Summarizing and Mining Inverse Distributions on Data Streams via Dynamic Inverse Sampling

Graham Cormode
Bell Laboratories
cormode@lucent.com

S. Muthukrishnan
Rutgers University
muthu@cs.rutgers.edu

Irina Rozenbaum
Rutgers University
rozenbau@paul.rutgers.edu

Abstract

Emerging data stream management systems approach the challenge of massive data distributions which arrive at high speeds while there is only small storage by summarizing and mining the distributions using samples or sketches. However, data distributions can be "viewed" in different ways. A data stream of integer values can be viewed either as the *forward* distribution $f(x)$, ie., the number of occurrences of x in the stream, or as its inverse, $f^{-1}(i)$, which is the number of items that appear i times. While both such "views" are equivalent in stored data systems, over data streams that entail approximations, they may be significantly different. In other words, samples and sketches developed for the forward distribution may be ineffective for summarizing or mining the inverse distribution. Yet, many applications such as IP traffic monitoring naturally rely on mining inverse distributions.

We formalize the problems of managing and mining inverse distributions and show provable differences between summarizing the forward distribution vs the inverse distribution. We present methods for summarizing and mining inverse distributions of data streams: they rely on a novel technique to maintain a dynamic sample over the stream with provable guarantees which can be used for variety of summarization tasks (building quantiles or equidepth histograms) and mining (anomaly detection: finding heavy hitters, and measuring the number of rare items), all with provable guarantees on quality of approximations and time/space used by our streaming methods. We also complement our analytical and algorithmic results by presenting an experimental study of the methods over network data streams.

Proceedings of the 31st VLDB Conference,
Trondheim, Norway, 2005

1 Introduction

Database systems are evolving to handle high speed data "streams" where transactions arrive rapidly and have to be processed while storing only a limited amount of information. Many applications generate data streams: IP traffic streams, click streams, financial transactions, text streams at application level, sensor streams. Each of these applications demands systems to manage the vast streams and provide basic analyses or mining capability. For example, in the IP traffic analysis example, there is a great demand for tools to analyze IP traffic in order to find patterns for network provisioning, detect anomalous behavior and intrusions for security purposes, verify Service Level Agreements based on type and volume of data, supply customer reports based on application-specific details such as peer-to-peer or proprietary protocols, monitor for law enforcement and governing purposes (eg., CALEA[1]), etc. Therefore, there is a suite of reasons for using DSMSs in IP networks. In AT& T for example, the Gigascope DSMS [11] is used operationally, while Sprint uses IPMON and CMON [2]; many small businesses (eg., NARUS[3]) build and service such systems for ISPs. A similar case has been built for using DSMSs in the financial industry (eg., StreamBase[4]) and in government [25]. Many general purpose DSMSs are also being developed [30, 1, 5].

DSMSs approach the task of handling and mining massive data streams by *summarizing* the streams in small space. These summaries may be various "samples" (selection of subsets of items by sampling with or without replacement, weighted sampling, deterministic sampling, etc) or "sketches" (inner product or aggregate of subsets of items using different hash functions that compactly describe the subsets in each inner product). Sampling and sketching solutions have been designed for a number of

[1]`http://www.calea.org`
[2]`http://ipmon.sprint.com/`
[3]`http://www.narus.com`
[4]`http://www.streambase.com/`

tasks such as finding heavy hitters, change detection, quantiling, histogramming, etc. (See recent surveys and tutorials [28, 18, 3] etc.) For most of these tasks, a precise answer is not paramount and also impossible to obtain within the limited space and time constraints of DSMSs. Therefore, workable approximations are necessary and indeed suffice in these applications. As a result, samples or sketches have proved to be a suitable fit in DSMSs since they provide accuracy guarantees and have small footprint. Both sampling and sketching are used in Gigascope [9] and CMON.

The departure of our work from extant literature emerges from our experience with IP traffic stream analysis: input streams can be "viewed" in different ways, and the summaries built to manage and mine one "view" may differ significantly from those used for another.

1.1 Motivating Example: Forward and Inverse Views

We will expose the phenomenon of different "views" of the input data stream using an example drawn from the IP traffic analysis case. Consider the IP traffic on a link as packet p representing (i_p, s_p) pairs where i_p is the source IP address and s_p is the size of the packet (there are other attributes of IP traffic on the link—destination IP addresses, port numbers, payload or content—but for exposition, we focus on these attributes).

Problem A. Which IP address sent the most bytes? That is, find i such that $\sum_{p|i_p=i} s_p$ is maximum.

Problem B. What is the most common volume of traffic sent by an IP address? That is, find traffic volume W such that $|\{i|W = \sum_{p|i_p=i} s_p\}|$ is maximum.[5]

Both Problem A and B arise naturally in IP traffic analysis. Problem A is a simplification of the problem of finding the "elephant flows" [14]. Problem B is related to estimating the number of "mice" (small flows) and is a generalization of the problem of estimating the number of flows with small number of packets [13]. "Port scanning" attacks, which probe a large number of ports looking for vulnerabilities by trying to open connections on each port have low volume per flow, but show up as small W's in Problem B.

For Problem A, there are many known solutions using samples [26] or sketches [10], and these solutions have even been tested in live DSMSs on IP traffic [9]. In contrast, we are not aware of any solutions for Problem B with strong guarantees.

1.2 Formalizing Different Views

We formalize the problems as follows:

• Problem A deals with the *forward* distribution, that is, we work on $f[0 \ldots U]$ where $f(x)$ is the number of bytes sent by IP address x. Each new packet (i_p, s_p) results in $f[i_p] \leftarrow f[i_p] + s_p$. We ask what is the x for which $f[x]$ is the largest.

• Problem B deals with the *inverse* distribution, that is, we work on $f^{-1}[0 \ldots K]$ where each new packet (i_p, s_p) results in $f^{-1}[f[i_p]] \leftarrow f^{-1}[f[i_p]] - 1$ and $f^{-1}[f[i_p] + s_p] \leftarrow f^{-1}[f[i_p] + s_p] + 1$. We ask which i gives the largest $f^{-1}(i)$.

For conventional DBMSs where the input can be stored, both views f and f^{-1} are equivalent as both can be expressed by nested SQL queries. So, if the data is stored, we can derive either. However, in DSMSs where we maintain only a summary of the data, f and f^{-1} can not be readily derived, and operating on one from the input data stream is fundamentally different from operating on the other. Of course, summaries of *both* f and f^{-1} are of interest since they give an idea of traffic size distribution in two, quite different ways. Similarly, mining f and f^{-1} for changes or anomalies will show quite different phenomena. However, much of extant literature has developed methods for summarizing and mining f, but not much is known for summarizing and mining f^{-1}.

Methods that have been successful in mining the forward distribution do not obviously apply to f^{-1}. Consider maintaining the popular AMS [2] sketch on f^{-1} on the data stream. Each new packet modifies f^{-1}; because its AMS sketch is based on *precisely* knowing $f[i_p]$, it is provably impossible to know all i_p's in a small space streaming setting. In other words, each new packet changes the "domain" itself in a way we can not track in small space over the stream. Hence, sketch methods that rely on knowing the precise domain value of each new update such as the AMS sketch and all its variations fail directly.

1.3 Our Contributions

Our contribution is to introduce the problems of summarizing and mining the inverse distribution, and proposing solutions in full generality for them. More precisely:

1. We formalize the problems of summarizing and mining the inverse data distribution on data streams. We have given intuition why samples and sketches developed for the forward distribution does not solve the inverse distribution problems. We go on to prove concrete lower bounds that separate the performance of algorithms for problems on forward vs. inverse distribution on data streams, no matter what techniques are used.
2. We present a general summary for the inverse distribution based on *dynamic inverse sampling* and an algorithm to maintain such samples dynamically, in presence of *both* inserts and deletes, with provable guarantee. No such dynamic sampling method was previously known. Using such samples, we present algorithms with provable guarantees for a number of inverse distributions problems including heavy hitters, range queries, quantiles, etc.
3. We complement our analytical and algorithmic results by a thorough implementation study on real data and show that our methods are both practical and effective.

