3

Summaries for Multisets

This chapter turns to multisets, namely, each item in the set A to be summarized is associated with a multiplicity representing its number of occurrences, and queries to the multiset relate to the multiplicity of items. Recall from Section 1.2.1 that a convenient way to represent a multiset is to use a vector v, where v_x denotes the multiplicity of x. In this chapter, we assume that the input consists of (x, w) pairs, where x is an item drawn from a universe U, and w is the weight. For simplicity, we assume the weights are integers, but generalization to real numbers is often possible. Some summaries only allow positive weights, while others allow both positive and negative weights, where negative weights correspond to deletion of items from the multiset. The multiplicity of x, v_x, is thus the sum of all the weights of x in the input. We assume every v_x to be non-negative whenever the summary is queried.

Error guarantees for queries over a multiset often depend on $\|v\|_p$, the ℓ_p-norm of the vector v, where $\|v\|_p = \left(\sum_i v_i^p\right)^{1/p}$. The most commonly used norms are the ℓ_1-norm (which is simply the total weight of all items) and the ℓ_2-norm.

The summaries covered in this chapter solve the following problems:

- Testing whether two multisets contain the exact same set of items and frequencies: the Fingerprint summary (Section 3.1).
- Tracking the high frequency items over weighted positive updates: the MG and SpaceSaving summaries (Section 3.2 and 3.3).
- Estimating the frequencies of items over weighted positive and negative updates under different error guarantees: the Count-Min Sketch and Count Sketch summaries (Section 3.4 and 3.5).

- Estimating the Euclidean norm of the vector of frequencies of a mulitset: the AMS Sketch (Section 3.6).
- Estimating arbitrary Minkowski (ℓ_p) norms of a multiset: the ℓ_p sketch (Section 3.7).
- Exactly recovering the items and (integer) frequencies for a multiset that has a bounded number of elements after a sequence of insertions and deletions: the SparseRecovery summary (Section 3.8).
- Sampling near-uniformly from the set of distinct items in a multiset with (integer) frequencies: the ℓ_0-sampler structure (Section 3.9).
- Sampling near-uniformly from a multiset with arbitrary weights, according to a class of weighting functions: the ℓ_p-sampler structure (Section 3.10).

3.1 Fingerprints for testing multiset equality

Brief Summary. A Fingerprint summary represents a large multiset of items as a very compact hash value. Given two Fingerprint summaries, if the summaries are different then the corresponding multisets must differ, while if the summaries are identical then with high probability we conclude that the corresponding multisets are identical. The probability of a false positive – of erroneously concluding that two distinct sets are identical – is governed by the size of the summary, and can be made vanishingly small. Fingerprint summaries of two multisets can be combined to generate a Fingerprint of the *sum* of the multisets, i.e., where multiplicities of the same item are added.

Algorithm 3.1: Fingerprint: UPDATE (x, w)

1 $f \leftarrow (f + w\alpha^x) \mod p$;

Operations on the summary. For now we assume that the universe U from which items are drawn is the integer domain $U = [u] = \{0, 1, \ldots, u-1\}$. The Fingerprint summary can be thought of as a (possibly large) integer number f in the range 0 to $p-1$, where p is a fixed prime number greater than u. It also requires a randomly chosen value α. To INITIALIZE an empty summary, we fix a prime number $p > u$ and set f to 0. We also choose α uniformly at random in the range 1 to $p-1$. To UPDATE the summary with a new item x, we set $f \leftarrow (f + \alpha^x) \mod p$. If x has an associated weight w (which can be either positive or negative), then

this UPDATE becomes $f \leftarrow (f + w\alpha^x) \mod p$. Algorithm 3.1 shows how to UPDATE the Fingerprint summary with an item x of weight w.

Algorithm 3.2: Fingerprint: MERGE (f_a, f_b)

```
1 return (f_a + f_b) mod p;
```

To MERGE two Fingerprint summaries, f_a and f_b, with the same parameters (i.e., both have the same p and α values), the merged summary is given by $(f_a + f_b) \mod p$. Thus, the MERGE operation shown in Algorithm 3.2 simply has to return the sum, modulo p. As indicated above, to QUERY whether two summaries represent the same multiset, we report true if the corresponding summaries are identical, and false otherwise.

Example. For a simple example, we pick $p = 13$ and $\alpha = 3$. Given input of the item 4, we compute the Fingerprint summary as $3^4 \mod 13 = 3$. Consider the input consisting of the set $\{1, 2\}$. The Fingerprint summary of this input is $3^1 + 3^2 \mod 13 = 12$, which is different to the previous Fingerprint summary, as desired. However, the input $\{5, 10\}$ also has the Fingerprint summary $3^5 + 3^{10} \mod 13 = 12$, a collision. This is due in part to the small value of p: choosing larger p values makes such collisions increasingly unlikely.

Implementation Issues. Computing α^x can be done efficiently by the method of *exponentiation by squaring*, that is, we first compute $\alpha, \alpha^2, \alpha^4, \alpha^8, \ldots$, in succession, and then pick the right collection of terms to make up α^x, depending on the location of 1's in the binary representation of x. All computations are done modulo p, so that the intermediate results can be kept no larger than p^2.

Alternative Fingerprint Construction. If the weight w is usually small, there is another fingerprint construction that is more efficient. We similarly pick some prime $p > u$ and a random number α in the range 1 to $p - 1$. The operations, however, are slightly different. To INITIALIZE an empty summary, we set f to 1 (as opposed to 0). To UPDATE the summary with an item x with weight w, we set $f \leftarrow f \cdot (\alpha + x)^w \mod p$. Using the exponentiation by squaring technique, this fingerprint thus requires $O(\log w)$ time to update. In particular, if each update only adds a single item, this fingerprint can be updated in $O(1)$ time.

However, if w is negative (i.e., deleting items from the summary), things become a bit tricky as we need to set $f \leftarrow f \cdot (\alpha + x)^w \mod p$. Here,

the exponentiation should be done in the finite field $[p]$. More precisely, we first need to compute $(\alpha + x)^{-1}$, which is the value $y \in [p]$ such that $(\alpha+x)\cdot y \mod p = 1$. This can be done using Euclid's gcd algorithm [152] in $O(\log p)$ time. Then we use exponentiation by squaring to compute y^{-w}.

Finally, to MERGE two such fingerprints f_a and f_b, we simply compute $f_a \cdot f_b \mod p$.

Further Discussion. To understand the properties of the Fingerprint summary, consider the frequency vector representation of the multiset. For a frequency vector v, the Fingerprint summary is given by

$$f(v) = \sum_{i \in U} v_i \alpha^i \mod p.$$

The analysis of this summary relies on the fact that it can be viewed as a polynomial in α of degree at most u. Such a polynomial can have at most u roots (values of α where it evaluates to zero). Technically, we are evaluating this polynomial over the finite field $[p]$. Testing whether two multisets D and D' are equal, based on the fingerprints of their corresponding frequency vectors, $f(v)$ and $f(v')$, is equivalent to testing the identity $f(v) - f(v') = 0$. Based on the definition of f, if the two multisets are identical then the fingerprints will be identical. But if they are different and the test still passes, the fingerprint will give the wrong answer. Treating $f()$ as a polynomial in α, $f(v) - f(v')$ has degree no more than u: so there can only be u values of α for which $f(v) - f(v') = 0$. Effectively, the fingerprint is the evaluation of this polynomial at a randomly chosen value of α; it can only result in 0 when we are unlucky and choose an α which happens to be a root of this polynomial. But we can bound the probability of this event. Specifically, if p is chosen to be at least u/δ, the probability (based on choosing a random α) of having picked a root is at most δ, for a parameter δ. This requires the modular arithmetic operations to be done using $O(\log u + \log 1/\delta)$ bits of precision, which is feasible for most reasonable choices of U and δ. This analysis also assumes that the coefficients of the polynomial f are representable within the field, i.e., all v_i are less than p. This means that p should be chosen such that $p \geq \max\{u/\delta, v_i, i \in [u]\}$.

Viewing the Fingerprint summary through the lens of polynomials, it is then clear why the UPDATE and MERGE operations are as stated: an UPDATE computes the change in $f(v)$ caused by the change in v, while MERGE takes advantage of the fact that the Fingerprint of the sum of two vectors is the sum of the corresponding Fingerprint summaries.

The above analysis also applies to the second fingerprint, by observing that it evaluates the following polynomial

$$f(v) = \prod_{i \in U} (\alpha - i)^{v_i} \mod p.$$

This polynomial has degree $\sum_i v_i$. Thus if we choose $p \geq \sum_i v_i/\delta$, then the probability that this fingerprint fails is at most δ.

So far, we have assumed that the input item identifiers x are integers, since the argument is all based on evaluating polynomials over the finite field $[p]$. However, it is natural to allow x to be drawn from an arbitrary domain U. We just require a suitable hash function which will map U to a large enough integer domain. Ideally, the size of the integer domain should be polynomial in the number of distinct items seen. That is, if there are n different input items, then mapping to a domain of size n^3 means that with high probability there will be no hash collisions. This affects the choice of p, but only by a constant factor in terms of bit length.

The Fingerprint summary does require that all the item frequencies v_i are integral. Fractional values can be tolerated, if these can be rescaled to be integral. Arbitrary real values are not allowed, since these conflict with the analysis over finite fields.

History and Background. The idea of a compact hash function to test equality or inequality of sets or vectors has been used many times in computer science. Many different forms of hash function are possible, but we adopt this form to define the Fingerprint summary due to its ability to support both UPDATE and MERGE operations quite efficiently. Note that while these appear similar to the family of hash functions described in Section 1.4.2, they are distinct: in Section 1.4.2, we compute a polynomial of degree t, where the coefficients are chosen randomly (then fixed), and the variable x is the single data item; here, we compute a polynomial of potentially much higher degree, where the data determines the coefficients, and the variable x is chosen randomly (then fixed). The form of hash used here is inspired by the style of 'rolling

hash' function used in the Karp-Rabin string matching algorithm [150], and its generalization by Lipton [166]. Applications are broad: fingerprints can be used anywhere that a checksum would be useful, to check correct storage or transmission of data. They can also be applied to find large matching pieces of data in files that have changed, and so reduce the communication cost of sending the new version, such as in the rsync protocol [218], also used by systems like Dropbox and Amazon S3. Fingerprints are also used by some of other summary constructions, such as SparseRecovery (Section 3.8).

3.2 Misra-Gries (MG)

Brief Summary. The Misra-Gries (MG) summary maintains a subset of items from a multiset v, with an associated weight for each item stored. It answers point queries approximately: the answer to a query x is the weight associated with x in the summary, if x is stored, and 0 otherwise. Given a parameter ε, the summary stores $1/\varepsilon$ items and weights, and guarantees that for any point query x, the additive error from its true weight v_x is at most $\varepsilon\|v\|_1$. This summary only supports updates with positive weights.

Operations on the summary. The MG summary is represented as a collection of pairs: items drawn from the input x and associated weights w_x, which will be an approximation of its true weight v_x. To INITIALIZE an empty MG summary, it suffices to create an empty set of tuples, with space to store $k = 1/\varepsilon$. Each UPDATE operation of an item x with weight w (which must be positive) tries to include x in the summary. If x is already stored with weight w_x, then UPDATE increases this weight to $w_x + w$. If not, provided there is room in the summary (i.e., there are fewer than k distinct items stored with non-zero weights), then a new tuple (x, w) is added to the summary. Else, there are already k tuples in the summary. Let m be the smallest weight among all these tuples and w itself. Then, the UPDATE operation reduces all weights in the summary by m. If $w - m$ is still positive, then it can add the tuple $(x, w - m)$ to the summary: some tuple in the summary must have had its weight reduced to 0, and so can be overwritten with x.

Algorithm 3.3: MG: UPDATE$((x, w))$

```
1  if i ∈ T then
2  |  w_x ← w_x + w;
3  else
4  |  T ← T ∪ {x};
5  |  w_x ← w;
6  |  if |T| > k then
7  |  |  m = min({w_x : x ∈ T}) ;
8  |  |  forall j ∈ T do
9  |  |  |  w_j ← w_j − m;
10 |  |  |  if w_j = 0 then T ← T\{j} ;
```

Pseudocode to illustrate the UPDATE operation is given in Algorithm 3.3, making use of set notation to represent the operations on the set of stored items T: items are added and removed from this set using set union and set subtraction respectively, and we allow ranging over the members of this set (thus implementations will have to choose appropriate data structures which allow the efficient realization of these operations). We also assume that each item j stored in T has an associated weight w_j. For items not stored in T, we define w_j to be 0 and does not need to be explicitly stored. Lines 6 to 10 handle the case when we need to find the minimum weight item and decrease weights by this much. The inner for-loop performs this decrease, and removes items with weight 0.

The MERGE of two MG summaries is a generalization of the UPDATE operation. Given two summaries constructed using the same parameter k, first merge the component tuples in the natural way: If x is stored in both summaries, its merged weight is the sum of the weights in each input summary. If x is stored in only one of the summaries, it is also placed in the merged summary with the same weight. This produces a new MG summary with between k and $2k$ tuples, depending on the amount of overlap of items between the two input summaries. To reduce the size back to k, we sort the tuples by weight, and find the $k+1$st largest weight, w_{k+1}. This weight is subtracted from *all* tuples. At most k tuples can now have weight above zero: the tuple with the $k+1$st largest weight, and all tuples with smaller weight, will now have weight 0 or below, and so can be discarded from the summary. The above UPDATE procedure can therefore be seen as the case of MERGE where one of the summaries contains just a single item.

