PART ONE

FUNDAMENTAL SUMMARY TECHNIQUES

2

Summaries for Sets

This chapter studies some fundamental and commonly used summaries for sets. The input consists of items drawn from a universe U, which define the set A to be summarized. By definition, a set does not contain duplicated items, but the input to the summary may or may not contain duplicates. Some summaries are able to remove duplicates automatically, while others treat each item in the input as distinct from all others. This will be pointed out explicitly when each summary is described.

The summaries described in this chapter address the following tasks:

- Approximately large quantities with few bits: the MorrisCounter (Section 2.1).
- Maintaining a random sample of unweighted items: the RandomSample (Section 2.2).
- Maintaining random samples where items in the set also have (fixed) weights: the WeightedRandomSample and PrioritySample summaries (Section 2.3 and Section 2.4).
- Estimating the number of distinct items in a collection: the KMV and HLL summaries (Section 2.5 and Section 2.6).
- Approximately representing the members of a set in a compact format: the BloomFilter (Section 2.7)

2.1 Morris Approximate Counter

Brief Summary. The very first question one could ask about a set is its cardinality. When no duplicates are present in the input, counting the items in the set A can be trivially done with a counter of $\log |A|$ bits. The

MorrisCounter summary provides an approximate counter using even fewer bits. Instead of increasing the counter for every item, a random process determines when to increase the counter, as a function of the current state of the counter.

Note that the MorrisCounter cannot deal with duplicates in the input; please use the summaries described in Section 2.5 and 2.6 if this is the case.

Algorithm 2.1: MorrisCounter: UPDATE ()

```
1 Pick y uniform over [0, 1];
2 if y < b^{-c} then c ← c + 1;
```

Algorithm 2.2: MorrisCounter: QUERY ()

```
1 return (b^c − 1)/(b − 1);
```

Operations on the summary. The MorrisCounter summary is simply a counter c, with a parameter $1 < b \leq 2$, that can be thought of as the (number) base over which the counter operates. The INITIALIZE (b) operation sets the counter to 0 and locks in the value of b. The UPDATE operation updates the counter when a new item is added to A. Specifically, UPDATE increases c by 1 with probability b^{-c}, and leaves it unchanged otherwise. Informally, we expect b items for this counter to go from 1 to 2, then a further b^2 to reach 3, b^3 more to reach 4, and so on in a geometric progression. This justifies the fact that the QUERY operation shown in Algorithm 2.2 provides an estimated count as $\frac{b^c-1}{b-1}$.

The analysis of this summary indicates that setting $b = 1 + 2\varepsilon^2\delta$ is sufficient to provide ε-relative accuracy of counts with probability at least $1 - \delta$. When $|A| = n$, this suggest that the counter c should go up to $\log((b-1)n)/\log b = O(\frac{1}{\varepsilon^2\delta}\log\varepsilon^2\delta n)$, and therefore requires $O(\log\frac{1}{\varepsilon} + \log\frac{1}{\delta} + \log\log\varepsilon^2\delta n)$ bits. This can be much more compact than the $\lceil\log n + 1\rceil$ bits required to keep an exact count of up to n items when n is very large indeed.

Algorithm 2.3: MorrisCounter: MERGE (c_a, c_b)

```
1 α = min(c_a, c_b), β = max(c_a, c_b) ;
2 for j ← 0 to α − 1 do
3 |   Pick y uniform over [0, 1];
4 |   if y < b^{j−β} then β ← β + 1;
5 return β;
```

To MERGE together two MorrisCounter summaries that used the same

base b, we can pick the larger of the two counters as the primary, and use the smaller to determine whether to further increase it. Let β denote the current count of the larger counter, and α denote the smaller counter. We perform α tests to determine whether to increment β. We increment β with probability $b^{i-\beta}$ for i from 0 to $\alpha-1$. This corresponds to stepping through the α items that prompted increments in the smaller counter. Algorithm 2.3 details the MERGE algorithm, incrementing the larger counter based on the appropriate conditional probabilities implied by the smaller counter.

Example. We set $b = 2$, and consider a stream of 9 items. The below table shows a sample state of the counter after each of these.

Timestep	1	2	3	4	5	6	7	8	9
c	1	1	1	2	2	3	3	3	3

After 9 items, the counter records 3, which corresponds to an estimate of 7.

Given two MorrisCounter summaries using base 2 which both contain 3, we MERGE by starting with a counter of 3. We first increment this with probability $2^{0-3} = 1/8$; say this test does not pass. Then we increment with probability $2^{1-3} = 1/4$. Suppose in this example, the test passes, so we now have a counter of 4. Finally, we increment with probability $2^{2-4} = 1/4$ (note, we use the current value of the counter). In our example, this test does not pass, so we conclude with a merged counter with count of 4, representing an estimate of 15.

Implementation Issues. To draw a random value with probability b^{-c}, it is convenient to choose b based on a power of two, such as $b = 1 + 1/(2^\ell - 1)$ for some integer ℓ. The test with probability $1/b$ passes with probability $(2^\ell - 1)/2^\ell$, which can be done by generating ℓ uniform random bits — for example, if not all of the bits are zero. The test with probability b^{-c} passes if c instances of the previous test pass. This requires a lot of randomness, and so can be approximated using a single floating-point random value tested against b^{-c}.

Further Discussion. The analysis of the expectation of the estimate $\frac{b^c-1}{b-1}$ under a sequence of UPDATE operations essentially shows that at each step, the expected change in the estimated count is one. Formally, let X_n denote the output of the counter after n UPDATE

operations, and let C_n denote the value of the stored count c. Then, inductively,

$$\begin{aligned}
\mathsf{E}[X_n] &= \sum_c \Pr[C_n = c]\frac{b^c - 1}{b-1} \\
&= \sum_c \left(\Pr[C_{n-1} = c-1]b^{-c} + \Pr[C_{n-1} = c](1 - b^{-c})\right)\frac{b^c - 1}{b-1} \\
&= \sum_c \Pr[C_{n-1} = c]\left(b^{-c}\frac{b^{c+1} - 1}{b-1} + (1 - b^{-c})\frac{b^c - 1}{b-1}\right) \quad \text{(regrouping the terms)} \\
&= \sum_c \Pr[C_{n-1} = c]\frac{b^c - 1}{b-1} + 1 \\
&= \mathsf{E}[X_{n-1}] + 1.
\end{aligned}$$

Therefore, since $X_0 = 0$, we have that $\mathsf{E}[X_n] = n$. For the variance, we first define $Y_n = X_n + \frac{1}{b-1}$ and compute

$$\begin{aligned}
\mathsf{E}[Y_n^2] &= \sum_c \Pr[C_n = c]\left(\frac{b^c}{b-1}\right)^2 \\
&= \sum_c \Pr[C_{n-1} = c]\left(b^{-c}(\frac{b^{c+1}}{b-1})^2 + (1 - b^{-c})(\frac{b^c}{b-1})^2\right) \\
&= \sum_c \Pr[C_{n-1} = c]\left((\frac{b^c}{b-1})^2 + \frac{b^{-c}}{(b-1)^2}((b^{c+1})^2 - (b^c)^2)\right) \quad \text{(regrouping)} \\
&= \mathsf{E}[Y_{n-1}^2] + \sum_c \Pr[C_{n-1} = c]\frac{b^{-c}}{(b-1)^2}(b^{c+1} - b^c)(b^{c+1} + b^c) \\
&= \mathsf{E}[Y_{n-1}^2] + (b+1)\sum_c \Pr[C_{n-1} = c]\frac{b^c}{b-1} \\
&= \mathsf{E}[Y_{n-1}^2] + (b+1)\mathsf{E}[Y_{n-1}] \\
&= \mathsf{E}[Y_{n-1}^2] + (b+1)(n-1) \\
&= \mathsf{E}[Y_0^2] + (b+1)\sum_{i=0}^{n-1} i.
\end{aligned}$$

Thus, since $Y_0 = \frac{1}{b-1}$, we have

$$\mathsf{E}[X_n^2] = \mathsf{E}[\left(Y_n - \frac{1}{b-1}\right)^2] \le \mathsf{E}[Y_n^2 - Y_0^2] = \frac{1}{2}(b+1)n(n-1),$$

and so

$$\mathsf{Var}[X_n] \leq \frac{1}{2}(b+1)n^2 - n^2 = \frac{1}{2}(b-1)n^2.$$

Via the Chebyshev inequality, we then have

$$\Pr[|X_n - n| \geq \varepsilon n] \leq \frac{1}{2}(b-1)n^2/\varepsilon^2 n^2 = \frac{b-1}{2\varepsilon^2}.$$

Therefore, if we choose $b \leq 1 + 2\varepsilon^2\delta$, we have the desired bound. Alternately, we can take $b \leq 1+\varepsilon^2$ to have this hold with probability at most $\frac{1}{2}$. Then taking the median of $O(\log 1/\delta)$ estimates will reduce the error probability to δ, via the Chernoff bounds argument of Section 1.4.1.

To merge two MorrisCounter summaries, it is helpful to think of the random decision of whether to update the counter as being determined by a random variable Y which is uniform over the range $[0, 1]$. The test at a step with probability b^{-c} is passed when $Y < b^{-c}$, i.e., $\Pr[Y < b^{-c}] = b^{-c}$. Associate the ith update with such a random variable, Y_i. Then fix these choices over the series of updates, that is, imagine that there is a fixed y_i value associated with each update. Now imagine taking the sequence of updates associated with the second (smaller) counter, and applying them as updates to the first (larger) counter, with this now fixed set of y_i values. The result is an updated counter which reflects the full set of updates, and has the correct distribution.

We now argue that there is enough information in the smaller counter that describes the set of y_i values observed to exactly simulate this process, without explicit access to them. First, consider those UPDATE events which did not change the value of the smaller counter. These must have been associated with y_i values greater than b^{-c}, where c is the value of the larger counter: since the smaller counter ended with a value at most c, these updates could not have had such a small y_i value, else they would have changed the counter. This leaves the UPDATE events that caused the smaller counter to increase from j to $j+1$. Here, the corresponding Y value must have been less than b^{-j}. Beyond this, we have no information. Since the Y random variable is uniform, conditioned on the fact that $y_i < b^{-j}$, Y_i is uniform in the range $[0, b^{-j}]$. Therefore, the probability that Y_i is below b^{-c} is $b^{-c}/b^{-j} = b^{j-c}$. It is acceptable to

make this randomized test at the time of the merge, by invoking the principle of deferred decisions.