[5]This can be thought of as determining the popular bandwidth requirement for hosts. In more detail, one may group bandwidth into ranges of volume 1—2KB, 2—3KB, etc. and ask this question on such ranges rather than precise volumes.

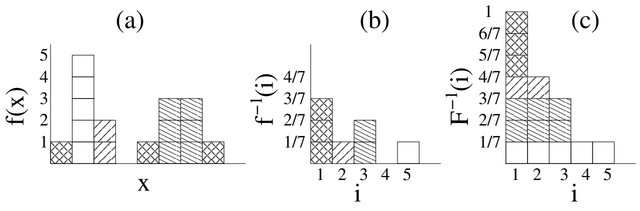


Figure 1: Example distribution, shaded to indicate items with the same count. (a) Forward distribution f where items have counts $\{1, 5, 2, 0, 1, 3, 3, 1, 0\}$ (b) Inverse distribution, f^{-1}. (c) Cumulative inverse distribution, F^{-1}.

Our approach extends to a variety of scenarios, and smoothly handles continuous distributions, fractional counts, working on the sum or difference of two distributions and so on.

2 The Inverse Distribution

Let f be a discrete distribution over a large set X, with the semantics that $f(x \in X) = i$ means that item x occurs i times. Let $N = \sum_{x \in X} f(x)$, the total number of items, and $D = |\{x|f(x) > 0\}|$, the number of distinct items. The inverse distribution is defined as follows:

Definition 1. *The* inverse distribution*,* $f^{-1}(i)$ *gives the fraction of items from* X *whose count is* i*. That is,* $f^{-1}(i) = |\{x|x \in X, f(x) = i, i \neq 0\}|/D$.[6]

The cumulative inverse distribution*,* $F^{-1}(i)$ *is defined as* $\sum_{j \geq i} f^{-1}(j)$*.*[7]

For clarity and simplicity, we assume that f is a discrete, integer valued distribution, but generalizations to continuous or real valued distributions follow naturally. An example is shown in Figure 1. From this figure, it can be seen that $N = \sum_i i f^{-1}(i) D = \sum_i F^{-1}(i) D$.

2.1 Queries on the Inverse Distribution

Queries on the inverse distribution give a variety of information about the distribution itself. We define the following queries on the inverse distribution:

• *Point Queries* on the inverse distribution are, given i, to return $f^{-1}(i)$. This corresponds to finding the fraction of items that occurred exactly i times. For example, finding $f^{-1}(1)$ over a stream of network flows corresponds to finding flows consisting of a single packet — possible indication of a probing attack if $f^{-1}(1)$ is large. This quantity is sometimes known as the *rarity* of the distribution.

• *Range Queries* on the inverse distribution generalize point queries and given a range $[j, k]$ return $\sum_{i=j}^{k} f^{-1}(i) = F^{-1}(j) - F^{-1}(k+1)$. Thus in a database of transactions, one could ask "what percentage of items sold between 10 and 20 units last month" by computing the Inverse Range Query $[10, 20]$ over the appropriate relation.

[6]This definition forces $f^{-1}(0) = 0$ so that $\sum_i f^{-1}(i) = 1$.

[7]This definition computes the cumulative distribution of items with counts i or *above*, not i or below, which is equal to $1 - F^{-1}(i)$.

• *Inverse Heavy Hitters* applies the notion of Heavy Hitters (frequent items) to the inverse distribution. Given a fraction ϕ, an Inverse Heavy Hitters Query must return $\{i|f^{-1}(i) > \phi\}$. That is, which are the item counts that occur most frequently?

• *Inverse Quantiles* takes a fraction ϕ and returns the ϕ-quantile of the inverse distribution. That is, return the i such that $F^{-1}(i-1) > \phi, F^{-1}(i) \leq \phi$. This allows to pose queries such as, over a stream of connections, what is the median number of connections made by consumers.

2.2 Computational Challenge

All the queries we have defined can be answered exactly by taking the original distribution and performing sorting and scanning passes over it. However, we seek solutions that can answer queries on high speed data streams, consisting of an arbitrary mix of insertions and deletions. Deletions arise in many traditional database settings, where records are inserted and deleted; they also occur in the network scenarios we have discussed as flows begin and end. Hence our solutions must consume only small space (much smaller than the number of updates, and also smaller than the size of the domain $|X|$). We analyze the complexity of answering these queries rapidly and using only small space, by allowing approximation and probabilistic methods. In general, several computations over the inverse distribution are strictly harder than their counterparts over the original distribution. We demonstrate this for both exact and approximate query answering:

Lemma 1. Fixed point queries *are point queries where the point is given ahead of the data. Fixed point queries can be answered exactly on the original distribution using constant space (by simply counting the number of times the given item occurs). They require space linear in the number of items,* $|X|$ *to compute on the inverse distribution. A probabilistic, relative error approximation still requires linear space.*

Lemma 2. *The number of distinct values in the original distribution,* $F_0(f)$ *can be approximated up to a fixed error with constant probability in* $O(1)$ *space. However, the number of distinct values in the inverse distribution* $F_0(f^{-1})$*, requires linear space to approximate to a constant factor.*

Both lemmas follow by reducing the communication complexity problem of disjointness [23]. to the queries over the inverse distribution (we omit proofs for brevity)

We seek good approximations for the queries we have defined over the inverse distribution, with strong guarantees of the quality. To do this, we develop a new technique, *Dynamic Inverse Sampling*, which effectively samples uniformly from the inverse distribution, as the original distribution is modified by insert and delete transactions. We will show how using this sample can give good estimators for the queries over the inverse distribution.

There are two challenges in this approach. First, maintaining random samples in the presence of inserts and deletes in one-pass is quite challenging. All known methods resort to rescanning the past relation for populating the sample when it dwindles under deletes. In order to make our goals feasible, we must disallow the "adversarial" strategy that asks for a sample from the inverse distribution and then deletes the sampled items, and repeats. Clearly, such a strategy can force any sampling method that uses sublinear space to end up with an empty sample. We are able to prove strong guarantees on our dynamic inverse sampling algorithm under the standard assumption in probabilistic algorithms that the randomization (coin tosses) our algorithm uses is not known to the adversary. The adversary may not use the output of queries to affect the stream of updates (equivalently, we assume that the updates are specified in advance). The second challenge is that as we show below, existing techniques of sampling from the original distribution, and sketch summarization, fail to answer our queries; this emphasizes the importance of sampling from the inverse distribution.

Lemma 3. *A uniform sample from the forward distribution based on probing records is insufficient to answer queries on the inverse distribution.*

Proof Sketch. Consider the distribution where one item occurs $N - k$ times, and k items occur once each, for some constant k, e.g. $k = 2$. Unless the sample of items is linear in N, it is unlikely to draw any of the k items which occur once, and so cannot distinguish this distribution from one where one item occurs N times. But in the first distribution, $f^{-1}(1) = 1 - 1/(k + 1)$, whereas in the second it is 0. To correctly distinguish between these two cases, a very large sample is required. $\square$

Lemma 4. *A sketch synopsis of the forward distribution is insufficient to answer queries on the inverse distribution.*

Proof Sketch. Queries to sketch data structures, such as the AMS sketch [2], estimate the count of individual items with additive error related to the L_2 norm of the distribution. To guarantee accurate answers to queries on the inverse distribution, this error must be very small, requiring the sketch to be at least linear in D (number of distinct values). $\square$

3 Dynamic Inverse Sampling: Insertions

Our methods to answer queries on the inverse distribution rely on a technique that we call "Dynamic Inverse Sampling" (DIS). The goal of this technique is to process a sequence of insertions and deletions and then be able to draw samples uniformly from the inverse distribution. Each sample is drawn with replacement, and returns a pair uniformly from the set of $\{(i, x) | x \in X, f^{-1}(x) = i\}$. The size of this set is D, the number of distinct items in X, and so the probability of returning any pair is $\frac{1}{D}$.