Algorithm 3.4: MG: MERGE(T_a, T_b)

```
1  T ← T_a;
2  forall j ∈ T_b do
3      if j ∈ T then w_j ← w_j + w_{b,j} ;
4      else
5          T ← T ∪ {j} ;
6          w_j ← w_{b,j};
7  if |T| > k then
8      w_{k+1} ← k + 1st largest weight ;
9      forall j ∈ T do
10         w_j ← w_j − w_{k+1} ;
11         if w_j ≤ 0 then T ← T \ {j};
```

Algorithm 3.4 shows the pseudocode to MERGE two MG summaries T_a and T_b together. Line 3 captures the case when j is present in both, and computes the new weight as the sum of the two weights. Lines 7–11 then reduce the size of the merged summary to k by reducing weights and removing items with non-positive weights.

To QUERY for the estimated weight of an item x, we look up whether x is stored in the summary. If so, QUERY reports the associated weight w_x as the estimate, otherwise the weight is assumed to be 0. Comparing the approximate answer given by QUERY, and the true weight of x (the sum of all weights associated with x in the input), the approximate answer is never more than the true answer. This is because the weight associated with x in the summary is the sum of all weights for x in the input, less the various decreases due to MERGE and UPDATE operations. The MG summary also ensures that this estimated weight is not more than εW below the true answer, where W is the sum of all input weights. A tighter guarantee provided is that this error is at most $(W - M)/(k + 1)$, where M is the sum of the counters in the structure, and k is the number of counters.

Example. Consider the input sequence

```
a, b, a, c, d, e, a, d, f, a, d
```

interpreted as a sequence of items of weight 1. If we UPDATE each item in turn into a MG summary of size k=3, we obtain at the end

a	d	–
2	1	0

There are two points where counts are decremented: on the UPDATE for the first `d`, and on the UPDATE for the `f`. Our final estimate of `a` is 2, which underestimates the true count by 2, which meets the bound of $(W - M)/(k + 1) = (11 - 3)/4 = 2$.

If we MERGE this summary with the MG summary

b	c	d
3	1	2

the final MG summary we obtain is

a	b	d
1	2	2

Further Discussion. To understand the guarantees of the MG summary, we study the effect of each operation in turn. First, consider a summary subject to a sequence of UPDATE operations alone. Let W be the sum of the input weights, and let M denote the sum of the weights stored by the summary. Consider the impact of decreasing the m value during an UPDATE operation. This impacts the weight of $k + 1$ items: the k items in the summary, and the new item that is the subject of the UPDATE. Consequently, we can "spread" the impact of this reduction across $k + 1$ items. The total impact is to increase the difference between W and M (the sum of stored weights) by $(k + 1) \cdot m$. Thus, at any point, the error in the estimated weight of an item is at most $(W - M)/(k + 1)$. This is because this difference, $(W - M)$, arises only from reductions in count to items during the UPDATE operations. Even if one item lost weight during all of these, the same weight loss was shared by k others (possibly different each time). So no item can suffer more than a $1/(k + 1)$ fraction of the total weight loss, $(W - M)$.

A similar argument holds in the case of MERGE operations. Let W_a and W_b denote the total weights of inputs summarized by the two MG summaries to be merged, and M_a, M_b represent the corresponding sum of weights stored in the summary. Following the MERGE operation, the new weight of the summary is M_{ab}, which at most the sum $M_a + M_b$. The MERGE operation means that the difference $(M_a + M_b) - M_{ab}$ is at least $(k+1)w_{k+1}$: we subtract w_{k+1} from the $k + 1$ largest weights (ignoring the impact on the smaller weights).

Rearranging, we have $w_{k+1} \leq \frac{1}{k+1}(M_a + M_b - M_{ab})$. The amount of additional error introduced by the MERGE into each estimated weight is at most w_{k+1}. If we assume (inductively) that the prior error from the two summaries is $(W_a - M_a)/(k+1)$ and $(W_b - M_b)/(k+1)$ respectively, then the new error is at most

$$\frac{1}{k+1}\left((W_a - M_a) + (W_b - M_b) + (M_a + M_b - M_{ab})\right)$$
$$= \frac{1}{k+1}(W_a + W_b - M_{ab}) = (W_{ab} - M_{ab})/(k+1).$$

Consequently, the MERGE operation also preserves the property that the query error is bounded by at most a $1/(k+1)$ fraction of the difference between the sum of true weights, and the sum of weights stored in the summary. Even in the worst case, when the weight of items stored in the summary is zero, this guarantees error of at most $W/(k+1) \leq \varepsilon W$. In general, the bound can be stronger. Let $W^{\mathrm{res}(t)}$ denote the *residual weight* of the input after removing the t heaviest items (as defined in Section 1.2.1). We can show a bound on the error in terms of $W^{\mathrm{res}(t)}$ for $t < k$. Let Δ denote the largest error in estimating the weight of any item. By the above analysis, we have that $\Delta \leq (W - M)/(k+1)$, and so $M \leq W - \Delta(k+1)$. Let w_i denote the (true) weight of the ith heaviest item. The estimated weight of this item is at least $w_i - \Delta$, so considering just the t heaviest items, $\sum_{i=1}^{t}(w_i - \Delta) \leq M$. Combining these two results, we have

$$\sum_{i=1}^{t}(w_i - \Delta) \leq W - \Delta(k+1).$$

Rearranging, we obtain

$$\Delta \leq W^{\mathrm{res}(t)}/(k+1-t).$$

In other words, for skewed distributions (where $W^{\mathrm{res}(t)}$ is small compared to W), the accuracy guarantee is stronger.

Implementation Issues. A limitation of the MG summary is that the weight stored for an item may be considerably lower than the true count, due to weight reductions caused by many other items. A simple adjustment which can improve the accuracy at the expense of increasing the space used, is to retain two weights for each item: the estimated weight as described above, and a second observed weight, which is increased in accordance with the procedures above, but which is never

decreased. This too is a lower bound on the true weight, but can be higher than the estimated weight that is used to determine when to retain the item. Note that this can still underestimate the true weight, since the item might be ejected from the summary at various points, and then the accrued weight of the item is lost.

The estimates provided are lower bounds. To turn these to upper bounds, we can add on a sufficient amount. It is straightforward to track W, the total weight of the whole input. Trivially, adding $W/(k+1)$ to any estimated count provides an upper bound over any input. We can give a better bound in some cases by using $(W - M)/(k+1)$ where M is the sum of the weights stored in the summary, as described above.

There has been much discussion in the research literature about how best to implement the MG summary to ensure that operations on the structure can be performed very quickly. When processing large sequences of UPDATE operations, we need to quickly determine whether x is currently stored in the structure (and if so, retrieve and modify its count); and also quickly find the minimum weight item, and reduce all items by this amount. The question of tracking whether an item is currently stored is a standard *dictionary* data structure question. It can be addressed deterministically, by keeping the current set of items T in a search-tree data structure, or via a randomized hash-table data structure. The former supports insertion, deletion and look up of items in worst case time $O(\log k)$, the latter supports these operations in expected time $O(1)$.

Tracking the minimum value and modifying all the others is a less standard step. The simplest solution is just to iterate among all the stored counts to find the minimum, and then a second time to reduce them by this amount. This however takes time $O(k)$, which would slow down UPDATE operations. Instead, we can optimize this step (at the cost of slowing down QUERY operations) by storing the items in sorted order of their weights, and representing their weights in terms of the *difference* in weight between the next heaviest item. That is, we store the lightest weight item first, then the next lightest and the amount by which it is heavier, and so on. To reduce the weights by the m value, it suffices to modify the weight of the lightest item, or remove it entirely. However, to insert a new item means stepping through this sorted list to find where the item belongs—which requires linear time $O(k)$ in the worst case. We can reduce this time cost, since we do not strictly require that the items are kept in sorted order of weight, only that we can find (and possibly remove) the item of least weight. As a result, it is possible

to adopt a min-heap data structure for this purpose. In the heap, we still store weights in terms of differences between the item and its parent in the heap. Using this representation, the heap operations to add a new item, modify the weight of an existing item, or remove the minimum weight item (and, implicitly, reduce the weights of the other items in the heap) can all be performed efficiently. In particular, these can all be done in time $O(\log k)$, proportional to the height of the heap. However, these are typically not supported in standard heap implementations, and so need to be implemented specially for this purpose. When we present the closely related SpaceSaving algorithm in Section 3.3, we will see how this summary can be implemented using an off-the-shelf heap structure.

In the special case when the updates are always 1 (so we are just counting the number of occurrences of each item), it is possible to reduce the UPDATE cost to constant time. In this case, we again represent the weight of items in terms of the difference between their weight and lighter items. However, we also group together all items which have the same weight, and arrange the groups in order of increasing weight. Now observe that when we increase the weight of an item, due to an arrival of weight 1, then the item must move from its current group, to the next group if this has a weight difference of 1, or to a new group in between the current group and the next group, if the weight difference to the next group is greater than 1. Working through all the stages in UPDATE, all these steps can be preformed in $O(1)$ time, if we maintain the items in groups via doubly-linked lists. However, the number of cases to handle is quite large, and many pointer operations are required. Careful use of hash tables and arrays can simplify this logic, and give a more compact representation [21].

History and Background. The history of this summary spans at least three decades. Initial interest arose from the question of finding the majority choice from a collection of votes. This problem and a candidate solution was described by Boyer and Moore [33]. The technical core of the solution is essentially the MG summary with $k = 1$ and all input weights set to 1, which therefore finds if there is one item that occurs more than half the time (a strict majority).

A generalization of this approach to process sequences of items with unit weight, and find all those that occur more than a $1/k$ fraction of the time, was first proposed by Misra and Gries [176]. The time cost of their algorithm is dominated by the $O(1)$ dictionary operations per up-

date, and the cost of decrementing counts. Misra and Gries described the use of a balanced search tree, and argued that the decrement cost is amortized $O(1)$. Refinements of this approach emerged two decades later, due to renewed interest in efficient algorithms for processing large streams of data. Karp *et al.* proposed a hash table to implement the dictionary [149]; and Demaine *et al.* show how the cost of decrementing can be made worst case $O(1)$ by representing the counts using offsets and maintaining multiple linked lists [81]. Bose *et al.* [32] observed that executing this algorithm with $k = 1/\epsilon$ ensures that the count associated with each item on termination is at most ϵW below the true value.

The extension to allow a MERGE operation, in addition to UPDATE, was made more recently [25], where the bounds in terms of $W^{\text{res}(k)}$ were also shown. The strong bounds on the effect of MERGE are due to Agarwal *et al.* [2]. Additional historical context for this summary is provided in [64]. An optimized implementation of MG is discussed by Anderson *et al.* [12], with particular attention paid to handling weighted updates and MERGE operations efficiently.

Available Implementations. The MG summary forms the basis of the Frequent Items implementation within the DataSketches library, `https://datasketches.github.io/docs/Frequency/FrequentItemsOverview.html`. Items within the summary are stored in a hash table, and the compression of the summary to decrement counts is performed with a linear pass through the current structure. Optimizations due to Anderson *et al.* [12] are included. These ensure that the implementation is easily capable of processing over ten million UPDATE operations per second.

3.3 SpaceSaving

Brief Summary. The SpaceSaving summary retains a subset of items from a multiset v, with an associated weight for each. It answers point queries approximately: the answer to a query x is the weight associated with x in the summary, if x is stored, and 0 otherwise. Given a parameter ε, the summary stores $1/\varepsilon$ items and counts, and answers point queries with additive error at most $\varepsilon\|v\|_1$. It only supports updates with positive weights. Conceptually, the SpaceSaving summary is very

Algorithm 3.5: SpaceSaving:UPDATE (x, w)

```
1 if x ∈ T then
2 |   w_x ← w_x + w;
3 else
4 |   y ← arg min_{j∈T} w_j;
5 |   w_x ← w_y + w;
6 |   T ← T ∪ {x}\{y};
```

similar to the MG summary (Section 3.2), with a few operational differences, and in fact the two structures can be considered almost identical.

Operations on the summary. Like the MG summary, the SpaceSaving summary is represented as a collection of pairs of items drawn from the input x, and associated weights w_x. To INITIALIZE an empty summary, an empty set of $k = 1/\varepsilon$ tuples is created. It is sometimes convenient to think of these tuples being initialized to k distinct (dummy) values, each with associated count zero.

Each UPDATE operation of an item x with weight w tries to include x in the summary. If x is already stored with weight w_x, the UPDATE increases this weight to $w_x + w$. Otherwise, the operation identifies the stored tuple with the smallest weight, say y, and replaces it with the new item x. The corresponding weight is set to $w_y + w$. When there are multiple items stored in the summary with the same, least weight, an arbitrary one can be selected for replacement.

Algorithm 3.5 presents pseudocode for the UPDATE operation, where the set T holds the currently monitored items. This assumes that the summary is initialized with k 'dummy' entries, so that each UPDATE preserves the size of the summary at k elements.

A MERGE operation is quite simple, and proceeds in the natural way, given two SpaceSaving summaries of the same size. The merged summary initially contains all items which occur in either of the input summaries. If an item x is stored in both summaries, then its weight in the new summary is the sum of its weights in the previous summaries. Otherwise, its weight is the weight from whichever summary it was stored in. The intermediate result may have more than k tuples, so to reach a summary of bounded size, we retain only the k tuples with the largest weights (breaking any ties arbitrarily). Algorithm 3.6 shows the MERGE operation for the SpaceSaving summary. A subtlety not ex-

Algorithm 3.6: SpaceSaving: MERGE(T_a, T_b)

```
1  T ← T_a;
2  forall j ∈ T_b do
3      if j ∈ T then w_j ← w_j + w_{b,j} ;
4      else
5          T ← T ∪ {j} ;
6          w_j ← w_{b,j};
7  if |T| > k then
8      w_{k+1} ← k + 1st largest weight ;
9      forall j ∈ T do
10         if w_j ≤ w_{k+1} then T ← T \ {j};
```

plicitly addressed in the pseudocode is when multiple items share the same weight w_{k+1}, the algorithm should remove just the right number of them so that the resulting size of the summary is k.

To QUERY for the estimated weight of an item x, we look up whether x is stored in the SpaceSaving summary. If so, QUERY reports the associated weight as the estimate. This approximate answer is an overestimate of the true weight, as explained within the further discussion section. If x is not stored in the summary, then the QUERY operation can report the smallest weight value stored in the summary, n, as an upper bound on the true weight, or 0 as a lower bound on the true weight.