History and Background. The notion of the approximate counter was first introduced by Morris in 1977 [182]. It is sometimes regarded as the first non-trivial streaming algorithm. A thorough analysis was presented by Flajolet in 1985 [100]. The analysis presented here follows that of Gronmeier and Sauerhoff [121]. The generalization to addition of approximate counters does not appear to have been explicitly considered before. The summary is considered a basic tool in handling very large data volumes, and can be applied in any scenario where there are a large number of statistics to maintain, but some inaccuracy in each count can be tolerated — for example, in maintaining counts of many different combinations of events that will instantiate a machine learned model. An early application of the MorrisCounter was to count the frequencies of combinations of letters in large collections of text with few bits per counter. Such frequency counts can be used to give more effective data compression models, even with approximate counts. Some additional historical notes and applications are described by Lumbroso [168].

2.2 Random Sampling

Brief Summary. A random sample is a basic summary for a set that can be used for a variety of purposes. Formally, a RandomSample (without replacement) of size s of a set A is a subset of s items from A (assuming $|A| \geq s$, and treating all members of A as distinct) such that every subset of s items of A has the same probability of being chosen as the sample.

The random sampling algorithms described in this section cannot handle duplicates in the input, i.e., if the same item appears multiple times, they will be treated as distinct items, and they will all get the chance to be sampled. If the multiplicities should not matter, please see *distinct sampling* in Section 3.9.

Operations on the summary. Let S be the random sample. In order to UPDATE and MERGE random samples, we also need to store n, the cardinality of the underlying set A, together with S. To INITIALIZE the summary with a set A of size s, we set $S = A$ and $n = s$. To UPDATE the summary with a new item x (which is not in A yet), we first increment n

Algorithm 2.4: RandomSample: UPDATE (x)

```
1 n ← n + 1;
2 Pick i uniformly from {1, ..., n};
3 if i ≤ s then S[i] ← x;
```

by 1. Then, with probability s/n, we choose an item currently in S uniformly at random and replace it by x; and with probability $1 - s/n$ we discard x. Algorithm 2.4 implements the UPDATE procedure by keeping the sampled items S in an array indexed from 1 to s. This way, the decision whether to add the new item and which existing item to replace can be combined by generating one random number. This also ensures that items in S are randomly permuted (for this to be true, we need to have randomly permuted S in the INITIALIZE step, too).

Algorithm 2.5: RandomSample: MERGE $((S_1, n_1), (S_2, n_2))$

```
1  k1 ← 1, k2 ← 1;
2  for i ← 1 to s do
3      Pick j uniformly over {1, ..., n1 + n2};
4      if j ≤ n1 then
5          S[i] ← S1[k1];
6          k1 ← k1 + 1;
7          n1 ← n1 − 1;
8      else
9          S[i] ← S2[k2];
10         k2 ← k2 + 1;
11         n2 ← n2 − 1;
12 return (S, n1 + n2 + s);
```

Next we describe how to MERGE two random samples S_1 and S_2, drawn from two sets A_1 and A_2 of cardinality n_1 and n_2, respectively. We proceed in s rounds, outputting one sampled item in each round. In the i-th round, with probability $n_1/(n_1+n_2)$ we randomly pick an item x in S_1 to the new sample, then remove x from S_1 and decrement n_1 by 1; with probability $n_2/(n_1 + n_2)$ we randomly pick an item x from S_2 to output to the sample, then remove it from S_2 and decrement n_2. Algorithm 2.4 implements the MERGE procedure, assuming that the two samples are stored in uniformly random order. Then the MERGE simply builds the

new sample by picking the next element from either S_1 or S_2 step by step.

Example. Suppose we are given a sequence of 10 items numbered from 1 to 10 in order. The random sample is initialized to contain the first three items after random permutation, say, [1, 3, 2]. Further suppose that the random numbers generated in line 2 of Algorithm 2.4 are 2, 5, 3, 3, 7, 9, 1. Then, the content of the array S will be as follows after each item has been processed by UPDATE.

After INITIALIZE: [1, 3, 2];
After item 4: [1, 4, 2];
After item 5: [1, 4, 2];
After item 6: [1, 4, 6];
After item 7: [1, 4, 7];
After item 8: [1, 4, 7];
After item 9: [1, 4, 7];
After item 10: [10, 4, 7];

Further Discussion. To see why UPDATE and MERGE draw a random sample, we relate random sampling to random shuffling, where an array A is randomly permuted in a way such that each permutation is equally likely. Then a random sample can be obtained by picking the first s items in the permutation.

One method for doing random shuffling is the *Fisher-Yates shuffle*. Given an array A (indexed from 1 to n), the procedure works as follows.

Algorithm 2.6: Fisher-Yates shuffle

```
1 for i ← 1 to n do
2 |   Pick j randomly from {1, ..., i};
3 |   Exchange A[i] and A[j];
```

By an easy induction proof, we can show that every permutation is possible by the above procedure. On the other hand, the procedure generates exactly $n!$ different sequences of random numbers, each with probability $1/n!$, so every permutation must be equally likely. Now, we see that if we only keep the first s items of A, each iteration of the Fisher-Yates shuffle algorithm exactly becomes the UPDATE algorithm described earlier.

To see that MERGE is also correct, imagine a process that permutes all the items in A_1 and A_2 randomly and chooses the first s of them, which form a random sample of size s from $A_1 \cup A_2$. Using the principle of deferred decisions, the first item in the permutation has probability $n_1/(n_1 + n_2)$ of being from A_1, and conditioned upon this, it is a randomly picked item from S_1. This is exactly how the MERGE algorithm picks the first sampled item for $A_1 \cup A_2$. Carrying out this argument iteratively proves the correctness of the algorithm.

The algorithms above maintain a random sample without replacement. If a random sample with replacement is desired, one can simply run s independent instances of the above algorithm, each maintaining a random sample of size 1 without replacement.

History and Background. The UPDATE algorithm is referred to as the *reservoir sampling* algorithm in the literature, first formalized by Knuth [153], who attributes it to Alan G. Waterman. The shuffling algorithm above was first described by Fisher and Yates in 1938, and later formalized by Durstenfeld [90]. The reservoir sampling algorithm has been frequently rediscovered (and used as an interview question for technical positions), but the proof being offered often only proves that the procedure samples each item with probability s/n. This is only a necessary condition for the sample to be a random sample. One can also use the definition of random sample, that is, every subset of s items is equally likely to be in the sample, but the correspondence to the Fisher-Yates shuffle yields the cleanest proof. Vitter [222] made a comprehensive study of reservoir sampling algorithm. In particular he considered the case where there is a constant-time "skip" operation that can be used to skip a given number of items in the input, and gave optimal reservoir sampling algorithms in this setting. The MERGE algorithm for merging two random samples appears to be folklore.

The applications of random sampling are so broad as to defy a concise summation. Suffice it to say, many statistical applications take a random sample of data on which to evaluate a function of interest. Random samples are used with many computer systems to estimate the cost of different operations and choose which method will be most efficient. Many algorithms use random samples of the input to quickly compute an approximate solution to a problem in preference to the slower evaluation on the full data.

Available Implementations. A version of reservoir sampling is implemented in the DataSketches library, with discussion at `https://datasketches.github.io/docs/Sampling/ReservoirSampling.html`. Experiments on commodity hardware show that speeds of tens of millions of UPDATE operations per second are achievable. Performing a MERGE depends on the size of the sample, but takes less than 1ms for samples of size tens of thousands.

2.3 Weighted Random Sampling

Brief Summary. In many situations, each item $x \in A$ is associated with a positive weight $w_x > 0$. Naturally, when we maintain a random sample of size s over weighted items, we would like to include an item i in the sample with probability proportional to w_i. In this section, we describe a WeightedRandomSample that achieves this goal. More precisely, since a probability cannot be greater than 1, item x will be included in the sample with probability $p_x = \min\{1, w_x/\tau\}$, where τ is the unique value such that $\sum_{x \in A} p_x = s$. Note that the value of τ solely depends on the weights of the items. This is assuming $s \leq n$, where n is the size of the underlying set A from which the sample is drawn. If $s > n$, we take all elements in A as the sample and set $\tau = 0$. We refer to τ as the *sampling threshold*, as all items with weight greater than τ are guaranteed to be in the sample. Note that when all weights are 1, we have $\tau = n/s$, so $p_x = 1/\tau = s/n$ for all x, and the WeightedRandomSample degenerates into a RandomSample.

Similar to a RandomSample, the WeightedRandomSample cannot handle duplicates in the input, i.e., each distinct item can be added to the WeightedRandomSample only once with a given weight, which can no longer be changed.

The most important QUERY on a WeightedRandomSample is to ask for the total weight of all items in some subset $Q \subseteq A$. If Q were given in advance, the problem would be trivial, as we can check if an item is in Q when the item is inserted to the summary. In many practical situations, Q is not known in advance. For example, in Internet traffic analysis, an item is an IP packet with various attributes like source IP, source port, destination IP, destination port, etc., while the packet size is the weight. Very often, many analytical questions are ad hoc and will

be asked after the streaming data has passed. For example, a customer might be interested in the total traffic volume of a certain application, which uses a specific source port and destination port. He or she might further narrow down the query to the traffic between two specific network domains. Such exploratory studies require a summary that supports estimating the total weight of an arbitrary subset.

Operations on the summary. Let S be the random sample. In addition, for each item $x \in S$, we maintain an adjusted weight $\tilde{w}_x = w_x/p_x$. In fact, the original weight w_x and the sampling probability p_x need not be maintained; just maintaining $\tilde{w}_x$ would be sufficient. To INITIALIZE the summary with a set A of size s, we set $S = A$ and $\tilde{w}_x = w_x$ for all $x \in S$. We split S into a subset of *large* items $L = \{x \in S \mid \tilde{w}_x > \tau\}$ and the *small* items $T = S \setminus L$. Items in L are sorted by their adjusted weights. We will maintain the invariant that $\tilde{w}_x = \tau$ for all $x \in T$, so for items in T, there is no need to record their adjusted weights explicitly. Initially, the sampling threshold τ is 0, so $T = \emptyset$, $L = S = A$, and $\tilde{w}_x = w_x$ for all $x \in L$.

The procedure to UPDATE the sample with a new item y with weight w_y is described in Algorithm 2.7. The details are a little more involved than the unweighted case, since it has more cases to handle based on whether items have weight above or below τ. The basic idea is to take a WeightedRandomSample of size s out of the $s + 1$ items, which consist of the s items in the summary plus the new one to be inserted. In this process, for the new item, we use its original weight, while for items in the current sample, we use their adjusted weights. This ensures that items survive in the sample with the correct probabilities.

