Whispers in the Data

Exploring Advice-Driven Sketching Algorithms

an undergraduate thesis presented
by
Anthony Cui
to
The Department of Computer Science

in partial fulfillment of the requirements
for the joint degree of
Bachelor of Arts
in the subjects of
Computer Science and Statistics
with Honors

Harvard University
Cambridge, Massachusetts
March 28, 2025

Thesis advisors: Prof. Michael Mitzenmacher
& Prof. Natesh Pillai Anthony Cui

Whispers in the Data

Abstract

In modern systems, sketching algorithms are a common tool to analyze vast amounts of data. Recent work has explored ways to apply an advice model on top of these algorithms, where machine learning models provide hints when processing elements. This enables these sketching algorithms to provide even better performance as they are guided by the models.

This thesis focuses on these sketching algorithms that use advice. I examine and consider the limitations of existing algorithms, then design a new sketching framework to work around these limitations, which I dub the Bucketing sketch. Using this framework, I focus on three key computational challenges: estimating high-frequency moments for increment-only data streams, handling turnstile streams, and quantile estimation for aggregate values. For high-frequency moments (defined as $f(\mathbf{x}) = \sum x^d$ for $d > 2$), I prove that my approach can achieve precise estimates using logarithmic space, requiring only minimal assumptions about the underlying data-generating process. The remaining two problems are challenging for existing sketching algorithms; nevertheless, I show strong experimental results for all three problems, highlighting the flexibility and power of the Bucketing sketch.

Contents

List of Figures

Acknowledgments

First and foremost, I would like to thank my advisors, Professors Michael Mitzenmacher and Natesh Pillai. This thesis would be a fraction of what it is now without their continued support, critiques, ideas and insights. I am forever grateful for their willingness to comb through my inscrutable notes, feeding me papers, connecting me with colleagues, and providing me with countless pieces of feedback that helped me grow as a researcher. They truly made this thesis a joy to work on.

I also want to thank all my friends that have supported me throughout this process. Whether you have helped me proofread, listened to me talk, or simply provided company during the long hours I have worked, I will always remember the time and energy you have given to me. Special thanks go out to my roommates—Bobby Degeratu, Jonathan Fu, Alex Guh-Siesel, Carter Hayes, and Krishi Kishore—who have been by my side since the very first day of school. Our shared experiences, late-night conversations, and unwavering friendship have been a source of strength and joy.

Finally, I would like to express my warmest gratitude to my family. The path I took to get here was made possible by you; I do not know how to thank you enough.

1 Introduction

In today's era of big data, we often have access to vast amounts of information, ready to be analyzed. However, many of the statistics we want to calculate are expensive to compute. Take a website owner as an example: they might want to track the top 100 clients who send the most traffic, but cannot afford to count the amount of traffic from every single client in order to identify those top 100. The computational power and resources required are simply impractical.

This is where sketching algorithms come into play. They provide accurate estimates of the answer while using significantly less memory—in some cases, their memory usage is only poly-logarithmic in the size of the dataset. To accomplish this, sketching algorithms sequentially scan through the elements of the dataset. Individual elements are not stored in memory; instead, some auxiliary memory is updated and eventually processed to provide the final

estimate. This makes them perfect for streams of data, as the algorithm can simply process every element as it arrives. In practice, sketching algorithms are an integral tool for processing large datasets in a quick and efficient manner, with implementations in popular libraries such as Apache DataSketches [3] and Redis [27].

To achieve even higher accuracy, it is sometimes possible to tune these algorithms for their specific dataset. Real-world datasets often follow a predictable structure, and it is possible to exploit this and guide the algorithm towards better estimates. As an example, imagine that we are running the classic binary search algorithm on a dataset with a known distribution [24]. The traditional algorithm starts searching from the middle of the list. Since we know the distribution, however, we can make an educated guess of the location of our target element. From here, we can look at nearby elements, doubling how far we look every time. When our guess is close to the true location, the runtime of binary search can be cut down significantly.

We should note that the worst-case runtime of this augmented binary search does not change. If the guess of the location is completely wrong, then the runtime is still logarithmic in the length of the array. However, if the distance between the guess and the true location stays below some error threshold, we can bound the runtime as logarithmic *in the size of this error* [24]. This means we have an algorithm that provides robust performance if the dataset is well behaved, but does not perform poorly if our assumptions are wrong.

In general, we can conceptualize the guess used above as some *advice* provided to the algorithm. This advice may be simple to calculate—in the previous example, the location of an element within a distribution can be calculated from its quantile—or it may require more sophisticated analysis. In the latter case, modern statistics and machine learning tools may be used. The usefulness of the advice hinges on the ability to observe and extract patterns in the data, which is exactly where modern ML shines.

The first exploration of advice-augmented sketching algorithms was studied by Hsu et al. [14]. They assumed that one has access to an advice model that can predict whether an element is a heavy hitter (one of the most frequent elements in the data set). With this, Hsu et al. were able to craft an improved sketching algorithm to estimate the frequency of *any* element within the data set. Working on a real-world network traffic dataset, they trained a neural network to provide the advice and achieved more accurate estimates than the classical sketching algorithm without prediction. Since then, many classical sketching algorithms have been extended by applying some advice model on top [1] [8].

This thesis attempts to contribute a new type of sketching algorithm that uses advice: the Bucketing sketch. I motivate this sketch by examining where existing algorithms fall short and considering how advice can be used to fix their limitations. The resulting design turns out to be a flexible and general framework that can be applied to many possible estimators and summaries.

The thesis is structured as follows. Chapter 2 introduces two baseline sketching algorithms, one that uses advice and one that does not. From there Chapter 3 introduces the Bucketing sketch and focuses on estimating the frequency moments of datasets. I prove theoretical bounds for this estimate, and show that its performance can be even better through a range of experiments. Finally, Chapter 4 empirically shows that the sketch can perform well for other types of summaries, focusing on statistics that are difficult for existing sketching algorithms to estimate. Chapter 5 discusses the results and ideas for future work.

2

Baseline Sketching Algorithms

To start, let us formalize the definitions and introduce the baseline algorithms we will be working with.

2.1 Definitions

Definition 2.1.1 (Dataset)**.** *The dataset is modeled as a data stream of elements* (x_t, v_t) *where* $t = 0, 1, 2, \ldots, T$ *[9]. These elements are key-value pairs, where* x_t *comes from the universe of possible keys* $\mathcal{X}$*, and* $v_t \in \mathbb{Z}^+$ *is a positive integer update. I define the weight of each key* w_x *to be the sum of all its corresponding values, and* $\mathbf{w}$ *to be the vector of these weights. Finally, I define* $M = |\mathcal{X}|$ *to be the number of keys, and* $N = ||\mathbf{w}||$ *to be the total weight of the dataset.*

In the literature, this is known as a cash register model for data

streams [9]. This can capture a variety of scenarios; for example, network traffic can be captured by treating each network packet as a key-value pair, where the client is the key and the size of the packet is the value.

One detail to note is that I define the data stream with a finite length T. At first glance, this might seem insufficient; after all, network traffic or online marketplace sales do not have a predefined end. However, the sketching algorithm only needs to care about the elements that have arrived before it is queried for its estimate. As such, the time T simply represents the very last element seen [15]. This means that the sketching algorithms should not assume that they know T when they are initialized. If this is absolutely necessary, however, there exist techniques to work around this restriction that are described later.

From here, I can formally define the advice model that all the following algorithms will use. I also refer to this as an oracle; the terms are used interchangeably.

Definition 2.1.2 (Oracle)**.** *Given an element x, the oracle returns a value $o(x)$ that is an estimate of its frequency $\frac{w_x}{||\mathbf{w}||}$. $o(x)$ is sometimes referred to as the oracle estimate or oracle frequency of x.*

Finally, many of the statistics we care about are formulated as some function of the final weights $\mathbf{w}$. A large number of these functions can be formulated as follows:

Definition 2.1.3 (f-statistic)**.** *Given a function f, we are interested in the statistic*

$$||f(\mathbf{w})|| = \sum_{x \in \mathcal{X}} f(w_x).$$

For example, we might want to estimate the number of unique elements ($f(w_x) = \mathrm{I}(w_x > 0)$) [12], or the number of elements with frequencies above a certain threshold ($f(w_x) = \mathrm{I}(w_x/||\mathbf{w}|| > T)$) [9]. $\mathrm{I}(\dots)$ here represents the indicator function.

For most of this thesis, I use the example of estimating frequency moments ($f(w_x) = w_x^d$), specifically for $d > 2$. This function is known to be "hard" to estimate, due to the existence of worst-case inputs that can thwart any sketching algorithm. If algorithms want to provide good estimates, they must use $\Omega\left(n^{1-2/d}\right)$ space [21], which is notably worse than the ideal of poly-logarithmic space.

Despite the existence of these worst-case inputs, there exist sketching algorithms that can provide accurate estimates on more realistic datasets [8]; the following baseline algorithms are such algorithms. Estimating the frequency moments has many useful applications as well. For example, they may be used to estimate the size of join operations in a database, which can then be used to allocate computational resources efficiently [28]. Other applications include estimating the skewness of a data distribution, or analyzing the structure of large graphs [11].

2.2 Properties of Sketching Algorithms

There are a few additional properties we would like our sketching algorithms to have, beyond the requirement of sub-linear space.

Definition 2.2.1 (Single-pass Sketching Algorithm)**.** *A single-pass sketching algorithm is an algorithm that only requires one pass over the input stream, and cannot influence the order in which the data arrive [15].*

I make this distinction because multi-pass sketching algorithms also exist—these are typically used in databases or other situations where the data stream can be stored and read from some bulk storage [5]. In practice, however, we want to ingest streams and perform queries in real time. This is only possible with single-pass algorithms, so I restrict myself to this case.

Definition 2.2.2 (Mergeability)**.** *A sketching algorithm is mergeable if copies of the algorithm can be ran on disjoint subsets of the data, and then merged to yield an estimate of the overall dataset with the same accuracy [2].*

This property helps improve the speed of sketching algorithms even more by allowing them to be parallelized. It also mirrors well-known distributed programming paradigms like MapReduce [2], so mergeable sketching algorithms are easily applied to existing distributed systems.

Finally, many sketching algorithms are probabilistic in nature; they rely on randomness to function, so their estimates are only guaranteed to be accurate with some high probability. To capture this fact, we formalize our proofs using (ϵ, δ) bounds:

Definition 2.2.3 ((ϵ, δ) bounds)**.** *Let $\hat{X}$ be the sketching algorithm's estimate for the statistic X. Then we want to prove that*

$$P\left(\left|\hat{X} - X\right| \leq \epsilon X\right) \geq 1 - \delta$$

for some $\epsilon, \delta > 0$ [15].

The combination of a given ϵ and δ often dictates how much space an algorithm may need. Alternatively, one may fix the space used, and then calculate the probability that the algorithm achieves an error below ϵ.

2.3 Baseline Algorithms

2.3.1 Probability Proportional to Size Without Replacement (PPSWOR) Sampling

PPSWOR is an ideal sampling algorithm that the following two algorithms take inspiration from. Its general strategy is to first sample some subset of keys, then calculate an estimate of the f-statistic $||f(\mathbf{w})||$ using this sample [7].

Definition 2.3.1 (The PPSWOR Sample)**.** *Let k be the number of keys we want to sample. PPSWOR sampling draws k keys without replacement to form the subset $\mathcal{S} \subseteq \mathcal{X}$. For each draw, the probability that a key is sampled*

is proportional to its weight. For example, for the first draw each key is sampled with probability $\frac{f(w_x)}{||f(\mathbf{w})||}$ *[7].*

Note that because PPSWOR samples keys without replacement, if $M < k$ *then only* M *keys will be drawn.*

We want to use this sample of keys $\mathcal{S}$ to calculate an estimate of $||f(\mathbf{w})||$. To do this, we first define a per-key estimate of $f(w_x)$ [7]:

Definition 2.3.2. *Let* $p_x = P(x \in \mathcal{S})$*. Then the per-key estimate of* $f(w_x)$ *is defined as*

$$\widehat{f(w_x)} = \begin{cases} \frac{f(w_x)}{p_x} & \text{if } x \in \mathcal{S} \\ 0 & o.w. \end{cases}. \tag{2.1}$$

To get an estimate for $||f(\mathbf{w})||$, we simply take the sum of these per-key estimates for each $x \in \mathcal{S}$:

Definition 2.3.3. *Using* $\widehat{f(w_x)}$ *defined in Equation 2.1, we define the estimate of* $||f(\mathbf{w})||$ *as*

$$\widehat{||f(\mathbf{w})||} = \sum_{x \in \mathcal{S}} \widehat{f(w_x)}. \tag{2.2}$$

Theorem 2.3.1. *This estimator for* $\widehat{||f(\mathbf{w})||}$ *is unbiased, and has the variance*

$$\text{Var}\left(\widehat{||f(\mathbf{w})||}\right) \leq \frac{1}{k}||f(\mathbf{w})||^2. \tag{2.3}$$

Proof. We start by noting that the per-key estimates $\widehat{f(w_x)}$ are unbiased via direct calculation:

$$\mathbb{E}\left[\widehat{f(w_x)}\right] = \frac{f(w_x)}{p_x} \cdot P(x \in S) + 0 \cdot P(x \notin S) = \frac{f(w_x)}{p_x} \cdot p_x = f(w_x).$$

By linearity of expectation, this yields that $\widehat{||f(\mathbf{w})||}$ is unbiased as well:

$$\begin{aligned}
\mathbb{E}\left[\widehat{||f(\mathbf{w})||}\right] &= \mathbb{E}\left[\sum_{x\in\mathcal{S}} \widehat{f(w_x)}\right] \\
&= \mathbb{E}\left[\sum_{x\in\mathcal{S}} \widehat{f(w_x)} + \sum_{x\notin\mathcal{S}} 0\right] \\
&= \mathbb{E}\left[\sum_{x\in\mathcal{X}} \widehat{f(w_x)}\right] \\
&= \sum_{x\in\mathcal{X}} \mathbb{E}\left[\widehat{f(w_x)}\right] \\
&= \sum_{x\in\mathcal{X}} f(w_x).
\end{aligned}$$

To get the variance bound in Equation 2.3, we can first derive the variance of the per-key estimators. By definition, we have

$$\begin{aligned}
\mathrm{Var}\left(\widehat{f(w_x)}\right) &= \mathbb{E}\left[\widehat{f(w_x)}^2\right] - \mathbb{E}\left[\widehat{f(w_x)}\right]^2 && (2.4)\\
&= \left(\left(\frac{f(w_x)}{p_x}\right)^2 \cdot p_x\right) - (f(w_x))^2 && (2.5)\\
&= f(w_x)^2\left(\frac{1}{p_x} - 1\right). && (2.6)
\end{aligned}$$

From here, we can first note that p_x is lower bounded by the probability of being chosen if we sampled *with* replacement. This implies that

$$p_x \geq 1 - \left(1 - \frac{f(w_x)}{||f(\mathbf{w})||}\right)^k \implies \frac{1}{p_x} - 1 \leq \frac{1}{k}\frac{||f(\mathbf{w})||}{f(w_x)}.$$

Thus, we have that

$$\mathrm{Var}\left(\widehat{f(w_x)}\right) \leq \frac{1}{k} f(w_x)||f(\mathbf{w})||.$$

Now, note that since we always sample k keys, each per-key estimate has negative covariance with each other; sampling key x means key y has a smaller chance of being sampled, and so $\widehat{f(w_y)}$ has a higher chance of being 0 when $\widehat{f(w_x)}$ is not 0. This means we can write

$$\begin{aligned}
\text{Var}\left(\widehat{||f(\mathbf{w})||}\right) &= \text{Var}\left(\sum_{x \in S} \widehat{f(w_x)}\right) \\
&= \text{Var}\left(\sum_{x \in \mathcal{X}} \widehat{f(w_x)}\right) \\
&\leq \sum_{x \in \mathcal{X}} \text{Var}\left(\widehat{f(w_x)}\right) \\
&\leq \sum_{x \in \mathcal{X}} \frac{1}{k} f(w_x) ||f(\mathbf{w})|| \\
&\leq \frac{1}{k} ||f(\mathbf{w})||^2.
\end{aligned}$$

□

Corollary 2.3.1. The error bound as defined by Definition 2.2.3 shrinks at a rate $\epsilon = O(1/\sqrt{k})$ for some fixed δ. This arises from the fact that the variance shrinks by $O(1/k)$; applying Chebyshev's inequality yields

$$\begin{aligned}
& P\left(\left|\widehat{||f(\mathbf{w})||} - \mathbb{E}\left[||f(\mathbf{w})||\right]\right| \geq c\sqrt{\frac{1}{k}||f(\mathbf{w})||^2}\right) \leq \frac{1}{c^2} \\
\implies & P\left(\left|\widehat{||f(\mathbf{w})||} - \mathbb{E}\left[||f(\mathbf{w})||\right]\right| \geq \frac{1}{\sqrt{\delta}} \cdot \frac{1}{\sqrt{k}}||f(\mathbf{w})||\right) \leq \delta \\
\implies & P\left(\left|\widehat{||f(\mathbf{w})||} - ||f(\mathbf{w})||\right| \geq \frac{1}{\sqrt{\delta k}}||f(\mathbf{w})||\right) \leq \delta \\
\implies & P\left(\left|\widehat{||f(\mathbf{w})||} - ||f(\mathbf{w})||\right| \leq \frac{1}{\sqrt{\delta k}}||f(\mathbf{w})||\right) \geq 1 - \delta.
\end{aligned}$$

Thus, $\epsilon = 1/\sqrt{\delta k} = O(1/\sqrt{k})$.

