STREAMING ALGORITHMS FOR MINING FREQUENT ITEMS

by

Nikita Ivkin

A dissertation submitted to Johns Hopkins University
in conformity with the requirements for the degree of
Doctor of Philosophy

Baltimore, Maryland
October, 2018

Abstract

Streaming model supplies solutions for handling enormous data flows for over 20 years now. The model works with sequential data access and states sublinear memory as its primary restriction. Although the majority of the algorithms are randomized and approximate, the field facilitates numerous applications from handling networking traffic to analyzing cosmology simulations and beyond. This thesis focuses on one of the most foundational and well-studied problems of finding heavy hitters, i.e. frequent items:

1. We challenge the long-lasting complexity gap in finding heavy hitters with ℓ_2 guarantee in the insertion-only stream and present the first optimal algorithm with a space complexity of $O(1)$ words and $O(1)$ update time. Our result improves on Count Sketch algorithm with space and time complexity of $O(\log n)$ by Charikar et al. 2002 [39].

2. We consider the ℓ_2-heavy hitter problem in the interval query settings, rapidly emerging in the field. Compared to well known sliding window model where an algorithm is required to report the function of interest computed over the last N updates, interval query provides query flexibility, such that at any moment t one can query the function value on any interval $(t_1, t_2) \subseteq (t - N, t)$. We present the first ℓ_2-heavy hitter algorithm in that model and extend the result to estimation all streamable functions of a frequency vector.

3. We provide the experimental study for the recent space optimal result on streaming quantiles by Karnin et al. 2016 [85]. The problem can be considered as a generalization to the heavy hitters. Additionally, we suggest several variations to the algorithms which improve the running time from $O(1/\varepsilon)$ to $O(\log 1/\varepsilon)$, provide twice better space vs. precision trade-off, and extend the algorithm for the case of weighted updates.

4. We establish the connection between finding "halos", i.e. dense areas, in cosmology N-body simulation and finding heavy hitters. We build the first halo finder and scale it up to handle datasets with up-to 10^{12} particles via GPU boosting, sampling and parallel I/O. We investigate its behavior and compare it to traditional in-memory halo finders. Our solution pushes the memory footprint from several terabytes down to less than a gigabyte, therefore, make the problem feasible for small servers and even desktops.

Primary reader: Vladimir Braverman, Alexander Szalay

Secondary reader: Raman Arora

Acknowledgements

I would like to thank my advisor Vladimir Braverman for all the support and guidance during the entire Ph.D. program. I especially appreciate his effort in introducing me to other researchers in the field and encouraging me to find new collaborations on my own. I have gained a lot in the skill of controlling my time more efficiently and manage it mindfully, while going through several teaching assistantships during my Ph.D. I would like to express my gratitude to all the researchers I had a chance to collaborate with: Tamas Budavari, Mohammad Hajiaghayi, Michael Jacobs, Zohar Karnin, Kevin Lang, Edo Liberty, Gerard Lemson, Morteza Monemizadeh, Muthu Muthukrishnan, Jelani Nelson, Mark Neyrinck, Alex Szalay, David Woodruff, Sepehr Assadi, Stephen Chestnut, Hossein Esfandiari, Ruoyuan Gao, Srini Suresh Kumar, Zaoxing Liu, Teodor Marinov, Poorya Mianjy, Jalaj Upadhyay, Xin Wang, Lin Yang, Zhengyu Wang. I owe a very important debt to all the professors whose classes I had a chance to participate in and to learn a lot from them. Each class was a big excitement for me thanks to Yanif Ahmad, Raman Arora, Amitabh Basu, Vladimir Braverman, Michael Dinitz, Jim Fill and Rene Vidal. I would like to offer my special thanks to Deborah DeFord, Cathy Thornton, Zachary Burwell, Laura Graham, Tonette Harris, Shani McPherson, Joanne Selinski, and Javonnia Thomas for all the help from administrative side of the department. I am deeply grateful to all my family and all my friends who were there for me when I needed it the most. Special thanks to all the readers of my thesis Raman Arora, Vladimir Braverman and Alex Szalay.

Finally, I would like to acknowledge that my work was financially supported by the following grants and awards: NSA-H98230-13-C-0265, NSF IIS -1447639, DARPA -111477, Research Trends Det. HR0011-16-P-0014 NSF EAGER -16050041, ONR N00014-18-1-2364, CISCO 90073352.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The rates at which data is generated are growing tremendously. Among examples: daily life (social media, fitness tracking, delivery vehicles tracking), infrastructure (networking telemetry, cloud services), science (bioinformatics, astrophysics), etc. It is pushing modern technology to deal with a large pool of new problems where data is too large to be stored due to either storage absence, or storage hardware being too slow. Network switch cannot store or keep in the memory trillions of packets going through it daily, Twitter needs to mine the hot topics over the window of last 15 minutes, which cannot be done offline, cosmology simulations are too big to fit into the memory of a regular size server. These are just a couple of examples where the analysis cannot be done either offline or in-memory, and data stream model naturally fits them and the numerous of other problems. But the model is very restrictive and it was proven that many problems cannot be solved in it. We describe the model in the following section.

1.1 Streaming model

Streaming model gained its fame after the seminal paper on estimating ℓ_p norms by Alon, Matias, and Szegedy [6]. Authors consider the stream of updates $S = \{s_1, \ldots, s_m\}$, where

s_i can represent any object with respect to the problem to be solved: numbers, graph edges, vectors in high dimensional space, etc. For simplicity we will set s_i to be an integer: $s_i \in [n] = \{1, \ldots, n\}$. Algorithm reads the data items s_i one at a time, it is allowed to make only a single pass over the dataset, and at the end of the stream it should return the value of the function of interest $f(S)$. The algorithm is expected to optimize the time complexity of an update and a final query, however minimizing the space complexity is at the first priority. Streaming model requires algorithm's memory footprint to grow sublinearly to the size of the dataset. Under these strict limitations the majority of the problems were proved not to have the solutions, but the compromise can be achieved, if one allows the algorithm, first of all, to return only an approximation rather than the exact value, and second of all, to be randomized, i.e. to be allowed to fail with some small probability.

In stated settings, streaming model found applications in a variety of fields: networking [14, 15], bioinformatics [125], machine learning [60], astrophysics [102], security [127], databases [129, 123], sensor networks [98], finance [38], optimization [5], etc. The motivation for each requirement of the model might seem to be too strict and unnatural, therefore we explain further several relevant examples to see the roots and the origin:

1. Network switch processes up to a billion packets a day, but the memory available for monitoring purposes can be as little as one hundred megabytes. At the same time storing packets cannot be done at the rate of the flow. Widely spread denial of service (DDoS) attack can be detected due to unusually high packet flow to the server under attack. Without knowing the target of the attack switch have to maintain the flow size to each of them, which is impossible due to the high number of servers and low memory allowance on the switch. However, in this problem, the exact values for the flows are not critically important, and an approximation can be used instead.

2. To perform distributed computation efficiently, one needs to balance out the load among the computing machines (workers). Given the internal distribution of the data, the problem becomes trivial. However, finding the distribution is challenging and often done on a single machine in the dynamic settings, i.e. tasks coming one by one to the controlling node and it should distribute the flow of tasks among the computing nodes (workers). As in the previous example, data cannot be stored on the controlling node, however exact balance is not required, therefore just an approximate distribution is enough.

3. The analysis of the cosmological N-body simulations requires finding halos (dense areas) in the dataset where 90% of particles are treated as noise. Simulations operate with more than billion particles and this number is growing, while all conventional approaches require to load entire dataset into a memory. Researchers without access to the machines with terabytes of memory cannot work with the dataset. Alternatively, streaming based solution let task to be performed on small servers with large storage but low memory. This example depicts the existing gap between the cost of storage and memory.

Due to the variety of applications streaming model was considered in different settings. The definition given earlier corresponds to the cash-register model, where items can only be "added", also known as an additions-only model. Alternatively, one can allow "deletions" in the stream, i.e. stream can be represented as a sequence of pairs (s_j, δ_j) with $\delta_j \in \{-1, +1\}$. Then, at the end of the stream, frequency of each item f_i can be represented as $f_i = \sum_{j:s_j=i} \delta_j$. This model is often called turnstile. For example, the result mentioned earlier [6] provide $(1 \pm \varepsilon)$ approximation of $\|f\|_2$ using only $O(\frac{1}{\varepsilon^2} \log^2 n)$ bits of space in turnstile model, where f is a vector defined by f_i's. Certain applications need

to work with infinite streams, however, the target function should be evaluated only on the most recent items. Given an infinite stream $S = \{s_1, \ldots, s_t, \ldots\}$ at any moment t report $f(\{s_{t-N}, \ldots, s_t\})$, where N is large, such that window $\{s_{t-N}, \ldots, s_t\}$ cannot be stored explicitly. Note, this model, called sliding windows, differs a lot from the turnstile one, as in the latter an algorithm is always aware which item to "delete", while in the former an algorithm needs at each moment t delete item s_{t-N} which might be not stored explicitly anywhere. For instance, Twitter shows the most discussed topics over the last hour, rather than the entire time. In the current chapter we only aim to give a brief glance at the model and the reader is encouraged to read more about the topic in [113, 3].

1.2 Contribution

1.2.1 ℓ_2 heavy hitters algorithm with fewer words

Given a stream of updates $S : s_1, s_2, \ldots s_m$ with $s_j \in \{1, \ldots, n\}$ we consider the problem of finding all (ε, ℓ_p) heavy hitters in the cash-register model for $p = 2$. Formally, item i is an (ε, ℓ_p) heavy hitter in the stream S if $f_i > \varepsilon \|f\|_p$, where f_i is the number of updates with the id i: $f_i = |\{j|\ s_j = i\}|$ and f is a frequency vector, i.e. i-th component of this vector equals f_i. Note, that ℓ_2 guarantee is significantly stronger than ℓ_1. In 1996, the seminal paper by Alon, Matias, and Szegedy [6] among other breakthrough results introduced the first algorithm for estimating $\|f\|_2$ in the streaming model. Later in 2002, Charikar, Chen, and Farach-Colton [39] presented Count Sketch, the algorithm which used $\|f\|_2$ estimation procedure from [6] as a subroutine to find all (ε, ℓ_2) heavy hitters. Count Sketch works in turnstile model and requires $O(\log n)$ words of memory. It is known that Count Sketch algorithm is tight in memory requirement for turnstile model [82, 10]. However, no better solution was proposed for cash-register model and the only

known lower bound is the naive one requiring $\Omega(1)$ words, coming from the need to store the identity of the heavy hitter. In Chapter 2, we present the algorithm CountSieve capable of finding all ℓ_2 heavy hitters in cash-register model with the space complexity of $O(\log \log n)$ words; then we improve it further by presenting BPTree algorithm, working with only $O(1)$ words of space. Both algorithms take advantage of Dudley-like chaining argument applied to ℓ_2 norm estimator.

1.2.2 Streaming Algorithms for cosmological N-body simulations

Cosmological N-body simulations are essential for studies of the large-scale distribution of matter and galaxies in the Universe. This analysis often involves finding clusters of particles and retrieving their properties. Detecting such "halos" among a very large set of particles is a computationally intensive problem, usually executed on the same supercomputers that produced the simulations, requiring terabytes of memory. Working with the simulation output in streaming settings can potentially make the problem of finding halos and other postprocessing analytical tasks feasible for the smaller machines or even desktops. In Chapter 5 we present a novel connection between the N-body simulations and the streaming algorithms. In particular, we investigate a link between halo finders and the problem of finding heavy hitters in a data stream, that should greatly reduce the computational resource requirements, especially the memory needs. Based on this connection, we can build a new halo finder by running efficient heavy hitter algorithms as a black-box. We implement two representatives of the family of heavy hitter algorithms, the Count-Sketch algorithm (CS) and the Pick-and-Drop sampling (PD), and evaluate their accuracy and memory usage. Comparison with other halo-finding algorithms from [92]

shows that our halo finder can locate the largest halos using significantly smaller memory space and with comparable running time. This streaming approach makes it possible to run and analyze extremely large data sets from N-body simulations on a smaller machine, rather than on supercomputers. Our findings demonstrate the connection between the halo search problem and streaming algorithms which we further investigate to scale it from proof-of-concept level experiments with relatively small datasets to larger state-of-the-art datasets. We present a robust streaming tool that leverages GPU boosting, sampling, and parallel I/O, to significantly improve performance and scalability. Our rigorous analysis of the sketch parameters improves the initial results from finding the centers of the 10^3 largest halos to $10^4 - 10^5$, and reveals the trade-offs between memory, running time and the number of halos. Our experiments show that our tool can scale to datasets with up to 10^{12} particles while using less than an hour of running time on a single GPU Nvidia GTX 1080.

1.2.3 Monitoring the Network with Interval Queries

Modern network telemetry systems collect and analyze massive amounts of raw data in space efficient manner, taking advantage streaming algorithms. Many statistics can be efficiently computed on the sliding window, providing crucial information only on the recent updates, however, many analytical tasks require more advanced capabilities such as drill down queries that allow iterative refinement of the search space, i.e. at any time moment t the data structure is required to report statistics over given time interval (t_1, t_2). Recently, in [14] new model was introduced to accomodate this need, [14] also presented efficient sketching algorithm for finding ℓ_1 heavy hitters in that model. We will refer to it as an interval query model. In Chapter 3, we present the first algorithm to find ℓ_2 heavy hitters in interval query model. Using technique of recursive sketching, we further

generalize our result to a much wider class of functions of the frequency vector, including entropy estimation, count distinct, etc. We implement the algorithm and estimate its performance on network switches datasets from CAIDA [137].

1.2.4 Streaming quantiles algorithms with small space and update time

Approximating quantiles and distributions over streaming data has been studied for roughly two decades now [2, 85, 104, 105, 143, 112]. Problem require data structure to input a multiset $S = \{s_i\}_{i=1}^{n}$, and upon any query ϕ return ϕn-th item of the sorted S. Recently, Karnin, Lang, and Liberty [85] proposed the first asymptotically optimal algorithm for doing so. Chapter 4 complements their theoretical result by providing improved variants of their algorithm. It improves accuracy/space tradeoff by provably decreasing the upper bound by almost twice, which was also verified experimentally. Our techniques exponentially reduce the worst case update time from $O(1/\varepsilon)$ down to $O(\log(1/\varepsilon))$. Also, we suggest two algorithms for a weighted stream of updates (a_i, w_i), with the worst case update times $O(\log^2(1/\varepsilon))$ and $O(\log(1/\varepsilon))$ correspondingly, which is a significant improvement over the naive extensions that require $O((\max w_i)\log 1/\varepsilon)$ update time.

Chapter 2

ℓ_2 heavy hitters algorithm with fewer words

This chapter is based on [31] and [32].

2.1 Introduction

As emphasized in Chapter 1 there are numerous applications of data streams, and the elements of the stream p_i may be numbers, points, edges in a graph, and so on. Examples include internet search logs, network traffic, sensor networks, and scientific data streams (such as in astronomy, genomics, physical simulations, etc.). The sheer size of the dataset often imposes very stringent requirements on an algorithm's resources. Moreover, in many cases only a single pass over the data is feasible, such as in network applications, since if the data on a network is not physically stored somewhere, it may be impossible to make a second pass over it. There are multiple surveys and tutorials in the algorithms, database, and networking communities on the recent activity in this area; we refer the reader to [113, 11] for more details and motivations underlying this area.

Within the study of streaming algorithms, the problem of finding frequent items is one of the most well-studied and core problems, with work on the problem beginning in 1981 [22, 23]. It has applications in flow identification at IP routers [53], iceberg queries [55], iceberg datacubes [20, 69], association rules, and frequent itemsets [4, 126, 142, 75, 68]. Aside from being an interesting problem in its own right, algorithms for finding frequent items are used as subroutines to solve many other streaming problems, such as moment estimation [80], entropy estimation [36, 72], ℓ_p-sampling [110], finding duplicates [62], and several others.

Formally, we are given a stream $p_1, \ldots, p_m$ of items from a universe $\mathcal{U}$, which, without loss of generality we identify with the set $\{1, 2, \ldots, n\}$. We make the common assumption that $\log m = O(\log n)$, though our results generalize naturally to any m and n. Let f_i denote the frequency, that is, the number of occurrences, of item i. We would like to find those items i for which f_i is large, i.e., the "heavy hitters".

Stated simply, the goal is to report a list of items that appear least τ times, for a given threshold τ. Naturally, the threshold τ should be chosen to depend on some measure of the size of the stream. The point of a frequent items algorithm is to highlight a *small* set of items that are frequency outliers. A choice of τ that is independent of f misses the point; it might be that all frequencies are larger than τ.

With this in mind, previous work has parameterized τ in terms of different norms of f with MisraGries [109] and CountSketch [39] being two of the most influential examples. A value $\varepsilon > 0$ is chosen, typically ε is a small constant independent of n or m, and τ is set to be $\varepsilon\|f\|_1 = \varepsilon m$ or $\varepsilon\|f\|_2$.

These are called the ℓ_1 and ℓ_2 guarantees, respectively. Choosing the threshold τ in this manner immediately limits the focus to outliers since no more than $1/\varepsilon$ items can have frequency larger than $\varepsilon\|f\|_1$ and no more than $1/\varepsilon^2$ can have frequency $\varepsilon\|f\|_2$ or

larger.

A moments thought will lead one to conclude that the ℓ_2 guarantee is stronger, i.e. harder to achieve, than the ℓ_1 guarantee because $\|x\|_1 \geq \|x\|_2$, for all $x \in \mathbb{R}^n$. Indeed, the ℓ_2 guarantee is much stronger. Consider a stream with all frequencies equal to 1 except one which is equal j. With $\varepsilon = 1/3$, achieving the ℓ_1 guarantee only requires finding an item with frequency $j = n/2$, which means that it occupies more than one-third of the positions in the stream, whereas achieving the ℓ_2 guarantee would require finding an item with frequency $j = \sqrt{n}$, such an item is a negligible fraction of the stream!

As we discuss further, the algorithms achieving ℓ_2 guarantee, like CountSketch [39], achieve essentially the best space-to-τ trade-off. But, since the discovery of CountSketch, which uses $O(\varepsilon^{-2} \log n)$ words of memory, it has been an open problem to determine the smallest space possible for achieving the ℓ_2 guarantee. Since the output is a list of up to ε^{-2} integers in $[n]$, $\Omega(\varepsilon^{-2})$ words of memory are necessary.

Work on the heavy hitters problem began in 1981 with the MJRTY algorithm of [22, 23], which is an algorithm using only two machine words of memory that could identify an item whose frequency was strictly more than half the stream. This result was generalized by the MisraGries algorithm in [109], which, for any $0 < \varepsilon \leq 1/2$, uses $2(\lceil 1/\varepsilon \rceil - 1)$ counters to identify every item that occurs strictly more than an εm times in the stream. This data structure was rediscovered at least two times afterward [48, 86] and became also known as the Frequent algorithm. It has implementations that use $O(1/\varepsilon)$ words of memory, $O(1)$ expected update time to process a stream item (using hashing), and $O(1/\varepsilon)$ query time to report all the frequent items. Similar space requirements and running times for finding ε-frequent items were later achieved by the SpaceSaving [107] and LossyCounting [103] algorithms.

A later analysis of these algorithms in [19] showed that they not only identify the

heavy hitters, but they also provided estimates of the frequencies of the heavy hitters. Specifically, when using $O(k/\varepsilon)$ counters they provide, for each heavy hitter $i \in [n]$, an estimate $\tilde{f}_i$ of the frequency f_i such that $|\tilde{f}_i - f_i| \leq (\varepsilon/k) \cdot \|f_{tail(k)}\|_1 \leq (\varepsilon/k)\|f\|_1$. Here $f_{tail(k)}$ is the vector f but in which the largest k entries have been replaced by zeros (and thus the norm of $f_{tail(k)}$ can never be larger than that of f). We call this the $((\varepsilon/k), k)$-*tail guarantee*. A recent work of [21] shows that for $0 < \alpha < \varepsilon \leq 1/2$, all ε-heavy hitters can be found together with approximate for them $\tilde{f}_i$ such that $|\tilde{f}_i - f_i| \leq \alpha\|f\|_1$, and the space complexity is $O(\alpha^{-1}\log(1/\varepsilon) + \varepsilon^{-1}\log n + \log\log\|f\|_1)$ bits.

All of the algorithms in the previous paragraph work in one pass over the data in the *insertion-only* model, also known as the *cash-register* model [113], where deletions from the stream are not allowed. Subsequently, many algorithms have been discovered that work in more general models such as the strict turnstile and general turnstile models. In the turnstile model, the vector $f \in \mathbb{R}^n$ receives updates of the form (i, Δ), which triggers the change $f_i \leftarrow f_i + \Delta$; note that we recover the insertion-only model by setting $\Delta = 1$ for every update. The value Δ is assumed to be some bounded precision integer fitting in a machine word, which can be either positive or negative. In the *strict* turnstile model we are given the promise that $f_i \geq 0$ at all times in the stream. That is, items cannot be deleted if they were never inserted in the first place. In the general turnstile model no such restriction is promised (i.e. entries in f are allowed to be negative). This can be useful when tracking differences or changes across streams. For example, if f^1 is the query stream vector with f^1_i being the number of times word i was queried to a search engine yesterday, and f^2 is the similar vector corresponding to today, then finding heavy coordinates in the vector $f = f^1 - f^2$, which corresponds to a sequence of updates with $\Delta = +1$ (from yesterday) followed by updates with $\Delta = -1$ (from today), can be used to track changes in the queries over the past day.

In the general turnstile model, an ε-heavy hitter in the ℓ_p norm is defined as an index $i \in [n]$ such that $|f_i| \geq \varepsilon \|f\|_p$. Recall $\|f\|_p$ is defined as $(\sum_{i=1}^{n} |f_i|^p)^{1/p}$. The CountMin sketch treats the case of $p = 1$ and uses $O(\varepsilon^{-1} \log n)$ memory to find all ε-heavy hitters and achieve the $(\varepsilon, 1/\varepsilon)$-tail guarantee [44]. The CountSketch treats the case of $p = 2$ and uses $O(\varepsilon^{-2} \log n)$ memory, achieving the $(\varepsilon, 1/\varepsilon^2)$-tail guarantee. It was later showed in [82] that the CountSketch actually solves ℓ_p-heavy hitters for all $0 < p \leq 2$ using $O(\varepsilon^{-p} \log n)$ memory and achieving the $(\varepsilon, 1/\varepsilon^p)$-tail guarantee. In fact, they showed something stronger: that *any* ℓ_2 heavy hitters algorithm with error parameter $\varepsilon^{p/2}$ achieving the tail guarantee automatically solves the ℓ_p heavy hitters problem with error parameter ε for any $p \in (0, 2]$. In this sense, solving the heavy hitters for $p = 2$ with tail error, as CountSketch does, provides the strongest guarantee among all $p \in (0, 2]$.

Identifying ℓ_2 heavy hitters is optimal in another sense, too. When $p > 2$ by Hölder's Inequality $\varepsilon \|f\|_p \geq \frac{\varepsilon}{n^{1/2-1/p}} \|f\|_2$. Hence, one can use an ℓ_2 heavy hitters algorithm to identify items with frequency at least $\varepsilon \|f\|_p$, for $p > 2$, by setting the heaviness parameter of the ℓ_2 algorithm to $\varepsilon / n^{1/2-1/p}$. The space needed to find ℓ_p heavy hitters with a CountSketch is therefore $O(\varepsilon^{-2} n^{1-2/p} \log n)$ which is known to be optimal [97]. We conclude that the ℓ_2 guarantee leads to the best space-to-frequency-threshold ratio among all $p > 0$.

It is worth pointing out that both the CountMin sketch and CountSketch are randomized algorithms, and with small probability $1/n^c$ (for a user-specified constant $c > 0$), they can fail to achieve their stated guarantees. The work [82] also showed that the CountSketch algorithm is optimal: they showed that *any* algorithm, even in the strict turnstile model, solving ℓ_p heavy hitters even with $1/3$ failure probability must use $\Omega(\varepsilon^{-p} \log n)$ memory.

Of note is that the MisraGries and other algorithms in the insertion-only model solve ℓ_1 heavy hitters using (optimal) $O(1/\varepsilon)$ memory, whereas the CountMin and CountSketch

algorithms use a larger $\Theta(\varepsilon^{-1} \log n)$ memory in the strict turnstile model, which is optimal in that model. Thus there is a gap of $\log n$ between the space complexities of ℓ_1 heavy hitters in the insertion-only and strict turnstile models.

In Section 2.2 we present an algorithm that solves ℓ_2 heavy hitters problem in the insertion-only model using only $O(\frac{1}{\varepsilon^2} \log \frac{1}{\varepsilon} \log \log n)$ words of memory, later in Section 2.3 we further improve the memory usage and present an algorithm using only $O(\frac{1}{\varepsilon^2} \log \frac{1}{\varepsilon})$.

2.2 Beating CountSketch for Heavy Hitters in Insertion Streams

This section is based on [31], work done in collaboration with Braverman V., Chestnut S. and Woodruff D.

2.2.1 Introduction

Our contribution. The main result of this section is the near resolution of the open question above.

Theorem 1 (ℓ_2-Heavy Hitters)**.** *For any $\varepsilon > 0$, there is a 1-pass algorithm in the insertion-only model that, with probability at least $2/3$, finds all those indices $i \in [n]$ for which $f_i \geq \varepsilon\sqrt{F_2}$, and reports no indices $i \in [n]$ for which $f_i \leq \frac{\varepsilon}{2}\sqrt{F_2}$. The space complexity is $O(\frac{1}{\varepsilon^2} \log \frac{1}{\varepsilon} \log n \log \log n)$ bits.*

The intuition of the proof is as follows. Suppose there is a single ℓ_2-heavy hitter H, $\varepsilon > 0$ is a constant, and we are trying to find the identity of H. Suppose further we could identify a substream S' where H is very heavy, specifically we want that the frequencies in the substream satisfy $\frac{f_H^2}{\text{poly}(\log n)} \geq \sum_{j \in S', j \neq H} f_j^2$. Suppose also that we could find certain

$R = O(\log n)$ "breakpoints" in the stream corresponding to jumps in the value of f_H, that is, we knew a sequence $p_{q_1} < p_{q_2} < \cdots < p_{q_R}$ which corresponds to positions in the stream for which f_H increases by a multiplicative factor of $(1 + 1/\Theta(R))$.

Given all of these assumptions, in between breakpoints we can partition the universe randomly into two pieces and run an F_2-estimate [6] (AMS sketch) on each piece. Since f_H^2 is more than a $\text{poly}(\log n)$ factor times $\sum_{j \in S', j \neq H} f_j^2$, while in between each breakpoint the squared frequency of H is $\Omega\left(\frac{f_H^2}{\log n}\right)$, it follows that H contributes a constant fraction of the F_2-value in between consecutive breakpoints, and so, upon choosing the constants appropriately, the larger in magnitude of the two AMS sketches will identify a bit of information about H, with probability say 90%. This is our algorithm Sieve. Since we have $\Theta(\log n)$ breakpoints, in total we will learn all $\log n$ bits of information needed to identify H. One view of this algorithm is that it is a sequential implementation of the multiple repetitions of CountSketch, namely, we split the stream at the breakpoints and perform one "repetition" on each piece while discarding all but the single bit of information we learn about H in between breakpoints.

However, it is not at all clear how to (1) identify S' and (2) find the breakpoints. For this, we resort to the theory of Gaussian and Bernoulli processes. Throughout the stream we can maintain a sum of the form $X_t = \sum_{i=1}^n f_i^{(t)} Z_i$, where the Z_i are independent $\text{Normal}(0, 1)$ or Rademacher random variables. Either distribution is workable. One might think as one walks through a stream of length $\text{poly}(n)$, there will be times for which this sum is much larger than $\sqrt{F_2}$; indeed, the latter is the standard deviation and a naïve union bound, if tight, would imply positions in the stream for which $|X_t|$ is as large as $\sqrt{F_2 \log n}$. It turns out that this cannot happen! Using a generic chaining bound developed by Fernique and Talagrand [135], we can prove that there exists a universal constant

C' such that

$$\mathbb{E} \sup_t |X_t| \leq C' \sqrt{F_2}.$$

We call this the Chaining Inequality.

We now randomly partition the universe into $O(\frac{1}{\varepsilon^2})$ "parts", and run our algorithm independently on each part. This ensures that, for a large constant C, H is C-heavy, meaning, $f_H^2 \geq C(F_2 - f_H^2)$, where here we abuse notation and use F_2 to denote the moment of the part containing H. We run the following two-stage algorithm independently on each part. The first stage, called Amplifier, consists of $L = O(\log \log n)$ independent and concurrent repetitions of the following: randomly split the set of items into two buckets and maintain two Bernoulli processes, one for the updates in each bucket. By the Chaining Inequality, a Markov bound, and a union bound, the total F_2 contribution, excluding that of H, in each piece in each repetition *at all times in the stream*, will be $O(\sqrt{F_2 - f_H^2})$. Since H is sufficiently heavy, this means after some time t^*, its piece will be larger in magnitude in most, say 90%, of the L repetitions. Furthermore, H will be among only $n/2^{\Omega(L)} = n/\text{poly} \log n$ items with this property. At this point, we can restrict our attention to a substream containing only those items.

The substream has the property that its F_2 value, not counting H, will be a factor $\frac{1}{\log^2 n}$ times the F_2 value of the original stream, making H $\Omega(\log^2 n)$-heavy. Finally, to find the breakpoints, our algorithm Timer maintains a Bernoulli process on the substream, and every time the Bernoulli sum increases by a multiplicative $\left(1 + \frac{1}{\theta(R)}\right)$ factor, creates a new breakpoint. By the Chaining Inequality applied in each consecutive interval of breakpoints, the F_2 of all items other than H in the interval is at most $O(\log n)$ larger than its expectation; while the squared frequency of H on the interval is at least $\frac{f_H^2}{\log n}$. Since H is $\Omega(\log^2 n)$-heavy, this makes f_H^2 to be the dominant fraction of F_2 on the interval.

One issue with the techniques above is they assume a large number of random bits

can be stored. A standard way of derandomizing this, due to Indyk [78] and based on Nisan's pseudorandom generator PRG [116], would increase the space complexity by a $\log n$ factor, which is exactly what we are trying to avoid. Besides, it is not clear we can even apply Indyk's method since our algorithm decides at certain points in the stream to create new F_2-sketches based on the past, whereas Indyk's derandomization relies on maintaining a certain type of linear sum in the stream, so that reordering of the stream does not change the output distribution. A first observation is that the only places we need more than limited independence are in maintaining a collection of $O(\log n)$ hash functions and the stochastic process $\sum_{i=1}^{n} f_i Z_i$ throughout the stream. The former can, in fact, be derandomized along the lines of Indyk's method [78].

In order to reduce the randomness needed for the stochastic process, we use a Johnson-Lindenstrauss transformation to reduce the number of Rademacher (or Gaussian) random variables needed. The idea is to reduce the frequency vector to $O(\log n)$ dimensions with JL and run the Bernoulli process in this smaller dimensional space. The Bernoulli process becomes $\sum_{i=1}^{O(\log n)} Z_i (Tf)_i$, where T is the JL matrix. The same technique is used by Meka for approximating the supremum of a Gaussian process [106]. It works because the Euclidean length of the frequency vector describes the variance and covariances of the process, hence the transformed process has roughly the same covariance structure as the original process. An alternative perspective on this approach is that we use the JL transformation in reverse, as a pseudorandom generator that expands $O(\log n)$ random bits into $O(n)$ random variables which fool our algorithm using the Bernoulli process.

In Section 2.2.5 we also use our techniques to prove the following.

Theorem 2 (F_2 at all points)**.** *For any $\varepsilon > 0$, there is a 1-pass algorithm in the insertion-only model that, with probability at least $2/3$, outputs a $(1 \pm \varepsilon)$-approximation of F_2 at all points in the stream. It uses $O(\frac{1}{\varepsilon^2} \log n (\log \frac{1}{\varepsilon} + \log\log n))$ bits of space.*

L	amplifier size	$O(\log\log n)$
τ	round expansion	$100(R+1)$
δ	small constant	$\Omega(1)$
$S^{t_1:t_2}$	interval of the stream	$(p_{t_1+1}, \ldots, p_{t_2})$
H	heavy hitter id	$\in [n]$
e_j	jth unit vector	
T	JL transformation	$\in \mathbb{R}^{k\times n}$
$f_H^{(k)}$	frequency on $S^{0:k}$	
m	stream length	$\mathrm{poly}(n)$
$f^{(k_1:k_2)}$	frequency on $S^{k_1:k_2}$	$f^{(k_2)} - f^{(k_1)}$
n	domain size	
R	# of Sieve rounds	$O(\log n)$
k	JL dimension	$O(\log n)$
C'	Chaining Ineq. const.	$O(1)$
d	dim. of Bern. proc.	$O(\log \delta^{-1})$
C	large const.	$\geq d^{\frac{3}{2}} C'/\delta$

TABLE 2.1: Notation and parameters used throughout the current section.

