4

Summaries for Ordered Data

This chapter continues to work on multisets, but in addition we assume that the items are drawn from an ordered universe U (in this chapter, we will use U to denote both the domain and its size). As in Chapter 3, we use a vector representation for the multiset, and the input consists of (x, w) pairs, where x is an item from U and w is its integer weight. Some summaries only allow the input to have positive weights, while others allow both positive and negative weights. Some summaries allow only unweighted input, which means w is always taken to be 1. The multiplicity of x, v_x, is the sum of all the weights of x in the input. We require every v_x to be non-negative.

Queries to the summaries will exploit the ordering of items, e.g., finding the median of the multiset. Formally, for an element x, either in the data set or not, we define $\text{rank}(x) = \sum_{y<x} v_y$ to be the total weight of elements from the data that are strictly less than x. This allows the computation of various "order statistics". For example, the median will be largest element whose rank is no greater than $W/2$, where W is the total weight of all elements. The summaries discussed in this chapter support two types of queries. A *rank query* finds the rank of a given element x, while a *quantile query* returns an element with a given rank. These queries are important in describing the distribution of the underlying data. As the summaries are lossy, they do not return the exact ranks. Let ε be an error parameter. For a rank query with a given element x, the summary will return an approximate rank $\tilde{r}$, such that $\text{rank}(x) - \varepsilon W \leq \tilde{r} \leq \text{rank}(x) + \varepsilon W$. Similarly, for a quantile query with a given rank r, the summary will return an element x such that $r - \varepsilon W \leq \text{rank}(x) \leq r + \varepsilon W$. Note that in a multiset, such an element may not exist, when there is an x with $\text{rank}(x) < r - \varepsilon W$ and $\text{rank}(x) + v_x > r + \varepsilon W$. In this case the summary returns x.

In this chapter, we consider four quite distinct approaches to this problem, which apply to different restrictions on the input data.

- The Q-Digest summary (Section 4.1) is based on maintaining a subset of simple counters of ranges, when the input is drawn from a fixed integer domain, and can be weighted.
- The GK summary (Section 4.2) can be viewed as maintaining a (deterministic) sample of the input. It is a comparison-based summary, meaning that it can be applied to any data type for which a comparator is defined, such as real numbers, user-defined types, etc., so it has a wider applicability than Q-Digest. However, it can only handle items with unit weights.
- The KLL summary (Section 4.3) is a randomized summary and provides a probabilistic guarantee, i.e., it answers queries within the ε-error bound with a high probability, which can be controlled by appropriate parameters. There is always a small chance that the error may exceed ε. This summary can also handle weighted items.
- The DCS summary (Section 4.4) is a sketching-based summary, so it can handle both insertion and deletion of items, which is its distinctive feature. However, it is also a randomized summary with a probabilistic guarantee. It also requires the input be drawn from a fixed integer domain. This summary can handle weighted items.

The following table summarizes the various features of these four quantile summaries:

Summary	Domain	Guarantee	Weights	Deletions
Q-Digest	Integer	Deterministic	Yes	No
GK	Comparison	Deterministic	No	No
KLL	Comparison	Probabilistic	Yes	No
DCS	Integer	Probabilistic	Yes	Yes

4.1 Q-Digest

Brief Summary. The Q-Digest provides a compact summary for data drawn from a fixed discrete input domain U, represented by the integers 0 to $U-1$. Each input is an (x, w) pair where w must be positive. The summary takes space at most $O(\frac{1}{\varepsilon}\log U)$, which is fixed and independent of the input. Summaries can be easily merged together. The

summary works by maintaining a tree-structure over the domain U: nodes in the tree correspond to subranges of U, and simple statistics are maintained for a small set of nodes.

Operations on the summary. The Q-Digest structure is represented as a collection of nodes within a binary tree whose leaves are the ordered elements of U. Each node v is associated with a weight w_v. Only nodes with non-zero weight are explicitly stored. The maintenance procedures on the data structure ensure that the number of nodes stored is bounded by $O(\frac{1}{\varepsilon} \log U)$. The sum of all weights of nodes is equal to the total weight of the input items summarized, W. The INITIALIZE operation creates an empty Q-Digest, i.e., all the nodes in the binary tree have (implicit) weight 0.

At any time, there is a notion of the "capacity" C of the non-leaf nodes in the structure. This capacity is set to be $\varepsilon W / \log U$. The intuition is that error in answering queries will come from a bounded number of nodes in the tree. By bounding the weight associated with each node, this will lead to a bound on the error.

UPDATE. Each UPDATE operation of an item x performs a walk down the tree structure, starting from the root, with the aim of adding the weight w associated with x to nodes in the tree. The algorithm proceeds recursively: given a current node v, if the current weight associated with the node w_v is equal to the capacity C, then it finds which child node x belongs to (either the left or the right child) and recurses accordingly. When the node is below capacity, the algorithm tries to store the new weight there. If v is a leaf node, since leaf nodes do not have to enforce a capacity, we can add w to w_v, and finish. For a non-leaf node v, if the sum of the weights, $w_v + w$ is at most the capacity C, then we can safely update the weight of node v to $w_v + w$ and finish. Otherwise, adding the full weight would take v over capacity. In that case, we update the weight of v to $w_v \leftarrow C$. This leaves $w - (C - w_v)$ of the weight of the update x still unassigned, so we recurse on the appropriate child of v as before. This procedure will take at most $\log U$ steps, since the length of a path from the root to the leaf level is $\log U$, and the procedure will always stop when it reaches the leaf level, if not before.

Algorithm 4.1 shows the pseudocode to update the data structure with an item x of weight w. The central while-loop determines what to do at a given node v. If v is below capacity, then lines 5–8 put as much of the weight w there as possible, and adjust w accordingly. If this reduces

w to zero, then the algorithm can break out of the loop. Otherwise, it recurses on either the left or right subtree of v (denoted by v.left and v.right, respectively). Finally, any remaining weight is assigned to node v in line 14: if v is not a leaf, then w will be 0 at this point.

Algorithm 4.1: Q-Digest: UPDATE (x, w)

```
1  W ← W + w ;
2  C ← εW/ log U ;
3  v ← root ;
4  while v is not a leaf node do
5      if (w_v < C) then
6          d ← min(C − w_v, w) ;
7          w_v ← w_v + d;
8          w ← w − d;
9      if w = 0 then break;
10     if x is in left subtree of v then
11         v ← v.left
12     else
13         v ← v.right
14 w_v ← w_v + w ;
```

QUERY. A rank query for an element x is answered by computing the sum of weights of all nodes in the structure which correspond to ranges of items that are (all) strictly less than x. Note that all the weights stored in such nodes arose from the insertion of items which were strictly less than x, so this gives a correct lower bound on the answer. Similarly, nodes in the structure corresponding to ranges of items that are (all) strictly more than x do not contribute to rank(x). This only leaves nodes in the structure which include x: the weight of these nodes may have arisen from items that were below, above, or equal to x. These lead to the approximation in the answer. Note however that there are only $\log U$ such nodes: only nodes on the root-to-leaf path for x contain x. As the weight of each non-leaf node is bounded by the capacity C, we have that the error in the answer is bounded by at most $C \log U$. Since we require that $C \leq \varepsilon W/\log U$, this error is in turn bounded by εW.

Algorithm 4.2 shows code for the query procedure. The algorithm proceeds down from the root. It assumes that the values of W_v, the total weight of all nodes below (and including) node v have already been

calculated. This can be done with a single depth-first walk over the data structure. The algorithm computes the sum of weights of all nodes that are strictly less than x. The algorithm walks the path from the root down to x. At each step, if x belongs to the right subbranch of the current node v, then it adds in the count of all nodes in the left sub-ranch as W_u (line 8) to the running sum r.

Algorithm 4.2: Q-Digest: QUERY (x)

```
1  v ← root ;
2  r ← 0;
3  while v is not a leaf node do
4  |  u ← v.left ;
5  |  if x is in left subtree of v then
6  |  |  v ← u
7  |  else
8  |  |  r ← r + W_u ;
9  |  |  v ← v.right
10 return (r)
```

COMPRESS. If we only follow the above procedures, then the number of nodes stored in the structure can grow quite large. This is because the capacity $C = \varepsilon W/\log U$ is growing as more elements are inserted into the summary, so nodes that are full will become underfull over time. To counter this, we shall COMPRESS the structure from time to time. The COMPRESS operation ensures that the number of nodes stored in the structure is bounded by $O(\log U/\varepsilon)$, while retaining all the properties needed to give the accuracy guarantees. The first step is to update the capacity C as $\varepsilon W/\log U$. Then the central idea of COMPRESS is to ensure that as many nodes in the structure as possible are at full capacity, by moving weight from nodes lower down in the tree up into their ancestors.

