bucket-sampling algorithms, and the hybrid-bucket-logarithmic algorithms in Sections 4 through 8, respectively. We comment on the analytics of the algorithms in Section 9.

4. LOGARITHMIC HASHING ALGORITHMS

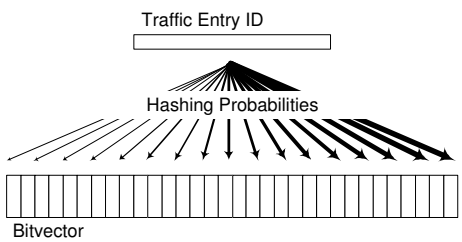


Figure 3: Logarithmic hashing algorithms employ a bitvector and a hash function from each traffic entry id to a single bit. The probability of hashing the id to a specific bit is proportional to the thickness of the arrow to this bit.

The insight of this class of algorithms is to keep track of the most uncommon element (traffic entry id) seen so far. If this most uncommon element is extremely uncommon, then this indicates that the algorithm has scanned so many distinct elements.

To achieve this, the algorithms define the commonness of an element using hashing. As shown in Figure 3, this class of algorithms assumes a bitvector, B, even though not all the algorithms store the bitvector. The bitvector, B, is of length $\log_2(|\mathcal{N}|)$ bits, where $\mathcal{N}$ is the domain of elements. Each traffic entry id in the domain is hashed using a hash function into this bitvector. The probability of hashing a traffic id to a specific bit decreases exponentially as the bit significance increases.

Initially, all bits are set to 0. As the traffic stream is observed, each traffic id is hashed, and its hashed-to bit is set to 1. After hashing all the elements in the stream, S, into the bitvector, the bitvector is queried for the bits that correspond to uncommon traffic IDs. Hence, the total number of distinct traffic IDs can be estimated from how common these bits are.

4.1 The PC Algorithm

The Probabilistic Counting (PC) algorithm [16] uses a hash function, h, to map each element, e, to the space $[1 \ldots \log_2(|\mathcal{N}|)]$ with a geometric distribution. That is, for each bit b in B, $\mathbf{Pr}\left[h(e) = b\right] = 2^{-b}$, where $b \in [1 \ldots \log_2(|\mathcal{N}|)]$.

The algorithm scans the traffic entries. For each element (traffic entry id), e, the algorithm sets the bit in B corresponding to $h(e)$ to 1. For notational purposes, we use $trail_1(.)$ to denote the number of trailing 1 bits. After hashing all the traffic stream, into B, the algorithm scans B from the least to the most significant bit, and finds the index of the first bit that was not flipped to 1, which is at position $trail_1(B) + 1$. Notice that $trail_1(B)$ is an estimate of the uncommonness of the observed elements seen in the traffic stream. The estimate of n is given by $\hat{n} = 1.29281 \times 2^{trail_1(B)+1}$, where 1.29281 is a statistical correction factor. However, the standard deviation of $trail_1(B)$ is 1.12127. Hence, the standard error of $\hat{n}$ is a factor of at least 2 [16].

To reduce the variance of $trail_1(B)$, [16] proposed using I independent bitvectors, $B_1, \ldots, B_I$, populated by I independent hash functions, $h_1, \ldots, h_I$. $\hat{n} = 1.29281 \times 2^{avg(trail_1(B_i))+1}$, where $avg(.)$ is the average function. Taking the average reduces $stderr(\hat{n})$, by a factor of $O(\sqrt{I})$. However, the processing cost and space usage increase by a factor of I, since I hash functions are used and I bitvectors are maintained.

4.2 The PCSA Algorithm

To offset this increase in cost of PC by a factor of I, [16] proposed the Probabilistic Counting with Stochastic Averaging (PCSA) algorithm. PCSA employs a hash function, g, to distribute each element id into one of the I bitvectors. When scanning an element, e, the bitvector corresponding to $g(e)$ is updated using $h(e)$. Hence, each bitvector is only responsible for around n/I distinct elements, and $\hat{n} = 1.29281 \times I \times 2^{avg(trail_1(B_i))+1}$. Notice that all the bitvectors use the same function, h, which reduces the memory used for storing the hash functions by a factor of I. The expected standard error of the PCSA estimator is given by $stderr(\hat{n}) = 0.78/\sqrt{I}$. However, the PCSA estimator is only asymptotically unbiased as $n \to \infty$. The bias is given by $bias(\hat{n}) = E(\frac{\hat{n}}{n} - 1) = \frac{0.31}{I}$, which is negligible when compared to the standard error.

4.3 The AMS Algorithm

The hash functions assumed by [16] are ideal, and could be unrealistic, as argued in [1]. Alon *et al.* [1] proposed an algorithm for estimating the distinct count based on linear (pair-wise independent) hashing. The algorithm employs a linear hash function of the form $h(e) = (a \times e + b) \mod p^2$. Let $trail_0(.)$ be the number of trailing 0 bits. Notice that $trail_0(h(e))$ is geometrically distributed since $h(e)$ is uniformly distributed.

The algorithm does not store the bitvector, B. Instead, for each traffic entry, e, in the traffic dataset, the algorithm scans $h(e)$ from the least to the most significant bit, finds $trail_0(h(e))$, and keeps track of $\max(trail_0(h(e)))$. We define $\max(trail_0(h(e)))$ to be the maximum $trail_0(h(e))$ over all the traffic entries in S. The estimate of n is given by $\hat{n} = 2^{\max(trail_0(h(e)))}$. Using Markov and Chebyshev's inequalities, the authors showed that $\mathbf{Pr}\left[\frac{n}{c} < \hat{n} < n \times c\right] < \frac{2}{c}$, for any real $c > 2$. The authors use the one-sided Chebyshev's inequality to prove $\mathbf{Pr}\left[\hat{n} > n \times c\right] < \frac{1}{c}$. By working back-

[2]The performance of linear hashing is acceptable in real life if $\mathcal{N} \subseteq \{1, \ldots, p\}$, $\mathcal{N}, p \to \infty$, p is a prime, a and b are chosen at random, and $a \neq 0$. This yields acceptable pair-wise independence [7, 6].

ward, $\mathbf{Pr}\left[\frac{\hat{n}}{n} - 1 > c - 1\right] < \frac{1}{c}$. Thus, $stderr(\hat{n}) \geq 1$, since $c > 2$. In fact, it was shown in [13] that $\sigma(\max(trail_0(h(e)))) = 1.87$.

4.4 The LogLog Algorithms

Durand and Flajolet [13] introduced a PCSA-like extension to reduce the standard error of AMS by a factor of $O(\sqrt{I})$ by using I bitvectors, $B_1, \ldots, B_I$, each of which is responsible for around n/I distinct elements[3]. For each bitvector, B_i, the LogLog algorithm maintains $\max(trail_{i0}(h(e)))$, the maximum $trail_0(h(e))$ over all elements in S that hashed to B_i. The estimate of n is given by $\hat{n} = 0.39701 \times I \times 2^{avg(\max(trail_{i0}(h(eiv)))+1)}$. it was shown that $\hat{n}$ is asymptotically unbiased and that $stderr(\hat{n}) = \frac{1.30}{\sqrt{I}}$.