We will build a set X (implemented as an array) with items outside of T and L whose weights we know are smaller than the new threshold $\tau' > \tau$. To start, if w_y is less than the current threshold τ, we set $X \leftarrow \{y\}$; otherwise item y is considered large, and so we set $X \leftarrow \emptyset$ and insert y into L (lines 3–6). Then, we are going to move items from the current L to X until L contains only items with weights greater than the new threshold τ. For that purpose, we will maintain the sum W of adjusted weights in $X \cup T$. The sum of T is known as $\tau|T|$ (line 2), to which we add w_y if $w_y < \tau$ (line 6).

Then we remove items in L in the increasing order of their weights. Let the current smallest item in L be h. We move h from L to X if setting

Algorithm 2.7: WeightedRandomSample: UPDATE (y)

```
1  X ← ∅, w̃_y = w_y;
2  W ← τ|T|;
3  if w_y > τ then insert y into L;
4  else
5      X ← {y};
6      W ← W + w̃_y;
7  while L ≠ ∅ and W ≥ (s − |L|)(min_{h∈L} w̃_h) do
8      h ← arg min_{h∈L} w̃_h;
9      move h from L to X;
10     W ← W + w̃_h;
11 τ ← W/(s − |L|);
12 generate r uniformly random from [0, 1];
13 i ← 1;
14 while i ≤ |X| and r ≥ 0 do
15     r ← r − (1 − w̃_{X[i]}/τ);
16     i ← i + 1;
17 if r < 0 then remove X[i − 1] from X;
18 else remove an item from T chosen uniformly at random;
19 T ← T ∪ X;
```

$\tau' = \tilde{w}_h$ is not enough to reduce the sample size to s, i.e.,

$$W/\tilde{w}_h + |L| \geq s, \tag{2.1}$$

which is the same as the condition checked in line 7. Whenever (2.1) is true, we move h from L to X while adding $\tilde{w}_h$ to W (lines 9–10). We repeat this step until L is empty or (2.1) is violated. Then we can compute the new threshold so that

$$W/\tau' + |L| = s,$$

i.e., $\tau' = W/(s - |L|)$ (line 11).

The remaining task is to find an item to delete so that each item remains in the sample with the right probability. More precisely, items in L must all remain in the sample, while an item $x \in T \cup X$ should remain with probability $\tilde{w}_x/\tau'$, i.e., it should be deleted with probability $1 - \tilde{w}_x/\tau'$. Recall that all items in T have the same adjusted weights, so the implementation can be made more efficient as described in lines 12–

18. Here, the value of r chosen uniformly in $[0, 1]$ is used to select one to delete.

The running time of the UPDATE algorithm can be analyzed quite easily. Inserting an item in to the sorted list L takes $O(\log s)$ time. The rest of the algorithm takes time proportional to $|X|$, the number of items being moved from L to T. Since an item is moved at most once, the amortized cost is just $O(1)$.

One can see that if all items have weight 1, then L is always empty, $\tau = s/n$ where n is the total number of items that have been added, and Algorithm 2.7 does indeed degenerate into Algorithm 2.4.

Algorithm 2.8: WeightedRandomSample: MERGE ($S_1 = (\tau_1, T_1, L_1), S_2 = (\tau_2, T_2, L_2), \tau_1 \geq \tau_2$)

```
1  merge L_1 and L_2 into L;
2  T ← T_1, W ← τ_1|T|;
3  for d ← 1 to |L_2| do
4  |   X ← ∅;
5  |   run lines 7–19 of Algorithm 2.7 replacing s with s + |L_2| − d;
6  for d ← 1 to |T_2| do
7  |   X ← {T_2[d]};
8  |   W ← W + τ_2;
9  |   run lines 7–19 of Algorithm 2.7;
10 return S = (τ, T, L);
```

To merge two samples $S_1 = (\tau_1, T_1, L_1)$ and $S_2 = (\tau_2, T_2, L_2)$ one can insert items from one sample to the other (using their adjusted weights) by repeatedly calling UPDATE, which would take $O(s \log s)$ time. Observing that the bottleneck in Algorithm 2.7 is to insert the new item into the sorted list L (line 3), we can improve the running time to $O(s)$ by first inserting all items in L_2 in a batch, and then inserting the items in T_2 one by one, as described in Algorithm 2.8. To insert all items in L_2, we first merge L_1 and L_2 into one combined sorted list. Since both L_1 and L_2 are already sorted, the merge takes $O(s)$ time. Then we iteratively reduce the sample size back to s, following the same procedure as in the UPDATE algorithm. Next, we insert all items of T_2 one by one using the same UPDATE algorithm. One trick is that if we make sure $\tau_1 \geq \tau_2$ (swapping S_1 and S_2 if needed), then the adjusted weight of all the items to be inserted, which is τ_2, is always smaller than the current

$\tau \geq \tau_1$, so we will never need to insert them into L, saving the $O(\log s)$ cost in the UPDATE algorithm.

For any subset $Q \subseteq A$, let $w(Q) = \sum_{x \in Q} w_x$. One can QUERY a WeightedRandomSample S for $w(Q)$ for an arbitrary subset $Q \subseteq A$, by simply returning $\tilde{w}(Q) = \sum_{x \in S \cap Q} \tilde{w}_x$, that is, we simply add up all adjusted weights of the items in the sample that fall in Q. This turns out to be an unbiased estimator of the true total weight $w(Q)$ with strong guarantees on the variance, as discussed later.

Example. Suppose we are to maintain a weighted random sample of size $s = 4$ over a sequence of 8 items numbered from 1 to 8 in order. The following example shows one possible execution of the UPDATE algorithm, together with the contents of τ, T, L, and X. We use the notation $x : w_x$ to denote an item x with weight w_x or adjusted weight $\tilde{w}_x$. Note that all items in T have the same adjusted weight τ.

INITIALIZE:	$\tau = 0, L = [2:1, 3:3, 1:4, 4:8], T = \emptyset$;
UPDATE (5 : 3):	Add $5:3$ to L: $L = [2:1, 3:3, 5:3, 1:4, 4:8]$
	New $\tau = 7/2$, $X = [2:1, 3:3, 5:3]$
	Deletion probabilities: $2:5/7, 3:1/7, 5:1/7$
	Suppose item 3 is deleted
	$T = \{2, 5\}, L = [1:4, 4:8]$;
UPDATE (6 : 5):	Add $6:5$ to L : $L = [1:4, 6:5, 4:8]$
	New $\tau = 16/3$, $X = [1:4, 6:5]$
	Deletion probabilities: $2:11/32, 5:11/32, 1:1/4, 6:1/16$
	Suppose item 5 is deleted
	$T = \{1, 2, 6\}, L = [4:8]$;
UPDATE (7 : 1):	Add $7:1$ to X : $X = [7:1]$
	New $\tau = 17/3$, $X = [7:1]$
	Deletion probabilities: $1:1/17, 2:1/17, 6:1/17, 7:14/17$
	Suppose item 7 is deleted
	$T = \{1, 2, 6\}, L = [4:8]$;
UPDATE (8 : 7):	Add $8:7$ to L : $L = [8:7, 4:8]$
	New $\tau = 8, X = [8:7, 4:8]$
	Deletion probabilities: $1:7/24, 2:7/24, 6:7/24, 8:1/8, 4:0$
	Suppose item 1 is deleted
	$T = \{2, 4, 6, 8\}, L = \emptyset$.

Next, suppose we want to merge two weighted samples $S_1 = (\tau_1 = 4, T_1 = \{1, 2\}, L_1 = \{3 : 5, 4 : 6\})$ and $S_2 = (\tau_2 = 3, T_2 = \{5\}, L_2 = \{6 :$

$4, 7 : 4, 8 : 11\}$). We first merge L_1 and L_2 into L, and then perform the deletions iteratively, as follows.

Merge L_1, L_2:	$T = \{1, 2\}, L = [6 : 4, 7 : 4, 3 : 5, 4 : 6, 8 : 11]$;
$s = 6$:	New $\tau = 21/4$, $X = [6 : 4, 7 : 4, 3 : 5]$
	Deletion probabilities: $1 : 5/21, 2 : 5/21, 6 : 5/21, 7 : 5/21, 3 : 1/21$
	Suppose item 1 is deleted
	$T = \{2, 6, 7, 3\}, L = [4 : 6, 8 : 11]$;
$i = 5$:	New $\tau = 27/4, X = [4 : 6]$
	Deletion probabilities: $2 : 2/9, 6 : 2/9, 7 : 2/9, 3 : 2/9, 4 : 1/9$
	Suppose item 7 is deleted
	$T = \{2, 6, 3, 4\}, L = [8 : 11]$;
$i = 4$:	New $\tau = 9, X = \emptyset$
	Deletion probabilities: $2 : 1/4, 6 : 1/4, 3 : 1/4, 4 : 1/4$
	Suppose item 3 is deleted
	$T = \{2, 6, 4\}, L = [8 : 11]$.

Next, we insert items in T_2 one by one, each with weight $\tau_2 = 3$.

UPDATE $(5 : 3)$:	$X = [5 : 3], W = 30$
	New $\tau = 10$, $X = [5 : 3]$
	Deletion probabilities: $2 : 1/10, 6 : 1/10, 4 : 1/10, 5 : 7/10$
	Suppose item 5 is deleted
	$T = \{2, 6, 4\}, L = [8 : 11]$;

Further Discussion. There is no clear generalization of the classical random sample definition that "every subset of size s is sampled with equal probability" to the weighted case. Nevertheless, the WeightedRandomSample described above has the following nice properties, which are sufficient to derive strong statistical properties regarding the estimation of an arbitrary subset sum.

(i). Inclusion probabilities proportional to size (IPPS). Each item x is sampled with probability $p_x = \min\{1, w_x/\tau\}$, where τ is the unique value such that $\sum_{x \in A} \min\{1, w_x/\tau\} = s$ if $s < n$; otherwise $\tau = 0$, meaning that all items are sampled. A sampled item is associated with adjusted weight $\tilde{w}_x = w_x/p_x$. Set $\tilde{w}_x = 0$ if x is not sampled. It is easy to see that $\mathsf{E}[\tilde{w}_x] = w_x$, and $\tilde{w}_x$ is known as the Horvitz-Thompson estimator.

(ii). The sample size is at most s. Together with (i), this means that the sample size must be $\min\{s, n\}$.

(iii). No positive covariances, i.e., for any two distinct $x, y \in A$, $\mathsf{Cov}[\tilde{w}_x, \tilde{w}_y] \leq 0$.