First, we compute for each node v the total weight of the subtree at v, as W_v. This can be done with a single depth-first walk over the tree. Then COMPRESS proceeds recursively, starting from the root. Given a current node v, if it is below capacity, i.e., $w_v < C$, then we seek to increase its weight by "pulling up" weight associated with its descendants. If W_v is sufficient to bring the node up to capacity, then we can adjust w_v up to C. This incurs a "debt" of $(C - w_v)$ which must be passed

on to the descendants of v. This debt is apportioned to the left and right children of v so that neither child is given a debt that exceeds the weight of their subtree. One way to do so is to assign as much of the debt as possible to one of the children (e.g., always the left child), while respecting this condition. The procedure continues recursively, although now the weight W_v of a subtree has to be reduced by the debt inherited from the parents. If this causes the weight of the subtree to be 0, then the node, and all its descendants now have zero weight, and so do not have to be retained. The recursion halts when a leaf node is reached: leaf nodes have no descendants, so no further weight can be pulled up from them.

Algorithm 4.3: Q-Digest: COMPRESS (v, d)

```
1  if w_v = 0 then return;
2  if v is a leaf node then
3  |  w_v ← w_v − d
4  else
5  |  if W_v − d > C then
6  |  |  d ← d + C − w_v;
7  |  |  w_v ← C ;
8  |  |  u ← v.left ;
9  |  |  COMPRESS (v.left, min(d, W_u)) ;
10 |  |  COMPRESS (v.right, max(d − W_u, 0));
11 |  else
12 |  |  w_v ← W_v − d ;
13 |  |  Set weight of all descendants of v to 0 ;
```

Algorithm 4.3 outlines the procedure for performing a COMPRESS. Again, we assume that the values W_v have been computed for each node as the sum of all weights in the subtree rooted at v. The algorithm is called initially with COMPRESS (v = root, $d = 0$), where v indicates the node for the algorithm to work on, and d indicates the current debt inherited from ancestor nodes. At the leaf level, the current debt is applied to the weight of the current node (line 3). The test at line 5 determines whether the current weight of the subtree, less the inherited debt, exceeds the capacity of the node v. If so, then the weight of v is increased to the capacity, and the debt adjusted to reflect this. Then as much of the debt as possible is passed on to the left subtree (rooted at

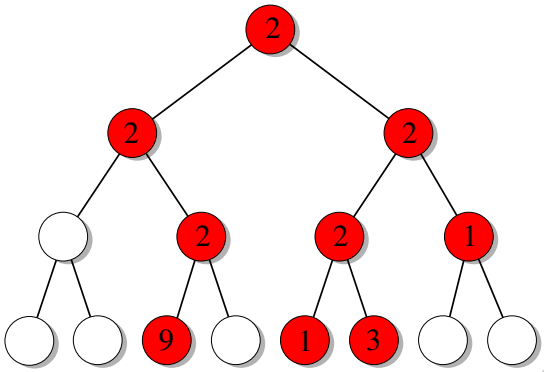

Figure 4.1 Example Q-Digest summary over $U = 0 \ldots 7$

u), which can be up to W_u, by the recursive call in line 9. The remainder of the debt is passed on to the right subtree in line 10. In the case that the weight of the tree rooted v less the current debt is below the current capacity, then line 12 updates the weight of v accordingly. All descendants of v have had all their weight assigned to ancestors, so their weight is now set to zero.

It is clear that this COMPRESS operation preserves the property that the weight associated with each node v is due to input items from the leaves of the subtree of v: since we can view the movement of weights in the tree as going towards the root, it is always the case that the weight of an input item x is associated with a node that is an ancestor of x in the tree. The COMPRESS operation can be implemented to take time linear in the number of nodes, since each node is visited a constant number of times, and is processed in constant time.

MERGE. It is straightforward to perform a MERGE operation on two Q-Digest summaries over the same domain U. We simply merge the nodes in the natural way: the weight of node v in the new summary is the sum of the weights of node v in the two input summaries. Following a MERGE, it may be desirable to perform a COMPRESS to ensure that the space remains bounded.

Example. Figure 4.1 shows a schematic illustration of an example Q-Digest summary. It shows a binary tree structure over the domain $0 \ldots 7$. Shaded nodes have non-zero counts, while empty nodes are associated with zero counts, and are not explicitly represented. Currently, the ca-

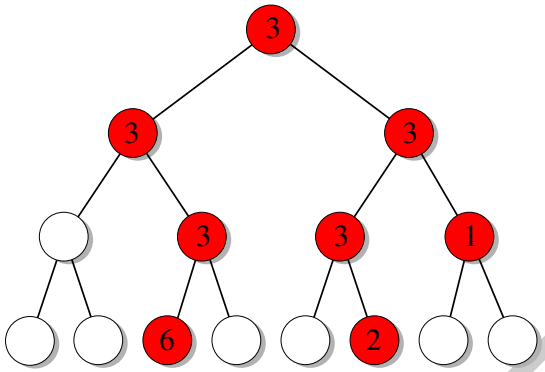

Figure 4.2 Example Q-Digest summary after COMPRESS operation with $C = 3$

pacity of nodes is set to $C = 2$. If we perform a rank query operation for $x = 5$, we obtain an answer of 14, which is the sum of all nodes strictly to the left of the 6th leaf (the one with a weight of 3 in the example of Figure 4.1). The uncertainty in this answer is 6, given by the sum of weights on the path from the root to x. Suppose we UPDATE the summary with an item with $x = 6$ and $w = 1$. Then the algorithm would walk down from the root until it reached the internal node with weight 1. This is below capacity, and so the weight can be increased to 2. Lastly, suppose we perform a COMPRESS operation on the version of the summary shown in Figure 4.1 after setting the capacity of nodes $C = 3$. Then we obtain the result shown in Figure 4.2. Three units of weight are taken from the third leaf to bring its ancestors up to the new capacity. One unit of weight is taken from each of the fifth and sixth leaves to give to their ancestors. This uses up all of the weight of the fifth leaf, so it is no longer stored. Thus, the COMPRESS operation has reduced the number of nodes stored by one.

Further Discussion. Following a COMPRESS operation, the number of nodes with non-zero count is bounded by $O(\log U/\varepsilon)$. This holds because, where possible, nodes now have a weight of C. Some nodes will have weight below capacity, but these nodes have no descendants with positive weight: if they did have such descendants, then the COMPRESS algorithm would have taken all this

weight and distributed it to the ancestors. Consequently, each node below capacity can be associated with its parent which is at capacity; and each node at capacity has at most two children that are below capacity. Suppose there are n_c nodes that are at capacity $C = \varepsilon W/\log U$. Then n_c can be at most $\log U/\varepsilon$, since the weights of all nodes sum to W. Thus there can be at most $O(\log U/\varepsilon)$ nodes with non-zero count in total. This bounds the space of the data structure following a COMPRESS operation.

The algorithm above to perform an insert takes time $O(\log U)$, as it performs a walk along the path from the root to x. Studying this path, it turns out that there should be a prefix of the path where all nodes have weight C, a suffix where all nodes have weight 0, and at most one intermediate node with weight between 0 and C. It is therefore possible to improve the running time to $O(\log\log U)$, by performing a binary search along this path to find the transition between nodes of weight C and nodes of weight 0. This assumes that the nodes are stored in an appropriate dictionary data structure, such as a hash table or BloomFilter.

A limitation of the Q-Digest summary is the requirement that the domain of items U be fixed, and known in advance. The summary is suitable for dealing with items from a well-understood, structured domain, such as 32-bit integers or IP addresses. However, it does not easily apply to arbitrary text strings or arbitrary precision real numbers. For these cases, other summaries such as the GK summary can be used.