Example. Consider (again) the input sequence

```
a, b, a, c, d, e, a, d, f, a, d
```

interpreted as a sequence of items of weight 1. If we UPDATE each item in turn into a SpaceSaving summary of size k=4, we may obtain at the end

a	d	e	f
4	3	2	2

Other instances of the summary are possible, depending on how we break ties when there are multiple items achieving the smallest frequency. For concreteness, in this example, we break ties by overwriting the lexicographically smallest item.

Further Discussion. An important observation which simplifies the study of this summary is that, despite differences in the description of the operations, it is essentially identical to the MG summary (Section 3.2). More precisely, an MG summary with size parameter k and a SpaceSaving summary with size parameter $k + 1$, subject to the same sequence of UPDATE and MERGE operations over the same input, contain the same information. The correspondence between the two summaries is that the estimated weight of x in the MG summary is the estimated weight of x in the SpaceSaving summary, less the weight of the smallest item in the summary, which we denote by n. For instance, comparing the example (SpaceSaving with $k = 4$), with the corresponding example in the previous section (MG with $k = 3$), we see that after subtracting the minimum value of 2 from the SpaceSaving example, we retain `a` with weight 2, and `d` with weight 1, the same information as in the MG examples.

The proof that this holds in general proceeds inductively. The correspondence is certainly true initially, since all estimated weights in both summaries are zero. Now consider the effect of an UPDATE operation when the minimum value is denoted n, assuming that the correspondence holds prior to this operation. If x is stored in both summaries, then the correspondence holds after the UPDATE, since both estimated weights increase by w. If x is not stored in the MG summary, but there is room to store it, then its old estimated weight was 0 and its new estimated weight is w. This means its estimated weight in SpaceSaving is n. Then either it is stored in the SpaceSaving summary with weight n, or it is not stored, and some item with weight n is overwritten. Either way, its new estimated weight becomes $n + w$, preserving the correspondence. Lastly, if x is not stored in the MG summary, and there is no room to store it, then all the estimated weights there are reduced by the amount m. Using the assumed correspondence, this means that there is a unique item with weight n in the SpaceSaving summary, which is overwritten by x, and whose new weight becomes $n + w$. Note that this has the effect of changing the smallest weight in the SpaceSaving summary to either $n + w$ or the smallest of the other estimated weights. Thus, the amount deducted from the counts in MG, m, is

exactly the amount by which the new smallest count in SpaceSaving differs from the old smallest count. Hence, the correspondence is preserved. By similar case-based reasoning, it follows that the MERGE operations also preserve this correspondence.

Thus the information provided by both summaries is essentially the same. The upper bound estimate stored by SpaceSaving can be transformed to the lower bound estimate, by subtracting n, the smallest of the stored counts.

Implementation Issues. The duality between the SpaceSaving and MG structures means that the same implementation issues affect both. However, the form of the SpaceSaving summary means that it is clearer to see an efficient way to process updates to the summary. Specifically, for an UPDATE we now only need to be able to find the smallest weighted item. This can be done using a standard min-heap data structure, with operations taking time $O(\log k)$. Merge operations requires summing up to k corresponding weights, and merging the structures, and so requires time $O(k)$.

History and Background. The SpaceSaving summary was first proposed as such by Metwally *et al.* for unit weight updates [175]. The generalization to weighted updates was given later by [25]. The observation of the correspondence between SpaceSaving and MG was made in [2], and used to provide an efficient MERGE operation. Efficient implementations of SpaceSaving with optimizations to allow constant time (amortized or worst-case) UPDATE operations is given by Basat *et al.* [23, 22]. These are based on using the (approximate) median weight element in the data structure to prune elements, similar to the concurrent work of Anderson *et al.* [12] – see the historical notes on the MG summary in the previous section. Parallel implementations have also been considered [76]. The summary was introduced in the context of counting clicks within Internet advertising, and more generally applies where we want to count popular items out of a very large number of possibilities.

3.4 Count-Min Sketch for frequency estimation

Brief Summary. Like the MG and the SpaceSaving summary, the Count-Min Sketch summary also summarizes a multiset v and answers point

queries with error at most $\varepsilon\|v\|_1$. But it is a linear sketch, in the sense that it can be written as the action of matrix on a vector (expanded in more detail in Section 9.3.4). It thus supports updates with both positive and negative weights. The Count-Min Sketch summary maintains an array of counts, which are updated with each arriving item. It answers point queries approximately by probing the counts associated with the item, and picking the minimum of these. It is a randomized algorithm. Given parameters ε and δ, the summary uses space $O(1/\varepsilon \log 1/\delta)$, and guarantees with probability at least $1 - \delta$ that any point query is answered with additive error at most $\varepsilon\|v\|_1$. A variant of the summary uses $O(1/\varepsilon^2 \log 1/\delta)$ space and has error at most $\varepsilon\|v\|_2$ with probability at least $1 - \delta$.

Operations on the summary. The Count-Min Sketch summary is represented as a compact array C of $d \times t$ counters, arranged as d rows of length t. For each row a hash function h_j maps the input domain U uniformly onto the range $\{1, 2, \ldots, t\}$. To INITIALIZE a new summary, the array of counters is created based on the parameters t and d, and every entry is set to 0. At the same time, a set of d hash functions is picked from a pairwise independent family (see Section 1.4.2). Algorithm 3.7 shows how to INITIALIZE the summary given parameters t and d, and randomly chooses the d hash functions (based on a suitable prime p).

Algorithm 3.7: Count-Min Sketch: INITIALIZE (t, d, p)

```
1 C[1,1] ... C[d,t] ← 0;
2 for j ← 1 to d do
3     Pick a_j, b_j uniformly from [1 ... p];
```

For each UPDATE operation to item i with weight w (which can be either positive or negative), the item is mapped to an entry in each row based on the hash functions, and the update applied to the corresponding counter. That is, for each $1 \le j \le d$, $h_j(i)$ is computed, and w is added to entry $C[j, h_j(i)]$ in the sketch array. Processing each update therefore takes time $O(d)$, since each hash function evaluation takes constant time. The pseudocode for UPDATE computes the hash function for each row of the update i, and updates the corresponding counter with the weight w (Algorithm 3.8).

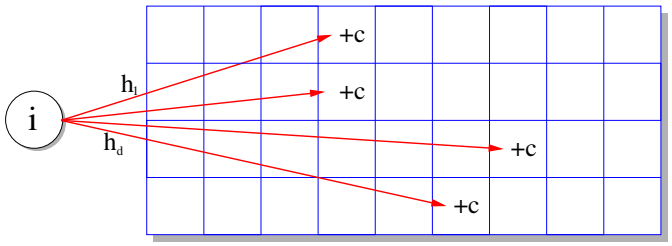

Figure 3.1 Count-Min sketch data structure with $t = 9$ and $d = 4$

Algorithm 3.8: Count-Min Sketch: UPDATE (i, w)

```
1 for j ← 1 to d do
2     h_j(i) = (a_j × i + b_j  mod p)  mod t ;
3     C[j, h_j(i)] ← C[j, h_j(i)] + w;
```

The Count-Min Sketch gets its name due to the two main operations used during a QUERY operation: counting of groups of items, and taking the minimum of various counts to produce an estimate. Specifically, we find all the locations where the queried item x is mapped, and retrieve the count from each such location. In the basic version of the summary, the estimated count is just the smallest of these counts. So the QUERY time is the same as the UPDATE time, $O(d)$. The procedure for QUERY in Algorithm 3.9 is quite similar to that for UPDATE: again, there is an iteration over the locations where x is mapped. A value e is maintained as the smallest of the values encountered, and returned as the final estimate.

Algorithm 3.9: Count-Min Sketch: QUERY (x)

```
1 e ← ∞;
2 for j ← 1 to d do
3     h_j(x) = (a_j × x + b_j  mod p)  mod t ;
4     e ← min(e, C[j, h_j(x)]) ;
5 return e
```

To MERGE two Count-Min Sketch summaries, they must be built using the same parameters. That is, they must share the same t and d values, and use the same set of d hash functions. If this is the case, then we can merge two summaries directly by summing the corresponding entries in the arrays.

Example. Figure 3.1 shows the UPDATE process: an item i is mapped to one entry in each row j by the hash function h_j, and the update of c is added to each entry. It can also be seen as modeling the QUERY process: a query for the same item i will result in the same set of locations being probed, and the smallest value returned as the answer.

Further Discussion. Let v denote the characteristic vector of the multiset, so v_i is the total weight of all updates to entry i. The entries of the sketch C then obey

$$C[j,k] = \sum_{1 \le i \le M : h_j(i) = k} v_i$$

That is, the kth entry in the jth row is the sum of frequencies of all items i which are mapped by the jth hash function to value k.

The effect of QUERY is to recover an estimate of v_i, for any i. Observe that for it to be worth keeping a sketch in place of simply storing v exactly, it must be that td is much smaller than the size of the input domain, U, and so the sketch will necessarily only approximate any v_i. The estimation can be understood as follows: in the first row, it is the case that $C[1, h_1(i)]$ includes the current value of v_i. However, since $t \ll U$, there will be many collisions under the hash function h_1, so that $C[1, h_1(i)]$ also contains the sum of all v_ℓ for ℓ that collides with i under h_1. Still, if the sum of such v_ℓs is not too large, then this will not be so far from v_i. In fact, we can state a bound in terms of $W = \sum_{1 \le \ell \le M} v_\ell$, the sum of all frequencies.

Fact 3.1 *The error in the estimate of v_i from a* Count-Min Sketch *is at most εW with probability $1 - \delta$ for a sketch with parameters $t = 2/\varepsilon$ and $d = \log 1/\delta$.*

We demonstrate this fact under the assumption that every entry in v is non-negative (below, we discuss the case when this is not so). Then $C[1, h_1(i)]$ is an overestimate for v_i. The same is true for all the other rows: for each j, $C[j, h_j(i)]$ gives an overestimate of v_i, based on a different set of colliding items. Now, if the hash functions are chosen at random, the items will be distributed uniformly over the row. So the expected amount of "noise" colliding with i in any given row is just $\sum_{1 \le \ell \le U, \ell \ne i} v_\ell / t$, a $1/t$ fraction of the total weight W. Moreover, by the Markov inequality (Fact 1.1 in

Section 1.4), there is at least a 50% chance that the noise is less than twice this much.

Expressing this in probability terms, we have

$$\Pr\left[C[j, h_j(i)] - v_i > \frac{2W}{t}\right] \leq \frac{1}{2}$$

Here, the probabilities arise due to the random choice of the hash functions. If each row's estimate of v_i is an overestimate, then the smallest of these will be the closest to v_i. By the independence of the hash functions, it is now very unlikely that this estimate has error more than $2\sum_{1\leq\ell\leq U} v_\ell/t$: this only happens if *every* row estimate is "bad", which happens with probability at most 2^{-d}. That is,

$$\Pr\left[\min_j C[j, h_j(i)] - v_i > \frac{2W}{t}\right] \leq \left(\frac{1}{2}\right)^d$$

Rewriting this, if we pick $t = 2/\epsilon$ and $d = \log 1/\delta$, then our estimate of v_i has error at most ϵW with probability at least $1 - \delta$. The value returned by QUERY is simply $\min_{j=1}^{d} C[j, h_j(i)]$.

For this analysis to hold, the hash functions are required to be drawn from a family of pairwise independent functions (Section 1.4.2). That is, over the random choice of the hash functions, for any i, ℓ and values x, y, $\Pr[h_j(i) = x \wedge h_j(\ell) = y] = \frac{1}{t}$. However this turns out to be quite a weak condition: such functions are very simple to construct, and can be evaluated very quickly indeed [44, 212]. Hash functions of the form $((ax + b) \mod p) \mod t$ where p is a prime, and a, b are chosen randomly in the range 1 to $p - 1$ meet these requirements.

Residual bound. The above analysis can be tightened to give a bound in terms of $\|v\|_1^{\text{res}\,k}$, the residual L_1 norm after removing the k largest entries. Consider the estimate of v_i for a row j. With probability $1 - 1/t$, the heaviest item is not mapped to $h_j(i)$. Taking a union bound over the k heaviest items, they all avoid $h_j(i)$ with probability $1 - k/t$. Picking $k = t/8$, say, ensures that this holds with constant probability. Conditioned on the event that all k avoid colliding with i, then we can apply the same expectation-based argument on the weight of the other colliding items, and show that this is $\|v\|_1^{\text{res}(k)}/t$ in expectation. Therefore, with constant probability, we have that none of the k heaviest items collide, and that the weight of items

that do collide is at most a constant times $\|v\|_1^{\text{res}(k)}/t$. Repeating over the d hash functions and taking the minimum estimate improves the probability of getting an estimate within $\varepsilon\|v\|_1^{\text{res}(k)}$ to $1-\delta$.

Unbiased estimator. The estimator $C[j, h_j(i)]$ is technically *biased*, in the statistical sense: it never underestimates but may overestimate, and so is not correct in expectation. However, it is straightforward to modify the estimator to be unbiased, by subtracting an appropriate quantity from the estimate. Specifically, we modify the QUERY procedure to compute

$$\hat{v}_{i,j} = C[j, h_j(i)] - \frac{1}{t-1}\sum_{k \neq h_j(i)} C[j,k] = \frac{tC[j, h_j(i)] - W}{t-1},$$

where W is the total weight of all updates. Since we have that the expectation of $C[j, h_j(i)] = v_i + (W - v_i)/t$, it follows that the expectation of this quantity is v_i, i.e., it is an unbiased estimator.

However, there is no way to preserve this unbiasedness across the d rows: ideas such as taking the median of estimators turn out not to work. One workaround is to convert the Count-Min Sketch to a Count Sketch, which produces an unbiased estimator across multiple rows — this is covered when we discuss the Count Sketch structure in Section 3.5.