Maintaining the maximum trail of 0 bits for the I hash functions only require $I \times \log\log(\mathcal{N})$ bits, and hence the algorithm naming. The authors modifies LogLog into the Super-LogLog algorithm by suggesting two pragmatic improvements. First, Super-LogLog averages only on the smallest $0.7 \times I$ of the bitvectors[4]. Empirically, this reduces $stderr(\hat{n})$ to $\frac{1.05}{\sqrt{I}}$. Second, Super-LogLog uses small registers of size $\log_2\left(\lceil \log_2\left(\frac{n_{max}}{I}\right) + 3\rceil\right)$ bits, where n_{max} is an upper bound on n.

4.5 Analytically Comparing Logarithmic Hashing Algorithms

The PC algorithm performs I hashes for every element to reduce its $stderr(\hat{n})$. This is inefficient in terms of processing time. To prove our point experimentally, we implemented PC and we show that its run time is around 3 orders of magnitude more than the other logarithmic hashing algorithms in Section 10. AMS is a theoretical algorithm with high variance. We used it to explain the LogLog algorithms built on top of it, and we do not consider it further.

Super-LogLog has the smallest $stderr(\hat{n})$ as shown by the authors. While the standard error of PCSA is $0.78/\sqrt{I}$, and the standard error of Super-LogLog is $1.05/\sqrt{I}$, Super-LogLog uses a lot smaller sketches of $\log_2\left(\lceil \log_2\left(\frac{n_{max}}{I}\right) + 3\rceil\right)$ bits, while PCSA uses sketches of size $\log(\mathcal{N})$ bits. In our application, $\log(\mathcal{N}) \geq 128$, and n_{max} is 1.5×10^6, the number of daily distinct surfers. To attain a standard error of 0.001, PCSA needs 608400 sketches, each of size 128 bits, i.e., 9734400 Bytes, while Super-LogLog needs 1102500 sketches, each of size 4 bits, i.e., 551250 Bytes.

5. INTERVAL-BASED ALGORITHMS

The idea behind this class of algorithms is to hash the traffic entries into a finite interval. The number of distinct elements can be estimated based on how packed the interval is. If the interval is highly packed with elements, then this is an indication that the algorithm has seen so many distinct elements.

To achieve this, the algorithms distributes the entries to the finite interval using a uniform hash function as shown in Figure 4. The algorithms vary in the nature of the interval and how to estimate the density of the interval after hashing the entire traffic stream into it.

[3]Like AMS, the LogLog Algorithms do not materialize the bitvectors.

[4]The authors use another statistical correction factor instead of 0.39701. This constant was not mentioned in the paper. We empirically found 1.09295 to minimize the bias.

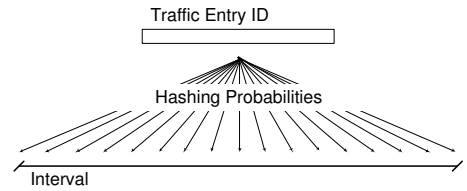


Figure 4: Interval-based algorithms employ a finite interval and a uniform hash function from each traffic entry id to a specific spot along the interval.

5.1 The Cohen Algorithm

Cohen [9] proposed an algorithm that uses a hash function, h, to hash[5] $\mathcal{N} \rightarrow [0 \ldots 1]$. The algorithm estimates the density of the interval $[0 \ldots 1]$ based on h_{min}, the minimum image of all the stream elements under h. The estimate of n is given by the unbiased estimator $\hat{n} = \frac{1}{h_{min}} - 1$. The algorithm has the right approximation (from the expectation sense) since by dissecting the unit interval with n boundaries into $n+1$ random segments, the smallest boundary is expected to be at $\frac{1}{n+1}$. To reduce the standard error by a factor of $O(\sqrt{I})$, for some constant I, the algorithm employs I parallel hash functions, $h_1, \ldots, h_I$. n is estimated by $\hat{n} = \frac{I}{\sum_{i=1}^{I} h_{min_i}} - 1$, where h_{min_i} is the minimum image of all the stream elements under h_i.

5.2 The BJKST1 Algorithm

Bar-Yossef *et al.* [3] proposed BJKST1, a modification of Cohen's algorithm [9] that uses a linear hash function, h, to hash $\mathcal{N} \rightarrow [1 \ldots |\mathcal{N}|^3]$, instead of the interval $[0 \ldots 1]$. To estimate the density of the interval, BJKST1 maintains a sample of the c_1 elements with the least hash values, where $c_1 = 96$. The estimate of n is given by $\hat{n} = \frac{c_1}{h_{max}} \times |\mathcal{N}|^3$, where h_{max} is the largest hash value in the sample. Intuitively, n independent elements, distributed uniformly in the interval $[1 \ldots |\mathcal{N}|^3]$, have around c_1 elements not greater than h_{max}. We note that this estimator is asymptotically unbiased with $bias(\hat{n}) = \frac{1}{n}$.

To reduce the standard error by a factor of $O(\sqrt{T})$, for some constant I, BJKST1 increases the number of samples to $c_1 \times T$, and estimates n by $\hat{n} = \frac{c_1 \times T}{h_{max}} \times |\mathcal{N}|^3$. The analysis in [3] shows that using $c_1 \times T$ samples, $\mathbf{Pr}\left[|\hat{n} - n| \geq n/\sqrt{T})\right] < 1/3$. Using Chebyshev's inequality, and working backward, $stderr(\hat{n}) \geq \sqrt{\frac{1}{3 \times T}}$. The algorithm runs I copies and takes the median result to reach a failure probability of δ, where I is $O(\log(\delta))$.

5.3 The Giroire Algorithm

Giroire proposed an alternative approach in [23] to reduce the standard error using I independent copies of the algorithm. Instead of averaging the results of the I independent copies of the algorithm, like in BJKST1, each copy is responsible for n/I elements. The results of the copies are combined in a stochastic averaging manner like in PCSA [16]. Alternative estimators were also proposed and analyzed by Giroire in [23] based on the same idea of the minimum hash[6].

[5]Alternative hashing distributions were considered in [9].

[6]Another analysis in [23] that does not consider pair-

Stochastic averaging coupled with the alternative estimators in [23] give lower $stderr(\hat{n})$ than BJKST1 when the number of samples is small. However, as the number of samples increases, the $stderr(\hat{n})$ of Giroire algorithm degenerates to that of BJST1[7].

5.4 Analytically Comparing Interval-Based Algorithms

When compared to Cohen's algorithm, BJKST1 is $O(T/\log(T))$ times faster since it hashes the stream elements only once, and inserts the hashed values in the sample using binary search trees in $O(\log(T))$ time. Giroire's modification makes the algorithm even faster by hashing every traffic element twice, to choose the sketch and insert the element into the sketch, instead of I times. In addition, the estimators in [23] give lower $stderr(\hat{n})$ than BJKST1 for limited space consumption, and are not any less accurate as the space usage increases. Therefore, we prefer Giroire's version of the BJKST1 algorithm over Cohen.