Implementation Issues. In the initial phase of the algorithm, it is not sensible to directly follow the UPDATE procedure, since this will lose too much information. Instead, while $W < \frac{1}{\varepsilon}\log U$, it is better to simply buffer the input items (and so provide exact answers to queries). Only when W exceeds this bound is it suitable to switch from buffering to using the approximate Q-Digest structure.

For a given element x, let r be the sum of the weights of all nodes that are strictly less than x, and Δ be the sum of the weighs of all ancestor nodes of x. As discussed about, we have $r \le \text{rank}(x) \le r + \Delta$ and $\Delta < \varepsilon W$, and we use r to estimate $\text{rank}(x)$. But $r + \Delta/2$ is a better estimate, whose (two-sided) error is at most $\Delta/2$. Using this estimator, we may double the threshold C and thus make the summary even more compact.

When there are a series of QUERY operations, we may pre-process the summary in order to speed up the query process. For each node in

the summary, we calculate the estimated rank of the largest value in the range represented by that node. All these ranks can be calculated with a post-order traversal, which can be done in time linear in the size of the summary. Now to find the rank of a given element x, we return the estimated rank of the largest value that is no greater than x; to find the element with a given rank, we return the item in the list whose estimated rank is closest to the given one. Both operations can be done efficiently with binary search.

The summary as defined above does not specify exactly when to perform a COMPRESS operation. This can be done whenever needed, without affecting the guarantees of the structure. For example, one could even perform a COMPRESS after every UPDATE operation, although this would slow down performance considerably. To keep the space bounded at $O(\log U/\varepsilon)$, we could only COMPRESS every time the weight W doubles: using the analysis from the discussion of COMPRESS, we still have that each node below capacity C is the child of a node whose weight is C, and we can have only $O(\log U/\varepsilon)$ such parents.

An intermediate strategy is to perform a compress operation every time the capacity C can increase by 1, i.e., every time W increased by $\log U/\varepsilon$. In this case, the amortized cost of an update is dominated by the cost of the UPDATE procedure, since the amortized cost of COMPRESS is $O(1)$ per step. However, in some high-throughput applications, we may want to ensure a worst-case time performance per update. In this case, the periodic execution of COMPRESS (as well as the need to switch over from buffering updates to using the Q-Digest) may cause an unacceptable delay. This can be resolved by performing small pieces of the COMPRESS operation with every UPDATE, at the cost of significantly complicating the implementation.

History and Background. The Q-Digest summary was first proposed by Shrivastava *et al.* in 2004 [204]. This version did not require that the nodes formed a tree: instead, it was proposed to maintain just a set of nodes and associated weights. Cormode *et al.* [67] observed that the same guarantees could be provided while additionally requiring that the nodes formed a tree. This simplifies the COMPRESS operation, and ensures that the space bounds are met following each execution of COMPRESS. This "top-down" version of the Q-Digest is then quite similar to a data structure proposed to track the "hierarchical heavy hitters" in streaming data by Zhang *et al.* , also in 2004 [234].

Available Implementations. Several implementations of Q-Digest are available across various languages, differing in how they choose to implement the COMPRESS operation. The implementation from stream-lib (`https://github.com/addthis/stream-lib/tree/master/src/main/java/com/clearspring/analytics/stream/quantile`) uses a similar root-to-leaf version of COMPRESS as discussed here.

4.2 Greenwald-Khanna (GK)

Brief Summary. The GK summary provides a compact summary of data drawn from an ordered input domain U. In contrast with Q-Digest, the GK summary does not need U to be a domain of integers. It is a comparison-based algorithm, i.e., it works on any ordered domain U where a comparison operator is defined. But it only accepts insertions of unweighted elements. So here the total number of elements inserted into the summary, N, is the same as the total weight W. This summary can be merged but its size grows after each MERGE.

Operations on the summary. The GK summary stores information about its input as a list of triples, of the form (x_i, g_i, Δ_i). Here, x_i is an item from the input, g_i indicates the total number of items that are represented by this triple, and Δ_i records information about the uncertainty in the position of x_i in the sorted order of the input. The summary ensures that, at any time, for each triple (x_i, g_i, Δ_i), we have that

$$\sum_{j=0}^{i} g_j \leq \text{rank}(x_i) + 1 \leq \Delta_i + \sum_{j=0}^{i} g_j, \tag{4.1}$$

$$g_i + \Delta_i < 2\varepsilon N. \tag{4.2}$$

As a result, we have that the number of input items between x_{i-1} and x_i is at most $2\varepsilon N - 1$.

To INITIALIZE a new GK summary, we create a list which contains only one triple $(\infty, 1, 0)$.

To UPDATE the summary with an item x, we find the smallest i such that $x < x_i$. If $g_i + \Delta_i + 1 < 2\varepsilon N$, we simply increase g_i by 1, in which case no extra space is used. Otherwise we insert a new triple $(x, 1, g_i + \Delta_i - 1)$

right before the i-th triple. Then we try to find an adjacent pair of triples (x_j, g_j, Δ_j) and $(x_{j+1}, g_{j+1}, \Delta_{j+1})$ such that $g_j + g_{j+1} + \Delta_{j+1} < 2\varepsilon N$. If such a pair can be found, we remove the first triple, and increase g_{j+1} by g_j, such that the size of the list will remain the same despite of the new element. If there is not such a pair, the size of the list is increased by 1. The pseudocode for UPDATE is shown in Algorithm 4.4.

Algorithm 4.4: GK: UPDATE (x)

```
1 N ← N + 1;
2 Find the smallest i such that x < x_i ;
3 if g_i + Δ_i + 1 < 2εN then
4 |   g_i ← g_i + 1 ;
5 else
6 |   Insert (x, 1, g_i + Δ_i − 1) before the i-th triple ;
7 |   if ∃j : (g_j + g_{j+1} + Δ_{j+1} < 2εN) then
8 |   |   g_{j+1} ← g_j + g_{j+1} ;
9 |   |   remove the j-th triple ;
```

To MERGE two GK summaries, we perform a merge of the sorted lists of triples. Let the head of the two lists be (x, g, Δ) and (x', g', Δ'), and assume that $x \leq x'$. Then we remove the first triple from its list, and create a new triple in the output GK structure with $(x, g, \Delta + g' + \Delta')$. When one list is exhausted, we just copy over the tail of the other list. This has the effect of combining the bounds on the ranks in the correct way. In the end we try to reduce the size by removing some tuples as in UPDATE. This is spelled out in Algorithm 4.5.

Algorithm 4.5: GK: MERGE (S, T)

```
1  i ← 0;
2  j ← 0;
3  R ← ∅;
4  while i < |S| and j < |T| do
5      if S_i.x ≤ T_j.x then
6          R ← R ∪ (S_i.x, S_i.g, S_i.Δ + T_j.Δ + T_j.g − 1);
7          i ← i + 1;
8      else
9          R ← R ∪ (T_j.x, T_j.g, S_i.Δ + T_j.Δ + S_i.g − 1);
10         j ← j + 1;
11 while i < |S| do
12     R ← R ∪ (S_i.x, S_i.g, S_i.Δ);
13     i ← i + 1;
14 while j < |T| do
15     R ← R ∪ (T_j.x, T_j.g, T_j.Δ);
16     j ← j + 1;
17 while there is j such that g_j + g_{j+1} + Δ_{j+1} < 2εN do
18     g_{j+1} ← g_j + g_{j+1} ;
19     remove the j-th triple ;
```

To QUERY a GK summary to find the estimated rank of a given item y, we scan the list of triples to find where it would fall in the summary. That is, we find triples (x_i, g_i, Δ_i) and $(x_{i+1}, g_{i+1}, \Delta_{i+1})$ so that $x_i \leq y < x_{i+1}$. We have

$$\sum_{j=0}^{i} g_j - 1 \leq \text{rank}(x_i) \leq \text{rank}(y) \qquad \text{and}$$

$$\text{rank}(y) \leq \text{rank}(x_{i+1}) \leq \sum_{j=0}^{i+1} g_j + \Delta_{i+1} - 1.$$

