9

Other Uses of Summaries

In this chapter, we discuss some other applications and manipulations for working with summaries. These include nearest neighbor search; reducing the significance of older updates; ways to combine summaries with other data transformations; and operations on summaries such as re-weighting and re-sizing.

9.1 Nearest Neighbor Search

The *nearest neighbor search* problem, also known as *similarity search*, is defined as follows: Given a set P of n points in a metric space with distance function D, build a data structure that, given any query point q, returns its nearest neighbor $\arg\min_{p \in P} D(q, p)$. This problem has a wide range of applications in data mining, machine learning, computer vision, and databases. However, this problem is known to suffer the "curse of dimensionality", i.e., when the dimensionality of the metric space is high, there does not seem to be a method better than the naive solution, which, upon a query, simply scans all the points and computes $D(p, q)$ for every $p \in P$.

The past few decades have seen tremendous progress on this problem. It turns out that if some approximation is allowed, then the curse of dimensionality can be mitigated, and solutions significantly more efficient than the linear search method exist. Many of them are actually based on computing a small summary for each point in P, as well as for the query point q. If the summaries are "distance-preserving", then the nearest neighbor of q can be found by searching through the summaries or performing a few index look-ups, which can be more efficient than searching over the original points in P.

Most solutions along this direction actually solve the approximate *near neighbor* problem, which is parameterized by an approximation factor $c > 1$ and a radius $r > 0$. This problem asks the data structure to return a $p \in P$ such that $D(p, q) \leq cr$, provided that there exists a point p' with $D(p', q) \leq r$. If no such p' exists, the data structure may return nothing. This is in effect a "decision version" of the nearest neighbor problem. Thus, to find the (c-approximate) nearest neighbor, one can build multiple copies of the data structure with $r = \Delta_{\min}, c\Delta_{\min}, c^2\Delta_{\min}, \ldots, \Delta_{\max}$, where $\Delta_{\min}$ and $\Delta_{\max}$ are the smallest and largest possible distances between the query and any point in P, respectively. In practice, this solution is often sufficient, as one does not have to set $\Delta_{\min}$ and $\Delta_{\max}$ to be the true minimum and maximum distances; instead, the user may use a range that is of his/her interest: a nearest neighbor with distance too large, even if found, may not be very interesting; on the other hand, neighbors with distances smaller than some $\Delta_{\min}$ may be equally good. Theoretically speaking, this approach leads to a space complexity that is not bounded by any function of n. However, more complicated reductions that incur a $\log^{O(1)} n$ loss do exist [124].

In this section, we review two types of solutions for the approximate near neighbor (ANN) problem based on computing small summaries over the points. The first achieves logarithmic query time, but with a high requirement on space; the second one strikes a better balance between space and time, and is used more often in practice.

9.1.1 ANN via Dimension Reduction

The Johnson-Lindenstrauss Transform as described in Section 5.5 is a summary over points in $\mathbb{R}^d$ that preserves the ℓ_2-distance up to a $(1+\varepsilon)$-factor. Most importantly, by setting $\delta = 1/n^2$, we see from Section 5.5 that the size of the summary, i.e., the dimensionality of the transformed points, is only $k = O\left(\frac{\log n}{\varepsilon^2}\right)$, which is completely independent of d. Thus, it offers an efficient way to solve high-dimensional ANN, by searching over the transformed points in $\mathbb{R}^k$.

More precisely, we perform the Johnson-Lindenstrauss Transform on every point $p \in P$. Let $f(p) \in \mathbb{R}^k$ be the transformed point of p. We build an ANN data structure over $f(P) = \{f(p) \mid p \in P\}$, as follows. For a given radius r, we discretize the space $\mathbb{R}^k$ into cubes of side length $\varepsilon r/\sqrt{k}$. Note that each cube can be identified by a k-integer tuple. For every $p \in P$, let $S(p)$ be the set of cubes that intersect the ball $B_{f(p),r}$. We

build a dictionary data structure[1] mapping each cube to an arbitrary point p such that the cube is in $S(p)$. If there is no such p, this cube will not be stored in the dictionary. For a given query q, we just compute $f(q)$, locate the cube that contains $f(q)$, and use the dictionary structure to find the cube and report its associated point p. If the cube does not exist in the dictionary, we return nothing.

Below, we will show that the algorithm solves the ANN problem with an approximation ratio of $c = 1 + 3\varepsilon$. The space needed by the data structure is $O(dn) + n^{O(\log(1/\varepsilon)/\varepsilon^2)}$ and the query time is $O\left(\frac{d \log n}{\varepsilon}\right)$. Note that by replacing 3ε with ε, the approximation ratio can be scaled to $1 + \varepsilon$ with no change in the asymptotic bounds on space and query time.

Further Discussion. Now we analyze the approximation guarantee provided by the above algorithm. Let q be the query point. We know from the Johnson-Lindenstrauss Transform that, with probability at least $1/2$, we have

$$(1 - \varepsilon)D(p, q) \leq D(f(p), f(q)) \leq (1 + \varepsilon)D(p, q)$$

for all $p \in P$. The following analysis will all be based on this happening.

Let p' be a point in P with $D(p', q) \leq r$. We need to show that the algorithm returns some $p \in P$ with $D(p, q) \leq cr$ for some c. As described, the algorithm will try to find in the dictionary the cube containing q, and returns some p such that this cube is in $S(p)$. Note that this cube must exist in the dictionary; indeed, because $D(p', q) \leq r$, this cube must belong to $S(p')$. But the point associated with this cube may not be p', but some other p such that $B_{f(p),r}$ also intersects this cube. Note that the maximum distance between any two points inside the cube is $\sqrt{k \cdot (\varepsilon r/\sqrt{k})^2} = \varepsilon r$. So if $B_{f(p),r}$ intersects the cube, by the triangle inequality, we must have

$$D(f(p), f(q)) \leq r + \varepsilon r = (1 + \varepsilon)r.$$

[1] A dictionary data structure stores a set S of elements from a discrete universe, each with some associated data. For any given x, the structure can return x and its associated data, or report that x does not exist in S. One concrete solution is a hash table [59], which uses linear space, and answers a query in $O(1)$ time in expectation.