Let $\textsc{Sample}_s(I)$ be a procedure that takes a sample of size s from a set of weighted items I so that properties (i)–(iii) above are satisfied. It turns out such a $\textsc{Sample}_s$ can be constructed recursively, as follows. Let $I_1, \ldots, I_m$ be disjoint nonempty subsets of I, and let $s_1, \ldots, s_m \geq s$. Then

$$\textsc{Sample}_s(I) = \textsc{Sample}_s\left(\bigcup_{i=1}^{m} \textsc{Sample}_{s_i}(I_i)\right). \tag{2.2}$$

Here, when running $\textsc{Sample}$ on a sample produced by another call to $\textsc{Sample}$, we simply treat the adjusted weights of the items as if they were their original weights. We omit the correctness proof of this recurrence, which can be found in [54]. Here, we only show how it leads to the $\textsc{Update}$ and $\textsc{Merge}$ algorithms described earlier.

First, the trivial base case of $\textsc{Sample}_s$ is when I has at most s items, in which case $\textsc{Sample}_s(I) = I$. We need another base case when I has $s + 1$ items, denoted $\textsc{Sample}_{s,s+1}$. In this case an item needs to be deleted at random with appropriate deletion probabilities. To determine the deletion probabilities, we first find the correct value of τ. Then, item x should be sampled with probability $p_x = \max\{1, w_x/\tau\}$, i.e., should be deleted with probability $1 - p_x$. After a randomly chosen item has been dropped, we compute the adjusted weights of the remaining items as $\tilde{w}_x = w_x/p_x$, which is τ if $w_x \leq \tau$, and w_x if $w_x > \tau$.

To use this for the $\textsc{Update}$ algorithm, we specialize (2.2) with $m = 2$, $s_1 = s_2 = s$, $I_1 = A$, $I_2 = \{y\}$. Note that $\textsc{Sample}_s(I_2) = \{y\}$. Then the $\textsc{Update}$ algorithm becomes exactly $\textsc{Sample}_{s,s+1}$, by treating the adjusted weights of items in $\textsc{Sample}_s(A)$ as their weights. To reduce the running time, we exploit the observation above, maintaining the items in two lists T and L. All items in T have the same adjusted weights τ, while items in L have their adjusted weights equal to their original weights.

To compute a merged sample $\textsc{Sample}_s(A)$ from $\textsc{Sample}_s(A_1)$ and $\textsc{Sample}_s(A_2)$, where $A = A_1 \cup A_2, A_1 \cap A_2 = \emptyset$, recurrence (2.2)

says that we can simply run SAMPLE$_s$ on SAMPLE$_s(A_1) \cup$SAMPLE$_s(A_2)$ treating their adjusted weights as their weights. To do so, we insert each item in SAMPLE$_s(A_2)$ into SAMPLE$_s(A_1)$ one by one by the UPDATE algorithm. The MERGE algorithm described above is just a more efficient implementation of this process.

Now we discuss the statistical properties of the subset sum estimator $\tilde{w}(Q) = \sum_{x \in S \cap Q} \tilde{w}_x$. First, because item x is sampled with probability p_x, and when it is sampled, its adjusted weight is set to w_x/p_x, and to 0 otherwise. Thus, we have

$$\mathsf{E}[\tilde{w}_x] = p_x \cdot w_x/p_x = w_x,$$

i.e., $\tilde{w}_x$ is an unbiased estimator of w_x. Because each $\tilde{w}_x$ is unbiased, the unbiasedness of $\tilde{w}(Q)$ follows trivially. The variance of $\tilde{w}(Q)$ enjoys the following three forms of guarantees.

1. *Average variance:* Consider all the subsets $Q \subseteq A$ of size $m \leq n$. Their average variance is

$$V_m = \frac{\sum_{Q \subseteq A, |Q|=m} \mathsf{Var}[\tilde{w}(Q)]}{\binom{n}{m}}.$$

It has been shown that [209]

$$V_m = \frac{m}{n}\left(\frac{n-m}{n-1}\Sigma V + \frac{m-1}{n-1} V\Sigma\right),$$

where

$$\Sigma V = \sum_{x \in A} \mathsf{Var}[\tilde{w}_x],$$

$$V\Sigma = \mathsf{Var}\left[\sum_{x \in A} \tilde{w}_x\right].$$

It is known [199] that the IPPS property (i) uniquely minimizes ΣV under a given sample size s. Also, as seen earlier, $\tilde{w}_x$ is either τ (when $w_x \leq \tau$ and x is sampled) or w_x (when $w_x \geq \tau$, in which case x must be sampled). Since τ is deterministic, $\sum_{x \in A} \tilde{w}_x$ is also deterministic due to property (ii). So $V\Sigma = 0$. Therefore we conclude that V_m is minimized simultaneously for all values of m, among all sampling schemes under a given sample size s.

2. *Expected variance:* Let W_p denote the expected variance of a random subset Q including each item $x \in A$ independently with some probability p. It is also shown that [209]

$$W_p = p((1-p)\Sigma V + pV\Sigma).$$

Thus by the same reasoning, W_p is minimized for all values of p.

3. *Worst-case variance:* Combining the IPPS property (i) and the non-positive covariance property (iii), we can also bound $\mathsf{Var}[\tilde{w}(Q)]$ for any particular Q. Since each individual $\tilde{w}_x$ is w_x/p_x with probability p_x and 0 otherwise, its variance is $\mathsf{Var}[\tilde{w}_x] = w_x^2(1/p_x - 1)$. Observing that $\tau \leq w(A)/s$, so $p_x = \max\{1, w_x/\tau\} \geq \min\{1, sw_x/w(A)\}$. If $p_x = 1$, then $\mathsf{Var}[\tilde{w}_x] = 0$. Otherwise, we must have $p_x \geq sw_x/w(A)$, and we can give two bounds on $\mathsf{Var}[\tilde{w}_x]$:

$$\mathsf{Var}[\tilde{w}_x] < w_x^2/p_x \leq w_x w(A)/s, \tag{2.3}$$

or

$$\mathsf{Var}[\tilde{w}_x] < w_x^2\left(\frac{w(A)}{sw_x} - 1\right) = \left(\frac{w(A)}{2s}\right)^2 - \left(w_x - \frac{w(A)}{2s}\right)^2 \leq \left(\frac{w(A)}{2s}\right)^2. \tag{2.4}$$

Combining with the non-positive covariance property, we have $\mathsf{Var}[\tilde{w}(Q)] \leq \sum_{x\in Q} \mathsf{Var}[\tilde{w}_x]$. Plugging in (2.3) and (2.4) respectively yields a weight-bounded variance $\mathsf{Var}[\tilde{w}(Q)] \leq w(Q)w(A)/s$, and a cardinality bounded variance $\mathsf{Var}[\tilde{w}(Q)] \leq |Q|(w(A)/2s)^2$.

History and Background. The development of the algorithms and analysis largely follows that of Cohen *et al.* [54], where it is referred to as variance optimal, or VarOpt sampling. The UPDATE algorithm described above has an amortized time cost of $O(\log s)$. This can be improved to $O(\log\log s)$ if we assume that the weights are integers or finite-precision floating point numbers, by storing L in a priority queue that supports fast updates (see the work of Thorup [213]). It can also be shown that if the stream is randomly ordered, then the amortized cost will be $O(1)$. The MERGE algorithm is not explicitly described in [54], though.

This version of weighted sampling was designed for the problem of sampling effectively from massive streams of network data. Here, uniform sampling of data connections would tend to over-represent the many small flows and miss the important large flows. Simply picking

out the largest flows would not give a full picture of the entire distribution. Weighted sampling allows the range of different flow sizes to be better represented, and volume-based queries (what fraction of network traffic is of a particular type, or headed to a particular set of destinations) to be answered accurately. Further examples in the networking domain are discussed by Duffield *et al.* [88].

Available Implementations. VarOpt sampling is implemented in the DataSketches library, and is discussed at `https://datasketches.github.io/docs/Sampling/VarOptSampling.html`, with routines implemented for UPDATE and MERGE procedure. The implementation also provides lower and upper bounds on the estimations of subset sums.

2.4 Priority Sampling

Brief Summary. A PrioritySample is also a random sample over a set A of weighted items, which can be used to estimate the total weight of an arbitrary subset, as supported by a WeightedRandomSample. The variance achievable by a PrioritySample is slightly worse than that of WeightedRandomSample. However, its simpler implementation often makes it a preferable choice in practice.

Operations on the summary. For each item $x \in A$ with weight w_x, we generate an independent uniformly distributed random number $\alpha_x \in (0, 1)$, and set its *priority* to $q_x = w_x/\alpha_x$. A PrioritySample S of size s consists of the s items of the highest priority. The threshold τ is the $(s+1)$-st highest priority. The summary simply stores these items, their weights and priorities, as well as the threshold τ. We can store them in a priority queue so that new items can be added to the summary easily in $O(\log s)$ time. Merging two such summaries can be also easily done in $O(s)$ time using the linear-time heap building algorithm [59].

For any subset $Q \subseteq A$, we can QUERY a PrioritySample S for $w(Q)$ in a similar way as on a WeightedRandomSample, by returning $\tilde{w}(Q) = \sum_{x \in S \cap Q} \tilde{w}_x$. The only difference is that the adjusted weight $\tilde{w}_x$ is defined as $\tilde{w}_x = \max\{w_i, \tau\}$. This is also an unbiased estimator of $w(Q)$, but its variance is slightly worse than WeightedRandomSample, as discussed later.

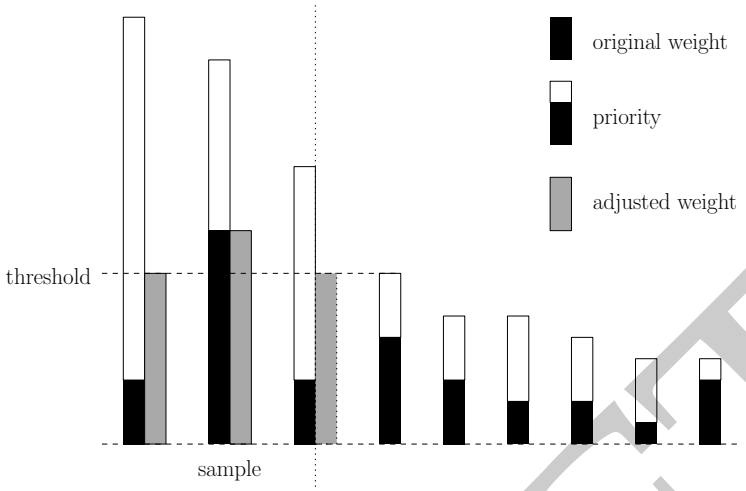


Figure 2.1 Priority sampling of size $s = 3$ from a set of 9 weighted items.

Example. A graphical example of a PrioritySample is shown in Figure 2.1.