6. PURE-BUCKET-BASED ALGORITHMS

Like interval-based algorithms, pure-bucket-based algorithms employ uniform hashing. However, the elements domain is hashed to a set of buckets as shown in Figure 4. Initially, all buckets are empty. After hashing the entire traffic stream into the buckets, the number of distinct elements is estimated based on the probability that a bucket is empty (non-empty). The higher this estimate of this probability, the less (more) likely it is that many distinct elements were observed. The algorithms vary in the nature of the buckets and how to estimate the probability of emptiness.

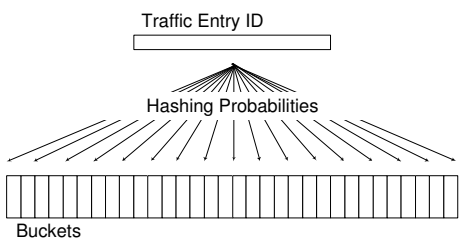


Figure 5: Pure-bucket-based algorithms employ a finite number of buckets and a uniform hash function from each traffic entry id to a specific bucket.

6.1 The BJKST2 Algorithm

BJKST2 [3] linearly hashes $\mathcal{N}$ to R buckets, where $2 \times n \leq R \leq 2c_2 \times n$, and $c_2 = 25$. BJKST2 estimates n based on the probability $r = 1 - (1 - (1/R))^n$, that a designated bucket, say bucket 0, is not empty. The authors handle the issue that R can be estimated only after the entire stream is processed[8] as we explain later. $\hat{n} = \frac{\ln(1-\hat{r})}{\ln(1-(1/R))}$, where $\hat{r}$ is an estimate of r.

We argue that estimating the probability r through using only one hash function (h), scanning the dataset, and checking if any element hashed to bucket 0, is a Bernoulli trial with a standard deviation of up to $\frac{1}{2}$. To reduce this standard deviation of $\hat{r}$, and consequently the standard error of $\hat{n}$, by a factor of $O(\sqrt{I})$, the authors pick I pair-wise independent hash function to hash the entire traffic. Let the number of hash functions that hashed any element in S to bucket 0 be J. BJKST2 estimates r by $\hat{r} = \frac{J}{I}$.

The authors handle the unknown R value by assuming that R is a power of 2, and by calculating $\hat{r}$ for all R's in $[2^1, \ldots, 2^{\lceil \log_2(\mathcal{N}) \rceil}]$. To simultaneously calculate $\hat{r}$ for all possible R's for each hash function, h_i, it suffices to maintain the maximum number of trailing 0 bits for h_i over all elements in S, $\max(trail_0(h_i(e)))$. To calculate $\hat{r}$ for $R_j = 2^j$, BJKST2 calculates the ratio of the hash functions whose $\max(trail_0(h_i(e))) \geq j$. Storing $\max(trail_0(h_i(e)))$ for all h_i's consumes $I \times \log\log(\mathcal{N})$ bits.

6.2 The Linear Counting Algorithm

Whang *et al.* proposed the simple Linear Counting (LC) in [42]. The algorithm keeps a bitvector, B, of size $m = \frac{n_{max}}{\rho}$ bits, where $0 < \rho \leq 12$ is the load factor. Each bit in B efficiently represents one bucket. Initially, all bits in B are set to 0. That is, all buckets are empty. The algorithm hashes every element in the traffic stream to a bit in the bitvector, which is set to 1, flagging that the corresponding bucket is not empty anymore. Assuming uniform hashing, the expected probability that a bucket is empty after inserting n distinct traffic entries is $V = (1 - \frac{1}{m})^n \approx e^{-\frac{n}{m}}$. This probability can be estimated by $\widehat{V}$, the ratio of 0 bits in the bitvector. Hence, n is estimated by the maximum likelihood estimator $\hat{n} = -m \ln(\widehat{V})$. The bias and the standard error of the estimator are $\frac{e^{\rho} - \rho - 1}{2n_{max}}$ and $\sqrt{\frac{e^{\rho} - \rho - 1}{\rho n_{max}}}$, respectively.

6.3 Analytically Comparing Pure-Bucket-Based Algorithms

BJKST2 runs I independent copies of the algorithm to reduce the standard error. Therefore, we do not expect its run time to be better than that of PC. On the other other hand, LC hashes each element only once. This run time difference is enough to recommend LC for any traffic analysis application.

We will not compare the standard errors for both algorithms here. Rather, we comment on the $stderr(\hat{n})$ and how it compares to the rest of the algorithms in Section 9.

7. HYBRID-BUCKET-SAMPLING ALGORITHMS

This class of algorithms employs a uniform hashing scheme to dissect the domain into buckets of almost equal number of elements. While scanning the traffic stream, the algorithms designates one of the buckets and counts the number of distinct traffic elements that hashed to this bucket. After scanning the entire traffic stream, the algorithm scales the number of elements in the designated buckets by the number of buckets to estimate the total distinct count in the entire traffic stream.

---

wise hashing collisions shows that using $c_1 \times I$ samples, $stderr(\hat{n}) \geq \sqrt{\frac{1}{c_1 \times I}}$, instead of $\sqrt{\frac{1}{3 \times I}}$.

[7]Built on top of the algorithms in [23], an algorithm for estimating distinct elements on sliding windows [12] was proposed in [17]. [17] has the same space complexity of the algorithm in [22], which is based on GT [21], but has less processing complexity per stream element.

[8]The authors use the AMS algorithm to estimate R to reduce the space usage. AMS is run in parallel to the algorithm.

7.1 The GT Algorithm

Gibbons and Tirthapura [21] presented an algorithm for counting distinct elements in the context of distributed streams using a linear hash function, h that hashes each traffic element into a number of buckets. The number of buckets used is 2^j, where j could be any number in the interval $[0 \ldots \lceil \log_2(\mathcal{N}) \rceil]$. For every j, GT counts s_j, the number of elements that hashed to a designated bucket, let it be bucket 0. The algorithm then scales up s_j by the number of buckets, 2^j, to estimate n. The bigger the value of j, the more the buckets, the smaller the number of elements hashing to bucket 0, the better the space complexity of the algorithm, and the higher the risk of undersampling. While undersampling is irreversible, oversampling is wasteful. The algorithm has to select a suitable j.