Preliminaries. Given a stream $S = (p_1, p_2, \ldots, p_m)$, with $p_i \in [n]$ for all i, we define the frequency vector at time $0 \leq t \leq m$ to be the vector $f^{(t)}$ with coordinates $f_j^{(t)} := \#\{t' \leq t \mid p_{t'} = j\}$. When $t = m$ we simply write $f := f^{(m)}$. Given two times $t_1 \leq t_2$ we use $f^{(t_1:t_2)}$ for the vector $f^{(t_2)} - f^{(t_1)}$. Notice that all of these vectors are nonnegative because S has no deletions. An item $H \in [n]$ is said to be an α-heavy hitter, for $\alpha > 0$, if $f_H^2 \geq \alpha \sum_{j\neq H} f_j^2$. The goal of our main algorithm, CountSieve, is to identify a single α-heavy hitter for α a large constant. We will assume $\log m = O(\log n)$, although our methods apply even when this is not true. It will be occasionally helpful to assume that n is sufficiently large. This is without loss of generality since in the case $n = O(1)$ the problem can be solved exactly in $O(\log m)$ bits.

A streaming algorithm is allowed to read one item at a time from the stream in the order given. The algorithm is also given access to a stream of random bits, it must pay

to store any bits that it accesses more than once, and it is only required to be correct with constant probability strictly greater than $1/2$. Note that by repeating such an algorithm k times and taking a majority vote, one can improve the success probability to $1 - 2^{-\Omega(k)}$. We measure the storage used by the algorithm on the worst case stream, i.e. worst case item frequencies and order, with the worst case outcome of its random bits.

The AMS sketch [6] is a linear sketch for estimating F_2. The sketch contains $O(\varepsilon^{-2} \log \delta^{-1})$ independent sums of the form $\sum_{j=1}^{n} S_j f_j$, where $S_1, S_2, \ldots, S_n$ are four-wise independent Rademacher random variables. By averaging and taking medians it achieves a $(1 \pm \varepsilon)$-approximation to F_2 with probability at least $(1 - \delta)$.

A *Gaussian process* is a stochastic process $(X_t)_{t \in T}$ such that every finite subcollection $(X_t)_{t \in T'}$, for $T' \subseteq T$, has a multivariate Gaussian distribution. When T is finite (as in this section), every Gaussian process can be expressed as a linear transformation of a multivariate Gaussian vector with mean 0 and covariance I. Similarly, a *Bernoulli process* $(X_t)_{t \in T}$, T finite, is a stochastic process defined as a linear transformation of a vector of i.i.d. Rademacher (i.e. uniform ± 1) random variables. Underpinning our results is an analysis of the Gaussian process

$$X_t = \sum_{j \in [n]} Z_j f_j^{(t)}, \text{ for } t = 0, \ldots, m,$$

where $Z_1, \ldots, Z_n \overset{\text{iid}}{\sim} \mathcal{N}(0, 1)$ are independent standard Normal random variables. The Bernoulli analogue to our Gaussian process replaces the distribution of the random vector Z as $Z_1, \ldots, Z_n \overset{\text{iid}}{\sim}$ Rademacher. Properties of the Normal distribution make it easier for us to analyze the Gaussian process rather than its Bernoulli cousin. On the other hand, we find Bernoulli processes more desirable for computational tasks. Existing tools, which we discuss further in Section 2.2.3 and Section 2.2.4, allow us to transfer the needed

properties of a Gaussian process to its Bernoulli analogue.

A $k \times n$ matrix T is a $(1 \pm \gamma)$-*embedding* of a set of vectors $X \subseteq \mathbb{R}^n$ if

$$(1-\gamma)\|x-y\|_2 \leq \|Tx-Ty\|_2 \leq (1+\gamma)\|x-y\|_2,$$

for all $x, y \in X \cup \{0\}$. We also call such a linear transformation a *JL Transformation*. It is well-known that taking the entries of the matrix T to be i.i.d. Normal random variables with mean 0 and variance $1/k$ produces a JL transformation with high probability. Many other randomized and deterministic constructions exist, we will use the recent construction of Kane, Meka, and Nelson [84].

The development and analysis of our algorithm relies on several parameters, some of which have already been introduced. Table 2.1 lists those along with the rest of the parameters and some other notation for reference. In particular, the values C, d, δ, and γ are constants that we will choose in order to satisfy several inequalities. We will choose δ and γ to be small, say $1/200$, and $d = O(\log 1/\delta)$. C and C' are sufficiently large constants, in particular $C \geq dC'/\delta$.

2.2.2 ℓ_2 heavy hitters algorithm

This section describes the algorithm CountSieve, which solves the heavy hitter problem for the case of a single heavy hitter, i.e. top-1, in $O(\log n \log\log n)$ bits of space and proves Theorem 1. By definition, the number of ε-heavy hitters is at most $1 + 1/\varepsilon^2$, so, upon hashing the universe into $O(1/\varepsilon^2)$ parts, the problem of finding all ε-heavy hitters reduces to finding a single heavy hitter in each part. Collisions can be easily handled by repeating the algorithm $O(\log 1/\varepsilon)$ times. When $\varepsilon = \Omega(1)$, using this reduction incurs only a constant factor increase in space over the single heavy hitter problem.

Suppose the stream has only a single heavy hitter $H \in [n]$. Sequentially, over the course of reading the stream, CountSieve will hash the stream into two separate substreams for $O(\log n)$ repetitions, and in each repetition it will try to determine which of the two substreams has the heavy hitter using the AMS Sketch. With high probability, H has a unique sequence of hashes, so if we correctly identify the stream containing H every time then we can correctly identify H. This holds even if we only correctly identify the stream containing H a large constant fraction of the repetitions. CountSketch accomplishes this by performing the $O(\log n)$ rounds of hashing in parallel, with $\Omega(\log^2 n)$ bits of storage. One of our innovations is to implement this scheme *sequentially* by specifying intervals of updates, which we call *rounds*, during each of which we run the two AMS Sketches. In total there could be as many as $\Theta(\log^2 n)$ of these rounds, but we will discard all except the last $R = O(\log n)$ of them.

Algorithm 1 is a simplified version of the Bernoulli process used by CountSieve. It has all of the properties we need for correctness of the algorithm, but it requires too many random bits. Chief among these properties is the control on the supremum of the process. The Chaining Inequality gives us a uniform bound on the maximum value of the BP

Algorithm 1 One Bernoulli process.

procedure BP(Stream S)
  Sample $Z_1, \ldots, Z_n \overset{\text{iid}}{\sim}$ Rademacher
  return $\langle Z, f^{(t)} \rangle$ at each time t
end procedure

process in terms of the standard deviation of the last value. This property is formalized by the definition of a tame process.

Definition 3. *Let $f^{(t)} \in \mathbb{R}^n$, for $t \in [m]$, and let $T : \mathbb{R}^n \to \mathbb{R}^k$ be a matrix. Let Z be a $d \times k$ matrix of i.i.d. Rademacher random variables. A d-dimensional Bernoulli process $y_t =$*

$d^{-\frac{1}{2}} ZTf^{(t)}$, for $t \in [m]$, is tame *if, with probability at least $1 - \delta$,*

$$\|y_t\|_2 \leq C\sqrt{\sum_{j=1}^{n} f_j^2}, \quad \textit{for all } t \in [m]. \tag{2.1}$$

The definition anticipates our need for dimension reduction in order to reduce the number of random bits needed for the algorithm. Our first use for it is for BP, which is very simple with $d = 1$ and T the identity matrix. BP requires n random bits, which is too many for a practical streaming algorithm. JLBP, Algorithm 2, exists to fix this problem. Still, if one is willing to disregard the storage needed for the random bits, BP can be substituted everywhere for JLBP without affecting the correctness of our algorithms because our proofs only require that the processes are tame, and BP produces a tame process, as we will now show. We have a similar lemma for JLBP.

Lemma 4 (BP Correctness)**.** *Let $f^{(t)}$, for $t \in [m]$, be the frequency vectors of an insertion-only stream. The sequence $Zf^{(t)}$ returned by the algorithm BP is a tame Bernoulli process.*

Proof. By the Chaining Inequality, Theorem 14, $\exists C'$ s.t. $\mathbb{E}\sup_t |X_t| \leq C'(\sum_j f_j^2)^{1/2}$. Let F be the event that the condition (2.1) holds. Then, for $C \geq C'/\delta$, Markov's Inequality implies that:

$$\Pr(F) = \Pr\left(\sup_t |X_t| \leq C\sqrt{\sum_j f_j^2}\right) \geq (1 - \delta).$$

□

In order to reduce the number of random bits needed for the algorithms we first apply JL transformation T to the frequency vector. The intuition for this comes from the covariance structure of the Bernoulli process, which is what governs the behavior of the process and is fundamental for the Chaining Inequality. The variance of an increment

of the Bernoulli process between times s and $t > s$ is $\|f^{(s:t)}\|_2^2$. The JL-property of the matrix T guarantees that this value is well approximated by $\|Tf^{(s:t)}\|_2^2$, which is the increment variance of the reduced-dimension process. Slepian's Lemma (Lemma 15) is a fundamental tool in the theory of Gaussian processes that allows us to draw a comparison between the suprema of the processes by comparing the increment variances instead. Thus, for $Z_1, \ldots, Z_n \overset{\text{iid}}{\sim}$ Rademacher, the expected supremum of the process $X_t = \sum_{i=1}^n Z_i f_i^{(t)}$ is closely approximated by that of $X'_t = \sum_{i=1}^k Z_i (Tf^{(t)})_i$, and the latter uses only $k = O(\log n)$ random bits. The following lemma formalizes this discussion, its proof is given in Section 2.2.4.

Lemma 5 (JLBP Correctness)**.** *Suppose the matrix T used by JLBP is an $(1 \pm \gamma)$-embedding of $(f^{(t)})_{t\in[m]}$. For any $d \geq 1$, the sequence $\frac{1}{\sqrt{d}} ZTf^{(t)}$ returned by JLBP is a tame d-dimensional Bernoulli process. Furthermore, there exists $d' = O(\log \delta^{-1})$ such that for any $d \geq d'$ and $H \in [n]$ it holds that $\Pr(\frac{1}{2} \leq \|d^{-\frac{1}{2}} ZTe_H\| \leq \frac{3}{2}) \geq 1 - \delta$.*

Algorithm 2 A Bernoulli process with fewer random bits.

procedure JLBP(Stream S)
  Let T be a JL Transformation ▷ The same T will suffice for all instance
  Sample $Z \in \{-1, 1\}^{d\times k}$, s.t. $Z_{i,j} \overset{\text{iid}}{\sim}$ Rademacher
  return $\frac{1}{\sqrt{d}} ZTf^{(t)}$ at each time t
end procedure

Now that we have established the tameness of our Bernoulli processes, let us explain how we can exploit it. We typically exploit tameness in two ways, one works by splitting the stream according to the items and the second splits the stream temporally. Given a stream and a tame Bernoulli process on that stream, every substream defines another Bernoulli process, and the substream processes are tame as well. One way to use this is for heavy hitters. If there is a heavy hitter H, then the substream consisting of all

updates except those to the heavy hitter produces a tame process whose maximum is bounded by $C(F_2 - f_H^2)^{1/2}$, so the value of the process in BP is $Z_H f_H \pm C(F_2 - f_H^2)^{1/2}$. When H is sufficiently heavy, this means that the absolute value of the output of BP tracks the value of f_H, for example if H is a $4C^2$-heavy hitter then the absolute value of BP's output is always a $(1 \pm \frac{1}{2})$-approximation to f_H. Another way we exploit tameness is for approximating F_2 at all points. We select a sequnece of times $t_1 < t_2 < \cdots < t_j \in [m]$ and consider the prefixes of the stream that end at times $t_1, t_2, \ldots$, etc. For each t_i, the prefix stream ending at time t_i is tame with the upper bound depending on the stream's F_2 value at time t_i. If the times t_i are chosen in close enough succession this observation allows us to transform the uniform additive approximation guarantee into a uniform multiplicative approximation.

Description of CountSieve. CountSieve primarily works in two stages that operate concurrently. Each stage uses independent pairs of Bernoulli processes to determine bits of the identity of the heavy hitter. The first stage is the Amplifier, which maintains $L = O(\log \log n)$ independent pairs of Bernoulli processes. The second stage is the Timer and Sieve. It consists of a series of rounds where one pair of AMS sketches is maintained during each round.

CountSieve and its subroutines are described formally in Algorithm 4. The random variables they use are listed in Table 2.2. Even though we reduce the number of random bits needed for each Bernoulli process to a manageable $O(\log n)$ bits, the storage space for the random values is still an issue because the algorithm maintains $O(\log n)$ independent hash functions until the end of the stream. We explain how to overcome this barrier in Section 2.2.4 as well as show that the JL generator of [84] suffices.

$A_{\ell,1}, \ldots, A_{\ell,n} \overset{\text{4-w}}{\sim}$ Bernoulli	$Z_1, \ldots, Z_k \overset{\text{iid}}{\sim}$ Rademacher
$B_{r,1}, \ldots, B_{r,n} \overset{\text{4-w}}{\sim}$ Bernoulli	$R_{r,1}, .., R_{r,n} \overset{\text{4-w}}{\sim}$ Rademacher

TABLE 2.2: Random vectors for CountSieve. Each vector is independent of the others, and $Z = (Z_i)_{i \in [k]}$ is sampled independently for every instance of JLBP.

We can now state an algorithm that maintains a pair of Bernoulli processes and prove that the bits that it outputs favor the process in the pair with the heavy hitter.

Algorithm 3 Split the vector f into two parts depending on A and run a Bernoulli process on each part. Return the identity of the larger estimate at each time.

procedure PAIR(Stream S, $A_1, \ldots A_n \in \{0,1\}$)
  For $b \in \{0,1\}$ let S_b be the restriction
    of S to $\{j \in [n] \mid A_j = b|\}$
  $X_0^{(t)}$ =JLBP($S_0^{(t)}$) at each time t
  $X_1^{(t)}$ =JLBP($S_1^{(t)}$) at each time t
  $b_t = \text{argmax}_{b \in \{0,1\}} \|X_b^{(t)}\|_2$
  return $b_1, b_2, \ldots$
end procedure

Lemma 6 (Pair Correctness)**.** *Let $t_0 \in [m]$ be an index such that:*

$$(f_H^{(t_0)})^2 > 4C^2 \sum_{j \neq H} f_j^2.$$

Let $A_1, \ldots, A_n \overset{p.w.}{\sim}$ Bernoulli and let $b_1, b_2, \ldots, b_m$ be the sequence returned by Pair$(f, A_1, \ldots, A_n)$. Then

$$\Pr(b_t = A_H \text{ for all } t \geq t_0) \geq 1 - 3\delta$$

and, for every $j \in [n] \setminus \{H\}$ and $t \geq t_0$,

$$\Pr(b_t = A_j) \leq \frac{1}{2} + 3\delta.$$

Furthermore, if each JLBP is replaced by an AMS sketch with size $O(\log n \log \delta^{-1})$ then, for all $t \geq t_0$ and $j \neq H$, $P(b_t = A_H) \geq 1 - 2\delta$ and $P(b_t = A_j) \leq \frac{1}{2} + 3\delta$.

Proof. Let $X_0^{(t)} = d^{-\frac{1}{2}} ZTf^{(t)}$ and $X_1^{(t)} = d^{-\frac{1}{2}} WTf^{(t)}$ be the two independent Bernoulli processes output by JLBP. Without loss of generality, suppose that $A_H = 1$, let $v = d^{-\frac{1}{2}} WTe_H$, and let $Y^{(t)} = X_1^{(t)} - f_H^{(t)} v$. By Lemma 5, with probability at least $1 - 2\delta$ all three of the following hold

1. $\|X_0^{(t)}\|_2^2 \leq C^2 \sum_{j:A_j=0} f_j^2$, for all t,
2. $\|Y^{(t)}\|_2^2 \leq C^2 \sum_{\substack{j \neq H \\ A_j = 1}} f_j^2$, for all t, and
3. $\|v\|_2 \geq 1/2$.

If the three events above hold then, for all $t \geq t_0$,

$$\begin{aligned} \|X_1^{(t)}\|_2 - \|X_0^{(t)}\|_2 &\geq \|v f_H^{(t)}\|_2 - \|Y^{(t)}\|_2 - \|X_0^{(t)}\|_2 \\ &\geq \frac{1}{2} f_H^{(t)} - C \sqrt{\sum_{j \neq H} f_j^2} > 0, \end{aligned}$$

which establishes the first claim. The second claim follows from the first using

$$\begin{aligned} \Pr(b_t = A_j) &= \Pr(b_t = A_j = A_H) + \Pr(b_t = A_j \neq A_H) \\ &\leq \Pr(A_j = A_H) + \Pr(b_t \neq A_H) = \frac{1}{2} + 3\delta. \end{aligned}$$

The third and fourth inequalities follow from the correctness of the AMS sketch [6]. □

Amplifier Correctness. The $L = O(\log \log n)$ instances of Pair maintained by Amplifier in the first stage of CountSieve serve to identify a substream containing roughly $n2^{-L} = n/\text{polylog}n$ elements in which H appears as a polylog(n)-heavy hitter. Correctness of Amplifier means that, after some "burn-in" period which we allow to include the first $f_H/2$ updates to H, all of the subsequent updates to H appear in the amplified substream while the majority of other items do not. This is Lemma 7.

Lemma 7 (Amplifier Correctness)**.** *Let $t_0 \in [m]$ be such that $(f_H^{(t_0)})^2 \geq 4C^2 \sum_{j \neq H} f_j^2$; let $a_t = (a_{1,t}, \ldots, a_{L,t})$ denote the length L bit-vector output by the Amplifier at step t. Let $M_{j,t} = \#\{\ell \in [L] \mid a_{\ell,t} = A_{\ell,j}\}$ and $W = \{j \in [n] \setminus \{H\} \mid \exists t \geq t_0, M_{j,t} \geq 0.9L\}$. Then, with probability at least $(1 - 2\delta)$, both of the following hold:*

1. *for all $t \geq t_0$ simultaneously, $M_{H,t} \geq 0.9L$ and*
2. *$\sum_{j \in W} f_j^2 \leq \exp(-\frac{L}{25}) \sum_{j \neq H} f_j^2$.*

Proof. Let $N = \#\{\ell \mid \text{for all } t \geq t_0, a_{\ell,t} = A_{\ell,H}\}$. If $N \geq 0.9L$ then 1 holds. Lemma 6 implies $\mathbb{E}N \geq (1 - 3\delta)L \geq 0.97L$, so Chernoff's Bound easily implies $P(N < 0.9L) = O(2^{-L}) \leq \delta$, where δ is a constant.

Now, let $j \neq H$ be a member of W and suppose that $M_{H,t} \geq 0.9L$. Let $t \geq t_0$ be such that $M_{j,t} \geq 0.9L$. Then it must be that

$$M'_j := \#\{\ell \in [L] \mid A_{\ell,j} = A_{\ell,H}\} \geq 0.8L.$$

However, $\mathbb{E}M'_j = \frac{1}{2}L$ by pairwise independence. Let E_j be the event $\{j \in W \text{ and } M_{H,t} \geq 0.9L\}$. Since the L instances of Pair are independent, an application of Chernoff's Inequality proves that

$$\Pr(E_j) \leq \Pr(M'_j \geq 0.8L) \leq \exp\{\frac{-0.6^2 L}{6}\} \leq e^{-L/20}.$$

We have

$$\mathbb{E}(\sum_{j\in W} f_j^2) = \mathbb{E}(\sum_{j\neq H} 1_{E_j} f_j^2) \leq e^{-L/20} \sum_{j\neq H} f_j^2.$$

Therefore Markov's Inequality yields

$$\Pr\left(\sum_{j\in W} f_j^2 \geq e^{-L/25} \sum_{j\neq H} f_j^2\right) \leq e^{-L/100} \leq \delta.$$

The lemma follows by a union bound. □

Timer and Sieve Correctness. The timing of the rounds in the second stage of CountSieve is determined by the algorithm Timer. Timer outputs a set of times $q_0, q_1, \ldots, q_R$ that break the stream into intervals so that each interval has roughly a $1/\log n$ fraction of the occurrences of H and not too many other items. Precisely, we want that H is everywhere heavy for q, as stated in the following definition. When this holds, in every round the Pair is likely to identify one bit of H, and Sieve and Selector will be likely to correctly identify H from these bits.

Definition 8. *Given an item $H \in [n]$ and a sequence of times $q_0 < q_1 < \cdots < q_R$ in a stream with frequency vectors $(f^{(t)})_{t\in[m]}$ we say that H is* everywhere heavy for *q if, for all $1 \leq r \leq R$,*

$$(f_H^{(q_{r-1}:q_r)})^2 \geq C^2 \sum_{j\neq H} (f_j^{(q_{r-1}:q_r)})^2.$$

Correctness for Timer means that enough rounds are completed and H is sufficiently heavy within each round, i.e., H is everywhere heavy for q.

Lemma 9 (Timer Correctness)**.** *Let S be a stream with an item $H \in [n]$ such that the following hold:*

Algorithm 4 Algorithm for a single F_2 heavy hitters.

procedure COUNTSIEVE(Stream $S = (p_1, p_2, \ldots, p_m)$)
  Maintain $a_t = (a_{1,t}, a_{2,t}, \ldots, a_{L,t}) \leftarrow$AMPLIFIER$(S)$
  Let $t_1 < t_2 < \cdots =$
    $\{t \in [n] \mid A_{\ell,p_t} = a_{\ell,t}$ for at least $0.9L$ values of $\ell\}$
  Let $S_0 = (p_{t_1}, p_{t_2}, \ldots,)$
  $q_0, q_1, \ldots, q_R \leftarrow$TIMER$(S_0)$
  $b_1, b_2, \ldots, b_R \leftarrow$SIEVE$(S_0, q_0, \ldots, q_R)$
  return SELECTOR$(b_1, b_2, \ldots, b_R)$ based on S_0
end procedure

procedure AMPLIFIER(Stream S) ▷ Find a substream where H is polylog(n)-heavy
  for $\ell = 1, 2, \ldots, L$ **do**
    $a_{\ell,1}, a_{\ell,2}, \ldots, a_{\ell,m} \leftarrow$PAIR$(S, A_{\ell,1}, \ldots, A_{\ell,n})$
  end for
  return $a_{1,t}, \ldots, a_{L,t}$ at each time t
end procedure

procedure TIMER(Stream S) ▷ Break the substream into rounds so H is heavy in each
  $q'_0 = 0$
  $Y_t \leftarrow$JLBP(S), for $t = 1, 2, \ldots,$ over S
  For each $r \geq 1$, find $q'_r = \min\{t \mid \|Y_t\|_2 > (1 + \frac{1}{\tau})^r\}$
  Let $q_0, q_1, \ldots, q_R$ be the last $R + 1$ of $q'_0, q'_1, \ldots$
  return $q_0, q_1, \ldots, q_R$
end procedure

procedure SIEVE(Stream $S, q_0, \ldots, q_R$) ▷ Identify one bit of information from each round
  for $r = 0, 1, \ldots, R - 1$ **do**
    $b_{q_r+1}, \ldots, b_{q_{r+1}} \leftarrow$PAIR$(S^{(q_r:q_{r+1})}, B_{r,1}, \ldots, B_{r,n})$
      ▷ Replace JLBP here with AMS
  end for
  return $b_{q_1}, b_{q_2}, \ldots, b_{q_R}$
end procedure

procedure SELECTOR$(b_1, \ldots, b_R)$ ▷ Determine H from the round winners
  return Any $j^* \in \text{argmax}_j \#\{r \in [R] : B_{r,j} = b_r\}$.
end procedure

1. $f_H \geq \tau^4$,
2. $f_H^2 \geq 400C^2 \sum_{j \neq H} f_j^2$, *and*

3. $(f_H^{(t^*:m)})^2 = \frac{1}{4}f_H^2 \geq 25C^2\tau^2 \sum_{j\neq H}(f_j^{(t^*:m)})^2$,

where $t^* = \min\{t \in [m] \mid f_H^{(t)} \geq 0.5 f_H\}$ *and* C *is the constant from Definition 3. If* $q_0, q_1, \ldots, q_R$ *is the sequence output by Timer*(S) *then, with probability at least* $1 - 4\delta$*,* H *is everywhere heavy for* q*.*

Proof. We begin by proving that at least R rounds occur after t^*, which shows that $q_0, \ldots, q_R$ is well defined, and then we show that H is everywhere heavy. Let Y_t be the sequence output by JLBP and let $X_t = Y_t - d^{-\frac{1}{2}}ZTe_H f_H^{(t)}$. Y_t and X_t are tame by Lemma 5 and $\Pr(0.5 \leq \alpha \leq 1.5) \geq 1 - \delta$ where $\alpha = \|d^{-\frac{1}{2}}ZTe_H\|_2$. Hereafter, we suppose that $\alpha \geq 1/2$ and the tameness property holds for Y_t and X_t. With probability at least $1 - \delta$, simultaneously for all $t \in [m]$, we have

$$\|X_t\|_2^2 \leq C^2 \sum_{j\neq H} f_j^2 \leq \frac{1}{400} f_H^2. \tag{2.2}$$

Therefore, $\|Y_{t^*}\|_2 \leq \|X_{t^*}\|_2 + \alpha f_H^{(t^*)} \leq (\frac{\alpha}{2} + \frac{1}{20})f_H$ and $\|Y_m\|_2 \geq \alpha f_H^{(m)} - \|X_m\|_2 \geq (\alpha - \frac{1}{20})f_H$. This implies that the number of rounds completed after t^*, which is

$$\log_{1+1/\tau} \frac{\|Y_m\|_2}{\|Y_{t^*}\|_2} \geq \log_{1+1/\tau} \frac{\alpha - 1/20}{\alpha/2 + 1/20} \geq \log_{1+1/\tau}(3/2),$$

is at least $R + 1$ by our choice of $\tau = 100(R + 1)$. Similarly $\|Y_{t^*}\|_2 \geq \alpha f_H^{(t^*)} - \|X_{t^*}\|_2 \geq (\frac{\alpha}{2} - \frac{1}{20})f_H$. Therefore we also get $q_i > q_{i-1}$ because $(1 + \tau^{-1})\|Y_{t^*}\|_2 \geq 1$ by our assumption that $f_H \geq \tau^4$. Hence $q_0, \ldots, q_R$ are distinct times.

Now we show that H is everywhere heavy for q. Let $W_t = X_t - X_{t^*}$, for $t \geq t^*$. By design, $W_t - W_s = X_t - X_s$, for $s, t \geq t^*$. By Lemma 5, W_t is also a tame process on the suffix of the original stream that has its first item at time $t^* + 1$. Specifically with

probability at least $1-\delta$, for all $t \geq t^*$,

$$\|W_t\|_2^2 \leq C^2 \sum_{j \neq H} (f_j^{(t^*:m)})^2 \leq \frac{1}{400\tau^2} f_H^2.$$

This inequality, with two applications of the triangle inequality, implies

$$\begin{aligned} \alpha f_H^{(q_{i-1}:q_i)} &\geq \|Y_{q_i} - Y_{q_{i-1}}\|_2 - \|W_{q_i} - W_{q_{i-1}}\|_2 \\ &\geq \|Y_{q_i} - Y_{q_{i-1}}\|_2 - \frac{2}{20\tau} f_H. \end{aligned} \tag{2.3}$$

To complete the proof we must bound $\|Y_{q_i} - Y_{q_{i-1}}\|_2$ from below and then apply the heaviness, i.e., assumption 3.

Equation (2.2) and the triangle inequality imply that, for every $t \geq t^*$, it holds that $\|Y_t\|_2 \geq \alpha f_H^{(t)} - \|X_t\|_2 \geq (\frac{\alpha}{2} - \frac{1}{20}) f_H$. Recalling the definition of $q'_0, q'_1, \cdots$ from Timer, since $t^* \leq q_0 < q_1 < \cdots < q_R$ and the rounds expand at a rate $(1 + 1/\tau)$,

$$\|Y_{q_{i+1}} - Y_{q_i}\|_2 \geq \frac{1}{\tau} \left(\frac{\alpha}{2} - \frac{1}{20} \right) f_H. \tag{2.4}$$

Using what we have already shown in (2.3) we have

$$\alpha f_H^{(q_i:q_{i+1})} \geq \frac{1}{\tau} \left(\frac{\alpha}{2} - \frac{1}{20} - \frac{2}{20} \right) f_H$$

so dividing and using $\alpha \geq 1/2$ and C sufficiently large we get

$$\begin{aligned} (f_H^{(q_i:q_{i+1})})^2 \geq \frac{1}{25\tau^2} f_H^2 &\geq C^2 \sum_{j \neq H} (f_j^{(t^*:m)})^2 \\ &\geq C^2 \sum_{j \neq H} (f_j^{(q_i:q_{i+1})})^2. \end{aligned}$$

Since this holds for all i, H is everywhere heavy for q. We have used the tameness of the three processes (X, Y, and W) and the bounds on α. Each of these fails with probability at most δ, so the probability that Timer fails to achieve the condition that H is everywhere heavy for q is at most 4δ. □

During each round, the algorithm Sieve uses a hash function A to split the stream into two parts and then determines which part contains H via Pair. For these instances of Pair, we replace the two instances of JLBP with two instances of AMS. This replacement helps us to hold down the storage when we later use Nisan's PRG, because computing the JL transformation T from [84] requires $O(\log n \log\log n)$ bits. Applying Nisan's PRG to an algorithm that computes entries in T would leave us with a bound of $O(\log n(\log\log n)^2)$. More details can be found in Section 2.2.4.

A total of $O(\log n)$ rounds is enough to identify the heavy hitter and the only information that we need to save from each round is the hash function A and the last bit output by Pair. Selector does the work of finally identifying H from the sequence of bits output by Sieve and the sequence of hash functions used during the rounds. We prove the correctness of Sieve and Selector together in the following lemma.

Lemma 10 (Sieve/Selector)**.** *Let $q_0, q_1, \ldots, q_R$ be the sequence output by* TIMER(S) *and let $b_1, \ldots, b_R$ be the sequence output by* SIEVE$(S, q_0, \ldots, q_R)$*. If H is everywhere heavy for q on the stream S then, with probability at least $1 - \delta$, Selector$(b_1, \ldots, b_R)$ returns H.*

Proof. Lemma 6 in the AMS case implies that the outcome of round r satisfies $\Pr(b_r = B_{r,H}) \geq 1 - 3\delta$ and $\Pr(b_r = B_{r,j}) \leq \frac{1}{2} + 3\delta$. The random bits used in each iteration of the for loop within Sieve are independent of the other iterations. Upon choosing the number of rounds $R = O(\log n)$ to be sufficiently large, Chernoff's Inequality implies that, with

high probability, H is the unique item in $\operatorname{argmax}_j \#\{r \in [R] \mid B_{r,j} = b_r\}$. Therefore, Selector returns H. □

Algorithm 5 ℓ_2 heavy hitters algorithm.

procedure ℓ_2HEAVYHITTERS($S = (p_1, p_2, \ldots, p_m)$)
  $Q \leftarrow O(\log \varepsilon^{-1})$, $B \leftarrow O(\varepsilon^{-2})$
  Select independent 2-universal hash functions
    $h_1, \ldots, h_Q, h'_1, \ldots, h'_Q : [n] \to [B]$
    and $\sigma_1, \ldots, \sigma_Q : [n] \to \{-1, 1\}$.
  $\hat{F}_2 \leftarrow (1 \pm \frac{\varepsilon}{10})F_2$ using AMS [6]
  $\hat{\mathcal{H}} \leftarrow \emptyset$
  for $(q, b) \in Q \times B$ **do**
    Let $S_{q,b}$ be the stream of items i with $h_q(i) = b$
    $c_{q,b} \leftarrow \sum_{j:h'_q(j)=b} \sigma_q(j) f_j$ ▷ The CountSketch [39]
    $H \leftarrow$ COUNTSIEVE($S_{q,b}$)
  end for
  Remove from $\hat{\mathcal{H}}$ any item i such that
    $\text{median}_q\{|c_{q,h_q(i)}|\} \leq \frac{3\varepsilon}{4}\hat{F}_2$.
  return $\hat{\mathcal{H}}$
end procedure

CountSieve Correctness. We now have everything in place to prove that CountSieve correctly identifies a sufficiently heavy heavy hitter H. As for the storage bound and Theorem 1, the entire algorithm fits within $O(\log n \log\log n)$ bits except the $R = O(\log n)$ hash functions required by Sieve. We defer their replacement to Theorem 19 in Section 2.2.4.