Therefore, we have

$$D(p,q) \le \frac{1}{1-\varepsilon} D(f(p), f(q)) = \frac{1+\varepsilon}{1-\varepsilon} r \le (1+3\varepsilon) r,$$

assuming $\varepsilon \le \frac{1}{3}$. Thus, this algorithm solves the ANN problem with an approximation ratio of $c = 1 + 3\varepsilon$.

It remains to analyze the space and query time of this ANN data structure. The query time is dominated by the Johnson-Lindenstrauss Transform. Using the Sparse Johnson-Lindenstrauss Transform (Section 5.5), this can be done in time $O\left(\frac{d \log n}{\varepsilon}\right)$. The subsequent dictionary lookup takes time $O(k) = O\left(\frac{\log n}{\varepsilon^2}\right)$, which is asymptotically smaller than $O\left(\frac{d \log n}{\varepsilon}\right)$ when $d > \frac{1}{\varepsilon}$. Note that $d \le \frac{1}{\varepsilon}$ is considered low-dimensional and there is no need to use this data structure.

The space needed by the data structure consists of two parts: We first need to store all the raw data points, which take space $O(dn)$. This is linear to the input size and is unavoidable. We also need to store the dictionary, which stores all the distinct cubes that appear in some $S(p)$. The number of such cubes is at most $\sum_{p \in P} |S(p)|$. Recall that $S(p)$ includes all cubes in $\mathbb{R}^k$ within a distance of r to p and each cube has side length $\varepsilon r / \sqrt{k}$. Using standard estimates on the volume of ℓ_2 balls, one can show that $|S(p)| \le (\frac{1}{\varepsilon})^{O(k)}$. Each cube needs space $O(k)$ to store in the dictionary, so the total space needed is

$$n \cdot \left(\frac{1}{\varepsilon}\right)^{O(k)} \cdot k = n^{O(\log(1/\varepsilon)/\varepsilon^2)}.$$

History and Background. The approaches described above were introduced in [134, 124]. A similar approach was introduced in [158] in the context of the Hamming space.

The dimension reduction is needed only when $d = \Omega(\log n)$. For smaller values of d, the ANN data structure described by Arya *et al.* [16] is better, which uses $O(dn)$ space and answers a query in $O(c(d,\varepsilon) \log n)$ time, where $c(d,\varepsilon)$ is a function that exponentially depends on d.

9.1.2 ANN via Locality-Sensitive Hashing

For constant ε, the method described above achieves logarithmic query time, which is ideal, but at the cost of a high polynomial space, which is not practical. Next, we describe an alternative approach, which needs a

smaller space of $O(dn + n^{1+\rho})$ for some $0 < \rho < 1$, although at the cost of increasing the query time from logarithmic to $O(dn^{\rho})$.

This approach, known as *locality-sensitive hashing (LSH)*, restricts the summary of a point p to take a single value. But the "distance-preserving" requirement is also weaker: instead of preserving the distance up to a constant factor, the summary is only required to distinguish between large distances and small distances probabilistically. More precisely, the summary will be a function h, randomly chosen from a family $\mathcal{H}$, such that the following holds for any x, y in the metric space:

- if $D(x, y) \leq r$, then $\Pr[h(x) = h(y)] \geq p_1$;
- if $D(x, y) \geq cr$, then $\Pr[h(x) = h(y)] \leq p_2$.

For this framework to be of use, it is required that $p_1 > p_2$. Indeed, the parameter ρ mentioned above depends on the gap between p_1 and p_2, and will be shown to be $\rho = \frac{\log p_1}{\log p_2}$. Note that when $p_1 > p_2$, we must have $\rho < 1$.

When the data points are taken from $\{0, 1\}^d$ and the distance measure is the Hamming distance, a simple LSH family is *bit sampling*, which just picks a random $i \in \{1, \ldots, d\}$ and maps the point to its i-th coordinate. More precisely, $\mathcal{H} = \{h_i(x) = x_i, i = 1, \ldots, d\}$. When the $D(x, y) \leq r$, x and y have at least $d - r$ equal coordinates, so $p_1 = 1 - \frac{r}{d}$. Similarly, when $D(x, y) > cr$, $\Pr[h(x) = h(y)] \leq p_2 = 1 - \frac{cr}{d}$. Thus, $\rho = \frac{\log(1-\frac{r}{d})}{\log(1-\frac{cr}{d})}$.

Another widely used LSH family is the *MinHash*. Here a data "point" is a set, and the distance measure is the Jaccard similarity $S(x, y) = \frac{|x \cap y|}{|x \cup y|}$. Because a "similarity" is high when two points are close, the definition of the LSH family is then restated as follows:

- if $S(x, y) \geq r$, then $\Pr[h(x) = h(y)] \geq p_1$;
- if $S(x, y) \leq r/c$, then $\Pr[h(x) = h(y)] \leq p_2$.

Let g be a function that maps each element in the universe to a real number in $[0, 1]$ independently. Then MinHash sets $h(x)$ to be the element that has the smallest $g(\cdot)$ value in the set x. It can be easily verified that $\Pr[h(x) = h(y)] = S(x, y)$, so we have $p_1 = r, p_2 = r/c$, and $\rho = \frac{\log r}{\log(r/c)}$. However, such a truly random function g cannot be practically constructed. Instead, an ℓ-wise independent function can be used instead, and it has been shown that with $\ell = O(\log(1/\varepsilon))$, the MinHash function can achieve $\Pr[h(x) = h(y)] = S(x, y) \pm \varepsilon$ [133]. Thus, this is essentially the same as the KMV summary described in Section 2.5, except that we set $k = 1$ and use a hash function with a higher degree of independence instead of pairwise independence.

Below, we describe a data structure for the ANN problem, assuming we have an LSH family $\mathcal{H}$ for a distance function $D(\cdot, \cdot)$; the same approach also works for the case when $S(\cdot, \cdot)$ is a similarity function satisfying the revised LSH definition.

The first attempt would be to simply pick an $h \in \mathcal{H}$ randomly, and build a dictionary structure mapping each unique hash value to the list of points with that hash value. Upon a query q, we search the list of points p with $h(p) = h(q)$, and stop as soon as we find an p such that $D(p, q) \leq cr$. Due to the LSH property, points close to q are more likely to be found than those far away from q. However, the problem with this simple algorithm is that the probabilistic guarantee of the LSH family is quite weak, and we may end up with searching in a long list without finding any qualified p. Indeed, when using the bit-sampling LSH family, there are only two distinct hash values (0 and 1), and each list will have $\Omega(n)$ points!