2.3.2 Bottom-k Sampling

Although we have shown that PPSWOR sampling works well in theory, we have not provided an algorithm to actually implement it. Bottom-k sampling [7] is an algorithm that adopts the ideas behind PPSWOR sampling. It works by leveraging special properties of the exponential distribution.

Theorem 2.3.2. *Define the score function $s(x) \sim \text{Expo}\left(f(w_x)\right)$, which follows an exponential distribution with mean $\frac{1}{f(w_x)}$. The smallest k scores will yield a k-sized PPSWOR sample.*

Proof. Let us prove this by induction. For our base case, consider when $k = 1$. We want to show that the key x is sampled with probability $\frac{f(w_x)}{||f(\mathbf{w})||}$. From our definition, x is sampled if its score is the smallest; thus, we have

$$P\left(x \in \mathcal{S}\right) = P\left(s(x) < \min_{y \neq x} s(y)\right).$$

Using the order statistics of an exponential, we know that $\min_{y \neq x} s(y) \sim \text{Expo}\left(\sum_{y \neq x} f(w_y)\right)$. This means that

$$P\left(s(x) < \min_{y \neq x} s(y)\right) = P\left(\text{Expo}\left(f(w_x)\right) < \text{Expo}\left(\sum_{y \neq x} f(w_y)\right)\right).$$

These two exponential distributions are independent, so we can use the fact that $P\left(X < Y\right) = \frac{\lambda_X}{\lambda_X + \lambda_Y}$ when $X \sim \text{Expo}\left(\lambda_X\right)$ and $Y \sim \text{Expo}\left(\lambda_Y\right)$ are independent:

$$P\left(\text{Expo}\left(f(w_x)\right) < \text{Expo}\left(\sum_{y \neq x} f(w_y)\right)\right) = \frac{f(w_x)}{f(w_x) + \sum_{y \neq x} f(w_y)} = \frac{f(w_x)}{||f(\mathbf{w})||}.$$

This completes our base case.

Now, assume the inductive hypothesis that we can get a PPSWOR sample of size k, and let us consider the case of $k + 1$. If an element x is in the

bottom k scores, then it was already sampled proportionally by the inductive hypothesis. Otherwise, it is in the top $n-k$ elements. In this case, x is sampled as the $(k+1)$th element if it has the smallest score among the remaining $n-k$ elements. However, all of these elements are larger than the kth smallest score; thus, by the memoryless principle of the exponential distribution, we can imagine dropping the k smallest scores without changing the distribution of the remaining scores. This reduces to the $k=1$ case and so x will still be sampled proportionally with respect to the remaining elements. □

Once we have a sample of k elements, the PPSWOR estimate requires us to calculate $p_x = P(x \in \mathcal{S})$. Exactly computing this is not possible, however, as it requires knowing the distribution of every score function, which we cannot do as we are working in a streaming environment. Instead, Bottom-k sampling defines a similar, but not identical, estimator:

Definition 2.3.4 (Bottom-k Sampling Estimator)**.** *Consider the element x' with the $(k+1)$th smallest score. If we let $\tau = s(x')$, then we can note that the elements in our sample are exactly those whose scores are smaller than τ. Thus, let $p'_x = P(s(x) \leq \tau)$, and define the per-key estimates as*

$$\widehat{f(w_x)} = \begin{cases} \frac{f(w_x)}{p'_x} & \text{if } x \in \mathcal{S} \\ 0 & \text{o.w.} \end{cases}.$$

The overall estimate of $||f(\mathbf{w})||$ then is defined as

$$\widehat{||f(\mathbf{w})||} = \sum_{x \in \mathcal{S}} \widehat{f(w_x)}. \tag{2.7}$$

The process to generate and identify the smallest k scores depends on the specific weight function f being used. For an example of this process, consider the frequency moment function $f(w_x) = w_x^d$ for $d > 0$.

Algorithm 1 Bottom-k Sampling for Frequency Moments

procedure INITIALIZE(k)
  $HH \leftarrow$ heavy hitter sketch (e.g., CountSketch [6]) that returns an estimate of the top $k+1$ elements and their weights
  $seed(x) \leftarrow$ random hash function over elements
  $r(seed) \leftarrow$ pseudo-random generator drawing from Expo (1) distribution
end procedure
procedure PROCESSELEMENT(x, v)
  $v' \leftarrow \frac{v}{r(seed(x))^{1/d}}$
  HH.PROCESS(x, v')
end procedure
procedure QUERYESTIMATE
  $S \leftarrow HH$.SAMPLE ▷ *Top $k+1$ weights in increasing order*
  $\_, \tau' \leftarrow S[k]$
  $est \leftarrow 0$
  for $(x, v') \in S[0 \dots k-1]$ **do**
    $v \leftarrow v' \cdot r(seed(x))^{1/d}$
    $p \leftarrow 1 - \exp\left(-(v/\tau')^d\right)$
    $est \leftarrow est + \frac{v^d}{p}$
  end for
  return est
end procedure

Theorem 2.3.3. *Algorithm 1 correctly implements the bottom-k sampling estimator for frequency moments.*

Proof. First, consider the result of the updates to the heavy hitter sketch. By definition, the total weight for a key x from the perspective of the heavy hitter sketch is

$$\sum \frac{v}{r(seed(x))^{1/d}} = \frac{\sum v}{r(seed(x))^{1/d}} = \frac{w_x}{r(seed(x))^{1/d}}.$$

Let us refer to this as $s'(x)$. Now, recall that $s(x) \sim \frac{1}{w_x^d}$ Expo (1). Since $r(seed(x)) \sim$ Expo (1), we can say that $s(x) = \frac{1}{s'(x)^d}$. However, this is simply a monotonic transformation between the score functions. This means that

taking the largest k keys from the heavy hitter is equivalent to taking the smallest k scores.

Once we sample the largest k keys from the heavy hitter, we can see that taking $v = s'(x) \cdot r(seed(x))^{1/d}$ correctly recovers w_x. From here, we want to calculate $p'_x = P(s(x) \leq \tau)$. We have $\tau = \frac{1}{\tau'^d}$; thus, we can calculate

$$p'_x = P(s(x) \leq \tau) = P\left(s(x) \leq \frac{1}{\tau'^d}\right) = 1 - \exp\left(-w_x^d/\tau'^d\right) = 1 - \exp\left(-(v/\tau')^d\right).$$

Aggregating the values $\frac{v^d}{p'_x}$ for the largest k keys from the heavy hitter then yields the estimate given in Equation 2.7. □

Remark 2.3.1. The Bottom-k sampling sketch is mergeable as long as the heavy hitter sketch used is mergeable, and the same random hash function $seed(x)$ is used for each copy of the algorithm. Merging the sketch is done by merging the underlying heavy hitter sketch, and using the new top $k+1$ keys like normal. Since the same random hash function is used across the copies, the mergeability of the heavy hitter sketch ensures this is correct.

We should note that this is not the same as the PPSWOR estimates, since $p'_x \neq p_x$ and the use of a heavy hitter sketch introduces some additional error. However, similar results can still be proven about this algorithm. I omit the proofs from this thesis as they are rather involved, but [7] shows that the estimate from this algorithm is also unbiased and has the variance

$$\text{Var}\left(\widehat{||f(\mathbf{w})||}\right) \leq \frac{1}{k-1}||f(\mathbf{w})||^2.$$

2.3.3 Sampling With Advice

The Sampling with Advice (SWA) algorithm, introduced by Cohen et al. [8], revisits Bottom-k sampling with the assumption that we now have an oracle. The key insight is that Bottom-k sampling is inaccurate because we need to approximate the scores $s(x)$ using a sketching algorithm. However, the oracle

allows us to estimate the frequency of keys $o(x) = \frac{w_x}{||\mathbf{w}||}$. Using this, it may be possible to directly calculate the score function $s(x) \sim \frac{1}{f(w_x)} \operatorname{Expo}(1)$ and bypass the need for a heavy hitter sketch.

Let us consider the frequency moment function again as an example. Consider the score function

$$s'(x) = \frac{1}{o(x)^d} \operatorname{Expo}(1).$$

This is distributed $s'(x) \sim \frac{||\mathbf{w}||^d}{w_x^d} \operatorname{Expo}(1)$. However, we only care about the relative ordering between the scores of keys. Since $||\mathbf{w}||^d$ is constant across all keys, taking the smallest k values of $s'(x)$ is equivalent to taking the smallest k scores.

In general, we can write

$$s'(x) = h(o(x)) \operatorname{Expo}(1)$$

for some function $h(f)$; $h(f) = \frac{1}{f^d}$ in the example above. This method allows us to exactly track the weights of the elements in the final sample. This is because we only need to calculate scores once for each key, making them *stable*. As a result, we can simply keep a list of the elements with the smallest k scores seen. If an element belongs in this list, it will be added the first time it appears in the data stream, and we can track updates to its weight using exact counters instead of a sketching algorithm. These changes lead to Algorithm 2.

We also need to consider the edge case when $o(x) = 0$, as these elements will never be sampled by the bottom-k_p sample. To correct for this, we can add an additional sampling scheme that uniformly samples these excluded elements, without regard to their frequency. This can be implemented using Algorithm 2 and choosing $h(f) = 1$—we still get exact counts since this score function is stable.

Finally, the inclusion of an oracle also enables us to create a top-k sample. This sample can capture all the elements with the highest estimated frequencies $o(x)$. Since these elements are likely to contribute the most to the final estimate, we would like to ensure that they are always included. Their weights can also be exactly counted by following the same procedure as the Bottom-k sketch. This implementation is given in Algorithm 3.

In order to use these samples together, we employ an overflow process. We want the top-k_h sample to take priority, as all the elements with the largest estimated frequencies should land in this sample. Thus, the Bottom-k_p sample should only be from the subset of elements not in the top-k_h sample. Similarly, the uniform sample only samples from elements where $o(x) = 0$, which are not considered by the other two samples. Therefore, an element overflows from the higher-priority samples down to the lower-priority ones. Altogether, this constitutes the Sampling With Advice (SWA) algorithm shown in Algorithm 4.

Theorem 2.3.4. *SWA yields an estimate that is unbiased.*

Additionally, define $\mathcal{X}_h$ as the keys with the k_h largest oracle estimates, $\mathcal{X}_p = \{x | x \notin \mathcal{X}_h, o(x) > 0\}$ as the elements considered for the Bottom-k_p sample with advice, and $\mathcal{X}_u = \mathcal{X} \setminus \left(\mathcal{X}_h \bigcup \mathcal{X}_p\right)$ as the remaining keys that are sampled by the uniform k_u sample. Suppose

$$\frac{f(w_x)}{\sum_{y \in \mathcal{X}_p} f(w_y)} \leq c_p \frac{h(o(x))}{\sum_{y \in \mathcal{X}_p} h(o(y))} \qquad \forall x \in \mathcal{X}_p \tag{2.8}$$

$$\frac{f(w_x)}{\sum_{y \in \mathcal{X}_u} f(w_y)} \leq c_u \frac{1}{|\mathcal{X}_u|} \qquad \forall x \in \mathcal{X}_u \tag{2.9}$$

for some $c_p, c_u \geq 0$; these represent the gap between the sampling probabilities

for PPSWOR vs. the Bottom-k and uniform samples respectively. Then

$$\mathrm{Var}\left(\widehat{||f(\mathbf{w})||}\right) \leq \frac{c_p}{k_p - 1}\left(\sum_{x\in\mathcal{X}_p} f(w_x)\right)^2 + \frac{c_u}{k_u - 1}\left(\sum_{x\in\mathcal{X}_u} f(w_x)\right)^2. \tag{2.10}$$

Proof. To start, we can note that $\mathcal{X}_h$, $\mathcal{X}_p$, and $\mathcal{X}_u$ are disjoint partitions of $\mathcal{X}$, and these partitions do not depend on the data stream; only on $o(x)$. Therefore, each sample is independent of the other, and we can analyze them separately.

For the top-k_h sample, it perfectly tracks all the keys in $\mathcal{X}_h$ every time. Therefore, its estimate is trivially unbiased and has 0 variance.