Theorem 11 (CountSieve Correctness)**.** *If H is a $400C^2$-heavy hitter then the probability that CountSieve returns H is at least* 0.95*. The algorithm uses $O(\log n \log\log n)$ bits of storage and can be implemented with $O(\log n \log\log n)$ stored random bits.*

Proof. We use Theorem 16 to generate the JL transformation T. Each of our lemmas requires that T embeds a (possibly different) polynomially sized set of vectors, so, for

$\delta = \Omega(1)$, Theorem 16 implies that, with probability at least $1 - \delta$, T embeds all of the necessary vectors with seed length $O(\log n)$, and the entries in T can be computed in space $O(\log n \log\log n)$ bits of space. Because of the heaviness assumption, the conclusion of Lemma 7 fails to hold for $t_0 = t^*$ (defined in Lemma 9) with probability at most 2δ. When that failure does not occur, the second and third hypotheses in Lemma 9 hold. The first hypothesis is that $f_H \geq \tau^4$, suppose it holds. Then the probability that H fails to be everywhere heavy for the sequence q that is output by Timer is at most 4δ. In this case, according to Lemma 10, Sieve and Selector correctly identify H except with probability at most δ. Therefore, the algorithm is correct with probability at least $1 - 8\delta \geq 0.95$, by choosing $\delta \leq 1/200$. If $f_H < \tau^4$, then because H is a heavy hitter, we get $\sum_{j \neq H} f_j^2 \leq \tau^8 = O(\log^8 n)$. Then we choose the constant factor in L large enough so that, the second conclusion of Lemma 7 implies $\sum_{j \in W} f_j^2 \leq e^{-L/25} < 1$. This means that H is the only item that passes the amplifier for all $t \geq t^*$, and, no matter what is the sequence output by Timer, H is everywhere heavy because it is the only item in the substream. Thus, in this case the algorithm also outputs H.

Now we analyze the storage and randomness. Computing the entries in the Kane-Meka-Nelson JL matrix requires $O(\log n \log\log n)$ bits of storage, by Theorem 16, and there is only one of these matrices. Amplifier stores $L = O(\log\log n)$ counters. Sieve, Timer, and Selector each require $O(\log n)$ bits at a time (since we discard any value as soon as it is no longer needed). Thus the total working memory of the algorithm is $O(\log n \log\log n)$ bits. The random seed for the JL matrix has $O(\log n)$ bits. Each of the $O(\log\log n)$ Bernoulli processes requires $O(\log n)$ random bits. By Theorem 19 below, the remaining random bits can be generated with Nisan's generator using a seed of $O(\log n \log\log n)$ bits. Using Nisan's generator does not increase the storage of the algorithm. Accounting for all of these, the total number of random bits used by CountSieve,

which also must be stored, is $O(\log n \log\log n)$. Therefore, the total storage used by the algorithm is $O(\log n \log\log n)$ bits. □

Theorem 1 (ℓ_2-Heavy Hitters)**.** *For any $\varepsilon > 0$, there is a 1-pass algorithm in the insertion-only model that, with probability at least $2/3$, finds all those indices $i \in [n]$ for which $f_i \geq \varepsilon\sqrt{F_2}$, and reports no indices $i \in [n]$ for which $f_i \leq \frac{\varepsilon}{2}\sqrt{F_2}$. The space complexity is $O(\frac{1}{\varepsilon^2}\log\frac{1}{\varepsilon}\log n\log\log n)$ bits.*

Proof. The algorithm is Algorithm 5. It has the form of a CountSketch [39] with $Q = O(\log 1/\varepsilon)$ "rows" and $B = 8(10C)^2/\varepsilon^2$ "buckets" per row, wherein we run one instance of CountSieve in each bucket to identify potential heavy hitters and also the usual CountSketch counter in each bucket. Finally, the algorithm discriminates against non-heavy hitters by testing their frequency estimates from the CountSketch. We will assume that the AMS estimate $\hat{F}_2$ is correct with probability at least $8/9$.

Let $\mathcal{H}_k = \{i \mid f_i \geq \frac{\varepsilon}{k}\sqrt{F_2}\}$ and let $\hat{\mathcal{H}}$ be set of distinct elements returned by Algorithm 5. To prove the theorem, it is sufficient to prove that, with probability at least $2/3$, $\mathcal{H}_1 \subseteq \hat{\mathcal{H}} \subseteq \mathcal{H}_2$.

Let $H \in \mathcal{H}_1$ and consider the stream $S_{q,h_q(H)}$ at position $(q, h_q(H))$. We have

$$\mathbb{E}\Big(\sum_{\substack{j\neq H\\ h_q(j)=h_q(H)}} f_j^2\Big) \leq \frac{\varepsilon^2}{8(10C)^2}F_2.$$

Let $E_{q,H}$ be the event that

$$\sum_{\substack{j\neq H\\ h_q(j)=h_q(H)}} f_j^2 \leq \frac{\varepsilon^2}{(10C)^2}F_2,$$

so by Markov's Inequality $\Pr(E_{q,H}) \geq 7/8$. When $E_{q,H}$ occurs H is sufficiently heavy in $S_{q,h_q(H)}$ for CountSieve. By Theorem 11, with probability at least $\frac{7}{8} - \frac{1}{20} \geq 0.8$, CountSieve

identifies H. Therefore, with the correct choice of the constant factor for Q, a Chernoff bound and a union bound imply that, with probability at least $1 - 1/9$, every item in $\mathcal{H}_1$ is returned at least once by a CountSieve.

Let $\hat{\mathcal{H}}'$ denote the set $\hat{\mathcal{H}}$ before any elements are removed in the final step. Since CountSieve identifies at most one item in each bucket, $|\hat{\mathcal{H}}'| = O(\varepsilon^{-2} \log \varepsilon^{-1})$. By the correctness of CountSketch [39] and the fact that it is independent of $\hat{H}'$, we get that, with probability at least $1 - 1/9$, for all $i \in \hat{H}'$

$$\left| f_i - \text{median}_q \{ |c_{q,h_q(i)}| \} \right| \leq \frac{\varepsilon}{10C} \sqrt{F_2}.$$

When this happens and the AMS estimate is correct, the final step of the algorithm correctly removes any items $i \notin \mathcal{H}_2$ and all items $i \in \mathcal{H}_1$ remain. This completes the proof of correctness.

The storage needed by the CountSketch is $O(BQ \log n)$, the total storage needed for all instances of CountSieve is $O(BQ \log n \log \log n)$, and the storage needed for AMS is $O(\varepsilon^{-2} \log n)$. Therefore, the total number of bits of storage is

$$O(BQ \log n \log \log n) = O(\frac{1}{\varepsilon^2} \log \frac{1}{\varepsilon} \log n \log \log n).$$

□

Corollary 11.1. *There exists an insertion-only streaming algorithm that returns an additive $\pm\varepsilon\sqrt{F_2}$ approximation to ℓ_∞, with probability at least $2/3$ and requires $O(\frac{1}{\varepsilon^2} \log \frac{1}{\varepsilon} \log n \log \log n)$ bits of space.*

Proof. Use Algorithm 5. If no heavy-hitter is returned then the ℓ_∞ estimate is 0, otherwise return the largest of the CountSketch medians among the discovered heavy hitters. The

correctness follows from Theorem 1 and the correctness of CountSketch. □

2.2.3 Chaining Inequality

We call these inequalities Chaining Inequalities after the Generic Chaining, which is the technique that we use to prove it. The book [136] by Talagrand contains an excellent exposition of the subject. Let $(X_t)_{t\in T}$ be a Gaussian process. The Generic Chaining technique concerns the study of the supremum of X_t in a particular metric space related to the variances and covariances of the process. The metric space is (T, d) where $d(s,t) = (\mathbb{E}(X_s - X_t)^2)^{\frac{1}{2}}$. The method takes any finite chain of finite subsets $T_0 \subseteq T_1 \subseteq \cdots \subseteq T_n \subseteq T$ and uses $(X_t)_{t\in T_i}$ as a sequence of successive approximations to $(X_t)_{t\in T}$ wherein X_t, for $t \notin T_i$, is approximated by the value of the process at some minimizer of $d(t, T_i) = \min\{d(t,s) \mid s \in T_i\}$. To apply the Generic Chaining one must judiciously choose the chain in order to get a good bound, and the best choice necessarily depends on the structure of the process. We will exploit the following lemma.

Lemma 12 ([136]). *Let $\{X_t\}_{t\in T}$ be a Gaussian process and let $T_0 \subseteq T_1 \cdots \subseteq T_n \subseteq T$ be a chain of sets such that $|T_0| = 1$ and $|T_i| \leq 2^{2^i}$ for $i \geq 1$. Then*

$$\mathbb{E}\sup_{t\in T} X_t \leq O(1)\sup_{t\in T}\sum_{i\geq 0} 2^{i/2} d(t, T_i). \tag{2.5}$$

The Generic Chaining also applies to Bernoulli processes, but, for our purposes, it is enough that we can compare related Gaussian and Bernoulli processes.

Lemma 13 ([136]). *Let $A \in \mathbb{R}^{m\times n}$ be any matrix and let G and B be n-dimensional vectors with independent coordinates distributed as $N(0,1)$ and Rademacher, respectively. Then the Gaussian process $X = AG$ and Bernoulli process $Y = AB$ satisfy $\mathbb{E}\sup_{t\in T} Y_t \leq \sqrt{\frac{\pi}{2}}\mathbb{E}\sup_{t\in T} X_t$.*

Theorem 14 (Chaining Inequality)**.** *Define independent $\mathcal{N}(0,1)$ random variables $Z_1, \ldots, Z_n$ and let $(f^{(t)})_{t\in[m]}$ be the sequence of frequency vectors of an insertion-only stream. There exists a universal constant $C' > 0$ such that if $X_t = \sum_{j=1}^{n} Z_j f_j^{(t)}$, for $t \in [m]$, then*

$$\mathbb{E} \sup_i |X_i| \leq C' \sqrt{\mathrm{Var}(X_m)} = C' \|f^{(m)}\|_2. \tag{2.6}$$

If $\bar{Z}_1, \ldots, \bar{Z}_n \ldots \overset{iid}{\sim}$ Rademacher and $Y_t = \sum_{j=1}^{n} \bar{Z}_j f_j^{(t)}$, for $t \in [m]$, then

$$\mathbb{E} \sup_i |Y_i| \leq C' \sqrt{\mathrm{Var}(Y_m)} = C' \|f^{(m)}\|_2. \tag{2.7}$$

Proof. Let $T = [m]$. Define $T_0 = \{t_0\}$, where t_0 is the index such that $\mathrm{Var}(X_{t_0}) < 0.5\,\mathrm{Var}(X_m) \leq \mathrm{Var}(X_{t_0+1})$ and $T_i = \{1, t_{i,1}, t_{i,2}, \ldots\}$ where for each index $t_{i,j} \in T_i$ $\mathrm{Var}(X_{t_{i,j}}) < \frac{j}{2^{2^i}} \mathrm{Var}(X_m) \leq \mathrm{Var}(X_{t_{i,j+1}})$. This is well-defined because $\mathrm{Var}(X_t) = \|f^{(t)}\|_2^2$ is the second moment of an insertion-only stream, which must be monotonically increasing. By construction $|T_i| \leq 2^{2^i}$ and, for each $t \in T$, there exist $t_{i,j} \in T_j$ such that

$$\begin{aligned} d(t, T_i) &= \min(d(t, t_{i,j}), d(t, t_{i,j+1})) \\ &\leq d(t_{i,j}, t_{i,j+1}) = (\mathbb{E}(X_{t_{i,j+1}} - X_{t_{i,j}})^2)^{\frac{1}{2}}, \end{aligned}$$

where the last inequality holds because $E(X_t^2)$ monotonically increasing with t.

Notice that every pair of increments has nonnegative covariance because the stream is insertion-only. Thus, the following is true:

$$\begin{aligned} d(t, t_{i,j+1})^2 &\le \mathbb{E}(X_{t_{i,j+1}} - X_{t_{i,j}})^2 \\ &\le \mathbb{E}(X_{t_{i,j+1}} - X_{t_{i,j}})^2 + 2\mathbb{E}X_{t_{i,j}}(X_{t_{i,j+1}} - X_{t_{i,j}}) \\ &= \mathbb{E}X^2_{t_{i,j+1}} - \mathbb{E}X^2_{t_{i,j}} \\ &\le \frac{j+1}{2^{2^i}}\mathbb{E}X_m^2 - \frac{j-1}{2^{2^i}}\mathbb{E}X_m^2 = \frac{2}{2^{2^i}}\mathbb{E}X_m^2. \end{aligned}$$

Then we can conclude that

$$\sum_{i\ge 0} 2^{i/2} d(t, T_i) \le \sum_{i\ge 0} 2^{i/2}\frac{2}{2^{2^i}}\sqrt{\mathbb{E}X_m^2} = O(1)\sqrt{\mathrm{Var}(X_m)}.$$

Applying (2.5) we obtain $\mathbb{E}\sup_{t\in T} X_t \le O(1)\sqrt{\mathrm{Var}(X_m)}$.

In order to bound the absolute value, observe

$$\begin{aligned} \sup_t |X_t| &\le |X_1| + \sup|X_t - X_1| \\ &\le |X_1| + \sup_{s,t}(X_t - X_s) \\ &= |X_1| + \sup_t X_t + \sup_s(-X_s). \end{aligned} \tag{2.8}$$

Thus, $\mathbb{E}\sup_t |X_t| \le \mathbb{E}|X_1| + 2\mathbb{E}\sup X_t \le O(1)\sqrt{\mathrm{Var}(X_m)}$, because $-X_t$ is also Gaussian process with the same distribution as X_t and $\mathbb{E}|X_1| = O(\sqrt{\mathrm{Var}(X_m)})$ because $f^{(1)} = 1$. This establishes (2.6) and (2.7) follows immediately by an application of Lemma 13. □

Theorem 14 would obviously *not* be true for a stream with deletions, since we may

have $\mathrm{Var}(X_m) = 0$. One may wonder if the theorem would be true for streams with deletions upon replacing $\mathrm{Var}(X_m)$ by $\max_t \mathrm{Var}(X_t)$. This is not true, and a counter example is the stream $(e_1, -e_1, e_2, \ldots, e_n, -e_n)$ which yields $\max_t \mathrm{Var}(X_t) = 1$, but $\mathbb{E}\sup_t |X_t| = \Theta(\sqrt{\log n})$.

Theorem 14 does not apply to the process ouput by JLBP, but the covariance structures of the two processes are very similar because T is an embedding. We can achieve basically the same inequality for the JLBP process by applying Slepian's Lemma, mimicking the strategy in [106].

Lemma 15 (Slepian's Lemma [95])**.** *Let X_t and Y_t, for $t \in T$, be Gaussian processes such that $\mathbb{E}(X_s - X_t)^2 \leq \mathbb{E}(Y_s - Y_t)^2$, for all $s, t \in T$. Then, $\mathbb{E}\sup_{t\in T} X_t \leq \mathbb{E}\sup_{t\in T} Y_t$.*

Corollary 15.1 (Chaining Inequality with JL)**.** *Let T be a $(1 \pm \gamma)$-embedding of $(f^{(t)})_{t\in[m]}$ and let $Z_1, \ldots, Z_k \overset{iid}{\sim} \mathcal{N}(0,1)$. There exists a universal constant $C' > 0$ such that if $X_t = \langle Z, Tf^{(t)}\rangle$, for $t \in [m]$, then $\mathbb{E}\sup_i |X_i| \leq C'\|f^{(m)}\|_2$. If $\bar{Z}_1, \ldots, \bar{Z}_k \overset{iid}{\sim}$ Rademacher and $Y_t = \langle \bar{Z}, Tf^{(t)}\rangle$, for $t \in [m]$, then $\mathbb{E}\sup_i |Y_i| \leq C'\|f^{(m)}\|_2$.*

Proof. Let W_t be the Gaussian process from Theorem 14. Since T is a JL transformation

$$\begin{aligned}\mathbb{E}(X_t - X_s)^2 = \|Tf^{(s:t)}\|_2^2 &\leq (1+\gamma)^2\|f^{(s:t)}\|_2^2 \\ &= (1+\gamma)^2\mathbb{E}(W_t - W_s)^2.\end{aligned}$$

The first claim of the corollary follows from Slepian's Lemma, Equation (2.8), and Theorem 14. The second inequality follows from the first and Lemma 13. □

2.2.4 Reduced randomness

This section describes how CountSieve can be implemented with only $O(\log n \log\log n)$ random bits. There are two main barriers to reducing the number of random bits. We have already partially overcome the first barrier, which is to reduce the number of bits needed by a Bernoulli process from n, as in the algorithm BP, to $O(\log n)$ by introducing JLBP. JLBP runs $d = O(1)$ independent Bernoulli processes in dimension $k = O(\log n)$ for a total of $dk = O(\log n)$ random bits. This section proves the correctness of that algorithm.

The second barrier is finding a surrogate for the R independent vectors of pairwise independent Bernoulli random variables that are used during the rounds of Sieve. We must store their values so that Selector can retroactively identify a heavy hitter, but, naïvely, they require $\Omega(\log^2 n)$ random bits. We will show that one can use Nisan's pseudorandom generator (PRG) with seed length $O(\log n \log\log n)$ bits to generate these vectors. A priori, it is not obvious that this is possible. The main sticking point is that the streaming algorithm that we want to derandomize must store the random bits it uses, which means that these count against the seed length for Nisan's PRG. Specifically, Nisan's PRG reduces the number of random bits needed by a space S algorithm using R random bits to $O(S \log R)$. Because CountSieve must pay to store the R random bits, the storage used is $S \geq R = \Omega(\log^2 n)$, so Nisan's PRG appears even to increase the storage used by the algorithm! We can overcome this by introducing an auxiliary (non-streaming) algorithm that has the same output as Sieve and Selector, but manages without storing all of the random bits. This method is similar in spirit to Indyk's derandomization of linear sketches using Nisan's PRG [78]. It is not a black-box reduction to the auxiliary algorithm and it is only possible because we can exploit the structure of Sieve and Selector.

We remark here that we are not aware of any black-box derandomization of the Bernoulli processes that suits our needs. This is for two reasons. First, we cannot reorder the stream

for the purpose of the proof because the order of the computation is important. Reordering the stream is needed for Indyk's argument [78] for applying Nisan's PRG. Second, the seed length of available generators is too large, typically in our setting we would require a seed of length at least $\log^{1+\delta} n$, for some $\delta > 0$.

The Bernoulli Process with $O(\log n)$ Random Bits. The main observation that leads to reducing the number of random bits needed by the algorithm is that the distribution of the corresponding Gaussian process depends only on the second moments of the increments. These moments are just the square of the Euclidean norm of the change in the frequency vector, so applying a Johnson-Lindenstrauss transformation to the frequency vector nearly preserves the distribution of the process and allows us to get away with $O(\log n)$ random bits. One trouble with this approach is that the heavy hitter H could be "lost", whereby we mean that although $\|Te_H\| \approx 1$ it may be that $\langle Z, Te_H \rangle \approx 0$, for the Rademacher random vector Z, whereupon H's contribution to the sum $\langle Z, Tf^{(t)} \rangle$ is lost among the noise. To avoid this possibility we keep $d = O(1)$ independent Bernoulli processes in parallel.

First, we state correctness of the Johnson-Lindenstrauss transformation that we use and the storage needed for it.

Theorem 16 (Kane, Meka, & Nelson [84])**.** *Let V be a set of n points in $\mathbb{R}^n$. For any constant $\delta > 0$ there exists a $k = O(\gamma^{-2} \log(n/\delta))$ and generator $G : \{0,1\}^{O(\log n)} \times [k] \times [n] \to \mathbb{R}$ such that, with probability at least $1 - \delta$, the $k \times n$ matrix T with entries $T_{ij} = G(R, i, j)$ is a $(1 \pm \gamma)$-embedding of V, where $R \in \{0,1\}^{O(\log n)}$ is a uniformly random string. The value of $G(R, i, j)$ can be computed with $O(\log n \log\log n)$ bits of storage.*

Lemma 5 (JLBP Correctness)**.** *Suppose the matrix T used by JLBP is an $(1 \pm \gamma)$-embedding of $(f^{(t)})_{t\in[m]}$. For any $d \geq 1$, the sequence $\frac{1}{\sqrt{d}} ZTf^{(t)}$ returned by JLBP is a tame d-dimensional Bernoulli process. Furthermore, there exists $d' = O(\log \delta^{-1})$ such that for any $d \geq d'$ and $H \in [n]$ it holds that $\Pr(\frac{1}{2} \leq \|d^{-\frac{1}{2}} ZTe_H\| \leq \frac{3}{2}) \geq 1 - \delta$.*

Proof. Let $X_{i,t} = \sum_{j=1}^{k} Z_{ij} (Tf^{(t)})_j$ and

$$X_t = \|\frac{1}{\sqrt{d}} ZTf^{(t)}\|_2^2 = \frac{1}{d} \sum_{i=1}^{d} X_{i,t}^2,$$

for $t = 1, \ldots, m$. Each process $X_{i,t}$ is a Bernoulli process with $\mathrm{Var}(X_{i,t}) = \|Tf^{(t)}\|_2^2 \leq (1+\gamma)^2 \|f^{(t)}\|_2^2$ and, for $s < t$, $\mathbb{E}(X_{i,t} - X_{i,s})^2 = \|Tf^{(s:t)}\|_2^2 \leq (1+\gamma)^2 \|f^{(s:t)}\|_2^2$.

Notice that for all i Gaussian processes $(X_{i,t})_{t\in[m]}$ are from same distribution. Let X'_t be a Gaussian process that is identical to $X_{i,t}$, except that the Rademacher random variables are replaced by standard Gaussians. X'_t and $X_{i,t}$ have the same means, variances, and covariances. Thus, $\mathbb{E} \sup_t |X_{i,t}| \leq \sqrt{\frac{\pi}{2}} \mathbb{E} \sup_t |X'_t|$, by Lemma 13.

Let $N_1, \ldots, N_n \overset{\text{iid}}{\sim} N(0,1)$. We will compare X'_t against the Gaussian process $X''_t = (1+\gamma)\frac{1}{\sqrt{d}} \langle N, f^{(t)} \rangle$. By the Chaining Inequality, there exists C' such that $\mathbb{E} \sup |X''_t| \leq C' \sqrt{\mathrm{Var}(X''_m)} = \frac{C'(1+\gamma)}{\sqrt{d}} \|f^{(m)}\|_2$. We have $\mathbb{E}(X''_t - X''_s)^2 = \frac{1}{d}(1+\gamma)^2 \|f^{(s:t)}\|_2^2$, so by Slepian's Lemma applied to X'_t and X''_t and by (2.8) we have

$$\begin{aligned} \mathbb{E} \sup_t |X_{i,t}| &\leq \sqrt{\frac{\pi}{2}} \mathbb{E} \sup |X'_t| \leq \sqrt{\frac{\pi}{2}} \sqrt{d} \mathbb{E} \sup_t |X''_t| \\ &\leq \sqrt{\frac{\pi}{2}} (1+\gamma) C' \|f^{(m)}\|_2. \end{aligned}$$

Now we apply Markov's Inequality to get $\Pr(\sup_t |X_{i,t}| \geq \frac{C}{\sqrt{d}} \|f^{(m)}\|_2) \leq \frac{\delta}{d}$, by taking $C \geq \sqrt{\frac{\pi}{2}} (1+\gamma) C' d^{3/2} / \delta$. From a union bound we find $\Pr(\sup_{i,t} |X_{i,t}| \geq \frac{C}{\sqrt{d}} \|f^{(m)}\|_2) \leq \delta$,

and that event implies $\sup_t |X_t| \leq C\|f^{(m)}\|_2$, which is (2.1) and proves that the process is tame.

For the second claim, we note that the matrix $\frac{1}{\sqrt{d}}Z$ is itself a type of Johnson-Lindenstrauss transformation (see [1]), hence $\frac{1}{2} \leq \|d^{-1/2}ZTe_H\| \leq \frac{3}{2}$, with probability at least $1 - 2^{-d} \geq (1 - \delta)$. The last inequality follows by our choice of d. □

Sieve and Selector. In the description of the algorithm, the Sieve and Selector use $O(\log n)$ many pairwise independent hash functions that are themselves independent. Nominally, this needs $O(\log^2 n)$ bits. However, as we show in this section, it is sufficient to use Nisan's pseudorandom generator [116] to generate the hash functions. This reduces the random seed length from $O(\log^2 n)$ to $O(\log n \log\log n)$. Recall the definition of a pseudorandom generator.

Definition 17. *A function $G : \{0,1\}^m \to \{0,1\}^n$ is called a* pseudorandom generator (PRG) for space(S) with parameter ε *if for every randomized space(S) algorithm A and every input to it we have that*

$$\|\mathcal{D}_y(A(y)) - \mathcal{D}_x(A(G(x))\|_1 < \varepsilon,$$

where y is chosen uniformly at random in $\{0,1\}^n$, x uniformly in $\{0,1\}^m$, and $\mathcal{D}(\cdot)$ is the distribution of $\cdot$ as a vector of probabilities.

Nisan's PRG [116] is a pseudorandom generator for space S with parameter 2^{-S} that takes a seed of length $O(S \log R)$ bits to R bits. The total space used by Sieve and Selector is $O(\log n)$ bits for the algorithm workspace and $O(\log^2 n)$ bits to store the hash functions.

We will be able to apply Nisan's PRG because Sieve only accesses the randomness in $O(\log n)$ bit chunks, where the rth chunk generates the 4-wise independent random variables needed for the rth round, namely $B_{r1}, \ldots, B_{rn}$ and the bits for two instances of

the AMS sketch. We can discard the AMS sketches at the end of each round, but in order to compute its output after reading the entire stream, Selector needs access to the bit sequence $b_1, b_2, \ldots, b_R$ as well as B_{ri}, for $r \in [R]$ and $i \in [n]$. Storing the B random variables, by their seeds, requires $O(\log^2 n)$ bits. This poses a problem for derandomization with Nisan's PRG because it means that Sieve and Selector are effectively a $O(\log^2 n)$ space algorithm, even though most of the space is only used to store random bits.

We will overcome this difficulty by derandomizing an auxiliary algorithm. The auxiliary algorithm computes a piece of the information necessary for the outcome, specifically for a given item $j \in [n]$ in the stream the auxiliary item will compute $N_j := \#\{r \mid b_r = B_{rj}\}$ the number of times j is on the "winning side" and compare that value to $3R/4$. Recall that the Selector outputs as the heavy hitter a j that maximizes N_j. By Lemma 6 for the AMS case, $\mathbb{E}N_j$ is no larger than $(\frac{1}{2} + 3\delta)R$, if j is not the heavy element, and $\mathbb{E}N_H$ is at least $(1 - 3\delta)R$ if H is the heavy element. When the Sieve is implemented with fully independent rounds, Chernoff's Inequality implies that $N_H > 3R/4$ or $N_j \leq 3R/4$ with high probability. When we replace the random bits for the independent rounds with bits generated by Nisan's PRG we find that for each j with high probability N_j remains on the same side of $3R/4$.

Here is a formal description of the auxiliary algorithm. The auxiliary algorithm takes the sequence $q_0, q_1, \ldots, q_R$ as input (which is independent of the bits we want to replace with Nisan's PRG), the stream S, and an item label j, and it outputs whether $N_j > 3R/4$. It initializes $N_i = 0$, and then for each round $r = 1, \ldots, R$ it draws $O(\log n)$ random bits and computes the output b_r of the round. If $b_r = B_{rj}$ then N_i is incremented, and otherwise it remains unchanged during the round. The random bits used by each round are discarded at its end. At the end of the stream the algorithm outputs 1 if $N_j > 3R/4$.

Lemma 18. *Let $X \in \{0,1\}$ be the bit output by the auxiliary algorithm, and let $\widetilde{X} \in \{0,1\}$ be the bit output by the auxiliary algorithm when the random bits it uses are generated by Nisan's PRG with seed length $O(\log n \log\log n)$. Then $|\Pr(X=1) - \Pr(\widetilde{X}=1)| \leq \frac{1}{n^2}$.*

Proof. The algorithm uses $O(\log n)$ bits of storage and $O(\log^2 n)$ bits of randomness. The claim follows by applying Nisan's PRG [116] with parameters $\varepsilon = 1/n^2$ and seed length $O(\log n \log\log n)$. □

Theorem 19. *Sieve and Selector can be implemented with $O(\log(n)\log\log n)$ random bits.*

Proof. Let N_j be the number of rounds r for which $b_r = B_{rj}$ when the algorithm is implemented with independent rounds, and let $\widetilde{N}_j$ be that number of rounds when the algorithm is implemented with Nisan's PRG. Applying Lemma 18 we have for every item j that $|\Pr(\widetilde{N}_j > 3R/4) - P(N_j > 3R/4)| \leq 1/n^2$. Thus, by a union bound, the probability that the heavy hitter H is correctly identified changes by no more than $n/n^2 = 1/n$. The random seed requires $O(\log n \log\log n)$ bits of storage, and aside from the random seeds the algorithms use $O(\log n)$ bits of storage. Hence the total storage is $O(\log n \log\log n)$ bits. □

2.2.5 F_2 at all points

One approach to tracking F_2 at all times is to use the median of $O(\log n)$ independent copies of an F_2 estimator like the AMS algorithm [6]. A Chernoff bound drives the error probability to $1/\text{poly}(n)$, which is small enough for a union bound over all times, but it requires $O(\log^2 n)$ bits of storage to maintain all of the estimators. The Chaining Inequality allows us to get a handle on the error during an interval of times. Our approach to tracking F_2 at all times is to take the median of $O(\log \frac{1}{\varepsilon} + \log\log n)$ Bernoulli processes. In any short enough interval—where F_2 changes by only a $(1 + \Omega(\varepsilon^2))$ factor—each of

the processes will maintain an accurate estimate of F_2 for the entire interval, with constant probability. Since there are only $O(\varepsilon^{-2}\log^2(n))$ intervals we can apply Chernoff's Inequality to guarantee the tracking on every interval, which gives us the tracking at all times. This is a direct improvement over the F_2 tracking algorithm of [76] which for constant ε requires $O(\log n(\log n + \log\log m))$ bits.

The algorithm has the same structure as the AMS algorithm, except we replace their sketches with instances of JLBP. Theorem 2 follows immediately from Theorem 21.

Algorithm 6 An algorithm for approximating F_2 at all points in the stream.

procedure F2ALWAYS(Stream S)
  $N \leftarrow O(\frac{1}{\varepsilon^2})$, $R \leftarrow O(\log(\frac{1}{\varepsilon^2}\log n))$
  $X_{i,r}^{(t)} \leftarrow \text{JLBP}(S)$ for $i \in [N]$ and $r \in [R]$.
    ▷ Use a $(1 \pm \frac{\varepsilon}{3})$-embedding T in this step.
  $Y_r^{(t)} = \frac{1}{N}\sum_{i=1}^{N} \|X_{i,r}^{(t)}\|_2^2$
  return $\hat{F}_2^{(t)} = \text{median}_{r\in R}\{Y_r^{(t)}\}$ at each time t
end procedure

Lemma 20. *Let $N = O(\frac{1}{\delta\varepsilon^2})$ and let $X_i^{(t)}$, for $i = 1,\ldots,N$, be independent copies of the output of* JLBP(S) *using a fixed $(1 \pm \frac{\varepsilon}{8})$-embedding T on an insertion only stream S. Let $Y_t = \frac{1}{N}\sum_{i=1}^{N} \|X_i^{(t)}\|_2^2$. Suppose that for two given times $1 \le u < v \le m$ the stream satisfies $256C^2F_2^{(u:v)} \le \varepsilon^2 F_2^{(u)}$, where $F_2^{(u:v)} = \sum_{i=1}^{n}(f_i^{(u:v)})^2$ is the second moment of the change in the stream. Then*

$$\Pr\left(|Y_t - F_2^{(t)}| \le \varepsilon F_2^{(t)}, \text{for all } u \le t \le v\right) \ge 1 - 2\delta.$$

Proof. We first write $|Y_t - F_2^{(t)}| \le |Y_t - Y_u| + |Y_u - F_2^{(u)}| + |F_2^{(t)} - F_2^{(u)}|$. It follows from the arguments of AMS and the fact that T is a $(1 \pm \varepsilon/8)$-embedding that, with an appropriate

choice for $N = O(\frac{1}{\delta\varepsilon^2})$, we arrive at

$$\Pr(|Y_u - F_2^{(u)}| \leq \frac{\varepsilon}{4} F_2^{(u)}) \geq 1 - \delta. \tag{2.9}$$

For the third term we have $F_2^{(t)} \geq F_2^{(u)}$ because $t \geq u$ and the stream is insertion only. We can bound the difference with

$$\begin{aligned} F_2^{(t)} = \|f^{(u)} + f^{(u:t)}\|_2^2 &\leq \|f^{(u)}\|_2^2 \left(1 + \frac{\|f^{(u:t)}\|_2}{\|f^u\|_2}\right)^2 \\ &\leq F_2^{(u)}(1 + \frac{\varepsilon}{4}), \end{aligned}$$

where the last inequality follows because $C \geq 2$ and $\varepsilon \leq 1/2$.