We take two steps to fix the problem, which entail maintaining multiple independent summaries. First, we pick $k = \log_{1/p_2} n$ functions $h_1, \ldots, h_k \in \mathcal{H}$ independently at random, and form a composite hash function $g(x) = (h_1(x), \ldots, h_k(x))$. We build the dictionary structure using g. This effectively creates more distinct hash values, so will reduce the lengths of the lists. But its also reduces the collision probability, i.e.,

- if $D(x, y) \leq r$, then $\Pr[g(x) = g(y)] \geq p_1^k$;
- if $D(x, y) \geq cr$, then $\Pr[g(x) = g(y)] \leq p_2^k$.

To make sure that close points will be found with high probability, we repeat the whole construction ℓ times, i.e., we pick a total of ℓk functions from $\mathcal{H}$ independently at random, and build ℓ dictionary structures, each using a composite g consisting of k functions. Note that each point will thus be stored ℓ times, once in each dictionary. To save space, we do not need to store the points in full (i.e., all d coordinates) every time. Instead, we just keep one copy of the points, while the lists can just store pointers to the full points. Upon a query q, we check the list of points with hash value $g(q)$ in each of the ℓ dictionaries, and stop as soon as we find a point p with $D(p, q) \leq cr$.

Below, we will show that by setting $\ell = O(n^\rho)$, the algorithm described can solve the ANN problem with at least constant probability, while achieving the claimed space/time bounds mentioned at the beginning of this subsection.

Further Discussion. First, with $\ell = O(n^\rho)$, it is immediate that the space needed is $O(dn + \ell n) = O(dn + n^{1+\rho})$, since we only store one copy of the full points, while only keeping pointers in the ℓ dictionaries.

Below, we analyze the query cost and the probability that the data structure finds a point p with $D(p, q) \leq cr$, provided that there exists a point $p' \in P$ with $D(p', q) \leq r$. Consider the following two events:

$\mathcal{E}_1$: $g(p') = g(q)$ in at least one of the ℓ dictionaries;

$\mathcal{E}_2$: the total number of points p with $D(p, q) \geq cr$ and $g(p) = g(q)$ in all dictionaries is at most 10ℓ.

Note that under $\mathcal{E}_1$, the algorithm will succeed in finding a point p with $D(p, q) \leq cr$, because at least p' can be such a p. Under $\mathcal{E}_2$, we search through at most $O(\ell)$ points. Computing the distance between q and each of these points takes $O(d)$ time, so the total query time will be $O(d\ell) = O(dn^\rho)$. We will next show that $\mathcal{E}_1$ and $\mathcal{E}_2$ each happen with probability at least 0.9. Then by the union bound, they happen simultaneously with probability at least 0.8.

We first analyze $\mathcal{E}_1$. From above, we know that

$$\Pr[g(p') = g(q)] \geq p_1^k = p_1^{\log_{1/p_2} n} = n^{\log_{1/p_2} p_1} = n^{\frac{\log p_1}{\log(1/p_2)}} = n^{-\frac{\log p_1}{\log p_2}} = n^{-\rho}.$$

By setting $\ell = c_1 n^\rho$ for some constant c_1, we have

$$\Pr[\mathcal{E}_1] \geq 1 - (1 - n^{-\rho})^{c_1 n^\rho} \geq 1 - e^{-c_1}.$$

It is clear that this can be made at least 0.9 by choosing a constant c_1 large enough.

Finally, we analyze $\mathcal{E}_2$. The expected number of points p with $g(p) = g(q)$ in one dictionary is at most

$$np_2^k = np_2^{\log_{1/p_2} n} = n^{1+\log_{1/p_2} p_2} = n^{1-1} = 1.$$

So the expected total number of such points in all ℓ dictionaries is at most ℓ. Then by the Markov inequality, we obtain that the probability of exceeding this expectation by a factor of ten or more is at most 0.1, and so $\Pr[\mathcal{E}_2] \geq 0.9$.

History and Background. The LSH framework follows from the work of Indyk and Motwani [134], and was further developed in [115]. It has since been significantly expanded in scope and applicability. Many met-

ric spaces have been shown to admit LSH families. For Hamming space, the bit-sampling family can be shown to have $\rho < 1/c$, which is near-optimal [190]. For Euclidean space, an LSH family based on random projection is shown to achieve $\rho < 1/c$ [80], which has been improved to optimal $\rho = 1/c^2 + o(1)$. These LSH families are "data-independent", i.e., their constructions depend only on the metric space, not the actual point set P. The aforementioned LSH families are optimal when restricted to such data-independent constructions. Recently, significant progress has been made towards data-dependent LSH families, i.e., one is allowed to construct the family after seeing P. This approach is very popular in practice as real-world datasets often have some implicit or explicit structure. This topic is beyond the scope of this volume, and interested readers are referred to excellent surveys [13, 223].

9.2 Time Decay

In our discussion of data summarization so far, we have assumed that while data may arrive incrementally, the importance placed on each update is equal. However, there are scenarios where we wish to down-weight the importance of some items over time: when computing statistics, we might want today's observations to carry more weight than last week's. This notion is referred to as "time decay". There is a large literature on combining summaries with different models of time decay. In this section, our aim is to introduce the key notions, and give some simple examples.

Timestamped data. For time decay to apply, we need that each data item also has an associated timestamp. We can assume that timestamps correspond to particular times, and give a total ordering of the data items. We further assume that items to be summarized are received in timestamp order. Note that in large, distributed systems, these assumptions may be questionable. For the exponential decay model, out of order arrivals can be fairly easily handled, while this is more challenging for sliding windows and other models.

9.2.1 Exponential Decay

The model of exponential decay stipulates that the weight of an item decreases as an exponential function of the difference between its times-

tamp and the current time. That is, item x with timestamp t_x is considered to have a weight of $2^{(-\lambda(t-t_x))}$ at time t. Here, λ is a parameter that controls the rate of the time decay. It can be thought of as encoding the "half-life" of items: every $1/\lambda$ time units, the weight of the item halves again.