In order to simplify the exposition, we introduce our dynamic inverse sampling method when the input consists of insertions only. This shows the main structure of the algorithm. In subsequent sections, we will show how to generalize this to our main case of interest, where the input can consist of an arbitrary sequence of insertions and deletions.

3.1 Data Structure and Update Procedure

We first describe the main structure, which draws a pair (i, x) from the inverse distribution. We later analyze how many independent copies of this data structure are required to guarantee a sample of sufficiently large size. At a high level, the procedure works by hashing the items to levels such that the likelihood of being hashed to level l is exponentially decreasing in l. So at some level $l \approx \log D$ there is a high probability that only one item hashes there, and we recover this item and its count as the sampled count. In order to prove correctness, we will have to show that this item is selected uniformly, and that there is at least constant probability that there is a level that has a unique item for us to return. Throughout, we assume that $X = [0 \dots m - 1]$ for some m such that any $x \in X$ is represented in a single machine word; our approach naturally generalizes to other settings but we focus on this case for simplicity.

Data Structure. Our data structure takes two parameters: (1) a ratio $0 < r < 1$ which is used to partition the input items (2) M, the range of the hash function used to determine where items are stored within the data structure. We fix values for these parameters based on our analysis. The size of the data structure is proportional to $L = \log_{1/r} M$. We keep three arrays of length L: $item$, which stores items from the input; $count$, which stores item counts; and boolean flags $uniq$. We initialize the array of counts to zero. We keep a hash function h which maps from $[1 \dots m]$ to $[1 \dots M]$. For the purposes of the analysis, we require h to be (strongly) universal. Such hash functions are very fast to compute and require only a constant amount of space [4].

Update process. For each insertion of an item x, we use the hash function h to determine where in the data structure it belongs. From h, we define

$$h_l(x) = \lfloor h(x)/(r^l * M) \rfloor$$
$$l(x) = l \iff h_l(x) = 0, h_{l+1}(x) \neq 0.$$

The value $l(x)$ determines the place where x is stored in the data structure (it is the greatest l such that $h_l(x) = 0$). Observe that $l(x)$ can be computed in constant time by solving $h_l(x) = l$, which sets $l(x) = \lceil \log_{1/r}(M/h(x)) \rceil$. We inspect $count[l(x)]$: if it is zero, then no item is stored there, and so we set $item[l(x)] = x$, and set $uniq[l(x)] =$ `true`. If $count[l(x)]$ is not zero, we inspect $item[l(x)]$. If $item[l(x)] \neq x$, then we have a *collision*, and we set $uniq[l(x)] =$ `false`. Lastly, in all cases we increment $count[l(x)]$.

Output process. In order to output an item from the data structure, we search the data structure. We describe two variations, one with guaranteed bounds, and a second, "greedy" approach that extracts as many samples as possible from the data structure. Begin by setting $l = L$. If

Time	Level 1			Level 2			Level 3		
Step	item	count	uniq	item	count	uniq	item	count	uniq
1.	4	1	T	0	0	T	0	0	T
2.	4	1	T	7	1	T	0	0	T
3.	4	2	T	7	1	T	0	0	T
4.	4	3	F	7	1	T	0	0	T
5.	4	3	F	7	2	F	0	0	T
6.	4	4	F	7	2	F	0	0	T
7.	4	4	F	7	2	F	2	1	T
8.	4	5	F	7	2	F	2	1	T
9.	4	6	F	7	2	F	2	1	T
10.	4	6	F	7	2	F	2	2	T

Figure 2: Example of state of data structure at each time step on sample input

count$[l]$ is not zero, then we inspect *uniq*$[l]$: if it is `true` then we output the pair $(count[l], item[l])$. If *uniq*$[l]$ is `false`, then we do not output an item, since we do not have an accurate count for the item. Else, *count*$[l]$ is zero, so we decrement l and repeat the process. In the basic output routine, we halt as soon as we find a level where $count[l] > 0$; in the "greedy" version, we process every level. The output routine scans the whole data structure, so the time to run the output process is $O(L)$.

Observe that one outcome is that no item is output from the data structure. In our analysis, we will show that for appropriate settings of the parameters r and M, the probability of this outcome is most a constant, $p < 1$. So by sufficiently many repetitions of this data structure with different hash functions h, we can guarantee high probability of returning a sample of the required size.

Example. We consider the following example sequence of insertions of items:

Input: `4, 7, 4, 1, 3, 4, 2, 6, 4, 2`

Suppose these hash to levels in an instance of our data structure as follows:

x	1	2	3	4	5	6	7	8
$l(x)$	1	3	2	1	1	1	2	1

Figure 2 shows the state of the data structure after each update. For each level we indicate whether there is a unique item at that level that can be recovered as the sampled value. Observe that at timesteps 5 and 6, no such item can be found, but at all other times we can recover a sampled item: at time 1 we return (1,`4`); between time 2 and 4 we would return (2,`7`), from time 7 to time 9 we would return item (1,`2`) and lastly at time 10 we return (2,`2`). The greedy output routine would also return item 4 at times 2 and 3.

3.2 Analysis

We show that the Dynamic Inverse Sampling returns uniform samples from the inverse distribution. First, we show that provided a unique item is found at some level then it is drawn uniformly from the set of items with non-zero counts. The main technical result is given in Lemma 6, which shows that there is at least a constant probability that such an item exists after our hashing procedure. Lastly, we show that repeating this procedure several times over will draw a sample (with replacement) of the desired size.

Lemma 5. *If a pair (i, x) is returned from the output procedure, x is selected uniformly from the inverse distribution and $f(x) = i$.*

Proof. Firstly, we observe that if we return a pair (i, x), then indeed $f(x) = i$, since we have counted the number of occurrences of x exactly. To show that x is drawn uniformly, we rely on the universal properties of the hash functions. The strong universality property of $h(x)$ means

$$\mathsf{Pr}[h(x) = a \wedge h(y) = b] = \frac{1}{M^2}$$

Applying this to $h_l(x)$ gives:

$$\mathsf{Pr}[h_l(x) = a \wedge h_l(y) = b] = \mathsf{Pr}[\lfloor \tfrac{h(x)}{r^l M} \rfloor = a \wedge \lfloor \tfrac{h(y)}{r^l M} \rfloor = b]$$
$$= \frac{r^l M * r^l M}{M^2} = r^{2l}$$

Thus, $h_l(x)$ is also strongly universal over r^{-l}. Hence (over choices of h), $\mathsf{Pr}[h_l(x) = 0] = r^l$, and this is independent of x. □

Lemma 6. *Over random choices of h, there is constant probability of the output process returning a pair (i, x).*

Proof. Let D denote the number of distinct items at output time, and let $B_l = 1/r^l$. The function h_l maps onto values $0 \ldots B_l - 1$. From the previous lemma, h_l is 2-universal onto this set. Let X_l denote the number of distinct items observed that satisfy $h_l(x) = 0$. $\mathsf{E}(X_l) = D/B_l$, and $\mathsf{Var}(X_l) \leq \mathsf{E}(X_l)$, using the pairwise-independence of h_l.