For the first term, since $X_i^{(t)}$, $i \in [n]$, are independent d-dimensional Bernoulli process, it follows that

$$X^{(t)} = \frac{1}{\sqrt{N}}((X_1^{(t)})^T, (X_2^{(t)})^T, \ldots, (X_N^{(t)})^T)^T$$

is an Nd-dimensional Bernoulli process. By Lemma 5 and due to the fact that $X^{(t)}$ can be represented as an output of JLBP procedure, the process $X^{(u:t)} = X^{(t)} - X^{(u)}$, is a tame process, so with probability at least $1 - \delta$, for all $u \leq t \leq v$ we have $\|X^{(u:t)}\|_2^2 \leq C^2 \sum_{j=1}^n (f_j^{(u:v)})^2$. Therefore, assuming the inequality inside (2.9),

$$\begin{aligned} Y_t = \|X^{(u)} + X^{(u:t)}\|_2^2 &\leq Y_u \left(1 + \frac{\|X^{(u:t)}\|_2}{\|X^{(u)}\|_2}\right)^2 \\ &\leq Y_u \left(1 + \frac{\sqrt{1+\varepsilon}}{\sqrt{1-\varepsilon}} \frac{\|F^{(u:t)}\|_2}{\|F^{(u)}\|_2}\right)^2 \\ &\leq Y_u \left(1 + \frac{2\varepsilon}{16C}\right)^2 \\ &\leq F_2^{(u)}(1 + \varepsilon/4), \end{aligned}$$

where the last inequality follows because $C \geq 2$ and $\varepsilon \leq 1/2$. The reverse bound $Y_t \geq F_2^{(u)}(1-\varepsilon/4)$ follows similarly upon applying the reverse triangle inequality in place of the triangle inequality. With probability at least $1-2\delta$,

$$\begin{aligned}|Y_t - F_2^{(t)}| &\leq |Y_t - Y_u| + |Y_u - F_2^{(u)}| + |F_2^{(t)} - F_2^{(u)}| \\ &\leq \varepsilon F_2^{(u)} \leq \varepsilon F_2^{(t)}.\end{aligned}$$

□

Theorem 21. *Let S be an insertion only stream and, for $t = 1, 2, \ldots, m$, let $F_2^{(t)} = \sum_{i=1}^{n}(f_i^{(t)})^2$ and let $\hat{F}_2^{(t)}$ be the value that is output by Algorithm 6. Then*

$$P(|\hat{F}_2^{(t)} - F_2^{(t)}| \leq \varepsilon F_2^{(t)}, \textit{for all } t \in [m]) \geq 2/3.$$

It uses $O\left(\frac{1}{\varepsilon^2}\log n\left(\log\log n + \log\frac{1}{\varepsilon}\right)\right)$ bits of space.

Proof. By Lemma 16, the (single) matrix used by all instances of JLBP is a $(1 \pm \varepsilon/3)$-embedding with probability at least 0.99, henceforth assume it is so. Let $q_0 = 0$ and

$$q_i = \max_t \left\{ t \; | F_2^{(t)} \leq (1 + \frac{\varepsilon^2}{256C^2})^i \right\},$$

until $q_K = m$ for some K. Notice that $K = O(\frac{1}{\varepsilon^2}\log n)$. Here, C is the constant from Definition 3.

By definition of q_i and using the fact that $(a-b)^2 \leq a^2 - b^2$ for real numbers $0 \leq b \leq a$ we have $F_2^{(q_i:q_{i+1})} \leq (F_2^{(q_{i+1})} - F_2^{(q_i)}) \leq \frac{\varepsilon^2}{256C^2}F_2^{(q_i)}$.

Applying Lemma 20 with $\delta = 1/10$, we have, for every $r \in [R]$ and $i \geq 0$ that

$$P(|Y_r^{(t)} - F_2^{(t)}| \leq \varepsilon F_2^{(t)}, \text{ for all } q_i \leq t \leq q_{i+1}) \geq 0.8.$$

Thus, by Chernoff bound, the median satisfies

$$P(|\hat{F}_2^{(t)} - F_2^{(t)}| \leq \varepsilon F_2^{(t)}, \text{for all } q_i \leq t \leq q_{i+1}) \geq 1 - e^{-R/12} \geq 1 - \frac{1}{4K},$$

by our choice of $R = 12 \log 4K = O(\log(\varepsilon^{-2} \log n))$. Thus, by a union bound over all of the intervals and the embedding T we get $P(|\hat{F}_2^{(t)} - F_2^{(t)}| \leq \varepsilon F_2^{(t)}, \text{ for all } t \in [m]) \geq \frac{2}{3}$, which completes the proof of correctness.

The algorithm requires, for the matrix T, the JL transform of Kane, Meka, and Nelson [84] with a seed length of $O(\log(n) \log(\frac{1}{\varepsilon} \log n))$ bits, and it takes only $O(\log(n/\varepsilon))$ bits of space to compute any entry of T. The algorithm maintains $NR = O(\varepsilon^{-2} \log(\frac{1}{\varepsilon} \log n))$ instances of JLBP which each requires $O(\log n)$ bits of storage for the sketch and random bits. Therefore, the total storage used by the algorithm is $O(\varepsilon^{-2} \log(n) \log(\frac{1}{\varepsilon} \log n))$. □

2.3 BPTree: an ℓ_2 heavy hitters algorithm using constant memory

This section is based on [32], work done in collaboration with Braverman V., Chestnut S., Woodruff D., Nelson J. and Wang Z.

2.3.1 Introduction

Our contributions. We provide a new one-pass algorithm, BPTree, which in the insertion-only model solves ℓ_2 heavy hitters and achieves the $(\varepsilon, 1/\varepsilon^2)$-tail guarantee. For any constant ε our algorithm only uses a constant $O(1)$ words of memory, which is optimal. This

is the first optimal-space algorithm for ℓ_2 heavy hitters in the insertion-only model for constant ε. The algorithm is described in Theorem 32.

En route to describing BPTree and proving its correctness we describe another result that may be of independent interest. Theorem 22 is a new limited randomness supremum bound for Bernoulli processes. Lemma 30 gives a more advanced analysis of the algorithm of Alon, Matias, and Szegedy (AMS) for approximating $\|f\|_2$ [6], showing that one can achieve the same (additive) error as the AMS algorithm *at all points in the stream*, at the cost of using 8-wise independent random signs rather than 4-wise independent signs. Note that section 2.2 describes an algorithm using $O(\log \log n)$ words that does F_2 tracking in an insertion only stream with a multiplicative error $(1 \pm \varepsilon)$. The multiplicative guarantee is stronger, albeit with more space for the algorithm, but the result can be recovered as a corollary to our additive F_2 tracking theorem, which has a much simplified algorithm and analysis compared to section 2.2.

After some preliminaries, Section 2.3.2 presents both algorithms and their analyses. The description of BPTree is split into three parts. Section 2.3.2 states and proves the chaining inequality. Section 2.3.3 presents the results of some numerical experiments.

Overview of approach. Here we describe the intuition for our heavy hitters algorithm in the case of a single heavy hitter $H \in [n]$ such that $f_H^2 \geq \frac{9}{10}\|f\|_2^2$. The reduction from multiple heavy hitters to this case is standard. Suppose also for this discussion we knew a constant factor approximation to $F_2 := \|f\|_2^2$. Our algorithm and its analysis use several of the techniques developed in section 2.2. We briefly review that algorithm for comparison.

Both CountSieve and BPTree share the same basic building block, which is a subroutine that tries to identify one bit of information about the identity of H. The one-bit subroutine hashes the elements of the stream into two buckets, computes one *Bernoulli process* in each

bucket, and then compares the two values. The Bernoulli process is just the inner product of the frequency vector with a vector of Rademacher (i.e., uniform ± 1) random variables. The hope is that the Bernoulli process in the bucket with H grows faster than the other one, so the larger of the two processes reveals which bucket contains H. In order to prove that the process with H grows faster, section 2.2 introduce a chaining inequality for insertion-only streams that bounds the supremum of the Bernoulli processes over all times. The one-bit subroutine essentially gives us a test that H will pass with probability, say, at least $9/10$ and that any other item passes with probability at most $6/10$. The high-level strategy of both algorithms is to repeat this test sequentially over the stream.

CountSieve uses the one-bit subroutine in a two-part strategy to identify ℓ_2 heavy hitters with $O(\log \log n)$ memory. The two parts are (1) amplify the heavy hitter so $f_H \geq (1 - \frac{1}{\text{poly}(\log n)})\|f\|_2$ and (2) identify H with independent repetitions of the one-bit subroutine. Part (1) winnows the stream from, potentially, n distinct elements to at most $n/\text{poly}(\log n)$ elements. The heavy hitter remains and, furthermore, we get $f_H \geq (1 - \frac{1}{\text{poly}(\log n)})\|f\|_2$ because many of the other elements are removed. CountSieve accomplishes this by running $\Theta(\log \log n)$ independent copies of the one-bit subroutine in parallel, and discarding elements that do not pass a super-majority of the tests. A standard Chernoff bound implies that only $n/2^{O(\log \log n)} = n/\text{poly}(\log n)$ items survive. Part (2) of the strategy identifies $\Theta(\log n)$ 'break-points' where $\|f\|_2$ of the winnowed stream increases by approximately a $(1 + 1/\log n)$ factor from one break-point to the next. Because H already accounts for nearly all of the value of $\|f\|_2$ it is still a heavy hitter within each of the $\Theta(\log n)$ intervals. CountSieve learns one bit of the identity of H on each interval by running the one-bit subroutine. After all $\Theta(\log n)$ intervals are completed the identity of H is known.

BPTree merges the two parts of the above strategy. As above, the algorithm runs a

series of $\Theta(\log n)$ rounds where the goal of each round is to learn one bit of the identity of H. The difference from CountSieve is that BPTree discards more items after every round, then recurses on learning the remaining bits. As the algorithm proceeds, it discards more and more items and H becomes heavier and heavier in the stream. This is reminiscent of work on adaptive compressed sensing [79], but here we are able to do everything in a single pass given the insertion-only property of the stream. Given that the heavy hitter is even heavier, it allows us to weaken our requirement on the two counters at the next level in the recursion tree: we now allow their suprema to deviate even further from their expectation, and this is precisely what saves us from having to worry that one of the $O(\log n)$ Bernoulli processes that we encounter while walking down the tree will have a supremum which is too large and cause us to follow the wrong path. The fact that the heavy hitter is even heavier also allows us to "use up" even fewer updates to the heavy hitter in the next level of the tree, so that overall we have enough updates to the heavy hitter to walk to the bottom of the tree.

Preliminaries. An insertion only stream is a list of items $p_1, \ldots, p_m \in [n]$. The frequency of j at time t is $f_j^{(t)} := \#\{i \leq t \mid p_i = j\}$, $f^{(t)} \in \mathbb{Z}_{\geq 0}^n$ is called the *frequency vector*, we denote $f := f^{(m)}$, $F_2^{(t)} = \sum_{i=1}^n (f_i^{(t)})^2$, $F_2 = \sum_{j=1}^n f_j^2$, and $F_0 = \#\{j \in [n] : f_j > 0\}$. An item $H \in [n]$ is a α-heavy hitter[1] if $f_H^2 \geq \alpha^2 \sum_{j \neq H} f_j^2 = \alpha^2(F_2 - f_H^2)$. For $W \subseteq [n]$, denote by $f^{(t)}(W) \in \mathbb{Z}_{\geq 0}^n$ the frequency vector at time t of the stream restricted to the items in W, that is, a copy of $f^{(t)}$ with the ith coordinate replaced by 0 for every $i \notin W$. We also define $f^{(s:t)}(W) := f^{(t)}(W) - f^{(s)}(W)$ and $F_2(W) = \sum_{j \in W} f_j^2$. In a case where the stream is semi-infinite (it has no defined end) m should be taken to refer to the time of a query of

[1]This definition is in a slightly different form from the one given in the introduction, but this form is more convenient when f_H^2 is very close to F_2.

interest. When no time is specified, quantities like F_2 and f refer to the same query time m.

Our algorithms make use of 2-universal (pairwise independent), 4-wise independent, and 8-wise independent hash functions. We will commonly denote such a function $h : [n] \to [p]$ where p is a prime larger than n, or we may use $h : [n] \to \{0,1\}^R$, which may be taken to mean a function of the first type for some prime $p \in [2^{R-1}, 2^R)$. We use $h(x)_i$ to denote the ith bit, with the first bit most significant (big-endian). A crucial step in our algorithm involves comparing the bits of two values $a, b \in [p]$. Notice that, for any $0 \leq r \leq \lceil \log_2 p \rceil$, we have $a_i = b_i$, for all $1 \leq i \leq r$, if and only if $|a - b| < 2^{\lceil \log_2 p \rceil - r}$. Therefore, the test $a_i = b_i$, for all $1 \leq i \leq r$, can be performed with a constant number of operations.

We will use, as a subroutine, and also compare our algorithm against CountSketch [39]. To understand our results, one needs to know that CountSketch has two parameters, which determine the number of "buckets" and "repetitions" or "rows" in the table it stores. The authors of [39] denote these parameters b and r, respectively. The algorithm selects, independently, r functions $h_1, \ldots, h_r$ from a 2-universal family with domain $[n]$ and range $[b]$ and r functions $\sigma_1, \ldots, \sigma_r$ from a 2-universal family with domain $[n]$ and range $\{-1, 1\}$. CountSketch stores the value $\sum_{j:h_t(j)=i} \sigma(j) f_j$, in cell $(t, i) \in [r] \times [b]$ of the table.

In our algorithm we use the notation $\mathbf{1}(A)$ denote the indicator function of the event A. Namely, $\mathbf{1}(A) = 1$ if A is true and 0 otherwise. We sometimes use $x \lesssim y$ to denote $x = O(y)$.

2.3.2 Algorithm and analysis

We will now describe and analyze the main algorithm, which is broken into several subroutines. The most important subroutine is HH1, Algorithm 7, which finds a single $O(1)$-heavy hitter assuming we have an estimate σ of $\sqrt{F_2}$ such that $\sqrt{F_2} \leq \sigma \leq 2\sqrt{F_2}$. Next is HH2, Algorithm 8, which removes the assumption entailing σ by repeatedly "guessing" values for σ and restarting HH1 as more items arrive. The guessing in HH2 is where we need F_2 tracking. Finally, a well known reduction from finding ε-heavy hitters to finding a single $O(1)$-heavy hitter leads us to the main heavy hitters algorithm BPTree, which is formally described in Theorem 32.

This section is organized as follows. The first subsection gives an overview of the algorithm and its analysis. Section 2.3.2 proves the bound on the expected supremum of the Bernoulli processes used by the algorithm. Section 2.3.2 uses the supremum bound to prove the correctness of the main subroutine HH1. Section 2.3.2 establishes the correctness of F_2 tracking. The subroutine HH2, which makes use of the F_2 tracker, and the complete algorithm BPTree are described and analyzed in Section 2.3.2.

Description of the algorithm. The crux of the problem is to identify one K-heavy hitter for some constant K. HH1, which we will soon describe in detail, accomplishes that task given a suitable approximation σ to $\sqrt{F_2}$. HH2, which removes the assumption of knowing an approximation $\sigma \in [\sqrt{F_2}, 2\sqrt{F_2}]$, is described in Algorithm 8. The reduction from finding all ε-heavy hitters to finding a single K-heavy hitter is standard from the techniques of CountSketch; it is described in Theorem 32.

HH1, Algorithm 7, begins with randomizing the item labels by replacing them with pairwise independent values on $R = \Theta(\log \min\{n, \sigma^2\})$ bits, via the hash function h.

Algorithm 7 Identify a heavy hitter.

procedure HH1(σ, p_1, p_2, ..., p_m)
  $R \leftarrow 3\lfloor \log_2(\min\{n, \sigma^2\} + 1) \rfloor$
  Initialize $b = b_1 b_2 \cdots b_R = 0 \in [2^R]$
  Sample $h : [n] \to \{0,1\}^R \sim$ 2-wise indep. family
  Sample $Z \in \{-1,1\}^n$ 4-wise indep.
  $X_0, X_1 \leftarrow 0$
  $r \leftarrow 1$, $H \leftarrow -1$
  $\beta \leftarrow 3/4$, $c \leftarrow 1/32$
  for $t = 1, 2, \ldots, m$ and $r < R$ **do**
    if $h(p_t)_i = b_i$, for all $i \leq r - 1$ **then**
      $H \leftarrow p_t$
      $X_{h(p_t)_r} \leftarrow X_{h(p_t)_r} + Z_{p_t}$
      if $|X_0 + X_1| \geq c\sigma\beta^r$ **then**
        Record one bit $b_r \leftarrow \mathbf{1}(|X_1| > |X_0|)$
        Refresh $(Z_i)_{i=1}^n$, $X_0, X_1 \leftarrow 0$
        $r \leftarrow r + 1$
      end if
    end if
  end for
  return H
end procedure

Since n and $\sigma^2 \geq F_2$ are both upper bounds for the number of distinct items in the stream, R can be chosen so that every item receives a distinct hash value.

Once the labels are randomized, HH1 proceeds in rounds wherein one bit of the randomized label of the heavy hitter is determined during each round. It completes all of the rounds and outputs the heavy hitter's identity within one pass over the stream. As the rounds proceed, items are discarded from the stream. The remaining items are called *active*. When the algorithm discards an item it will never reconsider it (unless the algorithm is restarted). In each round, it creates two Bernoulli processes X_0 and X_1. In the rth round, X_0 will be determined by the active items whose randomized labels have their rth bit equal to 0, and X_1 determined by those with rth bit 1. Let $f_0^{(t)}, f_1^{(t)} \in \mathbb{Z}_{\geq 0}^n$ be the frequency vectors of the active items in each category, respectively, initialized to

0 at the beginning of the round. Then the Bernoulli processes are $X_0^{(t)} = \langle Z, f_0^{(t)} \rangle$ and $X_1^{(t)} = \langle Z, f_1^{(t)} \rangle$, where Z is a vector of 4-wise independent Rademacher random variables (i.e. the Z_i are marginally uniformly random in $\{-1, 1\}$).

The rth round ends when $|X_0 + X_1| > c\sigma\beta^{r-1}$, for specified[2] constants c and β. At this point, the algorithm compares the values $|X_0|$ and $|X_1|$ and records the identity of the larger one as the rth bit of the candidate heavy hitter. All those items with rth bit corresponding to the smaller counter are discarded (made inactive), and the next round is started.

After R rounds are completed, if there is a heavy hitter then its randomized label will be known with good probability. The identity of the item can be determined by selecting an item in the stream that passes all of the R bit-wise tests, or by inverting the hash function used for the label. If it is a K-heavy hitter, for a sufficiently large $K = O(1)$, then the algorithm will find it with probability at least $2/3$. The algorithm is formally presented in Algorithm 7.

The most important technical component of the analysis is the following theorem, which is proved in Section 2.3.2. Theorem 22 gives us control of the evolution of $|X_0|$ and $|X_1|$ so we can be sure that the larger of the two identifies a bit of H.

Theorem 22. *If $Z \in \{-1, 1\}^n$ is drawn from a 4-wise independent family, $\mathbb{E} \sup_t |\langle f^{(t)}, Z \rangle| < 23 \cdot \|f^{(m)}\|_2$.*

We will use $C_* < 23$ to denote the optimal constant in Theorem 22.

The key idea behind the algorithm is that as we learn bits of the heavy hitter and discard other items, it becomes easier to learn additional bits of the heavy hitter's identity. With fewer items in the stream as the algorithm proceeds, the heavy hitter accounts for a larger and larger fraction of the remaining stream as time goes on. As the heavy hitter

[2] $c = 1/32$ and $\beta = 3/4$ would suffice.

gets heavier the discovery of the bits of its identity can be sped up. When the stream does not contain a heavy hitter this acceleration of the rounds might not happen, though that is not a problem because when there is no heavy hitter the algorithm is not required to return any output. Early rounds will each use a constant fraction of the updates to the heavy hitter, but the algorithm will be able to finish all $R = \Theta(\log n)$ rounds because of the speed-up. The parameter β controls the speed-up of the rounds. Any value of $\beta \in (\frac{1}{2}, 1)$ can be made to work (possibly with an adjustment to c), but the precise value affects the heaviness requirement and the failure probability.

Proof of Theorem 22. Let $Z \in \{-1, 1\}^n$ be random. We are interested in bounding $\mathbb{E} \sup_t |\langle f^{(t)}, Z\rangle|$. It was shown in [31] that if each entry in Z is drawn independently and uniformly from $\{-1, 1\}$, then $\mathbb{E} \sup_t |\langle f^{(t)}, Z\rangle| \lesssim \|f^{(m)}\|_2$. We show that this inequality still holds if the entries of Z are drawn from a 4-wise independent family, which is used both in our analyses of HH1 and our F_2 tracking algorithm. The following lemma is implied by [67].

Lemma 23 (Khintchine's inequality)**.** *Let $Z \in \{-1, 1\}^n$ be chosen uniformly at random, and $x \in \mathbb{R}^n$ a fixed vector. Then for any even integer p, $\mathbb{E} \langle Z, x\rangle^p \leq \sqrt{p}^p \cdot \|x\|_2^p$.*

of Theorem 22. To simplify notation, we first normalize the vectors in $\{f^{(0)} = 0, f^{(1)}, \ldots, f^{(m)}\}$ (i.e., divide by $\|f^{(m)}\|_2$). Denote the set of these normalized vectors by $T = \{v_0, \ldots, v_m\}$, where $\|v_m\|_2 = 1$. Recall that an ε-net of some set of points T under some metric d is a set of point T' such that for each $t \in T$, there exists some $t' \in T'$ such that $d(t, t') \leq \varepsilon$. For every $k \in \mathbb{N}$, we can find a $1/2^k$-net of T in ℓ_2 with size $|S_k| \leq 2^{2k}$ by a greedy construction as follows.

To construct an ε-net for T, we first take v_0, then choose the smallest i such that $\|v_i - v_0\|_2 > \varepsilon$, and so on. To prove the number of elements selected is upper bounded by

$1/\varepsilon^2$, let $u_0, u_1, u_2, \ldots, u_t$ denote the vectors we selected accordingly, and note that the second moments of $u_1 - u_0, u_2 - u_1, \ldots, u_t - u_{t-1}$ are greater than ε^2. Because the vectors $u_i - u_{i-1}$ have non-negative coordinates, $\|u_t\|_2^2$ is lower bounded by the summation of these moments, while on the other hand $\|u_t\|_2^2 \leq 1$. Hence the net is of size at most $1/\varepsilon^2$.

Let S be a set of vectors. Let $Z \in \{-1, 1\}^n$ be drawn from a p-wise independent family, where p is an even integer. By Markov and Khintchine's inequality,

$$\begin{aligned}\Pr(|\langle x, Z\rangle| > \lambda \cdot |S|^{1/p} \cdot \|x\|_2) &< \frac{\mathbb{E}|\langle x, Z\rangle|^p}{\lambda^p \cdot |S| \cdot \|x\|_2^p} \\ &< \frac{1}{|S|} \cdot \left(\frac{\sqrt{p}}{\lambda}\right)^p.\end{aligned}$$

Hence,

$$\begin{aligned}\mathbb{E} \sup_{x \in S} |\langle x, Z\rangle| &= \int_0^\infty \Pr(\sup_{x \in S} |\langle x, Z\rangle| > u) du \\ &= |S|^{1/p} \cdot \sup_{x \in S} \|x\|_2 \cdot \\ &\qquad \int_0^\infty \Pr(\sup_{x \in S} |\langle x, Z\rangle| > \lambda \cdot |S|^{1/p} \cdot \sup_{x \in S} \|x\|_2) d\lambda \\ &< |S|^{1/p} \cdot \sup_{x \in S} \|x\|_2 \cdot \left(\sqrt{p} + \int_{\sqrt{p}}^\infty \left(\frac{\sqrt{p}}{\lambda}\right)^p d\lambda\right) \\ &\qquad \text{(union bound)} \\ &= |S|^{1/p} \cdot \sup_{x \in S} \|x\|_2 \cdot \sqrt{p} \cdot \left(1 + \frac{1}{p-1}\right)\end{aligned}$$

Now we apply a similar chaining argument as in the proof of Dudley's inequality (cf. [50]). For $x \in T$, let x^k denote the closest point to x in S_k. Then $\|x^k - x^{k-1}\|_2 \leq \|x^k - x\|_2 + \|x - x^{k-1}\|_2 \leq (1/2^k) + (1/2^{k-1})$. Note that if for some $x \in T$ one has that $x_k = v_t$ is the closest vector to x in T_k (under ℓ_2), then the closest vector x_{k-1} to x in

T_{k-1} must either be the frequency vector $v_{t'}$ in T_{k-1} such that t' is the smallest timestamp after t of a vector in T_{k-1}, or the largest timestamp before t in T_{k-1}. Thus the size of $\{x^k - x^{k-1} | x \in T\}$ is upper bounded by $2|S_k| \leq 2^{2k+1}$, implying for $p = 4$

$$\begin{aligned}\mathbb{E} \sup_{x \in T} |\langle x, Z \rangle| &\leq \sum_{k=1}^{\infty} \mathbb{E} \sup |\langle x^k - x^{k-1}, Z \rangle| \\ &< 3 \cdot 2^{1/p} \sqrt{p} \left(1 + \frac{1}{p-1}\right) \sum_{k=1}^{\infty} (2^{2k})^{1/p} \cdot (1/2^k) \\ &< 23.\end{aligned}$$

□

Identifying a single heavy hitter given an approximation to F_2. This section analyzes the subroutine HH1, which is formally presented in Algorithm 7. The goal of this section is to prove Lemma 27, the correctness of HH1. We use $H \in [n]$ to stand for the identity of the most frequent item in the stream. It is not assumed to be a heavy hitter unless explicitly stated.

The first step of HH1 is to choose a hash function $h : [n] \to \{0,1\}^R$, for $R = O(\log n)$, that relabels the universe of items $[n]$. For each $r \geq 0$, let

$$\mathcal{H}_r \coloneqq \{i \in [n] \setminus \{H\} \mid h(i)_k = h(H)_k \text{ for all } 1 \leq k \leq r\},$$

and let $\bar{\mathcal{H}}_r \coloneqq \mathcal{H}_{r-1} \setminus \mathcal{H}_r$, with $\bar{\mathcal{H}}_0 = \emptyset$ for convenience. By definition, $\mathcal{H}_R \subseteq \mathcal{H}_{R-1} \subseteq \cdots \subseteq \mathcal{H}_0 = [n] \setminus \{H\}$, and, in round $r \in [R]$, our hope is that the active items are those in $\mathcal{H}_{r-1}$.

The point of randomizing the labels is as a sort of "load balancing" among the item labels. The idea is that each bit of $h(H)$ partitions the active items into two roughly equal

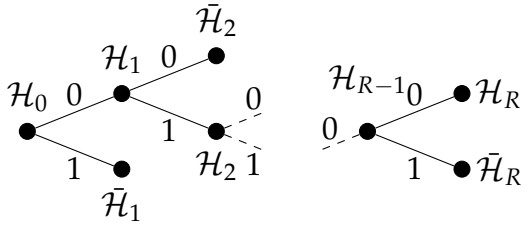


FIGURE 2.1: In this example of the execution of HH1, the randomized label $h(H)$ of the heavy hitter H begins with 01 and ends with 00. Each node in the tree corresponds to a round of HH1, which must follow the path from $\mathcal{H}_0$ to $\mathcal{H}_R$ for the output to be correct.

sized parts, i.e. $|\mathcal{H}_r| \approx |\bar{\mathcal{H}}_r|$ in every round r. This leads HH1 to discard roughly half of the active items after each round, allowing us to make progress on learning the (hashed) identity of a heavy hitter. We will make use of the randomized labels in the next section, within the proof of Lemma 24.

For h, we recommend choosing a prime $p \approx \min\{n, F_2\}^2$ and assigning the labels $h(i) = a_0 + a_1 i \bmod p$, for a_0 and a_1 randomly chosen in $\{0, 1, \ldots, p-1\}$ and $a_1 \neq 0$. We can always achieve this with $R = 3\log_2(\min\{n, F_2\} + 1)$, which is convenient for the upcoming analysis. This distribution on h is known to be a 2-wise independent family [35]. Note computing $h(i)$ for any i takes $O(1)$ time. It is also simple to invert: namely $x = a_1^{-1}(h(x) - a_0) \bmod p$, so x can be computed quickly from $h(x)$ when $p > n$. Inverting requires computing the inverse of a_1 modulo p, which takes $O(\log \min\{n, F_2\})$ time via repeated squaring, however this computation can be done once, for example during initialization of the algorithm, and the result stored for all subsequent queries. Thus, the time to compute $a_1^{-1} \bmod p$ is negligible in comparison to reading the stream.

After randomizing the labels HH1 proceeds with the series of R rounds to identify the (randomized) label $h(H)$ of the heavy hitter H. The sequence of rounds is depicted in Figure 2.1. Each node in the tree corresponds to one round of HH1. The algorithm

traverses the tree from left to right as the rounds progress. Correctness of HH1 means it traverses the path from $\mathcal{H}_0$ to $\mathcal{H}_R$. The 0/1 labels on the path leading to $\mathcal{H}_R$ are the bits of $h(H)$, and when $R = O(\log n)$ is sufficiently large we get $\mathcal{H}_R = \{H\}$ with high probability. Thus the algorithm correctly identifies $h(H)$, from which it can determine H with the method discussed earlier.

Now let us focus on one round and suppose that H is a K-heavy hitter, for some large constant K. Suppose the algorithm is in round $r \geq 1$, and recall that the goal of the round is to learn the rth bit of $h(H)$. Our hope is that the active items are those in $\mathcal{H}_{r-1}$ (otherwise the algorithm will fail), which means that the algorithm has correctly discovered the first $r-1$ bits of $h(H)$. The general idea is that HH1 partitions $\mathcal{H}_{r-1} \cup \{H\}$ into $\mathcal{H}_r \cup \{H\}$ and $\bar{\mathcal{H}}_r$, creates a Bernoulli process for each of those sets of items, and compares the values of the two Bernoulli processes to discern $h(H)_r$. Suppose that the active items are indeed $\mathcal{H}_{r-1}$ and, for the sake of discussion, that the rth bit of the heavy hitter's label is $h(H)_r = 0$. Then the Bernoulli processes X_0 and X_1, defined in Algorithm 7, have the following form

$$X_0(t) = Z_H f_H^{(s:t)} + \sum_{i \in \mathcal{H}_r} Z_i f_i^{(s:t)}, \quad X_1(t) = \sum_{i \in \bar{\mathcal{H}}_r} Z_i f_i^{(s:t)},$$

where $s \leq m$ is the time of the last update to round $r-1$ and t is the current time. To simplify things a little bit we adopt the notation $f(S)$ for the frequency vector restricted to only items in S. For example, in the equations above become $X_0 = Z_H f_H^{(s:t)} + \langle Z, f^{(s:t)}(\mathcal{H}_r)\rangle$ and $X_1 = \langle Z, f^{(s:t)}(\bar{\mathcal{H}}_r)\rangle$.

The round is a success if $|X_0| > |X_1|$ (because we assumed $h(H)_r = 0$) at the first time when $|X_0 + X_1| > c\sigma\beta^r$. When that threshold is crossed, we must have $|X_0| \geq c\sigma\beta^r/2$, $|X_1| \geq c\sigma\beta^r/2$, or both. The way we will ensure that the round is a success is by

establishing the following bound on the Bernoulli process X_1: $|X_1| = |\langle Z, f^{(s:t)}(\bar{\mathcal{H}}_r)\rangle| < c\sigma\beta^r/2$, for all times $t \geq s$. Of course, the round does not end until the threshold is crossed, so we will also establish a bound on the complementary Bernoulli process $|X_0 - Z_H f_H^{(s:t)}| = |\langle Z, f^{(s:t)}(\bar{\mathcal{H}}_r)\rangle| < c\sigma\beta^r/2$, at all times $t \geq s$. When this holds we must have $|X_0 + X_1| > c\sigma\beta^r$ no later than the first time t where $f_H^{(s:t)} \geq 2c\sigma\beta^r$, so the round ends after at most $2c\sigma\beta^r$ updates to H. In total over all of the rounds this uses up no more than $\sum_{r\geq 0} 2c\sigma\beta^r < f_H$ updates to H, where we have used $\sigma < 2\sqrt{F_2} < 3f_H$ by our assumption that H is a heavy hitter. In truth, both of those inequalities fail to hold with some probability, but the failure probability is $O(1/\beta^r 2^{r/2})$ so the probability that the algorithm succeeds will turn out to be $1 - \sum_{r\geq 0} O(1/\beta^r 2^{r/2}) > 2/3$.

The next lemma establishes the control on the Bernoulli processes that we have just described (compare the events E_i with the previous paragraph). We will use it later with $K \approx \beta^r$ so that, while the rounds progress, the upper bounds on the process maxima and the failure probabilities both decrease geometrically as desired. This means that the lengths of the rounds decreases geometrically and the latter means that a union bound suffices to guarantee that all of the events E_i occur. In our notation $F_2 - f_H^2 = F_2(\mathcal{H}_0)$.