For the bottom-k_p sample, recall that [7] showed that it is unbiased. To prove the variance bound, we can start from the per-key estimate of a bottom-k_p sample proved by [7]:

$$\mathrm{Var}\left(\widehat{f(w_x)}\right) \leq \frac{1}{k_p - 1} f(w_x)\left(\sum_{y\in\mathcal{X}_p} f(w_y)\right) = \frac{1}{k_p - 1}\frac{1}{q_x} f(w_x)^2$$

where $q_x = \frac{f(w_x)}{\sum_{y\in\mathcal{X}_p} f(w_y)}$ is the PPSWOR sampling probability. The bottom-k_p sample with advice does not achieve these exact sampling probabilities, since the oracle estimates are not exact; instead, the sampling probabilities are $q'_x = \frac{h(o(x))}{\sum_{y\in\mathcal{X}_p} h(o(y))}$. However, [7] also shows that the variance of estimates with a different sampling probability follow the similar form

$$\begin{aligned}
\mathrm{Var}\left(\widehat{f(w_x)}\right) &\leq \frac{1}{k_p - 1}\frac{1}{q'_x} f(w_x)^2 \\
&= \frac{1}{k_p - 1}\frac{1}{q_x}\frac{q_x}{q'_x} f(w_x)^2 \\
&= \frac{1}{k_p - 1} f(w_x)\left(\sum_{y\in\mathcal{X}_p} f(w_y)\right)\frac{q_x}{q'_x}.
\end{aligned}$$

Applying the assumed bound in Equation 2.8, we have that $\frac{q_x}{q'_x} \leq c_p$. Thus,

the total variance of the bottom-k_p estimate is

$$\begin{aligned}\mathrm{Var}\left(\widehat{\sum_{x\in\mathcal{X}_p} f(w_x)}\right) &\leq \sum_{x\in\mathcal{X}_p} \frac{1}{k_p-1} f(w_x)\left(\sum_{y\in\mathcal{X}_p} f(w_y)\right) c_p \\ &= \frac{c_p}{k_p-1}\left(\sum_{x\in\mathcal{X}_p} f(w_x)\right)^2.\end{aligned}$$

A similar argument can be used for the uniform k_u sample, since it is implemented as a bottom-k sketch with sampling probabilities $\frac{1}{|\mathcal{X}_u|}$. Thus, this sample will also yield an estimate that is unbiased and have variance

$$\mathrm{Var}\left(\widehat{\sum_{x\in\mathcal{X}_u} f(w_x)}\right) \leq \frac{c_u}{k_u-1}\left(\sum_{x\in\mathcal{X}_u} f(w_x)\right)^2.$$

As each sample works on disjoint sets of data, adding together their estimates also yields an unbiased estimate for $||f(\mathbf{w})||$. Additionally, there is no covariance between their estimates as they are independent; thus, the variance of the final estimate is

$$\begin{aligned}\mathrm{Var}\left(\widehat{||f(\mathbf{w})||}\right) &= \mathrm{Var}\left(\widehat{\sum_{x\in\mathcal{X}_h} f(w_x)}\right) + \mathrm{Var}\left(\widehat{\sum_{x\in\mathcal{X}_p} f(w_x)}\right) + \mathrm{Var}\left(\widehat{\sum_{x\in\mathcal{X}_u} f(w_x)}\right) \\ &\leq \frac{c_p}{k_p-1}\left(\sum_{x\in\mathcal{X}_p} f(w_x)\right)^2 + \frac{c_u}{k_u-1}\left(\sum_{x\in\mathcal{X}_u} f(w_x)\right)^2.\end{aligned}$$

□

Corollary 2.3.2. The error bound of SWA shrinks at a rate of $\epsilon = O\left(\sqrt{\frac{c_p}{k_p-1} + \frac{c_u}{k_u-1}}\right)$. To prove this, we can further bound the variance of SWA as

$$\mathrm{Var}\left(\widehat{||f(\mathbf{w})||}\right) \leq \left(\frac{c_p}{k_p-1} + \frac{c_u}{k_u-1}\right)||f(\mathbf{w})||^2.$$

We then apply Chebyshev's inequality in a similar fashion to Corollary 2.3.1.

Depending on the data distribution, however, k_h can also have a significant impact on lowering the error by removing the variance from the largest elements.

Remark 2.3.2. SWA is also mergeable, assuming the same oracle and seed functions are used across all copies of the algorithm. This is thanks to the fact that the scores are stable, so the final elements in the merged sample will also be present in samples of any subset of the data. Thus, merging can be done by combining all the samples and taking the new top k_h and bottom k_p and k_u elements. The elements that are pushed out of these samples will overflow into the lower-priority samples, and will also need to be considered when merging.

Algorithm 2 Bottom-k_p Sample With Advice

procedure INITIALIZE($k_p, h(f), o(x)$)
    $S \leftarrow$ empty sample of $k_p + 1$ elements, storing $(x, v, s(x))$, sorted by $s(x)$
    $seed(x) \leftarrow$ random hash function over elements
    $r(seed) \leftarrow$ pseudo-random generator drawing from Expo (1) distribution
end procedure
procedure PROCESSELEMENT(x, v)
    if $x \in S$ **then**
        $S[x].v \leftarrow S[x].v + v$
    else
        $s \leftarrow h(o(x)) \cdot r(seed(x))$
        if $s < S[k_p].s$ **then** ▷*Check if among smallest $k_p + 1$ scores*
            Insert (x, v, s) into S ▷*May remove element with largest score if sample full*
        end if
    end if
end procedure
procedure QUERYESTIMATE
    $\tau \leftarrow S[k_p].v$
    $est \leftarrow 0$
    for $(x, v, s) \in S[0 \ldots k_p - 1]$ **do**
        $p \leftarrow 1 - \exp(-h(o(x)) \cdot \tau)$
        $est \leftarrow est + \frac{v^d}{p}$
    end for
    return est
end procedure

Algorithm 3 Top-k_h Sample

```
1: procedure Initialize(k_h, o(x))
2:     S ← empty sample of k_h elements, storing (x, v, o(x)), sorted by o(x)
3: end procedure
4: procedure ProcessElement(x, v)
5:     if x ∈ S then
6:         S[x].v ← S[x].v + v
7:     else
8:         if o(x) > S[k_h − 1].o then
9:             Insert (x, v, o(x)) into S
10:        end if
11:    end if
12: end procedure
13: procedure QueryEstimate
14:    est ← 0
15:    for (x, v, o) ∈ S do
16:        est ← est + v
17:    end for
18:    return est
19: end procedure
```

Algorithm 4 Sampling With Advice (SWA)

procedure INITIALIZE($k_h, k_p, k_u, h(f), o(x)$)
    $S_h \leftarrow$ TOP-k_h.INITIALIZE(k_h, o)
    $S_p \leftarrow$ BOTTOM-k_p.INITIALIZE(k_p, h, o)
    $S_u \leftarrow$ UNIFORM-k_u.INITIALIZE(k_u)
end procedure
procedure PROCESSELEMENT(x, v)
    S_h.PROCESSELEMENT(x, v)
    if an element was pushed out of S_h **then**
        $(x, v) \leftarrow$ overflowed element ▷*Ensure overflowed elements are processed further*
    end if
    if $x \notin S_h$ **then**
        if $o(x) > 0$ **then**
            S_p.PROCESSELEMENT(x, v)
        else
            S_u.PROCESSELEMENT(x, v)
        end if
    end if
end procedure
procedure QUERYESTIMATE
    $est \leftarrow S_h$.QUERYESTIMATE
    $est \leftarrow est + S_p$.QUERYESTIMATE
    $est \leftarrow est + S_u$.QUERYESTIMATE
    return est
end procedure

3

The Bucketing Sketch

3.1 Definition

One detriment of PPSWOR-based algorithms is that they sample a fixed number of elements. Although the estimates they provide are unbiased, the fact that our sample has a fixed size means that there is a certain amount of variance inherent in our estimate, which can only be reduced by increasing the size of the sample. The bucketing sketch attempts to break past this variance using two observations:

- The PPSWOR estimator relies on estimating the weights of individual elements. We cannot hope to do this exactly for too many elements due to our space constraints. However, if many elements have similar weights, a single weight estimate can apply for all of them.

- Advice models help us get **stable advice** that we can use to group elements with similar weights. This means that we don't need to worry about moving elements from one group to another, and so we can maintain exact counts for each group.

At a high level, the Bucketing sketch places elements with similar estimated frequencies in the same bucket. Although we do not have exact counts for any individual element, since all elements in a bucket have similar behavior, we can achieve a good approximation for the weight of *every* element in the bucket. As such, the bucketing sketch is not limited by only sampling a fixed number of elements.

To begin the definition of the Bucketing sketch, I define the preliminary parameters for the sketch:

Definition 3.1.1. *Define B different buckets corresponding to disjoint intervals of $(0, 1]$. Bucket B_i covers the range $(l_i, r_i]$ with $l_1 = 0$, $r_b = l_{b+1}$, and $r_B = 1$. Element $x \in B_i$ if $o(x) \in (l_i, r_i]$.*

Furthermore, define K summary functions $g_i^{(k)}(f)$ for each bucket i. Each bucket keeps track of K summaries, which are updated by $g_i(o(x)) \cdot v$ when the element (x, v) is seen.

Let $S_i^{(k)}$ be the kth summary of bucket i, and define $\mathbf{S}$ to be the vector of all these summaries. Once the entire data stream is processed, we have

$$S_i^{(k)} = \sum_{x \in B_i} g_i^{(k)}(o(x)) \cdot w_x.$$

The final estimate of $||f(\mathbf{w})||$ is calculated as a function $h(\mathbf{S})$ of these summaries.

The definition of this sketch is given in Algorithm 5.

I also consider a variation where we add a top-k_h sample, similar to the SWA sketch. This sample captures the elements with the k_h largest oracle frequencies, and any elements that do not land in this sample are then

Algorithm 5 Bucketing Sketch

```
1: procedure Initialize(B[], g[], h, o) ▷B[]: list of buckets, g[]: summary functions
2:     S[B][K] ← array of size B × K for bucket summaries
3: end procedure
4: procedure ProcessElement(x, v)
5:     Find bucket b such that o(x) ∈ (B[b].l, B[b].r]
6:     for k ← 0, ..., K − 1 do
7:         S[b][k] ← S[b][k] + g[k](o(x)) · v
8:     end for
9: end procedure
10: procedure QueryEstimate
11:     return h(S)
12: end procedure
```

considered for the Bucketing sketch. This achieves the exact same benefits where, depending on the data distribution, the error bounds of the estimator can drastically improve as the most influential elements are now being perfectly counted.

Remark 3.1.1. The Bucketing sketch, both with and without the top-k_h sample, is mergeable. Merging bucket summaries can be done by simply taking their sum, since elements contribute to the same summaries across copies. The top-k_h sample can also be merged by taking the top k_h elements across all the samples, and overflowing the remaining elements into the buckets.

The general framework of the Bucketing sketch offers a lot of flexibility in terms of implementation details: bucket ranges, summaries, and the final estimator h are all up to choice. In the remainder of this thesis, I provide an exploration of various implementation choices, some of which come from first principles. However, it remains an open question as to how far these choices are from optimal. It would be interesting for future work to try and prove lower bounds on what can be achieved with this framework.

3.2 Estimating Frequency Moments

My primary evaluation of the Bucketing sketch will be on estimating high-frequency moments: $f(w_x) = w_x^d$ for $d > 2$. I begin by deriving a very simple estimator and proving theoretical results for it. I then continue by considering some alternative approaches to estimation. These estimators are more complicated, so I do not offer any theoretical guarantees for them; empirically, however, this simple estimator seems to have comparable performance.

3.2.1 Defining the Estimator

As I have previously stated, the bucketing sketch relies on the fact that elements in the same bucket have very similar behavior. Every element in the bucket B_i has a predicted frequency $o(x)$ in the range $(l_i, r_i]$, which allows us to form a close approximation for the expected weight of each element.

To leverage this fact to create an estimator, we can note that each bucket gathers summaries of the form

$$S_i^{(k)} = \sum_{x \in B_i} g_i^{(k)}(o(x)) \cdot w_x$$

and we want to estimate

$$\sum_{x \in B_i} w_x^d.$$

This suggests that we want $g_i^{(k)}(o(x))$ to approximate w_x^{d-1}. Since we know that $o(x) \in (l_i, r_i]$, we can use the center of this range to approximate w_x: $w_x \approx ||\mathbf{w}|| \cdot \frac{l_i + r_i}{2}$. From here, a simple and natural estimator is to approximate w_x^{d-1} by $\left(||\mathbf{w}|| \cdot \frac{l_i + r_i}{2}\right)^{d-1}$.

Estimator 1 (Central Estimator)**.** *For each bucket, we keep track of two*

summary functions

$$g_i^{(1)}(f) = 1$$
$$g_i^{(2)}(f) = \left(\frac{l_i + r_i}{2}\right)^{d-1}.$$

We can note that $\sum_i S_i^{(1)} = ||\mathbf{w}||$. *Using this fact, the final estimator is*

$$\begin{aligned}\widehat{||f(\mathbf{w})||}_c &= \sum_{i=1}^{B} \sum_{x \in B_i} \left(||\mathbf{w}|| \cdot \frac{l_i + r_i}{2}\right)^{d-1} w_x \\ &= ||\mathbf{w}|| \cdot \sum_{i=1}^{B} \sum_{x \in B_i} \left(\frac{l_i + r_i}{2}\right)^{d-1} w_x \\ &= \left(\sum_{i=1}^{B} S_i^{(1)}\right)\left(\sum_{i=1}^{B} S_i^{(2)}\right).\end{aligned}$$

I also pair this estimator with a specific bucketing scheme. Intuitively, we want to construct our buckets so that the central approximation is close for all elements. One way to achieve this is to have the width of buckets shrink at some exponential rate, so $\frac{r_i}{l_i} = \gamma$ for some constant $\gamma > 1$. This causes the centers of buckets to always be a constant factor away from their endpoints; I will show in later sections that this scheme turns out to be both analytically convenient and performant.

Definition 3.2.1 (Exponential Bucketing Scheme)**.** *Let* $\gamma > 1$ *be the width of each bucket. Also let* $N = ||\mathbf{w}||$, $\Delta > 0$ *be the expected error of the oracle such that*

$$(1 - \Delta)\mathbb{E}\left[\frac{w_x}{||\mathbf{w}||}\right] \leq o(x) \leq (1 + \Delta)\mathbb{E}\left[\frac{w_x}{||\mathbf{w}||}\right] \quad \forall x \in \mathcal{X},$$

and $\delta > 0$ *be the desired probability of success for this sketch as used in an* (ϵ, δ) *bound. Then the exponential bucketing scheme is defined as follows:*

- *Set the first bucket to be* $(0, f_{\min}]$*, where*

$$f_{\min} = (1 - \Delta)\left(1 - (1 - \delta)^{1/N}\right). \tag{3.1}$$

- *Set* $r_i = \gamma l_i$ *and* $l_i = r_{i-1}$ *for* $i \geq 2$*.*
- *Continue until* $r_i \geq 1$*.*

Theorem 3.2.1. *The exponential bucketing scheme results in* $B = O(\log N)$ *buckets.*

Proof. We know that each bucket will increase exponentially by γ, starting at $f_{\min}$. Thus the total number of buckets B is given by $\gamma^B f_{\min} = 1$, or

$$B = \frac{\log(1/f_{\min})}{\log(\gamma)} = \frac{-1}{\log(\gamma)}\left(\log(1 - \Delta) + \log(1 - (1 - \delta)^{1/N})\right).$$

Finally, we can approximate $\log(1 - (1 - \delta)^{1/N})$ by using the fact that $e^x \geq 1 + x$:

$$\begin{aligned}
\log(1 - (1 - \delta)^{1/N}) &= \log\left(1 - \exp\left(\frac{\log(1 - \delta)}{N}\right)\right) \\
&\leq \log\left(1 - \left(1 + \frac{\log(1 - \delta)}{N}\right)\right) \\
&= \log\left(-\frac{\log(1 - \delta)}{N}\right) \\
&= \log\left(\log\left(\frac{1}{1 - \delta}\right)\right) - \log(N).
\end{aligned}$$

Thus, we can see that $B = O(\log N)$, as desired. □

This formula for $f_{\min}$ depends on $N = ||\mathbf{w}||$, which is a value our algorithm will not know ahead of time. However, we can still implement this scheme by using an estimate N_{est} for N. If the data stream grows beyond N_{est}, we can simply increase our estimate and replace the $(0, f_{\min}]$ bucket with more buckets. To ensure that the $(0, f_{\min}]$ bucket is empty when we do this, we

can keep exact counts of any elements that land in this bucket. Once the number of elements that land in $(0, f_{\min}]$ exceeds some threshold, this can also trigger more buckets to be added and these elements to be placed within these new buckets.

This method works because adding new buckets does not impact any of the existing buckets; thus, the only elements that are impacted are those inside the $(0, f_{\min}]$ bucket that is being replaced. Since we track these elements exactly, we ensure that they are placed in their correct buckets.

We also need to consider the edge case of elements x that have an estimated frequency of 0: $o(x) = 0$. Since these elements will always be placed in $(0, f_{\min}]$, it may not be possible to exactly count all elements of this type. One solution is to adopt a uniform sample just like SWA. Another solution is to replace their estimated frequency with some small ϵ_f; for example, $\epsilon_f = 10^{-9}$. I chose to adopt the second solution to ensure the Bucketing scheme still tracks counts for every element in the data stream.