Further Discussion. We now show that for any x, $\tilde{w}_x$ is an unbiased estimator of w_x. In fact, we will prove a stronger result, that $\mathsf{E}[\tilde{w}_x] = w_x$ regardless of the values of all $\alpha_y, y \in A, y \neq x$. Let τ' be the s-th highest of the priorities $q_y, y \in A, y \neq x$. More formally, we will show

$$E[\tilde{w}_x \mid \Lambda(\tau')] = w_x, \tag{2.5}$$

where $\Lambda(\tau')$ is the event that τ' is the s-th highest in $\{q_y : y \in A, y \neq x\}$. Proving (2.5) for any value τ' implies that $\mathsf{E}[\tilde{w}_x] = w_x$.

We first analyze the probability that item x is picked into the sample S, conditioned on $\Lambda(\tau')$. If $q_x < \tau'$, there are at least s priorities higher than q_x, so x will not be picked. If $q_x > \tau'$, then τ' becomes the $(s+1)$-st priority among all priorities, so $\tau = \tau'$ and x will be picked into S. Thus,

$$\Pr[x \in S \mid \Lambda(\tau')] = \Pr[q_x > \tau'] = \Pr[\alpha_x < w_x/\tau'] = \min\{1, w_x/\tau'\}.$$

Therefore, we have

$$\begin{aligned}\mathsf{E}[\tilde{w}_x \mid \Lambda(\tau')] &= \Pr[x \in S \mid \Lambda(\tau')] \cdot \mathsf{E}[\tilde{w}_x \mid x \in S \cap \Lambda(\tau')] \\ &= \min\{1, w_x/\tau'\} \cdot \max\{w_x, \tau'\} \\ &= w_x.\end{aligned}$$

The last equality follows by observing that both the min and the max take their first value iff $w_x > \tau'$.

Therefore, PrioritySample also returns an unbiased estimator $\tilde{w}(Q)$ for any subset $Q \subseteq A$, like WeightedRandomSample. In addition to unbiasedness, a PrioritySample also trivially has property (ii) sample size at most s, as well as (iii) no positive covariances, which WeightedRandomSample enjoys as described in Section 2.3. In fact, a PrioritySample has exactly 0 covariance between any $\tilde{w}_x$ and $\tilde{w}_y$, for $x \neq y$ (see [87] for a proof).

However, the variance of PrioritySample is slightly worse than that of WeightedRandomSample in the following two aspects. First, PrioritySample does not have property (i) inclusion probabilities proportional to size (IPPS). Although we have shown that conditioned on $\Lambda(\tau')$, the probability that x is sampled into S is indeed proportional to w_x, it does not imply that the overall sampling probability is proportional to w_x. Therefore, PrioritySample does not achieve the minimum ΣV that WeightedRandomSample achieves. Nevertheless, as shown in [208], a PrioritySample with sample size $s + 1$ can achieve a ΣV that is as good as the minimum ΣV achievable by a WeightedRandomSample with sample size s. Thus, the difference between the two sampling methods in terms of ΣV is really negligible.

Second, we know from Section 2.3 that WeightedRandomSample has $V\Sigma = 0$. However, for a PrioritySample, we have $V\Sigma = \Sigma V$ (because the covariances are 0). So, it has larger variances for larger subsets Q.

2.5 k Minimum Values (KMV) for set cardinality

Brief Summary. Similar to the MorrisCounter, the KMV summary also counts the number of elements in the set being summarized. But unlike the MorrisCounter, the KMV summary is able to remove duplicates in the input automatically. In other words, KMV summarizes a multiset A

of items and can be used to report the approximate number of distinct items observed, i.e., to approximate the size of the support set. The accuracy of the approximation determines the size of the summary, and there is a small probability (δ) of giving an answer which is outside the desired approximation guarantee (ϵ). The summary works by applying a hash function to the input items, and keeping information about the k distinct items with the smallest hash values. The size of the summary, k, is determined by the accuracy parameters ϵ and δ.

Operations on the summary. One instance of a KMV summary can be represented as a list of hash values, and corresponding items which led to those hash values.

Algorithm 2.9: KMV:INITIALIZE (k)

```
1 Pick hash function h, and store k;
2 Initialize list L = ∅ for k item, hash pairs;
```

The INITIALIZE operation picks a hash function h mapping onto a range $1 \ldots R$, and creates an empty list capable of holding up to k hashes and items.

Algorithm 2.10: KMV: UPDATE (x)

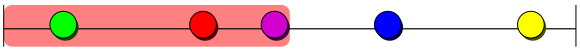
```
1 if x ∉ L then
2     L ← L ∪ {(x, h(x))};
3     if |L| > k then Remove item x with largest hash value h(x)
        from L;
```

Each UPDATE operation receives an item, and applies the function h to it. If the item is already present in the list it is dropped, and need not be stored. If the hash value is among the k smallest hash values, the item and its hash value are stored in the list, and the list pruned to retain only the k smallest. We use set notation to refer to the list of items L, augmented so that we can find the kth largest hash value (Algorithm 2.11). This can be supported with a heap data structure, linked to a hash table to test for presence of an item.

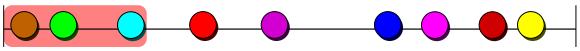
Algorithm 2.11: KMV: QUERY ()

```
1 v_k ← largest hash value in L;
2 return (k − 1) ∗ R/v_k
```

To QUERY the summary for the estimated number of distinct items seen, find the kth smallest hash value seen so far, as v_k, and compute the estimate as $R(k-1)/v_k$. If there are fewer than k values stored, then



(a) KMV summary after 5 distinct items have been processed



(b) KMV summary after 9 distinct items have been processed

Figure 2.2 KMV summary with $k = 3$

these represent the set of distinct items seen, and so the exact answer can be given.

To MERGE together two summaries (where, we require that both were built using the same hash function h), and obtain a summary of the union of the inputs, the two lists are merged, and the set of distinct pairs of hashes and items is found. The k pairs corresponding to the k smallest hash values are retained.

Example. Figure 2.2 shows a schematic representation of the KMV summary as items are processed, with $k = 3$. Different items are shown as different colored balls, with each ball being represented as its hash value, smallest on the left and largest on the right. Figure 2.2(a) shows the status after 5 distinct items have been observed. The highlighted items, corresponding to the 3 smallest hash values, are stored in the summary. As more items arrive (Figure 2.2(b)), the summary is updated: two new items enter the k smallest, and only information about the current k smallest are stored. In this example, the ratio R/v_k is approximately 4 (the kth smallest hash value falls about $1/4$ of the way along the range), and so the estimate given is $2 \times 4 = 8$, which is close to the true answer of 9.

Further Discussion. The initial properties of KMV can be understood as follows. Suppose that the input domain is the range $1...M$, and the output range R is chosen to be M^3. This is large enough that hash collisions are so unlikely we can discount them. Let the hash function h be chosen from a family of pairwise independent hash functions.

We now study the chance that the estimated answer D' is much

larger than the true answer, $|A|$. Suppose $R(k-1)/v_k > (1+\epsilon)|A|$ for some $\epsilon < 1$. This implies that at least k items from the input had a hash value that was below a value $\tau := R(k-1)/(1+\epsilon)|A|$, i.e., v_k was smaller than it should have been. We prefer to have the dependence on ϵ in the numerator rather than the denominator, so we upper bound this value of τ using the fact that $\epsilon < 1$:

$$\tau = \frac{R(k-1)}{(1+\epsilon)|A|} \leq \frac{R(k-1)}{|A|}\left(1 - \frac{\epsilon}{2}\right).$$

Considering a single item x, the probability that its hash value is below this threshold τ is at most

$$\frac{1}{R} \cdot \left(1 - \frac{\epsilon}{2}\right)\frac{(k-1)R}{|A|} = \left(1 - \frac{\epsilon}{2}\right)\frac{k-1}{|A|} := p.$$

Note that technically, this probability should be slightly increased by a small amount of $1/M$, to account for rounding of the hash values; however, this does not significantly affect the analysis, so we gloss over this quantity in the interest of keeping the expressions clear to read.

For the analysis, we create Bernoulli random variables X_i for each of the distinct items, indexed from 1 up to $|A|$, to indicate the event that the item's hash value is below τ. Each of these is 1 with probability at most p. Let Y be the sum of these $|A|$ random variables: a necessary condition for the estimate being too large is $Y \geq k$. We compute

$$\mathsf{E}[Y] = |A|p \leq \left(1 - \frac{\epsilon}{2}\right)(k-1).$$

The variance of Y can be computed as the sum of variances of the X_is. Each $\mathsf{Var}[X_i] \leq p(1-p)$, and these are (pairwise) independent, so

$$\mathsf{Var}[Y] = |A|p(1-p) \leq \left(1 - \frac{\epsilon}{2}\right)(k-1).$$

We can then apply the Chebyshev inequality, to conclude that

$$\Pr[Y \geq k] \leq \Pr\left[|Y - \mathsf{E}[Y]| > \frac{\epsilon}{2}k\right] \leq 4\frac{\mathsf{Var}[Y]}{\epsilon^2 k^2} \leq \frac{4}{\epsilon^2 k}.$$

The analysis for the case that the estimate falls below $(1-\epsilon)|A|$ is

very similar. This occurs when fewer than k items hash *below*

$$\frac{(k-1)R}{(1-\epsilon)|A|} \geq (1+\epsilon)\frac{(k-1)R}{|A|}.$$

As before, the probability of this can be bounded using the Chebyshev inequality over a summation of Bernoulli random variables with parameter $p' := (1+\epsilon)(k-1)/|A|$. The estimate is too small when more than k of these events occur out of the $|A|$ trials. Thus we can similarly show that the probability of this undesirable outcome is bounded by $\frac{1}{\epsilon^2 k}$.

Combining these two probabilities, we have that

$$\Pr[|R(k-1)/v_k - |A|| \leq \epsilon|A|] \leq \frac{5}{\epsilon^2 k}$$

To make this at most $1/4$, we choose $k = 20/\epsilon^2$. To reduce the probability to δ, we repeat the process $4 \log 1/\delta$ times with different hash functions and take the median estimate (following the argument of Section 1.4.1). This implies that we need space to store $80/\epsilon^2 \log 1/\delta$ items in total.

A tighter analysis was provided by [26]. They study higher moments of the estimator, under the assumption of fully random hash functions, and show that the estimator is unbiased. They show in particular that the variance of this estimator is proportional to $|A|^2/k$, and so setting k to $1/\epsilon^2$ guarantees that the standard deviation is proportional to $\epsilon|A|$, which bounds the expected absolute error. Thus the asymptotic performance is essentially the same: to give an ϵ relative error, a summary of size proportional to $1/\epsilon^2$ is needed, but the true constants of proportionality are smaller than the above basic analysis would indicate.