This bounds the rank of y, and leaves uncertainty of at most $g_{i+1} + \Delta_{i+1}$. So we return $\sum_{j=0}^{i} g_j - 1 + (g_{i+1} + \Delta_{i+1})/2$ as the approximate rank of y. Therefore, the maintenance algorithms ensure that this quantity is bounded. Similarly, suppose we want to QUERY to find an item of rank approximately r. Then we seek for a triple (x_i, g_i, Δ_i) so that

input	N	$\lceil 2\varepsilon N \rceil$	GK summary
	0	0	$(\infty, 1, 0)$
1	1	1	$(1, 1, 0), (\infty, 1, 0)$
4	2	1	$(1, 1, 0), (4, 1, 0), (\infty, 1, 0)$
2	3	2	$(1, 1, 0), (2, 1, 0), (4, 1, 0), (\infty, 1, 0)$
8	4	2	$(1, 1, 0), (2, 1, 0), (4, 1, 0), (8, 1, 0), (\infty, 1, 0)$
5	5	2	$(1, 1, 0), (2, 1, 0), (4, 1, 0), (5, 1, 0), (8, 1, 0), (\infty, 1, 0)$
7	6	3	$(1, 1, 0), (2, 1, 0), (4, 1, 0), (5, 1, 0), (7, 1, 0), (8, 1, 0), (\infty, 1, 0)$
			$(1, 1, 0), (2, 1, 0), (4, 1, 0), (5, 1, 0), (8, 2, 0), (\infty, 1, 0)$
6	7	3	$(1, 1, 0), (2, 1, 0), (4, 1, 0), (5, 1, 0), (6, 1, 1), (8, 2, 0), (\infty, 1, 0)$
			$(2, 2, 0), (4, 1, 0), (5, 1, 0), (6, 1, 1), (8, 2, 0), (\infty, 1, 0)$
7	8	4	$(2, 2, 0), (4, 1, 0), (5, 1, 0), (6, 1, 1), (7, 1, 0), (8, 2, 0), (\infty, 1, 0)$
			$(2, 2, 0), (4, 1, 0), (5, 1, 0), (6, 1, 1), (8, 3, 0), (\infty, 1, 0)$
6	9	4	$(2, 2, 0), (4, 1, 0), (5, 1, 0), (6, 1, 1), (6, 1, 2), (8, 3, 0), (\infty, 1, 0)$
			$(4, 3, 0), (5, 1, 0), (6, 1, 1), (6, 1, 2), (8, 3, 0), (\infty, 1, 0)$
7	10	4	$(4, 3, 0), (5, 1, 0), (6, 1, 1), (6, 1, 2), (7, 1, 2), (8, 3, 0), (\infty, 1, 0)$
			$(4, 3, 0), (6, 2, 1), (6, 1, 2), (7, 1, 2), (8, 3, 0), (\infty, 1, 0)$
2	11	5	$(2, 1, 2), (4, 3, 0), (6, 2, 1), (6, 1, 2), (7, 1, 2), (8, 3, 0), (\infty, 1, 0)$
			$(4, 4, 0), (6, 2, 1), (6, 1, 2), (7, 1, 2), (8, 3, 0), (\infty, 1, 0)$
1	12	5	$(1, 1, 3), (4, 4, 0), (6, 2, 1), (6, 1, 2), (7, 1, 2), (8, 3, 0), (\infty, 1, 0)$
			$(1, 1, 3), (4, 4, 0), (6, 2, 1), (7, 2, 2), (8, 3, 0), (\infty, 1, 0)$

Table 4.1 *Example of a* GK *summary,* $\varepsilon = 1/5$

$$\text{rank}(x_i) \geq \sum_{j=0}^{i} g_j - 1 \geq r - \varepsilon N \text{ and } \text{rank}(x_i) \leq \Delta_i + \sum_{j=0}^{i} g_j - 1 \leq r + \varepsilon N.$$

For such an i, we report x_i as the item with approximate rank r: from (4.1), we have that

$$r - \varepsilon N \leq \text{rank}(x_i) \leq r + \varepsilon N.$$

Note that we can always find such an x_i in the summary. If $r + 1 \leq N - \varepsilon N$ (otherwise, picking the maximum value will suffice), then we consider the first triple such that $\Delta_i + \sum_{j=0}^{i} g_j > r + 1 + \varepsilon N$. Then, necessarily, $\text{rank}(x_{i-1}) \leq r + \varepsilon N$, and

$$\text{rank}(x_{i-1}) \geq \sum_{j=0}^{i-1} g_j - 1 = \Delta_i + \sum_{j=0}^{i} g_j - 1 - (g_i + \Delta_i) \geq (r + \varepsilon N) - (2\varepsilon N) = r - \varepsilon N.$$

Example. Table 4.1 shows how a GK summary works with input 1, 4, 2, 8, 5, 7, 6, 7, 6, 7, 2, 1. In this example we set $\varepsilon = 1/5$, and after each insertion, we always remove the first triple that can be removed. In the end we have 6 triples, and Table 4.2 shows the accuracy of this summary.

x	rank(x)	$\tilde{r}(x)$	Error
1	0	2	2
2	2	2	0
4	4	5.5	1.5
5	5	5.5	0.5
6	6	8	2
7	8	9.5	1.5
8	11	11.5	0.5

Table 4.2 QUERY *with the example* GK *summary*

We need to estimate each rank within error of $\varepsilon N = 2.4$, and indeed the summary can estimate all ranks well.

Further Discussion. The original paper [119] introduced two versions of the GK summary, a complicated one with a strict bound on the summary size, and a simplified one without proven guarantees on its size (hence also update and query time). The version introduced above is the second version. Despite the lack of a proof bounding its size, this version of the GK summary performs very well in practice, with size often smaller than the strict version. It is also faster since it is much simpler.

In the strict version, the new triple inserted on the arrival of x is always $(x, 1, \lfloor 2\varepsilon N - 1 \rfloor)$ regardless of the successor triple. And the UPDATE algorithm does not attempt to removed tuples immediately. Instead, a COMPRESS procedure is introduced, which guarantees to reduce the space of the summary to $O\left(\frac{1}{\varepsilon}\log(\varepsilon N)\right)$. The COMPRESS procedure essentially takes the search through the data structure for triples that can be combined out of the UPDATE procedure, but also enforces some additional restrictions on which triples can be merged. The interested readers may find the proof and more details about the strict version in the original paper [119].

The MERGE operation is always guaranteed to be correct, in that it allows us to answer queries with the required accurate. However, performing a MERGE operation doubles the size of the summary. We can then perform a COMPRESS to reduce the size if there is redundancy, but it is not clear that the size will always be reduced substantially. It remains an open question whether the GK summary can be merged while keeping a small size.

Implementation Issues. The list of triples can be kept sorted in a binary tree (e.g., using std::map in C++), such that in line 2 of Algorithm 4.4, we may find i by performing a binary search.

The naive implementation of line 7 of Algorithm 4.4 requires a linear scan of the whole list. Alternatively, we may also maintain an auxiliary min-heap, which manages the values of $g_j + g_{j+1} + \Delta_{j+1}$. Now, we can simply check the minimum value in the heap, and remove the corresponding triple if possible; otherwise it would be safe to claim that no triples can be merged. For each incoming element, the heap may be updated in the same time as UPDATE, so it will not introduce much overhead. Additional speed improvements can be achieved (at the expense of adding some complexity) by standard techniques, such as buffering the most recent set of new items to add to the structure, then sorting them and performing a bulk UPDATE at the same time as a COMPRESS. This will reduce the amortized cost of maintaining the data structure. To answer multiple queries, we can calculate the prefix sums of the g_i's with a linear scan. Subsequently, a query can be answered easily by binary search.

History and Background. The GK summary was introduced by Greenwald and Khanna in 2001 [119]. It has been widely used: for example, in systems for large scale log analysis [196], or for summarizing data arising from networks of sensors [120]. The limitations that no strong guarantees hold for the size of summaries subject to many MERGE operations, and requirement for unit weight input, has led to much interest in generalizations and variations that can overcome these limitations.

While very effective in practice, the bound of $O(\frac{1}{\epsilon}\log \epsilon n)$ is unsatisfying. A lower bound of $\Omega(\frac{1}{\epsilon}\log \frac{1}{\epsilon})$, due to Hung and Ting [131], applies to any deterministic, comparison-based algorithm. The lower bound takes advantage of the assumed deterministic algorithm, to generate input sequences that force the algorithm to keep track of sufficiently much information if it is to give an acceptable algorithm. As shown in the next section, if a small probability of failure is allowed, then this lower bound can be broken with a randomized summary whose size is only $O(\frac{1}{\varepsilon}\log\log \frac{1}{\varepsilon})$.