L_2 bound. The Count-Min Sketch can also produce an estimator whose error depends on $\|v\|_2$, the ℓ_2-norm of v. We use the unbiased estimator $\hat{v}_{i,j}$ as described above. Its variance can be bounded as follows, using standard properties of the variance (Section 1.4):

$$\begin{aligned}
\mathsf{Var}\left[\frac{tC[j, h_j(i)] - W}{t-1}\right] &= \mathsf{Var}\left[\frac{t}{t-1}C[j, h_j(i)]\right] \\
&= \left(\frac{t}{t-1}\right)^2 \mathsf{Var}\left[v_i + \sum_{k \neq i, h_j(k) = h_j(i)} v_k\right] \\
&= \left(\frac{t}{t-1}\right)^2 \sum_{k \neq i} \mathsf{Var}[v_k I(h_j(k) = h_j(i))] \\
&= \left(\frac{t}{t-1}\right)^2 \sum_{k \neq i} v_k^2 \left(\frac{1}{t}\right)\left(1 - \frac{1}{t}\right) \\
&= \frac{1}{t-1}\sum_{k \neq i} v_k^2 \leq \frac{\|v\|_2^2}{t-1},
\end{aligned}$$

where $I(E)$ is an indicator random variable for the event E. This analysis relies on the fact that whether two items are mapped to $h_j(i)$ can be treated as independent, over the choice of the hash function, and so the covariance of corresponding random variables is zero. We can apply the Chebyshev bound to this estimator (Fact 1.3 from Section 1.4), and obtain

$$\Pr\left[|\hat{v}_{i,j} - v_i| > \frac{2\|v\|_2}{\sqrt{t-1}}\right] \leq \frac{1}{4},$$

i.e., the estimate is within two standard deviations from its expectation, which is v_i, with probability at least 3/4. Then we take the median of all the estimates (one from each row of the sketch), and return it as the final estimate. Note that the median of multiple unbiased estimators is not necessarily unbiased, but we can still obtain an (ε, δ) guarantee via the standard Chernoff bound argument (Section 1.4.1):

Fact 3.2 *The error in the (unbiased) estimate of v_i from a* Count-Min Sketch *with parameters $t = O(1/\varepsilon^2)$ and $d = O(\log 1/\delta)$ is at most $\varepsilon\|v\|_2$ with probability at least $1 - \delta$.*

To better contrast the two forms of guarantees from Fact 3.1 and Fact 3.2, we may rewrite Fact 3.2 by substituting ε with $\sqrt{\varepsilon}$, so that the sketch size is the same $t = O(1/\varepsilon), d = O(\log 1/\delta)$. Then the L_2 error bound becomes $\sqrt{\varepsilon}\|v\|_2$. Note that this is in general incomparable to the L_1 error bound of $\varepsilon\|v\|_1$. When the frequency vector v is highly skewed, say with only one or two nonzero entries, then $\|v\|_1 = \|v\|_2$ and the L_1 bound is better. But when v is quite flat, say $v_1 = v_2 = \cdots = 1$, then $\|v_2\| = \sqrt{\|v\|_1} \ll \|v\|_1$, and the L_2 bound will be much better.

Negative frequencies. The same sketch can also be used when the items have both positive and negative frequencies when the sketch is queried. In this case, the sketch can be built in the same way, but now it is not correct to take the smallest row estimate as the overall estimate: this could be far from the true value if, for example, all the v_i values are negative. However, an adaptation of the above argument shows that the error still remains bounded. Consider instead the sum of the *absolute* values of items that collide with i when estimating v_i. This is a positive quantity, and so can still be bounded

by the Markov inequality (Fact 1.1) to be at most $\|v\|_1/t$ with constant probability, where $\|v\|_1$ denotes the 1-norm (sum of absolute values) of v. If this is the case, then the error in the estimate must lie between $-\|v\|_1/t$ (if all the colliding items are negative) and $+\|v\|_1/t$ (if all the colliding items are positive). So with constant probability, each estimate of v_i is within $2\|v\|_1/t$ additive error. To improve this to hold with high probability, we can take the *median* of the row estimates, and apply the Chernoff bounds argument described in Section 1.4 to obtain an accurate estimate. Then it follows that:

Fact 3.3 *The error in the estimate of v_i from a* Count-Min Sketch *over a multiset with both positive and negative frequencies is at most $\varepsilon\|v\|_1$ with probability $1-\delta$, for a sketch with parameters $t = O(1/\varepsilon)$ and $d = O(\log 1/\delta)$.*

Conservative update. The *conservative update* method can be applied on a Count-Min Sketch when there are many UPDATE operations in sequence. It tries to minimize overestimation by increasing the counters by the smallest amount possible given the information available. However, in doing so it breaks the property that the summary is a linear transform of the input, so it can no longer support updates with negative weights. Consider an update to item i in a Count-Min Sketch. The update function maps i to a set of entries in the sketch. The current estimate of v_i is given by the least of these, as $\hat{v}_i$: this has to increase by at least the amount of the update u to maintain the accuracy guarantee. But if other entries are larger than $\hat{v}_i + u$, then they do not need to be increased to ensure that the estimate is correct. So the conservative update rule is to set

$$C[j, h_j(i)] \leftarrow \max(\hat{v}_i + u, C[j, h_j(i)])$$

for each row j. The MERGE and QUERY operations under conservative update remain the same. However, to enjoy the maximum benefit from conservative update, we must expect a large amount of UPDATE operations: if the sketch is obtained from mostly MERGE operations rather than UPDATE operations, then the final sketch will be no different to the sketch obtained under the usual UPDATE operation.

History and Background. The central concept of the Count-Min Sketch is quite fundamental, and variations of the idea have appeared in sev-

eral contexts. Variations of Bloom Filters (see Section 2.7) have used the idea of hashing to counters to deal with set updates [97] and more specifically for count estimation [56]. In the context of networking, Estan and Varghese's multistage filters are a data structure which correspond to sketches, under assumptions of fully-independent hash functions [96]. This work pioneered the idea of conservative update. The sketch was formally introduced with strong guarantees from weak hash functions in 2003 [69, 72], building on earlier work on retrieving heavy hitter items [70]. Kirsch and Mitzenmacher discuss how the hashing can be made faster, by performing arithmetic on combinations of hash values [151].

Several variations and extensions have been suggested. The approach of taking the minimum value as the estimate from Count-Min Sketch is appealing for its simplicity. But it is also open to criticism: it does not take full account of all the information available to the estimator. Lee *et al.* studied using a least-squares method to recover estimated frequencies of a subset of items from a Count-Min Sketch [161]. That is, using the fact that the sketch is a linear transform of the input, write the sketch as a multiplication between a version of the sketch matrix and a vector of the frequencies of the items of interest. Lu *et al.* use Message Passing, which also tries to find a distribution of counts which is consistent with the values recorded in the sketch of the observed data [167]. Jin *et al.* empirically measure the accuracy of an instance of a Count-Min sketch [141]. They estimate the frequency of some items which are known to have zero count, say $U + 1, U + 2 \ldots$ etc. The average of these estimates is used as τ, the expected error, and all estimated counts are reduced by τ. This was sometimes referred to as "Count-Mean-Min" (CMM), by [82, 50]. Bianchi *et al.* [27] and Einziger and Friedman [92] both give an analysis of the conservative update method. A line of subsequent work has studied tighter bounds on estimation for Count-Min Sketch. Ting [215] provides new estimators with tight error bounds; Cai *et al.* [43] adopt a Bayesian perspective to analyze the sketch.

The sketch has been used for a variety of different tasks. Some examples include counting the frequency of particular strings in long genetic sequences [233]; tracking the popularity of different passwords, and warning if a chosen password is too common [200]; and a general technique to speed up machine learning in high-dimensional feature spaces [203]. The sketch has been used by Twitter to track the popularity of individual tweets across the web [28], and by Apple to reduce the size of information gathered under its private data collection tool [211].

Available Implementations. Many implementations of the Count-Min Sketch are available. Typically these are quite simple and straightforward: one example `https://github.com/rafacarrascosa/countminsketch` implements all methods in a few dozen lines of python. Other implementations listed on GitHub implement in C, C++, Java, JavaScript, Go, Juia and Clojure. The stream-lib implementation (`https://github.com/addthis/stream-lib/tree/master/src/main/java/com/clearspring/analytics/stream/frequency`) includes both the basic and conservative update variants.

3.5 Count Sketch for frequency estimation

Brief Summary. The Count Sketch summary is very similar to Count-Min Sketch. It also summarizes a multiset v and answers point queries approximately. It has asymptotically the same guarantees as the Count-Min Sketch, but with slightly worse hidden constant factors: Given parameters ε and δ, the summary uses space $O(1/\varepsilon \log 1/\delta)$, and guarantees with probability at least $1-\delta$ that any point query is answered with additive error at most $\varepsilon\|v\|_1$. Alternatively, with space $O(1/\varepsilon^2 \log 1/\delta)$, it has error at most $\varepsilon\|v\|_2$ with probability at least $1-\delta$. The benefit of the Count Sketch is that it produces an unbiased estimator, which is important in certain applications (see, for example, Section 4.4).

Algorithm 3.10: Count Sketch: INITIALIZE (t, d, p)

```
1 C[1,1] ... C[d,t] ← 0;
2 for j ← 1 to d do
3 |   Pick a_j, b_j uniformly from [1 ... p];
4 |   Pick c_j, d_j uniformly from [1 ... p];
```

Operations on the summary. The Count Sketch summary is represented as a compact array C of $d \times t$ counters, arranged as d rows of length t. It can be described as being similar in nature to the Count-Min Sketch summary, but with some key differences. For each row j a *hash function* h_j maps the input domain U uniformly onto the range $\{1, 2, \ldots, t\}$. A second hash function g_j maps the input domain U uniformly on the range $\{-1, +1\}$. To INITIALIZE a new summary, the array

of counters is created based on the parameters t and d, and every entry is set to 0. The two sets of d hash functions, g and h are picked at the same time. Algorithm 3.10 shows the procedure to INITIALIZE a Count Sketch: the array is set to zero, and the parameters for the hash function are chosen randomly based on the prime p.

Algorithm 3.11: Count Sketch: UPDATE (i, w)

```
1 for j ← 1 to d do
2     h_j(i) = (a_j × i + b_j mod p) mod t ;
3     g_j(i) = 2 × ((c_j × i + d_j mod p) mod 2) − 1 ;
4     C[j, h_j(i)] ← C[j, h_j(i)] + w × g_j(i);
```

For each UPDATE operation to item i with weight w (which can be either positive or negative), the item is mapped to an entry in each row based on the hash functions h, and the update applied to the corresponding counter, multiplied by the corresponding value of g. That is, for each $1 \leq j \leq d$, $h_j(i)$ is computed, and the quantity $wg_j(i)$ is added to entry $C[j, h_j(i)]$ in the sketch array. Processing each update therefore takes time $O(d)$, since each hash function evaluation takes constant time. Each UPDATE shown in Algorithm 3.11 computes $h_j(i)$ and $g_j(i)$ for each row j, and updates the corresponding entry of the array.

Algorithm 3.12: Count Sketch: QUERY (x)

```
1 for j ← 1 to d do
2     h_j(x) = (a_j × x + b_j mod p) mod t ;
3     g_j(x) = 2 × ((c_j × x + d_j mod p) mod 2) − 1 ;
4     e_j ← g_j(x) × C[j, h_j(x)]) ;
5 return median(e)
```

The QUERY operation on item x extracts the count associated with x in each row, and returns the median of these. That is, in row j it computes $g_j(i)C[j, h_j(i)]$ as the estimate for that row. This generates d estimates, and the median of these is the final estimate. Therefore, the QUERY time is also $O(d)$. For a QUERY operation in Algorithm 3.12, the procedure is quite similar to an UPDATE. A vector of estimates e is built, and the median of this vector is returned as the final answer.

To MERGE two Count Sketch summaries, they must have the same parameters, i.e., they must have the same t and d values, and use the same sets of hash functions h and g. Then they can be merged by summing the corresponding entries in the arrays. This works since the re-

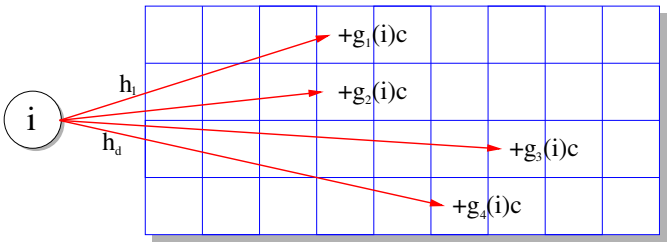

Figure 3.2 Count Sketch data structure with $t = 9$ and $d = 4$

sulting sketch is identical to the one obtained if all UPDATE operations had been made to a single sketch.

Example. Figure 3.2 shows an example of the Count Sketch: during an UPDATE operation, an item is mapped into one entry in each row by the relevant hash function, and multiplied by a second hash function g. The figure serves to emphasize the similarities between the Count Sketch and Count-Min Sketch: the main difference arises in the use of the g_j functions.

Further Discussion. As with the Count-Min Sketch summary, the accuracy of the estimate obtained is analyzed by considering the distribution of the "noise" of other items that affect the estimate. This is shown to be very likely to be small, under the random choice of the hash functions. Let v denote the characteristic vector of the multiset summarized, so that v_i is the total weight of all update to entry i. Then we can write the sketch C as

$$C[j,k] = \sum_{1 \le i \le M : h_j(i) = k} g_j(i) v_i.$$

So the kth entry in the jth row is a sum of frequencies of items i mapped to k by the jth hash function h_j, where each is multiplied by either $+1$ or -1, according to g_j. The intuition is that by hashing to different entries, i is unlikely to collide with too much "weight" from other items. Further, the use of g_j functions is intended to help the colliding items to 'cancel' out with each other.