3.2.2 Theoretical Bounds

In order to provide some theoretical guarantees about our estimator, we need to provide a model of how we expect our dataset to look like. I chose to model the data as follows. First, I take the size of the dataset $N = ||\mathbf{w}||$ and the number of distinct keys $M = |\mathcal{X}|$ to be fixed. This dataset may be split up over many key-value pairs (x, v). However, the Bucketing sketch is invariant over how keys are ordered or how the values are distributed across entries. Thus, we only care about the final aggregate weights w_x.

To model these weights, we can assume that every key x has some true, unchanging frequency f_x. Additionally, let us assume that these keys are independent; the appearance of one key should not influence that of another. This is natural to assume in most common applications; for example, in networking applications traffic from one client should not influence traffic from another client. This means that we can capture the weights as following

a Multinomial distribution:

$$W_1, \ldots, W_M \sim \text{Multinomial}\left(N, (f_1, \ldots, f_M)\right).$$

Under this model, we can prove the following performance claim:

Theorem 3.2.2. *Let us assume the oracle satisfies* $(1-\Delta)f_x \leq o(x) \leq (1+\Delta)f_x \ \forall x \in \mathcal{X}$. *Given a* Δ, γ, *and* δ, *the central estimator (Estimator 1) using the exponential bucketing scheme with parameter* γ *(Definition 3.2.1) satisfies the* (ϵ, δ) *bound with*

$$\epsilon = \max\left\{1 - \left(\frac{(1+\gamma)(1-\Delta)}{2\gamma}\right)^{d-1}, \left(\frac{(1+\gamma)(1+\Delta)}{2}\right)^{d-1} - 1\right\} + O(1/\sqrt{N}). \tag{3.2}$$

To prove this, let us first establish a few lemmas.

Lemma 3.2.1. *Recall from Equation 3.1 that*

$$f_{\min} = (1-\Delta)\left(1 - (1-\delta)^{1/N}\right).$$

With probability $1-\delta$, *any element that belongs in the bucket* $(0, f_{\min}]$ *will not appear in the data stream with probability* $1-\delta$.

Proof. We know that if an element x lands in the bucket $(0, f_{\min}]$, then its frequency f_x is bounded by

$$f_x \leq \frac{1}{1-\Delta} f_{\min} \leq 1 - (1-\delta)^{1/N}.$$

Each element is marginally distributed according to a Binomial distribution. This means we have $W_x \sim \text{Bin}(N, f)$, and so the probability x does not appear in the data stream is exactly

$$P(W_x = 0) = (1-f_x)^N \leq 1 - \delta,$$

as desired. □

Lemma 3.2.2. *Given $X \sim \text{Bin}(N, p)$ and some $\delta < 1$, we have that*

$$P(X \geq (1 + c_1)\mu) \leq \delta \tag{3.3}$$

$$P(X \leq (1 - c_2)\mu) \leq \delta \tag{3.4}$$

with

$$c_1 = \frac{\log(1/\delta) + \sqrt{\log^2(1/\delta) + 8\mu \log(1/\delta)}}{2\mu} \tag{3.5}$$

$$c_2 = \sqrt{\frac{2\log(1/\delta)}{\mu}}. \tag{3.6}$$

Proof. These bounds are derived by directly applying Chernoff bounds for the Binomial distribution. These bounds are

$$P(X \geq (1 + c_1)\mu) \leq e^{-c_1^2\mu/(2+c_1)} \tag{3.7}$$

$$P(X \leq (1 - c_2)\mu) \leq e^{-c_2^2\mu/2}. \tag{3.8}$$

Starting with Equation 3.7, we can equate the right-hand side to δ and solve for c_1 to get

$$\begin{aligned} e^{-c_1^2\mu/(2+c_1)} &= \delta \\ \frac{c_1^2\mu}{2 + c_1} &= \log(1/\delta) \\ c_1^2\mu - (2 + c_1)\log(1/\delta) &= 0 \\ c_1 &= \frac{\log(1/\delta) + \sqrt{\log^2(1/\delta) + 8\mu \log(1/\delta)}}{2\mu} \end{aligned}$$

where we take the larger root since the other root will always be negative.

Repeating for Equation 3.8 yields

$$e^{-c_2^2\mu/2} = \delta$$
$$\frac{c_2^2\mu}{2} = \log(1/\delta)$$
$$c_2 = \sqrt{\frac{2\log(1/\delta)}{\mu}}.$$

□

Now, let us prove the original theorem:

Proof. In order to bound the probability when $\left|||\mathbf{w}^d|| - \widehat{||f(\mathbf{w})||}\right| \leq \epsilon||\mathbf{w}^d||$, we can focus on individual W_xs. For this error bound to not hold, it must be that there exists some W_x where either $\widehat{W_x^d} < (1-\epsilon)W_x^d$ or $\widehat{W_x^d} > (1+\epsilon)W_x^d$. Thus, if we can bound the probability of these two events by $\frac{\delta}{2M}$, then by a union bound we can bound the overall probability of the error bound holding. Note that this provides a conservative lower bound on ϵ; since we model our W_xs as a Multinomial distribution, they will have some negative correlation. This means if one W_x is too large and makes $\widehat{W_x^d}$ deviate slightly more than ϵ away from the true value, it is likely for another W_y to be small, counterbalancing the first error. This correlation between individual errors means that the overall error bound on $\widehat{||f(\mathbf{w})||}$ has a higher probability to hold; nevertheless, this is a simpler bound that is still analytically tractable.

Now, recall that the central estimator uses the center of a bucket's range as the approximation for an element. Let l_{W_x} and r_{W_x} be the left and right ranges of the bucket containing W_x. Then we can express

$$\widehat{W_x^d} = W_x \cdot \left(N \cdot \frac{l_{W_x} + r_{W_x}}{2}\right)^{d-1}.$$

This means we want to bound the following probabilities:

$$P\left(\widehat{W_x^d} < (1-\epsilon)W_x^d\right) = P\left(W_x \cdot \left(N \cdot \frac{l_{W_x} + r_{W_x}}{2}\right)^{d-1} < (1-\epsilon)W_x^d\right) \tag{3.9}$$

$$= P\left(N \cdot \frac{l_{W_x} + r_{W_x}}{2(1-\epsilon)^{\frac{1}{d-1}}} < W_x\right) \tag{3.10}$$

and

$$P\left(\widehat{W_x^d} > (1+\epsilon)W_x^d\right) = P\left(W_x \cdot \left(N \cdot \frac{l_{W_x} + r_{W_x}}{2}\right)^{d-1} > (1+\epsilon)W_x^d\right) \tag{3.11}$$

$$= P\left(N \cdot \frac{l_{W_x} + r_{W_x}}{2(1+\epsilon)^{\frac{1}{d-1}}} > W_x\right). \tag{3.12}$$

First, we can apply Lemma 3.2.2 to Equation 3.10. Using $\mu = Nf_i$, we get

$$\frac{l_{W_x} + r_{W_x}}{2f_i(1-\epsilon)^{\frac{1}{d-1}}} - 1 = \frac{\log(2M/\delta) + \sqrt{\log^2(2M/\delta) + 8Nf_i\log(2M/\delta)}}{2Nf_i}$$

$$(1-\epsilon)^{\frac{1}{d-1}} = \frac{N(l_{W_x} + r_{W_x})}{2Nf_i + \log(2M/\delta) + \sqrt{\log^2(2M/\delta) + 8Nf_i\log(2M/\delta)}}$$

$$\epsilon = 1 - \left(\frac{N(l_{W_x} + r_{W_x})}{2Nf_i + \log(2M/\delta) + \sqrt{\log^2(2M/\delta) + 8Nf_i\log(2M/\delta)}}\right)^{d-1}.$$

Lemma 3.2.1 gives us that with high probability, all the elements will land in buckets where $\frac{r_{W_x}}{l_{W_x}} = \gamma$. Additionally, we know that the oracle predicts $o(x) \in (l_{W_x}, r_{W_x}]$. Using the bounds we assumed for the oracle, this means

that $f_i \in (\frac{l_{W_x}}{1+\Delta}, \frac{r_{W_x}}{1-\Delta}]$. Thus, using $f_i \leq \frac{r_{W_x}}{1-\Delta}$ gives that

$$\begin{aligned}\epsilon &\leq 1 - \left(\frac{N l_{W_x}(1+\gamma)}{2N\frac{l_{W_x}\gamma}{1-\Delta} + \log(2M/\delta) + \sqrt{\log^2(2M/\delta) + 8N\frac{l_{W_x}\gamma}{1-\Delta}\log(2M/\delta)}} \right)^{d-1} \\ &\leq 1 - \left(\frac{2\gamma}{(1+\gamma)(1-\Delta)} + O\left(\frac{1}{\sqrt{N}}\right) \right)^{1-d}.\end{aligned}$$

Now, we can note that $O\left(\frac{1}{\sqrt{N}}\right)$ is small, so we can apply the Binomial approximation of $(1+x)^d \approx 1 + xd$ to get

$$\begin{aligned}\epsilon &\leq 1 - \left(\frac{(1+\gamma)(1-\Delta)}{2\gamma} \right)^{d-1} \left(1 + \left(\frac{(1+\gamma)(1-\Delta)}{2\gamma} \right) O\left(\frac{1}{\sqrt{N}}\right) \right)^{1-d} \\ &\approx 1 - \left(\frac{(1+\gamma)(1-\Delta)}{2\gamma} \right)^{d-1} \left(1 + (1-d) O\left(\frac{1}{\sqrt{N}}\right) \right) \\ &\approx \left[1 - \left(\frac{(1+\gamma)(1-\Delta)}{2\gamma} \right)^{d-1} \right] + O\left(\frac{1}{\sqrt{N}}\right).\end{aligned}$$

If we repeat this for Equation 3.12, we can use $f_i \geq \frac{l_{W_x}}{1+\Delta}$ to get

$$\begin{aligned}1 - \frac{l_{W_x} + r_{W_x}}{2f_i(1+\epsilon)^{\frac{1}{d-1}}} &= \sqrt{\frac{2\log(2M/\delta)}{N f_i}} \\ (1+\epsilon)^{\frac{1}{d-1}} = \frac{l_{W_x} + r_{W_x}}{2f_i\left(1 - \sqrt{\frac{2\log(2M/\delta)}{N f_i}}\right)} &= \frac{l_{W_x} + r_{W_x}}{2f_i - \sqrt{8 f_i \log(2M/\delta)/N}}\end{aligned}$$

$$
\begin{aligned}
\epsilon &= \left(\frac{l_{W_x} + r_{W_x}}{2f_i - \sqrt{8f_i \log(2M/\delta)/N}} \right)^{d-1} - 1 \\
&\leq \left(\frac{l_{W_x}(1+\gamma)}{2\frac{l_{W_x}}{1+\Delta} - \sqrt{8\frac{l_{W_x}}{1+\Delta} \log(2M/\delta)/N}} \right)^{d-1} - 1 \\
&\leq \left(\frac{2}{(1+\gamma)(1+\Delta)} - O\left(\frac{1}{\sqrt{N}}\right) \right)^{1-d} - 1 \\
&\approx \left[\left(\frac{(1+\gamma)(1+\Delta}{2} \right)^{d-1} - 1 \right] + O\left(\frac{1}{\sqrt{N}}\right).
\end{aligned}
$$

As we want both bounds to hold, we take the maximum of both to arrive at Equation 3.2, as desired. □

We see that, in the worst case, there is some irreducible error that does not decrease by sampling more elements. In particular, this error is restricted by the performance of our oracle. No matter how small we shrink our buckets by shrinking γ, the best our bound can become is $\max\{1 - (1-\Delta)^{d-1}, (1+\Delta)^{d-1} - 1\}$. This is different from SWA, where oracle errors can be mitigated by increasing the sample size.

However, this term stems from assuming that every element is maximally far from the center approximation. In practice, we would expect elements to be more spread throughout a bucket's range, and not be clustered around the extreme limits. As such, this first term should be smaller. My experiments in section 3.3 validate this intuition, showing that this upper bound is not tight.

3.2.3 Addressing Existing Lower Bounds

At first glance, these theoretical bounds claiming good performance with logarithmic space may seem contradictory to the "hard" nature of estimating frequency moments. Li et al. [20] showed that any sketch that tracks a linear combination of element weights cannot break through the $\Omega(n^{1-2/d})$ space bound needed to estimate frequency moments. The Bucketing sketch

certainly falls into this category of linear sketches; how, then, can we claim good performance using logarithmic space?

The key difference in my analysis is that I assume a multinomial model where each element has some fixed frequency. In contrast, the worst-case input given by Li et al. consists of a distribution where elements can have large jumps in value, from 1 up to $(2n)^{1/d}$ [20]. The nature of this distribution means that the amount of information needed to track these jumps is $\Omega(n^{1-2/d})$. This kind of input is not possible under my more restrictive multinomial model.

Given the context where the Bucketing sketch is most applicable, I believe that this more restrictive model is acceptable. As discussed previously, sketching algorithms are most useful in large-scale systems. In these settings, it is natural to assume that elements have relatively stable probabilities. Additionally, the immense scale of data sources makes it likely for elements to grow smoothly and not exhibit the sudden jumps seen in Li et al.'s distribution. Although there are certainly real-world settings where an adversary can control the entire data stream to create arbitrary distributions, I am focusing on settings where we can assume normal behavior. The strength of the Bucketing sketch is that it can provide better estimators specifically under these normal conditions.

3.2.4 Alternative Estimators

In this section, I will offer some alternative approaches to derive estimators.

To start, another natural estimator one might want to derive is an unbiased estimator. Assuming our oracle is perfect (so $o(x) = f_x$), the following estimator is unbiased.

Estimator 2 (Unbiased Estimator)**.** *Define*

$$g(f) = \sum_{j=1}^{d} \left\{ {d \atop j} \right\} \frac{(N-1)!}{(N-j)!} f^{j-1}$$

where $\left\{ {d \atop j} \right\}$ *are the Stirling numbers of the second kind. Then we can simply use a single bucket, and use the summary as our final estimator:*

$$\widehat{||f(\mathbf{w})||}_u = S_1 = \sum_x g(o(x)) \cdot w_x.$$

Theorem 3.2.3. *Estimator 2 is unbiased, assuming* $o(x) = f_x$.

Proof. First, note that in expectation

$$\mathbb{E}\left[||f(\mathbf{w})||\right] = \mathbb{E}\left[\sum_{i=1}^{M} W_x^d\right] = \sum_{i=1}^{M} \mathbb{E}\left[W_x^d\right].$$

Since each $W_x \sim \text{Bin}\,(N, f_x)$, we have a closed-form expression for $\mathbb{E}\left[W_x^d\right]$ as

$$\mathbb{E}\left[W_x^d\right] = \sum_{j=1}^{d} \left\{ {d \atop j} \right\} \frac{N!}{(N-j)!} f^j.$$

Plugging in the definition for $g(f)$ then yields the final result:

$$\begin{aligned}
\mathbb{E}\left[S_1\right] &= \mathbb{E}\left[\sum_{i=1}^{M} g(o(x)) \cdot W_x\right] \\
&= \sum_{i=1}^{M} \mathbb{E}\left[\sum_{j=1}^{d} \left\{ {d \atop j} \right\} \frac{(N-1)!}{(N-j)!} o(W_x)^{j-1} \cdot W_x\right] \\
&= \sum_{i=1}^{M} \sum_{j=1}^{d} \left\{ {d \atop j} \right\} \frac{(N-1)!}{(N-j)!} f_i^{j-1} \cdot \mathbb{E}\left[W_x\right] \\
&= \sum_{i=1}^{M} \sum_{j=1}^{d} \left\{ {d \atop j} \right\} \frac{(N-1)!}{(N-j)!} f_i^{j-1} \cdot N f_i \\
&= \mathbb{E}\left[||f(\mathbf{w})||\right].
\end{aligned}$$

Thus, we see that $\widehat{||f(\mathbf{w})||}_u$ is unbiased under the assumption that $o(x) = f_x$. □

For our second alternative estimator, we can return to the idea of using the behavior of the "average element" in a bucket to approximate each w_x. The central estimator uses $||\mathbf{w}|| \cdot \frac{l_i + r_i}{2}$ as the weight of the average element in the bucket. However, this is a rather inflexible approximation. Another approach is to try and use the average weight in a bucket.