Consider the level l such that $\alpha/r \leq D/B_l \leq \alpha/r^2$ for an appropriate scaling constant $\alpha > r$. We analyze the number of items that satisfy $h_l = 0$, and show that there is constant probability that this is small. By the Chebyshev inequality,

$$\mathsf{Pr}[|\mathsf{E}(X_l) - X_l| \geq \mathsf{E}(X_l)] \leq \mathsf{Var}(X_l)/\mathsf{E}(X_l)^2$$
$$\leq 1/\mathsf{E}(X_l) = B_l/D \leq r/\alpha.$$

We use this expression to analyze the probability that X_l is either 1 or 2. The event $|\mathsf{E}(X_l) - X_l| \geq \mathsf{E}(X_l)$ occurs only if $X_l \leq 0$ or if $X_l \geq 2\mathsf{E}(X_l)$. We set $E(X_l) = 3/2$, which fixes $r = \sqrt{2\alpha/3}$. Because $2\mathsf{E}(X_l) \leq 3$, and X_l takes on only integer values, $|\mathsf{E}(X_l) - X_l| < \mathsf{E}(X_l) \Rightarrow X_l \in \{1, 2\}$. Hence, $\mathsf{Pr}[X_l \notin \{1, 2\}] \leq \frac{r}{\alpha} = \sqrt{2/3\alpha}$. This is a constant provided $2/3 < \alpha < 3/2$ (since both this probability and r must be less than 1).

If $X_l = 1$, then there is one item, x, stored at level l or above, and we can easily identify this item and its count. However, if $X_l = 2$, it is possible that both items (say, x and y), are stored at the same level, and we are unable to find the identity of either of them. Assuming $X_l = 2$, we bound the probability that both x and y are stored at the same level. Using the universality of h_l again,

$$\mathsf{Pr}[l(x) \geq l + a | l(x) \geq l] = r^a] \Rightarrow$$
$$\mathsf{Pr}[l(x) = l + a | l(x) \geq l] = r^a - r^{a+1} = r^a(1 - r) \Rightarrow$$
$$\mathsf{Pr}[l(x) = l(y) | l(x), l(y) \geq l] = \sum_{a=0}^{l} (r^a(1 - r))^2 + \frac{1}{r^{2L}},$$

since $\mathsf{Pr}[h(x) = h(y) = 0] = \frac{1}{M^2} = (r^{-l})^2$. Then:

$$(1-r)^2 \sum_{a=0}^{l} (r^2)^a + \frac{1}{r^{2L}} \leq \frac{(1-r)^2}{1-r^2} = \frac{1-r}{1+r}$$

This relies on the fact that the r^{2L} term is dominated by the residue of the infinite sum, which is true if M is chosen sufficiently large. This is achieved provided $M = \Omega(m)$, so we set $M = 2m$.
Using the Markov inequality, $\mathsf{Pr}[X_l \geq 2] \leq \frac{\mathsf{E}(X_l)}{2} \leq \frac{r}{2\alpha}$. So $\mathsf{Pr}[X_l = 2 \wedge l(x) = l(y)] \leq \frac{r(1-r)}{2\alpha(1+r)}$, using (3.2).

The probability, p, that the output process does not return a pair (if there are not one or two items at level l or below, or the two items are mapped to the same level) is

$$p = \mathsf{Pr}[(|X_l - \mathsf{E}(X_l)| > \mathsf{E}(X_l)) \vee (X_l = 2 \wedge l(x) = l(y))]$$
$$= \frac{r}{\alpha} + \frac{r(1-r)}{2\alpha(1+r)} = \frac{r}{2\alpha}\frac{2(1+r)+(1-r)}{1+r} = \frac{1}{3r}\frac{3+r}{1+r}$$

Which follows since we have set $\alpha = 3r^2/2$. Our constraints on r and α are that r and all probabilities should be strictly less than 1. For concreteness, we set $\alpha = 1$, and find $p = \frac{3\sqrt{3}+\sqrt{2}}{2\sqrt{3}+3\sqrt{2}} = 0.8577\ldots$ Thus there is constant probability that the output function will return a pair. □

Having set $r = \sqrt{2/3}$ and $M = 2m$, the size of the data structure is therefore $O(\log_{1/r} M) = O(\log m)$. This gives constant probability at least $1-p$ of extracting a sampled item from the data structure. By keeping $\log(1/\delta)/\log(1/p)$ independent copies of the data structure the failure probability is reduced to arbitrarily small δ. If we require a sample of size k and we keep $k/(1-p)$ copies of the data structure, we recover k items in expectation. In general we need a stronger guarantee on the number of items returned. For small k, we can just keep $k\log(k/\delta)/\log(1/p)$ copies of the data structure: each group of $\log(k/\delta)/\log(1/p)$ guarantees probability of $1-\delta/k$ of returning a sample, so overall, there is probability of $1-\delta$ of getting k samples. Asymptotically, the cost is $O(k\log k)$ copies. For larger k we can give tighter guarantees, using Chernoff bounds:

Lemma 7. *Let $\epsilon = \sqrt{\frac{2\log 1/\delta}{k}}$. If $k \geq 8\log 1/\delta$ and we keep $K = (1+2\epsilon)k/(1-p)$ copies of the data structure, then with probability at least $1-\delta$ we are able to recover at least k samples.*

Based on the above results, our main theorem follows.

Theorem 1. *We can maintain $O(k)$ independent copies of DIS in $O(k\log m)$ space, and guarantee with high probability to return a uniform sample of size k from the inverse distribution. Each insertion operation takes time $O(k)$; extracting the sample takes time $O(k\log m)$.*

4 Dynamic Inverse Sampling: Deletions

In generalizing the data structure to handle deletions, we will perform updates so each deletion precisely counteracts the effect of a previous insertion of the same item, leaving the data structure as if both the deletion and corresponding insertion had never happened. To do this, we make both insertion and deletion *linear* operations on the data structure, which do not inspect the contents of the data structure but rather have the effect of *adding on* or *subtracting off* quantities to various counters, independent of their current values. The correctness of this approach then follows immediately from the commutativity of addition and subtraction.

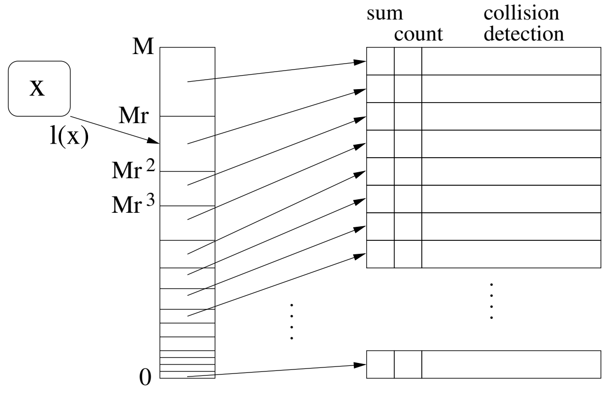


Figure 3: Dynamic Inverse Sampling Data Structure: hash function l maps item x to a level where $count$, sum and collision detection information are updated.

```
Procedure update(x,tt)
Input: Item x, tt=insert/delete
1. $h = h(x)$;
2. if (tt= insert) then
3.   $a = +1$
4.   else $a = -1$;
5. $l = \lceil \log(M/h)/\log(1/r) \rceil$;
6. $sum[l] = sum[l] + x * a$;
7. $count[l] = count[l] + a$;
8. collision-update(x, a);
```

```
Procedure query(gr)
Input: gr flag for greedy output
Output: Samples from $f^{-1}$
1. for $l = L$ downto 0 do
2.   if $count[l] > 0$ then
3.     $x = sum[l]/count[l]$;
4.     if (($\lfloor x \rfloor = x$) and
5.     (collision-test()) then
6.       output $(count[l], x)$;
7.     if (!gr) then break;
```

Figure 4: Pseudo-code for the dynamic inverse sampling

We keep the basic format of the data structure, but make some modifications to how we treat it. Firstly, we replace the $item$ array with an array sum initialized to zero, which will store the sum of item identifiers (which we treat as integers). We also replace $uniq$ with a very small (few bytes in size) "collision detection" data structure, which we will discuss in the next section. The collision detection data structure maintains a distribution of items (which is a subset of the original distribution) under insertions and deletions, and can be queried to find whether there is one distinct item in the distribution or more than one.