Available Implementations. Several implementations of GK are available online, across a variety of languages.

4.3 Karnin-Lang-Liberty (KLL)

Brief Summary. The KLL summary provides a summary of data drawn from an ordered domain U. It is a comparison-based summary, just like GK. The size of the summary is $O\left(\frac{1}{\varepsilon}\sqrt{\log\frac{1}{\varepsilon}}\right)$ after any number of UPDATE and MERGE operations. It is more space- and time-efficient than the GK summary, but it is a randomized algorithm that may exceed the ε error bound with a small probability. The version of KLL described below assumes unweighted items; weighted items can also be handled, and a pointer is given under **History and Background**.

Algorithm 4.6: KLL: UPDATE (x)

```
1 B[0] ← B[0] ∪ {x};
2 COMPRESS;
```

Operations on the summary. The KLL summary consists of a list of buffers B of varying capacities. Buffer $B[l]$ is said to be at level l, and all items in $B[l]$ have weights 2^l. Intuitively, each item in $B[l]$ represents 2^l original input items. The capacity of buffer $B[l]$ is defined to be $c^{h-l}k$ (but at least 2), where h is the highest level with a non-empty buffer, $k = O\left(\frac{1}{\varepsilon}\sqrt{\log(1/\varepsilon)}\right)$, and $c \in (0.5, 1)$ is a constant. Note that with $c < 1$, the buffer capacities decrease as we go down the levels.

The UPDATE procedure is shown in Algorithm 4.6. To UPDATE the summary with an item x, we simply add it to $B[0]$. At this point, there is no loss of information. When $B[0]$ reaches its capacity, we perform an operation called *compaction*, (i.e., a COMPRESS routine) which moves half of the items of $B[0]$ to $B[1]$, while resetting $B[0]$ to empty. This in turn may cause $B[1]$ to reach or exceed its capacity, and a series of compaction operations might be triggered as a result, as shown in Algorithm 4.7. Each compaction introduces some error to the rank estimation, but we will show that the accumulated error for any rank is at most εN with high probability, where N is the number of items summarized.

The compaction operation is spelled out in line 3–9 of Algorithm 4.7, where we compact $B[l]$ and move half of items to $B[l+1]$. First, we sort all items in $B[l]$ (ties can be broken arbitrarily). Then with probability $1/2$ we take all items at odd positions, and with probability $1/2$ take all

Algorithm 4.7: KLL: COMPRESS

```
for l = 0, ..., h do
    if |B[l]| ≥ max{2, c^{h−l}k} then
        Sort B[l];
        i ← a fair coin toss;
        if i is heads then
            B[l + 1] ← B[l + 1] ∪ {items at odd positions in B[l]};
        else
            B[l + 1] ← B[l + 1] ∪ {items at even positions in B[l]};
        B[l] ← ∅;
if B[h + 1] has been created then
    h ← h + 1;
```

items at even positions. These items are added to $B[l+1]$, while $B[l]$ is reset to empty.

To MERGE two KLL summaries S_1 and S_2, we first merge the two buffers at the same level (if they exist). Then we perform a series of compaction operations in a bottom-up fashion, just like in the UPDATE algorithm, to make sure that every buffer is below its capacity. See Algorithm 4.8.

Algorithm 4.8: KLL: MERGE (S_1, S_2)

```
for l = 0, ..., max{S_1.h, S_2.h} do
    S.B[l] ← S_1.B[l] ∪ S_2.B[l];
S.COMPRESS;
```

To QUERY about $\text{rank}(x)$ of an element x, we return the value

$$\tilde{r}(x) = \sum_l 2^l \, \text{rank}_l(x),$$

where $\text{rank}_l(x)$ denotes the number of elements in the buffer $B[l]$ that are strictly less than x. This can be done with a single pass over the summary, as shown in Algorithm 4.9. Recall that, each item in $B[l]$ represents 2^l original items, so $\tilde{r}(x)$ is really just the rank of x among the weighted items in all the buffers.

On the other hand, to QUERY for an element whose rank is r, we

Algorithm 4.9: KLL: QUERY (x)

```
r ← 0;
foreach l do
    foreach y ∈ B[l] such y < x do
        r ← r + 2^l
return r
```

return an element x in the summary whose estimated rank $\tilde{r}(x)$ is closest to r.

Example. Suppose that the following table represents the current status of the KLL summary, where we use $k = 8, c = 1/\sqrt{2}$.

l	weight	capacity	$B[l]$
5	32	8	33, 71, 105, 152, 165, 184
4	16	6	61, 112, 123, 175
3	8	4	23, 81, 134
2	4	3	92, 142
1	2	2	16
0	1	2	84

First consider a query that asks for the estimated rank of 100. This is estimated as

$$\tilde{r}(100) = 1 \cdot 1 + 2 \cdot 1 + 4 \cdot 1 + 8 \cdot 2 + 16 \cdot 1 + 32 \cdot 2 = 103.$$

Next, consider the operation UPDATE(44). We first add 44 to $B[0]$, which causes a compaction on $B[0] = \{44, 84\}$. Suppose we take the odd-positioned items. Then $B[0] = \emptyset$, $B[1] = \{16, 44\}$ after the compaction. This in turn will trigger a series of compaction operations, as follows.

1. Compact $B[1]$: Suppose we take the even-positioned items. Then $B[1] = \emptyset, B[2] = \{44, 92, 142\}$.
2. Compact $B[2]$: Suppose we take the odd-positioned items. Then $B[2] = \emptyset, B[3] = \{23, 44, 81, 92, 142\}$.
3. Compact $B[3]$: Suppose we take the even-positioned items. Then $B[3] = \emptyset, B[4] = \{44, 61, 92, 112, 123, 175\}$.
4. Compact $B[4]$: Suppose we take the even-positioned items. Then $B[4] = \emptyset, B[5] = \{33, 61, 71, 105, 112, 152, 165, 175, 184\}$.

5. Compact $B[5]$: Suppose we take the odd-positioned items. Then $B[5] = \emptyset$, $B[6] = \{33, 71, 112, 165, 184\}$.

After UPDATE(4), h has increased by one, so the capacities of the buffers will change (but the weights do not), as follows.

l	weight	capacity	$B[l]$
6	64	8	33, 71, 112, 165, 184
5	32	6	$\emptyset$
4	16	4	$\emptyset$
3	8	3	$\emptyset$
2	4	2	$\emptyset$
1	2	2	$\emptyset$
0	2	2	$\emptyset$

Now on this summary, the result of a rank QUERY for item 100 will result in $64 * 2 = 128$, increased somewhat from before.

Further Discussion. *Error analysis.* As seen above, each item in buffer $B[l]$ represents 2^l original, potentially different, items, so there will be some loss of information after each compaction operation. Consider, e.g., the compaction operation on $B[4] = \{44, 61, 92, 112, 123, 175\}$ in the example above by taking the even-positioned items and moving them to $B[5]$. Before this operation, $B[4]$ contributed 3 items to the computation of $\tilde{r}(100)$, each with a weight of 16. After this operation, only one item, 61, has survived with a weight of 32, so we have incurred an error of -16. On the other hand, if we had taken the odd-positioned items, then two items would survive and we would incur an error of $+16$. So, we accumulate an error of $+2^l$ or -2^l from this compaction, each with equal probability. More importantly, the expected error is 0, meaning that if $\tilde{r}(100)$ was an unbiased estimator, it would remain so. On the other hand, no new error will be introduced for queries like $\tilde{r}(120)$, whether the odd- or even-positioned items are taken.

Before doing any compaction operations, $\tilde{r}(x)$ is obviously accurate. The observations above then imply that, for any x, $\tilde{r}(x)$ is

always an unbiased estimator, with error equal to

$$\text{Err} = \sum_{l=0}^{h-1} \sum_{i}^{m_l} 2^l X_{i,l}, \tag{4.3}$$

where the $X_{i,l}$'s are independent random variables, each taking +1 or −1 with equal probability. Here m_l is the number of compaction operations done on level l, and it can be shown [135, 148] that $m_l = O((2/c)^{h-l})$. This is quite easy to understand: The top level must have not seen any compactions (otherwise level $h+1$ would have been created), so $m_h = 0$. Level $h-1$ have had at most $2/c$ compactions, because each compaction at level $h-1$ promotes $ck/2$ items to level h, so $2/c$ compactions will make level h overflow. In general, because the capacities of two neighboring levels differ by a factor of c, and each compaction on the lower level promotes half of it capacity to the higher level, so every $2/c$ compactions on the lower level triggers one compaction on the higher level, hence $m_l = O((2/c)^{h-l})$. This simple analysis ignores rounding issues (the capacities of the bottom levels are all 2; see the example above), which are handled in [135].