L_1 error bound. Formally, let $\hat{v}_{i,j} = g_j(i)C[j, h_j(i)]$ be the estimator in

the j-th row for v_i. We consider the random variable $|v_i - \hat{v}_{i,j}|$, which is the absolute error in the estimator. Its expectation is

$$\begin{aligned}
\mathsf{E}[|v_i - \hat{v}_{i,j}|] &= \mathsf{E}\left[\left|\sum_{\ell \neq i} I(h_j(\ell) = h_j(i)) \cdot v_\ell \cdot g_j(i) \cdot g_j(\ell)\right|\right] \\
&\leq \sum_{\ell \neq i} \mathsf{E}[|I(h_j(\ell) = h_j(i)) \cdot v_\ell \cdot g_j(i) \cdot g_j(\ell)|] \\
&= \sum_{\ell \neq i} |v_\ell| \cdot \mathsf{E}[I(h_j(\ell) = h_j(i))] \\
&= \frac{1}{t} \sum_{\ell \neq i} |v_\ell| \leq \|v\|_1 / t.
\end{aligned}$$

Applying the Markov inequality to the (non-negative) variable $|v_i - \hat{v}_{i,j}|$, we have that the error in this estimate is at most $3\|v\|_1/t$ with probability at least $2/3$. Taking the median of $O(\log 1/\delta)$ repetitions reduces this failure probability to δ, via a Chernoff bounds argument. So we have an L_1 error bound for the Count Sketch that is asymptotically the same as the Count-Min Sketch:

Fact 3.4 *The error in the estimate of v_i from a* Count Sketch *is at most $\epsilon\|v\|_1$ with probability $1 - \delta$ from a sketch with parameters $t = O(\frac{1}{\epsilon})$ and $d = O(\log 1/\delta)$.*

Although the Count Sketch has asymptotically the same L_1 error guarantee as the Count-Min Sketch, the hidden constants are worse. This is due to the use of the *median* operator to select one out of the d rows to return as the final estimate. The estimate will be outside of the error interval as long as half of the d estimates are outside. This is to be contrasted with the Count-Min Sketch, which uses the *min* operator, and thus fails only if *all* the d estimates exceed the error limit. In addition, to be able to use the Chernoff bound argument, the success probability of each individual estimate has to be strictly greater than $1/2$, whereas in the case of the Count-Min Sketch, any constant will do.

L_2 bound. The Count Sketch can similarly provide an L_2 error guarantee. First, we show that the estimator from each row is unbiased:

$$\mathsf{E}[\hat{v}_{i,j}] = \mathsf{E}[g_j(i) \cdot C[j, h_j(i)]] = \mathsf{E}\left[g_j(i) \sum_{1 \leq \ell \leq M : h_j(i) = h(\ell)} g_j(\ell) v_\ell\right]$$

$$
\begin{aligned}
&= (g_j(i))^2 v_i + \mathsf{E}\left[\sum_{\ell \neq i} g_j(i) g_j(\ell) v_\ell\right] \\
&= v_i + \sum_{\ell \neq i} v_\ell (\Pr[g_j(i) g_j(\ell) = +1] - \Pr[g_j(i) g_j(\ell) = -1]) \\
&= v_i.
\end{aligned}
$$

This analysis breaks the expectation of the result into two pieces: the contribution from v_i, and the contribution from the other items. Whatever the value of $g_j(i)$, we obtain a contribution of v_i from this term. For the second term, the expectation is multiplied by $g_j(i)g_j(\ell)$. Since we chose g_j from a pairwise independent family of hash-functions, over this random choice, the product is +1 half the time, and −1 the rest of the time, and so is zero in expectation. Of course, while this product is zero in expectation, it is never actually zero (it is either +1 or −1), and so we consider the variance of the estimator from row j in order to study its accuracy.

$$
\begin{aligned}
\mathsf{Var}[\hat{v}_{i,j}] &= \mathsf{Var}[g_j(i) \sum_{1 \leq \ell \leq M : h_j(\ell) = h_j(i)} g_j(\ell) v_\ell] \\
&= \mathsf{Var}[v_\ell + g_j(i) \sum_{\ell \neq i} g_j(\ell) v_\ell I(h_j(\ell) = h_j(i))] \\
&= \sum_{\ell \neq i} \mathsf{Var}[g_j(i) g_j(\ell) v_\ell I(h_j(\ell) = h_j(i))] \\
&= \sum_{\ell \neq i} v_\ell^2 \mathsf{Var}[I(h_j(\ell) = h_j(i)] \leq \frac{\|v\|_2^2}{t}.
\end{aligned}
$$

This analysis uses the standard properties of variance (Section 1.4), along with the fact that the events of two items both being mapped to $h_j(i)$ have zero covariance, due to the independence properties of the hash functions. Thus, applying a Chebyshev bound (Fact 1.3), we have (for any row j)

$$
\Pr\left[|\hat{v}_{i,j} - v_i| > \frac{2\|v\|_2}{\sqrt{t}}\right] \leq \frac{1}{4},
$$

showing that the estimate is within two times the standard deviations of its expectation with probability at least 3/4. Taking the median of $O(\log 1/\delta)$ repetitions and applying a Chernoff bounds argument (Section 1.4.1) reduces the probability to δ.

We can also observe that the random variable $\hat{v}_{i,j}$ has a pdf that

is symmetric around its expectation, so the median of an odd number of such estimators retains the expectation. So the Count Sketch returns an unbiased estimator.

Fact 3.5 *The* Count Sketch *returns an unbiased estimator for any point query. The error in the estimate, with parameters* $t = O(1/\varepsilon^2)$ *and* $d = O(\log 1/\delta)$*, is at most* $\varepsilon\|v\|_2$ *with probability at least* $1 - \delta$*.*

Note that the Count Sketch achieves both the L_1 and the L_2 error guarantees with the same QUERY algorithm, namely, the estimate returned always obeys the smaller of the two error guarantees. This can be considered as an additional benefit of the Count Sketch over the Count-Min Sketch, which needs to use different QUERY algorithms for achieving different error guarantees.

L_2 residual bound. We can also express the error in terms of $\|v\|_2^{\text{res}(k)}$, the residual L_2 norm, after removing the k largest (absolute) entries of v. This follows by adjusting the argument used to show Fact 3.5. Consider the estimate of $\hat{v}_{i,j}$ from row j. The probability that the heaviest item collides with i in row j of the sketch is at most $\frac{1}{t}$, using the pairwise independence of the hash functions. This also holds for the rest of the k heaviest items. The probability that all k of these items avoid i in row j is at least $1 - \frac{k}{t}$, using a union bound. So for $k = O(t)$, this holds with at least constant probability. If we condition on this event, then the variance of the estimator is reduced to $\|v\|_2^{\text{res}(k)}/t$. Applying the Chebyshev bound to this case, there is constant probability that the estimate is at most $\sqrt{\|v\|_2^{\text{res}(k)}/t}$ from its expected value. This constant probability captures the probability of avoiding collisions with the k heaviest items, and then of the conditioned random variable falling close to its expectation. Thus, with some rescaling of constants, we also have that a sketch of width $t = O(1/\varepsilon^2)$ and $d = O(\log 1/\delta)$ estimates v_i with error at most $\varepsilon\|v\|_2^{\text{res}(k)}/t$ for $k = O(t)$.

Converting Count-Min Sketch *to* Count Sketch*.* A different QUERY algorithm on the Count-Min Sketch can actually turn the Count-Min Sketch into a Count Sketch. We use the following estimator:

$$\hat{v}_{i,j} = C[j, h_j(i)] - C[j, h_j(i) + 1 - 2(h_j(i) \mod 2)],$$

where the term $1 - 2(h_j(i) \mod 2)$ has the result of picking a neigh-

boring entry in the sketch. The effect of this is to produce an estimate which is identical to that arising from the Count Sketch of size $t/2$: the effect of the subtraction is to mimic the role of the hash function g_j. Consequently, a Count-Min Sketch structure can simulate a Count Sketch and provide the same guarantees as the Count Sketch, though at the cost of twice the space.

Implementation Issues. The pairwise hash functions needed are very simple: picking parameters a and b uniformly in the range 1 to p for a prime p, then computing $((ax + b) \mod p) \mod t$ is sufficient (Section 1.4.2). Still, in some high performance cases, where huge amounts of data is processed, we wish to make this part as efficient as possible. One way to reduce the cost of hashing is to compute a single hash function for row j that maps to the range $2t$, and use the last bit to determine g_j (+1 or −1), while the remaining bits determine h_j. This is essentially equivalent to the above observation that a Count Sketch is equivalent to taking a Count-Min Sketch of twice the width, and computing the differences of pairs of adjacent entries.

History and Background. The Count Sketch summary was first proposed by Charikar, Chen and Farach-Colton in 2002 [49], although technically it can be thought of as an extension of the earlier AMS Sketch [10], which also used the idea of +1/-1 hash values to provide an unbiased estimator. The key innovation is the use of hashing in the Count Sketch to select a cell in each row, instead of averaging all cells. This means that the UPDATE operation is much faster than in the AMS sketch.

The Count Sketch has found many practical uses, due to its relative simplicity, and good accuracy. It is implemented in various tools for manipulating large data, such as the Sawzall system at Google for analyzing log structured data [196]. The simple structure has also allowed it to be extended to solve other problems. Pagh [192] showed that one could rapidly construct a Count Sketch of the product of two matrices, without explicitly performing the matrix multiplication; this is explained in Section 6.4. For similar reasons, it is possible to efficiently build a Count Sketch of polynomial kernels used in machine learning [195].

The Count Sketch is a key tool in the design of advanced algorithms for problems in high dimensional data analysis. Many problems in randomized numerical linear algebra such as regression can be approximately solved using Count Sketch [198, 51]. Problems relating to com-

pressed sensing — recovering approximately sparse vectors from few linear measurements — make use of the Count Sketch [113]. Lastly, the Count Sketch is the core part of a general approach to approximately computing functions over high dimensional vectors [36].

3.6 (Fast) AMS Sketch for Euclidean norm

Brief Summary. The AMS Sketch summary maintains an array of counts which are updated with each arriving item. It gives an estimate of the ℓ_2-norm of the vector v corresponding to the multiset being summarized, by computing the norm of each row, and taking the median of all rows. Given parameters ε and δ, the summary uses space $O(1/\varepsilon^2 \log 1/\delta)$, and guarantees with probability at least $1 - \delta$ that its estimate is within relative ε-error of the true ℓ_2-norm, $\|v\|_2$.

Algorithm 3.13: AMS Sketch: QUERY (x)

```
1 for j ← 1 to d do
2 |   e_j ← 0 ;
3 |   for k ← 1 to t do
4 |   |   e_j ← e_j + (C[j, k])^2 ;
5 return median(e)
```

Operations on the summary. The AMS Sketch summary is almost identical to the Count Sketch summary, with one key difference—sometimes the terms Count Sketch and AMS Sketch are used interchangeably. It is represented as a compact array C of $d \times t$ counters, arranged as d rows of length t. In each row j, a hash function h_j maps the input domain U uniformly to $\{1, 2, \ldots t\}$. A second hash function g_j maps elements from U uniformly onto $\{-1, +1\}$. So far, this is identical to the description of the Count Sketch. An additional technical requirement is that g_j is *four-wise* independent (Section 1.4.2). That is, over the random choice of g_j from the set of all possible hash functions, the probability that any four distinct items from the domain get mapped to $\{-1, +1\}^4$ is uniform: each of the 16 possible outcomes is equally likely.

The INITIALIZE, UPDATE and MERGE operations on the AMS Sketch summary are therefore identical to the corresponding operations on the Count Sketch summary (Section 3.5). Hence the algorithms for INITIALIZE and UPDATE match those given in Algorithms 3.10 and 3.11, the

difference being the stronger requirements on hash functions g_j. Only QUERY is different. Here, the QUERY operation takes the sum of the squares of row of the sketch in turn, and finds the median of these sums. That is, for row j, it computes $\sum_{k=1}^{t} C[j,k]^2$ as an estimate, and takes the median of the d such estimates. Algorithm 3.13 gives code for the QUERY operation on AMS Sketch summaries: each estimate e_j is formed as the sum of squares of the jth row, and the median of these is chosen as the final estimate. The QUERY time is linear in the size of the sketch, $O(td)$. The time for INITIALIZE and MERGE operations is the same, $O(td)$. Meanwhile, UPDATE operations take time $O(d)$.