Unions, Intersections and Predicates. The MERGE operation corresponds to computing a summary of the union of the sets being combined. The correctness of this process is easy to see: since the hash values of items can be treated as fixed, the KMV summary of the union of the sets is defined by the k smallest hash values of items in the union, which can be found from the k smallest hash values of the two input sets. These in turn are given by the KMV summaries of these sets.

For intersections between sets, we can look to the intersection of their corresponding summary. Specifically, we can count the num-

ber of elements in the intersection of the corresponding KMV summaries. Roughly speaking, we expect the fraction of elements in the intersection of the two summaries to be proportional to the overall fraction of elements in the intersection. That is, for sets A and B it should be proportional to $\frac{|A \cap B|}{|A \cup B|}$. Let ρ be the observed fraction of elements intersection, i.e., $\rho = i/k$, where i is the number of intersecting elements. We use this to scale the estimate obtained for the size of the union. The variance of this estimator can be shown to scale according to $|A \cap B| \cdot |A \cup B|/k$ [26]. That is, if we pick $k \propto 1/\epsilon^2$, the error (standard deviation) will be bounded by $\epsilon \sqrt{|A \cap B| \cdot |A \cup B|}$. Since we do not know how big $|A \cap B|$ will be in advance, this can be bounded by $\epsilon |A \cup B|$ in the worst case, i.e., the (additive) error in estimating an intersection scales at worst in accordance with the size of the union of the sets.

The idea for the intersection can be applied to other concepts. Suppose we want to estimate the number of distinct items in the input that satisfy a particular property (i.e., a predicate). Then we can see what fraction of items stored in the summary satisfy the property, and then scale the estimated number of distinct items by this fraction. Again, the error scales with $\epsilon |A \cup B|$ in the worst case, so this will not work well when the desired property is very rare. This should not be surprising: if the property is rare, then it is unlikely that any of the items within the summary will possess it, and hence our estimate will be 0.

Generalization to multisets. We can generalize the summary to track information about the multiplicity of items in a multiset. The approach is natural: we additionally keep, for each item stored in the summary, the number of times it has occurred. This is useful when we wish to additionally pose queries that involve the multiplicity of items: for example, we could ask how many items occur exactly once in the data (this statistic is known as the *rarity*), and estimate this based on the fraction of items in the summary which occur exactly once. This can be addressed using the approach above for counting the number of items satisfying a given property.

The summary can still operate correctly when the input stream includes operations which delete an item, as well as just insertions of items. We keep track of the multiplicity of each item in the summary, and remove any whose count reaches zero after a

deletion operation. This can have the effect of bringing the size of the summary down below k. To ensure a correct estimate, we must keep track of the least value of the kth smallest hash value that has been maintained, m_k. We must ensure that the summary contains information about all items in the input which hash below m_k. Consequently, even when the summary has fewer than k items in it, we cannot fill it up with elements from the input that hash above m_k. Instead, we must use the summary with the current "effective" value of k, i.e., the number of items that hash below m_k. This may mean that in a very dynamic situation, where there are a large number of deletions, the effective value of k may become very small, meaning that the summary gives poor accuracy. There are summaries which can cope with such deletions, which are described later in Section 2.6, at the cost of using more space.

Implementation Issues. In our description, we have stated that we should keep the original items in addition to the hashed values. For the basic uses of the summary, this is not necessary, and we can retain only the hash values to perform the estimation of set size, and union and intersection sizes. It is only when applying predicate tests that we also need to keep the original items, so that the tests can be applied to them. Thus, the space can be reduced to the size of the hash values, which should depend on the expected maximum number of distinct items, rather than the size of the input items. This can be very space efficient when the input items are very large, e.g., long text strings. Additional space efficiency can be obtained based on the observation that as the number of distinct items increases, the fraction of the hash range occupied decreases: the hash values of the stored items begin with a prefix of zero bits. This prefix can therefore be omitted, to reduce the size of the stored hashes.

History and Background. The KMV summary was introduced in the work of Bar-Yossef *et al.* [18]. This paper introduced a selection of algorithms for tracking the number of distinct items in a stream of elements. Of these, the KMV has had the most impact, as it blends space efficiency with simplicity to implement. A version of KMV with $k = 1$ was first proposed by Cohen [52]. The idea can be seen as related to the work of Gibbons and Tirthapura [109], which also uses the idea of hash functions to select a set of items from the input based on their hash value. This in turn is conceptually (though less clearly) related to

the earliest work on distinct counting, due to Flajolet and Martin [103]. The tighter analysis of the KMV summary is due to Beyer *et al.* [26]. This proposed an unbiased version of the estimator (previous presentations used a biased estimator), and analyze its higher moments under the assumption of fully random hash functions. A generalization of KMV is presented by Dasgupta *et al.* [77] in the form of the theta-sketch framework. This generalizes KMV by considering sketching methods defined by a threshold θ derived from the stream which is used to maintain a set of hash values of stream elements below the threshold. It is instantiated by different choices of function to determine the threshold θ. The framework can naturally handle MERGE (union) operations, as well as estimate the cardinality of intersections and more general set expressions.

Available Implementations. The DataSketches library provides an implementation of the KMV summary within the more general framework of θ-sketches. Concurrent versions are also implemented that can efficiently parallelize the computation. Experiments on commodity hardware show that the sketch can process tens of millions of UPDATE operations per second in a single thread. Evaluation of standard error shows that a summary with k around ten thousand is sufficient to obtain relative error of 1% with high probability.

2.6 HyperLogLog (HLL) for set cardinality

Brief Summary. Like KMV, the HLL summary also summarizes a multiset A of items in order to approximate the number of distinct items observed, but in a very bit-efficient way. It tracks information about hashed values of input items, and uses this to build an accurate approximation of the counts. Different summaries can be combined to estimate the size of the union of their inputs, and from this, the size of the intersection can also be estimated.

Algorithm 2.12: HLL:INITIALIZE (m)

```
1 Pick hash functions h, g and store m;
2 C[1] ... C[m] ← 0;
```

Operations on the summary. The HLL summary is stored as an array of m entries. Each input item is mapped by a hash function to an en-

try, and a second hash function is applied; the array entry keeps track of statistics on the hash values mapped there. Specifically, it tracks the largest number of leading zeros in the binary encoding of evaluations of the hash function. The INITIALIZE function creates the two hash functions, h to map to entries in the array, and g to remap the items. It initializes the array of size m to all zeros (Algorithm 2.12).

Algorithm 2.13: HLL: UPDATE (x)

```
1 C[h(x)] ← max(C[h(x)], z(g(x)));
```

To UPDATE a summary with item x, we make use of the function $z(\cdot)$, which returns the number of leading zeros in the binary representation of its argument. For example, $z(1010)$ is 0 (applied to the 4 bit binary input 1010, corresponding to the integer 12), while $z(0001)$ is 3. The new item is mapped under h to an entry in the array, and we test whether $z(g(x))$ is greater than the current item. If so, we update the entry to $z(g(x))$. This is expressed in Algorithm 2.13.

To MERGE two summaries built using the same parameters ($h()$, $g()$, and m), we merge the arrays in the natural way: for each array entry, take the maximum value of the corresponding input arrays.

Algorithm 2.14: HLL: QUERY ()

```
1 X ← 0;
2 for j ← 1 to m do X ← X + 2^{-C[j]};
3 return (α_m m^2 / X);
```

To QUERY the summary for the approximate number of distinct items, we extract an estimate for the number of distinct items mapped to each array entry, and then combine these to get an estimate for the total number. Specifically, for each entry of the matrix, we take 2 raised to the power of this value, and then take the harmonic mean of these values. This is shown in Algorithm 2.14. The estimate is then an appropriately rescaled value of this estimate, based on a constant $\alpha_m = 0.7213/(1 + 1.079/m)$, discussed below.

Example. We show a small example with $m = 3$. Consider 5 distinct items a, b, c, d, e, with the following hash values:

x	a	b	c	d	e
$h(x)$	1	2	3	1	3
$g(x)$	0001	0011	1010	1101	0101

From this, we obtain the following array:

3	2	1

Applying the QUERY function, we obtain $X = 7/8$, and choosing $\alpha_m = 0.5305$, we get an estimate of the number of distinct items as 5.45, which is close to the true result of 5.

Further Discussion. The central intuition behind the HLL summary is that tracking the maximum value of $z(g(x))$ gives information about the number of distinct items mapped to that array entry. If we assume that g appears sufficiently random, then we expect half the hashed items seen to have 0 as their first bit (and hence have $z(g(x)) = 1$). Similarly, we expect a quarter of the items to have two leading zeros, an eighth to have three, and so on. Inverting this relationship, if we see a value of ρ in an array entry, then we interpret this as most likely to have been caused by 2^ρ distinct items. As storing the value ρ only takes $O(\log \rho) = O(\log \log n)$ bits, where n is the number of distinct elements, this leads to the "log log" in the name of the summary.

Applying this argument to each of the array entries in turn, we have an estimate for the number of distinct items mapped to each entry. The most direct way to combine these would to sum them, but this has high variability. Instead, we take advantage of the property of the hash function h as mapping items approximately uniformly to each array entry. So we can interpret each of the estimates obtained as an estimate of n/m, where n is the number of distinct items. We could directly average these, but instead we adopt the harmonic mean, which is more robust to outlying values. The harmonic mean of m values x_i is $(\frac{1}{m}\sum_{i=1}^{m} x_i^{-1})^{-1}$. In Algorithm 2.14, X computes the needed sum of values, so the harmonic mean is given by m/X. This is our estimate for n/m, so we scale this by a further factor of m to obtain m^2/X as the estimate. However, this turns out to be biased, so the factor α_m is used to rescale out the bias.

A full discussion of the analysis of this estimator is beyond the scope of this presentation (see the original paper [101] for this). This shows that α_m should be picked to be $\frac{1}{2\ln 2}(1 + \frac{1}{m}(3\ln 2 - 1))$, i.e.,

$0.7213/(1 + 1.079/m)$, for m larger than 100. In this case, the bias of the estimator is essentially removed. The variance of the estimator is shown to approach $1.08/m$ as m increases, and is bounded for all $m \geq 3$. Thus the estimator achieves ϵ relative error with constant probability provided $m \geq 1/\epsilon^2$, by the Chebyshev inequality. The probability of exceeding this bound can be driven down to δ by performing $O(\log 1/\delta)$ independent repetitions and taking the median, following the Chernoff bounds argument (Section 1.4.1). However, it is argued by Flajolet *et al.* [101] that the central limit theorem applies, and the estimator follows a Gaussian distribution; in this case, increasing m by a factor of $O(\log 1/\delta)$ is sufficient to achieve the same result. Therefore, the HLL summary takes a total of $O(1/\varepsilon^2 \log(1/\delta) \log \log n)$ *bits*. This is more bit-efficient than the KMV summary, which keeps $O(1/\varepsilon^2 \log(1/\delta))$ hash values, amounting to $O(1/\varepsilon^2 \log(1/\delta) \log n)$ bits.