Let 2^{j^*} be the minimum number of buckets such that $s_{j^*} = |h_{j^*}^{-1}(0)| \leq d$, where $d = 192$. GT estimates n by $\hat{n} = s_{j^*} \times 2^{j^*}$. To find j^*, GT initially assumes $\widehat{j^*} = 0$, and scans each element e in S and inserts it, along with $trail_0(h(e))$, in a buffer of size d. At any point, if the buffer overflows, the algorithm discards from the buffer any elements, e, whose $trail_0(h(e)) = \widehat{j^*}$, and increments $\widehat{j^*}$. By the end of the stream, an element, e, is sampled in the buffer if and only if $trail_0(h(e)) \geq \widehat{j^*}$. The transition from one $\widehat{j^*}$ to the next can be enhanced by grouping the elements in linked lists by their trails. Hence, when GT increments $\widehat{j^*}$, it deletes the linked list containing all the elements whose trail equals the old value of $\widehat{j^*}$. This yields a constant amortized processing time per stream element. GT further improves the processing time by not deleting the list whose trail equals the old value of $\widehat{j^*}$ in bulk. Rather, the memory allocated for elements in this list is reused for new elements inserted in lists with larger trail values. This *lazy discarding* technique yields constant *expected* time[9].

By maintaining $d \times T$ samples, $\mathbf{Pr}\left[|\hat{n} - n| \geq n/\sqrt{T})\right] < 1/3$, for some $T \geq 1$. Using Chebyshev's inequality, and working backward, we deduce that $stderr(\hat{n}) \geq \sqrt{\frac{1}{3 \times T}}$. The algorithm runs I copies and takes the median result to reach a failure probability of δ, where I is $O(\log(\delta))$.

7.2 The BJKST3 Algorithm

The BJKST3 [3], an improvement on the algorithm in [4], highly resembles GT [21]. BJKST3 deviates from GT by storing each element, e, in the buffer in a hashed form, $g(e)$, using some linear function $g : [\mathcal{N}] \rightarrow [3 \times ((\log(\mathcal{N}) + 1) \times T)^2]$, for some $T \geq 1$, where $stderr(\hat{n})$ is $O(1/\sqrt{T})$. This increases the probability of error due to collisions on g. To make up for this source of error, BJKST3 keeps a buffer of size $c_3 = 576$ instead of $d = 192$. The analysis in [3] is almost identical to that in [21] and shows that using $c_3 \times T$ samples, $\mathbf{Pr}\left[|\hat{n} - n| \geq n/\sqrt{T}\right] < 1/3$. Hence, we deduce that $stderr(\hat{n}) \geq \sqrt{\frac{1}{3 \times T}}$.

[9]Due to the importance of sampling algorithms to conventional database systems, in [20], Gibbons combine the GT algorithm with reservoir sampling [41] to sample rows from a database. The goal is to enable counting distinct multi-attribute records satisfying subsequent predicates, by not destroying the database records by hashing. For every sample, e, in the buffer maintained by GT, Gibbons' algorithm apply reservoir sampling to collect a uniform sample of bounded size

BJKST3 improves on GT by indexing the elements in the buffer on their trails using balanced search trees, instead of the linked lists used by GT, to purge the buffer when incrementing $\widehat{j^*}$. When observing an element in the stream, the balanced search tree reduces the time to look up the element to know if it was already sampled. The *lazy discarding* technique can still be used to avoid deleting groups of elements at one time, and the expired balanced trees do not need to be re-balanced. However, BJKST3 pays the space penalty of the NULL pointers at the leaves of the tree.

7.3 Analytically Comparing Hybrid-Bucket-Sampling Algorithms

These algorithms have the advantages of generating a sample of the data that can be used later for other purposes. This can be valuable for some applications like database query optimization. However, in our application of traffic analysis, the problem with hybrid-bucket-sampling algorithms is that storing the samples can be very expensive, since the data space, $\mathcal{N}$, is huge, even though not all the traffic IDs exist in reality. This is why the BJKST3 algorithm stores samples in a hashed form. Now, we compare BJKST3 to GT, and show that BJKST3 is of limited benefit for traffic analysis.

BJKST3 resembles GT but uses $2\log(T\sqrt{3}(\log(\mathcal{N}) + 1))$ bits to store each element. Thus the total space for storing the sample is $576 \times T \times 2\log(T\sqrt{3}(\log(\mathcal{N}) + 1))$ bits, i.e., $288 \times T \log(T\sqrt{3}(\log(\mathcal{N}) + 1))$ Bytes, while GT uses $192 \times T \times \log(\mathcal{N})$ bits, i.e., $24 \times T \times \log(\mathcal{N})$ Bytes. In the case of search engines, cookie IDs identifying surfers are at least 128 bits. By substituting $\log(\mathcal{N}) = 128$, GT is more space efficient in cases where $T > 7.275$. Therefore, if $stderr(\hat{n})$ is required to be less than 0.2141, GT is preferred. In our case, this standard error is very high and we prefer GT always.

8. HYBRID-BUCKET-LOGARITHMIC ALGORITHMS

The only algorithm in this category is the Multiresolution Bitmap (MRB) Algorithm by Estan *et al.* [14]. The MRB and GT [21] algorithms were independently proposed. However, at a high level, the LC algorithm was used to compress the sampling buffer of GT.

The Multiresolution Bitmap Algorithm

The MRB, like GT, uses a hash function, h, and dissects the hash space to 2^j buckets[10], where $j \in [1 \ldots l]$ and l will be specified shortly. For every j, MRB counts $s_j = |h_j^{-1}(0)|$, the number of elements that hash to bucket 0. Scaling up s_j by 2^j yields an estimate of n. MRB deviates from GT by not physically keeping the sample of s_j elements, but by using a Linear Counting sketch with m bits to calculate $\widehat{s_j}$, an estimate of s_j. Notice that GT keeps $h_j^{-1}(0)$, the elements that hash to bucket 0 when the number of buckets is 2^j, from which it can extract $h_{j+1}^{-1}(0)$ when the number of buckets becomes 2^{j+1}. This is not possible for MRB since only an LC sketch of $h_j^{-1}(0)$ is maintained. To remedy this problem, MRB keeps parallel LC sketches for every possible j. All the LC sketches contain the same number of bits, m, but are responsible for ranges that are exponentially decreasing in j.

[10]In the paper, the authors considered dissecting the space to k^j buckets, and then optimized k to the value 2.

The LC sketches of the smaller j's are responsible for larger ranges, and hence can accurately estimate smaller distinct counts. Such LC sketches might have all their bits set to 1 if the distinct count is large, and only the LC sketches of the larger j's can estimate such larger distinct counts.

Keeping parallel m-bit LC sketches for every possible j results in any elements, e, being inserted in potentially several LC sketches. This is remedied by inserting e only in the LC sketch with the least possible range (the largest possible j) that contains $h(e)$. Since $e \in h_{j+1}^{-1}(0) \implies e \in h_j^{-1}(0)$, by inserting e only once in the LC sketch with the largest possible j containing e, half the space is saved, constant processing time per element is achieved, and all the LC sketches containing e can still be reconstructed at query time.