Exponential decay can be motivated by analogy with physical processes, such as radioactive decay, where the intensity of radioactive emissions decay according to an exponential pattern. However, exponential decay is popular as a time decay model in part because it is easy to implement. Consider for example a simple counter, where we wish to maintain the exponentially decayed sum of weights of items. Without exponential decay, we simply maintain the sum of all weights. With exponential decay, we similarly maintain a sum of (decayed) weights. Here, all the weights decay at the same (multiplicative) rate, so it suffices to apply the decay factor to the sum. If timestamps are kept as integers (say, number of seconds), then every timestep, we multiply the current counter by the factor $2^{-\lambda}$, and add on any new weights that have arrived in the new timestep. Merging two summaries is achieved by just adding the counters.

In the case that time stamps are treated as arbitrary real values, we modify this approach by keeping a timestamp t_c with the counter c, which records the most recent timestamp of an update to the counter. When a new item with weight w_i and timestamp $t_i \geq t_c$ arrives, we update the counter in two steps. First, we decay the counter to the new timestamp: we update $c \leftarrow c2^{-\lambda(t_c-t_i)}$, and set $t_c \leftarrow t_i$. Then we add in the new item at full weight: $c \leftarrow c + w_i$. A convenient feature of exponential decay is that it can tolerate updates which arrive out of time-sorted order: if $t_i < t_c$, we instead down-weight the item weight, and add it to the counter, via $c \leftarrow c + w_i 2^{-\lambda(t_c-t_i)}$.

Further Discussion. The correctness of these procedures can be understood by expanding out the definitions of exponential decay. For a given time t, the correct value of the decayed count is given by $c_t = \sum_i w_i 2^{-\lambda(t-t_i)}$.

First, observe that to decay the counter to any desired timestamp t', we can simply multiply by an appropriate factor:

$$\begin{aligned} c_{t'} &= \sum_i w_i 2^{-\lambda(t'-t_i)} \\ &= \sum_i w_i 2^{-\lambda(t'-t+t-t_i)} \\ &= \sum_i w_i 2^{-\lambda(t'-t)} 2^{-\lambda(t-t_i)} \\ &= 2^{-\lambda(t'-t)} \sum_i w_i 2^{-\lambda(t-t_i)} = 2^{-\lambda(t'-t)} c_t \end{aligned}$$

It is also immediate from this sum expansion that if we want to update a summary with timestamp t with an item with timestamp t_i, the decayed weight of the item is given by $w_i 2^{-\lambda(t_c-t_i)}$, and this can be just added to the counter.

Exponential decay for other summaries. Due to the simple way that exponential decay can be applied to a counter, we can apply exponential decay to other summaries by simply replacing their internal counters with decayed counters as appropriate. Some care is needed, since not all summaries preserve their guarantees with this modification. Some examples where exponential decay can be straightforwardly applied include the various "sketch" summaries, such as Count-Min Sketch, Count Sketch, AMS Sketch, and ℓ_p sketch. Here, since each count stored in the sketch summary is a (weighted) sum of item weights, they can be replaced with exponentially decayed counters, and we obtain a summary of the exponentially decayed input. This allows us to, for example, estimate the decayed weight of individual items, or the L_p norm of the decayed weight vector.

For other summaries, similar results are possible, although more involved. Replacing the counters in the Q-Digest structure with decayed counters allows a weighted version of quantiles and range queries to be answered, based on the decayed weights of items. Similarly, using decayed counters in the SpaceSaving structure allows a decayed version of point queries and heavy hitters to be answered. In both cases, the operations on the data structures are largely as before, with the addition of decay operations on the counters, but the analysis to show the correctness needs to be reworked to incorporate the decay. An implementation issue for both of these is to take a "lazy" approach to decaying counters. That is, we try to avoid requiring that all counters are decayed to reflect the very latest timestamp t, which may take time pro-

portional to the size of the data structure for each update. Instead, we allow each counter to maintain a local timestamp t_c, and only apply the decay operation whenever that counter is accessed by an operation.

History and Background. The notion of exponential decay applied to statistics is one that has appeared in many settings, and does not appear to have a clear point of origin. For similar reasons, the idea of applying exponential decay to sketches seems to be "folklore". Initial efforts to extend exponential decay to other summaries is due to Manjhi *et al.* [169] for tracking item frequencies, and Aggarwal [6] for sampling with decaying weights. The comments here about generalizing Q-Digest and SpaceSaving are based on a note by Cormode *et al.* [68].

9.2.2 Sliding Windows

A second approach to time decay is to consider only a window of recent updates as being relevant. That is, if every update has an associated timestamp, then we only want to answer queries based on those updates falling within the recent window. A window can be defined either time-based — e.g., we only consider updates arriving within the last 24 hours — or sequence based — e.g., we only consider the most recent 1 billion updates. Following the trend in the scientific literature, we focus our main attention on the sequence based model, where the size of the window is denoted W. The examples we give also allow queries to be posed in the time-based window model.

If the space needed is not an issue, then we could in principle just retain the most recent W updates in a buffer, and compute the desired function of interest on demand for this buffer. However, with the assumption that W is still a large quantity, we will discuss summary structures that use much smaller than W space.

We first consider an approach to computing the count of items falling within a sliding window, then see how this can be generalized to other problems.

Exponential histograms for windowed counts. The *exponential histogram* method allows us to keep track of how many item arrivals occur within a recent window. Our input is defined by a sequence of timestamps t_i that record item arrivals. We assume that these are seen in timestamp order. The structure will allow us to approximately determine how many items arrived within a recent time window of size w and,

conversely, find the window size W containing approximately N recent items.

The structure divides the past into a sequence of buckets, with associated counts and timestamps. The most recent $k+1$ buckets each contain a single item, and record the timestamp at which that item arrived as b_i. The next (upto) $k+1$ buckets have a count of 2, corresponding to two items, and record the timestamp of the older item. Then, the jth collection of (upto) $k+1$ buckets have each bucket holding a count of 2^j and recording the timestamp of the oldest item in each bucket.

Updating the structure is done so that the bounds on the number and weights associated with each bucket are maintained. A new item is placed in a bucket of weight 1 at the head of the list. If this causes there to be more than $k+1$ buckets of weight 1, then we take the oldest two weight 1 buckets and merge them together to obtain a bucket of weight 2, whose timestamp is set to the older of the two timestamps. Similarly, if we obtain more than $k+1$ buckets of weight 2^j, we merge the two oldest such buckets to obtain one new bucket of weight 2^{j+1}.

To find the number of items from timestamp t to the present (time-window model), we add up the weights of all buckets with timestamps more recent than t. Similarly, to estimate the duration of the time window containing N recent items (sequence-based model), we find the most recent timestamp such that the sum of all bucket weights from that point onwards exceeds N.