Intersections. Since the HLL summary can give an accurate estimate of the size of the union of two sets, it can also estimate the size of their intersection. We take advantage of the identity $|A \cap B| = |A| + |B| - |A \cup B|$, and form our estimate of $|A \cap B|$ from the corresponding estimates of $|A|$, $|B|$ and $|A \cup B|$. The error then depends on the size of these quantities: if we estimate $|A|$ and $|B|$ with relative error ϵ, then the error in $|A \cap B|$ will also depend on $\epsilon(|A| + |B|)$. Hence if $|A|$ or $|B|$ is large compared to $|A \cap B|$, the error will overwhelm the true answer.

The same principle can be extended to estimate higher order intersections, via the principle of inclusion-exclusion:

$$|A \cap B \cap C| = |A \cup B \cup C| + |A \cap B| + |A \cap C| + |B \cap C| - |A| - |B| - |C|.$$

Indeed, arbitrary expressions can be decomposed in terms of unions alone: $|(A \cap B) \cup C| = |A \cup C| + |B \cup C| - |A \cup B \cup C|$. However, the number of terms in these expressions increases quickly, causing the error to rise accordingly.

Handling deletions. The HLL summary cannot support deletions directly as it only keeps the maximum of all the $z(g(x))$'s ever seen. However, one can easily support deletions, though at the cost of using more space. The idea is for each possible value of $z(g(x))$, we keep track of the number of x's that correspond to this value,

i.e., we use a two-dimensional array C where $C[i, j] = |x : h(x) = i, z(g(x)) = j|$. This makes the summary a linear sketch of the data, so can support arbitrary deletions. However, the total space needed increases substantially to $O(1/\varepsilon^2 \log(1/\delta) \log^2 n)$ bits.

Implementation Issues. Due to its wide adoption, the HLL summary has been subject to much scrutiny, and hence a number of engineering issues have been suggested to further increase its usefulness.

Hash function issues. We have described the HLL summary in terms of two hash functions, h and g. However, implementations typically use a single hash function f, and derive the two values from this. Assume that m is chosen to be a power of two, and that $\log_2 m = b$. Also assume that g is chosen to map onto a domain of 2^d bits. Then $h(x)$ can be taken as the first b bits of a hash function, and g taken as the last d bits, providing that the single hash function f provides at least $b + d$ bits. Implementations have tended to look at specific choices of parameters: Flajolet *et al.* [101] study $d = 32$, meaning that each value of $z(g(x))$ can be represented in 5 bits. For larger cardinalities, Heule *et al.* [127] use $d = 64$, and so store 6 bits for each array entry. Choosing $d = 32$ is sufficient to handle inputs with billions of entries (hundreds of billions if m is large enough that each array entry is unlikely to exceed 32 zeros in the g hash value), while $d = 64$ allows accurate counting up to the trillions. The space of the summary is $O(m \log \log n)$ bits, but as these implementations show, $\log \log n$ can be effectively treated as a constant: choosing $d = 2^8$ (so storing 8-bit values) is enough to count to $2^{256} = 10^{77}$, larger than most natural quantities.

The analysis of the algorithm requires assuming that the hash functions used are as good as a random function (i.e., fully random). Hence, simple hash functions (pairwise or k-wise independent) are insufficient for this purpose. Instead, stronger non-cryptographic hash functions are used in practice (Section 1.4.2).

High end correction. When dealing with large cardinalities, there is the possibility of hash collisions under g. That is, assuming that g maps onto 2^d bits, there may be items $x \neq y$ such that $g(x) = g(y)$. The result is to underestimate the number of distinct items. One solution is to make a correction to the estimate, if the estimate is so large that this is a possibility. The original HLL paper [101] uses the correction that if the estimate E is more than $2^{2^d}/4$, then the new estimate is $\log_2(1 - E/2^{2^d})/2^{2^d}$.

This expression for the correction comes from considering the probability of such collisions. However, a more direct approach is simply to increase d to the point where such collisions are unlikely: given n distinct items, the expected number of collisions is approximately $n/2^{1+2^d}$, so increasing d by 1 is usually sufficient to eliminate this problem.

Low end correction. The opposite problem can occur with small counts. This is more significant, since in many applications it is important to obtain more accurate estimates for small counts. However, the HLL summary shows a notable bias for small counts. In particular, consider an empty summary with m entries. The harmonic mean of the estimates is 1, and so the estimate is approximately $0.7m$, much larger than the true value. Instead, we can use a different estimator, based on the number of entries in the summary that are non-zero (indicating that some item has updated them). After seeing n distinct items, the probability that an entry in the array is empty is $(1 - 1/m)^n$, assuming a random mapping of items to entries. This is well-approximated by $\exp(-n/m)$. So we expect a $\exp(-n/m)$ fraction of the array entries to be empty. If we observe that there are V empty entries in the array, then we make our estimator to be $m \ln(m/V)$, by rearranging this expression. This correction can be used when it is feasible to do so, i.e., when V is non-zero. This estimator is known as LinearCounting, since it works well when the number of distinct items n is at most linear in m.

More advanced approaches can be used when this correction cannot be applied but the estimate is still small. Heule *et al.* [127] empirically measure the bias, and build a mapping from the estimated value based on tabulated values, and using these to interpolate a "corrected" estimate. This can be applied when the initial estimate indicates that n is small, e.g., when it appears to be below $5m$. Storing and accessing this mapping comes at some extra space cost, but this can be less significant if there are a large number of instances of HLL being run in parallel (to count different subsets of items), or if this look-up table can be stored in slower storage (slow memory or disk) and only accessed relatively rarely.

Sparse representation for small counts. When space is at a high premium, as may be the case when a large number of HLL summaries are deployed in parallel to count a large number of different quantities, it is desirable to further compact the summary. Observe that when n is small, the number of entries of the array that are occupied is correspondingly small. In this case, it is more space efficient to store infor-

mation about only the non-empty entries, in a list. Further space reductions are possible by storing the list in sorted order, and encoding the entries based on differences between subsequent entries in a bit-efficient variable-length encoding. However, this complicates the implementation, which has to convert between formats (array-based and list-based), and handle more cases. There is considerable scope for further encodings and optimizations here, which are described and evaluated at greater length in the literature.

History and Background. The development of the HLL sketch spans three decades, and due to its popularity, additional variations and implementation issues continue to be discussed. The central idea, of mapping items under a hash function, and looking at the number of leading zeros in the hash value was introduced by Flajolet and Martin in 1983 [102]. In this early paper, alternate estimators were considered (such as tracking other statistics of the z function), and taking the usual arithmetic average of the derived estimators. This process was called "Probabilistic Counting with Stochastic Averaging", or PCSA for short. A simple analysis to show that a similar algorithm gives a constant factor approximation with bounded independence hash functions is given by Alon *et al.* [11]. The work on LogLog counting, nearly 20 years later, [89] is similar to HLL, but computes the (arithmetic) mean of the array entries first (with some truncation and restriction to reduce variance), then raises 2 to this power to obtain the estimate. The variation of using the hypergeometric mean was introduced by Flajolet, Fusy, Gandouet and Meunier in 2007 [101], and shown to be very effective. Further commentary on the evolution of the data structure is given by Lumbroso [168]. A different branch of this work is due to Lang [159], who proposed a different approach to compressing the Flajolet-Martin sketch. The resulting approach is comparable in speed to HLL, while being very space efficient.

The summary has been used widely in practice, and a subsequent paper [127] studies multiple engineering optimizations to provide a more effective implementation. The "low end correction" makes use of the LinearCounting algorithm, due to Whang *et al.* [225]. The example uses of the summary given by Heule *et al.* [127] include speeding up database queries that ask for the number of distinct elements in group-by queries (within the PowerDrill system [123]); and within tools for analyzing massive log files to apply distinct counting e.g., number of

distinct advertising impressions [196] or distinct viewers of online content [48].

Available Implementations. Implementations of HLL are available in various software libraries and systems, such as stream-lib, the Redis database and Twitter's Summingbird streaming data processing tool. The DataSketches library implements HLL and several variants, such as the corrections due to Heule *et al.* [127], and with choices of 4, 6 or 8 bits per bucket, to accommodate different anticipated cardinalities. Processing speeds of tens of millions of UPDATE operations per second are easily achievable. An implementation of Lang's "Compressed Probabilistic Counting" [159] is also available in the library for comparison.

2.7 Bloom Filters for set membership

Brief Summary. The BloomFilter summarizes a set A of items in a compact (bit-efficient) format. Given a candidate item as a query, it answers whether the item is in A or not. It provides a one-sided guarantee: if the item is in the set, then it will always respond positively. However, there is the possibility of false positives. The probability of a false positive can be bounded as a function of the size of the summary and $|A|$: as the latter increases, false positives become more likely. The summary keeps a bit string, entries of which are set to one based on a hash function applied to the updates. In its simplest form, only insertions are allowed, but generalizations also allow deletions. The BloomFilter handles duplicates in the input automatically.

Operations on the summary. The BloomFilter summary consists of a binary string B of length m and k hash functions $h_1 \dots h_k$, which each independently map elements of U to $\{1, 2, \dots m\}$. The INITIALIZE operation creates the string B initialized to all zeros, and picks the k hash functions. If the size of the set A is expected to be n, a good choice of m and k is $m = 10n$ and $k = m/n \ln 2 = 7$; the analysis below will give more guidance on how to set these parameters.

Algorithm 2.15: BloomFilter: UPDATE (i)

```
1 for j ← 1 to k do
2 |   B[h_j(i)] ← 1;
```

For each UPDATE operation to insert an element i into the set A, the BloomFilter sets $B[h_j(i)] = 1$ for all $1 \leq j \leq k$, as shown in Algorithm 2.15. Hence each update takes $O(k)$ time to process.

Algorithm 2.16: BloomFilter: QUERY (x)

```
1 for j ← 1 to k do
2 |   if B[h_j(x)] = 0 then return false;
3 return true
```

The QUERY operation on a BloomFilter summary takes an element x from U, and tries to determine whether it was previously the subject of an UPDATE operation, i.e., whether $x \in A$. The QUERY operation inspects the entries $B[h_j(x)]$, where x would be mapped to if it were inserted. If there is some $j \in [k]$ for which $B[h_j(x)] = 0$, then the item is surely not present: UPDATE would have set all of these entries to 1, and no other operation ever changes this. Otherwise, it is concluded that x is in A. Hence, Algorithm 2.16 inspects all the locations where x is mapped, and returns false if any one of them is 0.