Lemma 24. *For any $r \in \{0, 1, \ldots, R\}$ and $K > 0$, the events*

$$E_{2r-1} := \left\{ \max_{s,t\leq m} |\left\langle Z, f^{(s:t)}(\bar{\mathcal{H}}_r)\right\rangle| \leq KF_2(\mathcal{H}_0)^{1/2} \right\}$$

and

$$E_{2r} := \left\{ \max_{s,t\leq m} |\left\langle Z, f^{(s:t)}(\mathcal{H}_r)\right\rangle| \leq KF_2(\mathcal{H}_0)^{1/2} \right\}$$

have respective probabilities at least $1 - \frac{4C_*}{K2^{r/2}}$ of occurring, where $C_* < 23$ is the constant from Theorem 22.

Proof. By the Law of Total Probability and Theorem 22 with Markov's Inequality we have

$$\begin{aligned}
&\Pr\left(\max_{t\leq m}|\langle Z, f^{(t)}(\mathcal{H}_r)\rangle| \geq \frac{1}{2}KF_2(\mathcal{H}_0)^{1/2}\right) \\
&= \mathbb{E}\left\{\Pr\left(\max_{t\leq m}|\langle Z, f^{(t)}(\mathcal{H}_r)\rangle| \geq \frac{1}{2}KF_2(\mathcal{H}_0)^{1/2}\middle|\mathcal{H}_r\right)\right\} \\
&\leq \mathbb{E}\left\{\frac{2C_*F_2(\mathcal{H}_r)^{1/2}}{KF_2(\mathcal{H}_0)^{1/2}}\right\} \leq \frac{2C_*F_2(\mathcal{H}_0)^{1/2}}{KF_2(\mathcal{H}_0)^{1/2}2^{r/2}},
\end{aligned}$$

where the last inequality is Jensen's. The same holds if $\mathcal{H}_r$ is replaced by $\bar{\mathcal{H}}_r$.

Applying the triangle inequality to get

$$|\langle Z, f^{(s:t)}(\mathcal{H}_r)\rangle| \leq |\langle Z, f^{(s)}(\mathcal{H}_r)\rangle| + |\langle Z, f^{(t)}(\mathcal{H}_r)\rangle|$$

we then find $P(E_{2r}) \geq 1 - \frac{4C_*}{K2^{r/2}}$. A similar argument proves $P(E_{2r-1}) \geq 1 - \frac{4C_*}{K2^{r/2}}$. □

From here the strategy to prove the correctness of HH1 is to inductively use Lemma 24 to bound the success of each round. The correctness of HH1, Lemma 27, follows directly from Lemma 25.

Let U be the event $\{h(j) \neq h(H) \text{ for all } j \neq H, f_j > 0\}$ which has, by pairwise independence, probability $\Pr(U) \geq 1 - F_0 2^{-R} \geq 1 - \frac{1}{\min\{n,F_2\}^2}$, recalling that $F_0 \leq \min\{n, F_2\}$ is the number of distinct items appearing before time m. The next lemma is the main proof of correctness for our algorithm.

Lemma 25. *Let $K' \geq 128$, $c = 1/32$, and $\beta = 3/4$. If $2K'C_*\sqrt{F_2(\mathcal{H}_0)} \leq \sigma \leq 2\sqrt{2}f_H$ and $f_H > 2K'C_*\sqrt{F_2(\mathcal{H}_0)}$ then, with probability at least $1 - \frac{1}{\min\{F_2,n\}^2} - \frac{8}{K'c(\sqrt{2}\beta-1)}$ the algorithm HH1 returns H.*

Proof. Recall that H is *active* during round r if it happens that $h(H)_i = b_i$, for all $1 \leq i \leq r-1$, which implies that updates from H are not discarded by the algorithm during round

r. Let $K = K(r) = K'cC_*\beta^r$ in Lemma 24, and let E be the event that U and $\cap_{r=1}^{2R} E_r$ both occur. We prove by induction on r that if E occurs then either $b_r = h(H)_r$, for all $r \in [R]$ or H is the only item appearing in the stream. In either case, the algorithm correctly outputs H, where in the former case it follows because $E \subseteq U$.

Let $r \geq 1$ be such that H is still active in round r, i.e. $b_i = h(H)_i$ for all $1 \leq i \leq r-1$. Note that all items are active in round 1. Since H is active, the remaining active items are exactly $\mathcal{H}_{r-1} = \mathcal{H}_r \cup \bar{\mathcal{H}}_r$. Let t_r denote the time of the last update received during the rth round, and define $t_0 = 0$. At time $t_{r-1} \leq t < t_r$ we have

$$\begin{aligned} c\sigma\beta^r &> |X_0 + X_1| \\ &= |\langle Z, f^{(t_{r-1}:t)}(\mathcal{H}_r \cup \bar{\mathcal{H}}_r)\rangle + Z_H f_H^{(t_{r-1}:t)}| \\ &\geq f_H^{(t_{r-1}:t)} - K(r-1)F_2(\mathcal{H}_0)^{1/2}, \end{aligned}$$

where the last inequality follows from the definition of $E_{2(r-1)}$. Rearranging and using the assumed lower bound on σ, we get the bound

$$K(r-1)F_2(\mathcal{H}_0)^{1/2} \leq \frac{K(r-1)}{2K'C_*}\sigma = \frac{1}{2}c\sigma\beta^{r-1}. \tag{2.10}$$

Therefore, by rearranging we see $f_H^{(t_{r-1}:t_r)} \leq 1 + f_H^{(t_{r-1}:t)} < 1 + \frac{3}{2}c\sigma\beta^{r-1}$. That implies $f_H^{(t_r)} = \sum_{k=1}^{r} f_H^{(t_{k-1}:t_k)} < r + \frac{3}{2}c\sigma\sum_{k=1}^{r}\beta^{k-1} \leq r + \frac{3\sqrt{2}c}{1-\beta}f_H$. Thus, if $f_H - \frac{3\sqrt{2}c}{1-\beta}f_H > R$ then round $r \leq R$ is guaranteed to be completed and a further update to H appears after the round. Suppose, that is not the case, and rather $R \geq f_H - \frac{3\sqrt{2}c}{1-\beta}f_H \geq \frac{1}{2}f_H$, where the last inequality follows from our choices $\beta = 3/4$ and $c = 1/32$. Then, by the definition of R, $9(1 + \log_2 8f_H^2)^2 \geq R^2 \geq \frac{1}{4}f_H^2$. One can check that this inequality implies that $f_H \leq 104$. Now $K' \geq 128$ and the heaviness requirement of H implies that $F_2(\mathcal{H}_0) = 0$. Therefore, H

is the only item in the stream, and, in that case the algorithm will always correctly output H.

Furthermore, at the end of round r, $|X_0 + X_1| \geq c\sigma\beta^r$, so we have must have either $|X_0| \geq c\sigma\beta^r/2$ or $|X_1| \geq c\sigma\beta^r/2$. Both cannot occur for the following reason. The events E_{2r-1} and E_{2r} occur, recall these govern the non-heavy items contributions to X_0 and X_1, and these events, with the inequality (2.10), imply

$$|\langle Z, f^{(t_{r-1}:t_r)}(\mathcal{H}_r)\rangle| \leq K(r)F_2(\mathcal{H}_0)^{1/2} < \frac{1}{2}c\sigma\beta^r$$

and the same holds for $\bar{\mathcal{H}}_r$. Therefore, the Bernoulli process not including H has its value smaller than $c\sigma\beta^r/2$, and the other, larger process identifies the bit $h(H)_r$. By induction, the algorithm completes every round $r = 1, 2, \ldots, R$ and there is at least one update to H after round R. This proves the correctness of the algorithm assuming the event E occurs.

It remains to compute the probability of E. Lemma 24 provides the bound

$$\begin{aligned}
\Pr(U \text{ and } \cap_{i=1}^{2R} E_i) &\geq 1 - \frac{1}{\min\{n, F_2\}^2} - \sum_{r=0}^{R} \frac{8C_*}{K(r)2^{r/2}} \\
&= 1 - \frac{1}{\min\{n, F_2\}^2} - \sum_{r=0}^{R} \frac{8}{K'c\beta^r 2^{r/2}} \\
&> 1 - \frac{1}{\min\{n, F_2\}^2} - \frac{8}{K'c(\sqrt{2}\beta - 1)}.
\end{aligned}$$

□

Proposition 26. *Let $\alpha \geq 1$. If $F_2^{1/2} \leq \sigma \leq 2F_2^{1/2}$ and $f_H \geq \alpha\sqrt{F_2(\mathcal{H}_0)}$ then $\alpha\sqrt{F_2(\mathcal{H}_0)} \leq \sigma \leq 2\sqrt{2}f_H$.*

Proof. $\sigma^2 \geq F_2 \geq f_H^2 = F_2 - F_2(\mathcal{H}_0) \geq (1 - \frac{1}{1+\alpha^2})F_2 \geq \frac{1}{8}\sigma^2$. □

Lemma 27 (HH1 Correctness)**.** *There is a constant K such that if H is a K-heavy hitter and $\sqrt{F_2} \leq \sigma \leq 2\sqrt{F_2}$, then with probability at least $2/3$ algorithm HH1 returns H. HH1 uses $O(1)$ words of storage.*

Proof. The Lemma follows immediately from Proposition 26 and Lemma 25 by setting $K' = 2^{13}$, which allows $K = 2^{14}C_* \leq 380,000$. □

F_2 Tracking. This section proves that the AMS algorithm with 8-wise, rather than 4-wise, independent random signs has an additive εF_2 approximation guarantee at all points in the stream. We will use the tracking to "guess" a good value of σ for input to HH1, but, because the AMS algorithm is a fundamental streaming primitive, it is of independent interest from the BPTree algorithm. The following theorem is a direct consequence of Lemma 30 and [6].

Theorem 28. *Let $0 < \varepsilon < 1$. There is a streaming algorithm that outputs at each time t a value $\hat{F}_2^{(t)}$ such that $\Pr(|\hat{F}_2^{(t)} - F_2^{(t)}| \leq \varepsilon F_2$, for all $0 \leq t \leq m) \geq 1 - \delta$. The algorithm use $O(\frac{1}{\varepsilon^2}\log\frac{1}{\delta})$ words of storage and has $O(\frac{1}{\varepsilon^2}\log\frac{1}{\delta})$ update time.*

Let us remark that it follows from Theorem 28 and a union bound that one can achieve a $(1 \pm \varepsilon)$ *multiplicative* approximation to F_2 at all points in the stream using $O(\varepsilon^{-2}\log\log m)$ words. The proof that this works breaks the stream into $O(\log m)$ intervals of where the change in F_2 doubles.

In its original form [6], the AMS sketch is a product of the form Πf, where Π is a random $k \times n$ matrix chosen with each row independently composed of 4-wise independent ± 1 random variables. The sketch uses $k = \Theta(1/\varepsilon^2)$ rows to achieve a $(1 \pm \varepsilon)$-approximation with constant probability. We show that the AMS sketch with $k \simeq 1/\varepsilon^2$ rows and 8-wise independent entries provides ℓ_2-tracking with additive error $\varepsilon\|f\|_2$ at all

times. We define $v_t = f^{(t)}/\|f^{(m)}\|_2$ so $\|v_t\|_2 \leq 1$, for all $t \geq 0$, and $\|v_m\|_2 = 1$. Define $T = \{v_0, v_1, \ldots, v_m\}$. We use $\|A\|$ to denote the spectral norm of A, which is equal to its largest singular value, and $\|A\|_F$ for the Frobenius norm, which is the Euclidean length of A when viewed as a vector. Our proof makes use of the following moment bound for quadratic forms. Recall that given a metric space (X, d) and $\varepsilon > 0$, an *ε-net* of X is a subset $N \subseteq X$ such that $d(x, N) = \inf_{y \in N} d(x, y) \leq \varepsilon$ for all $x \in X$.

Theorem 29 (Hanson-Wright [70])**.** *For $B \in \mathbb{R}^{n \times n}$ symmetric with (Z_i) uniformly random in $\{-1, 1\}^n$, for all $p \geq 1$, $\|Z^T BZ - \mathbb{E}Z^T BZ\|_p \lesssim \sqrt{p}\|B\|_F + p\|B\|$.*

Observe the sketch can be written $\Pi x = A_x Z$, where

$$A_x := \frac{1}{\sqrt{k}} \sum_{i=1}^{k} \sum_{j=1}^{n} x_j^i e_i \otimes e_{n(i-1)+j}$$
$$= \frac{1}{\sqrt{k}} \begin{bmatrix} -x- & 0 & \cdots & 0 \\ 0 & -x- & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & -x- \end{bmatrix}.$$

We are thus interested in bounding $\mathbb{E}_Z \sup_{x \in T} |Z^T B_x Z - \mathbb{E}Z^T B_x Z|$,for $B_x = A_x^T A_x$. Note for any $\|x\|_2, \|y\|_2 \leq 1$,

$$\|xx^T - yy^T\|_F \leq 4\|x - y\|_2. \tag{2.11}$$

Lemma 30 (F_2 tracking)**.** *If $k \gtrsim 1/\varepsilon^2$ and $Z \in \{-1, +1\}^{kn}$ are 8-wise independent then*

$$\mathbb{E} \sup_t \left| \|\Pi f^{(t)}\|_2^2 - \|f^{(t)}\|_2^2 \right| \leq \varepsilon \|f\|_2^2.$$

Proof. Let A_x, $x \in T$, as defined above and $B_x = A_x^T A_x$. By (2.11),

$$\|B_x - B_y\|_F \leq \frac{4}{\sqrt{k}}\|x - y\|_2,$$

for all $x, y \in T$. In particular,

$$\sup_{x \in T} \|B_x\| \leq \sup_{x \in T} \|B_x\|_F \leq 1/\sqrt{k}.$$

Let T_ℓ be a $(1/2^\ell)$-net of T under ℓ_2; we know we can take $|T_\ell| \leq 4^\ell$. $\mathcal{B}_\ell = \{B_x : x \in T_\ell\}$ is a $1/\sqrt{k}2^\ell$-net under $\|\cdot\|$ and also under $\|\cdot\|_F$. For $x \in T$, let $x_\ell \in T_\ell$ denote the closest element in T_ℓ, under ℓ_2. Then we can write $B_x = B_{x_0} + \sum_{\ell=1}^{\infty} \Delta_{x_\ell}$, where $\Delta_{x_\ell} = B_{x_\ell} - B_{x_{\ell-1}}$. For brevity, we will also define $\gamma(A) \coloneqq |Z^T A Z^T - \mathbb{E} Z^T A Z^T|$. Thus if the (Z_i) are $2p$-wise independent

$$\begin{aligned}
\mathbb{E} \sup_{x \in T} \gamma(B_x) &\leq \mathbb{E} \sup_{x \in T} \gamma(B_{x_0}) + \mathbb{E} \sup_{x \in T} \sum_{\ell=1}^{\infty} \gamma(\Delta_{x_\ell}) \\
&\lesssim \frac{p}{\sqrt{k}} + \sum_{\ell=1}^{\infty} \mathbb{E} \sup_{x \in T} \gamma(\Delta_{x_\ell}) \qquad (2.12)
\end{aligned}$$

If $A \in \mathbb{R}^{n \times n}$ is symmetric, then by the Hanson-Wright Inequality

$$\Pr(\gamma(A) > \lambda \cdot S^{1/p}) < \frac{1}{S} \cdot \left[\left(\frac{C\sqrt{p}\|A\|_F}{\lambda} \right)^p + \left(\frac{Cp\|A\|}{\lambda} \right)^p \right]$$

for some constant $C > 0$. Thus if $\mathcal{A}$ is a collection of such matrices, $|\mathcal{A}| = S$, choosing $u^* = C(\sqrt{p} \cdot \sup_{A\in\mathcal{A}} \|A\|_F + p \cdot \sup_{A\in\mathcal{A}} \|A\|)$

$$\begin{aligned}
\mathbb{E} \sup_{A\in\mathcal{A}} \gamma(A) &= \int_0^\infty \Pr(\sup_{A\in\mathcal{A}} \gamma(A) > u) du \\
&= S^{1/p} \cdot \int_0^\infty \Pr(\sup_{A\in\mathcal{A}} \gamma(A) > \lambda \cdot S^{1/p}) d\lambda \\
&= S^{1/p}(u^* + \int_{u^*}^\infty \Pr(\sup_{A\in\mathcal{A}} \gamma(A) > \lambda \cdot S^{1/p}) d\lambda) \\
&\lesssim S^{1/p}(\sqrt{p} \cdot \sup_{A\in\mathcal{A}} \|A\|_F + p \cdot \sup_{A\in\mathcal{A}} \|A\|) \qquad (2.13)
\end{aligned}$$

Now by applying (2.13) to (2.12) repeatedly with

$$\mathcal{A} = \mathcal{A}_\ell = \{B_{x_\ell} - B_{x_{\ell-1}} : x \in T\},$$

noting $\sup_{A\in\mathcal{A}_\ell} \|A\| \le \sup_{A\in\mathcal{A}_\ell} \|A\|_F \le 1/\sqrt{k}2^\ell$ and $|\mathcal{A}_\ell| \le 2|T_\ell| \le 2 \cdot 2^{2\ell}$,

$$\mathbb{E} \sup_{x\in T} \gamma(B_x) \lesssim \frac{p}{\sqrt{k}} + \sum_{\ell=1}^\infty \frac{p 2^{2^{\frac{\ell}{p}} - \ell}}{\sqrt{k}} \lesssim \frac{p}{\sqrt{k}}$$

for $p \ge 4$. Thus it suffices for the entries of Z to be $2p$-wise independent, i.e. 8-wise independent. □

The complete heavy hitters algorithm We will now describe HH2, formally Algorithm 8, which is an algorithm that removes the assumption on σ needed by HH1. It is followed by the complete algorithm BPTree. The step in HH2 that guesses an approximation σ for $\sqrt{F_2}$ works as follows. We construct the estimator $\hat{F}_2$ of the previous section to (approximately) track F_2. HH2 starts a new instance of HH1 each time the estimate $\hat{F}_2$ crosses a power of 2. Each new instance is initialized with the current estimate of $\sqrt{F_2}$ as the value for σ,

Algorithm 8 Identify a heavy hitter by guessing σ.

procedure HH2($p_1, p_2, \ldots, p_m$)
  Run $\hat{F}_2$ from Theorem 28 with $\varepsilon = 1/100$ and $\delta = 1/20$
  Start HH1 ($1, p_1, \ldots, p_m$)
  Let $t_0 = 1$ and $t_k = \min\{t \mid \hat{F}_2^{(t)} \geq 2^k\}$, for $k \geq 1$.
  for each time t_k **do**
    Start HH1$((\hat{F}_2^{(t_k)})^{1/2}, p_{t_k}, p_{t_k+1}, \ldots p_m)$
    Let H_k denote its output if it completes
    Discard H_{k-2} and the copy of HH1 started at t_{k-2}
  end for
  return H_{k-1}
end procedure

but HH2 maintains only the two most recent copies of HH1. Thus, even though, overall, it may instantiate $\Omega(\log n)$ copies of HH1 at most two will running concurrently and the total storage remains $O(1)$ words. At least one of the thresholds will be the "right" one, in the sense that HH1 gets initialized with σ in the interval $[\sqrt{F_2}, 2\sqrt{F_2}]$, so we expect the corresponding instance of HH1 to identify the heavy hitter, if one exists.

The scheme could fail if $\hat{F}_2$ is wildly inaccurate at some points in the stream, for example if $\hat{F}_2$ ever grows too large then the algorithm could discard every instance of HH1 that was correctly initialized. But, Theorem 28 guarantees that it fails only with small probability.

We begin by proving the correctness of HH2 in Lemma 31 and then complete the description and correctness of BPTree in Theorem 32.

Lemma 31. *There exists a constant $K > 0$ and a 1-pass streaming algorithm HH2, Algorithm 8, such that if the stream contains a K-heavy hitter then with probability at least 0.6 HH2 returns the identity of the heavy hitter. The algorithm uses $O(1)$ words of memory and $O(1)$ update time.*

Proof. The space and update time bounds are immediate from the description of the algorithm. The success probability follows by a union bound over the failure probabilities in

Lemma 27 and Theorem 28, which are $1/3$ and $\delta = 0.05$ respectively. It remains to prove that there is a constant K such that conditionally given the success of the F_2 estimator, the hypotheses of Lemma 27 are satisfied by the penultimate instance of HH1 by HH2.

Let K' denote the constant from Lemma 27 and set $K = 12K'$, so if H is a K-heavy hitter then for any $\alpha > 2/K$ and in any interval $(t, t']$ where $(F_2^{(t')})^{1/2} - (F_2^{(t)})^{1/2} \geq \alpha\sqrt{F_2}$ we will have

$$f_H^{(t:t')} + F_2(\mathcal{H}_0)^{1/2} \geq \|f^{(t:t')}\|_2 \geq \|f^{(t')}\|_2 - \|f^{(t)}\|_2 \geq \alpha\sqrt{F_2}.$$

If follows with in the stream $p_t, p_{t+1}, \ldots, p_{t'}$ the heaviness of H is at least

$$\frac{f_H^{(t:t')}}{F_2^{(t:t')}(\mathcal{H}_0)^{1/2}} \geq \frac{\alpha\sqrt{F_2} - \sqrt{F_2(\mathcal{H}_0)}}{\sqrt{F_2(\mathcal{H}_0)}} \geq K\alpha - 1 \geq \frac{K\alpha}{2}. \tag{2.14}$$

Of course, if $F_2^{(t:t')}(\mathcal{H}_0) = 0$ the same heaviness holds.

Let k be the last iteration of HH2. By the definition of t_k, we have $(\hat{F}_2^{(t_{k-1})})^{1/2} \geq \frac{1}{4}(\hat{F}_2)^{1/2} \geq \frac{1}{4}\sqrt{(1-\varepsilon)F_2}$. Similar calculations show that there exists a time $t_* > t_{k-1}$ such that $(F_2^{(t_*)})^{1/2} - (F_2^{(t_{k-1})})^{1/2} \geq \frac{1}{6}F_2$ and $\|f^{t_{k-1}:t_*}\|_2 \leq (\hat{F}_2^{(t_{k-1})})^{1/2} \leq 2\|f^{t_{k-1}:t_*}\|_2$. In particular, the second pair of inequalities implies that $\hat{F}_2^{(t_k-1)}$ is a good "guess" for σ on the interval $(t_{k-1}, t^*]$. We claim H is a K' heavy hitter on that interval, too. Indeed, because of (2.14), with $\alpha = 1/6$, we get that H is a K'-heavy hitter on the interval $(t_{k-1}, t^*]$. This proves that the hypotheses of Lemma 27 are satisfied for the stream $p_{t_{k-1}+1}, \ldots, p_{t^*}$. It follows that from Lemma 27 that HH1 correctly identifies $H_{k-1} = H$ on that substream and the remaining updates in the interval $(t_*, t_m]$ do not affect the outcome. □

A now standard reduction from ε-heavy hitters to $O(1)$-heavy hitters gives the following theorem. The next section describes an implementation that is more efficient in

practice.

Theorem 32. *For any $\varepsilon > 0$ there is 1-pass streaming algorithm BPTree that, with probability at least $(1 - \delta)$, returns a set of $\frac{\varepsilon}{2}$-heavy hitters containing every ε-heavy hitter and an approximate frequency for every item returned satisfying the $(\varepsilon, 1/\varepsilon^2)$-tail guarantee. The algorithm uses $O(\frac{1}{\varepsilon^2} \log \frac{1}{\varepsilon\delta})$ words of space and has $O(\log \frac{1}{\varepsilon\delta})$ update and $O(\varepsilon^{-2} \log \frac{1}{\varepsilon\delta})$ retrieval time.*

Proof. The algorithm BPTree constructs a hash table in the same manner as CountSketch where the items are hashed into $b = O(1/\varepsilon^2)$ buckets for $r = O(\log 1/\varepsilon\delta)$ repetitions. On the stream fed into each bucket we run an independent copy of HH2. A standard $r \times b$ CountSketch is also constructed. The constants are chosen so that when an ε-heavy hitter in the stream is hashed into a bucket it becomes a K-heavy hitter with probability at least 0.95. Thus, in any bucket with a the ε-heavy hitter, the heavy hitter is identified with probability at least 0.55 by Lemma 31 and the aforementioned hashing success probability.

At the end of the stream, all of the items returned by instances of HH2 are collected and their frequencies checked using the CountSketch. Any items that cannot be ε-heavy hitters are discarded. The correctness of this algorithm, the bound on its success probability, and the $(\varepsilon, 1/\varepsilon^2)$-tail guarantee follow directly from the correctness of CountSketch and the fact that no more than $O(\varepsilon^{-2} \log(1/\delta\varepsilon))$ items are identified as potential heavy hitters. □

We can amplify the success probability of HH2 to any $1 - \delta$ by running $O(\log(1/\delta))$ copies in parallel and taking a majority vote for the heavy hitter. This allows one to track $O(1)$-heavy hitters at all points in the stream with an additional $O(\log \log m)$ factor in space and update time. The reason is because there can be a succession of at most $O(\log m)$ 2-heavy hitters in the stream, since their frequencies must increase geometrically, so setting $\delta = \Theta(1/\log m)$ is sufficient. The same scheme works for BPTree tree, as

well, and if one replaces each of the counters in the attached CountSketch with an F_2-at-all-times estimator of [31] then one can track the frequencies of all ε-heavy hitters at all times as well. The total storage becomes $O(\frac{1}{\varepsilon^2}(\log\log n + \log\frac{1}{\varepsilon}))$ words and the update time is $O(\log\log n + \log\frac{1}{\varepsilon})$.

2.3.3 Experimental Results

We implemented HH2 in C to evaluate its performance and compare it against the CountSketch for finding one frequent item. The source code is available from the authors upon request. In practice, the hashing and repetitions perform predictably, so the most important aspect to understand the performance of BPTree is determine the heaviness constant K where HH2 reliably finds K-heavy hitters. Increasing the number of buckets that the algorithm hashes to effectively decreases n. Therefore, in order to maximize the "effective" n of the tests that we can perform within a reasonable time, we will just compare CountSketch against HH2.

The first two experiments help to determine some parameters for HH2 and the heaviness constant K. Afterwards, we compare the performance of HH2 and CountSketch for finding a single heavy hitter in the regime where the heavy hitter frequency is large enough so that both algorithms work reliably.

Streams. The experiments were performed using four types of streams (synthetic data). In all cases, one heavy hitter is present. For a given n and α there are n items with frequency 1 and one item, call it H, with frequency $\alpha\sqrt{n}$. If α is not specified then it is taken to be 1. The four types of streams are (1) all occurrences of H occur at the start of the stream, (2) all occurrences of H at the end of the stream, (3) occurrences of H placed randomly in the stream, and (4) occurrences of H placed randomly in blocks of $n^{1/4}$.

	avg. maximum F_2 tracking error					worst maximum F_2 tracking error				
b \ r	**1**	**2**	**4**	**8**	**16**	**1**	**2**	**4**	**8**	**16**
1	1.2	0.71	0.82	0.66	0.59	4.3	1.2	2.7	0.85	0.86
10	0.35	0.30	0.33	0.19	0.16	1.1	0.68	0.91	0.28	0.20
100	0.12	0.095	0.080	0.074	0.052	0.24	0.17	0.13	0.13	0.10
1000	0.044	0.030	0.028	0.018	0.017	0.076	0.060	0.045	0.029	0.024

TABLE 2.3: Average and maximum F_2 tracking error over 10 streams for different choices of b and r.

The experiments were run on a server with two 2.66GHz Intel Xenon X5650 processors, each with 12MB cache, and 48GB of memory. The server was running Scientific Linux 7.2.

F_2 tracking experiment. The first experiment tests the accuracy of the F_2 tracking for different parameter settings. We implemented the F_2 tracking in *C* using the speed-up of [140]. The algorithm uses the same $r \times b$ table as a CountSketch. To query F_2 one takes the median of r estimates, each of which is the sum of the squares of the b entries in a row of the table. The same group of ten type (3) streams with $n = 10^8$ and $\alpha = 1$ was used for each of the parameter settings.

The results are presented in Table 2.3. Given the tracker $\hat{F}_2(t)$ and true evolution of the second moment $F_2(t)$, we measure the maximum F_2 tracking error of one instance as $\max_t |\hat{F}_2(t) - F_2(t)|/F_2$, where F_2 is the value of the second moment at the end of the stream. We report the average maximum tracking error and the worst (maximum) maximum tracking over each of the ten streams for every choice of parameters.

The table indicates that, for every choice of parameter settings, the worst maximum tracking error is not much worse than the average maximum tracking error. We observe that the tracking error has relatively low variance, even when $r = 1$. It also shows that the smallest possible tracker, with $r = b = 1$, is highly unreliable.

Implementations of HH2 and CountSketch. HH2 implementation details. We have implemented the algorithm HH2 as described in Algorithm 8. The maximum number of rounds is $R = \min\{\lceil 3\log_2 n\rceil, 64\}$. We implemented the four-wise independent hashing using the "CW" trick using the *C* code from [141] Appendix A.14. We use the code from Appendix A.3 of [141] to generate 2-universal random variables for random relabeling of the item. The F_2 tracker from the previous section was used, we found experimentally that setting the tracker parameters as $r = 1$ and $b = 30$ is accurate enough for HH2. We also tried four-wise hashing with the tabulation-based hashing for 64-bit keys with 8 bit characters and compression as implemented in *C* in Appendix A.11 of [141]. This led to a 48% increase in speed (updates/millisecond), but at the cost of a 55 times increase in space.

CountSketch implementation details. We implemented CountSketch in C as described in the original paper [39] with parameters that lead to the smallest possible space. We use the CountSketch parameters as $b = 2$ (number of buckets/row) and $r = \lceil 3 + \log_2 n\rceil$ (number of rows). The choice of b is the smallest possible value. The choice of r is the minimum needed to guarantee that, with probability $7/8$, there does not exist an item $i \in [n] \setminus \{H\}$ that collides the heavy hitter in every row of the data structure. In particular, if we use only $r' < r$ rows then we expect $2^{\log_2 N - r'}$ collisions with the heavy hitter, which would break the guarantee of the CountSketch. Indeed, suppose there is a collision with the heavy hitter and consider a stream where all occurrences of H appear at the beginning, then CountSketch will not correctly return H as the most frequent item because some item that collides with it and appears after it will replace H as the candidate heavy hitter in the heap. In our experiments, the CountSketch does not reliably find the α-heavy hitter with these parameters when $\alpha < 32$. This gives some speed and storage advantage to the CountSketch in the comparison against HH2, since b and/or r would need to increase to

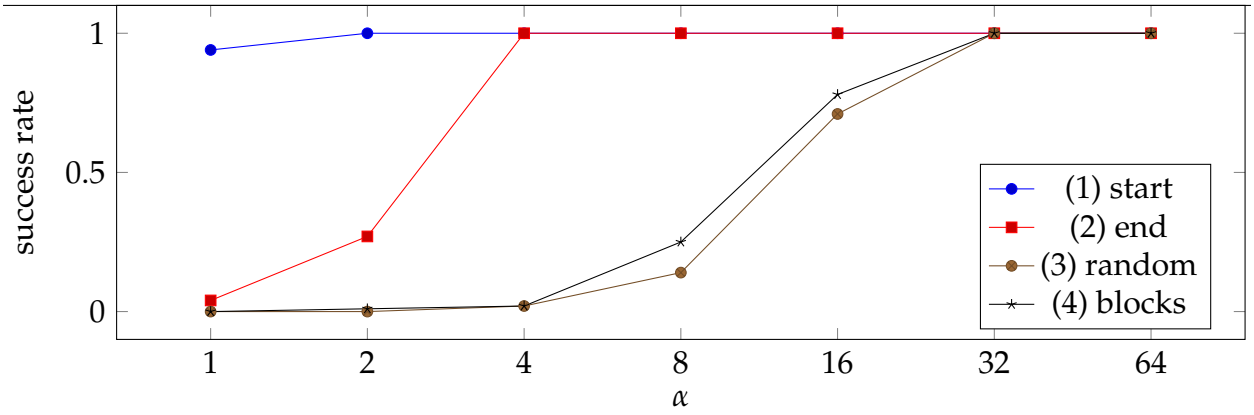


FIGURE 2.2: Success rate for HH2 on four types of streams with $n = 10^8$ and heavy hitter frequency $\alpha\sqrt{n}$.

make CountSketch perform as reliably as HH2 during these tests.

We also tried implementing the four-wise hashing with the Thorup-Zhang tabulation. With the same choices of b and r this led to an 18% speed-up and a 192 times average increase in space. Since the hash functions are such a large part of the space and time needed by the data structure this could likely be improved by taking $b > 2$, e.g. $b = 100$, and $r \approx \lceil \log_b n \rceil$. No matter what parameters are chosen the storage will be larger than using the CW trick because each tabulation-based hash function occupies 38kB, which already ten times larger than the whole CountSketch table.