Next, we apply the Chernoff-Hoeffding bound (Fact 1.4) on (4.3):

$$\begin{aligned}
\Pr[|\text{Err} > \varepsilon N|] &\le 2\exp\left(\frac{-2(\varepsilon N)^2}{\sum_{l=0}^{h-1} O((2/c)^{h-l}2^{2l})}\right) \\
&= 2\exp\left(\frac{-2(\varepsilon N)^2}{\sum_{l=0}^{h-1} O((2/c)^{h}(2c)^{l})}\right) \\
&= 2\exp\left(\frac{-2(\varepsilon N)^2}{O((2/c)^{h}) \cdot O((2c)^{h})}\right) \qquad \text{(Recall } c \in (0.5, 1)\text{)} \\
&= 2\exp\left(\frac{-2(\varepsilon N)^2}{O(2^{2h})}\right).
\end{aligned}$$

Since each item on level h represents 2^h original items, and $B[h]$ has a capacity of k, so h will be the smallest integer such that $N \le 2^h k$, or $h \le \log_2(N/k) + 1$. Thus,

$$2^{2h} \le 2^{2\log_2(N/k)+2} = 4 \cdot 2^{\log_2(N/k)^2} = 4(N/k)^2.$$

Thus, setting $k = O(\frac{1}{\varepsilon}\sqrt{\log(1/\varepsilon)})$ will reduce the failure probability $\Pr[|\text{Err} > \varepsilon N|]$ to $O(\varepsilon)$. By the union bound, with at least constant probability, the estimated ranks are accurate (within error of εN) for all the $1/\varepsilon - 1$ elements that rank at $\varepsilon N, 2\varepsilon N, \ldots, (1-\varepsilon)N$, which is

enough to answer all rank queries within 2ε error. Rescaling ε can bring the error down to ε. When all rank queries can be answered within ε error, all the quantile queries can also be answered with the desired error guarantee.

Space and time analysis. Because the buffer capacities decrease geometrically, the total space is dominated by the capacity of the largest buffer at level h, which is $O(k) = O(\frac{1}{\varepsilon}\sqrt{\log(1/\varepsilon)})$. However, since the smallest meaningful buffer capacity is 2, there might be a stack of such small buffers at the lower end, as illustrated in the example above, so the total space is $O(\frac{1}{\varepsilon}\sqrt{\log(1/\varepsilon)}+h) = O(\frac{1}{\varepsilon}\sqrt{\log(1/\varepsilon)}+\log(\varepsilon N))$.

To analyze the time costs, observe that if we keep all buffers sorted, the merge in line 6 or 8 of Algorithm 4.7 can be performed in time proportional to $|B[l]|$. After this compact operation, $|B[l]|/2$ items have been promoted one level higher, so we can change the cost of each compact operation to the promotion of these items. Overall, half of the items have been promoted to level 1, 1/4 to level 2, 1/8 to level 3, ..., k to level h. Thus, the total cost of processing N items is $\sum_{l=1}^{h} \frac{N}{2^l} = O(N)$, and the amortized cost per item is $O(1)$. Furthermore, as all the algorithm does is merging of sorted arrays, this makes the summary very fast in practice due to good cache locality.

By keeping all buffers sorted, a QUERY can also be done more efficiently, by doing a binary search in each buffer. The time needed will be $O(\log^2 \frac{1}{\varepsilon})$. For multiple QUERY operations without UPDATE or MERGE, we may preprocess the summary by assigning each element x from $B[l]$ with a weight 2^l, sorting elements from all buffers and calculating prefix sums of the weights. After that a QUERY can be done by a single binary search, which takes only $O(\log \frac{1}{\varepsilon})$ time.

Further improvements. One can reduce the size of the summary, by observing that the stack of capacity-2 buffers are not really necessary. Every time we compact such a buffer, we simply promote one of its two items randomly, so the stack of buffers can be replaced by a simple sampler, which samples one item out of $2^{h'}$ items uniformly at random, where h' is the level of the highest capacity-2 buffer. This reduces the size of the summary to $O(k) = O(\frac{1}{\varepsilon}\sqrt{\log(1/\varepsilon)})$. However, this makes the MERGE algorithm more complicated, as discussed in detail in [148].

Furthermore, by replacing the top $O(\log\log(1/\varepsilon))$ buffers with the GK summary, the size of KLL can be reduced to $O(\frac{1}{\varepsilon}\log\log\frac{1}{\varepsilon})$ [148]. This improvement in theory may be of less interest in practice: for $\varepsilon = 2^{-16}$ (i.e., less than 1 in 65,000), the absolute values of $\sqrt{\log 1/\varepsilon} = \log\log 1/\varepsilon = 4$. Hence, the overhead in costs for the GK summary are such that we will not see much improvement in space efficiency unless ε is much, much smaller than this already small value. Consequently, this optimization may be considered only of a theoretical interest.

Implementation Issues. As described in [135], there are several techniques that can be applied to improve the practical accuracy and efficiency of KLL. We mention two simple but effective ones here.

First, as observed in the example above, the actual size of the summary fluctuates over time. In most practical scenarios, we are given a fixed space budget. Thus, instead of compacting a buffer as soon as it reaches its capacity, we only have to do so if the total space budget is also exceeded. This means that sometimes buffers may exceed their capacities. Note that this will not affect the error analysis; in fact, this will reduce the error even more, because each compaction will compact a larger buffer, resulting in smaller m_l's.

The second technique reduces randomness via anti-correlation. Again consider the error introduced when compacting $B[4] = \{44, 61, 92, 112, 123, 175\}$ in the example above. As seen above, this compaction introduces an error of ± 16 to $\tilde{r}(100)$, each with equal probability. Suppose at a later time $B[4]$ becomes $\{34, 50, 90, 101, 135, 145\}$, which is about to be compacted again. Now, instead of making a random choice here, we deterministically choose the opposite decision, i.e., if the last compaction chose odd-positioned items, we would choose the even-positioned items this time, and vice versa. This has the effect of canceling the previous error, resulting in a net error of 0. On the other hand, if the contents of $B[4]$ are $\{34, 50, 90, 95, 135, 145\}$, then this compaction will not cancel the previous error, but will not introduce new error, either. So, the total error of the every two compactions is at most ± 16, effectively reducing m_l by half. More precisely, we flip a fair coin for every two consecutive compactions: with probability $1/2$, we do even $\rightarrow$ odd, and with probability $1/2$, we do odd $\rightarrow$ even.

History and Background. Quantile summaries based on the compaction operation date back to Munro and Paterson [186], who gave a simple

algorithm that made multiple passes over the data. By considering the guarantee from the first pass of the Munro-Paterson algorithm, Manku *et al.* [170] showed that this offered an ε accuracy guarantee with a space bound of $O(\frac{1}{\varepsilon}\log^2(\varepsilon N))$. Manku *et al.* [171] combined this with random sampling, giving an improved bound of $O(\frac{1}{\varepsilon}\log^2\frac{1}{\varepsilon})$. By randomizing the compaction operation (choose odd- and even-position items with equal probability), Agarwal *et al.* [2] pushed the space down to $O(\frac{1}{\varepsilon}\log^{1.5}\frac{1}{\varepsilon})$, and they also show how to merge their summaries. The space was then further improved to $O(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon})$ by Felber and Ostrovsky [99], and finally to $O(\frac{1}{\varepsilon}\log\log\frac{1}{\varepsilon})$ by Karnin, Lang, and Liberty [148]. The algorithm described in this section is a simpler version of their algorithm which instead obtains $O(\frac{1}{\varepsilon}\sqrt{\log\frac{1}{\varepsilon}})$. The extra $O(\log\log\frac{1}{\varepsilon})$ factor is due to the requirement that all queries must be correct with a constant probability. If only a single quantile (say, the median) is required with probability $1-\delta$, then the space needed is $O(\frac{1}{\varepsilon}\log\log\frac{1}{\delta})$, which is known to be optimal [148].