To do this, we need to count the number of unique items in the bucket. We cannot do this exactly, since elements may be split across key-value pairs and we require linear space to prevent double-counting of elements. However, we can note that $\mathbb{E}\left[\frac{W_x}{Nf_i}\right] = 1$. This means that we can approximate the number of unique elements in a bucket as

$$n_i \approx \sum_{x \in B_i} \frac{w_x}{N f_x}.$$

From here, we can say the average element of a bucket is $\frac{\sum_{x \in B_i} w_x}{n_i}$. We can translate this into the following estimator:

Estimator 3 (Counting Estimator)**.** *Define the two summary functions*

$$g_i^{(1)}(f) = 1$$
$$g_i^{(2)}(f) = \frac{1}{f}.$$

Then $\frac{S_i^{(2)}}{\sum_i S_i^{(1)}}$ *approximates the number of elements in bucket* i*. From here, we define the final estimator as*

$$\begin{aligned}\widehat{||f(\mathbf{w})||}_n &= \sum_i S_i^{(1)} \cdot \left(\frac{S_i^{(1)}}{S_i^{(2)} / \sum_i S_i^{(1)}}\right)^{d-1} \\ &= \left(\sum_i S_i^{(1)}\right)^{d-1} \left(\sum_i \frac{\left(S_i^{(1)}\right)^d}{\left(S_i^{(2)}\right)^{d-1}}\right).\end{aligned}$$

A third way we can approximate the average element is through sampling.

If each bucket uniformly samples a constant k_u number of elements, it can use that sample to estimate the average weight of an element in that bucket.

Estimator 4 (Sampling Estimator)**.** *Define k_u as some small constant. For each bucket, maintain a uniform sample $\mathcal{S}_i$ of k_u elements. Each bucket also maintains the summary function*

$$g_i^{(1)}(f) = 1.$$

To define the final estimator, let $\hat{w}_i$ be the average of each element in $\mathcal{S}_i$:

$$\hat{w}_i = \frac{1}{|\mathcal{S}_i|} \sum_{x \in \mathcal{S}_i} w_x.$$

Then the final estimator is

$$\widehat{||f(\mathbf{w})||}_s = \sum_i S_i^{(1)} \cdot \hat{w}_i^{d-1}.$$

3.3 Experiments

To evaluate the performance of the Bucketing sketch, I implement the baseline algorithms and all the estimators defined above.

3.3.1 Experimental Design

In order to get a thorough evaluation, I used one real dataset and several synthetic ones:

- A real dataset of AOL search data [26]. This dataset consists of search queries across multiple days, with around 3 million search queries per day. Each query is treated as the key-value pair $(x, 1)$ where x is the query; thus, the weight of a given query is the number of times it was searched. The goal is to estimate the frequency moment of one day of queries.

- A synthetic dataset of 1,000,000 elements generated as

$$W_i = \lfloor X_i \rfloor$$

  where each X_i is i.i.d. Pareto, with scale parameter 100 and shape parameter $\alpha = 0.1, 0.3, 0.5$. The goal is to estimate the frequency moment of this distribution. This allows us to test how the performance of our algorithm depends on the heavy tailed-ness of the dataset, with larger α parameters corresponding with less heavy tailed datasets.

I also explore three different types of oracles to see how the performance of the oracle affects the final sketch. These types are:

- **Relative**: $o(x) \sim \text{Unif}\left((1-\epsilon)f_x, (1+\epsilon)f_x\right)$. This multiplicative error is an ideal case where each key's estimate stays close to its true value. I use $\epsilon = 0.05$.
- **Absolute**: $o(x) \sim f_x + \text{Unif}\left(-\epsilon, \epsilon\right)$. This additive error captures an error model where estimates for large elements are substantially more precise than small elements. I use $\epsilon = 0.001$.
- **Train/Test**: To capture a more realistic oracle, I split each dataset into a training and testing set. I define $o(x)$ as the frequency of x within the training set or $\frac{1}{N}$ if it did not appear in the training set, where N is the length of the training set. I form the train/test split as follows:
  - For the AOL dataset, I use the first day of searches as the training set, and the second day of searches as the test set.
  - For the synthetic datasets, the training and test sets are independent generations.

To achieve a fair comparison between all the algorithms, I standardize them to all use k units of space. Each unit corresponds to a counter, which is used either to sample one element or to track the summary of a bucket.

For the bottom-k sketch, I chose to use the CountSketch [6] data structure as the heavy hitter sketch. This is a common mergeable sketch that achieves asymptotically small errors for its estimates, especially on power-law distributions that model common real-world datasets [23]. Following common practices [19], I initialize the CountSketch structure to have 7 number of rows. We want the length of each row s to be large, since the error falls by $O(1/s^2)$ when s is large [19]; thus, I set $s = \frac{1}{7} \cdot \frac{15}{16}k$, and have the sketch sample $\frac{k}{16}$ heavy hitters.

For the SWA sketch, three choices of k_h, k_p, and k_u were chosen to explore multiple configurations and capture the best performance in the various datasets:

1. $k_h = 0$, $k_p = k - 16$, $k_u = 16$.
2. $k_h = \frac{k-16}{2}$, $k_p = \frac{k-16}{2}$, $k_u = 16$.
3. $k_h = \frac{k}{2}$, $k_p = \frac{k}{4}$, $k_u = \frac{k}{4}$.

I also use two choices of k_h and k_B for the Bucketing sketch, where k_B is the total space allocated for the bucket summaries:

1. $k_h = 0$, $k_B = k$.
2. $k_h = \frac{k}{2}$, $k_B = \frac{k}{2}$.

For the sampling estimator, I also set $k_u = 16$. This was chosen to be a small constant to ensure that each bucket does not use too much space, but still enough for a decent sample.

Alongside these two configurations and the various estimators described above, I also explore two different bucketing schemes. The intent is to explore how much the bucketing scheme influences performance. This also

allows us to explore whether the scheme explored in the theoretical analysis is actually good or whether other schemes can perform better:

1. **Exponential buckets**: this follows the scheme used in the theoretical analysis above. I set γ after determining $f_{\min}$ to ensure that there are B buckets. Specifically, γ is defined as

$$\gamma = \exp\left(-\frac{\log(f_{\min})}{B-2}\right).$$

   I also choose $\epsilon_f = 10^{-9}$ for any elements where $o(x) = 0$.

2. **Linear buckets**: each bucket has the same width, where

$$l_i = \frac{i-1}{B}, \quad r_i = \frac{i}{B}.$$

I ran simulations for each of the following space sizes: $k = 2^6, 2^7, \ldots, 2^{16}$. For each space size, 10 estimates were taken to get a good capture of its behavior. I then calculate the Root Mean Squared Percentage Error (RMSPE), defined as

$$\text{RMSPE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{X_i - \hat{X}_i}{X_i}\right)^2} \tag{3.13}$$

where for each estimate, X_i is the true value and $\hat{X}_i$ is its estimate. This captures the algorithm's accuracy as a percentage deviation from the true value.

3.3.2 Bucketing Schemes

To start, let us compare the performance of the two bucketing schemes. From Figure 3.3.1, we can see that there is a clear performance difference. Looking at the estimators under the relative error oracle, we see that simply

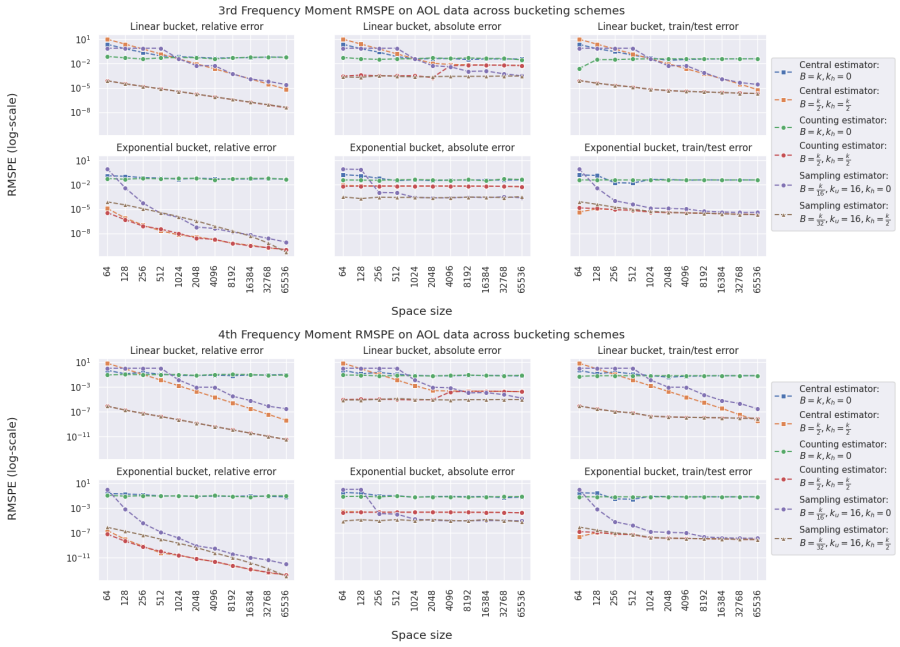

Figure 3.3.1: RMSPE achieved by various Bucketing sketch estimators, across both exponential and linear bucketing schemes.

changing the bucketing scheme can improve the performance by multiple orders of magnitude. Intuitively, it makes sense that the exponential buckets work better with our estimators. Since our estimators use an average element as an approximation, we want all the elements within a bucket to have similar frequencies. Given that our datasets are heavy-tailed, many elements will have small frequencies. Exponential buckets are much tighter in the small frequency range compared to linear buckets; as such, we should expect that exponential buckets perform better.

We should also note, however, that the performance gap largely fades away once the oracle errors become larger. With both absolute and train/test oracle errors, the performance gap virtually disappears. This suggests that once the oracle has larger errors in the small frequencies, exponential buckets lose their strength. As such, the choice of buckets may not be a major factor in practice.

For the remainder of the experiments, I will only show results with the exponential buckets for clarity.

3.3.3 Importance of Top-k_h Sampling

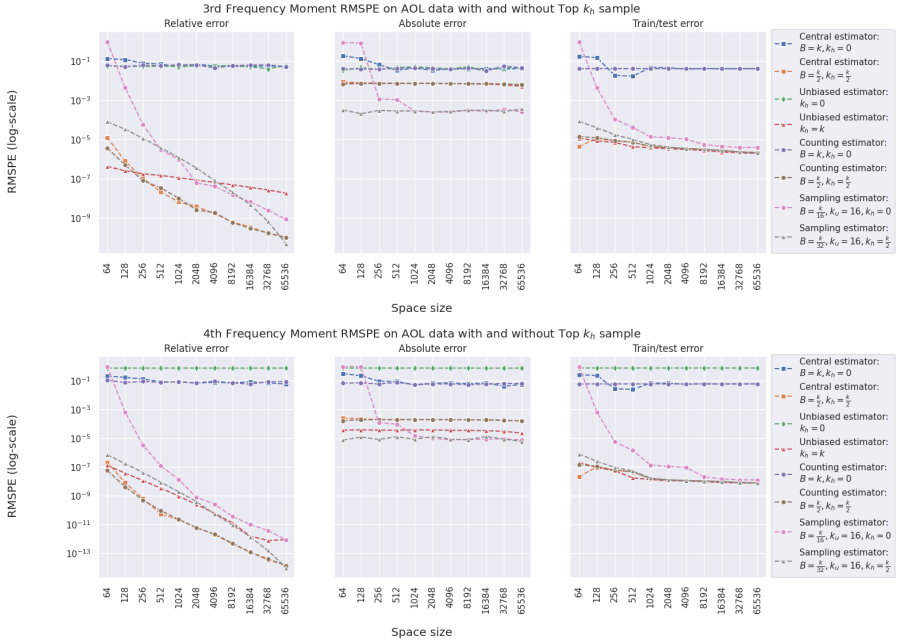

Figure 3.3.2: RMSPE achieved by various Bucketing sketch estimators, with and without a top-k_h sample.

As mentioned in subsection 3.2.1, we can choose to include a top-k_h sample alongside the Bucketing sketch. To see whether this is useful or not, we compare the performance of sketches with and without this sample.

Examining Figure 3.3.2, it is evident that adding the top-k_h sample is always beneficial. There is a performance improvement for every estimator and across every oracle error type. Its impact is most evident with the unbiased estimator. Since this estimator uses a constant number of counters, adding the top-k_h sample does not affect the Bucketing estimator's error. As such, this sketch isolates the benefit of the top-k_h sample, and we see that it

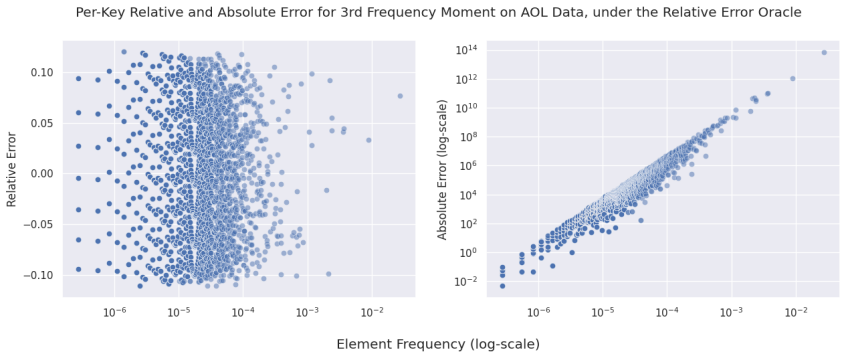

Figure 3.3.3: The per-key relative and absolute error achieved by the Bucketing sketch. Here, we use the central estimator, with $B = 1024$ buckets and relative oracle error. The per-key relative and absolute error are defined as $\frac{W_x^d - \widehat{W_x^d}}{W_x^d}$ and $\left|W_x^d - \widehat{W_x^d}\right|$ respectively.

reduces the error by over a factor of 10^{-7}. This is not to say that the Bucketing sketch is useless; despite allocating more space to its top-k_h sample, the unbiased estimator is never better than the other estimators. However, it is clear that allocating some space for a top-k_h sample is necessary for good performance; it would be interesting for future work to quantify this trade-off.

One way to understand where this performance improvement comes from is to look at the per-key errors achieved by the Bucketing sketch. Recall from the theoretical results that the central estimator, under an oracle with relative Δ error, achieves a relative ϵ error for every key. Figure 3.3.3 shows this exact behavior in practice. However, this small relative error translates into a fairly large *absolute error* for the more frequent elements. Thus, if we have a top-k_h sample that removes the error from these heaviest elements, we can imagine the overall estimation error dropping substantially.

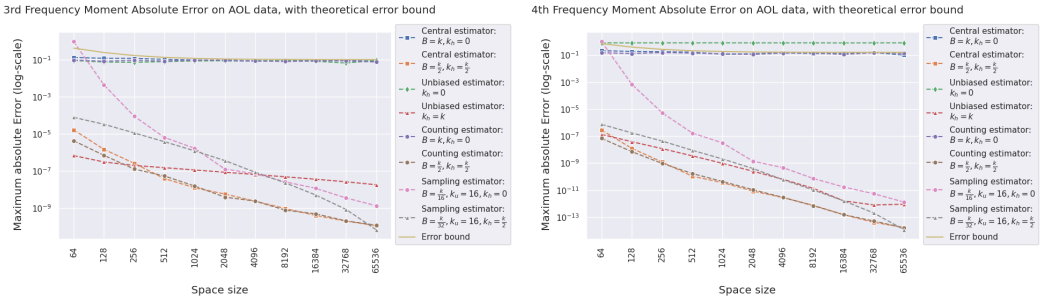

Figure 3.3.4: The theoretical bound plotted against the maximum absolute error ($\max_i \left|\frac{\hat{X}_i - X_i}{X}\right|$) achieved by the Bucketing sketch using the exponential bucketing scheme, under relative error. For the theoretical bound, we leave out the $O(1/\sqrt{N})$ term.