Heaviness. The goal of this experiment is to approximately determine the minimum value K where if $f_H \geq K\sqrt{n}$ then HH2 correctly identifies H. As shown in Lemma 31, $K \leq 12 \cdot 380,000$ but we hope this is a very pessimistic bound. For this experiment, we take $n = 10^8$ and consider $\alpha \in \{1, 2, 2^2, \ldots, 2^6\}$. For each value of α and all four types of streams we ran HH2 one hundred times independently. Figure 2.2 displays the success rate, which is the fraction of the one hundred trials where HH2 correctly returned the heavy hitter. The figure indicates that HH2 succeeds reliably when $\alpha \geq 32$.

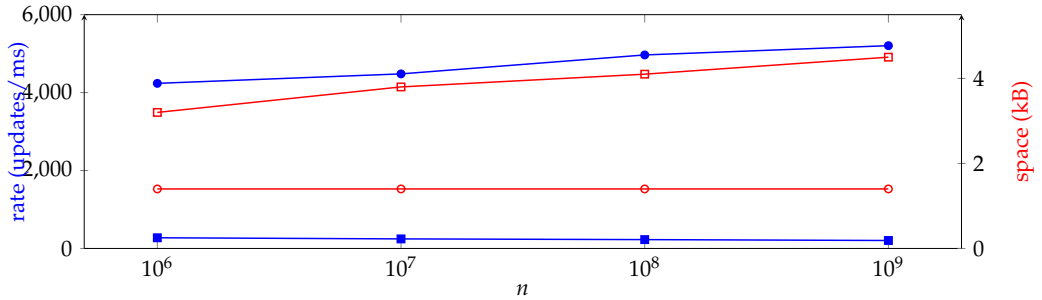


FIGURE 2.3: Update rate in updates/ms (●) and storage in kB (○) for HH2 and CountSketch (■ and □, respectively) with the CW trick hashing.

HH2 versus CountSketch comparison. In the final experiment we compare HH2 against CountSketch. The goal is to understand space and time trade-off in a regime where both algorithms reliably find the heavy hitter.

For each choice of parameters we compute the update rate of the CountSketch and HH2 (in updates/millisecond) and the storage used (in kilobytes) for all of the variables in the associated program. The results are presented in Figure 2.3.

The figure shows that HH2 is much faster and about one third of the space. The dramatic difference in speed is to be expected because two bottlenecks in CountSketch are computing the median of the $\Theta(\log n)$ values and evaluating $\Theta(\log n)$ hash functions. HH2 removes both bottlenecks. Furthermore, as the subroutine HH1 progresses a greater number of items are rejected from the stream, which means the program avoids the associated hash function evaluations in HH2. This phenomena is responsible for the observed *increase* in the update rate of HH2 as n increases. An additional factor that contributes to the speedup is amortization of the start-up time for HH2 and of the restart time for each copy of HH1.

Experiments summary. We found HH2 to be faster and smaller than CountSketch. The number of rows strongly affects the running time of CountSketch because during each update r four-wise independent hash functions must be evaluated and of a median r values is computed. The discussion in Section 2.3.3 explains why the number of rows r cannot be reduced by much without abandoning the CountSketch guarantee or increasing the space. Thus, when there is a K-heavy hitter for sufficiently large K our algorithm significantly outperforms CountSketch. Experimentally we found $K = 32$ was large enough.

The full BPTree data structure is needed to find an item with smaller frequency, but for finding an item of smaller frequency CountSketch could outperform BPTree until n is very large. For example, to identify an α-heavy hitter in the stream our experiments suggest that one can use a BPTree structure with about $\lceil (32/\alpha)^2 \rceil$ buckets per row. In comparison a CountSketch with roughly $\max\{2, 1/\alpha^2\}$ buckets per row should suffice. When α is a small constant, e.g. 0.1, what we find is that one can essentially reduce the number rows of the data structure from $\log(n)$ to just a few, e.g. one or two, at the cost of a factor $32^2 = 1024$ increase in space.[3] This brief calculation suggests that CountSketch will outperform BPTree when the heaviness is $\alpha < 1$ until $n \gtrsim 2^{1024}$—which is to say always in practice. On the other hand, our experiments demonstrate that HH2 clearly outperforms CountSketch with a sufficiently heavy heavy hitter. More experimental work is necessary to determine the heaviness threshold (as a function of n) where BPTree outperforms CountSketch. There are many parameters that affect the trade-offs among space, time, and accuracy, so such an investigation is beyond the scope of the preliminary results reported here.

[3]Recall, $\Omega(\log n / \log(1/\alpha))$ rows are necessary CountSketch whereas BPTree needs only $O(\log 1/\alpha)$ rows.

2.4 Conclusion

In this chapter we studied the heavy hitters problem, which is arguably one of the most important problems for data streams. The problem is heavily inspired from practice and algorithms for it are used in commercial systems. We presented two results leading to the first space and time optimal algorithm for finding ℓ_2-heavy hitters, which is the strongest notion of heavy hitters achievable in polylogarithmic space. By optimal, we mean the time is $O(1)$ to process each stream update, and the space is $O(\log n)$ bits of memory. These bounds match trivial lower bounds (for constant ε). We also provided new techniques which may be of independent interest: (1) a one-pass implementation of a multi-round adaptive compressed-sensing scheme where we use that after filtering a fraction of items, the heavy item is becoming even heavier (2) a derandomization of Bernoulli processes relevant in this setting using limited independence. Both are essential in obtaining an optimal heavy hitters algorithm with $O(1)$ memory. Technique (1) illustrates a new power of insertion-only streams and technique (2) can be stated as a general chaining result with limited independence in terms of the size of the nets used. Given the potential practical value of such an algorithm, we provided preliminary experiments showing our savings over previous algorithms.

Chapter 3

Monitoring the Network with Interval Queries

This chapter is based on the work done in collaboration with Liu Z., Braverman V., Ben-Basat R., Einziger G. and Friedman R.

3.1 Introduction

Network monitoring is at the heart of many networking protocols and network functions, such as traffic engineering [18], load balanced routing [108, 146], attack and anomaly detection [15, 52, 111, 128], and forensic analysis [88, 131]. Over the years, a large number of metrics have been defined, including *per-flow frequency* [51, 145], *heavy hitter detection* [16, 43], *distinct heavy hitters* [56], *cardinality estimation* [58, 71, 73], *change detection* [59], *entropy estimation* [9, 118], and more. With limited memory and computing resources on the network device, it is often infeasible to compute these statistics at line rate. Thus, approximated results are often a reasonable choice [145].

Interestingly, all the above metrics can be efficiently measured by streaming algorithms with a small amount of memory. To that end, [29] has shown that any metric

computable in polylogarithmic time per packet, can be obtained from the per-flow frequency and the L_2-norm of the subsets of the flows in the stream. In other words, instead of maintaining separate data-structures and algorithms for each possible metric, it is enough to estimate the flow frequencies and L_2-norm as promoted by the seminal UnivMon work [101].

Since computer networks usually operate continuously, for many applications, it is particularly important to measure the network statistics that only reflect the recent traffic. This model is known as the *sliding window* model [46], where the metrics are always computed over a fixed-size window of recent data.

However, a measurement on a window of fixed size does not provide visibility into any intervals within the window, e.g., a heavy hitter detection algorithm computed over a 1 min window might not easily detect a 3-second microburst flow. Thus we are interested in a more refined model in which the application provides a time interval of interest at query time. That is, the desired metric is estimated over a specific time interval rather than on a fixed sized sliding window. We refer to this model as the *Interval Queries* (IQ) model. This model is useful when there are multiple interesting intervals, or if the window of interest is not known a-priory. Also, it enables performing drill-down queries of finer and finer intervals and comparing what happens at various intervals for root cause analysis. For example, a security application may use the IQ model to determine exactly when a suspicious pattern has emerged and how it changed over time. The IQ model was previously studied for L_1 heavy hitters in [14]. Yet, that work is limited to flow size estimation and L_1 heavy hitters only. In [115], work done in collaboration of Ivkin, Liu, Ben-Basat, Einziger, Friedman, and Bravermane, we introduce the first set of measurement algorithms for L_2 heavy hitters and L_2 estimation in the IQ model. Further, we extend our techniques to adapt UnivMon [101] the IQ model queries, where a variety of

traffic statistics can be efficiently measured. Therefore, we are the first to facilitate drill-down queries for a large variety of useful metrics using sub-linear space. We evaluate our algorithms using real Internet traces and show that they achieve good accuracy for network measurement tasks within acceptable memory space limitations.

3.2 Preliminaries

As it was introduced in Section 3.2 the streaming model targets the applications where the data items arrive sequentially, and each item is only accessed at the moment of its arrival. One is given a stream of updates $S = \{s_1, \ldots, s_m\}$, where $s_i \in D$ and D is a dictionary of all possible elements, and the goal is to compute a target function $f(S)$ while using the space sublinear in m and $|D|$. Space constraints often render the exact computation of a function infeasible; instead, streaming algorithms usually provide a (ε, δ)-approximation scheme. That is, randomized algorithms that return $\hat{f}(S) \in (1 \pm \varepsilon) f(S)$ with probability at least $1 - \delta$. For more details on the streaming model and its variations refer to [3, 113].

In many applications, the stream of data is considered to be infinite, and a target function should be computed only on the last n updates and "forget" older ones. The Sliding Window model [46] addresses the pool of such problems. Formally, given a stream of updates $S = \{s_1, \ldots, s_t, \ldots\}$ and $s_i \in D$, the goal of a sliding window algorithm is to report $f(t-n, t) = f(S(t-n, t)) = f(\{s_{t-n}, \ldots, s_t\})$ at any given moment t. Similarly, the algorithm should use the space sublinear in n and $|D|$ and follow the approximation scheme $\hat{f}(t-n, t) \in (1 \pm \varepsilon) f(t-n, t)$.

In this work, we address typical measurement tasks in the IQ model. First considered in [14, 100], its goal is estimating a function over the interval (t_1, t_2) (of the stream S) that is specified at query time. Given a stream of updates S, the goal of an algorithm

in the IQ model is to compute $f(t_1, t_2) = f(S(t_1, t_2))$ at any moment t, and any given interval $(t_1, t_2) \subset (t - n, t)$, while using space sublinear in n and $|D|$. In section 3.3, we show that achieving approximation $\hat{f}(t_1, t_2) = (1 \pm \varepsilon) f(t_1, t_2)$ is infeasible as it requires $\Omega(n)$ bits of memory. Thus, we call an IQ algorithm (ε, δ)-approximate if it returns $\hat{f}(t_1, t_2) = f(t_1, t_2) \pm \varepsilon f(t_1, t)$ with probability at least $1 - \delta$. That is, the allowed error is an ε fraction of the value of the function when applied on the suffix (t_1, t) and not only on (t_1, t_2). Specifically, this means that if t_2 is the current time, we get a multiplicative $(1 + \varepsilon)$-approximation for a t_1-sized window whose size is given at query time.

Finding heavy hitters in streaming data is a well-studied problem in analysis of large datasets; optimal or nearly optimal results were achieved in different models [21, 24, 32, 33, 39, 44, 107, 109, 147]. In this section for the sake of completeness we give a brief overview of the heavy hitters problem, including the formal problem statement and major differences between L_1 and L_2 settings. For more details on the problem please refer to [3, 43, 113].

Item i is an (ε, L_p)-heavy hitter in the stream $S = \{s_i\}_{i=1}^{m}, s_i \in \{1, \ldots, N\}$, if $f_i \geq \varepsilon L_p(f)$, where $f_i = \#\{j | s_j = i\}$ is number of occurrences of item i in the stream S, and $L_p(f) = \sqrt[p]{\sum f_i^p}$ is the L_p norm of frequency vector f, and j-th coordinate in the vector equals to f_j. An approximation algorithm for L_p heavy hitters returns all the items that appear at least εL_p times and no item that appears less than $\frac{\varepsilon}{2} L_p$ times and error with probability at most δ. It was shown in [12, 37] that for $p > 2$ any algorithm will require the space at least polynomial in m and N. Therefore, the central interest is around finding L_1 and L_2 heavy hitters. Note that finding L_2 is provably more difficult, compared to L_1 heavy hitters. While to be an (ε, L_1)-heavy hitter an item needs to appear in a constant fraction of the stream updates, in some cases to be an (ε, L_2)-heavy hitter the item can appear just in $O(1/\sqrt{n})$ fraction of updates. Note that to catch such L_1 heavy hitters using uniform

sampling, one will need to sample at most $O(1/\varepsilon^2)$ items, while catching L_2 heavy hitters will require the number of samples to be polynomial in n. Moreover, any L_2 algorithm can find all L_1 heavy hitters while the opposite is not always the case. The L_1 heavy hitters problem has optimal algorithms in both the cash register [107, 109] and turnstile [44] streaming models and was considered in both sliding windows [46] and Interval Query [14] computational models. The L_2 heavy hitters problem has tight results for both cash register [32] and turnstile [39] streaming models. Recent results on L_2 heavy hitters in the sliding window were shown to be space optimal [33]; however, to the best of our knowledge, the problem was not considered in the Interval Query model. In this work, we consider the following approximation L_2 heavy hitters problem in the Interval Query model.

Definition 33 ((ε, L_2)-heavy hitters problem in IQ)**.** *For $0 < \varepsilon < 1$ output set of items $T \subset [N]$, such that T contains all items with $f_i(t_1, t_2) \geq \varepsilon L_2(t_1, t)$ and no items with $f_i(t_1, t_2) \leq \frac{\varepsilon}{2} L_2(t_1, t)$.*

There is a strong connection between the Interval Query and Sliding Window (SW) models: any algorithm that solves the problem in IQ model can answer SW queries as well, one only needs to query the largest permitted interval, i.e., $(t_1, t_2) = (t - n, n)$. Therefore, we expect the current SW approaches to be useful for understanding the IQ model. Further, we introduce some background on SW and (ε, L_2)-heavy hitters algorithms in it.

Currently, two general SW frameworks are known: Exponential Histogram [46] and Smooth Histogram [28]. We now provide a brief overview of the core techniques of those frameworks.

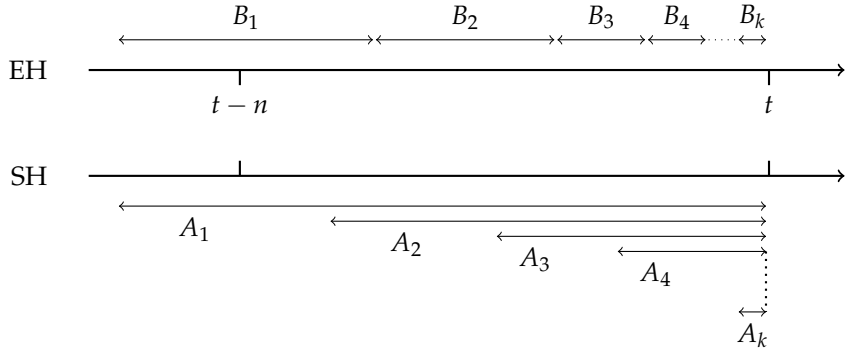


FIGURE 3.1: Interval (bucket) structure for EH and SH.

Exponential Histograms (EH). In [46], the authors suggest to break the sliding window $W = (t-n, n)$ into a sequence of k non-overlapping intervals $B_1, B_2, \ldots, B_k$, as depicted in Fig. 3.1. Window W is covered by $\cup_{i=1}^{k} B_i$, and contains all B_i except the first one. Then, if a target function f admits a composable sketch, maintaining such a sketch on each bucket can provide us with an estimator for f on window $W' = \cup_{i=2}^{k} B_i$. Note, that $f(W)$ is "sandwiched" between $f(W')$ and $f(B_1 \cup W')$. Therefore, a careful choice of each bucket endpoints provides control over the difference between $f(W)$ and $f(W')$, thereby making $f(W')$ a good estimator for $f(W)$. As the window slides, new buckets are introduced, old buckets get expired and deleted, and buckets in between get merged. The EH approach admits non-negative, polynomially bounded functions f which admit composable sketch and are weakly additive, i.e., $\exists C_f \geq 1$, such that $\forall S_1, S_2$:

$$f(S_1) + f(S_2) \leq f(S_1 \cup S_2) \leq C_f(f(S_1) + f(S_2)).$$

For such functions, [46] can ensure that $k = O(C_f^2 \log n)$ by maintaining two invariants: $f(B_j) \leq \frac{C_f}{k} \sum_{i=j}^{k} f(B_j)$ and $f(B_{j-1}) + f(B_j) \leq \frac{1}{k} \sum_{i=j}^{k} f(B_j)$.

Smooth Histograms (SH). In [28], the authors relax the weak additivity to more general property of smoothness — $\exists 0 < \beta \leq \alpha \leq \varepsilon \ \ \forall S_1, S_2, S_3$:

$$(1-\beta)f(S_1 \cup S_2) \leq f(S_2) \Rightarrow (1-\alpha)f(S_1 \cup S_2 \cup S_3) \leq f(S_2 \cup S_3).$$

Additionally, in SH buckets $A_1, \ldots, A_k$ overlap; therefore, [28] extends the class of admitted target functions even further by relaxing the composability requirement. Similarly to [46], $f(W)$ is "sandwiched" between $f(A_1)$ and $f(A_2)$, see Figure 3.1. The memory overhead is $O(\frac{1}{\beta}\log n)$ and the maintained invariants are $(1-\alpha)f(A_i) \leq f(A_{i+1})$ and $f(A_{i+2}) < (1-\beta)f(A_i)$.

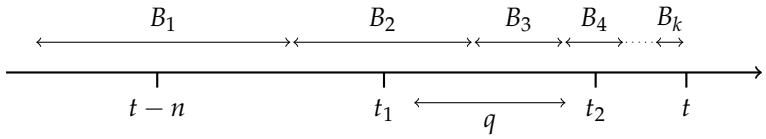


FIGURE 3.2: Interval query in the prism of EH.

Interval queries on EH and SH. The IQ model is more general than the SW model, however we use similar building blocks. Fig. 3.2 depicts query interval $q = (t_1, t_2)$ and buckets of Exponential Histogram with window of size n. Note that q is "sandwiched" between $B_2 \cup B_3 \cup B_4$ and B_3, while $f(\cup_{j=2}^{k} B_j) = (1 \pm \varepsilon)f(t_1, t)$ and $f(\cup_{j=4}^{k} B_j) = (1 \pm \varepsilon)f(t_2, t)$. Intuitively, $f(t_1, t_2)$ can be approximated by $f(B_2 \cup B_3)$ with an additive error of $\pm\varepsilon f(t_1, t)$. Similar approach can be applied to Smooth Histograms, if the sketches preserve approximation upon subtraction.

3.3 Interval Algorithms

Lower bound.

Theorem 34. *Any algorithm which maintains a sketch over the stream and at any moment t: $\forall t_1, t_2 \in (t-n, t)$ outputs $\hat{L}_2(t_1, t_2) = (1 \pm \alpha) L_2(t_1, t_2)$ requires $\Omega(n)$ bits of memory.*

Proof. The proof uses a reduction from the INDEX problem in communication complexity. Alice is given a string $x = \{0,1\}^n$ and Bob gets an index $k \in \{1, \ldots, n\}$. Alice sends a message to Bob and he should report the value of x_k. Bob can err with probability at most δ. A known lower bound on the size of the message, even for randomized algorithms, is $\Omega(n)$ [94].

Suppose there exists an algorithm which maintains a sketch over the stream and at any moment t, for any interval $(t_1, t_2) \subset (t-n, t)$, outputs $\hat{L}_2(t_1, t_2) = (1 \pm \alpha) L_2(t_1, t_2)$, while using only $o(n)$ bits of space. Such an algorithm can distinguish between L_2 and $L_2(1+3\alpha)$; denote $p = (1+3\alpha)^2$.

Alice encodes the string x as a stream of updates $f(x_1), f(x_2), \ldots, f(x_n)$, where $f(1) = 1, \ldots, 1$ and $f(0) = 1, 2, \ldots, p$ with p updates in both cases. Therefore, the entire stream has length pn, $L_2(f(0)) = \sqrt{p}$, and $L_2(f(1)) = p$. Hence, $\frac{L_2(f(1))}{L_2(f(0))} = \sqrt{p}$ and the algorithm can distinguish them. After running the stream through the algorithm, Alice sends the content of the data structure to Bob. Bob queries interval $(t - pn + p(k-1), t - pn + pk)$ and due to approximation guarantees of the algorithm can infer whether $x_k = 0$ or 1. Therefore the INDEX problem was resolved with a message size of $o(n)$ bits, which contradicts the $\Omega(n)$ lower bound. □

Algorithm 1 Braverman, Gelles, and Ostrovsky, in [25], presented the first L_2 heavy hitter algorithm in the SW model. They apply the SH framework described earlier to

get a $(1+\varepsilon)$-approximation of L_2 norm. In addition, each bucket of the SH maintains an instance of the Count Sketch algorithm [39].

Our algorithm, whose pseudo-code is given in Algorithm 9, utilizes a similar technique: when querying with an interval (t_1, t_2), it finds the largest SH suffix (a^1, t) contained in (t_1, t) and the largest suffix (a^2, t) contained in (t_2, t). It then calculates Count Sketch of interval (a^1, a^2) as the difference of two Count Sketches: (a^1, t) and (a^2, t). The frequencies on the interval (t_1, t_2) can be approximated by the Count Sketch of interval (a^1, a^2), as we show later. Next, we prove that the described technique provides a good estimation for $L_2(t_1, t_2)$.

Lemma 35. *Let A be an SH construction for (α, β)-smooth L_2 and $A_i = (a_i, t)$ are the suffixes of A (see Fig. 3.1). Then for any interval (t_1, t_2):*

$$L_2(a^1, a^2) = L_2(t_1, t_2) \pm \sqrt{2\alpha} L_2(t_1, t),$$

where $a^1 = \min a_i \geq t_1$ and $a^2 = \min a_i \geq t_2$.

Proof. From SH invariant [28]: $L_2(a^1, t) > (1-\alpha) L_2(t_1, t)$. then since $L_2^2(t_1, t) \geq L_2^2(a^1, t) + L_2^2(t_1, a^1)$

we have: $L_2(t_1, a^1) \leq \sqrt{2\alpha} L_2(t_1, t)$,

and similarly: $L_2(t_2, a^2) \leq \sqrt{2\alpha} L_2(t_2, t) \leq \sqrt{2\alpha} L_2(t_1, t)$.

Due to triangle inequality and monotonicity of L_2:

$L_2(t_1, t_2) \leq L_2(t_1, a^1) + L_2(a^1, t_2) \leq L_2(t_1, a^1) + L_2(a^1, a^2)$, thus $L_2(a^1, a^2) \geq L_2(t_1, t_2) - \sqrt{2\alpha} L_2(t_1, t)$.

Similarly: $L_2(t_1, t_2) + L_2(t_2, a^2) \geq L_2(t_1, a^2)$

$L_2(t_1, t_2) \geq L_2(t_1, a^2) - L_2(t_2, a^2) \geq L_2(a^1, a^2) - L_2(t_2, a^2)$

$L_2(a^1, a^2) \leq L_2(t_1, t_2) + \sqrt{2\alpha} L_2(t_1, t)$. □

Algorithm 9 L_2 heavy hitter algorithm based on [25]

function UPDATE(item)
    maintain SH for L_2 with $(\alpha, \beta) = (\frac{\varepsilon^2}{2^7}, \frac{\varepsilon^4}{2^{15}})$
    on each bucket A_i maintain Count Sketch CS_i
end function
function QUERY(t_1, t_2)
    find $a^1 = \min a_i \geq t_1$ and $a^2 = \min a_i \geq t_2$
    subtract sketches $CS = CS_{a^1} - CS_{a^2}$
    query L_2 of suffix: $\hat{L}_2(a^1, t) = SH$.query(a^1, t)
    for each item i in CS_{a^1}.heap **do**
        $\hat{f}_i(t_1, t_2) = CS$.estimateFreq(i)
        if $\hat{f}_i(t_1, t_2) > \frac{3\varepsilon}{4}\hat{L}_2(a^1, t)$ **then** report $(i, \hat{f}_i)$
    end for
end function

Theorem 36. *Algorithm 9 solves (ε, L_2)-heavy hitters problem in the IQ model (Definition 33) using $O(\frac{1}{\varepsilon^6}\log^3 n \log\frac{1}{\delta\varepsilon})$ bits of space.*

Proof.

$$\forall i : f_i(a^1, a^2) = f_i(t_1, t_2) - f_i(t_1, a^1) + f_i(t_2, a^2).$$

Note that $f_i(t_1, a^1) \leq L_2(t_1, a^1)$ and $f_i(t_2, a^2) \leq L_2(t_2, a^2)$. From Lemma 35 and due to the choice of SH parameters:

$$f_i(t_1, a^1) \leq \frac{\varepsilon}{8}L_2(t_1, t), \quad f_i(t_2, a^2) \leq \frac{\varepsilon}{8}L_2(t_1, t)$$

For a given stream S Count Sketch approximates the counts as follows:

$$\hat{f}_i(S) = f_i(S) \pm \frac{\varepsilon}{8}L_2(S)$$

and its heap contains all $(\frac{\varepsilon}{8}, L_2)$-heavy hitters. Thus sketch $CS = CS_{a^1} - CS_{a^2}$ will estimate the count of any queried item i as

$$\hat{f}_i(a^1, a^2) = f_i(a^1, a^2) \pm \frac{\varepsilon}{8} L_2(t_1, t) = f_i(t_1, t_2) \pm \frac{\varepsilon}{4} L_2(t_1, t).$$

where the last equality follows from the derivations above.

To prove the correctness of the algorithm, we need to show that:

1. every (ε, L_2)-heavy on (t_1, t_2) item will appear in the heap of CS_{a^1} (line 8) and survive pruning (line 10).

2. no items $f_j(t_1, t_2) \leq \frac{\varepsilon}{2} L_2(t_1, t)$ will survive pruning.

Note that if item i is (ε, L_2)-heavy on (t_1, t_2), then $f_i(t_1, t_2) > \varepsilon L_2(t_1, t)$ which implies $f_i(a^1, a^2) > \frac{7\varepsilon}{8} L_2(t_1, t)$ and $f_i(a^1, t) > \frac{7\varepsilon}{8} L_2(t_1, t)$. The latter one implies that item i will be in the heap of CS_{a^1}. The former one given that

$$\hat{f}_i(a^1, a^2) > f_i(a^1, a^2) - \frac{\varepsilon}{8} L_2(t_1, t) > \frac{3}{4} L_2(t_1, t)$$

and $\hat{L}_2(a^1, t) = (1 \pm \frac{\varepsilon^2}{2^7}) L_2(t_1, t)$ guarantees that heavy item i will survive pruning procedure. Similarly,

$$\hat{f}_i(a^1, a^2) < f_i(a^1, a^2) + \frac{\varepsilon}{8} L_2(t_1, t) < \frac{3}{4} L_2(t_1, t),$$

therefore light items will be filtered out.

According to Theorem 3 from [28], SH approach requires $O(\frac{1}{\beta} g(\varepsilon, \frac{\delta\beta}{n}) \log n)$ bits of memory. Here, $g(\varepsilon, \delta)$ is the amount of memory needed for sketch to get ε approximation of target function with failure probability at most δ. Note that to avoid mistaken deletions of suffixes in the SH construction, every target function sketch should succeed. Therefore

L_2 sketch should fail with probability at most $O(\frac{\delta}{n})$. At the same time, Count Sketch is needed only at the moment of a query, therefore it should fail with probability at most $O(\frac{\delta}{\log n})$. Each amplified L_2 sketch, according to [6], requires $O(\frac{1}{\varepsilon^2}\log^2 n)$ bits of space – same as the amount of memory that is needed for each Count Sketch. Thus, in total, the algorithm requires $O(\frac{1}{\varepsilon^6}\log^3 n)$ bits of memory. □

Algorithm 2 A natural question to ask is whether the Exponential Histogram framework can be used instead of Smooth Histograms. Note that the target function $L_2^2(\cdot)$ admits a composable sketch [6] and is weakly additive with $C_f = 2$. According to Theorem 7 of [46], an admittable target function f can be estimated with the relative error

$$E_r \leq (1+\varepsilon)\frac{C_f^2}{k} + C_f - 1 + \varepsilon,$$

where ε is relative error of the L_2^2 sketch and k is a parameter of EH framework. Note that for $C_f = 2$, no k can get an error better than $E_r = 1 + o(1)$, which only implies a 2-approximation of the L_2-norm on the sliding window. The IQ model is more general than SW, therefore, the same idea would not work out of the box. Instead, we suggest the following decoupling tweak to the weak additivity requirement:

$$\exists C_f, C'_f \; \forall S_1, S_2 : f(S_1 \cup S_2) \leq C_f f(S_1) + C'_f f(S_2).$$

Keeping the rest of the framework the same and repeating the argument as in Theorem 7 of [46], the relative error becomes:

$$E_r \leq (1+\varepsilon)\frac{C_f^2}{k} + C'_f - 1 + \varepsilon.$$

To find appropriate constants C_f and C'_f for L_2^2, note that for any positive integers a and b and any $\varepsilon \in (0,1)$:

$$(a+b)^2 \leq \frac{2}{\varepsilon}a^2 + (1+\varepsilon)b^2 - (\frac{1}{\sqrt{\varepsilon}}a + \sqrt{\varepsilon}b)^2 \leq \frac{2}{\varepsilon}a^2 + (1+\varepsilon)b^2.$$

Therefore, $\forall S_1, S_2 : L_2^2(S_1 \cup S_2) \leq \frac{1}{2\varepsilon}L_2^2(S_1) + (1+\varepsilon)L_2^2(S_2)$, i.e., $C_f = \frac{2}{\varepsilon}$ and $C'_f = 1+\varepsilon$. Setting $k = O(\frac{1}{\varepsilon^3})$ gives a $(1+\varepsilon)$-approximation of L_2^2 on sliding windows using EH framework. Note that the described tweak will work for any admittable function f for which $f(S_1 \cup S_2) \leq C_f f(S_1) + C'_f f(S_2)$, as no other properties of L_2 were used in the proof. Further, we argue that same approximation can be achieved with smaller k. We maintain EH histogram framework for $f = L_2^2$ as proposed in [46] with $C_f = 2$. However, when queried, we output $\sqrt{f} = L_2$ rather than $f = L_2^2$. Note that, due to the triangle inequality, $\sqrt{f(S_1+S_2)} \leq \sqrt{f(S_1)} + \sqrt{f(S_2)}$. Denote $t_0 = t - n$; then the core derivation for the relative error E_r from Theorem 7 of [46] can be rewritten as follows:

$$E_r = \frac{\sqrt{f(t_0,t)} - \sqrt{f(b_1,t)}}{\sqrt{f(t_0,t)}} \leq \sqrt{\frac{f(t_0,b_1)}{f(t_0,t)}} \leq \sqrt{\frac{f(b_0,b_1)}{f(b_1,t)}}$$

$$\leq \sqrt{\frac{C_f \sum_{i>0} f(B_i)}{k f(b_1,t)}} \leq \sqrt{\frac{C_f f(b_1,t)}{k f(b_1,t)}} \leq \sqrt{\frac{C_f}{k}}.$$

Setting $k = O(\frac{1}{\varepsilon^2})$ provide necessary ε-approximation for L_2 on the sliding window.

Lemma 37. *Let B be an EH construction for L_2 with parameters C_f and k, and let $B_i = (b_i, b_{i+1})$ denote the buckets of B (see Fig. 3.1). Then for any interval (t_1, t_2):*

$$L_2(b^1, b^3) = L_2(t_1, t_2) \pm \sqrt{\frac{C_f}{k}} L_2(t_1, t),$$

where (b^0, b^1) is the bucket containing t_1 and (b^2, b^3) is the bucket containing t_2.