Finally, the KLL summary can also handle weighted items. The size remains $O(\frac{1}{\varepsilon}\sqrt{\log(1/\varepsilon)})$ but the time to UPDATE the summary becomes $O(\log(1/\varepsilon))$ [135].

Available Implementations. Implementations of KLL are available across multiple languages. There is a reference implementation from the authors themselves in Python (`https://github.com/edoliberty/streaming-quantiles`), which shows how the COMPRESS operation can be written very concisely. There is a more robust implementation in the DataSketches library in Java (`https://datasketches.github.io/docs/Quantiles/KLLSketch.html`). The DataSketches implementation is compared with the antecedent method from Agarwal *et al.* [2], as well as to various heuristic approaches. Heuristic methods for quantile tracking, such as t-digest and moment sketch, have been widely implemented, but are shown to have poor performance on some input data sets that can occur. The implementation of KLL is shown to have strong accuracy in practice, while performing tens of millions of updates per second.

4.4 Dyadic Count Sketch (DCS)

Brief Summary. The DCS summary provides a compact summary of data drawn from an integer domain $[U] = [0, 1, \ldots, U-1]$. It takes (x, w) pairs as input where w can be an integer that is either positive or negative. But the multiplicity of x for any x in the data set must remain non-negative when the summary is queried for it to be meaningful. It is a randomized algorithm. This summary takes space $O(\frac{1}{\varepsilon}\log^{1.5} U \log^{1.5}\frac{\log U}{\varepsilon})$, and returns all ranks within εW error with constant probability, where W is the total weight of input integers. The summary works upon a *dyadic* structure over the universe U and maintains a Count Sketch structure for frequency estimation in each level of this dyadic structure.

Operations on the summary. The DCS structure is represented as $\log U$ levels of sub-structures. Each one supports point queries over a partition of $[U]$ into subsets, which we refer to as a "reduced domain of $[U]$". These $\log U$ levels make up a dyadic structure imposed over the universe. More precisely, we decompose the universe U as follows. At level 0, the reduced domain is $[U]$ itself; at level j, the universe is partitioned into intervals of size 2^j, and the projected domain consists of $U/2^j$ elements which are intervals in $[U]$: $[0, 2^j-1], [2^j, 2\cdot 2^j-1]], \ldots, [U-2^j, U-1]$; the top level thus consists of only two intervals: $[0, U/2-1], [U/2, U-1]$. Each interval in every level in this hierarchy is called a *dyadic interval*. Each level keeps a frequency estimation sketch that should be able to return an estimate of the weight of any dyadic interval on that level. Specifically, the DCS makes use of a Count Sketch at each level but with a different set of parameters. Note that in the j-th level, an element is actually a dyadic interval of length 2^j, and the frequency estimation sketch summarizes a reduced universe $[U/2^j]$. So for j such that the reduced universe size $U/2^j$ is smaller than the sketch size, we should directly maintain the frequencies exactly, rather than using a sketch. We call a level a *lower level* if it maintains a Count Sketch, and an *upper level* if it maintains the frequencies exactly.

Recall that the Count Sketch consists of an array C of $\omega \times d$ counters. For each row, we use pairwise independent hash functions: $h_i : [U] \to \{0, 1, \ldots, \omega-1\}$ maps each element in $[U]$ into a counter in this row, and $g_i : [U] \to \{-1, +1\}$ maps each element to -1 or $+1$ with equal probability. To update with (x, w) in the sketch, for each row i, we add $g_i(x) \cdot w$ to $C[i, h_i(x)]$. To estimate the frequency of x, we return the median of

$g_i(x) \cdot C[i, h_i(x)]$, $i = 1, \ldots, d$. Note that the Count Sketch has size $\omega \cdot d$, which can be large. For the highest (upper) levels of aggregation, it is more space efficient to simply keep exact counts. So we only use such a sketch for levels $0, 1, 2, \ldots, s = \lfloor \log(\frac{U}{\omega \cdot d}) \rfloor$.

The INITIALIZE operation creates a dyadic structure over the universe U. For each upper level j, we allocate an array $D_j[.]$ of $U/2^j$ counters for recording frequencies of all dyadic intervals. For each lower level j, we allocate a Count Sketch structure, i.e., an array $C_j[.,.]$ with parameters $\omega = \frac{1}{\varepsilon}\sqrt{\log U \log(\frac{\log U}{\varepsilon})}$, $d = \log(\frac{\log U}{\varepsilon})$.

The UPDATE operation of item x with weight w performs an update on each of the $\log U$ levels. An update operation on level j consists of two steps: (1) map x to an element on this level, denoted by k; and (2) update the element k in the Count Sketch or the frequency table. In step (1), the element on level j corresponding to x is simply encoded by the first $\log U - j$ bits of x. In step (2), when level j is an upper level, we add the weight w to the counter $D_j[k]$; otherwise, we add $w \cdot g_i(x)$ to each $C_j[i, h_i(k)]$ for $1 \le i \le d$ just as performing an update in the Count Sketch.

Algorithm 4.10: DCS:UPDATE (x, w)

```
1 for j ← 0 to ⌈log U⌉ do
2     if (j > s) then
3         D_j[x] ← D_j[x] + w ;
4     else
5         for 0 ≤ i ≤ d do
6             C_j[i, h_i(x)] ← C_j[i, h_i(x)] + w · g_i(x) ;
              // UPDATE the jth Count Sketch
7     x ← ⌊x/2⌋;
```

Algorithm 4.10 shows the pseudocode to update the data structure with an item x of weight w. The for-loop traverses the dyadic levels from bottom to top. Within this loop, it updates the structures on each level. At the start x is the full item identifier; after each step it updates x to be the index for the next level, by halving it and ignoring any remainder. If the level index $j > s$, it simply adds the weight w to the x-th counter of D, else it updates d counters in the sketch array C.

Rank Queries. The routine "rank QUERY" takes a parameter $x \in U$, and returns the (approximate) rank of x. We first decompose the interval $[0, x-1]$ into the disjoint union of at most $\log U$ dyadic intervals, at

most one from each level; then we estimate the weight of the interval from each level, and add them up. More precisely, we can view this dyadic structure as a binary tree. The rank QUERY procedure can be carried out by performing a binary search for x, which yields a root-to-leaf path in the binary tree. Along this path, if and only if a node is a right child, its left neighbor corresponds to a dyadic interval in the dyadic decomposition of range $[0, x)$. For these dyadic intervals, we can query them from the corresponding frequency estimation structures: for the k-th dyadic interval on an upper level j, its frequency is $D_j[k]$; and for the k-th dyadic interval on a lower level j, its frequency is $\text{median}_{1 \le i \le d}\{C_j[i, h_i(k)]g_j(k)\}$.

Algorithm 4.11: DCS:rank QUERY (x)

```
1 R ← 0 ;
2 for i ← 0 to ⌈log U⌉ do
3     if x is odd then
4         if j > s then
5             R ← R + D_j[x − 1] ;
6         else
7             R ← R + median_{0≤i≤d}{C_j[i, h_i(x − 1)] · g_i(x − 1)} ;
8     x ← ⌊x/2⌋;
9 return R;
```

Algorithm 4.11 shows the pseudocode to query for the rank of x for an given item x. The algorithm proceeds down from the top level. It uses a variable R recording the sum of weights of all dyadic intervals in the dyadic decomposition of $[0, x - 1]$. When this algorithm halts, R gives an estimate of the rank of the queried item.

Quantile query. However, for a quantile query, we want to find which item achieves a particular rank. So we provide a second "quantile QUERY" function that takes parameter $\phi \in [0, W - 1]$ as the desired rank, and returns the integer x whose rank is approximately ϕ. We still view this dyadic structure as a binary tree. The quantile QUERY can be carried out by performing a binary search. More precisely, supposing the procedure arrives at a node u, it computes the rank of u's right child in the reduced universe on that level from the corresponding frequency estimation structures, and check whether it is smaller than ϕ. If yes, it visits u's right child; otherwise it visits u's left child. At the end, this proce-

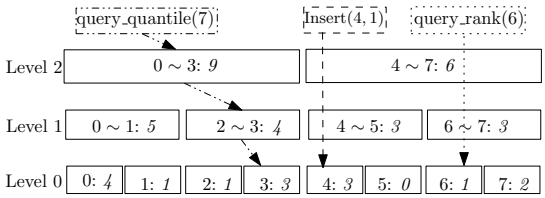

Figure 4.3 Example DCS summary over $U = 0 \dots 7$

dure reaches a leaf corresponding to an integer x at the bottom level, which is then returned.