Further Discussion. To understand the accuracy of the estimates, consider the expectation and variance of the result obtained from each row. Let X_j denote the estimate obtained from row j, i.e., $X_j = \sum_{k=1}^{t} C[j,k]^2$. We have

$$\begin{aligned}
\mathsf{E}[X_j] &= \sum_{k=1}^{t} \mathsf{E}[C[j,k]^2] \\
&= \sum_{k=1}^{t} \mathsf{E}\left[\left(\sum_{i:h_j(i)=k} v_i g_j(i)\right)^2\right] \\
&= \sum_{k=1}^{t} \mathsf{E}\left[\sum_{i:h_j(i)=k} v_i^2 + 2 \sum_{i\neq\ell, h_j(i)=h_j(\ell)=k} g_j(i) g_j(\ell) v_i v_\ell\right] \\
&= \sum_{k=1}^{t} \sum_{i:h_j(i)=k} v_i^2 = \sum_i v_i^2 = \|v_i\|_2^2,
\end{aligned}$$

because the expectation of terms in $g_j(i)g_j(\ell)$ is zero: this product is $+1$ with probability $\frac{1}{2}$ and -1 with probability $\frac{1}{2}$. For the variance, we have

$$\begin{aligned}
\mathsf{Var}[X_j] &= \mathsf{Var}\left[\sum_{k=1}^{t} C[j,k]^2\right] \\
&= \mathsf{Var}\left[\|v\|_2^2 + 2\sum_{k=1}^{t}\sum_{i\neq\ell} g_j(i) v_i I(h_j(i)=k) \cdot g_j(\ell) v_\ell I(h_j(\ell)=k)\right] \\
&= \mathsf{Var}[2\sum_{i\neq\ell} g_j(i) v_i g_j(\ell) v_\ell I(h_j(i)=h_j(\ell))].
\end{aligned}$$

We will use random variables $Y_{i,\ell} = g_j(i)v_i g_j(\ell)v_\ell I(h_j(i) = h_j(\ell))$ to simplify this expression. Note that $\mathsf{E}[Y_{i,\ell}] = 0$, by the pairwise independence of g_j. We will use the fact that $Y_{i,\ell}$ is independent from $Y_{q,r}$ when all four of $\{i, \ell, q, r\}$ are different, using the four-wise independence property of g_j. That implies $\mathsf{Cov}[Y_{i,\ell}, Y_{q,r}] = 0$ under this condition. Then

$$\begin{aligned}
\mathsf{Var}[X_j] &= \mathsf{Var}\left[2\sum_{i\neq\ell} Y_{i,\ell}\right] \\
&= 4\sum_{i\neq\ell, q\neq r, h_j(i)=h_j(q)} \mathsf{Cov}[Y_{i,\ell}, Y_{q,r}] \\
&= 4\sum_{i\neq\ell} \mathsf{Cov}[Y_{i,\ell}, Y_{i,\ell}] + 4\sum_{i\neq\ell, i\neq q} \mathsf{Cov}[Y_{i,\ell}, Y_{i,q}] \\
&= 4\sum_{i\neq\ell} \mathsf{E}[Y_{i,\ell}^2] + 0 \\
&= 4\sum_{i\neq\ell} v_i^2 v_\ell^2/t \leq \|v\|_2^4/t.
\end{aligned}$$

Here, we use the fact that $\mathsf{E}[Y_{i,\ell}Y_{i,q}] = 0$ when $i \neq q, \ell \neq q$ (again, using the four-wise independence of g_j), and hence $\mathsf{Cov}[Y_{i,\ell}, Y_{i,q}] = 0$ also.

To summarize, we have that $\mathsf{E}[X_j] = \|v\|_2^2$ and $\mathsf{Var}[X_j] \leq 4\|v\|_2^4 \leq 4\mathsf{E}[X_j]^2/t$. Thus, via the Chebyshev inequality, the probability that $|X_j - \mathsf{E}[X_j]| \geq \|v\|_2^2/\sqrt{t}$ is at most a constant. Taking the median of d repetitions drives this probability down to δ. So we conclude

Fact 3.6 *The error in the estimate of $\|v\|_2^2$ from an* AMS Sketch *with parameters $t = O(1/\varepsilon^2)$ and $d = O(\log 1/\delta)$ is at most $\varepsilon\|v\|_2^2$ with probability at least $1 - \delta$.*

Equivalently, we have that the estimate is between $(1-\varepsilon)\|v\|_2^2$ and $(1+\varepsilon)\|v\|_2^2$. Taking the square root of the estimate gives a result that is between $(1-\varepsilon)^{1/2}\|v\|_2$ and $(1+\varepsilon)^{1/2}\|v\|_2$, which means it is between $(1-\varepsilon/2)\|v\|_2$ and $(1+\varepsilon/2)\|v\|_2$.

Note that since the input to AMS Sketch can be general, it can be used to measure the Euclidean distance between two vectors v and u: we can build an AMS Sketch of v and one of $-u$, and MERGE them together. Note also that a sketch of $-u$ can be obtained from a sketch of u by negating all the counter values.

Implementation Issues. For the analysis, we required that the random variables $Y_{i,\ell}$ and $Y_{q,r}$ have zero covariance. This requires that the expectation of terms in $g_j(i)g_j(\ell)g_j(q)g_j(r)$ are zero. For this to hold, we required that the hash functions g_j are drawn from a family of four-wise independent hash functions. This can be achieved by using polynomial hash functions, $g_j(x) = 2((ax^3 + bx^2 + cx + d \mod p) \mod 2) - 1$ (Section 1.4.2).

History and Background. The AMS Sketch was introduced in the work of Alon, Matias and Szegedy in 1996 [10]. That version had the same size, but was structured differently. Instead of hashing into a row of t entries, all entries are used to make a single estimator (with hash functions g mapping to +1/-1 values). The average of $O(1/\varepsilon^2)$ estimates was taken to reduce the variance, then the median of $O(\log 1/\delta)$ repetitions used to drive down the error probability. The idea to use hashing instead of averaging to achieve the same variance but with a lower UPDATE time cost is seemingly inspired by the Count Sketch summary (the hashing-to-replace-averaging technique is also referred to as "stochastic averaging" in the earlier work of Flajolet and Martin [103]). Several works adopted this idea, notably that of Thorup and Zhang [214]. The version with hashing to a row of t counters is often referred to as the "fast AMS Sketch" summary (see Cormode and Garofalakis [62]). The AMS Sketch has many other applications, due to the importance of estimating the ℓ_2 norm (i.e., the length of a vector in Euclidean space). In particular, its application to estimating the inner-product between pairs of vectors is discussed in Section 6.1.

3.7 L_p sketch for vector norm estimation

Brief Summary. An ℓ_p sketch gives an estimate of the ℓ_p-norm of the vector v corresponding to the multiset being summarized. It is similar in nature to the AMS Sketch, in that it builds a set of estimates, where each estimate is a projection of the input vector, and the median of all the estimates gives an approximation to the desired ℓ_p norm. Given parameters ϵ and δ, the summary uses space $O(1/\epsilon^2 \log 1/\delta)$, and guarantees that its estimate is within relative ϵ-error of the true ℓ_p norm , $\|v\|_p$.

Operations on the summary. The ℓ_p sketch is based on taking projections with vectors whose entries are randomly chosen from so-called "stable" distributions. Each entry of the projection vector is drawn independently from an identically distributed symmetric and strictly stable distribution. Stable distributions are parameterized by α, where Gaussian distributions are stable with parameter $\alpha = 2$, and Cauchy distributions are stable with parameter $\alpha = 1$. The class of stable distributions are defined by the property that if X and Y are independent random stable variables with parameter α, then $aX + bY$ is distributed as a stable distribution scaled by $(|a|^\alpha + |b|^\alpha)^{1/\alpha}$. So to estimate $\|v\|_p$, we take the inner product of v with a vector s whose entries are sampled iid p-stable. The result is then distributed as $\|v\|_p$ scaling a p-stable variable. We repeat this multiple times to estimate the scaling parameter, which we report as the answer.

To INITIALIZE the sketch, we initialize the seeds for $k = O(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$ independent repetitions, and create an empty sketch of k entries, all set to 0.

Algorithm 3.14: ℓ_p sketch: UPDATE (i, w, p)

```
1 for j ← 1 to k do
2     C[j] ← C[j] + w * stable_p(i, j)
```

To process an UPDATE to the sketch, we make use of a function $\text{stable}_p(i, j)$, which consistently draws a random value from a stable distribution with parameter p. We assume that $\text{stable}_p(i, j)$ gives the same value every time it is called with the same values of i, j, and that $\text{stable}_p(i, j)$ is independent of $\text{stable}_p(i', j')$ if $i \neq i'$ or $j \neq j'$.

To QUERY a sketch, we compute the scaled median of the (absolute) stored values, i.e., $\text{median}_{j=1\ldots k} |C[j]|/\beta_p$. Here, β_p is the median of absolute values drawn from the p-stable distribution.

Finally, to MERGE two sketches together which are built using the same $\text{stable}_p(i, j)$ values, we simply have to add together their sketch vectors C.

Example. Consider the input vector

$$v = [1, 2, 3, -4]$$

so we have that $\|v\|_1 = 10$.

We will sample from the Cauchy distribution, since this distribution

is stable with parameter 1, and has $\beta_1 = 1$. We draw the following sets of four random values from iid Cauchy random variables for $k = 3$:

$$\begin{array}{rrrr} -1.639 & -1.722 & 0.338 & -0.238 \\ 0.450 & -0.015 & -0.212 & -7.742 \\ -1.641 & 12.280 & -1.048 & 7.914 \end{array}$$

Taking the inner product of v with each row of these draws generates $\{3.117, 30.752, -11.81\}$. We can observe that there is quite high variation among these three estimates, but that taking the median of the absolute values returns 11.81, which is tolerably close to the true value of 10.

Further Discussion. The sketch relies on the fact that linear scaling composes with the operations of taking absolute values and taking medians. That is, we have that for a random variable X,

$$\text{median}(|sX|) = s\,\text{median}(|X|).$$

In particular,

$$\text{median}(|\|v\|_p X_p|) = \|v\|_p\,\text{median}(|X_p|).$$

Combined with the defining properties of stable distributions, we have that each random projection yields an estimate centered on our target value of $\|v\|_p$.

The repetitions allow us to invoke concentration of measure results, and give bounds on the repetitions. The analysis follows using an additive Chernoff-Hoeffding bound (Fact 1.4) in a variant of the standard median argument (Section 1.4.1). We outline the steps. Let τ be the value so that $\Pr[X \leq \tau] = \frac{1}{2} - \epsilon$, and consider the median of k copies of the estimator defined above. A necessary condition for this empirical median to fall below τ is for more than half the samples to fall below τ. The probability of each one falling below τ is captured by a Bernoulli random variable with probability $\frac{1}{2} - \epsilon$. Applying the Chernoff-Hoeffding bound to these $k = O(\frac{1}{\epsilon^2} \log 1/\delta)$ Bernoulli variables gives us a probability of at most $\delta/2$ for this failure mode. The case for the estimate going above τ' such that $\Pr[X \leq \tau'] = \frac{1}{2} + \epsilon$ is symmetric. This ensures that, with probability at least $1 - \delta$, we obtain an estimate which is close to the true median.

This argument shows that the estimate is close to the true median in terms of the quantiles of the distribution. It is desirable to transform this into being close in absolute value. This is done by arguing that the relevant stable distribution is not flat close to the median. That is, that τ and τ' are both close to the median of the distribution. This can be verified analytically for the Cauchy distribution (stable with $p = 1$), and empirically for other values of p.

Implementation Issues. The key technical step needed is to sample from a stable distribution. We first observe that certain stable distributions have a closed form. The Gaussian distribution is 2-stable, while the Cauchy distribution is 1-stable. We can sample from the Cauchy distribution by transforming a uniform distribution. If U is a uniform random distribution over $[0, 1]$, then $\tan(\pi(U - \frac{1}{2}))$ follows a Cauchy distribution.

For other p values in the range $0 < p < 2$, a transformation due to Chambers, Mallows and Stuck [45] generates a sample from a stable distribution using two random variables. Let U and V be independent uniform variables from $[0, 1]$. Define the transform $\theta(U) = \pi(U - \frac{1}{2})$. Then the value

$$X_p = \frac{\sin(p\theta(U))}{\cos^{1/p}(\theta(U))} \left(\frac{\cos((1-p)\theta(U))}{-\ln V} \right)^{\frac{1-p}{p}}$$

is distributed according to a p-stable distribution.

We also need to compute $\beta_p = \text{median}|X_p|$, the median of absolute values drawn from the p-stable distribution. For $p = 1$, the Cauchy distribution yields $\beta_1 = 1$ (and similarly, for $p = 2, \beta_2 = 1$ for the Gaussian distribution). For other values of p, β_p can be found with high accuracy by sampling a large enough number of values, and taking the median of them.

A practical consideration is that for very small values of p, approaching 0, the values sampled from stable distributions are typically very large in magnitude, and can exceed the capacity of fixed-size variables. Consequently, working with small p requires higher precision arithmetic, and the true space cost grows larger as p gets asymptotically close to zero. Note that p close to zero is interesting, since it tends to flatten out the contribution of each non-zero value, and so approximately counts the number of non-zero values.

When k the number of projections is large, the UPDATE procedure can

be rather slow, since $O(k)$ time is needed each update. Li [163] proposes a sparse random projection, where many entries in the projection matrix are chosen to be 0, and we only need to process the non-zero entries. Formally, Li shows that it suffices to ensure that only a $1/\sqrt{M}$ fraction of the entries need to be non-zero, where M is the dimensionality of the input vector. Li also shows that a modified estimation procedure based on taking the geometric mean of the individual estimates, rather than the median, provides accurate guarantees.

History and Background. Stable distributions have been extensively studied within the statistics literature – see the book by Zolotarev [236]. Their use in data summarization was pioneered by Indyk [132]. Experiments on the use of stable distributions with low p values are given in [60]. Extensions on the method for $p = 1$ to obtain fast, small summaries are due to Nelson and Woodruff [189].

3.8 Sparse vector recovery

Brief Summary. A SparseRecovery structure summarizes a multiset A of items under insertions and deletions of items. It allows the multiset to be recovered in its entirety, but only when the number of distinct items in the multiset is small. It is of value when at some intermediate point the multiset A may be very large, but due to deletions it later becomes small again. The structure is defined based on a parameter s, so that when the multiset has at most s distinct items, it will almost certainly recover them correctly; when the number of items is more than s then no guarantee is made (the structure may return FAIL indicating it could not recover the full multiset). The summary keeps structures similar to the sketches discussed previously, and uses extra information to retrieve the identity and count of items currently in the multiset.

Operations on the summary. The sparse recovery structure resembles the BloomFilter as described in Section 2.7. As with the BloomFilter, we use k hash functions $h_1, \ldots, h_k$ to map each item to k locations in the array. For technical reasons, we will need these k hash functions to map any item to distinct locations. Unlike the BloomFilter, which uses a simple bit array, SparseRecovery uses an array C of t cells, where each cell stores more information about the elements that are mapped there. Specifically, each cell contains three pieces information: a *count* of the

items that have been mapped to that location, a *sum* of all the item identifiers, and a *fingerprint* (Section 3.1) of the collection of items there.

Algorithm 3.15: SparseRecovery: INITIALIZE (t)

```
1 for j ← 1 to t do
2     C[j].count ← 0;
3     C[j].sum ← 0;
4     C[j].fingerprint.INITIALIZE ();
```

To INITIALIZE the structure (Algorithm 3.15), the array is created, with all counts and sums set to zero; the fingerprints are initialized to 0 or 1 depending on which version of Fingerprint is used (see Section 3.1).