At query time, to estimate n, MRB scans all the LC sketches in a decreasing order in j until it finds a *base* LC sketch, as will be specified shortly. Let the base LC sketch be responsible for bucket 0 when the number of buckets is 2^{j^*}. Since MRB inserts an element e only once in the LC sketch with the largest possible j containing e, it estimates $s_{j^*} = h_{j^*}^{-1}(0)$, the number of elements hashing to bucket 0 when the number of buckets is 2^{j^*}, by summing up the estimates of all the elements inserted in the LC sketches whose $j \geq j^*$. That is, $\widehat{s_{j^*}} = \sum_{j=j^*}^{l} -m \ln(V_j)$, where V_j is the ratio of 0 bits in the LC sketch with 2^j buckets. Thus, $\hat{n} = \widehat{s_{j^*}} \times 2^{j^*}$.

Now, we explain how MRB determines the base sketch. It was shown that the estimate of MRB is asymptotically unbiased, and $stderr(\hat{n}) = \frac{\sqrt{e^{\rho}+e^{\rho/2}+2e^{\rho/4}-4}}{2\rho\sqrt{m}}$, which is minimized when $\rho = 1.9491$, where ρ is the average number of elements hashing to a bit. Hence, j^* is chosen such that ρ_{j^*} is the closest possible to 1.9491, where ρ_{j^*} is ρ of the base LC sketch.

MRB allocates m bits for every LC sketch. A larger number of bits was specified for $j = l$. The number of bits, m, allocated in all the levels including the last level, m_l, as well as the number of levels, l, is determined by an initialization procedure that minimizes the memory usage given a bound on $stderr(\hat{n})$. The number of LC sketches is $l = 2 + \lceil \log_2(\frac{n_{max}}{\rho_{max} \times m}) \rceil$. Hence, the total bit consumption is $((l-1) \times m + m_l) = \left(\left(1 + \lceil \log_2(\frac{n_{max}}{\rho_{max} \times m}) \rceil\right) \times m + m_l\right)$. Based on the experimental data used by *Estan et al.*, Durand *et al.* [13] crudely estimated $stderr(\hat{n}) \approx \frac{4.4}{\sqrt{\left(1+\lceil \log_2(\frac{n_{max}}{\rho_{max} \times m}) \rceil\right) \times m + m_l}}$.

9. ANALYTICAL EVALUATION

Now, we complete the analytical discussion of the algorithms aforementioned in Sections 4 through 8, and we make an analytical conclusion that Linear Counting is the most appropriate algorithm for our traffic analysis application.

9.1 Comparing the Best Candidate of each Category

We compare the candidates from each category that are most appropriate to the traffic analysis application. To start with, a quick comparison between interval-based candidates and hybrid-bucket-sampling candidates deems the hybrid-bucket-sampling candidates more space efficient. The reason is that Bar-Yossef *et al.* [3] compared their first algorithm, BJKST1, to their third algorithm, BJKST3, and concluded that the space consumption by BJKST3 is significantly less. Both algorithms have the same amortized per-element processing complexity. The Giriore's optimization on BJKST1 does not make up for this difference in space usage since it does not improve the asymptotic standard error as the space usage increases. In addition, GT is yet preferable over BJST3.

We follow by comparing the most prominent logarithmic hashing algorithm to the most prominent hybrid-bucket-sampling algorithm, GT. Super-LogLog is more space efficient than GT. The standard errors of Super-LogLog and GT are $1.05/\sqrt{I}$ and $\geq \left(\frac{1}{\sqrt{3}}\right)/\sqrt{I}$, respectively, where I is the number of times the algorithms are replicated. For each sketch, B_i, Super-LogLog does not store d trails along with their corresponding elements. Super-LogLog only keeps the maximum trail, $\max(trail_0(h(e)))$. Clearly, Super-LogLog is more accurate if both algorithms are run using the same space.

We compare LC, the most appealing candidate in the pure-bucket-based category, to Super-LogLog, the best candidate so far. Assuming n_{max} is 1.5×10^6, and the required standard error is 0.001, Super-LogLog runs in 551250 Bytes. If 551250 Bytes are allocated to LC, its standard error is 3.5694×10^{-4}.

It is very difficult to analytically compare the accuracy of MRB to any other algorithm since the initialization procedure that determines the space consumption is complex. Thus, it is difficult to analytically reason about the standard error in relation to the space usage. Even more, in our real data experiments in Section 10, we noticed that often times the initialization procedure of MRB finds that $l = 1$ minimizes the space for stringent standard error requirements that are required for traffic analysis. In such cases, MRB transforms into LC with some extra overhead. We leave comparing MRB to other algorithms to Section 10.

9.2 The Advantage of Linear Counting

LC has a $stderr(\hat{n})$ of $\sqrt{\frac{e^{\rho}-\rho-1}{\rho n_{max}}}$, where $\rho = n_{max}/m$, and m is the number of bits used. Assuming $10 \geq \rho \geq 1$, then LC achieves very low $stderr(\hat{n})$ of $O(1/\sqrt{n})$.

All of the algorithms discussed above, except for MRB that relies on LC, employ I independent copies of the estimators and average them, or collect a sample of $O(I)$, to reduce their constant standard errors by a factor of $\sqrt{I}$. Hence, to achieve the low standard error of LC, these algorithms need to set I to $O(n)$. Thus, the space usage of LC would almost be a lower bound on the space usage of these algorithms. This conforms with the lower bound of $\tilde{O}(\frac{1}{\epsilon^2})$ bits that was established in [43] on the space used by any approximation algorithm, for any error bound of ϵ of $\Omega(\frac{1}{\sqrt{N}})$, and suppressing the dependence on the failure probability δ.

10. EXPERIMENTAL RESULTS

In this section we motivate carrying out experiments for evaluating the discussed algorithms in traffic analysis environment. We follow by reporting our experimental results.

10.1 Motivating Experiments to Complement Theoretical Analysis

The theoretical analysis gives us good estimate of the accuracy of the algorithms given the space they use, as well as

their per-traffic element processing complexity. However, it does not serve as a sole basis for selecting an algorithm for our high accuracy and efficiency constraints. We mention some reasons why we need comprehensive experiments to select an algorithm.

- The theoretical analysis is not unified among all the algorithms. For instance some research works gave their space usage in terms of big-O notation ignoring the overhead of data structures (e.g. [21]) or that of storing hash functions (e.g. [9, 16]).
- Some works (e.g. [1, 3, 21]) made a clear distinction in their analysis between perfectly uniform hashing and linear (pair-wise independent) hashing, while other works (e.g. [14]) assumed perfectly uniform hashing which is not realizable. For instance, BJKST1 had a different $stderr(\hat{n})$ than that of Giriore, even though they are essentially the same algorithm. The reason is that Giriore assumed perfectly uniform hashing, while BJKST1 assumed linear hashing.
- In addition, not all works provided a concise form of their standard error (e.g., [1, 3, 21]). This is what made us estimate their standard error from their probabilistic analysis by using Chebyshev's inequality and working backward. This technique assumes the probabilistic analysis to be tight, which might not be true.