Algorithm 4.12: DCS:quantile QUERY (ϕ)

```
1  x ← 0;
2  R ← 0 ;
3  for j ← ⌈log U⌉ down to 0 do
4  |   x ← 2 · x ;
5  |   if j > s then
6  |   |   M ← D_j[x];
7  |   else
8  |   |   M ← median_{0≤i≤d}{C_j[i, h_i(x)]g_i(x)};
9  |   if R + M < φ then
10 |   |   x ← x + 1;
11 |   |   R ← R + M;
12 return x;
```

Algorithm 4.12 shows the pseudocode to query for the integer x whose (approximate) rank interval contains the given rank ϕ. The algorithm proceeds down from the top level. It uses a variable x to indicate the index of the node visited by it, and uses another variable R recording the rank of the visited node in its reduced universe. In addition, another variable M is used to store the (approximate) weight of the left child of the current visited node. Obviously, if $R + M < \phi$, the rank of its right child is smaller than ϕ and it should go to the right child, i.e., update x and R; else it goes to the left child with no change to x and R.

Example. Figure 4.3 shows a DCS summary. It maintains a dyadic structure over the domain $0 \dots 7$. The j-th row consists of all dyadic intervals of the form $[b \cdot 2^j, (b+1)2^j - 1]$ for $b = 0, 1, \dots, 8/2^j - 1$, and maintains a Count Sketch for the reduced universe consisting of all the dyadic

intervals. Each rectangle in the figure corresponds to a dyadic interval, labeled by its dyadic range and its estimated weight from the Count Sketch and the stored counts in arrays D_j. Suppose we UPDATE the summary with an item 4 with weight 1. The algorithm walks down the levels, and at each level it finds the rectangle where the item 4 falls into. In this example, the dyadic intervals that need to be updated are those labeled $(4 \sim 7)$ on level 2, labeled $(4 \sim 5)$ on level 1, and labeled 4 on level 0. Suppose we QUERY the rank of a given integer $x = 6$ on the summary. The algorithm again walks down these levels and visits the rectangles labeled $(4 \sim 7)$ on level 2, labeled $(6 \sim 7)$ on level 1 and labeled 6 on level 0. For each rectangle, the algorithm checks whether it corresponds to an odd number in the reduced universe. If yes, it adds the weight of its left neighbor to the final estimation. In this example, the rectangles on level 2 and 1 satisfy this condition, so it adds the weights of rectangle labeled $(0 \sim 3)$ on level 2 and rectangle labeled $(4 \sim 5)$ on level 1 together and return 12 as the estimation. Suppose we QUERY the quantile of a given rank $\phi = 7$ on this summary. This algorithm walks down these levels by performing a binary search and visits the rectangles labeled $(0 \sim 3)$ on level 2, labeled $(2 \sim 3)$ on level 1 and labeled 3 on level 0, and reaches element 3 on the bottom level.

Further Discussion. In principle, any summary that provides good count estimation for multisets could be used in place of the Count Sketch, such as the Count-Min Sketch. However, the Count Sketch is chosen as the workhorse of this summary as it has additional properties that allow it to obtain a better space/accuracy trade off. Specifically, rather than just add up the accuracy bounds of the different sketches pessimistically, we combine them together to give a tighter analysis of the distribution of the errors.

It should be clear that the DCS summary has space cost $O(\omega \cdot d \cdot \log U) = O(\frac{1}{\varepsilon} \log^{1.5} U \log^{1.5}(\frac{\log U}{\varepsilon}))$ and its update time is $O(\log U \log(\frac{\log U}{\varepsilon}))$. Next we analyze its accuracy when answering a rank QUERY. An each level, note that the Count Sketch produces an unbiased estimator. Since we add up the estimates from $\log U$ sketches, it is likely that some of the positive and negative errors will cancel each other out, leading to a more accurate final result. Below, we formalize this intuition. Let us consider the estimator $Y_i = g_i(x) \cdot C[i, h_i(x)]$ on each level. Clearly each Y_i is unbiased, since $g_i(x)$ maps to -1

or +1 with equal probability. Let Y be the median of $Y_i, i = 1 \ldots, d$ (assuming d is odd). The median of independent unbiased estimators is not necessarily unbiased, but if each estimator also has a symmetric pdf, then this is the case. In our case, each Y_i has a symmetric pdf, so Y is still unbiased. To keep the subsequent analysis simple, we let $\varepsilon' = \varepsilon \Big/ \sqrt{\log U \log(\frac{\log U}{\varepsilon})}$. Then using the same Markov inequality-based argument as for the Count-Min sketch, we have

$$\Pr[|Y_i - \mathsf{E}[Y_i]| > \varepsilon' W] < 1/4.$$

Since Y is the median of the Y_i's, by a Chernoff bound, we have

$$\Pr[|Y - \mathsf{E}[Y]| > \varepsilon' W] < \exp(-O(d)) = O\left(\frac{\varepsilon}{\log U}\right).$$

Now consider adding up $\log U$ such estimators; the sum must still be unbiased. By the union bound, the probability that every estimate has at most $\varepsilon' W$ error is at least $1 - O(\varepsilon)$. Conditioned upon this event happening, we can use Hoeffding's inequality (Fact 1.4) to bound the probability that the sum of $\log U$ such (independent) estimators deviate from its mean by more than $t = \varepsilon W$ as

$$2\exp\left(-\frac{2t^2}{(2\varepsilon' W)^2 \log U}\right) = 2\exp\left(-\frac{2t^2 \log(\frac{\log U}{\varepsilon})}{(2\varepsilon W)^2}\right) = O\left(\frac{\varepsilon}{\log U}\right).$$

We can understand a quantile query in terms of this analysis of rank queries: we obtain a sufficiently accurate approximation of each quantile under the same conditions that a rank query succeeds, and so each quantile is reported accurately except with probability $O(\varepsilon/\log U)$. Applying another union bound on the $1/\varepsilon$ different quantiles, the probability that they are all estimated correctly is at least a constant. We can further increase d by a factor of $\log 1/\delta$ to reduce this error probability to δ.

Implementation Issues. In a DCS structure, at each level, it uses a Count Sketch that is $\omega \times d$ array of counters. The above settings of ω and d is chosen to ensure that the analysis will go through. In practice, it is usually sufficient to set parameters smaller, with $\omega = \frac{\sqrt{\log U}}{\varepsilon}$ and $d = 5$ or 7.

History and Background. The general outline of using a dyadic structure of aggregations, and expressing a range in terms of a sum of a bounded number of estimated counts has appeared many times. The choice of which summary to use to provide the estimated counts then has knock-on effects for the costs of the compound solution. Gilbert *et al.* [114] first proposed the *random subset sum* sketch for this purpose, which results in an overall size of $O(\frac{1}{\varepsilon^2} \log^2 U \log(\frac{\log U}{\varepsilon}))$. The disadvantages of this method are that the time to update was very slow, and the leading factor of $\frac{1}{\varepsilon^2}$ is very large, even for moderate values of ε. Cormode and Muthukrishnan introduced the Count-Min sketch and showed how when used within the dyadic structure it can reduce the overall size by a factor of $\frac{1}{\varepsilon}$ [72]. This use of DCS was proposed and analyzed in [224]. Directly placing the Count Sketch in the dyadic structure and applying its standard bounds would not provide the right dependence in terms of ε, so a new analysis was made that reduced the overall size to $O(\frac{1}{\varepsilon} \log^{1.5} U \log^{1.5}(\frac{\log U}{\varepsilon}))$. Further applications of dyadic decompositions with summaries are discussed in Section 9.3.

Available Implementations. Compared to the other summaries surveyed in this chapter, the DCS has attracted fewer implementations, to the extent that there do not appear to be any conveniently available online. Code generally implementing dyadic sketches for range queries is available, for example in the Apache MADlib library, `https://github.com/apache/madlib`.