Proof. By monotonicity of L_2^2 and EH invariant:

$$L_2^2(t_1, b^1) \leq L_2^2(b^0, b^1) \leq \frac{C_f}{k} \sum_{b_i > b^0} L_2^2(B_i) \leq \frac{C_f}{k} L_2^2(b^1, t).$$

Repeating the argument for (t_2, b^3) and taking the square root:

$$L_2(t_1, b^1) \leq \sqrt{\frac{C_f}{k}} L_2(t_1, t) \text{ and } L_2(t_2, b^3) \leq \sqrt{\frac{C_f}{k}} L_2(t_1, t).$$

Applying triangle inequality the same way as in the proof of Lemma 35 leads to the statement of the current lemma. □

Algorithm 10 L_2 heavy hitter algorithm based on EH

function UPDATE(item)
    maintain EH for L_2^2 with $k = O(\frac{1}{\varepsilon^2})$, $C_f = 2$.
    on each bucket B_i maintain Count Sketch CS_i
end function
function QUERY(t_1, t_2)
    find $b^{0,\cdots,3} : t_1 \in (b^0, b^1) = B_i$ and $t_2 \in (b^2, b^3) = B_j$
    compute union sketch $CS = \cup_{b^1 \leq l \leq b^3} CS_l$
    query L_2 of suffix: $\hat{L}_2(t_1, t) = EH$.query(b^1, t)
    for each heavy hitter (i, f_i) in CS **do**
        if $\hat{f}_i > \frac{3\varepsilon}{4} \hat{L}_2(t_1, t)$ **then** report $(i, \hat{f}_i)$
    end for
end function

Theorem 38. *Algorithm 10 solves (ε, L_2)-heavy hitters problem in IQ model (Definition 33) using $O(\frac{1}{\varepsilon^4} \log^3 n \log \frac{1}{\delta\varepsilon})$ bits of space.*

Proof. Due to Lemma 37 for given parameters C_f and k:

$$\forall i : f_i(b^1, b^3) = f_i(t_1, t_2) \pm \frac{\varepsilon}{16} L_2(t_1, t).$$

Approximation guarantees of Count Sketch CS can be rewritten as:

$$\forall i : \hat{f}_i(b^1, b^3) = f_i(b^1, b^3) \pm \frac{\varepsilon}{16} L_2(b^1, t).$$

Therefore, $\forall i : \hat{f}_i(b^1, b^3) = f_i(t_1, t_2) \pm \frac{\varepsilon}{8} L_2(t^1, t)$.

The same lemma applied to interval (t_1, t), given $\frac{\varepsilon}{16}$-approximation of L_2 on each bucket B_i, leads to $\hat{L}_2(t_1, t) = (1 \pm \frac{\varepsilon}{8}) L_2(t_1, t)$. Thus, any heavy item with $f_i(t_1, t_2) \geq \varepsilon L_2(t_1, t)$ will be reported:

$$\hat{f}_i(b^1, b^3) \geq f_i(t_1, t_2) - \frac{\varepsilon}{8} L_2(t_1, t) \geq \frac{7\varepsilon}{8} L_2(t_1, t) \geq \frac{3\varepsilon}{4} \hat{L}_2(t_1, t).$$

Similarly, any item with $f_i(t_1, t_2) \leq \frac{\varepsilon}{2} L_2(t_1, t)$ will be pruned away:

$$\hat{f}_i(b^1, b^3) \leq \frac{5\varepsilon}{8} L_2(t_1, t) \leq \frac{3\varepsilon}{4} \hat{L}_2(t_1, t).$$

According to Theorem 7 from [46], the EH approach requires $O(kg(\varepsilon, \frac{\delta\beta}{n}) \log n)$ bits of memory, where $g(\varepsilon, \delta)$ is amount of memory needed for sketch to get a $(1+\varepsilon)$-approximation of the target function with a failure probability of at most δ. Similarly to the SH case, L_2 sketch should succeed on $O(n)$ instances, while Count Sketch only on $O(\log n)$. Thus, in total, the algorithm requires $O(\frac{1}{\varepsilon^4} \log^3 n)$ bits of memory. □

Algorithm 3 Recently, a new algorithm for finding L_2-heavy hitters in the SW model was introduced in [33]. The Authors show a significant improvement in space complexity and provide a matching lower bound. Memory footprint of the solution proposed in [33] is $O(\frac{1}{\varepsilon^2} \log^2 n \log \frac{1}{\varepsilon\delta})$ bits, while previous result [25] needed at least $O(\frac{1}{\varepsilon^4} \log^3 n \log \frac{1}{\varepsilon\delta})$ bits. Although the new approach uses the SH framework, it differs conceptually in the way of

catching the heavy hitters. Recall that [25] requires a $(1+\varepsilon)$-approximation of the L_2 to make sure that no heavy hitters are lost between neighboring buckets.

Streaming L_2-heavy hitter algorithm can report a heavy hitter after seeing it $\frac{\varepsilon}{16}L_2$ times. [33] uses that property and approximates the counter for each reported item using a separate SH data structure. To report every (ε, L_2)-heavy and no $(\frac{\varepsilon}{2}, L_2)$-heavy items, one only needs a constant factor approximation for both L_2 and frequency of the potentially-heavy items. [33] exploits this and runs an SH with *constant* α and β for both L_2-norm approximation and for independently tracking the frequency of each potentially-heavy item reported by the Count Sketch.

Additionally, [33] suggests to replace the Count Sketch algorithm with BPTree [32] and use shared randomness and use the strong tracking argument from [31] to avoid union bound for all bucket sketches to succeed.

We reuse this approach for the IQ model. First, we show a SW solution for ε-approximation of f_i, which can be also considered as sum problem in a binary zero-one stream, proved the following guarantees in IQ model:

$$\hat{f}_i(t_1, t_2) = f_i(t_1, t_2) \pm \varepsilon f_i(t_1, t)$$

.

Lemma 39. *Let A be an SH construction for an (α, β)-smooth sum function S and consider a zero-one stream. Denote by $A_i = (a_i, t)$ the suffixes of A (see Fig. 3.1). Then for any interval (t_1, t_2):*

$$S(a^1, a^2) = S(t_1, t_2) \pm \alpha S(t_1, t),$$

where $a^1 = \min a_i \geq t_1$ and $a^2 = \min a_i \geq t_2$.

Proof. From the SH invariant [28]: $S(a^1, t) > (1 - \alpha)S(t_1, t)$.

Since $S(t_1, t) = S(a^1, t) + S(t_1, a^1)$, we have: $S(t_1, a^1) \leq \alpha S(t_1, t)$ and similarly: $S(t_2, a^2) \leq \alpha S(t_2, t)$. Therefore, the statement of the lemma follows from

$S(t_1, t_2) = S(a^1, a^2) + S(t_1, a^1) - S(t_2, a^2)$. □

Algorithm 11 L_2 heavy hitter algorithm based on [33]

function INIT
  init SH_{L_2} with $(\alpha, \beta) = (\frac{1}{10}, \frac{1}{200})$
  init CS_i Count Sketch on each SH_{L_2} bucket A_i
  init HH_p an array for potential heavy items
  − if $i \in HH_p$ then $HH_p[i].SH$ tracks f_i in SW
end function
function UPDATE(item)
  update SH_{L_2} and all CS_i
  if item $\in HH_p$ **then** update HH_p[item].SH
  for all i **for each** item $(j, \hat{f}_j)$ in CS_i.heap
    if $\hat{f}_j > \frac{3\varepsilon}{4} SH_{L_2}$.query$(A_i)$ **and** $j \notin HH_p$:
      HH_p.add(j)
      init $HH_p[j].SH$ with $(\alpha, \beta) = (\frac{\varepsilon}{16}, \frac{\varepsilon}{16})$
end function
function QUERY(t_1, t_2)
  $a^1 = \min a_i \geq t_1$ and $\hat{L}_2(a^1, t) = SH$.query(a^1, t)
  for each item $i \in HH_p$ **do**
    $a^1 = \min a_i \geq t_1$ and $a^2 = \min a_i \geq t_2$
    $\hat{f}_i(t_1, t) = HH_p[i].SH$.query$(a_1, t)$
    $\hat{f}_i(t_1, t_2) = \hat{f}_i(t_1, t) - HH_p[i].SH$.query$(a_2, t)$
    if $\hat{f}_i(t_1, t_2) > \frac{3\varepsilon}{4} \hat{L}_2(a^1, t)$ **then** report $(i, \hat{f}_i)$
  end for
end function

Theorem 40. *Algorithm 11 solves (ε, L_2)-heavy hitters problem in IQ model (Definition 33) using $O(\frac{1}{\varepsilon^3} \log^3 n \log \frac{1}{\delta\varepsilon})$ bits of space.*

Proof. First, let's show that all items with $f_i(t_1, t_2) \geq \varepsilon L_2(t_1, t)$ will appear in HH_p. Denote $a_0 = \max a_i \leq t_1$ then

$$f_i(a^0, t_2) \geq f_i(t_1, t_2) \geq \varepsilon L_2(t_1, t) \geq \varepsilon L_2(t_1, t_2).$$

Therefore, by moment t_2, Count sketch of the bucket (a^0, t_2) should have reported it in line 10 of Algorithm 11.

Now we argue, that all heavy items will survive pruning in line 19. Let (a_0, t) be the first bucket of $HH_p[i].SH$ then we should consider two cases: $t_1 \geq a_0$ and $t_1 \leq a_0$.

If $t_1 \geq a_0$ then it follows from Lemma 39, that $\hat{f}_i(t_1, t_2) \geq f(t_1, t_2) - \frac{\varepsilon}{16} f(t_1, t)$. At the same, time SH framework guarantees $\hat{L}_2(t_1, t) \leq 1.1 L_2(t_1, t)$. Therefore, if $f_i(t_1, t_2) \geq \varepsilon L_2(t_1, t)$, then:

$$\hat{f}_i(t_1, t_2) \geq \frac{15}{16} \varepsilon L_2(t_1, t) \geq \frac{3}{4} \varepsilon \hat{L}_2(t_1, t)$$

Recall, that Count Sketch reports an item after seeing $\frac{\varepsilon}{16} L_2$ its instances, therefore, for $t_1 \leq a_0$ using the same lemma we can conclude that $\hat{f}_i(t_1, t_2) \geq f(t_1, t_2) - \frac{\varepsilon}{8} f(t_1, t)$, while the rest of computation is the same. Therefore, all items with $f_i(t_1, t_2) \geq \varepsilon L_2(t_1, t)$ will be reported by the Algorithm 11.

For every non-heavy item with $f_i(t_1, t_2) \leq \frac{\varepsilon}{2} L_2(t_1, t)$, if $i \in HH_p$, then from Lemma 39 $\hat{f}_i(t_1, t_2) \leq f_i(t_1, t_2) + \frac{\varepsilon}{16} f_i(t_1, t) \leq \frac{\varepsilon}{2} L_2(t_1, t) + \frac{\varepsilon}{16} L_2(t_1, t) \leq \frac{3\varepsilon}{4} \hat{L}_2(t_1, t)$, and item i will be pruned out in line 19.

According to Theorem 3 from [33], modified SH approach requires $O(\frac{1}{\beta} g(\varepsilon, \delta) \log n)$ bits of memory. Algorithm 11 uses $\beta = O(1)$ for SH_{L_2}, with $g(\varepsilon, \delta) = O(\frac{1}{\varepsilon^2} \log^2 n)$ due to L_2 sketch and Count Sketch instances. Thus, SH_{L_2} requires $O(\frac{1}{\varepsilon^2} \log^2 n)$ bits of space, and have $O(\log n)$ buckets and each can potentially generate up to $O(\frac{1}{\varepsilon^2})$ items in HH_p. Therefore, in total, to track ε-approximation to frequencies of all potential

heavy hitters, data structure spend $O(\frac{1}{\varepsilon^3}\log^3 n)$ bits of space. Summing the two derived quantities, the space complexity of Algorithm 11 is $O(\frac{1}{\varepsilon^3}\log^3 n \log\frac{1}{\delta\varepsilon})$ bits. □

Note that replacing Count Sketch in Algorithm 11 with BPTree [32] improves the space complexity by another $\log n$ factor.

Corollary 40.1. *There exists an algorithm that solves (ε, L_2)-heavy hitters problem in IQ model using space $O(\frac{1}{\varepsilon^3}\log^2 n \log\frac{1}{\delta\varepsilon})$ bits of space.*

In this work, we mainly focus on the trade-off between memory and precision. In Table 3.1 we compare space complexity, update and query time for all proposed algorithms. Optimizing algorithms towards improving the query time is the subject of future research.

Alg.	Space complexity	Update time	Query time
1	$O(\varepsilon^{-6}\log^3 n \log\delta^{-1})$	$O(\varepsilon^{-4}\log n)$	$O(\varepsilon^{-2}\log n)$
2	$O(\varepsilon^{-4}\log^3 n \log\delta^{-1})$	$O(\varepsilon^{-2}\log n)$	$O(\varepsilon^{-3}\log n)$
3	$O(\varepsilon^{-3}\log^2 n \log\log\delta^{-1})$	$O(\log n)$	$O(\varepsilon^{-2}\log n)$

TABLE 3.1: Space complexity, update and query time for all proposed algorithms.

Extending to a wider class of functions Many streaming algorithms and frameworks use L_2-heavy hitter algorithm as a subroutine. One of them is UnivMon [101], the framework which promotes recent results on universal sketching [34] in the field of network traffic analysis. The main power of the framework is its ability to maintain only one sketch for many target flow functions, rather than an ad-hoc sketch per each function. The class of functions that can be queried is wide and covers the majority of those used

in practice, among examples are the L_0, L_2 norms and entropy. Therefore, L_2-heavy hitters is an important step towards UnivMon in IQ model. For more details on universal sketching refer to [34]; here we will cover the necessary basics.

Given a function $g : \mathbb{N} \to \mathbb{N}$, the goal of universal sketching is to approximate $G = \sum_{i=1}^{n} g(f_i)$ by making one or several passes over the stream. Theorem 2 in [34] states: if $g(x)$ grows slower then x^2, drops no faster than sub-polynomially, and has predictable local variability, then there is an algorithm that outputs an ε-approximation to G, using sub-polynomial space and only one pass over the data. The algorithm consists of two major subroutines: (g, ε)-heavy hitters and Recursive Sketch. The first one finds all items i such that $g(f_i) \geq \varepsilon G$ together with an ε-approximation to $g(f_i)$. In [34], the authors show that if an item is (g, ε)-heavy then it is also $(L_2, \frac{\varepsilon}{h})$-heavy for sub-polynomial h; therefore, Count Sketch can be used to find (g, ε)-heavy items. Recursive Sketch was initially introduced in [80] and further generalized in [27]. It finds ε-approximation of G using a (g, ε)-heavy hitters algorithm as the black box, by recursively subsampling the universe, and by estimating the sum G of the subsampled stream.

In [34] (g, ε)-heavy hitter algorithm requires subroutine which find all items i such that $f_i(t_1, t_2) \geq \varepsilon L_2(t_1, t_2)$. However, due to the limitations of the IQ model all three proposed algorithms only find all items i such that $f_i(t_1, t_2) \geq \varepsilon L_2(t_1, t)$. Further, we adjust the argument from [34] to argue that one can use algorithms from previous sections to find (g, ε)-heavy hitter with guarantee defined as follows:

Definition 41 ((ε, g)-heavy hitters problem in IQ)**.** *For $0 < \varepsilon < 1$ output set of items $T \subset [N]$, such that T contains all items with $g(f_i(t_1, t_2)) \geq \varepsilon G(t_1, t) = \varepsilon \sum_j g(f_j(t_1, t))$ and no items with $g(f_i(t_1, t_2)) \leq \frac{\varepsilon}{2} G(t_1, t)$.*

Propositions 15 and 16 in [34] show that if function g is slow-jumping and slow-dropping, then there exist a sub-polynomial function H:

$$\forall x \leq y : g(y) \geq \frac{g(x)}{H(y)},\ g(y) \leq \left(\frac{y}{x}\right)^2 y^{\alpha} H(y) g(x) \tag{3.1}$$

We can adjust the argument of Lemmas 17 and 18 in [34] to handle the heavy hitters with additive error.

Lemma 42. *Let $HH(\varepsilon, \delta)$ be an algorithm that solves (ε, L_2)-heavy hitters problem in IQ model (Definition 33), and g is a slow-jumping and slow-dropping function. Then $HH(\frac{\varepsilon}{2H(n)}, \delta)$ solves (ε, g)-heavy hitters problem in IQ model (Definition 41).*

Proof. Note that for any (g, ε)-heavy i:

$$g(f_i(t_1, t_2)) \geq \varepsilon \textstyle\sum_j g(f_j(t_1, t))$$

Therefore, applying the second statement of Equation 3.1:

$$g(f_i(t_1, t_2)) \geq \sum_{f_j(t_1,t) \leq f_i(t_1,t_2)} \frac{\varepsilon g(f_i(t_1, t_2)) f_j^2(t_1, t)}{H(n) f_i^2(t_1, t_2)}$$

$$f_i^2(t_1, t_2) \geq \frac{\varepsilon}{H(n)} \sum_{f_j(t_1,t) \leq f_i(t_1,t_2)} f_j^2(t_1, t)$$

Similarly, applying the first statement of Equation 3.1:

$$g(f_i(t_1, t_2)) \geq \sum_{f_j(t_1,t) \geq f_i(t_1,t_2)} \frac{\varepsilon g(f_i(t_1, t_2))}{H(n)}$$

Hence, there are at most $\frac{H(n)}{\varepsilon}$ items with $f_j(t_1, t) \geq f_i(t_1, t_2)$, and $HH(\frac{\varepsilon}{2H(n)}, \delta)$ will detect it. □

Further we argue that Recursive Sketch with (g, ε)-heavy hitter algorithm which finds all i such that $g(f_i(t_1, t_2)) \geq \varepsilon G(t_1, t)$ will return $\hat{G}(t_1, t_2) = G(t_1, t_2) \pm \varepsilon G(t_1, t)$.

Algorithm 12 Recursive GSum(S_0, ε) [34]

1: $H_1, \ldots, H_{\phi=O(\log n)}$ are pairwise independent 0/1 vectors
2: S_j is a subsampled stream $\{s \in S_{j-1} : H_j(s) = 1\}$
3: Compute, in parallel, $HH_j = HH(S_j)$
4: Compute $Y_\phi = G_\phi(t_1, t_2) \pm \varepsilon G_\phi(t_1, t)$
5: **for** $j = \phi - 1, \ldots, 0$ **do**
6: compute $Y_j = 2Y_{j+1} + \sum_{i \in HH_j} (1 - 2H_j(i))\hat{g}(f_i)$
7: **end for**
8: **return** Y_0

Theorem 43. *Let HH be an algorithm that finds all i such that $g(f_i(t_1, t_2)) \geq \varepsilon G(t_1, t)$ together with ε-approximation of $g(f_i(t_1, t_2))$. Then Algorithm 12 computes $\hat{G}(t_1, t_2) = G(t_1, t_2) \pm \varepsilon G(t_1, t)$ and errs with probability at most 0.3. Its space overhead is $O(\log n)$.*

Proof. Note that G_j is a G-sum computed on the subsampled stream S_j and according to the line 4 of Algorithm 12 $Y_\phi = G_\phi(t_1, t_2) \pm \varepsilon G_\phi(t_1, t)$ our goal is to evaluate the error propagation from the top level of subsampling ϕ to the bottom one and show that $Y_0 = G(t_1, t_2) \pm \varepsilon G(t_1, t)$.

Consider r.v. X_j:

$$X_j = \sum_{i \in HH_j} g(f_i) + 2 \sum_{i \notin HH_j} H_j(i) g(f_i).$$

X_j is an unbiased estimator of $G_j(t_1, t_2)$ with variance bounded as:

$$Var(X_j) = \sum_{i \notin HH_j} g^2(f_i) \leq \varepsilon G_j(t_1, t) \sum g(f_i) \leq \varepsilon G_j^2(t_1, t)$$

by definition of HH_j and monotonicity of G. Therefore, by conditioning on HH_j success and Chebyshev inequality:

$$\Pr(|X_j - G_j(t_1, t_2)| \geq \varepsilon' G_j(t_1, t)) \leq \frac{\varepsilon}{\varepsilon'^2} + \delta. \tag{3.2}$$

By definition of H_j, X_j can be rewritten as

$$X_j = 2G_{j+1} + \sum_{i \in HH_j} (1 - 2H_{j+1}(i))g(f_i).$$

Then, $|X_j - Y_j| \leq 2|G_{j+1} - Y_{j+1}| + \sum_{i \in HH_j} |\hat{g}(f_i) - g(f_i)|$. To simplify further derivations, denote $E_j^1 = |X_j - G_j|$, $E_j^2 = |G_j - Y_j|$ and $E_j^3 = \sum_{i \in HH_j} |\hat{g}(f_i) - g(f_i)|$.

$$E_j^2 = |G_j - Y_j| \leq |G_j - X_j| + |X_j - Y_j| \leq E_j^1 + 2E_{j+1}^2 + E_j^3.$$

Therefore, the error will propagate to layer 0 as: $E_0^2 \leq E_0^1 + 2E_1^2 + E_0^3 \leq \ldots \leq 2^\phi E_\phi^2 + \sum_{j=0}^{\phi} 2^j E_j^1 + \sum_{j=0}^{\phi} 2^j E_j^3$. Denote event $2^\phi E_\phi^2 \geq \varepsilon'' G(t_1, t)$ as A, event $\sum_{j=0}^{\phi} 2^j E_j^1 \geq \varepsilon'' G(t_1, t)$ as B, and event $\sum_{j=0}^{\phi} 2^j E_j^3 \geq \varepsilon'' G(t_1, t)$ as C.

Apply formula 3.2 for all j, then $\Pr(B)$ is upper bounded by:

$$\Pr\left(\varepsilon' \sum_{j=0}^{\phi} 2^j G_j(t_1, t) \geq \varepsilon'' G(t_1, t)\right) + (\phi + 1)\left(\frac{\varepsilon}{\varepsilon'^2} + \delta\right)$$

Note, that $E\left(\sum_{j=0}^{\phi} 2^j G_j(t_1, t)\right) = (\phi + 1)G_j(t_1, t)$, therefore by Markov:

$$\Pr(B) \leq (\phi + 1)\frac{\varepsilon'}{\varepsilon''} + (\phi + 1)\left(\frac{\varepsilon}{\varepsilon'^2} + \delta\right).$$

Recall that HH_j fails with probability at most δ, there are at most $1/\varepsilon'$ items such

that $g(f_i) \geq \varepsilon G_j(t_1, t)$, and if HH_j succeeds then $\hat{g}(f_i) = (1 \pm \varepsilon)g(f_i)$ for all i such that $g(f_i) \geq \varepsilon G_j(t_1, t)$. Therefore, $\sum_{i \in HH_j} |\hat{g}(f_i) - g(f_i)| < \varepsilon G_j(t_1, t)$, and we can bound $\Pr(C)$ from above with:

$$\Pr\left(\varepsilon \sum_{j=0}^{\phi} 2^j G_j(t_1, t) \geq \varepsilon'' G(t_1, t)\right) + (\phi + 1)\delta.$$

Finally, applying Markov: $\Pr(C) \leq (\phi + 1)\frac{\varepsilon}{\varepsilon''} + (\phi + 1)\delta$.

To bound $\Pr(A)$ recall that $Y_\phi = G_\phi(t_1, t_2) \pm \varepsilon G_\phi(t_1, t)$ with probability at least $1 - \delta$ and $E(2^\phi G_\phi(t_1, t)) = G(t_1, t)$. Therefore, $P(A) \leq \frac{\varepsilon}{\varepsilon''} + \delta$, and putting all together: $P(A \cup B \cup C) \leq (\phi + 2)\frac{\varepsilon}{\varepsilon''} + (\phi + 1)\frac{\varepsilon'}{\varepsilon''} + (\phi + 1)(\frac{\varepsilon}{\varepsilon'^2} + \delta)$. Choosing $\varepsilon \leq \frac{0.1\varepsilon''^2}{(\phi+1)^3}$ and $\varepsilon' \leq \frac{0.1\varepsilon''}{\phi+1}$ we get the statement of the theorem

$$P(|Y_0 - G(t_1, t_2)| \geq \varepsilon'' G(t_1, t)) \leq 0.3.$$

□

Extending to wider class of queries Recall that the interval queries (t_1, t_2) introduced earlier were not measured in time, but rather in number of packets passed through. However, in practice it is often more use when one can query some statistic in time-based interval, for example one hour interval half a day ago or from 5PM to 6PM yesterday. All presented algorithms, are easily extendable to answer time-based interval queries, one only need to create a time stamp for each bucket of SH or EH framework, and use that timestamp when searching for corresponding buckets approximating the interval. Note that all presented algorithms support weighted packets, i.e. when each packet i arrives with it's weight w_i, which corresponds to the update $f_i = f_i + w_i$.

3.4 Evaluation

Next, we evaluate our algorithms for various network measurement tasks. We have implemented a prototype in C and evaluated the accuracy v.s. memory using four CAIDA Internet Traces: "Equinix-Sanjose" in 2014 (SanJose2014) [74] "Equinix-Chicago" in 2016 (Chicago2016) [139], and from "Equinix-NewYork" in 2018 (NewYork2018) [137]; and a data center trace from the University of Wisconsin (Univ2) [17].

All experiments are based on 10M sized traces. We evaluate multiple measurement metrics on two experiments. In the first experiment, we select a packet once every 30k packets and estimate the frequency of the corresponding flow on an interval between 10k-20k packets ago (suffix length of 20k). In the second, we estimate frequencies on varying suffixes and show how the suffix length affects the empirical error. To do so, we select a packet once per 200k packets and estimate the frequency of the corresponding flow for every possible suffix length from 100K to the window size of 1M. The depicted figures for the second experiment are for the NewYork2018 dataset.

Frequency estimation: We with the frequency estimation problem. The results in Figure 3.3 show the trade-off between the memory consumption and the empirical error for different network traces. As expected, having more memory increases the accuracy of frequency estimation. The difference between the traces is mainly attributed to the workload characteristics and namely how the L_2 norm changes during each trace.

Figure 3.4 shows results for the second experiment on the NY2018 trace. Notice that (i) longer suffixes indeed have larger estimation error than shorter ones. This is expected as our analysis indicates that the error is proportional to the suffix length. (ii) Notice that more memory increases accuracy for every suffix length. This is also expected as the error is proportional to the accuracy parameter ε which decreases with the memory.

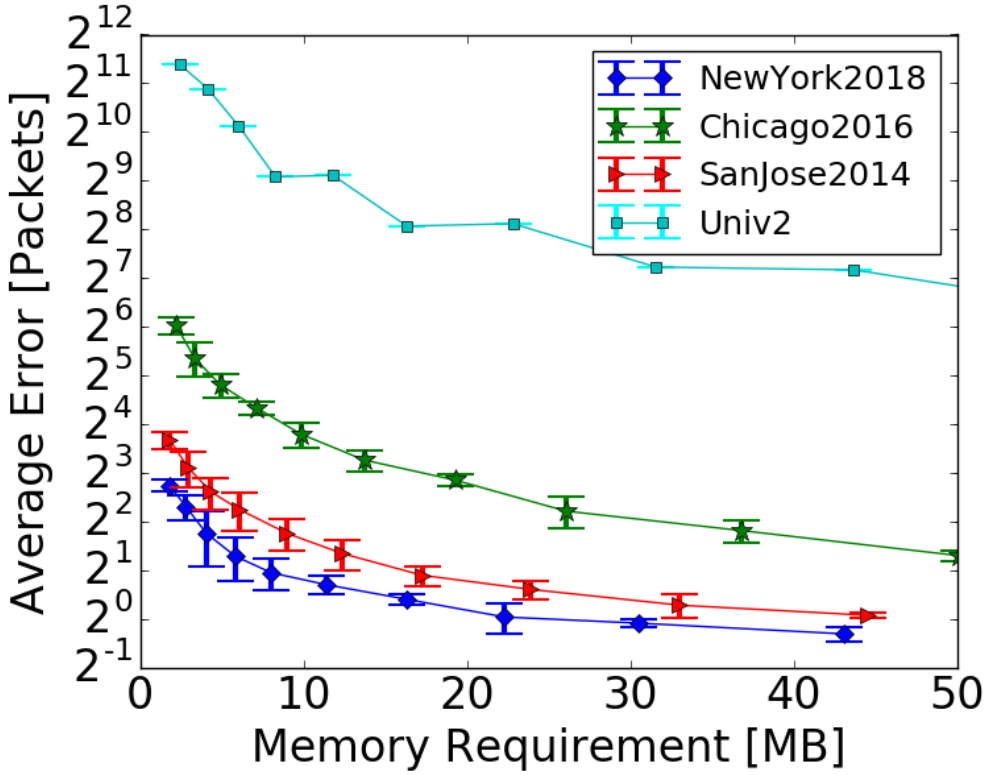


FIGURE 3.3: Average frequency estimation error for flows in 10-20k interval.

(iii) the average error is small for moderate memory consumption, e.g. given 18MB, the average error is less than 64 packets for all suffix lengths.

L_2 **norm estimation:** We repeat the above experiments for L_2 norm estimation. Figure 3.5 shows results for the first experiment. Notice that we get the same trend as before, more memory leads to better accuracy in estimating the L_2 norm. Figure 3.6 shows results for the second experiment. As can be seen, (i) we get better L_2 estimations for small suffixes, (ii) additional memory increases the accuracy.

Heavy hitters estimation: We now evaluate L_2 heavy hitters, with $\theta = 0.5\%$. We evaluate three metrics for heavy hitters estimation. (i) Precision: measure how many of the reported flows are indeed heavy hitters, (ii) Recall: measure how many of the real heavy hitters are reported, and the F_1 norm that factors both Precision and Recall into a

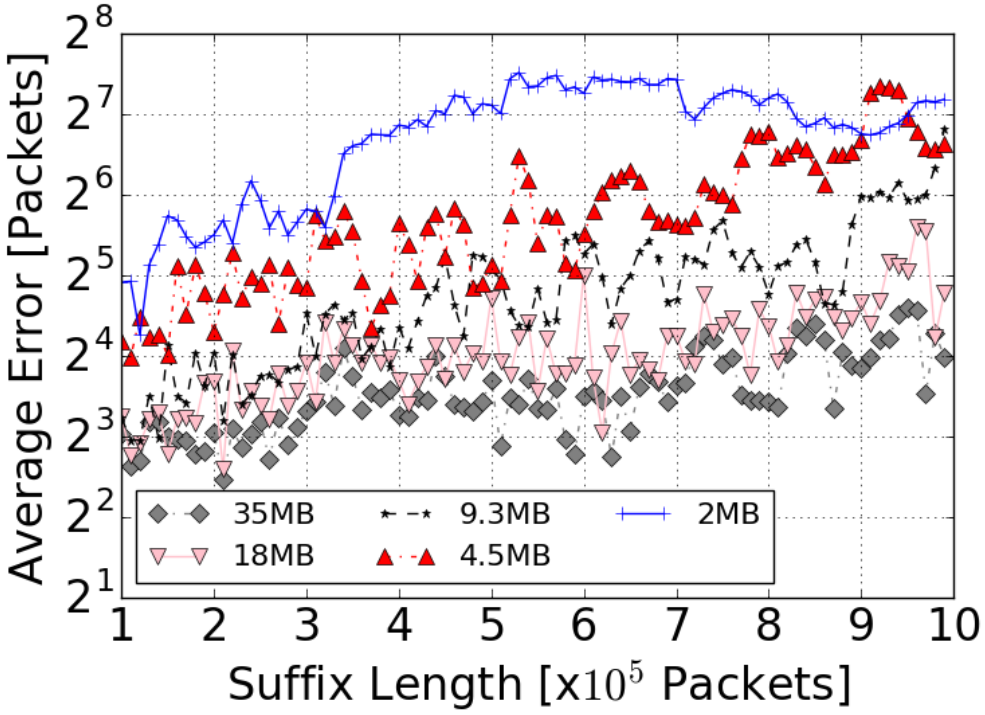


FIGURE 3.4: Average frequency estimation error for flows for various suffix lengths on the NY2018 dataset.

single measure. The perfect solution has $F_1 = 1$, and the closer the value is to 1, the more accurate the solution.

Figure 3.7 shows precision (solid curves) and recall (dashed) for various workloads for the first experiment. Notice that (i) increasing memory increases both precision and recall. (ii) that we get to around 90% recall with enough memory, and 95+% accuracy. Figure 3.8 shows the F_1 values in the same experiment. As can be observed, the value converges close to 1 as we increase the memory.

Figures 3.9, 3.10, and 3.11 shows result for the second experiment. Figure 3.9 shows precision, Figure 3.10 shows recall and Figure 3.11 shows the F_1 metric. As can be observed, (i) longer suffixes yield less accurate results. For precision, there is a slight anomaly for 2 and 4.5 MB sized algorithm. Note that however, at the same time recall drops, and

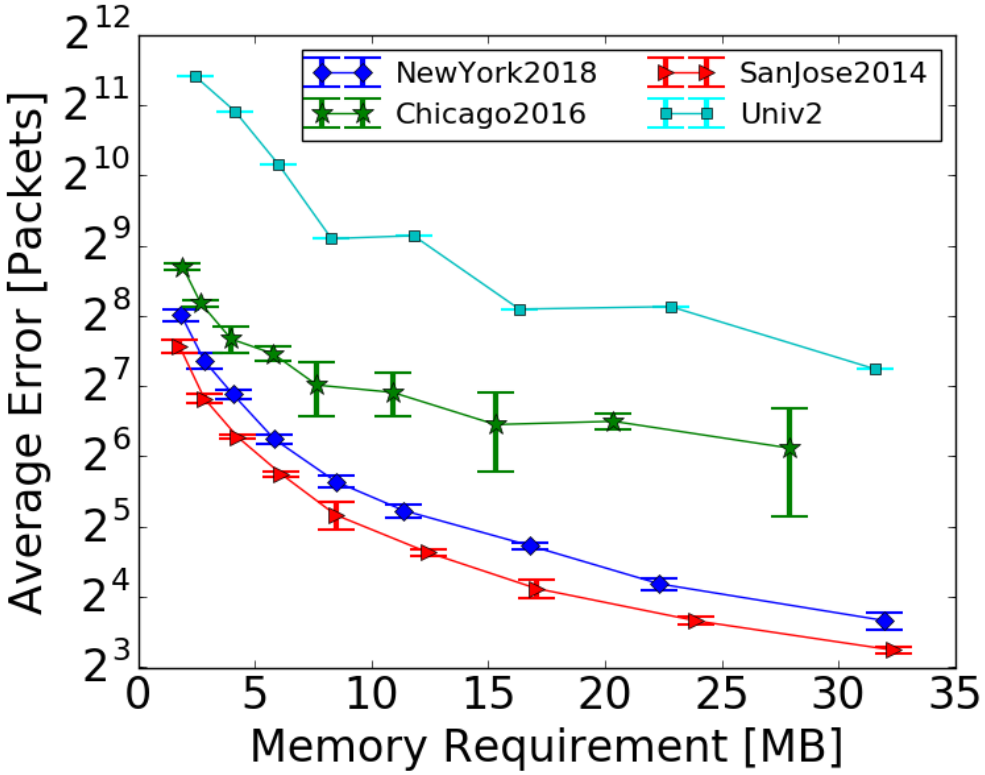


FIGURE 3.5: Average L_2 norm estimation error for flows in 10-20k intervals.

the overall quality (F_1) remains constant. This means that less heavy hitters are detected, but less non heavy hitters are falsely detected. (ii) increasing memory improves all quality metrics for any suffix length.