10.2 The Need for New Experimental Analysis

This is not the first experimental comparison between several distinct counting algorithms. In [2], Linear Counting was compared against PCSA; in [14], MRB was compared against PCSA; and in [13], LogLog, Super-LogLog, MRB, and PCSA were compared against each other. However, previous comparisons were not delineating in our application. In [2], the distinct counts estimated were of limited sizes not exceeding 24,359. For the more recent work, the accuracy regions considered are far from our needs. For instance, [13] considered 4% accuracy with probability of around 88%, while [14] experimented with various error ranges up to 10%.

10.3 Experimental Setup

In order to have a better practical comparison of the algorithms, we implemented eight prominent algorithms to compare their performance. We implemented Linear Counting, PC, PCSA, LogLog, Super-LogLog, MRB, GT (with some BJKST3 modifications), and BJKST1 (with all Giroire modifications). The algorithms we did not implement are AMS, Cohen, and BJKST2. Clearly, we implemented all the algorithms discussed above except AMS, which is originally proposed as a theoretical algorithm with high variance, as well as the algorithms performing I hashes for every element. However, we implemented one such algorithm, PC, to show the run time drawback of such algorithms.

The algorithms were implemented in VC++. All the hash functions used were linear of the form $h(e) = a \times e + b \bmod p$, where a and b are 64-bit numbers selected at random. Smaller a and b did not perform well for all algorithms. The experiments were run on a Personal Computer with Centrino Duo processor running at 2.0GHz with 1GB of main memory. The experiments were run using real surfer's cookie IDs from Ask.com with 1,925,423 distinct surfers, which is significantly larger than datasets of previous experiments.

10.4 Implementation Details

The sanitized dataset had the elements hashed from their domain of 256 bits to a smaller domain of 32 bits. The purpose of this domain reduction was to make it harder to guess the original cookie IDs. However, this actually biases the results to favor some logarithmic hashing algorithms like PC, PCSA, and with a lesser degree, the sampling algorithms, GT and BJKST1. For logarithmic hashing algorithms, the size of the sketches is dropped from 256 to 32 bits. For sampling algorithms, the samples' sizes are reduced from 256 bits 32 bits, though the overhead of the data structures keeping the samples are not reduced. The other algorithms whose memory consumption relies on the upper bound on the distinct count, n_{max}, such as LogLog, Super-LogLog, LC, and MRB, are not affected by this domain shrinking.

For MRB, we had difficulty constraining MRB to run in a predetermined space, since its input parameters are n_{max} and the required standard error. To limit MRB to a specified space usage, several runs were made with different standard errors until it used space that roughly equals the required space. We find this an inconvenience in usability.

The implementations of Linear Counting, PC, and PCSA were exactly like discussed in their respective papers. For LogLog, and Super-LogLog, the sketches were of size 8 bits. For GT, we stored the samples in an AVL tree as suggested in [3]. However, contrary to what [3] suggests, we stored the elements, rather than their signatures, as discussed in Section 7.3. In addition, we incorporated the *lazy discarding* technique in [21] to have an expected constant processing time per stream element. For BJKST1, we implemented both modifications suggested by Giriore. Specifically, we used the logarithmic estimator proposed in [23] and the results of the sketches were stochastically avergaed.

10.5 Comparing Algorithms' Run Times

This section is intended to show the run time drawback of the algorithms performing I hashes for every element, PC, Cohen and BJKST2. We only implemented one such algorithm, PC. Figure 6 shows the run time, averaged on 50 runs, for all the algorithms using various spaces. Most algorithms ran in roughly the same time, with LC being the fastest (around 1 second), since it performs only one hash per element, and BJKST1 being the slowest (around 5 seconds). The reason BJKST1 is relatively slow is that it inserts each element in two independent structures, the heap that keeps track of the minimum elements and the balanced tree that is used to check if an element has been sampled before or not in logarithmic time. This speed difference is not critical in real life, since all the algorithms are fast enough to handle the traffic arrival rate.

The only exception was PC. The run time of PC was larger than most algorithms by at least two orders of magnitude when the space usage was the least (1,250 sketches that consumed 25K). As more space was given to the algorithms, PC used more sketches and more hash functions, and its run time increased linearly. When the space usage was the most (12,500 sketches that consumed 250K), the run time of PC increased from 7.9 minutes to 1.3 hours, which is an increase by a factor of 10. We expect Cohen and BJKST2 to demonstrate similar behavior as their accuracy and space usage increase. This behavior is not suitable for applications with high data arrival rates like search engines.

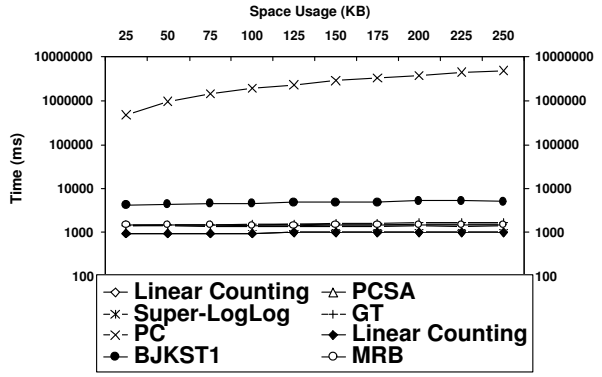


Figure 6: Average Run Time of 50 Experiments (Time axis has logarithmic scale)

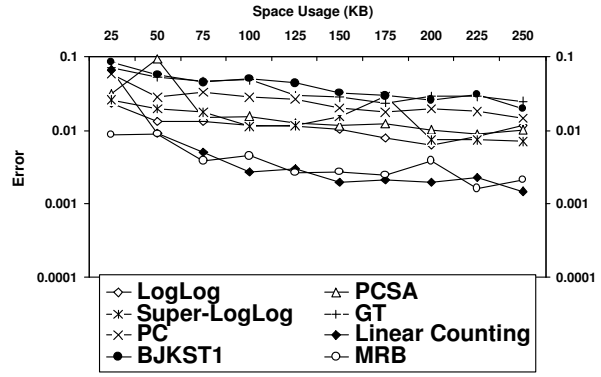


Figure 7: The Maximum Error of 50 Experiments (Error axis has logarithmic scale)

10.6 Comparing Algorithms' Accuracy

To compare the accuracy of the discussed algorithms, we fixed the space usage of all the algorithms and ran them 50 times on the real dataset. We repeated this for several space usages and recorded the maximum, and median error of estimation. The maximum and the median errors in estimation are plotted in the Figures 7 and 8, respectively.

10.6.1 Comparing the Algorithms to each other with Different Spaces

From Figures 7 and 8, it is worth noting that when the space usage was limited to 25KB, MRB had the least maximum error, followed by LogLog and Super-LogLog. The maximum error of BJKST1 was the worst. followed by GT, LC, PC and PCSA. However, When the space usage was limited to 50KB, both the maximum and the median errors of LC were the second best after MRB, which maintained only 2 levels of LC sketches. The difference between LC and MRB is very minor with the maximum and the median errors of LC being 9.1637×10^{-3} and 2.2863×10^{-3}, and those of MRB being 8.9430×10^{-3} and 1.3311×10^{-3}, respectively. BJKST1, GT, PC and PCSA continued to have the highest maximum errors. As is clear from Figure 7, this ranking continued to be true when the space usage was limited to 75KB, with the exception that Super-LogLog did not perform well and replaced PCSA in the list of bottom four algorithms.