Algorithm 3.16: SparseRecovery: UPDATE (i, w)

```
1 for j ← 1 to k do
2     C[h_j(i)].count ← C[h_j(i)].count + w;
3     C[h_j(i)].sum ← C[h_j(i)].sum + w · i;
4     C[h_j(i)].fingerprint.UPDATE(i, w);
```

Each UPDATE maps the given item to its corresponding locations in the array, and updates the sums, counts and fingerprints there based on the weight w (w can be either positive or negative). The count is modified by adding on w, while the sum is updated by adding on w times the item identifier i. We invoke the procedure to update the fingerprint as described in Section 3.1. Pseudocode is shown in Algorithm 3.16.

As with the BloomFilter, the procedure to MERGE two SparseRecovery structures requires that they are created with the same parameters (the same t and the same hash functions h_j). Then we merge the sets of sums, counts and fingerprints. That is, pairs of corresponding sums are added, as are counts. The MERGE procedure for pairs of corresponding fingerprints is applied (Algorithm 3.2).

The QUERY process is a bit involved. The idea behind QUERY is that if the size of the multiset stored in the structure is small, then we can search the array and pick out the entries one by one, potentially uncovering more as we go. In particular, if there is some entry of the array that only stores information about a single item, then we use the information to retrieve the identifier and weight of that item. In this case, the total weight w can be read directly from the count value of the cell. Using standard arithmetic, since the sum contains $i \cdot w$, we can divide it by w to find i. However, there are additional issues to be concerned with: we need some way to know for sure whether a particular cell contains

a single item, or contains a mixture of different items. Simply checking that the sum is an integral multiple of the count is not sufficient — it is possible that a mixture of different items add up to give the impression of a single one. A simple example is if one copy of each of items with identifiers 1 and 3 are placed in the same cell, then it would appear that there are two copies of the item with identifier 2 in the cell. This is why we also store the fingerprint: we attempt to recover an item from the cell, then compute the fingerprint corresponding to the conjectured item and weight. If it matches the stored fingerprint, then we are (almost) certain that this is a correct decoding, else we assume that it is an error.

The key observation behind the ability to recover many items is the following: Once an item has been recovered, it can be removed from the other cells in which it has been placed. That is, we can perform the operation UPDATE $(i, -w)$ on the item i with weight w. This can cause more cells to become decodable (removing colliding items and leaving a single undecoded item left in the cell). This process is repeated until no more items can be identified and removed. At this point, either the structure is empty (i.e., all items have been recovered and every cell in the array is empty), or some items remain and have not been identified. In this case, the process might output FAIL. This approach is sometimes called "peeling", since once an item is found, it is "peeled" away from the structure. This peeling process can also be implemented to run efficiently in $O(t)$ time, and the full pseudocode is provided in Algorithm 3.17.

The analysis presented below argues that the size of the array C just needs to be $O(s)$ to allow this recovery to happen almost certainly, if no more than s items are stored in the structure. It has been suggested to use some k between 3 and 7, which will make the failure probability $O(t^{-k+2})$ as long as $t > c_k s$, where c_k is some constant depending on k. The precise values of c_k are known, and are actually quite small, as given in Table 3.1.

Example. A small example is shown in Figure 3.3 with 6 elements in an array with 8 cells and $k = 3$. For simplicity, the figure shows in each cell the set of elements that are mapped there, rather than the corresponding sums, counts and fingerprints. Successive rows show the state of the structure after successive steps of peeling have recovered one or more items from the structure. Initially, only item 2 can be recovered,

Algorithm 3.17: SparseRecovery: QUERY ()

```
1  Q ← empty queue;
2  for j ← 1 to t do
3      if C[j].count ≥ 1 and
        (C[j].fingerprint = fingerprint of (C[j].sum / C[j].count, C[j].count))
        then
4          Q.enqueue(C[j].sum/C[j].count, C[j].count)
5  while Q ≠ ∅ do
6      (i, w) ← Q.dequeue();
7      output (i, w);
8      for j ← 1 to k do
9          C[h_j(i)].count ← C[j, h_j(i)].count − w;
10         C[h_j(i)].sum ← C[j, h_j(i)].sum − w · i;
11         C[h_j(i)].fingerprint.UPDATE(i, −w);
12         if C[h_j(i)].count ≥ 1 and
            (C[h_j(i)].fingerprint = fingerprint of (C[h_j(i)].sum/C[h_j(i)].count, C[h_j(i)].count))
            then
13             Q.enqueue(C[h_j(i)].sum/C[h_j(i)].count, C[h_j(i)].count)
```

k	3	4	5	6	7
c_k	1.222	1.295	1.425	1.570	1.721

Table 3.1 *Values of c_k for $k = 3, 4, 5, 6, 7$.*

as all other cells have more than one items mapped there. However, as the peeling proceeds, we are able to recover all items eventually.

3, 4	1, 3, 4, 6	2	1, 5	2, 3, 5	1, 4	2, 6	5, 6	2 recovered
3, 4	1, 3, 4, 6		1, 5	3, 5	1, 4	6	5, 6	6 recovered
3, 4	1, 3, 4		1, 5	3, 5	1, 4		5	5 recovered
3, 4	1, 3, 4		1	3	1, 4			1, 3 recovered
4	4				4			4 recovered

Figure 3.3 An example on the peeling process.

Implementation Issues. Remember that we require the k hash functions to map each item to distinct locations. There are three ways to implement this.

The strictest way to follow this requirement is to have a single hash function h that maps the universe of items to the range $\{0, 1, ..., \binom{t}{k} - 1\}$. Note that there are a total of $\binom{t}{k}$ subsets of k distinct locations in an array of size t, so this hash function will pick one subset uniformly at random. For any item i, the k actual locations can be then extracted from the hash value $h = h(i)$ as follows. The first location is $h_1 = h \mod t$. Then we update $h \leftarrow h - t \cdot h_1$. The second location is $h_2 = h \mod (t-1)$, counting from the beginning of the array but skipping location h_1. So we need to add 1 to h_2 if $h_1 \leq h_2$. Then we update $h \leftarrow h - (t-1)h_2$. This process continues until we get h_k.

Alternatively, one can have k separate tables each of size t, with hash function h_j mapping items to the j-th table. This will also ensure that any item is hashed to k distinct locations. However, the downside is that the structure size is now k times larger. It is believed that using this version, the size of each table can be smaller than $c_k s$, but no formal analysis is known so far.

Finally, one can simply discard the duplicated hash values returned by the k hash functions, that is, an item may be mapped to fewer than k locations when there are duplicated hash values. Strictly speaking, the analysis below does not hold for this version, but empirically this may not lead to much difference for t not too small.

There are cases where the SparseRecovery structure can be further simplified. If we are not dealing with a multiset but a set, i.e., at most one copy of each item can exist in the structure, then the *sum* field can be replaced by the XOR of all the item identifiers mapped there. This eliminates any concern of bit overflow. Second, if we are dealing with a set and we are guaranteed that there will be no extraneous deletions, i.e., deleting a nonexistent item, then the *fingerprint* field is not needed. We simply check if the count is 1 for a cell, and recover the item from the cell if it is.

Further Discussion. The analysis relies on the connection between the peeling process and the existence of a 2-core in a random hypergraph. A *hypergraph* $G = (V, E)$ is similar to a standard graph, the only difference being that an edge in a standard graph connects

two vertices but a hypergraph edge (or 'hyperedge') may connect any number of vertices, i.e., each edge is now any subset of V.

To make the above connection, think of the cells as being vertices in the hypergraph, and the items as being hyperedges, with the vertices for an edge corresponding to the cells that the item is mapped to. In our case, each edge covers k randomly chosen vertices in the hypergraph. The *2-core* is the largest sub-hypergraph that has minimum degree at least 2, i.e., each vertex must be covered by at least 2 edges. The peeling process for SparseRecovery is then exactly the same as the standard peeling algorithm to find the 2-core of a hypergraph: while there exists a vertex with degree 1, delete it and the corresponding hyperedge. The equivalence between the peeling process and the QUERY algorithm for SparseRecovery is thus immediate.

We can recover the vector completely if the 2-core of the corresponding hypergraph is empty. This is determined by the random choice of the hash functions, which we can treat as generating a random hypergraph. Prior studies on the 2-core in a random hypergraph have established tight bounds on its existence probability. Let t be the number of vertices and s the number of hyperedges. If $t > (c_k + \varepsilon)s$ for any constant $\varepsilon > 0$ and the values of c_k in Table 3.1, then the probability that a non-empty 2-core exists is $O(t^{-k+2})$, which yields the claimed bounds above.

A full analysis on the existence probability of 2-cores is rather technical and can be found in [83, 180]; here we just provide some intuition. First let us consider the probability that any two given hyperedges form (part of) a 2-core. For this to happen, they must cover exactly the same set of k vertices, which happens with probability $O(t^{-k})$. As there are $O(s^2) = O(t^2)$ pairs of hyperedges, by the union bound, the probability that there exists a 2-core with 2 hyperedges is $O(t^{-k+2})$.

The full analysis will similarly examine the probability that there exists a 2-core with j hyperedges, for $j = 3, 4, \ldots, s$, and the calculation is quite involved. What has been shown is that, when $t > (c_k + \varepsilon)s$, these larger 2-cores are unlikely to occur, and the total probability of their existence is dominated by that of a 2-hyperedge 2-core.

History and Background. Structures which allow the recovery of a small number of items have appeared in many different problems within

computer science. For example, many problems in coding theory relate to sending information (a bit string), which is corrupted in a small number of places. The difference between the transmitted and received bit string can be interpreted as a set of locations, and the goal of error correction is to identify these locations, and correct them. Hence, many techniques from coding theory, such as Reed-Solomon and Reed-Muller codes [183], can be adapted to solve this sparse recovery problem.

More recently, interest in problems of "compressed sensing" have also produced algorithms for sparse recovery. The area of compressed sensing is concerned with defining a matrix M so that given the product Mx for a vector x, x can be recovered accurately. Note that exactly recovering x is impossible unless M has many rows, in which case simply setting M to be the identity matrix would solve the problem. Instead, it is usually assumed that x is 'sparse' in some sense, in which case much more compact matrices M can be defined. This sparsity could manifest in the sense that few entries of x are large and most are small (but non-zero). The case when only k entries are non-zero and the rest are zero is known as the *strict k-sparsity* or *exactly k sparse* case. The idea of using sum and count in this way to simply decode an isolated item seems to have appeared first in the work of Ganguly [105]. This technique can also be understood as a limiting case of Reed-Solomon coding, when two polynomial extensions of the input are taken, corresponding to the sum and count of the symbols.

Most recently, there has been a line of work developing "invertible Bloom look-up tables" (IBLT) [94, 118]. The goal of these is to function as a BloomFilter but with the additional property of being able to recover the encoded set when the number of items is small enough. In some cases, the focus can be limited to where the input describes a set rather than a multiset (i.e., the multiplicity of each item is restricted to 1); this slightly simplifies the problem. Our description of the SparseRecovery structure is close to that of Eppstein and Goodrich [94], though the simple (slightly weaker) analysis was presented in [61]. A distillation of these ideas appears in the form of "Biff codes", whose name acknowledges the Bloom Filter [179]. The key idea is to use an IBLT data structure to act as the error correcting part of a code, to handle character errors (or erasures). The resulting code is considered to be attractive, as encoding and decoding is very fast – encoding is essentially linear in the size of the input, while error correction takes time proportional to the number of errors. Although the constant factors that result

for using the Biff code are not the smallest possible, the fast operation and simple algorithm make the code attractive for settings where there are large volumes of data in large blocks, and quite low error rates, which is a common scenario.

3.9 Distinct Sampling / ℓ_0 Sampling

Brief Summary. An ℓ_0-sampler structure summarizes a multiset A of items under insertions and deletions. It allows one (or more) items to be sampled uniformly from those items currently in A, i.e., those that have non-zero count. Formally, if $|A|$ denotes the distinct cardinality of the multiset, that is, the number of items with non-zero count, then the summary promises to sample each item with probability $(1 \pm \varepsilon)/|A| \pm \delta$, for parameters ε and δ. With ε and δ sufficiently small, this sampling is approximately uniform. The structure is based on multiple instances of a SparseRecovery structure, where each instance samples its input with decreasing probability.

Operations on the summary. The sampling process is applied to items drawn from a known universe of possibilities, denoted U. The ℓ_0-sampler structure consists of m SparseRecovery data structures, where $m = \log n$ and $n = |U|$ is the size of the set of possible items in the multiset A (or a bound on the largest value of $|A|$). The i'th SparseRecovery structure is applied to a subset of U where items are included in the subset with probability 2^{-i+1}, which we call "level i". If we write S_i to denote the i'th SparseRecovery structure, then S_1 applies to all possible items, S_1 applies to half, S_3 to a quarter, and so on. Let U_i be the subset of the universe selected at level i, and we write A_i to denote the input multiset restricted to U_i. The idea is that there is some level i where $0 < |A_i| < k$, where k is the parameter of the SparseRecovery structures. At this level, we can recover A_i exactly, and from this we can sample an item. The subsequent analysis shows that this is sufficiently close to uniform.

Algorithm 3.18: ℓ_0-sampler: INITIALIZE (n, δ)

```
1 m ← log n;
2 for j ← 1 to m do
3 |   S_j ← SparseRecovery.INITIALIZE (log^2 1/δ) ;
4 h ← randomly chosen hash function ;
```

To INITIALIZE the structure, we fix the value of m, and INITIALIZE m SparseRecovery data structures. We also initialize the mechanism that will be used to determine the mapping of items to subsets. This is done with a hash function (so the same item is mapped consistently). Picking a hash function h that maps to the range $[0 \ldots 1]$, we define A_i to contain all those items j so that $h(j) \leq 2^{-i+1}$.