3.3.4 Comparison with Theoretical Bounds

The importance of the top-k_h sample is further highlighted once we examine the experimental performance against the theoretical bounds. Figure 3.3.4 shows that the central estimator without a top-k_h sample is fairly close to the theoretical bound. However, once the top-k_h sample is introduced, then the performance falls well below this bound.

This highlights one limitation of my theoretical analysis: it does not account for the fact that a top-k_h sample removes a large amount of variance from the estimate. Since the largest elements are now always counted, the estimates for the smaller elements have to fluctuate a lot more to create a sizeable error in the estimate. In other words, errors in the smallest estimates do not matter as long as the larger estimates are precise. The top-k_h sample essentially guarantees this.

3.3.5 Oracle Errors

From Figure 3.3.2, we can see that the oracle error model has a significant impact on the performance of the Bucketing sketch. If we compare Figure 3.3.3 and Figure 3.3.5, it is clear that the other oracle error models

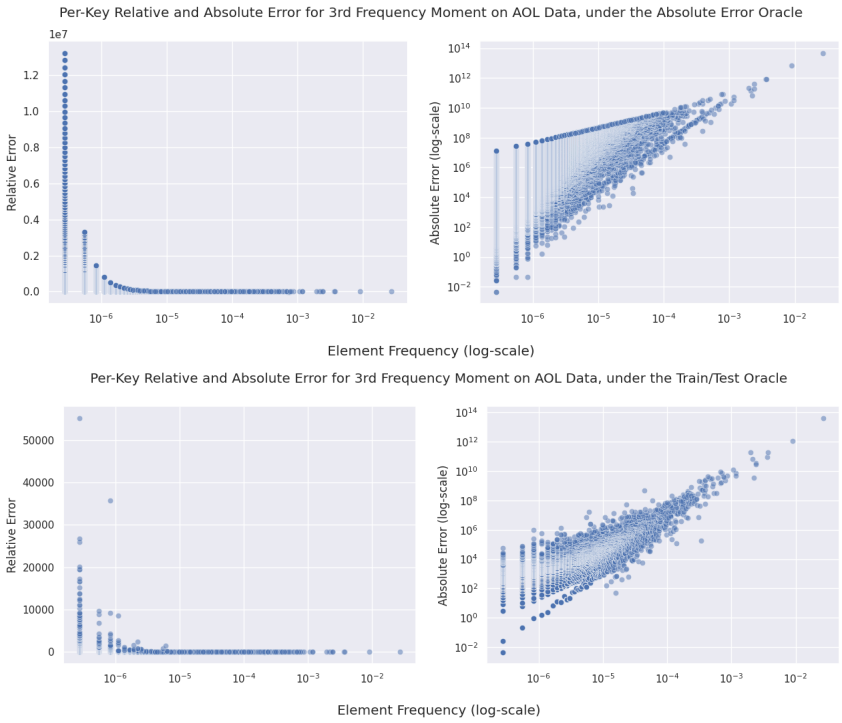

Figure 3.3.5: The per-key relative and absolute error achieved by the Bucketing sketch. Here, we use the central estimator, with $B = 1024$ buckets. The per-key relative and absolute error are defined as $\frac{W_x^d - \widehat{W_x^d}}{W_x^d}$ and $\left|W_x^d - \widehat{W_x^d}\right|$ respectively.

introduce more noise in the smallest elements. This leads to elements getting mixed together and a wider range of different frequencies being placed in each bucket, so the average approximations that our estimators rely on become more inaccurate.

This highlights how crucial a performant and reliable oracle is towards the performance of the Bucketing sketch. At the same time, however, the train/test oracle is still able to achieve reasonable error rates. This suggests that the oracle model chosen in my theoretical analysis may be overly strict. It may be possible to allow larger errors for smaller elements and obtain similar error bounds; this is one possible direction for future theoretical work.

3.3.6 Alternative Estimators

Figure 3.3.2 also shows the performance between the different Bucketing sketch estimators. To start, we can note that the central and counting estimators have nearly identical performance. Interestingly, the sampling estimator has a very different error profile, performing comparatively worse under relative oracle error and better under absolute oracle error. This suggests that sampling within buckets may be more robust once the elements in a bucket become less uniform and adopt a more skewed distribution.

All of the estimators, however, exhibit similar performance under train/test oracle error once a top-k_h sample is added. This is somewhat surprising, given the differences that are present under both relative and absolute error. One possible explanation is that the lower frequency elements are too mixed up between buckets, leading to some amount of irreducible noise. As such, once the more common elements are sampled by the top-k_h sample, no estimator can gain a meaningful edge over another.

3.3.7 Comparisons against Baseline Sketches

Now, let us compare how the Bucketing sketch performs against the baseline sketches: Bottom-k and SWA. Figure 3.3.6 shows that in every case the Bucketing sketch is no worse than SWA. In fact, under relative error, the Bucketing sketch performs up to an order of magnitude better. This reflects how the Bucketing sketch can better approximate the mid-sized elements that bottom-k and SWA leave out from their sample, validating the motivation that I used when designing the Bucketing sketch.

We also see that bottom-k's performance continues to improve as the space allocated increases. This is in contrast to the sketching algorithms, which end up plateauing under absolute and train/test oracle errors. This suggests that in applications where a good oracle is difficult to obtain, standard sketching algorithms may still be the best choice. However, bottom-k was

Figure 3.3.6: RMSPE achieved by the Bucketing sketch compared with PPSWOR and SWA, on the AOL dataset.

unable to surpass the other estimators under the space sizes tested. In the small-space regime, sketching algorithms with advice still perform the best.

3.3.8 Additional datasets

Using the synthetic datasets, we can explore how the Bucketing sketch performs based on how heavy the tail of the dataset is. Figure A.0.1 and Figure A.0.2 show that the Bucketing sketch begins to lose its performance gap to bottom-k and SWA, with the gap shrinking more as the tail gets lighter. Looking at the performance under train/test error, we see that the Bucketing sketch still has the lowest error when the space size is small. This is especially true for the unbiased estimator. Interestingly, the unbiased estimator was never the best on the AOL dataset; this likely reflects how the synthetic datasets are more idealistic, as elements are likely clustered around their expected value.

Thus, Figure A.0.1 and Figure A.0.2 solidify two contexts where the Bucketing sketch performs well: when there is low oracle error, and when the space size is small. In these two contexts, the Bucketing sketch repeatedly achieves lower RMSPE than all the other sketching algorithms.

4

Further Applications to Challenging Problems

In the previous chapter, I showed that the Bucketing sketch is able to achieve a performance comparable to that of SWA when estimating the high-frequency moments of a dataset, and both algorithms are able to outperform the Bottom-k sketch.

In this chapter, I extend these results by focusing on settings that are difficult for SWA to provide good estimates for. Through empirical experiments, I consider two settings: turnstile streams with negative updates, and estimating quantiles of aggregate weights.

4.1 Turnstile Streams

Previously, we considered cash register streams where all the item updates are positive. However, certain scenarios may require negative updates to the dataset. For example, if our dataset is the number of sales for various items from an online marketplace, customers may occasionally choose to return their purchases, causing negative updates to the number of sales. The generalization of the cash register stream to negative updates is known as the turnstile model [9].

Definition 4.1.1 (Turnstile Data Stream)**.** *A turnstile data stream is a stream of elements* (x_t, v_t) *where* $t = 0, 1, 2, \ldots, T$*. Just like cash register streams (Definition 2.1.1),* x_t *comes from the universe of possible keys* $\mathcal{X}$*.* $v_t \in \mathbb{Z}$ *is a positive or negative integer update. The weight of each key* w_x *is still defined to be the sum of all its corresponding values.*

Although individual updates can be negative, we assume the weight of an element never drops below 0 for all t*.*

The introduction of negative updates makes turnstile streams much harder to process. Take SWA as an example. With cash register streams, we assumed that once an element is pushed out of the SWA sample, it can never return. Negative updates invalidate this assumption. Now, the weight of an element might be updated to 0, removing it from the dataset and creating a vacancy in the sample to which a previously removed element may return. However, SWA does not track information about elements outside of its sample; thus, it has no way of knowing what element should fill this vacancy and what its weight is.

The bucketing sketch does not face this problem. Since every element gets placed in a bucket, it does not matter if its weight increases or decreases; the corresponding summaries will be correct at query time. In fact, since none of the final weights w_x are negative, the theoretical analysis performed in Chapter 3 also holds for turnstile streams.

However, the top-k_h sample does need to be modified for turnstile streams. The sample should not include elements that have been placed in buckets once a vacancy is created, since there is no way to recover the weights of elements once they are placed in the buckets. This can be achieved by keeping elements in the sample even if their weight has dropped to 0, and only replacing them with elements whose oracle frequency is larger than that of the smallest oracle frequency in the sample: $\min_{x \in \text{top-}k_h} o(x)$. These elements are guaranteed not to have been placed in the buckets.

Algorithm 6 Top-k_h sample for turnstile streams

```
1:  procedure Initialize(k_h, o(x))
2:      S ← empty sample of k_h elements, storing (x, v, o(x)), sorted by o(x)
3:  end procedure
4:  procedure ProcessElement(x, v)
5:      if x ∈ S then
6:          S[x].v ← S[x].v + v
7:      else
8:          if o(x) > S[k_h].o then
9:              for (x', v', o') ∈ S do                ▷ Check for vacancies
10:                 if v' = 0 then
11:                     replace entry with (x, v, o(x))
12:                     return
13:                 end if
14:             end for
15:             Insert (x, v, o(x)) into S             ▷ Insert normally into sample.
16:         end if
17:     end if
18: end procedure
19: procedure QueryEstimate
20:     est ← 0
21:     for (x, v, o) ∈ S do
22:         est ← est + v
23:     end for
24:     return est
25: end procedure
```

Just like SWA, it may be possible for the top-k_h sample to have 0 elements at the end of the stream, if all the elements happen to have their weights updated to 0. However, this is not an issue for the bucketing sketch because the remaining elements are still tracked by the bucket summaries; the result is the same as if the top-k_h sample was never used.

4.1.1 Experimental Design

To evaluate the performance of the bucketing sketch on a realistic turnstile stream, I focus once again on the problem of estimating high frequency moments. I used a real dataset of anonymized passive traffic traces taken by a real ISP in 2002, collected by CAIDA [4]. Each captured packet is treated as the key-value pair $(\langle \text{srcip}, \text{dstip} \rangle, 1)$ where the key is the pair of its source and destination IP addresses. Similar to [18], I then use this dataset to simulate a turnstile model by setting up a sliding window of 10 minutes. Since CAIDA splits up the dataset into packet captures of 5-minute intervals, I store 2 of these intervals at once, and remove all the packets in the earlier 5-minute interval before adding in the next one. The goal is to estimate the frequency moment every time I update the window, yielding 10 estimates across a 55-minute period.

Similar to the experiments in Chapter 3, I explore three different types of oracles. To recap, these three oracles are:

- **Relative**: $o(x) \sim \text{Unif}\left((1-\epsilon)f_x, (1+\epsilon)f_x\right)$, where $\epsilon = 0.05$.
- **Absolute**: $o(x) \sim f_x + \text{Unif}\left(-\epsilon, \epsilon\right)$, where $\epsilon = 0.001$.
- **Train/Test**: $o(x)$ is the frequency of x within the training set or $\frac{1}{N}$ if it did not appear in the training set, where N is the length of the training set. Here, I use the first 5-minute interval as the training set, and the next 55 minutes as the test set.

Now, however, the definition of f_x for the relative and absolute errors are slightly unclear. Since we are adding and removing elements between queries,

the true frequency of f_x is different every time we estimate the frequency moment. However, the bucketing sketch requires the oracle estimate to be stable; they cannot change when we move from one window to another. As such, I choose to define f_x as the total frequency of x across the entire 55-minute period of test packets. The relative and absolute errors are then defined according to this definition.

The estimators I evaluate are the central estimator, unbiased estimator, and counting estimator, as defined in Chapter 3. I leave out the sampling estimator because it does not work under turnstile streams, for the same reasons that SWA does not work.

I also keep the same two choices of k_h and k_B, to see how much of a difference the top-k_h sample adds in the turnstile setting. The same space sizes are tested as well: $k = 2^6, 2^6, \ldots, 2^{16}$.

4.1.2 Experimental Results

Figure 4.1.1 shows the results of these experiments. Immediately, we can see that estimation for this dataset is much more challenging. Only the relative error oracle achieves an RMSPE below 0.1, while the best estimators under train/test error achieve RMSPEs around 0.4 for both the 3rd and 4th frequency moments.

We also see that the performance does not improve as the space size increases, which implies that the performance is being limited by the oracles. This highlights the additional complexity introduced by turnstile streams. Now that elements can be added or removed, the assumption that oracle and element frequencies are stable may not be a good assumption anymore. Since we require the estimated oracle frequencies to stay within a small multiplicative factor of the true frequencies, the performance of the bucketing sketch is severely limited if the true frequencies of elements fluctuate a lot.

Just like when estimating frequency moments, the top-k_h sample is still necessary for good performance. Even though it is not guaranteed to sample

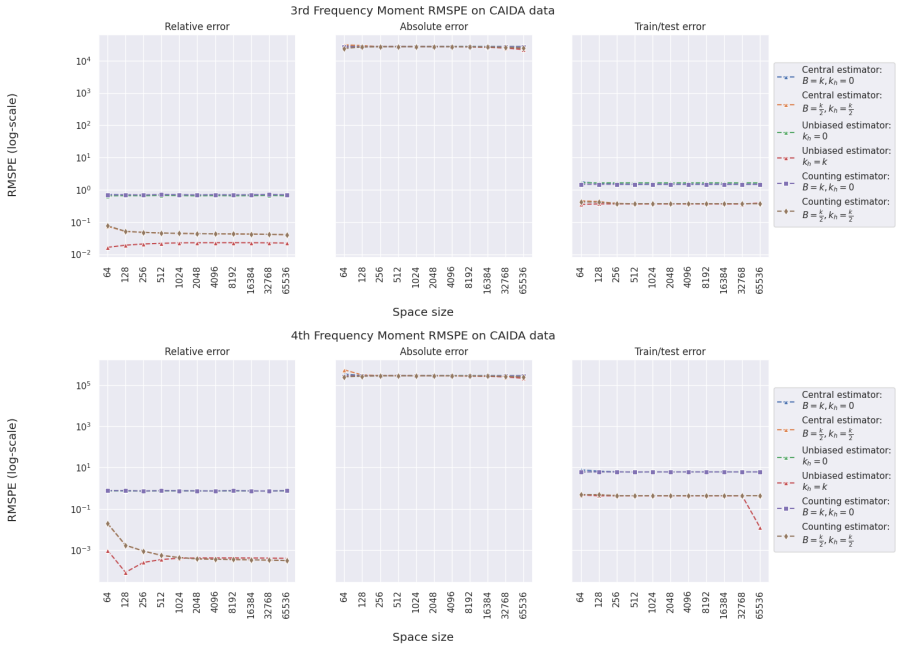

Figure 4.1.1: RMSPE achieved by various Bucketing sketch estimators on the CAIDA turnstile model dataset. All of these estimators use the exponential bucketing scheme.

k_h elements in the turnstile setting, in practice the sample remains filled. After all, the most common elements are unlikely to be completely removed from the dataset. Real-world applications are more likely to resemble the bounded deletion turnstile model [17], where the number of deletions remains small. In this setting, our reservations about the usability of such samples may be less relevant, since the majority of elements will not see a large number of deletions.