To limit the space needed, we can delete all buckets whose weight is more than W/k, for a target window size of W. The space of the data structure is bounded by $O(k \log(W/k))$, and queries are answered with an uncertainty in the count of $1/k$, according to the below analysis.

A limitation of this data structure is that it does not allow the MERGE operation: if two exponential histograms have been kept over different inputs, it is likely that they have witnessed sufficiently different patterns of arrivals that we cannot combine their bucket structures and obtain a summary of the union of their inputs.

Further Discussion. According to the above invariants, we have at most $k+1$ buckets of each weight class. Hence, if we only keep buckets of weight up to W/k, there are $\log_2(W/k)$ weight classes, each of which keeps at most $k+1$ buckets. Hence the space needed is $O(k \log_2(W/k))$.

To understand the accuracy of the structure, observe that for the k most recent items, we know their timestamps exactly. For each weight class 2^j, note that there are at least k buckets in each smaller class. Hence, the total weight of more recent items is at least $\sum_{\ell=0}^{j-1} k2^\ell = k2^j$. So the uncertainty in the timestamps we have corresponds to 2^j out of at least $k2^j$ items, which is a $1/k$ fraction. If we set $1/k = \epsilon$ for a target accuracy ϵ, we obtain uncertainty in our queries of ϵ relative error, with space proportional to $O(\frac{1}{\epsilon}\log(\epsilon W))$.

Exponential histograms for summaries. As with exponential decay, the structure of the exponential histogram is sufficiently simple that it can be combined with other summary structures. In this case, the natural thing is to keep the overall structure of the histogram and its buckets, and augment the counts with instances of summaries that can be merged. For example, we could keep a SpaceSaving data structure in each bucket, and follow the merging rules. This would allow us to estimate item frequencies within a sliding window. To find item frequencies from (approximately) timestamp t to the present, we would MERGE together all the summaries in histogram buckets with timestamps more recent than t. This would potentially omit a small number of the oldest updates, at most a $1/k$ fraction of those that do fall in the window. The space cost is then the product of the chosen summary size by the number of buckets. The method is suitably general that it can be applied to all summaries that possess a MERGE method. However, because of the blow-up in space cost, there has been a lot of research to find algorithms for specific problems with reduced space bounds.

Timestamps in summaries. A technique that can be used to introduce sliding windows to certain summaries is to replace binary flags that report the presence of an observation with a timestamp. Two such structures are the BloomFilter and HLL. In both of these, we use bits to record whether an update has been mapped there by a hash function. We can replace these bits with timestamps of the most recent item that has been mapped to that location (or a null value if no item has been mapped there). This allows us to query the summary for an arbitrary time window: given the window, we make a copy of the summary with a 1 bit value in a given location if there is a timestamp that falls within the time window at the corresponding location, and a 0 bit value otherwise. Thus, we obtain a version of the original summary with exactly the configuration as if we had only seen the items that fall within the

queried time window. The space cost is a constant factor blow up, as we replace each bit with a timestamp (typically 32 bits).

History and Background. The exponential histogram was introduced by Datar *et al.* [79], where they also discussed how to generalize it to containing other summary structures. The paper also suggested the idea of replacing bits with timestamps. Other work has considered how to provide efficient solutions for specific problems under sliding windows. For example, Lee and Ting [162] adapt the MG summary to keep information on timestamps to allow sliding window frequency queries, while still only requiring $O(1/\epsilon)$ space. A naive approach which combined MG or SpaceSaving with the exponential histogram above would instead require $O(1/\epsilon^2 \log(\epsilon W))$ space.

Subsequent work by Braverman and Ostrovsky introduced the concept of *smooth histograms* [35]. These allow a broad class of functions to be approximated under the sliding windows model in a black box-fashion, under some assumptions on the function being approximated. The high-level idea is to start a new summary data structure at each time step to summarize all subsequent updates. Many of these summaries can later be deleted while still ensuring accurate answers. "Smoothness" properties of the target function are used to bound the total space needed.

9.2.3 Other Decay Models

We briefly mention other models of time decay.

Latched windows. The effort needed to give strong algorithmic guarantees for sliding window queries may seem too large. A simpler approach known as "latched windows" is tolerate a weaker notion of approximation. For example, if the goal is to monitor the state of a system over the last 24 hours, it may suffice to create a summary for each hour, and merge together the last 24 summaries to approximately cover the last day. If there is no strong need to have the summaries cover the window to the last microsecond, this is a simple and practical approach.

Arbitrary functions via sliding windows. Alternatively, if the desire is to approximate an arbitrary decay function on the data, this can be simulated by making multiple calls to a sliding window summary. We

assume that the decay function is differentiable and is monotone non-increasing as the age increases. The observation is that we can rewrite the value of a decayed query as an integral over time of the derivative of the decay function times the result of the function for the corresponding window. This formulation can be accurately approximated by replacing the continuous integral with a sum over differences in the decay function value, scaled by approximate window queries.

History and Background. The term "latched window" was coined by Golab and Özu [117]. The observation about arbitrary function approximation for sums and counts is due to Cohen and Strauss [55], and applied to summaries by Cormode *et al.* [75].

9.3 Data Transformations

The applicability of summary techniques can be extended when they are applied not to raw data, but to a (usually linear) transformation of the input data. This provides most flexibility when the transformation can be computed on each individual update as it arrives, so the summary (or summaries) processes a sequence of transformed updates.

We have already seen examples of this, in the form of the Dyadic Count Sketch (DCS, Section 4.4). The method can be understood as first mapping each update into a collection of updates, each of which is processed by a separate summary. There are a number of uses of this approach to solve new problems by making use of existing solutions. Moreover, these can be chained together: we apply a sequence of transformations to each update, before it is summarized. The ability to MERGE the original summaries means that we can also MERGE summaries of transformed data.

9.3.1 Dyadic Decompositions for Heavy Hitters

Several of the summaries for multisets (Chapter 3) address the question of finding items from the input domain whose aggregate weight is heavy — often called the "frequent items" or "heavy hitters" problem. For example, the items kept by a MG or SpaceSaving summary include those whose weight is larger than an ϵ fraction of the total weight, $\|v\|_1$. Sketch-based methods, such as Count-Min Sketch and Count Sketch, can also solve this problem, with guarantees in terms of $\|v\|_1$ or $\|v\|_2$.