Update process. For each insertion of an item x, we compute $l(x)$ as before. We increment $count[l(x)]$, and set $sum[l(x)] \leftarrow sum[l(x)] + x$. We update the collision detection structure with x. For a deletion, we decrement $count[l(x)]$ and set $sum[l(x)] \leftarrow sum[l(x)] - x$, and delete a copy of x from the collision detection structure. Observe that a deletion of x precisely cancels out the effect of a prior insertion of x.

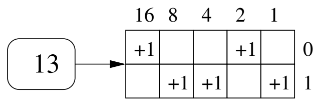


Figure 5: Deterministic collision detection: to insert 13 (represented as a $b = 5$ bit integer) we write $13_2 = 01101$, and so increment $c[1,1], c[2,0], c[3,1], c[4,1], c[5,0]$, corresponding to the 1, 2, 4, 8 and 16 bits.

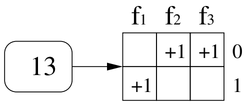


Figure 6: Probabilistic collision detection: to insert 13, with $t = 3$ hash functions, compute $g_1(13) = 1$, $g_2(13) = 0$, $g_3(13) = 0$ and so increment $c[1, 1]$, $c[2, 0]$, $c[3, 0]$.

Output process. In order to output an item from the data structure, we search the data structure in a similar way to before, by searching levels l from L down to 0. If $count[l]$ is not zero, then we try to extract an item from the sample. Suppose that x is the only item that is stored at this level in the data structure. Then x can be recovered as $sum[l]/count[l]$. However, we need to be sure that x is the only item stored at this level. So we make use of the collision detection data structure to tell us (either deterministically or with some probability of error) whether there is only one distinct item stored here, or more than one.

The structure of our data structure is shown in Figure 3, and pseudo-code for insert and query operations is given in Figure 4. The cost per update is now dominated by the cost of updating the collision detection mechanism, since the rest of the update can be completed in constant time.

4.1 Collision Detection

We require a data structure that can be updated in the presence of insertions and deletions of items so that at query time, we can distinguish between the following two events for a given level: (a) a single item occurs at that level one or more times; or (b) there are a mixture of items at that level. One check we can make is to see that $count[l]$ divides $sum[l]$ exactly: if not, then case (b) must hold. But this is not sufficient: if we observe $sum[l] = 20$ and $count[l] = 2$, the input can be any pair of items that sum to 20, not necessarily two copies of item `10`. To avoid outputting items that did not occur in the input we define three approaches, which trade off speed, space and accuracy.

Deterministic. Suppose $|X| = m = 2^b$ so each $x \in X$ is represented as a b bit integer. We can keep $2b$ counters $c[j, k]$ indexed by $j = 1 \dots b$ and $k \in \{0, 1\}$. Every time we see an insertion of x, we increment the counts one count for each value of j: we add one to $c[j, bit(j, x)]$, where $bit(j, x)$ returns the jth bit of the binary representation of x. Symmetrically, for a deletion of x, we decrement the corresponding counts. At output, we can tell whether there is exactly one item or more than one item stored: if and only if there is one item in the bucket, then for all j exactly one of $c[j, 0]$ and $c[j, 1]$ is non-zero. The space required is $O(b)$ counters, and the time to process each update is also $O(b)$. An example update is shown in Figure 5.

Probabilistic. The deterministic approach requires a lot of space for large values of b. We can trade a small probability of error for reduced space. A natural first approach is to use an approximate counter capable of processing insertions and deletions [15]. Applying such an algorithm can distinguish between 1 item and 2 or more items in space $O(\log m \log 1/\delta)$. But this space cost is still large.

Instead, a similar method to the deterministic approach uses hashing to give a probabilistic test for collisions. We draw t hash functions, $g_1 \dots g_t$ which map items uniformly onto $\{0, 1\}$, and use a set of $t \times 2$ counters $c[j, k]$. For every insertion, we increment $c[j, g_j(x)]$, and decrement the same counter for a deletion. We apply the same test as in the deterministic case. If for any j, $c[j, 0] \neq 0$ and $c[j, 1] \neq 0$, then there is more than one distinct item in the bucket. The probability of wrongly declaring a single distinct item in the bucket is at most 2^{-t}. The space used is $O(t)$ counters, and it takes $O(t)$ time per update. An example update is shown in Figure 6.

Heuristic. The previous method may still consume too much space. A simple heuristic gives faster updates and few errors in practice (we make no formal claims about the error probability here). We compute q new hash functions $g_j[x]$ mapping items x onto $0 \dots m$ and track the summation of $g(x)$ as $sumg[j, l(x)]$. For every insertion of an item, we add $g(x)$ to $sumg[j, l(x)]$, and for every deletion, we subtract $g(x)$ from $sumg[j, l(x)]$. At query time, we extract x from the bucket as $sum[l]/count[l]$. If x is the only distinct item in the bucket, then $sumg[j, l] = g_j(x) * count[l]$ for all j. We can check this condition and reject if it is not satisfied by any hash. The space required for the heuristic collision detection mechanism is $O(q)$ counters per level, and $O(q)$ time per update.

In all three cases, the collision detection data structures are updated by summing positive and negative values, without examining the contents of the counters. Therefore arbitrary combinations of insertions and deletions can be handled by them. Pseudo-code for the three different collision detection methods is shown in Figure 7. The analysis of Lemma 6 can be applied again, leading to:

Theorem 2. *Using $O(k \log m)$ space, we can maintain $O(k)$ dynamic inverse sampling data structures to process a sequence of insertions and deletions and that guarantee with high probability to return a uniform sample from the inverse distribution of size k. Each update operation takes time $O(k)$; extracting the sample takes time $O(k \log m)$.*

4.2 Extensions

We have discussed insertion and deletion of single items. We now observe other ways in which our data structures can be manipulated:

```
Procedure deterministic-update($x$, $val$)
Input: Item $x$, $val$=+1/-1 for insert/delete
1. for $j = 1$ to $b$ do
2.     $bit = x\&1$
3.     $c[j, bit] = c[j, bit] + val$;
4.     $x = x \gg 1$;

Procedure deterministic-collision-test()
Output: $true$ if no collision else $false$
1. for $j = 1$ to $b$ do
2.     if $c[j, 0] \neq 0$ and $c[j, 1] \neq 0$ then
3.         return $false$;
4. return $true$;
```

```
Procedure probabilistic-update($x$, $val$)
Input: Item $x$, $val$ =+1/-1 for insert/delete
1. for $j = 1$ to $t$ do
2.     $bit = g_j(x)$;
3.     $c[j, bit] = c[j, bit] + val$;

Procedure probabilistic-collision-test()
Output: $true$ if no collision, else $false$
1. for $j = 1$ to $t$ do
2.     if $c[j, 0] \neq 0$ and $c[j, 1] \neq 0$ then
3.         return $false$;
4. return $true$;
```

```
Procedure heuristic-update($x$, $val$)
Input: Item $x$, $val$=+1/-1 for insert/delete
1. for $j = 1$ to $q$ do
2.     $sumg[j] = sumg[j] + val * g_j(x)$;

Procedure heuristic-collision-test()
Output: $true$ if no collision else $false$
1. for $j = 1$ to $q$ do
2.     if $g_j(sum/count) * count$
           $\neq sumg[j]$ then
3.         return $false$;
4. return $true$;
```

Figure 7: Pseudo-code for the different collision detection mechanisms.