4.2 Estimating Quantiles

The quantiles of a dataset are another group of summary statistics that is widely used in practice [22]. Quantiles like the median (50th percentile) help capture the common behavior of the dataset, while the tail quantiles (e.g. 99th percentile) have become ubiquitous in monitoring for anomalies and

outliers [10]. These statistics are also impossible to calculate exactly without storing all the data in memory [25]; nevertheless, there is a rich literature of sketching algorithms to estimate quantiles. [22].

Every existing quantile sketching algorithm, however, works over a stream of *data values* instead of a stream of key-value pairs. In other words, they do not worry about aggregating multiple elements across a data stream. This is natural if you are receiving many independent measurements—a common example is if you are measuring website latencies and you want to monitor the 99th percentile tail latency [10]. Unfortunately, not every use case fits this description. As a motivating example, imagine you are running an online marketplace and you want to monitor how well different items are selling. This would require you to aggregate counts across many different sales, which the current algorithms do not support.

The Bucketing sketch is able to provide one solution to this problem. In this section, I design and experimentally test a quantile estimator that works over aggregated data. These results serve more as a proof-of-concept; I do not provide any theoretical guarantees about the performance of this estimator. Nevertheless, I intend for this section to further show the flexibility of the Bucketing sketch framework, and to inspire future work into this problem, which I believe has been overlooked in the literature so far.

4.2.1 Formal Definitions

To start, let us define what the quantile of a dataset is:

Definition 4.2.1 (Quantile)**.** *Given a dataset over the elements $x \in \mathcal{X}$, each with weight w_x, the qth quantile is the smallest weight w_q such that $q|\mathcal{X}|$ elements have a weight smaller than or equal to w_q.*

Equivalently, we can consider calculating the rank in a dataset of a given value:

Definition 4.2.2 (Rank)**.** *Given a dataset over the elements $x \in \mathcal{X}$, each*

with weight w_x, the rank $R(w)$ of the value w is equal to the fraction of elements whose weight is smaller than or equal to w.

Note that $R(w_q) = q$. This means if we have an algorithm to estimate one value, we can estimate the other value as well. For instance, if we can estimate the rank $R(w)$, then given a target q we can binary search over w to find when $R(w) = q$.

A quantile sketching algorithm should be able to estimate these values for *every possible input at query time*. This means that the quantile (or rank) to be estimated does not need to be determined ahead of time. Instead, the sketching algorithm should perform a single pass over the data, and then answer queries for an arbitrary quantile or rank.

4.2.2 Defining the estimator

At its core, this estimator attempts to model the distribution of elements within every bucket of the sketch. By doing so, the estimator can calculate how many elements are less than or equal to some weight w, according to the modeled distributions. Taking this number and dividing it by the total number of elements in the dataset then yields the estimated rank of w.

Thus, the main challenge is modeling the distribution of elements within each bucket. A naive approach is to treat each bucket as approximately uniform, just as we did for the high-frequency moment estimators in Chapter 3. This means we can use the same strategies to estimate the number of elements within a bucket, then model their frequencies as a linear interpolation between the endpoints of the bucket. The implementation for this is shown in Algorithm 7.

Unfortunately, this does not perform very well. Looking at the rank errors ($|R_{\text{estimate}}(w) - R_{\text{actual}}(w)|$) in Figure 4.2.1, we can see that it achieves very high rank error for the smallest elements—up to 0.61 under the train/test oracle error model. The uniform approximation worked well for the

Algorithm 7 Naive Quantile Sketch

procedure INITIALIZE($B[], o$) ▷ *$B[]$: list of buckets*
  $S[B][1] \leftarrow$ array of size $B \times 1$ for bucket summaries ▷ *Track total sum per bucket*
end procedure
procedure PROCESSELEMENT(x, v)
  Find bucket b such that $o(x) \in (B[b].l, B[b].r]$
  $S[b][0] \leftarrow S[b][0] + v$
end procedure
procedure QUERYESTIMATE(w) ▷ *w: Element to estimate the rank of*
  $num\_smaller \leftarrow 0$
  $total\_elements \leftarrow 0$
  $N \leftarrow \sum_b S[b][0]$ ▷ *Total weight of dataset*
  for $b \leftarrow 0 \ldots B-1$ **do** ▷ *Calculate elements smaller than w*
    $m \leftarrow B[b].l \cdot N$ ▷ *Estimated minimum weight in bucket*
    $M \leftarrow B[b].r \cdot N$ ▷ *Estimated maximum weight in bucket*
    $n \leftarrow \frac{S[b][0]}{(m+M)/2)}$ ▷ *Estimate count in bucket*
    if $m \leq w \leq M$ **then** ▷ *Estimate number of elements smaller than w, using linear interpolation*
      $num\_smaller \leftarrow num\_smaller + \lfloor \frac{(w-m)(n-1)}{M-m} \rfloor + 1$
    end if
    $total\_elements \leftarrow total\_elements + n$
  end for
  return $\frac{num\_smaller}{total\_elements}$
end procedure

high-frequency moment estimators because they did not require very accurate estimates for the smallest weights. Quantile sketches, however, cannot afford large errors for any element.

Intuitively, there are two main reasons for this failure. The first is that the weights of elements within a bucket are not always bounded by the bucket endpoints. The endpoints only bound the estimated frequencies from the oracle, and due to oracle errors (or pure randomness), the true weights of elements may be larger or smaller than these bounds. As such, the

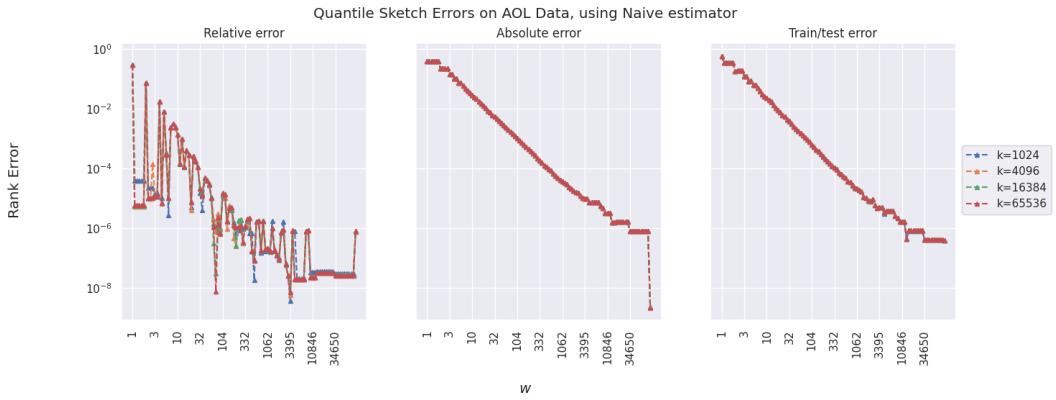

Figure 4.2.1: Rank error achieved by the naive quantile sketch, on the AOL dataset. The maximum rank error under each oracle error model is 0.30, 0.39, and 0.61 for the relative, absolute, and train/test error models respectively.

distribution model should take into account the actual observed weights of the elements.

The second reason is that the dataset simply does not match this kind of linear approximation. Real-world datasets are much more likely to be heavy-tailed [8]; as such, the models should mirror this.

Combining these insights leads us to the final estimator. Using the results of Chapter 3 as guidance, I estimate the minimum and maximum weight elements in a bucket by uniformly sampling a small number of elements. Then, I model the distribution within the bucket as following draws from a Zipfian distribution. This distribution, defined over the positive integers, is parameterized by the real number $s \geq 0$ and has the PDF

$$P(X = x) = \frac{1}{C}\frac{1}{x^s}$$

where C is the normalizing constant.

Definition 4.2.3 (Model of Bucket Distribution)**.** *Let $\widehat{m}$ and $\widehat{M}$ be the estimated minimum and maximum weight for elements in the bucket, $N = \sum_x w_x$ be the total weight of the dataset, and s be the chosen parameter*

for the Zipfian distribution. Define

$$\widehat{k_m} = \left\lceil \left(\frac{N}{\widehat{m}}\right)^{1/s} \right\rceil \quad \textit{and} \quad \widehat{k_M} = \left\lfloor \left(\frac{N}{\widehat{M}}\right)^{1/s} \right\rfloor$$

to be the estimated range of elements included in the bucket. Then the distribution of elements is modeled as

$$\left\lfloor \frac{N}{k^s} \right\rfloor \quad \textit{for } k = \widehat{k_M}, \ldots, \widehat{k_m}.$$

To actually choose the parameter s, we can fit the modeled distribution against the observed total weight of a bucket. By searching over s, we can find the best fit for each individual bucket. The implementation for this is provided in Algorithm 8.

Altogether, this yields the bucketing quantile sketch seen in Algorithm 9.

Algorithm 8 Fitting the Zipfian Model

```
1:  procedure FitZipfianModel(N, m, M, sum)  ▷N: total weight, m: min bucket weight, M: max bucket weight
2:      Find s such that EstimatedSum(N, m, M, s) = sum
3:      return s
4:  end procedure
5:  procedure EstimatedSum(N, m, M, s)
6:      k_m ← ⌈(N/m)^{1/s}⌉
7:      k_M ← ⌊(N/M)^{1/s}⌋
8:      sum ← 0
9:      for k ← k_m ... k_M do
10:         sum ← sum + ⌊N/k^s⌋
11:     end for
12:     return sum
13: end procedure
```

Figure 4.2.2: Rank error achieved by the bucketing quantile sketch, on the AOL dataset. The maximum rank error under each oracle error model is 0.001, 0.13, and 0.36 for the relative, absolute, and train/test error models respectively.

4.2.3 Experimental Design

To evaluate this algorithm, I used the same real and synthetic datasets as Chapter 3. For each dataset, I ran the the sketch using four different space parameters: $k = 1024, 4096, 16384, 65536$. I use a constant size $k_u = 16$ for the uniform sample of each bucket. I also implement the search for s as a binary search between $s = 0.5$ and $s = 5$, stopping once the step size drops below 10^{-5}.

After each run, I queried the rank of 100 weights, spaced out across the entire weight range of the dataset. I then measured the rank error for each query:

$$\text{RankError}(w) = |R_{\text{estimate}}(w) - R_{\text{actual}}(w)|\,.$$

4.2.4 Experimental Results

Examining Figure 4.2.2, we can see that the second estimator performs much better than the naive estimator across the entire data range. Even under the more difficult oracle models, the rank error rarely goes above 10%. This means that an element estimated to be the median could be anywhere

between the 45th and 55th percentiles; in some applications, this discrepancy may be acceptable.

Interestingly, these experiments show that allocating more space to the sketch does not actually improve its performance. All four space parameters exhibit similar behavior in their rank error, suggesting that our sketch is limited by the accuracy of the Zipfian model I apply. This is likely due to the fact that the Zipfian model is rather simplistic, with only one parameter s that can be tweaked. To achieve even better models, it could be interesting to explore moment matching algorithms using the estimates of the frequency moments of a bucket (which we now know we can do!). A similar idea has already been explored for the standard quantile sketching problem by [13]; if the moment estimates for each bucket are not too noisy, this method may provide additional improvements compared to the Zipfian model.

The performance on the synthetic datasets is similar. Figure A.0.3 also shows rank errors that largely stay below 10%, and which get better for larger elements. We also see that the space used does not have an effect on the rank errors, suggesting that this is a general drawback of the bucketing quantile sketch and not just an artifact of the AOL dataset. Overall, however, the success of the sketch on both real and synthetic datasets suggests that it is well-suited for many types of heavy-tailed applications.

Algorithm 9 Bucketing Quantile Sketch

```
1: procedure INITIALIZE(B[], k_u, o)                    ▷B[]: list of buckets
2:     S[B][1] ← array of size B × 1 for bucket summaries  ▷Track total sum per bucket
3:     Samples[B] ← empty uniform sample of size k_u for each bucket
4: end procedure
5: procedure PROCESSELEMENT(x, v)
6:     Find bucket b such that o(x) ∈ (B[b].l, B[b].r]
7:     S[b][0] ← S[b][0] + v
8:     Update Samples[b] with (x, v)
9: end procedure
10: procedure QUERYESTIMATE(w)                          ▷w: Element to estimate rank
11:     num_smaller ← 0
12:     total_elements ← 0
13:     N ← ∑_b S[b][0]                                 ▷Total weight of dataset
14:     for b ← 0 ... B − 1 do
15:         m ← min(Samples[b])
16:         M ← max(Samples[b])
17:         if m = M then                               ▷All the elements are the same weight
18:             n ← S[b][0]/m
19:             if w >= m then
20:                 num_smaller ← num_smaller + n
21:             end if
22:             total_elements ← total_elements + n
23:         else                                        ▷Use Zipfian model
24:             s ← FITZIPFIANMODEL(N, m, M, S[b][0])
25:             k_m ← ⌈(N/m)^{1/s}⌉
26:             k_M ← ⌊(N/M)^{1/s}⌋
27:             for k ← k_m ... k_M do
28:                 if w ≥ ⌊N/k^s⌋ then
29:                     num_smaller ← num_smaller + 1
30:                 end if
31:             end for
32:             total_elements ← total_elements + (k_m − k_M + 1)
33:         end if
34:     end for
35:     return num_smaller/total_elements
36: end procedure
```

5 Conclusion

In this thesis, I explored a new framework for creating sketching algorithms that leverage advice: information about individual elements that can guide the algorithm to achieve better estimates. I started with two key observations: first, that we can collect items with **similar weights** and apply a single weight estimate for all of them; and second, that we can use advice models to obtain **stable advice** for each item, avoiding the problem of moving items between groups.

Using these two observations, I developed the Bucketing sketch: a framework that places elements into buckets, maintains summaries for each bucket, and uses these summaries to provide a final estimator. This simple framework only requires a single pass over the data stream and can be parallelized across disjoint partitions of the dataset, making it well-suited to the distributed applications where modern sketching algorithms are often

applied today [2].

From here, I focused on the problem of estimating high-frequency moments. I started with a simple estimator, the Central Estimator, that can be implemented by the bucketing sketch. Despite its simplicity, I showed that this estimator can achieve a small (ϵ, δ) bound using logarithmic space in the size of the dataset. These bounds only required a small assumption about the underlying process that generated the dataset; although they do not cover worst-case scenarios that require polynomial space for estimation, I argued that my assumptions are general enough to cover many realistic data-generating processes.

Experimental evaluations demonstrated the sketch's practical effectiveness. By comparing a number of baseline algorithms with multiple bucketing sketch estimators, I showed that the Bucketing sketch performs very well in practice. When the advice model has very small errors, my sketch outperforms the baseline algorithms. As the advice model becomes less precise, the performance gap shrinks, but my sketch consistently matches the performance of the existing algorithms across a range of space sizes.

Finally, I ended this thesis with an exploration of some more challenging problems. I focused on turnstile streams and estimating quantiles of aggregate values, two problems for which the baseline algorithms are ill-suited. In both problems, I experimentally showed that the bucketing sketch can achieve good performance. This highlights the flexibility of my framework to apply to a variety of applications.

In conclusion, this thesis introduces a promising framework for creating sketching algorithms that use advice. As modern statistics and machine learning techniques become better, faster, and more widespread, I believe that it will become more natural for algorithms to rely on advice models. While I have shown the promise of this particular framework, I have only scratched the surface; there is a rich landscape of bucketing schemes, summaries, and estimators to be discovered. Even if this specific approach does not have more promise, I hope that this thesis inspires further

investigation into the strengths of advice-based algorithmic design.

5.1 Future Work

To wrap up this thesis, I provide some concluding thoughts on future work that can be built on top of the Bucketing sketch. Some of these directions are specific to this framework; others may be of independent interest for the general problem of sketching algorithms with advice.

Improving Estimators: Although the estimators I developed perform well in practice, their optimality remains uncertain. It would be interesting for future work to try and design better estimators, or to prove lower bounds that describe just how good an estimator can be.