Algorithm 3.19: ℓ_0-sampler: UPDATE (j, w)

```
1 for i ← 1 to m do
2     if h(j) ≤ 2^{-i+1} then
3         S_i.UPDATE (j, w)
```

The UPDATE procedure applies the hash function to map to levels. Given an update to an item j and a change in its weight w, we apply the hash function h to determine which levels it appears in, and update the corresponding SparseRecovery structures. Algorithm 3.19 makes this explicit: $h(j)$ is evaluated, and compared against 2^{-i+1} to determine if $j \in U_j$.

Algorithm 3.20: ℓ_0-sampler: QUERY ()

```
1 for i ← 1 to m do
2     A_i ← S_i.QUERY ();
3     if A_i ≠ FAIL and A_i ≠ ∅ then
4         j ← random item from A_i ;
5         return (j) ;
```

To QUERY the structure, we iterate over each level, and find the first level at which we can successfully retrieve the full set A_i (i.e., the size of A_i is not too large). Then a random element from this set is returned (Algorithm 3.20). For this to work, we have to ensure that there is some A_i for which A_i is not too large and is not empty. This is determined by the analysis. Lastly, to MERGE two ℓ_0-sampler structures (provided they were initialized with the same parameters, and share the same h function) we just have to MERGE corresponding pairs of SparseRecovery structures.

Example. Figure 3.4 shows a schematic of how the ℓ_0-sampler structure stores information. Each level is shown with the list of items that are stored at that level. The input describes the set $A = \{3, 6, 7, 8, 10, 14, 18, 19, 20\}$, which is also represented as A_1. About half of these elements are also

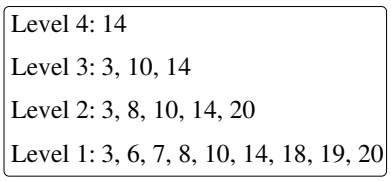

Figure 3.4 Example of ℓ_0-sampler structure with four levels

represented in $A_2 = \{3, 8, 10, 14, 20\}$. At level 5 and above, no elements from A are selected, so $A_5 = \emptyset$ (not shown).

Suppose we set the parameter k of the SparseRecovery structure to be 4. Then it is not possible to recover A_1 or A_2 — their cardinality is too large. However, A_3 can be recovered successfully, and one of these items can be picked as the sample to return.

If occurrences of items 3, 10 and 14 happened to be deleted, then level A_4 and A_3 would become empty. However, it would now be the case that $|A_2| = 3$, and so A_2 could be recovered and sampled from. The analysis detailed below argues that it will be possible to return a sample with good probability whatever the pattern of updates.

Further Discussion. The analysis has two parts. The first part is to argue that with a suitable setting of the parameter k for the SparseRecovery structures, there will be a level at which recovery can succeed. The second is to argue that the item(s) sampled will be chosen almost uniformly from A.

Given a particular input A, consider the event that some item is returned by the structure in response to a QUERY. For a level i, let the random variable $N_i = |A_i|$ be the number of (distinct) elements from the input that are mapped there. For the structure to succeed, we need that $1 \leq N_j \leq k$. Observe that

$$\mathsf{E}[N_i] = \sum_{j \in A} \Pr[j \in A_i] = \sum_{j \in A} 2^{-i+1} = \frac{|A|}{2^{i-1}},$$

by linearity of expectation since the chance that any item is included in A_i is 2^{-i+1}.

To ensure that there is at least one level with sufficiently many

items, consider the level i such that

$$\frac{k}{4} \leq \mathsf{E}[N_i] \leq \frac{k}{2}$$

We will be able to successfully recover A_i if its size is not too far from its expected size. Specifically, we need that the absolute difference between N_i and $\mathsf{E}[N_i]$ is at most $\mathsf{E}[N_i]$. If we assume that the choices of which items get included in A_i are fully independent, then we can apply Chernoff bounds of the form stated in Fact 1.5, and obtain that the probability of this event is exponentially small in k, i.e., $\Pr[1 \leq N_i \leq k] \geq 1 - \exp(-k/16)$. We note that it is not necessary to assume that the choices are fully independent; similar results follow assuming only k-wise independence [201, 61].

For the second part, we want to show that the sampling is almost uniform. This requires an argument that the items which survive to level i or above are uniformly selected from all items. This is the case when the selection is done with full independence, and this also holds approximately when limited independence is used. The fact that the sampling may fail with some small probability may also bias the distribution, but this gives only a very small distortion. Consequently, it holds that each item is sampled with probability $(1 \pm \varepsilon)/|A| \pm \delta$, where both ε and δ are exponentially small in k [61]. Equivalently, if we pick k to be $\log 1/\delta$, then the sampling probability is $1/|A| \pm \delta$.

Implementation Issues. As discussed in the analysis, an implementation must use hash functions to select items to level i that have only limited independence. The analysis shows that it is sufficient to have small ($\log 1/\delta$) independence for the guarantees to hold. Further, rather than map to $\{0 \ldots 1\}$, it is more common to have hash functions that map to integers and select items that hash to a 2^{-i+1} fraction of the range. However, provided the range of the integers is sufficiently large (say, more than n^3 given the bound n on the size of the input), this does not affect the selection probabilities. For the analysis to hold, it seems that we should select the level i such that $\frac{k}{4} \leq |A|/2^{i-1} \leq \frac{k}{2}$ to perform the recovery. This requires knowing a good estimate of $|A|$. However, it is seen that simply greedily picking the first level where recovery succeeds does not substantially affect uniformity, and does not decrease the failure probability [61].

The discussion and analysis has assumed the case where the goal is to

recover a single sample. However, when the goal is to recover a larger set of size s, the structure is almost identical. The only changes needed are to increase the size of k proportional to s [20].

History and Background. The notion of ℓ_0-sampling was first introduced under the name of distinct sampling, in order to solve problems in geometric data analysis [104] and to capture properties of frequency distributions [74]. Variations on the basic outline followed above have been presented, based on how sampling to levels is performed, and what kind of sparse recovery is applied. The underlying concept, of hashing items to levels and using recovery mechanisms, is universal [104, 74, 181, 144]. More recently, the variant where multiple items must be recovered has also been studied using similar constructions [20].

The notion has a surprising number of applications in computer science, although mostly these are more theoretical than practical. The chief surprise is that graph connectivity can be tracked in near-linear space, as edges are added and removed. This construction, reliant on ℓ_0 sampling, is described in Section 7.1. Other applications of ℓ_0 sampling appear in computational geometry, where they can be used to track the width (a measure of spread) of a pointset [14], the weight of a spanning tree in Euclidean space [104], and for clustering high dimensional data [34].

3.10 L_p sampling

Brief Summary. An ℓ_p-sampler structure summarizes a multiset A of items under insertions and deletions. It allows items to be sampled from those currently in A according to the ℓ_p distribution, where each item is weighted based on the p'th power of its current frequency. Formally, the objective is to sample each element j whose current frequency is v_j with probability proportional to $(1 \pm \epsilon)\frac{v_j^p}{\|v\|_p^p} \pm \delta$, where ϵ and δ are approximation parameters. The structure is built on top of the Count Sketch data structure which allows items to be selected based on the ℓ_2 distribution. With appropriate re-weighting, this achieves the desired sampling distribution.

Operations on the summary. The ℓ_p-sampler structure uses a hash function u to re-weight the input, where each hash value $u(j)$ is treated as

a uniform random real number in the range 0 to 1. We describe the method that works for $0 < p < 2$.

Algorithm 3.21: ℓ_p-sampler: INITIALIZE (p, ϵ, δ)

```
1 k ← 2⌈log(2/ε)⌉ ;
2 u ← random k-wise independent hash function to [0, 1];
3 if p = 1 then
4 |   m ← O(log 1/ε)
5 else
6 |   m ← O(ε^{−max(0,p−1)})
7 Count Sketch.INITIALIZE (m,log 1/δ);
8 ℓ_p sketch.INITIALIZE ()
```

To INITIALIZE the structure (Algorithm 3.21), we sample the necessary hash function u, and initialize the sketches that will be used to track the frequencies.

Algorithm 3.22: ℓ_p-sampler: UPDATE (j, w)

```
1 Count Sketch.UPDATE (j, u(j)^{−1/p} · w) ;
2 ℓ_p sketch.UPDATE (j, w, p)
```

To UPDATE the structure (Algorithm 3.22), we update the associated sketches: we use the hash function u to adjust the update to the Count Sketch, and we update the ℓ_p sketch to estimate the ℓ_p norm of the (unadjusted) input.

To query the structure and attempt to draw a sample according to the ℓ_p distribution, we try to identify the largest element in the rescaled vector. If the input defines a vector v, let z be the vector so that $z_j = v_j/u(j)^{1/p}$. We use the Count Sketch to find the (estimated) largest entry of z, as z_i, and also estimate $\|v\|_p$ by setting $r = \ell_p$sketch.QUERY (). We also use the Count Sketch to find the m largest entries of z, and estimate the ℓ_2 norm of the vector z after removing these (approximated) entries as s. We apply two tests to the recovered values, whose purpose is explained further below. These tests require $s \leq \epsilon^{1-1/p} m^{1/2} r$ and $|z_i| \geq \epsilon^{-1/p} r$. If these tests pass, we output i as the sampled index; otherwise, we declare that this instance of the sampler failed. To ensure sufficiently many samples are generated, this process is repeated independently in parallel multiple times.

Example. Consider the vector $v = [1, 2, 3, -4]$, so that $\|v\|_1 = 10$. We consider trying to ℓ_1 sample from this vector with $\epsilon = \frac{1}{2}$. For simplicity of

presentation, we assume that we obtain $m = 1$, and the sketch correctly find the largest items.

Suppose that the hash function u gives us a random vector

$$u = [0.919, 0.083, 0.765, 0.672].$$

Then we have the rescaled vector z as

$$z = [1.08, 24.09, 3.92, -5.95]$$

The largest magnitude entry of z is $z_2 = 24.09$. Suppose our estimate of $\|v\|_1$ gives $r = 11.81$, and we find s (the 2-norm of z after removing z_2) as 6.67. Our two tests are that $\|1.08, 3.92, -5.45\|_2 = 6.67 \leq 11.81$ and $24.09 \geq 2 \cdot 11.81$. Consequently, both tests pass, and we output 2 as the sampled index.

Further Discussion. The essence of the ℓ_p sampling algorithm is that the values of $u(j)^{1/p}$ give a boost to the probability that a small value becomes the largest entry in the rescaled vector z. It is chosen so that this probability corresponds to the desired sampling distribution. However, reasoning directly about an element becoming the largest in the rescaled vector is a little tricky, so we modify the requirement to considering the probability that any given element takes on a value above a threshold τ, and that no other element also exceeds that threshold.

We start by observing that the probability that the reciprocal of a uniform random value u_i in $[0, 1]$ exceeds a threshold τ is exactly $1/\tau$. That is, $\Pr[u_i^{-1} \geq \tau] = 1/\tau$. Now set $\tau = \|v\|_p^p / |v_i|^p$ and substitute this in: we get

$$\Pr[u_i^{-1} \geq \|v\|_p^p / |v_i|^p] = |v_i|^p / \|v_i\|_p^p.$$

Rearranging, we have

$$\Pr[|v_i|^p / u_i \geq \|v\|_p^p] = |v_i|^p / \|v_i\|_p^p$$
$$\text{and so } \Pr[|v_i| / u_i^{1/p} \geq \|v\|_p] = |v_i|^p / \|v_i\|_p^p$$

This looks promising: the quantity $|v_i| / u_i^{1/p}$ corresponds to the entries of the rescaled vector z, while $|v_i|^p / \|v_i\|_p^p$ is the desired ℓ_p sampling probability for element i.

However, there are some gaps to fill, which the formal analysis takes account of. First, we are not working with the exact vec-

tor of $v_i/u_i^{1/p}$ values, but rather a Count Sketch, with u given by a hash function. This is addressed in the analysis, using similar arguments based on properties of limited independence hash function and concentration of measure that we have seen already. Secondly, it turns out that using the threshold of $\|v\|_p$ is not desirable, since it could be that multiple indices pass this threshold once rescaled by u. This is handled by raising the threshold from $\|v\|_p$ to a higher value of $\|v\|_p/\epsilon^{1/p}$. This decreases the chance that multiple indices exceed the threshold. However, this also lowers the chance that any one index does pass the threshold. The analysis shows that the probability of returning any index from the structure is approximately proportional to ϵ. By repeating the procedure $O(1/\epsilon \log 1/\delta)$ times, the method returns a sample according to the desired distribution with probability at least $1 - \delta$.

The formal analysis is rather lengthy and detailed, and can be found in the references below.

History and Background. The notion of ℓ_p sampling was foreshadowed in some early works that provided algorithms for ℓ_0 sampling by Cormode *et al.* [74] and Frahling *et al.* [104]. The first solution to the more general problem was given by Monemizadeh and Woodruff [181], and subsequently simplified by Jowhari *et al.* [144]. More recently, Jayaram and Woodruff [136] showed that ℓ_p-sampling can be done perfectly, i.e., achieving $\varepsilon = 0$ in the sampling distribution, only using polylogarithmic space, although the summary is allowed to fail (i.e., not returning a sample) with a small probability.

ℓ_p sampling has numerous applications for computational problems. It can be applied to sampling rows and columns of matrices based on their so-called *leverage scores*, which reduces to ℓ_2 sampling [86], allowing the solution of various regression problems. More complex norms of data, known as "cascaded norms" can be approximated by ℓ_p sampling [181]. ℓ_2 sampling can also be used as the basis of estimating ℓ_q norms for $q > 2$, as first observed by Coppersmith and Kumar [58]. In graph algorithms, ℓ_2 sampling can be used to sample "wedges" nearly uniformly, as part of an algorithm to count triangles: a wedge is a pair of edges that share a vertex [173]. For a more comprehensive survey of ℓ_p sampling and its applications, see [66].