However, these sketches do not directly make it convenient to find the heavy items. The most direct approach is to QUERY for the estimated weight of every item in the domain — which is very slow for large domains.

Instead, one can make use of the Dyadic Count Sketch (DCS) for this problem. Recall that this applies a "dyadic decomposition" to the domain of the input items, $[U]$. A sketch is kept on the original items, on pairs of adjacent items in the domain, on items grouped into fours, and so on. This allows a recursive top-down search process to be applied to find the heavy items. We begin by querying the weight of the left half and the right half of the domain, using the statistics kept by the DCS. For each dyadic interval whose total weight (according to the DCS) exceeds a ε fraction of the total weight, we split the dyadic interval into its two constitute subintervals of half the length, and recursively proceed. If we reach an interval consisting of a single item, then this can be returned as a "heavy hitter". Provided all items have non-negative weight, then every range containing a heavy hitter item will be a heavy range, and so we will not miss any heavy items.

Further Discussion. The guarantees of this search procedure follow from those for the DCS. Every query posed is transformed into a point QUERY to one of the Count Sketch data structures that make up the DCS structure. Consequently, the error is bounded with sufficient probability. The total number of queries is bounded: assuming that each query meets its accuracy bound, then the number of queries posed to a given level in the dyadic decomposition is bounded, since the total weight at each level is bounded in terms of the total weight of the input. For example, if we are seeking to find all items of weight at least $\varepsilon\|v\|_1$, then there are at most $O(1/\varepsilon)$ dyadic ranges at each level whose true weight exceeds $\varepsilon\|v\|_1$. Consequently, we can bound the time taken for the search as a function of $(1/\varepsilon)$ and the parameters of the sketch. Naively, this cost is polylogarithmic in the size of the domain from which items are drawn, U, since we rescale the accuracy parameter by factors of $\log U$ (see Section 4.4). A slightly improved analysis is due to Larsen *et al.* [160, Appendix E], who show that the expected time to search for all heavy hitters can be kept to $O(1/\varepsilon \log U)$ by keeping sketches of total size only $O(\frac{1}{\varepsilon} \log U)$.

9.3.2 Coding transforms for heavy item recovery.

The problem becomes more challenging when items may have negative weights, which can happen in more general scenarios. Here, the divide and conquer approach can fail, since a heavy item can be masked by surrounding items with the opposite sign. An approach is to still make use of sketch summaries, but to arrange them based on ideas from coding theory.

A first example is based on the Hamming code. Given an m bit binary message, the Hamming code adds $\lceil \log m \rceil$ additional "parity" bits. The first parity bit is computed from the bits at odd locations in the message (i.e., indices 1, 3, 5, ...). The jth parity bit is computed from the bits for which the jth bit of their binary index is a 1. That is, the 2nd parity bit depends on indices 2, 3, 6, 7, 10, 11, ...

This structure can be adapted to give a data transformation. Given items drawn from a domain $[U]$, we map to $\lceil \log U \rceil + 1$ summaries — for concreteness, we will use a Count Sketch, where each instance uses the same parameters and the same set of hash functions. We follow the same pattern: only those items whose jth bit of their index is 1 are mapped into the jth sketch, and processed by the usual UPDATE routine. For the 0'th sketch, we process all updates without any exceptions.

To QUERY this modified summary, we can probe each cell of the sketches in turn. For each cell in the 0'th sketch, we examine the magnitude of the count stored there, $|c|$. If it is above a threshold τ, then there is evidence that there could be a heavy item mapping to this cell — or there could be multiple items mapping there which sum to c. The process attempts to "decode" the identity of an item, using the sketches of the transformed data. The cell will be abandoned if it is not possible to clearly determine the identity of an item.

The aim is to recover each bit of the heavy item's identifier, if there is a heavy item. Consider first the least significant bit (index 1), which determines whether an item's identifier is odd or even. We can inspect the corresponding cell of the first sketch, which has included only the odd items. If the magnitude of this cell, c_1, is also above τ, and $|c - c_1|$ is less than τ, then we can conclude that there is potentially an odd item that is heavy. Meanwhile, if $|c - c_1| \geq \tau$ and $|c_1| < \tau$, we conclude that there is potentially an even item that is heavy. The other two cases are $|c - c_1| < \tau$ and $|c_1| < \tau$, which implies that there are no items in this cell above the threshold of τ; or $|c - c_1| \geq \tau$ and $|c_1| \geq \tau$, which implies that there could be more than one heavy item in this cell. In either of the last

two cases, we abandon this cell, and move on to the next cell in the 0th sketch.

This process can be repeated for every sketch. If at the end we have not abandoned a cell, then we have a bit value (0 or 1) for each location, which can be concatenated to provide an item identifier for an item which is assumed to be heavy. This can be confirmed by, for example, keeping an additional independent Count Sketch to cross-check the estimated weight, and also by checking that the candidate item is indeed mapped to the cell it was recovered from.

Further Discussion. It's clear that the above process will recover some candidate items, but it is less clear whether items which are heavy will be successfully identified. Consider some item which is heavy (above the threshold τ), and assume that it is mapped by the hash function into the zero'th sketch so that the total (absolute) weight of other items in the same cell is less than the threshold τ. Then we can be sure that every test that is applied to determine the value of a particular bit in its item identifier will succeed — we will find the correct value for that bit. Consequently, we will recover the item correctly. Hence, the argument comes down to bounding the amount of weight from other items that collides with it in a particular cell. This can be done with a Markov-inequality argument, similar to the Count-Min Sketch: the expected amount of colliding mass is a $1/w$ fraction of the total mass (measured in terms of either $\|v\|_1$ or $\|v\|_2^2$), where w is the width of the sketch data structure. Choosing τ to be at least $2\|v\|_1/w$, (or at least $2\|v\|_2/\sqrt{w}$ in the ℓ_2 case) means that there is at least a constant probability to find a given heavy item in each row of the sketch. This probability is amplified over the rows of the sketches. If we ensure that the number of rows is $O(\log w/\delta)$, then we ensure that we can find all heavy items with probability at least $1 - \delta$.