From this description, it can be seen that the data structure guarantees no false negatives, but may report false positives. False positives occur if a collection of other items inserted into the summary happen to cause the corresponding entries $B[h_j(x)]$ to be 1, but x itself is never actually inserted.

Algorithm 2.17: BloomFilter: MERGE (B_a, B_b)

```
1 for i ← 1 to m do
2 |   B_a[i] ← max(B_a[i], B_b[i]);
```

To MERGE two BloomFilter summaries, they must be built with the same parameters i.e., with the same size m, the same number of hash functions k, and the same set of hash functions $h_1 \ldots h_k$. The resulting BloomFilter is the bit-wise OR of the two bitstrings. That is, the new B has $B[j] = 1$ if this entry was 1 in either of the input summaries, and is 0 if this entry was 0 in both. Algorithm 2.17 takes a pass through the two input summaries, and merges the second into the first.

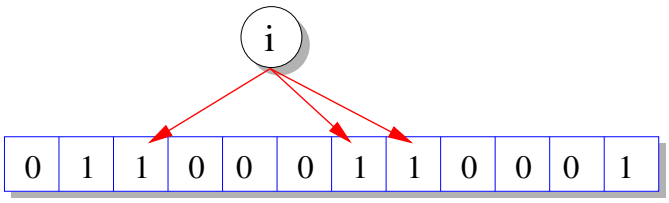


Figure 2.3 Bloom Filter with $k = 3$, $m = 12$

Example. A simple example is shown in Figure 2.3: an item i is mapped by $k = 3$ hash functions to a filter of size $m = 12$, and these entries are set to 1.

Further Discussion. A basic rule-of-thumb is that the size of the BloomFilter in bits should be roughly proportional to the size of the set S which is to be summarized. That is, the summary cannot encode a very large set in space dramatically smaller than the innate size of the set. However, the filter can be much smaller than explicitly representing the set A, either by listing its members, or storing them in a hash table. This is particularly pronounced when the identifiers of the elements are quite large, since the BloomFilter does not explicitly store these. If the size of the set A is expected to be n, allocating $m = 10n$ (i.e., 10 bits per item stored) and $k = 7$ gives a false positive probability of around 1%. To understand the relation between these parameters in more detail, we present further detailed analysis below.

Detailed Analysis of BloomFilter. The false positive rate can be analyzed as a function of $|A| = n$, m and k: given bounds on n and m, optimal values of k can be set. We follow the outline of Broder and Mitzenmacher [38] to derive the relationship between these values. For the analysis, the hash functions are assumed to be fully random. That is, the location that an item is mapped to by any hash function is viewed as being uniformly random over the range of possibilities, and fully independent of the other hash functions. Consequently, the probability that any entry of B is zero after n

distinct items have been seen is given by

$$p' = (1 - \frac{1}{m})^{kn}$$

since each of the kn applications of a hash function has a $(1 - \frac{1}{m})$ probability of leaving the entry zero.

A false positive occurs when some item not in A hashes to locations in B which are all set to 1 by other items. For an arbitrary item not in A, this happens with probability $(1-\rho)^k$, where ρ denotes the fraction of bits in B that are set to 0. In expectation, ρ is equal to p', and it can be shown that ρ is very close to p' with high probability. Given fixed values of m and n, it is possible to optimize k, the number of hash functions. Small values of k keep the number of 1s lower, but make it easier to have a collision; larger values of k increase the density of 1s. The false positive rate is

$$q = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx (1 - e^{-kn/m})^k = \exp(k \ln(1 - e^{kn/m})). \qquad (2.6)$$

The smallest value of q as a function of k is given by minimizing the exponent. This in turn can be written as $-\frac{m}{n} \ln(p) \ln(1 - p)$, for $p = e^{-kn/m}$, and so by symmetry, the smallest value occurs for $p = \frac{1}{2}$. Rearranging gives $k = (m/n) \ln 2$.

This has the effect of setting the occupancy of the filter to be 0.5, that is, half the bits are expected to be 0, and half 1. This causes the false positive rate to be $q = (1/2)^k = (0.6185)^{m/n}$. To make this probability at most a small constant, it is necessary to make $m > n$. Indeed, setting $m = cn$ gives the false positive probability at 0.6185^c: choosing $c = 9.6$, for example, is sufficient to make this probability less than 1%.

Other operations on BloomFilter *summaries.* The semantics of the MERGE operator is to provide a BloomFilter summary of the union of the two sets summarized by each input summary. It is also possible to build a summary that corresponds to the intersection of the input sets, by taking the bitwise-AND of each bit in the bitstrings, instead of the bitwise-OR. That is, if we were to keep the two BloomFilter summaries and test whether x was present in both, we would check the locations $h_j(x)$ in each summary. If there is any such location in either summary which is 0, then we would conclude x is

not present in both. This is equivalent to taking the bitwise-AND of each of these locations, and checking them all. So the BloomFilter summary which is formed as the bitwise-AND of the two input summaries will give the same response for every query x.

The summary as described so far only allows UPDATE operations which insert an item into the set A. More generally, we might like to allow deletions of elements as well. There are two possible semantics for deletions. If we treat A as a set, then a deletion of element x ensures that x is not stored in the set any more. If we treat A as a multiset, then each element in the set has a multiplicity, such that insertions increase that multiplicity, and deletions decrement it (down to 0). In both cases, to handle deletions we will replace the bit-string B with a collection of counters, but how we use these counters will differ.

We first consider the set semantics. The data structure looks much the same as before, except each bit is replaced with a small counter. Here, an UPDATE operation corresponding to an insert of i first performs a QUERY to test whether i is already represented in the summary. If so, nothing changes. Otherwise, the operation increments the counter $B[h_j(i)]$ for $h_1 \dots h_k$. Likewise, an UPDATE that is a deletion of i also checks whether i is stored in the summary. If so, the operation decrements the counters $B[h_j(i)]$. Otherwise, nothing changes. Lastly, the QUERY operation remains as shown in Algorithm 2.16: it checks all locations where i is mapped, and returns false if any is 0. The interpretation of the summary is that each counter stores the number of distinct items which have been mapped there. Insertions increase these counts, while deletions decrease these counts. Because multiple insertions of the same item do not alter the structure, these counters do not need to grow very large. However, note that it is now possible to have false negatives from this structure: an insertion of an item that is not present in the structure may not lead to counter increments if there is a false positive for it. The subsequent deletion of this item may lead to zero values, causing false negatives for other items. Analysis shows that counters with a bit depth of only four bits will suffice to keep accurate summaries without overflow. This analysis is due to Fan *et al.* , who dub this variant of the BloomFilter the "counting Bloom Filter" [97].

The case for the multiset semantics is simpler to describe, but re-

quires more space in general. Instead of a bitmap, the Bloom filter is now represented by an array of counters. When performing an UPDATE that inserts a new copy of i, we increase the corresponding counters by 1, i.e., $B[h_j(i)] \leftarrow B[h_j(i)] + 1$. Likewise, for an UPDATE that removes a copy of i, we decrease the corresponding counters by 1. Now the transform is linear, and so it can process arbitrary streams of update transactions. The number of entries needed in the array remains the same, but now the entries are counters (represented with 32 or 64 bits) rather than single bits, and these counters may now need to represent a large number of elements. This variation is referred to as a spectral Bloom filter, and is described by Cohen and Matias, who also discuss space efficient ways to implement the counters [56].

History and Background. The BloomFilter summary, named for its inventor [29], was first proposed in 1970 in the context of compactly representing information. One of the first applications was to compactly represent a dictionary for spell-checking [174]. Its attraction is that it is a compact and fast alternative to explicitly representing a set of items when it is not required to list out the members of the set. Interest in the structure was renewed in the late nineties, in the context of tracking the content of caches of large data in a network [97]. Subsequently, there has been a huge interest in using BloomFilter summaries for a wide variety of applications, and many variations and extensions have been proposed. Broder and Mitzenmacher provide a survey of some of these uses [38, 177], but there have been many more papers published on this summary and its variants in the interim.

A common theme of applications of Bloom filters is that they require a small fraction of false positives must be acceptable, or that it be feasible to double-check a positive report (at some additional cost) using a more authoritative reference. For example, they can be used to avoid storing items in a cache, if the item has not been previously accessed. That is, we use a Bloom filter to record item accesses, and only store an item if it has been seen at least once before. In this example, the consequence of false positives is just that a small number of unpopular items ends up being cached, which should have minimal impact on system performance. This exemplar is now found in practice in many large distributed databases, such as Google BigTable, Apache Cassandra and HBase. These systems keep a Bloom filter to index distributed segments of sparse tables of data. The filter records which rows or columns of the

table are stored, and can use a negative response to avoid a costly look-up on disk or over the network.

Another example is from web browsers, which aim to protect their users from malware by warning about deceptive or dangerous sites. This check requires looking up the web site address (URL) in a database of known problematic URLs, and reporting if a match is found. This database is sufficiently large that it is not convenient to keep locally on the (mobile) device. Instead, the browser can keep a Bloom filter encoding the database to check URLs against. Most URLs visited do not trigger the Bloom filter, and so do not affect the behavior. When the Bloom filter reports a potential positive, it can be checked against the centralized database with a look-up over the network. This slows things down, but so long as false positives are sufficiently rare then the impact is minimal. Thus, the correct response is found for each URL, while the common case (URL is allowed) is made fast. The space is kept low, based on a Bloom filter of a few megabytes in size. This approach has been adopted by browsers including Chrome and Firefox. Current versions have adopted variant summaries, which trade off a higher time for queries and reduced capacity to handle updates (since the sets are not updated more than once a day) for reduced size.

Other examples of Bloom filters in practice range from speeding up actions in blockchain implementations to providing private collection of browsing statistics [95]. Given these applications, it is likely that Bloom filter is the most widely used of the summaries discussed in this book, after "classical" summaries based on random sampling.

Available Implementations. Given its ubiquity, many implementations of the BloomFilter are available online, across a wide range of languages. The code hosting site GitHub lists over a thousand references to BloomFilter (`https://github.com/search?q=bloom+filter`), in languages including Java, JavaScript, C, C++, Go, Python and more. The implementation from stream-lib (`https://github.com/addthis/stream-lib/tree/master/src/main/java/com/clearspring/analytics/stream/membership\index{membership}`) takes around 100 lines of Java, although the core logic for UPDATE and QUERY are a handful of lines each.