As the space usage limit grew to 100KB, the MRB initialization procedure that minimizes the memory usage given a specific bound on the standard error found it more space efficient to keep only one level of LC sketches. Hence, MRB practically reduced to LC with some minor overhead in hashing and storage. Since then, LC was the most accurate estimation algorithm, which is clear from Figures 7 and 8. In all these figures, LC clearly had the lowest maximum and median estimation errors. MRB followed.

The list of bottom four algorithms continued to have BJKST1, GT, PC and PCSA with Super-LogLog entering this list occasionally. We attribute the poor performance of BJKST1, GT and PC to the high waste of space used to store the hash functions, in the case of PC, and the samples in the cases of GT, and BJKST1. GT and BJKST1 have very attractive theoretical space complexity. However, we expect them

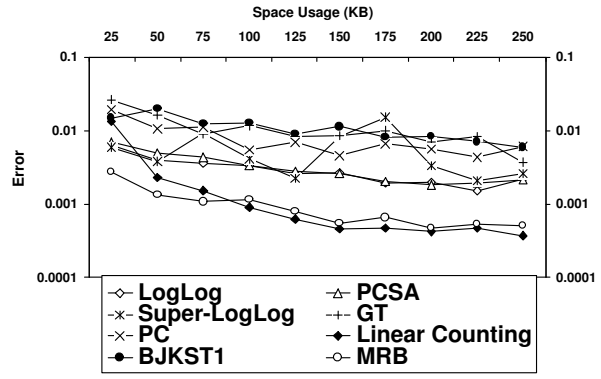


Figure 8: The Median Error of 50 Experiments (Error axis has logarithmic scale)

to be only valuable for applications that need to collect a sample of the dataset, like database optimization.

Between the space usages of 100KB and 225KB, BJKST1 performed the worst. However, GT performed the worst only when the space usage was 250KB. PCSA usually performed better then PC. We found Super-LogLog not to perform significantly better than LogLog on our dataset. The optimization of considering only the smallest $0.7 \times I$ sketches, where I is the number of sketches used, was based on empirical findings based on the dataset used in [13], which contrasts our results.

Notice that most of these experimental results conform with our analysis. As discussed in Section 9.1, Super-LogLog performs better than GT, and BJKST1 is expected to perform worse than GT. LC is expected to perform the best, as discussed in Section 9.2.

10.6.2 Commenting on the Change in some Algorithms' Accuracy with Different Spaces

We comment on how the maximum and median errors of some algorithms changed with the increase in space usage. From Figure 7, the maximum error of LC reduced the most with the increase in space usage. The maximum errors of LC, MRB, BJKST1, GT and PC were the most stable. That is, they did not have unexplained peaks as the space usage varied. The maximum error of PCSA peaked when the space

usage was 50KB, and that of Super-LogLog peaked when the space usage was 175KB.

Again, from Figure 8, the median error of LC reduced the most with the increase in space usage. This is explained in Section 9.2. All algorithms had a stable median error except Super-LogLog. Super-LogLog had unexplained peaks when the space usages were 75KB, and 175KB.

11. DISCUSSION AND FUTURE WORK

In this paper, we have reviewed the literature of estimating the distinct count of a huge dataset under the constraints of a modern application that pushes for highly accurate estimates. We explained how search engines benefit from this stream analysis problem, and have motivated the need for using an algorithm whose sketches are accurate and mergeable. We have described the algorithms proposed to accurately solve this problem, and have selected Linear Counting due to its superb accuracy, and run time, and ease of implementation. While this algorithm has always been discarded as using linear space in most research works, we have shown both analytical and experimental comparison to other algorithms favor Linear Counting when highly accurate results are required. For other estimators to attain comparable accuracy, they need space that is more than what is used by Linear Counting.

It is straightforward to verify that the merging of two sketches of any of the aforementioned algorithms yields the same accuracy as running the algorithm on the union of the two streams. However, the merging process could be faster for some algorithms than others. The first future work direction is to analytically and experimentally compare the processing time for merging the sketches of different algorithms. We expect Linear Counting to be the fastest at merging sketches, since its merging process is a mere OR-ing of the space used by the sketches with no added complexity.

We also plan to devise a variant of Linear Counting for sliding windows. In addition, we like to analytically and empirically examine the tradeoff between running several parallel images of the algorithm and averaging the results versus running one image of the algorithm if the same space is used.

12. REFERENCES

[1] N. Alon, Y. Matias, and M. Szegedy. The Space Complexity of Approximating the Frequency Moments. In *Proceedings of the 28th ACM STOC Symposium on the Theory of Computing*, pages 20–29, 1996. An extended version appeared in the Journal of Computer and System Sciences, 58(1):137–147, 1999.

[2] M. Astrahan, M. Schkolnick, and K-Y. Whang. Approximating the Number of Unique Values of an Attribute Without Sorting. *Information Systems*, 12(1):11–15, 1987.

[3] Z. Bar-Yossef, T. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting Distinct Elements in a Data Stream. In *Proceedings of the 6th RANDOM International Workshop on Randomization and Approximation Techniques*, pages 1–10, 2002.

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

[5] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *Proceedings of the 18th ACM SIGMOD International Conference on Management of Data*, pages 359–370, 1999.

[6] T. Bohman, C. Cooper, and A. Frieze. Min-Wise Independent Linear Permutations. *Electronic Journal of Combinatorics*, 7:R26, 2000.

[7] A. Broder, M. Charikar, A. Frieze, and M. Mitzenmacher. Min-Wise Independent Permutations (Extended Abstract). In *Proceedings of the 30th ACM STOC Symposium on Theory Of Computing*, pages 327–336, 1998. An extended version appeared in the Journal of Computer and System Sciences, 60(3):630–659, 2000.

[8] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards Estimation Error Guarantees for Distinct Values. In *Proceedings of the 19th ACM PODS Symposium on Principles of Database Systems*, pages 268–279, 2000.

[9] E. Cohen. Size-Estimation Framework with Applications to Transitive Closure and Reachability. *Journal of Computer and System Sciences*, 55(3):441–453, 1997.

[10] G. Cormode, M. Datar, P. Indyk, and S. Muthukrishnan. Comparing Data Streams Using Hamming Norms (How to Zero In). In *Proceedings of the 28th VLDB International Conference on Very Large Data Bases*, pages 335–345, 2002. An extended version appeared in IEEE TKDE Transactions on Knowledge and Data Engineering, 15(3):529–540, 2003.

[11] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Diamond in the Rough: Finding Hierarchical Heavy Hitters in Multi-Dimensional Data. In *Proceedings of the 23rd ACM SIGMOD International Conference on Management of Data*, pages 155–166, 2004.