History and Background. This Hamming code-inspired approach to finding heavy items is described by Cormode and Muthukrishnan [70, 71], with similar ideas appearing previously in [110]. It is possible to use the same approach with more complex code constructions, see for example [192]. Other approaches are based on the idea of applying an invertible remapping to item identifiers, and breaking these into smaller chunks for sketching, yielding "reversible sketches" [202]; and

the counter-braids approach which use a random mapping to sketch entries of varying capacity, based on low-density parity check codes [167]. Similar ideas are used to construct algorithms with improved space bounds based on coding items so that heavy items appears as "clusters" within a graph [160].

9.3.3 Range Transformations

Given a data vector v indexed over the integer domain $[U]$, a range query $[l, r]$ is to find the quantity $\sum_{i=l}^{r} v_i$. With a summary that allows us to estimate entries of v, we can simply sum up the corresponding estimates. However, the query time, and the error, tends to scale proportional to the length of the range, $|r-l+1|$. The dyadic decomposition allows us to answer range queries more efficiently, and with lower error. This is already provided by the DCS (Section 4.4), where the case of a rank query is discussed. Since a rank query for index x corresponds to the range query $[0, x-1]$, we can answer range queries by computing $\text{rank}(r-1) - \text{rank}(l)$, and obtain error $\epsilon\|v\|_1$.

Range queries generalize to higher dimensions. One approach discussed in Section 5.1 is based on ϵ-nets and ϵ-approximations is effective for a data set of points arriving in d dimensions. However, when data points may arrive and depart, a different approach may be needed. The notion of dyadic decompositions can be extended to multiple dimensions: we perform a dyadic decomposition on each dimension, and summarize vectors of membership of the cross-product of these decompositions. Then each data point falls into $O(\log^d U)$ cells, and the space required to provide an $\epsilon\|v\|_1$ guarantee scales with $O(\log^{2d} U)$. This can be effective for small d, but is often too costly for more than 2-3 dimensions.

History and Background. The idea of dyadic decompositions is a ubiquitous idea in computer science and computational geometry, and has been independently suggested within the context of data summarization several times. An early example is due to Gilbert *et al.* [114]. Discussion and evaluation of different sketching approaches to range query estimation is given by Rusu and Dobra [84, Section 6], in the context of estimating the join size between database relations (which is equivalent to inner products of vectors).

9.3.4 Linear Transforms

A large class of data transformations can be described as linear transformations. That is, if the data can be considered to define a vector, v, the transformation can be considered to be a (fixed) matrix A so that the transformed data is the vector Av. Note that our above examples, such as the dyadic decomposition, meet this definition of a linear transformation.

Many other transformations meet this definition, in particular basis transformations, where in addition the (implicit) transformation A represents an (orthonormal) basis transformation, so that $AA^T = I$. That is, distinct rows of A are orthogonal, and the (Euclidean) norm of each row is one. These naturally compose with sketching techniques. Since many of the sketch summaries we have seen, such as Count-Min Sketch, Count Sketch, AMS Sketch and Sparse JLT are also linear transformations, we can write the joint action of a sketch S and a transformation A as the product SA. Further, the structure of the transform may allow it to be applied quickly. We give two examples that have been used in the context of data summaries.

Discrete Haar Wavelet Transform (DHT). The Discrete Haar Wavelet Transform (Haar transform, or wavelets for short) is used extensively in signal processing as a way of transforming the data to extract high level detail. Similar to other transformations such as (discrete) Fourier transforms, it transforms an input vector of U entries into a vector of U wavelet coefficients. The inverse transform of a set of coefficients rebuilds the corresponding input vector. It is often used as the basis of data compression, since reducing the accuracy with which some coefficients are stored, or dropping some entirely, still allows a fairly accurate reconstruction of the original vector. Another useful feature of the transform is that (one-dimensional) range queries can be answered by combining only $O(\log U)$ wavelet coefficients (similar to the dyadic decomposition). Last, the transform is an orthonormal basis, as described above, and so the Euclidean norm of of the coefficients is equal to the Euclidean norm of the input vector.

In more detail, each row in the Haar transform matrix is formed as the concatenation of two adjacent dyadic ranges (i.e., a range whose length is a power of two, beginning at an integer multiple of its length), where the first range is multiplied by +1, and the second by -1, normalized so that the row norm is 1. Hence, the transform is somewhat

similar to the dyadic decomposition discussed above. As a result, it can be combined with sketching: we can apply the Haar transform to each update, then take each result and use them to UPDATE a sketch, such as a Count Sketch. By combining this with methods such as the coding approach outlined above, it is possible to recover the wavelet transform accurately from the sketch. An important feature of the wavelet transform is that it is relatively sparse: each column of the $2^d \times 2^d$ transform matrix has only $d+1$ non-zero entries, and these can be computed directly without storing the full matrix. Therefore, each data update translates into $d + 1$ wavelet coefficient updates.

Discrete Hadamard Transform. The Discrete Hadamard Transform (Hadamard, for short), is the Fourier transform when we consider the data to be indexed as a d-dimensional hypercube. Compared to other Fourier transforms, it is perhaps simpler to express: the $2^d \times 2^d$ Hadamard transform is given by $H_{i,j} = 2^{-d/2}(-1)^{\langle i,j \rangle}$, where $\langle i, j \rangle$ is the modulo-2 inner product of the binary representations of i and j. A sketch of the Hadamard transform of a dataset can be computed as for the DHT: each update is transformed to generate a set of updates to the Hadamard transform, which are then used to UPDATE the sketch. However, the Hadamard transform is very dense, so every update to the data produces 2^d updates to the transform. It may then be advantageous to buffer up some updates before computing their transformation, which can be added on to the sketch, using the properties of linearity: $SH(x + y) = SHx + SHy$. We have mentioned one application of Hadamard transform already, in our discussion of the Sparse JLT (Section 5.5).

History and Background. Computing the wavelet transform of a stream of data was one of the first problems actively studied in that area. Gilbert *et al.* [112] suggested first building a sketch of the original data, then estimating wavelet coefficients by building a sketch of each wavelet basis vector, and estimating the inner product. The idea to instead transform the input data into the wavelet domain as it arrives was suggested by, among others, Cormode, Garofalakis and Sacharidis [63]. Computing summaries to find the (sparse) Fourier transform of data has also had a long history, going back to the start of the century [111]. As noted, the Hadamard transform specifically plays an important role in instantiations of the Johnson-Lindenstrauss transform [8]. More recently, there has been a line of work to build summaries of the Fourier transform so that the k biggest Fourier coefficients of data of size n can be found

faster than the (already quite fast) $O(n \log n)$-time Fast Fourier Transform [125].