Sliding Window. In many settings, we only want to draw a sample from a recent history of an (insertions-only) stream. The sliding window model [12] specifies that only the most recent W updates (or updates that occurred within W time units) should be considered, for some fixed value of W. When W is too large to buffer the most recent W updates, we can apply a variation of our technique. For each update x, we overwrite the current item stored at level $l(x)$. We can modify the deterministic and probabilistic collision detection mechanisms so that instead of incrementing a counter, we overwrite the current contents with the *timestamp* of the new item x. At output time, we check the collision detection mechanism to see if there has been any collision within the last W time units: if there is one unique item, then for each pair of counters, exactly one will store a timestamp from within the last W time units. Hence, we can use this modified version of the data structure to draw an item x uniformly from the inverse distribution over the sliding window. However, this does not give us a value for i; and to give the exact value of i is impossible without using $\Omega(W)$ space. Instead, we can use the counting techniques of [12] to approximate the value of i for x, which gives a doubly-approximate answer to queries on the inverse distribution.

Multiple insertions or deletions. We have considered the case where a single item arrives or departs at a time. We can easily generalize this to handle arbitrarily many copies of a single item by appropriate scaling of the counts that we add or subtract.

Fractional and negative item counts. Our analysis does not require the counts of items to be positive integers, hence we can allow counts to become negative and fractional. The interpretation of the sample values returned is that these are selected uniformly from the set of items whose count is non-zero.

Unioning and Differencing of summaries. We can combine two summaries that were created with the same parameters and hash functions by summing the values in their corresponding counters. The result is exactly identical to the result if all updates had been processed by a single summary structure. Hence, the algorithm can easily be carried out in a distributed fashion over a variety of streams, and then the summaries merged to allow investigation of the inverse distribution of the union of all the streams. Similarly, we can compute the difference of two summaries by subtracting corresponding counters; scale all counts by a scalar value; and so on.

5 Inverse Distribution Queries

We now show how to use a sample drawn by the Dynamic Inverse Sampling algorithm to answer queries on the inverse distribution.

Theorem 3. *Given a sample from the inverse distribution of size $O(\frac{1}{\epsilon^2}\log\frac{1}{\delta})$, we can answer Inverse Point Queries with additive error less than ϵ with probability at least $1-\delta$.*

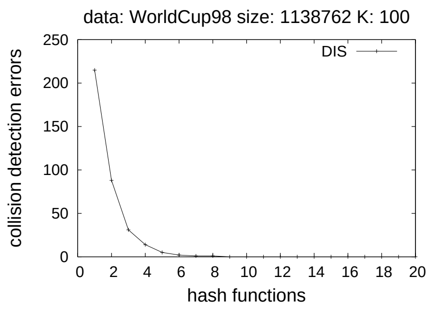
Proof. Let S be the sample drawn from by Dynamic Inverse Sampling, which is a multiset of pairs. We approximate $f^{-1}(i)$ with $\frac{|\{(i,x)\in S\}|}{|S|}$. To analyze this estimator, we set up an indicator variable for each sample in S. Let $Y_j = 1$ if the jth sample in S is a pair (i, x), and $Y_j = 0$ if the jth sample is a pair (i', x') for $i' \neq i$. Since each sample is drawn uniformly, $\Pr[Y_j = 1] = \{x|f(x) = i\}/D = f^{-1}(i)$. So the estimate is correct in expectation. By applying the Hoeffding inequality to $\sum_j Y_j/|S|$, we get $\Pr[|\sum_j Y_j/|S| - f^{-1}(i)| \leq \epsilon] \geq 1-\delta$, as required. □

Corollary 1. *Given a sample from the inverse distribution of size $O(\frac{1}{\epsilon^2}\log\frac{1}{\delta})$, we can answer Inverse Range Queries and queries to the cumulative inverse distribution with additive error less than ϵ with probability at least $1-\delta$.*

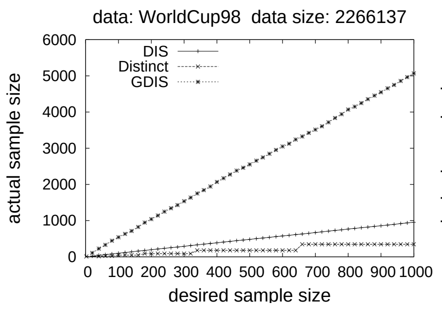
Proof. For an inverse range query $[j, k]$, our estimator is $\frac{|\{(i,x)\in S, j\leq i\leq k\}|}{|S|}$. A similar proof to the above shows that this estimator is correct in expectation, and within ϵ with probability at least $1-\delta$. Queries to the cumulative inverse distribution can be reduced to open-ended inverse range queries $[i, \infty]$, and so the same bounds apply. □

Corollary 2. *Given a sample from the inverse distribution of size $O(\frac{1}{\epsilon^2}\log\frac{1}{\delta})$, Inverse heavy hitters can be answered with additive error ϵ with probability at least $1-\delta$.*

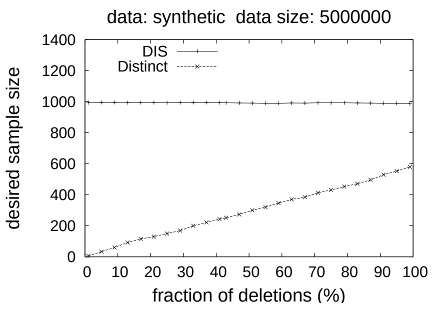
Proof. In order to answer inverse heavy hitter queries, we compute our estimate of $f^{-1}(i)$ for each i that is in the sample, and output those for which $\frac{|\{(i,x)\in S\}|}{|S|} \geq \phi$. By the



(a)



(b)



(c)

Figure 8: (a) Evaluating number of hash functions required for the probabilistic collision detection. (b) Number of samples returned by the different inverse sampling methods as a function of desired sample size. (c) Number of samples returned by the different inverse sampling methods as a function of deletion frequency.

above theorem, for each i that is output, there is ϵ error in the estimate with probability $1-\delta$, and so we guarantee (with this probability) that every i that is output satisfies $f^{-1}(i) > \phi - \epsilon$. Similarly, since every i that does not appear in the sample is approximated by $f^{-1}(i) = 0$, we conclude that with the same probability, every item with $f^{-1}(i) > \phi + \epsilon$ is output. □

Corollary 3. *Given a sample from the inverse distribution of size $O(\frac{1}{\epsilon^2}\log\frac{1}{\delta})$, Inverse quantiles queries can be answered with additive error ϵ with probability at least $1-\delta$.*

Proof. For Inverse Quantile Queries, we compute the estimate of $F^{-1}(i)$ for all i in the sample. Observe that this estimate gives a decreasing function as i increases. We output the (unique) i such that the estimate of $F^{-1}(i-1) > \phi$ and $F^{-1}(i) \leq \phi$. By the guarantees on cumulative inverse distribution queries, we have (with probability $1-\delta$) the i that is output has $\phi - \epsilon \leq F^{-1}(i) \leq \phi + \epsilon$. □

6 Experimental Study

We implemented our Dynamic Inverse Sampling algorithm, and evaluated it on large sets of network data drawn from HTTP log files from the 1998 World Cup Web Site (stored in the Internet Traffic Archive [24]), as well as on a large synthetic data set of randomly generated distinct values. We took client ID and size attributes from the log data totaling several million records. Our synthetic data set contains 5 million randomly generated distinct items. To give a data set with a large number of deletions, we built a dynamic transaction set by inserting all the records and then deleting a fraction of these. Since one cannot predict which records will survive the deletions, it gives a challenging test for our methods.

For comparison, we implemented the Distinct Sampling method [19, 21] augmented to handle deletions since this can be used to draw a sample from the inverse distribution under insertions only streams (see the discussion in Section 7). The algorithms were implemented in C and were run on a 2.4GHz processor desktop computer.

Collision Detection Experiments. We compared the different collision detection mechanisms for the Dynamic Inverse Sampling (DIS). We ran the algorithm over a data set consisting of insertions only, and counted the number of times that the approximate methods reported no collision (at any level in the data structure), when the deterministic method (correctly) indicated that there was a collision.