Parsimonious Querying: This idea comes from [16], who made the observation that it is usually expensive to query advice models. As such, it would be preferable if the algorithm does not need to query the advice model for every element in the data stream. It is unclear how this idea could be applied to the bucketing sketch, since we rely on advice to simply put elements into their bucket. This may simply be an impossible problem; however, I still offer it as an interesting and potentially fruitful direction for future work.

Detecting Oracle Drift: When deploying advice-based algorithms in practice, it would be useful to monitor if the oracle frequencies drift significantly from the true frequencies. Since the performance of these algorithms directly depends on the accuracy of the oracle, detecting oracle failure would enable a graceful fallback to non-advice sketching algorithms. This problem is fundamentally a hypothesis test: determining if the data set distribution matches the oracle's predicted distribution. Solving this in the streaming context would be useful, not just for the bucketing sketch but for any future streaming algorithm that relies on frequency advice. Such a method could help identify distribution drifts or adversarial attacks that compromise oracle reliability before they impact estimation quality.

A

Additional Figures

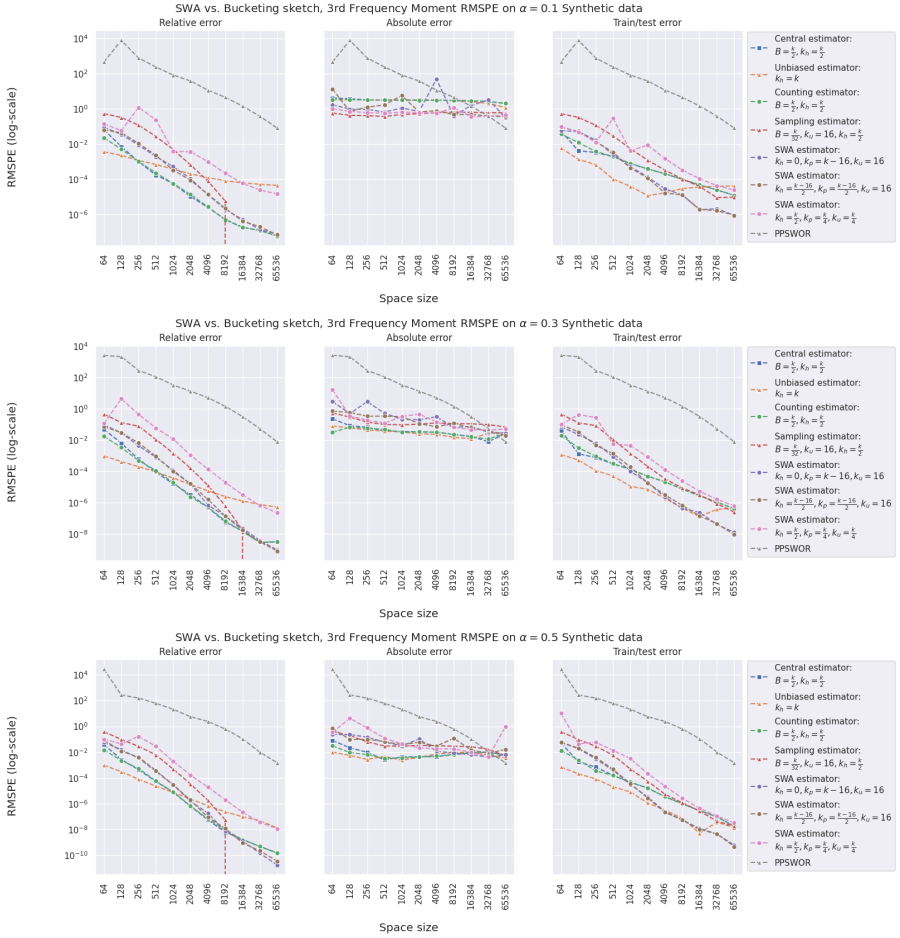

Figure A.0.1: RMSPE achieved by the Bucketing sketch compared with PPSWOR and SWA on the synthetic datasets, for 3rd frequency moments.

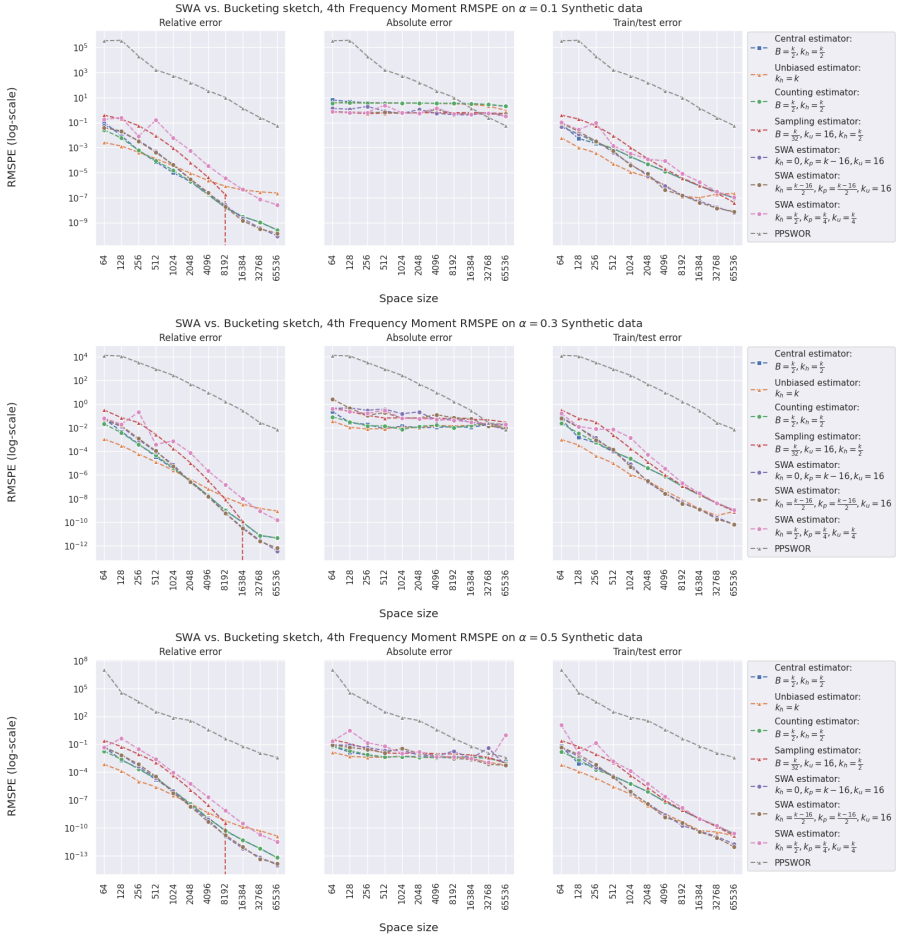

Figure A.0.2: RMSPE achieved by the Bucketing sketch compared with PPSWOR and SWA on the synthetic datasets, for 4th frequency moments.

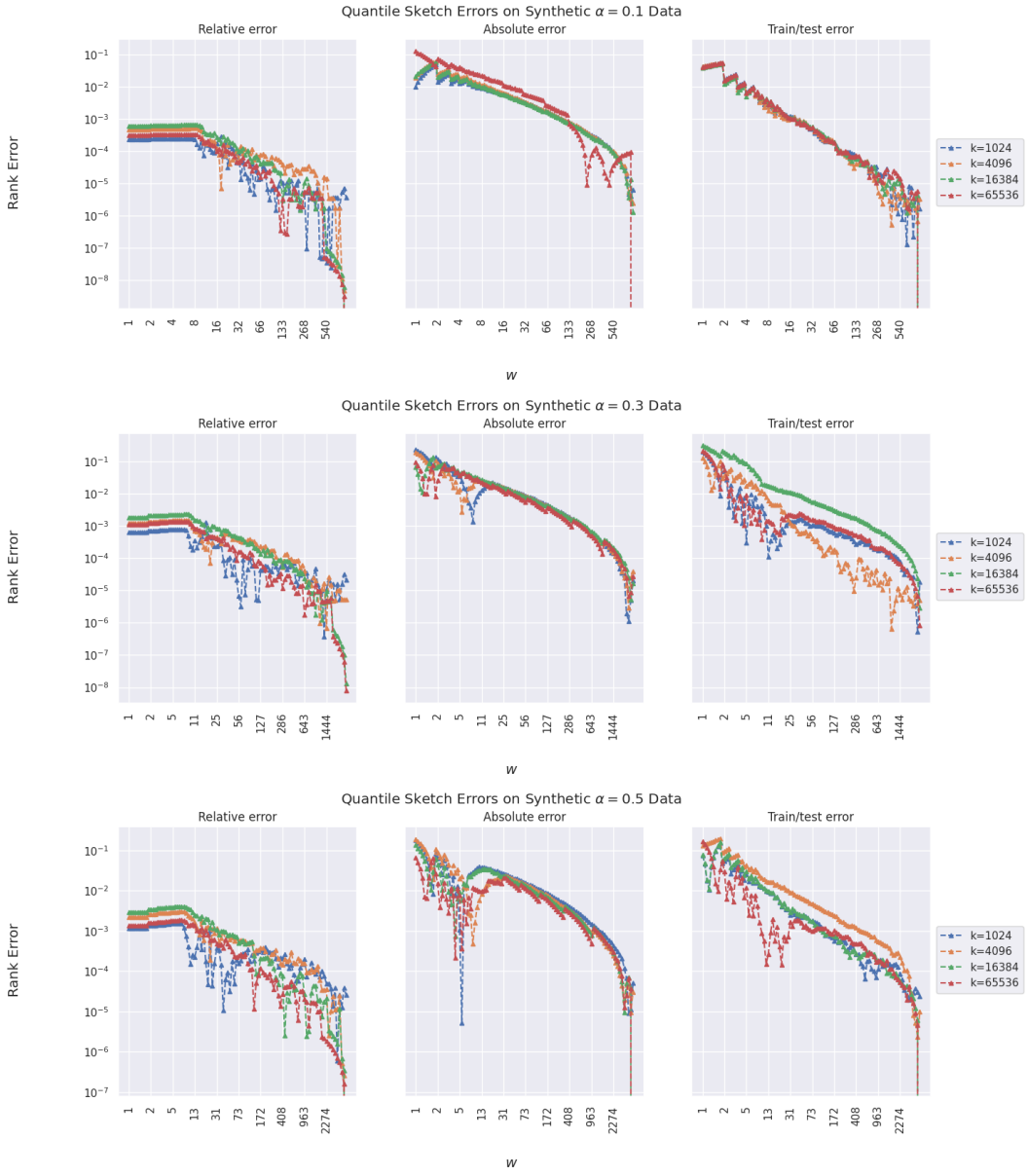

Figure A.0.3: Rank error achieved by the bucketing quantile sketch, on the synthetic datasets.

B

Code Sources

All the code used to run the experiments and generate the graphs seen in this thesis can be found at https://github.com/ac-dap/thesis.

Bibliography

[1] Anders Aamand, Justin Chen, Huy Nguyen, Sandeep Silwal, and Ali Vakilian. Improved frequency estimation algorithms with and without predictions. *Advances in Neural Information Processing Systems*, 36, 2024.

[2] Pankaj K Agarwal, Graham Cormode, Zengfeng Huang, Jeff M Phillips, Zhewei Wei, and Ke Yi. Mergeable summaries. *ACM Transactions on Database Systems (TODS)*, 38(4):1–28, 2013.

[3] Apache Software Foundation. Apache datasketches. https://datasketches.apache.org/. Accessed: February 26, 2025.

[4] CAIDA, UC San Diego. The CAIDA UCSD Anonymized Passive OC48 Internet Traces Dataset – 2002-08-14, 2002–2003. Accessed: 2025-02-28.

[5] Timothy M Chan and Eric Y Chen. Multi-pass geometric algorithms. In *Proceedings of the twenty-first annual symposium on Computational geometry*, pages 180–189, 2005.

[6] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 312(1):3–15, 2004. Automata, Languages and Programming.

[7] Edith Cohen. Sampling big ideas in query optimization. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles*

of Database Systems, PODS '23, page 361–371, New York, NY, USA, 2023. Association for Computing Machinery.

[8] Edith Cohen, Ofir Geri, and Rasmus Pagh. Composable sketches for functions of frequencies: Beyond the worst case. In *International Conference on Machine Learning*, pages 2057–2067. PMLR, 2020.

[9] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[10] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[11] Talya Eden, Dana Ron, and C Seshadhri. Sublinear time estimation of degree distribution moments: The arboricity connection. *SIAM Journal on Discrete Mathematics*, 33(4):2267–2285, 2019.

[12] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete mathematics & theoretical computer science*, (Proceedings), 2007.

[13] Edward Gan, Jialin Ding, Kai Sheng Tai, Vatsal Sharan, and Peter Bailis. Moment-based quantile sketches for efficient high cardinality aggregation queries. *arXiv preprint arXiv:1803.01969*, 2018.

[14] Chen-Yu Hsu, Piotr Indyk, Dina Katabi, and Ali Vakilian. Learning-based frequency estimation algorithms. In *International Conference on Learning Representations*, 2019.

[15] Elena Ikonomovska and Mariano Zelke. Algorithmic Techniques for Processing Data Streams. In Phokion G. Kolaitis, Maurizio Lenzerini, and Nicole Schweikardt, editors, *Data Exchange, Integration, and*

Streams, volume 5 of *Dagstuhl Follow-Ups*, pages 237–274. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2013.

[16] Sungjin Im, Ravi Kumar, Aditya Petety, and Manish Purohit. Parsimonious learning-augmented caching. In *International Conference on Machine Learning*, pages 9588–9601. PMLR, 2022.

[17] Rajesh Jayaram and David P Woodruff. Data streams with bounded deletions. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 341–354, 2018.

[18] Cheqing Jin, Weining Qian, Chaofeng Sha, Jeffrey X Yu, and Aoying Zhou. Dynamically maintaining frequent items over a data stream. In *Proceedings of the twelfth international conference on Information and knowledge management*, pages 287–294, 2003.

[19] Kasper Green Larsen, Rasmus Pagh, and Jakub Tětek. Countsketches, feature hashing and the median of three. In *International Conference on Machine Learning*, pages 6011–6020. PMLR, 2021.

[20] Yi Li, Huy L. Nguyen, and David P. Woodruff. Turnstile streaming algorithms might as well be linear sketches. In *Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '14, page 174–183, New York, NY, USA, 2014. Association for Computing Machinery.

[21] Yi Li and David P Woodruff. A tight lower bound for high frequency moment estimation with small error. In *International Workshop on Approximation Algorithms for Combinatorial Optimization*, pages 623–638. Springer, 2013.

[22] Charles Masson, Jee E Rim, and Homin K Lee. Ddsketch: A fast and fully-mergeable quantile sketch with relative-error guarantees. *arXiv preprint arXiv:1908.10693*, 2019.

[23] Gregory T Minton and Eric Price. Improved concentration bounds for count-sketch. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 669–686. SIAM, 2014.

[24] Michael Mitzenmacher and Sergei Vassilvitskii. Algorithms with predictions. *Commun. ACM*, 65(7):33–35, June 2022.

[25] J Ian Munro and Mike S Paterson. Selection and sorting with limited storage. *Theoretical computer science*, 12(3):315–323, 1980.

[26] Greg Pass, Abdur Chowdhury, and Cayley Torgeson. A picture of search. In *Proceedings of the 1st International Conference on Scalable Information Systems*, InfoScale '06, page 1–es, New York, NY, USA, 2006. Association for Computing Machinery.

[27] Reids. Redis top-k probabilistic data structure. Accessed: March 14, 2025.

[28] Florin Rusu and Alin Dobra. Sketches for size of join estimation. *ACM Transactions on Database Systems (TODS)*, 33(3):1–46, 2008.