9.4 Manipulating Summaries

Given different ways to summarize data as building blocks, there are many possible ways to extend their applications by manipulating summary structures, such as combining or nesting them in ways beyond the basic UPDATE and MERGE procedures.

9.4.1 Algebra on Summaries

Whenever we apply a MERGE operation on a pair of summaries, we are relying on an algebraic property. This may be that $\text{MERGE}(X, \text{MERGE}(Y, Z)) = \text{MERGE}(\text{MERGE}(X, Y), Z)$, i.e., the MERGE operation is associative; or a weaker statement that $\text{MERGE}(X, \text{MERGE}(Y, Z)) \approx \text{MERGE}(\text{MERGE}(X, Y), Z)$ – that is, that the result of changing the merge order may not provide identical results, but that the results are equivalent in terms of the approximation guarantees that they promise (encoded by the $\approx$ relation).

For some summaries, we have stronger algebraic properties. In particular, when the summary is a linear transform, as discussed in Section 9.3.4. Recall that we say a summary is a linear transform when it can be written as Av, when the input is described by a vector v. It then immediately follows from properties of linear algebra that $A(\alpha v + \beta w) = \alpha Av + \beta Aw$. That is, we can apply linear scaling to the inputs v and w by directly scaling the result of summarizing. This gives flexibility in manipulating summaries. In particular, it means that $A(v-w) = Av - Aw$: we can summarize inputs v and w separately, then subtract the summaries to obtain a summary of this difference. This allows, for example, using a Count Sketch to estimate the difference in frequencies between two observations of a distribution (say, the difference between the distribution yesterday and today). The error is proportional to $\|v - w\|_2$, which can be much smaller than the alternative approach of separately estimating the frequency of the two instances, which incurs error proportional to $\|v\|_2 + \|w\|_2$.

9.4.2 Resizing a Summary

For traditional data structures that are initialized to a fixed size, it is common to allow the structure to be "resized" to accommodate more or fewer items, even if this requires emptying the structure of stored items, and filling a new instance from scratch. While many summaries we have seen are also initialized based on a fixed parameter, in order to provide a particular approximation guarantee ϵ, say, it is rarely straightforward to resize a summary.

In several cases, it is possible to reduce the size of a summary, say by halving the size parameter. For example, it is possible to resize a BloomFilter summary down, e.g., to go from m to $m/2$ bits (when m is even), by treating the left and right halves of the summary as the subjects of a MERGE operation, and ignoring the most significant bit of the hash values going forward. Similar results hold for sketches like Count-Min Sketch, Count Sketch, AMS Sketch and Sparse JLT.

However, the inverse operation (doubling the size of the summary) does not lend itself to such tricks. In general, we would not expect summaries to be increased in size without some penalty in space or accuracy. To provide the approximation guarantees associated with the larger size would entail retrieving information which was previously "forgotten" in order to ensure a smaller space bound. The best one might hope for is that far in advance of the current structure filling up with information, we would start a new instance of the summary in parallel to summarize the subsequent updates. Then the total magnitude of those initial updates which were ignored would eventually be sufficiently small that they would not impact the approximation guarantee.

9.4.3 Nesting Summaries

A natural approach to building new summaries is to "nest" summaries inside one another. That is, we use one summary type as a sub-structure within another one. We have already seen some examples that meet this description. For example, the ℓ_0-sampler is built by nesting SparseRecovery summaries inside a sampling structure. We next describe some more examples where one summary type is "nested" inside another.

Summaries using Probabilistic Counters Many summaries (particularly sketch summaries) are based on collections of counters that count

the number of items mapped to those counters by hash functions. In Section 1.2.2, we saw that a counter could be considered as a first example of a summary. It is natural to replace an exact counter with a MorrisCounter probabilistic counter. The result is to reduce the space required for the summary when dealing with inputs truly huge in volume, since the bit size of the counters is reduced to the logarithm of this amount. With careful argument, it can be shown that the resulting estimates remain approximately correct.

Summaries using Distinct Counters Consider the problem where our input is described by a sequence of pairs (x, y), and we want to find those x values that are associated with a large number of distinct y values. For example, the pairs (x, y) could be edges in a graph, and we want to find those nodes that have a high number of (distinct) neighbors. If there were only a few x values, then we would keep a distinct counter for each x (a KMV or HLL structure). If there were no duplicate (x, y) pairs, we would use a frequent items structure such as SpaceSaving or Count-Min Sketch summary. It therefore makes sense that to solve the general form of the problem when there are many x values and many repetitions that we can combine a frequent items structure with distinct counters. A simple instantiation comes from taking a Count-Min Sketch and replacing each counter with an HLL summary. Each (x, y) update uses the Count-Min Sketch hash functions to map x to set of cells. Each cell contains an HLL, which is updated with the value of y. The analysis of the new nested summary is similar to that for the basic Count-Min Sketch: to estimate the number of distinct y's associated with a given x, we inspect the cells where that x is mapped. Each one counts (approximately) the number of distinct y's for that x along with other colliding items. Taking the smallest of these estimates minimizes the amount of noise from colliding items.

Summaries with Quantile Summaries Last, we can use quantile summaries such as GK or Q-Digest as the nested summary. For example, consider replacing the counters in a Count-Min Sketch with a GK summary. We can now process data represented by pairs (x, y), where x is used to map into the Count-Min Sketch, and then y is used to update the corresponding GK summary in the mapped cell. This allows us to estimate the distribution of y values associated with a given x, by probing all the cells associated with x and interrogating the GK summary with the lowest total weight. For x's that are relatively rare, this will

have a higher error, but fairly accurate answers can be provided for x's with high frequency.

History and Background. The idea of building sketches on top of probabilistic counters is explored by Gronemeier and Sauerhoff [121]. Considine *et al.* [57] describe combinations of the Count-Min Sketch and Q-Digest with the Flajolet-Martin sketch, a precursor to HLL. Motivation for this problem in the networking domain is given by Venkataraman *et al.* [221], which defines the notion of "superspreaders". These correspond to network addresses that communicate with a large number of distinct other addresses. Related problems, such as combining frequency moments with distinct counts, are covered by Cormode and Muthukrishnan [73]. Problems which require applying different stages of aggregation have been studied variously under the labels of "correlated aggregates" [217] and "cascaded norms" [139].