Collision Detection	**Hash Functions**	**Space Factor**	**Time Cost**
None	—	—	96s
Deterministic	—	32	132s
Heuristic	$q=1$	1	119s
Heuristic	$q=2$	2	140s
Heuristic	$q=3$	3	162s
Probabilistic	$t=5$	5	165s
Probabilistic	$t=10$	10	225s

Table 1: Timing results and space/time tradeoff for different collision detection methods. 'Space factor' denotes relative space cost of each method.

We tested the probabilistic collision detection mechanism by gradually increasing the number of hash functions from 1 to 40. The results are shown in Figure 8 (a); it can be observed that the total number of errors over all levels in all data structures drops to 0 when we use 9 or more hash functions. The heuristic collision detection mechanism was run with the number of hash functions ranging from 1 to 5. With one hash function, there were 3 collision detection errors on a dataset of 1.3 million records. There were no collision detection errors with two or more hash functions.

We compared the time cost of all three methods to process a total of 260 million updates to the data structures. Timing results are showing in Table 1. They show that our method is capable of processing several million updates per second (for comparison, our implementation of Distinct sampling was faster still, processing 9 million items/second). We see that the Deterministic method is quite fast, since it requires no additional hash function computation. But it still requires space for $\log m$ counters. With two hash functions, an undetected collision under the heuristic method is very unlikely, and this requires only two additional counters per level, plus two hash functions per copy of the DIS structure. This gives a good trade-off of time against space used. For the remainder of our experiments, we worked with the deterministic method only, knowing that for suitable settings of q and t we would get identical results using the heuristic or probabilistic collision detection methods.

Returned Sample Size. We compared the size of sample returned by the different methods over the datasets we used in our experiments. We ran our experiments on the client ID attribute of the HTTP log data. Each network dataset generated a sequence of insertion and deletion transactions, with over 3 million operations in total for each dataset. We measured the actual sample size returned by the algorithms after processing all the insertions and deletions, when 50% of the inserted records were deleted. The results for other network datasets were similar; we show a representative plot in Figure 8 (b). For the desired sample size of 100, the distinct sampling ("Distinct") technique returned a sample of about 45% of the desired size. When the desired sample size was increased to 1000, the size of the sample was only 30% of the desired size. These results support our claim that this approach has difficulty with handling a large number of delete operations.

The Dynamic Inverse Sampling algorithm (DIS) returned a sample of almost 100% of the desired size for all sample sizes (for instance, for $k = 1000$ it returned 998 samples when there were 1% deletions, 981 samples at 10%, 970 for 20% and 955 for 50%) which indicates that in practice the probability of obtaining at least one sampled record from each dynamic inverse sampling structure is close to 1. Using the greedy output routine (GDIS) which extracts all possible sample records from every dynamic structure, returned approximately five items from each data structure. Both variations of the Dynamic Inverse Sampling method are not affected by the order and amount of insert and delete operations.

Next we investigate the dependency between the size of the sample returned by the methods and the fraction of deletions in the data set. We ran our experiment on the synthetic data set of distinct items, when the desired sample size is 1000. The results are shown in Figure 8 (c). For a data set with a large number of deletions, the distinct sampling technique performs poorly. When 80% of the inserted records were deleted from the sampling structure, the sample size was about 12% of the desired sample size. As the number of deletions approaches the number of insertions the sample size returned by the distinct sampling algorithm decreases linearly. When the number of deleted records was increased to 99% of the number of insertions, the resulting size of the sample was less than 1% of the desired sample size. The Dynamic Inverse Sampling algorithm was stable under any number of deletions and returned a sample (with replacement) of size almost 100% of the desired size.

Sample Quality. Lastly, we measured how well the obtained sample represented the sampled dataset. To calculate this estimate, we posed a series of inverse range queries $F^{-1}(i)$ on the samples (to compute the fraction of records with size greater than i), and compared it to the exact value of this query computed offline. Figure 9 shows experiments on two different network datasets for $i = 1000$, the first on a linear scale and the second on a log scale. In Figure 9 (a), we see that both the regular and the greedy output procedure give very low error for small sample sizes — in particular, the greedy procedure achieves close to zero error for as sample size as small as 15. This shows that this output function seems to do very well in practice. In contrast, for very small sample sizes, Distinct sampling is unable to return any sample at all. In Figure 9 (b), we see that GDIS consistently outperforms Distinct sampling, by up to an order of magnitude, making it the method of choice.

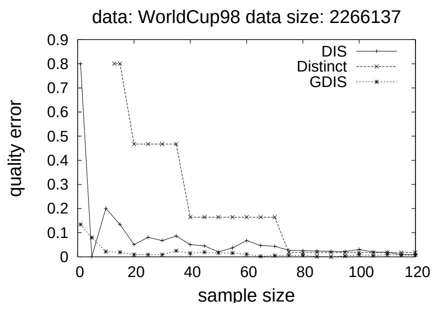
Another set of experiments was performed on the network data set with over 4 million records by posing a series of inverse quantile queries on the samples using the client ID attribute of the records. In particular, we estimated the median (to find i that $F^{-1}(i-1) > 0.5, F^{-1}(i) \leq 0.5$) of the inverse distribution using the resulting sample, and measured how far the true position of the returned item i was from 0.5. Figure 9 (c)shows the results of the experiment ("quality error" is computed as $2|F^{-1}(i) - 0.5|$). We can see that for small desired sample sizes (under 100), the distinct sampling algorithm does not have a large enough sample to give any results. The algorithm's error of median estimation becomes sufficiently small only when the desired sample size is about 350 or higher. In contrast, both versions of the dynamic sampling algorithm are much more accurate in their estimation of the median value even for small sample sizes.

We ran a variety of other experiments to test the quality of our methods to approximate functions over the inverse distribution. We omit detailed analysis for brevity, but in all cases we saw that our dynamic inverse sampling methods were able to give high quality estimates of the queries of interest, in the presences of streams consisting of both insertions and deletions.

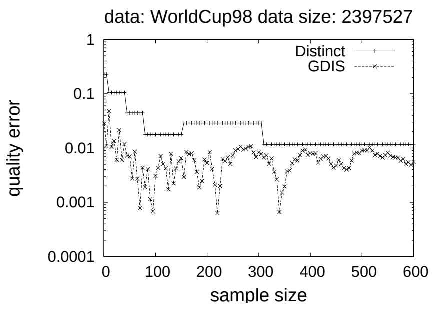
7 Previous Work

The research community has developed a rich literature on applications of random sampling algorithms in databases and data streams. One of the most common and well studied applications of sampling in large data warehouse environments is to provide fast approximate answers to complex aggregation queries based on statistical summaries which are created and maintained using various sampling techniques [29, 20, 22]. Random sampling is a standard technique for constructing approximate summary statistics, such as histograms, for query optimization and query planning purposes [22, 8]. Random sampling is widely used for distinct-values estimators [19, 21, 6] which play an important part in network monitoring and online aggregation systems.

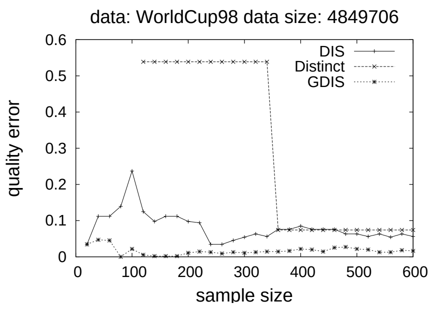
In today's database systems random sampling is routinely used for a variety of purposes. Microsoft SQL Server 2000 uses sampling to build and maintain histograms which provide various statistics for the query optimizer to choose the most efficient plan for retrieving and



(a)



(b)



(c)

Figure 9: (a) Accuracy of sampling methods on inverse range query (linear scale) (b) Accuracy of sampling methods on inverse range query (logarithmic scale) (c) Accuracy of sampling methods on inverse quantiles

Algorithm	Type	Method	Deletions	Random
Reservoir Sampling [31]	Fwd	WoR	No	Full
Backing Sample [22]	Fwd	WoR	Few	Full
Weighted Sampling [7]	Fwd	WR	No	Full
Concise Sampling [20]	Fwd	CF	No	Full
Count Sampling [20]	Fwd	CF	Few	Full
Minwise-hashing [13]	Inv	WR	No	$\frac{1}{\epsilon}$-wise
Distinct Sampling [19, 21]	Inv	CF	Few	Pairwise
Dynamic Inverse Sampling (here)	Inv	CF, WR	**Yes**	Pairwise

Figure 10: Key features of existing sampling methods.

processing data. Statistics are maintained by re-sampling column values whenever substantial update activity has occurred[8]. The Oracle database system uses "dynamic sampling" to improve server performance by determining more accurate selectivity and cardinality estimates, which allow the optimizer to produce better performance plans. Oracle determines at compile time whether a query would benefit from dynamic sampling. If so, a recursive SQL statement is issued to scan a small random sample of the table's blocks to estimate predicate selectivities. [9] Thus, while commercial DBMSs need dynamic sampling, they resort to rescanning or re-sampling from stored databases, and therefore, do not work in one pass.