[12] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining Stream Statistics over Sliding Windows. In *Proceedings of the 13th ACM-SIAM SODA Symposium On Discrete Algorithms*, pages 635–644, 2002.

[13] M. Durand and P. Flajolet. Loglog Counting of Large Cardinalities (extended abstract). In *Proceedings of the 11th ESA European Symposium on Algorithms*, pages 605–617, 2003.

[14] C. Estan, G. Varghese, and M. Fisk. Bitmap Algorithms for Counting Active Flows on High Speed Links. In *Proceedings of the 3rd ACM SIGCOMM IMC conference on Internet Measurement*, pages 153–166, 2003. An extended version appeared in the IEEE/ACM Transactions on Networking, 14(5):926–937, 2006.

[15] Y. Feng, D. Agrawal, A. El Abbadi, and A. Metwally. Range cube: Efficient cube computation by exploiting data correlation. In *Proceedings of the 20th IEEE ICDE International Conference on Data Engineering*, pages 658–670, 2004.

[16] P. Flajolet and G. Martin. Probabilistic Counting

Algorithms. *Journal of Computer and System Sciences*, 31:182–209, 1985.

[17] É. Fusy and F. Giroire. Estimating the Number of Active Flows in a Data Stream over a Sliding Window. In *Proceedings of the 4th SIAM Workshop on Analytic Algorithmics and Combinatorics*, 2007.

[18] S. Ganguly. Counting Distinct Items over Update Streams. In *Proceedings of the 16th ISAAC International Symposium on Algorithms and Computation*, pages 505–514, 2005.

[19] S. Ganguly, M. Garofalakis, and R. Rastogi. Processing Set Expressions over Continuous Update Streams. In *Proceedings of the 22nd ACM SIGMOD International Conference on Management of Data*, pages 265–276, 2003.

[20] P. Gibbons. Distinct Sampling for Highly-Accurate Answers to Distinct Values Queries and Event Reports. In *Proceedings of the 27th VLDB International Conference on Very Large Data Bases*, pages 541–550, 2001.

[21] P. Gibbons and S. Tirthapura. Estimating Simple Functions on the Union of Data Streams. In *Proceedings of the 13th ACM SPAA Symposium on Parallel Algorithms and Architectures*, pages 281–291, 2001.

[22] P. Gibbons and S. Tirthapura. Distributed Streams Algorithms for Sliding Windows. In *Proceedings of the 14th ACM SPAA Symposium on Parallel Algorithms and Architectures*, pages 63–72, 2002.

[23] F. Giroire. Order Statistics and Estimating Cardinalities of Massive Datasets. In *Proceedings of the 6th DMTCS Discrete Mathematics and Theoretical Computer Science*, pages 157–166, 2005.

[24] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[25] P. Haas, J. Naughton, S. Sehadri, and L. Stokes. Sampling-Based Estimation of the Number of Distinct Values of an Attribute. In *Proceedings of the 21st VLDB International Conference on Very Large Data Bases*, pages 311–322, 1995.

[26] P. Haas and L. Stokes. Estimating the Number of Classes in a Finite Population. *Journal of the American Statistical Association*, 93:1475–1487, 1998.

[27] J. Han, Y. Chen, G. Dong, J. Pei, B. Wah, J. Wang, and Y. Cai. Stream Cube: An Architecture for Multi-Dimensional Analysis of Data Streams. *Distributed and Parallel Databases*, 18(2):173–197, 2005.

[28] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing Data Cubes Efficiently. *ACM SIGMOD Record*, 25(2):205–216, 1996.

[29] W.-C. Hou, G. Özsoyoglu, and E. Dogdu. Error-Constrained COUNT Query Evaluation in Relational Databases. In *Proceedings of the 10th ACM SIGMOD International Conference on Management of Data*, pages 278–287, 1991.

[30] W.-C. Hou, G. Özsoyoglu, and B. Taneja. Statistical Estimators for Relational Algebra Expressions. In *Proceedings of the 7th ACM PODS Symposium on Principles of Database Systems*, pages 276–287, 1988.

[31] W.-C. Hou, G. Özsoyoglu, and B. Taneja. Processing Aggregate Relational Queries with Hard Time Constraints. In *Proceedings of the 8th ACM SIGMOD International Conference on Management of Data*, pages 68–77, 1989.

[32] L. Lakshmanan, J. Pei, and J. Han. Quotient Cube: How to Summarize the Semantics of a Data Cube. In *Proceedings of the 28th VLDB International Conference on Very Large Data Bases*, pages 778–789, 2002.

[33] L. Lakshmanan, J. Pei, and Y. Zhao. QC-Trees: an Efficient Summary Structure for Semantic OLAP. In *Proceedings of the 22nd ACM SIGMOD International Conference on Management of Data*, pages 64–75, 2003.

[34] X. Li, J. Han, and H. Gonzalez. High-Dimensional OLAP: A Minimal Cubing Approach. In *Proceedings of the 30th VLDB International Conference on Very Large Data Bases*, pages 528–539, 2004.

[35] A. Metwally, D. Agrawal, A. El Abbadi, and Q. Zheng. On Hit Inflation Techniques and Detection in Streams of Web Advertising Networks. In *Proceedings of the 27th IEEE ICDCS International Conference on Distributed Computing*, 2007.

[36] K. Morfonios and Y. Ioannidis. CURE for Cubes: Cubing Using a ROLAP Engine. In *Proceedings of the 32nd VLDB International Conference on Very Large Data Bases*, pages 379–390, 2006.

[37] J. Naughton and S. Seshadri. On Estimating the Size of Projections. In *Proceedings of the 3rd International Conference on Database Theory*, pages 499–513, 1990.

[38] F. Olken and D. Rotem. Random Sampling from Databases: A Survey. *Journal of Statistics and Computing*, 5:25–42, 1995.

[39] G. Özsoyoglu, K. Du, A. Tjahjana, W.-C. Hou, and D. Y. Rowland. On Estimating COUNT, SUM, and AVERAGE Relational Algebra Queries. In *Proceedings of the 2nd DEXA International Conference on Database and Expert Systems Applications*, pages 406–412, 1991.

[40] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: Shrinking the PetaCube. In *Proceedings of the 21st ACM SIGMOD International Conference on Management of Data*, pages 464–475, 2002.

[41] J. Vitter. Random Sampling with a Reservoir. *ACM TOMS Transactions on Mathematical Software*, 11(1):37–57, 1985.

[42] K-Y. Whang, B. Vander-Zanden, and H. Taylor. A Linear-Time Probabilistic Counting Algorithm for Database Applications. *ACM TODS Transactions on Database Systems*, 15:208–229, 1990.

[43] D. Woodruff. Optimal Space Lower Bounds for all Frequency Moments. In *Proceedings of the 15th ACM-SIAM SODA Symposium On Discrete Algorithms*, pages 167–175, 2004.