Our focus is on providing a uniform sample of the inverse distribution which can be used to approximate queries on the inverse distribution. Despite the many works on sampling in databases, there is very little work that directly applies to inverse distributions. Following [7] sampling methods broadly fall into three categories: sampling With Replacement (WR), Without Replacement (WoR), and coin flipping (CF)[10]. All the sampling methods we consider can be classified with one or more of these labels. In addition, two other factors are relevant to our focus:

Processing of Deletions. Existing methods either do not handle deletions (that is, it is unclear how to process a deletion and still retain a uniform sample), or can handle only a limited number of deletions: the result is still a uniform sample, but in the presences of many deletions, the size of the sample shrinks to zero.

Amount of Randomness Required. Early works assume "truly random" numbers, but more recent work considers what strength of randomness is needed. k-wise random hash functions guarantee that any k items collide under the hash function with independent probability [27], and such functions are efficient to compute and store for small k (eg pairwise hash functions with $k = 2$ [4]).

We summarize the relevant sampling techniques that can draw a sample from a stream of updates in Figure 10. We classify them on which distribution they sample from — the forward distribution (fwd) or the inverse distribution (inv); deletion handling; and the randomness required. Although many algorithms maintain a uniform random sample of data items of the forward distribution in the presence of insertions, none handle a significant number of deletions to the data set while guaranteeing a sample of a certain size.

There is a limited prior work that relates to inverse distributions. Some existing techniques can be used to create a sample from the inverse distribution on insert-only streams. The Distinct Sampling technique of Gibbons *et al.* [19, 21] draws a sample based on a coin-tossing procedure using a pairwise-independent hash function on item values. This effectively draws a uniform sample from the inverse distribution, which we can use to answer queries on the inverse distribution, as discussed in Section 5. As with all other existing sampling methods, deletions can deplete the sample, and it is not possible to recover a sufficiently large sample—in our streaming scenario, backtracking on the past data for a rescan is simply not possible.

An alternative approach is to make use of Min-wise hash functions, which sample uniformly from the set of items seen. These were applied in [13] but deletions were not considered; one can apply a "best effort" approach by decrementing the counts of deleted items in the sample until these fall to zero—but it is not possible to give worst case bounds on the size of the sample stored. Work on estimating the cardinality of set expressions over data streams [17] uses a similar data structure to the one we propose here, and with some amount of modifications can be used to draw a sample from the inverse distribution. However, this is not the goal of that work, and the given analysis requires hash

[8]`http://msdn.microsoft.com/library/en-us/dnsql2k/html/statquery.asp`

[9]`http://www.dba-oracle.com/art_dbazine_oracle10g_dynamic_sampling_hint.htm`

[10]Where the sample size is not fixed but rather each item is chosen to be in the sample with some probability p.

functions that are at least $\log 1/\epsilon$-wise independent. Here, we show that for the purpose of sampling from the inverse distribution, a simpler structure is sufficient, with only pair-wise independence. Similar results have been recently obtained by Indyk, and Frahling and Sohler [16] which address problems in geometric data streams.

8 Concluding Remarks

Many of the existing methods for summarizing and mining data streams focus on the forward distribution. In contrast, we formulate summarization and mining problems on the inverse distribution. We introduced the notion of the inverse distribution for massive data streams, and gave algorithms that draw uniform samples from the inverse distribution when the data stream consists of insertions only, as well as insertions and deletions. With a sample of size $O(\frac{1}{\epsilon^2})$, we can answer a variety of summarization and mining tasks on the inverse distribution up to an additive approximation of ϵ. These are the first such results known for managing inverse distributions on data streams. In our experiments we saw that the methods we propose can process massive data streams of updates at very high rates, and answer queries on the inverse distribution with high accuracy.

Summarizing and mining the inverse distribution on the stream provides insights on different aspects of the data stream than is revealed by working with the summaries of the forward distribution. It remains open to answer more complex queries over the inverse distribution, such as computing frequency moments or detecting anomalies. A fundamental question that arises is to design algorithms to maintain a uniform sample of the *forward* distribution under *both* insertions and deletions over data streams or show that this is impossible — as noted in the previous section, no existing algorithms guarantee to return a non-empty sample in this setting.

References

[1] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. Aurora: a data stream management system. In *ACM SIGMOD*, 2003.

[2] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *STOC*, 1996.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.

[4] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *JCSS* , 18(2):143–154, 1979.

[5] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: continuous dataflow processing. In *ACM SIGMOD*, 2003.

[6] M. Charikar, S. Chaudhuri, R. Motwani, and V. R. Narasayya. Towards estimation error guarantees for distinct values. In *PODS*, 2000.

[7] S. Chaudhuri, R. Motwani, and N. Narasayya. On random sampling over joins. In *ACM SIGMOD*, 1999.

[8] S. Chaudhuri, R. Motwani, and V. Narasayya. Random sampling for histogram construction: How much is enough? In *ACM SIGMOD*, 1998.

[9] G. Cormode, F. Korn, S. Muthukrishnan, T. Johnson, O. Spatscheck, and D. Srivastava. Holistic UDAFs at streaming speeds. In *ACM SIGMOD* , 2004.

[10] G. Cormode and S. Muthukrishnan. What's hot and what's not: Tracking most frequent items dynamically. In *ACM PODS*, 2003.

[11] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *ACM SIGMOD*, 2003.

[12] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *SODA* , 2002.

[13] M. Datar and S. Muthukrishnan. Estimating rarity and similarity over data stream windows. In *ESA*, 2002.

[14] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *ACM SIGCOMM*, 2002.

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

[16] G. Frahling and C. Sohler. Coresets in dynamic geometric data streams. In *STOC*, May 2005.

[17] S. Ganguly, M. Garofalakis, and R. Rastogi. Processing set expressions over continuous update streams. In *ACM SIGMOD*, 2003.

[18] M. Garofalakis, J. Gehrke, and R. Rastogi. Querying and mining data streams: You only get one look. In *ACM SIGMOD*, 2002.

[19] P. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB*, 2001.

[20] P. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *ACM SIGMOD*, 1998.

[21] P. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *SPAA*, 2001.

[22] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. In *VLDB*, 1997.

[23] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1997.

[24] Internet traffic archive. `http://ita.ee.lbl.gov/`.

[25] D. Madigan. DIMACS working group on monitoring message streams. `http://stat.rutgers.edu/~madigan/mms/`, 2003.

[26] G.S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB*, 2002.

[27] R. Motwani and P. Raghavan. *Randomized Algorithms*. CUP, 1995.

[28] S. Muthukrishnan. Data streams: Algorithms and applications. In *SODA*, 2003.

[29] F. Olken. *Random Sampling from Databases*. PhD thesis, Berkeley, 1997.

[30] Stanford stream data manager. `http://www-db.stanford.edu/stream/sqr`.

[31] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.