Entropy estimation: Finally, we evaluate the error in Entropy estimation. We used Algorithm 4, and extended UnivMon [101] to provide Entropy estimations. Figures 3.12 and 3.13 show the results. As illustrated, we estimate the entropy for this interval very accurately throughout the range of memory. As expected, more memory means a better estimation.

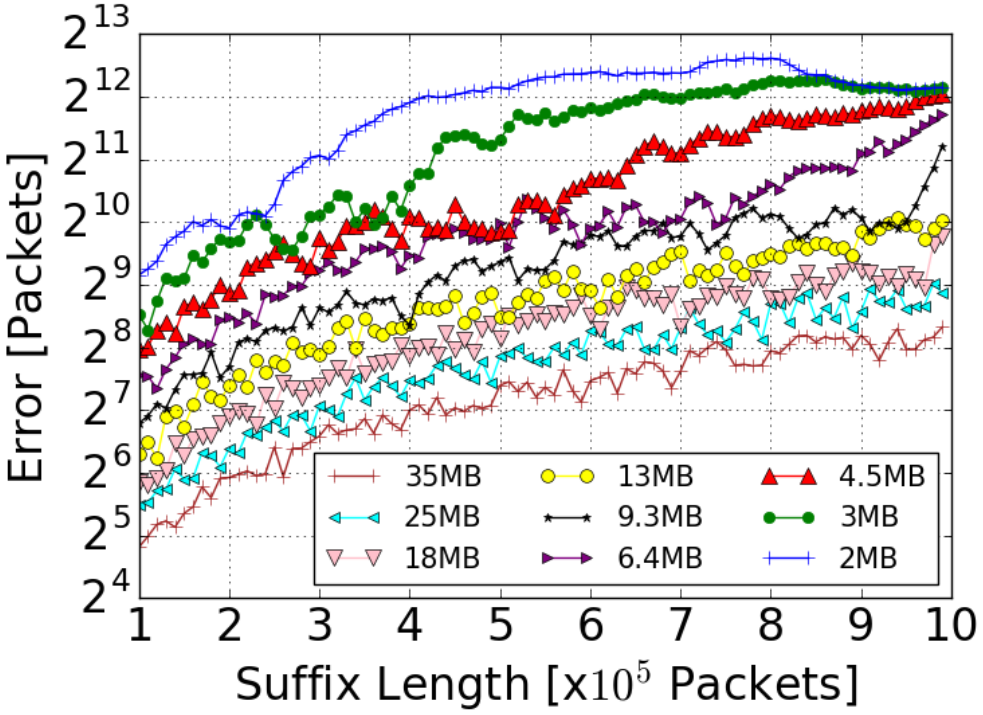


FIGURE 3.6: Average L_2 norm estimation error for flows for various suffix lengths on the NY2018 dataset.

3.5 Conclusion

Our study investigates the IQ model that allows calculating metrics on any intervals within a recent window. Such capabilities are often provided by databases that maintain a full subset of the data. Our novelty lies in providing such capabilities in a space-efficient manner which is considerably smaller than any database. Thus, it is a step towards realizing efficient interval queries in network devices. We anticipate that such capabilities would enable a new generation of advanced network algorithms, given the access to more fine-grained primitives. For example, security applications would be able to compare a variety of measurement metrics in different time scales to more accurately identify attacks.

Our work studies how to extend L_2 heavy hitter algorithms to the IQ model. We justify

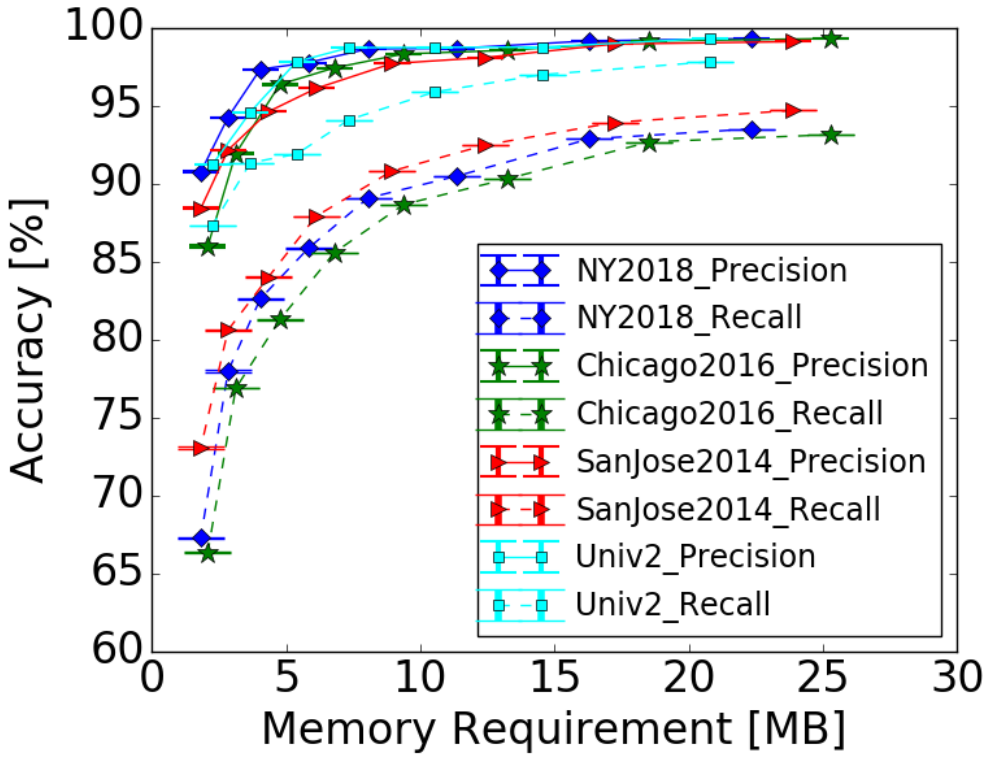


FIGURE 3.7: Quality of HH solution for 10k-20k interval (first experiment). Precision and Recall.

the focus on L_2 as it is used as a building block in monitoring a large variety of network measurement tasks such as frequency estimation, L_2 norm, the number of distinct flows, and the entropy of traffic distribution. We suggest three different techniques to do so, each improving on the space requirement of the previous one. We also extend the analysis of the previously suggested Exponential Histogram [46], which enables it to be used as a building block in the IQ model. Finally, we use our best algorithm to extend UnivMon capable of all the network tasks mentioned above [101] to the IQ model, based on our best L_2 heavy hitter algorithm. Our work is the first to provide these measurement tasks in the IQ model, and through an extensive evaluation on real Internet traces, we show that these capabilities are practical within the available memory range of network devices.

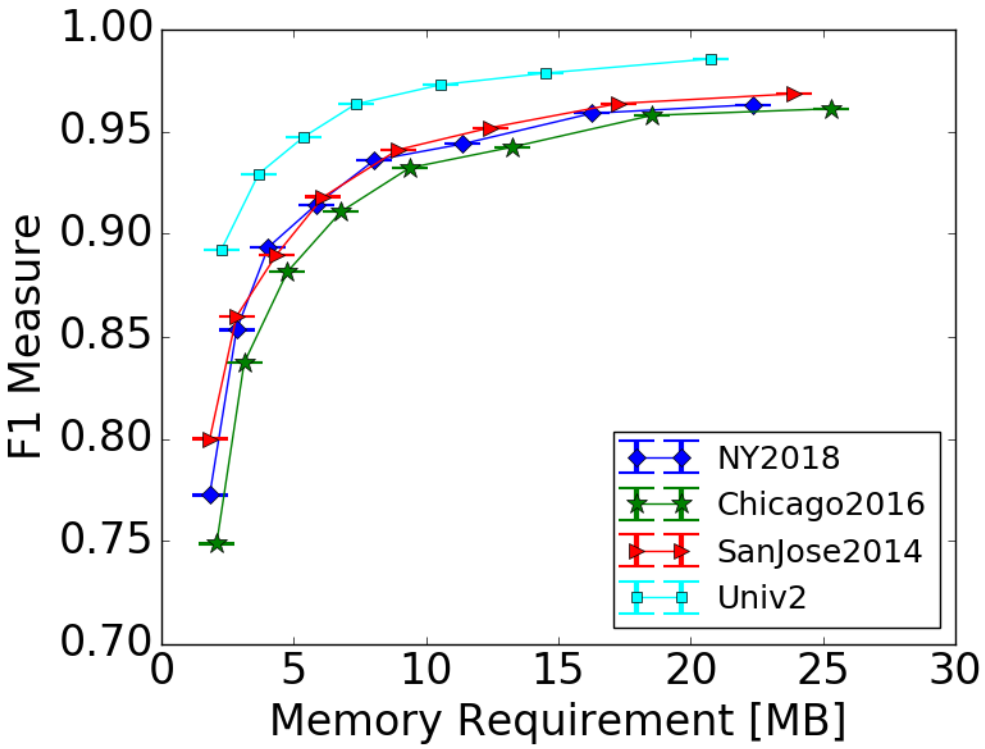


FIGURE 3.8: Quality of HH solution for 10k-20k interval (first experiment). F_1 Measure.

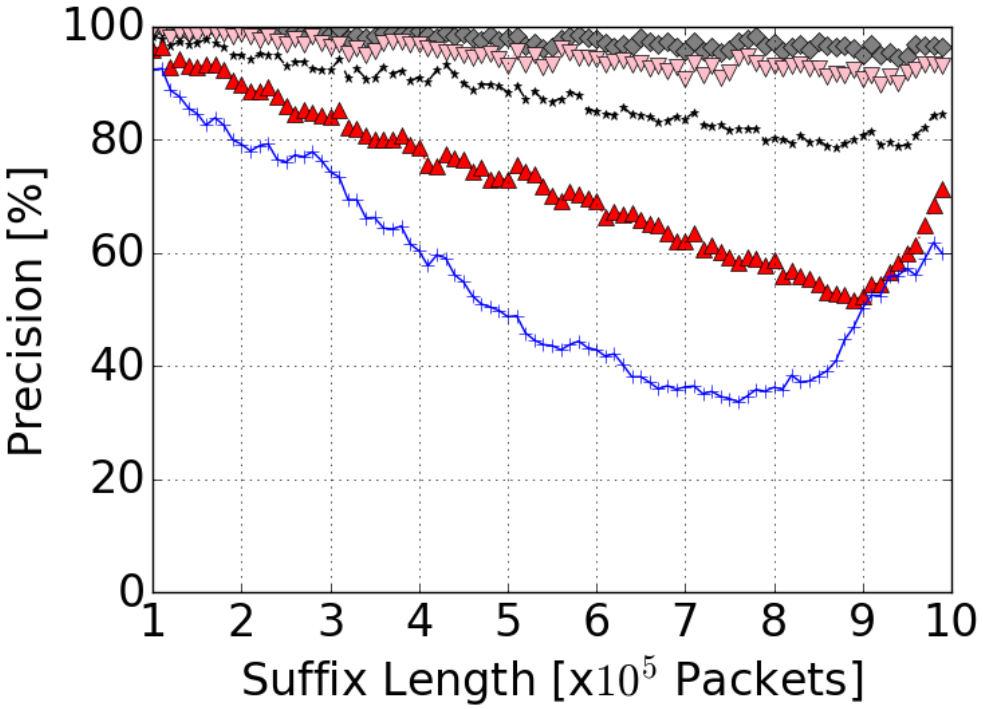


FIGURE 3.9: Quality of HH solution for varying suffix lengths (second experiment). Precision.

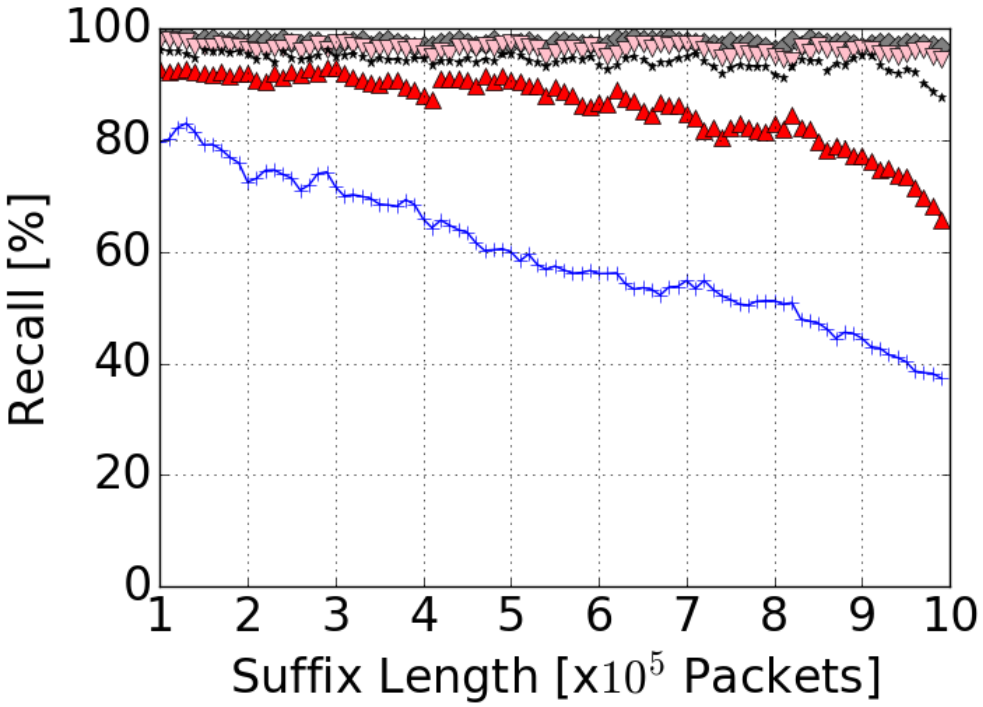


FIGURE 3.10: Quality of HH solution for varying suffix lengths (second experiment). Recall.

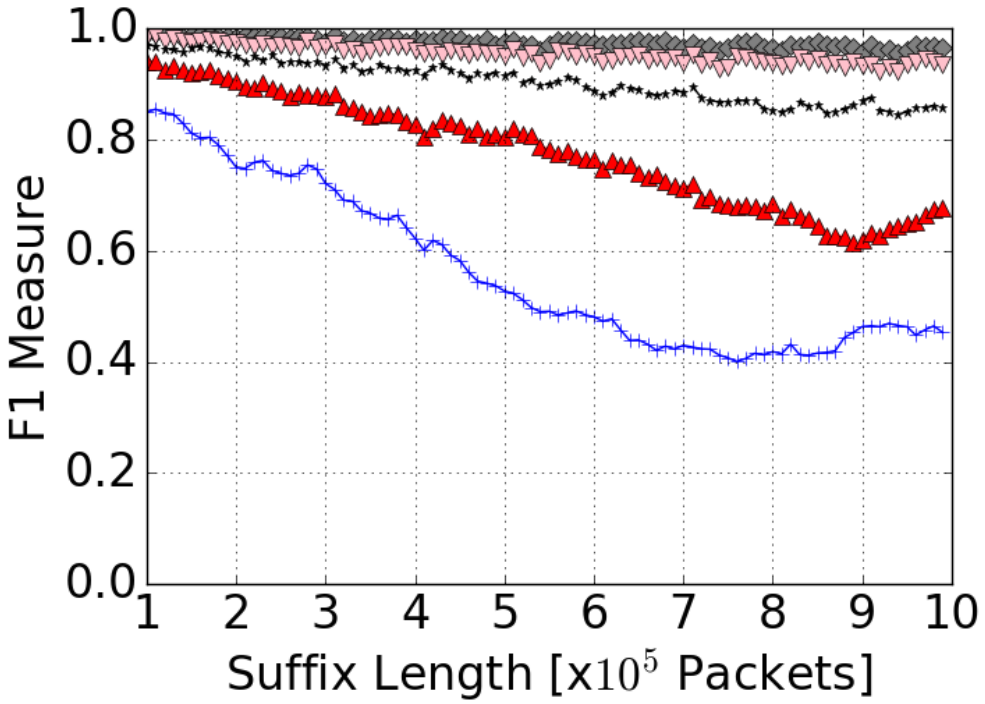


FIGURE 3.11: Quality of HH solution for varying suffix lengths (second experiment). F_1 Measure.

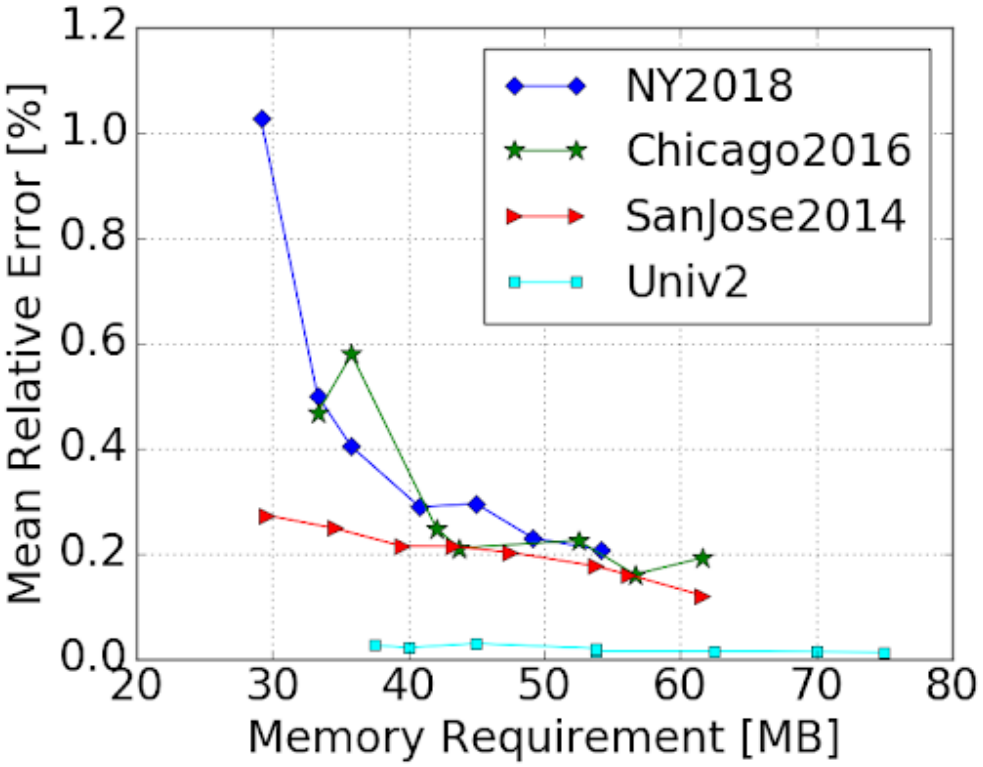


FIGURE 3.12: Average entropy relative error for 10-20k intervals.

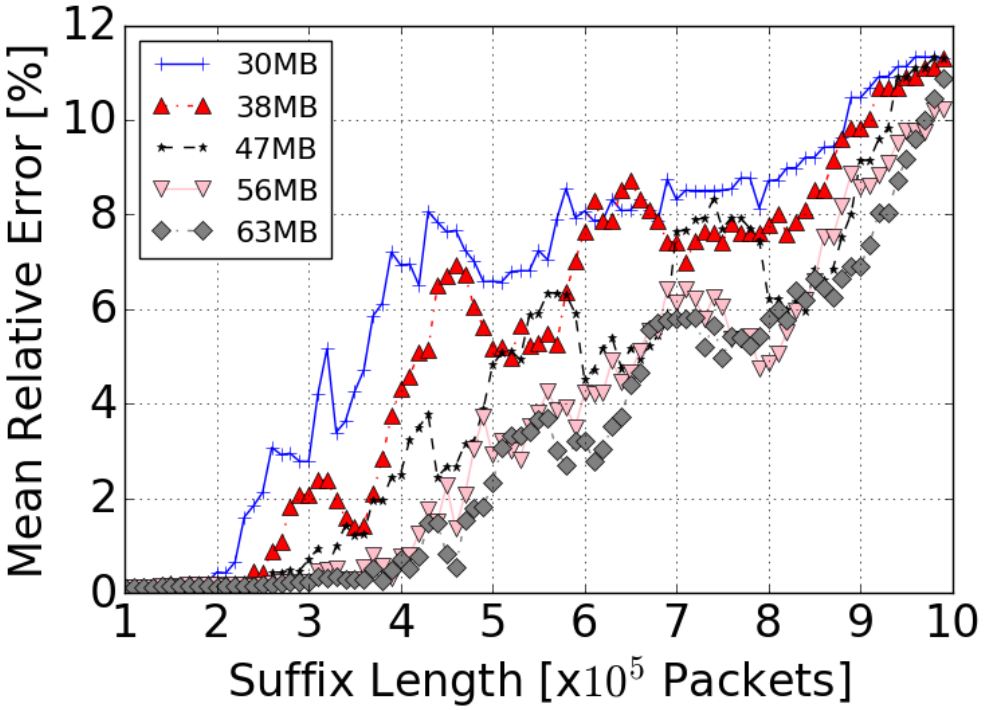


FIGURE 3.13: Average entropy relative error for various suffix lengths on the NY2018 dataset.

Chapter 4

Streaming quantiles algorithms with small space and update time

Current chapter is based on the work done in collaboration with Liberty E., Karnin Z., Lang K. and Braverman V.

4.1 Introduction

Estimating the underlying distribution of data is crucial for many applications. It is common to approximate an entire Cumulative Distribution Function (CDF) or specific quantiles. The median (0.5 quantile) and 95-th and 99-th percentiles are widely used in financial analytics, statistical tests, and system monitoring. Quantiles summary found applications in databases [129, 123], sensor networks [98], logging systems [121], distributed systems [49], and decision trees [40]. While computing quantiles is conceptually very simple, using compressed summaries becomes unavoidable for very large data.

Formally the quantiles problem can be defined as follows: data structure inputs a multiset $S = \{s_i\}_{i=1}^{n}$, and upon a query ϕ it should output an item with the rank ϕn, i.e. ϕn-th item in the sorted S. An approximate version of the problem relaxes the requirement and

admits any item with rank in $[(\phi - \varepsilon)n, (\phi + \varepsilon)n]$. In a randomized version, the algorithm is allowed to make a mistake with a probability at most δ. Note that in order for the randomized version to provide a correct answer to all possible queries, it suffices to amplify the probability of success, by running the algorithm with failure probability of $\delta\varepsilon$, and applying the union bound over all $O(1/\varepsilon)$ quantiles.

In network monitoring [101] and other applications it is critical to maintain statistics while making only a single pass over the data and minimizing the communication and update time. As a result, the problem of quantiles has been considered in several models: distributed settings [49, 65, 130], continuous monitoring [45, 147], streaming [2, 85, 104, 105, 143, 64, 66], sliding windows [8, 100]. In the present chapter, the quantiles problem is considered in a standard streaming settings: at time 0 the initial set S_0 is empty, and at every time moment $t \in [n]$ the data structure is updated with new item s_t, i.e. $S_t = S_{t-1} \cup \{s_t\}$, at the end of the stream, the data structure is queried with a quantile ϕ. The algorithm's space complexity and approximation guarantees should not depend on the order or the content of the updates s_t, and should depend on n at most sublinearly. In the pioneering paper [112], Munro and Paterson showed that one would need $\Omega(n^{1/p})$ space and p passes over the dataset to find a median. They also suggested an optimal iterative algorithm to find it. Later Manku *et al*. [104] showed that the first iteration of the algorithm in [112] can be used to solve the ε approximate quantile problem in one pass using only $O(1/\varepsilon \log^2 n)$ words of space. Note that, for a small enough ε, this is a significant improvement over the naive algorithm, which samples $O(1/\varepsilon^2 \log(1/\varepsilon))$ items of the stream using reservoir sampling. The algorithm in [104] is deterministic, however, compared with the reservoir sampling it assumes the length of the stream to be known in advance. In many applications such an assumption is unrealistic. In their follow-up

paper [105] the authors suggested a randomized algorithm without that assumption. Further improvement by Agarwal *et al*. [2] via randomizing the core subroutine, pushed the space requirements down to $O(1/\varepsilon \log^{3/2}(1/\varepsilon))$. In addition, new data structure has been proven to be fully mergeable. Greenwald and Khanna in [64] presented an algorithm that maintains upper and lower bounds for each quantile individually, rather than one bound for all quantiles. It is deterministic and requires only $O(1/\varepsilon \log \varepsilon n)$ words of memory, however, it is not known to be fully mergeable. Later Felber and Ostrovsky [57] suggested novel techniques of feeding sampled items into a sketch from [64] and improved the space complexity to $O(1/\varepsilon \log 1/\varepsilon)$ words of memory.

Recently Karnin *et al*. in [85] presented an asymptotically optimal but non-mergeable data structure with space usage of $O(1/\varepsilon \log \log 1/\varepsilon)$ words. In the same paper authors proved matching lower bound and presented another algortihm which is fully mergeable however with space requirement of $O(\frac{1}{\varepsilon} \log^2 \log \frac{1}{\varepsilon})$ words. In the current chapter, we suggest several further improvements to the algorithms introduced in [85]. These improvements do not affect the asymptotic guarantees of [85], but improve the constant terms, both in theory and practice. Also suggested techniques will improve the worst-case update time. Additionally we suggest 2 algorithms for the extended version of the problem when each update comes with a weight. All the algorithms discussed in this chapter are in the comparison model, i.e. algorithm should explicitly store the items it will output, therefore it can not "compute" the item in the contrary to the fixed universe model. For the more detailed review of the quantiles algorithms in the streaming model we refer the reader to [66, 143].

4.2 A unified view of previous randomized solutions

To introduce further improvements to the streaming quantiles algorithms we will first re-explain the previous work using simplified concepts of one pair compression and a compactor. Consider a simple problem in which your data set contains only two items a and b, while your data structure (DS) can only store one item. We focus on the comparison based framework where we can only compare items and cannot compute new items via operations such as averaging. In this framework, the only option for the DS is to pick one of them and store it explicitly. The stored item x is assigned weight 2. Given a rank query q the DS will report 0 for $q < x$, and 2 for $q > x$. Note that for $q \notin [a, b]$ the output of the DS will be correct; however, for $q \in [a, b]$ the correct rank is 1, the DS will introduce a $+1/-1$ error depending on which item the DS stored. From here we will call q with respect to the compaction (a, b) as inner query if $q \in [a, b]$ and an outer query otherwise; such notion lets us distinguish those queries for which an error is introduced, from those that were not influenced by a compression. Figure 4.1 depicts the above example of *one pair compression*.

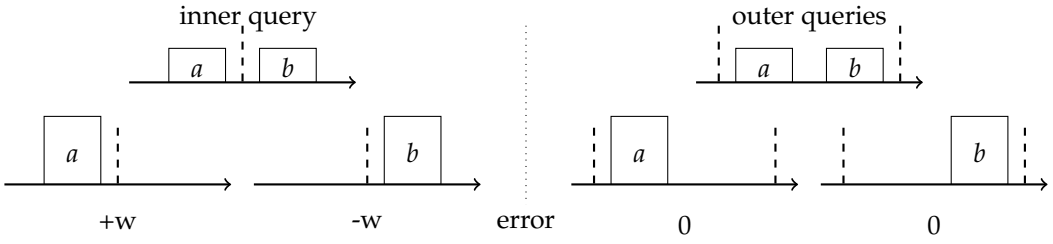


FIGURE 4.1: One pair compression: initially each item has weight w, compression introduces $\pm w$ error for inner queries and no error for outer queries.

The example gives a rise to a high-level method for the original problem with the dataset of size n and memory capacity of k items: 1)keep adding items to the DS until it is full; 2)choose any pair of items with the same weight and compress them. Notice that if

we choose those pairs without care, in the worst-case we might end up representing the full dataset by its top k elements, introducing an error of almost n, that is much larger than εn. It follows that we should make sure that the pairs being compacted (i.e. compressed) should have their ranks as close as possible, thereby affecting as few queries as possible.

This intuition is implemented via a *compactor*. First introduced by Manku *et al*. in [104], it defines an array of k items with weight w each, and a compaction procedure which compress all k items into $k/2$ items with weight $2w$. A compaction procedure first sorts all items, then deletes either even or odd positions, and doubles the weight of the rest. Figure 4.2 depicts the error introduced for different rank queries q, by a compaction procedure applied to an example array of items $[1, 3, 5, 8]$. Notice that the compactor utilizes the same idea as the one pair compression, but on the pairs of neighbors in the sorted array; thus by performing $k/2$ non-intersecting compressions it introduces an overall error of w as opposed to $kw/2$.

The algorithm introduced in [104], defines a stack of $H = O(\log n/k)$ compactors, each of size k. Each compactor obtains as an input a stream and outputs a stream with half the size by performing a compact operation each time its buffer is full. The output of the final compactor is a stream of length k that can simply be stored in memory. The

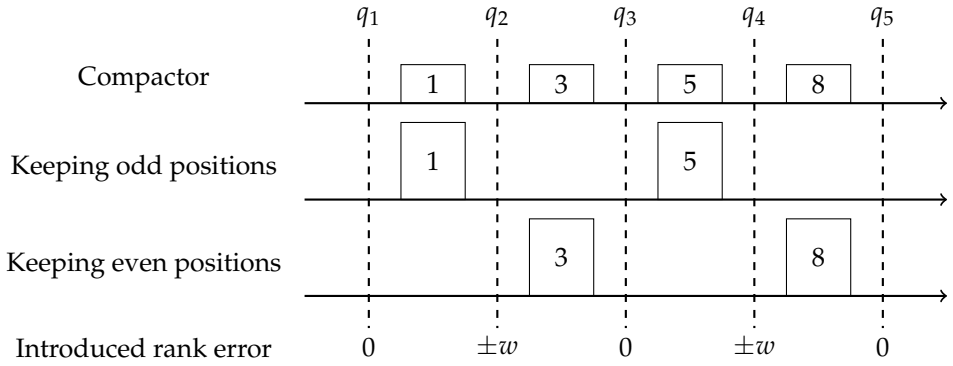


FIGURE 4.2: Compaction procedure: rank error $\pm w$ is introduced to inner queries $q_{2,4}$, no error to outer queries $q_{1,3,5}$

bottom compactor that observes items has a weight of 1; the next one observes items of weight 2 and the top one 2^{H-1}. The output of a compaction on h-th level is an input of the compactor on $(h+1)$-th layer. Note that error introduced on h-th level is equal to the number of compactions $m_h = n/kw_h$ times the error introduced by one compaction w_h. The total error can be computed as

$$\text{Err} = \sum_{h=1}^{H} m_h w_h = Hn/k = O(n/k \log n/k).$$

Setting $k = O(1/\varepsilon \log \varepsilon n)$ will lead to an approximation error of εn. Space is used by H compactors of size k each — $O(1/\varepsilon \log^2 \varepsilon n)$ words. Note that the algorithm is deterministic.

Later, Agarwal *et al*. [2] suggested the compactor to choose the odd or even positions randomly and equiprobably, pushing the introduced error to zero in expectation. Additionally, authors suggested a new way of feeding subsampled stream into the data structure, recalling that $O(1/\varepsilon^2 \log 1/\varepsilon)$ samples preserve quantiles with $\pm\varepsilon n$ approximation error. Proposed algorithm requires $O(1/\varepsilon \log^{3/2} 1/\varepsilon)$ words of space and succeed with high constant probability.

To prove the result the authors introduced a random variable $X_{i,h}$ denoting the error introduced on the i-th compaction at h-th level. Then the overall error is:

$$\text{Err} = \sum_{h=1}^{H} \sum_{i=1}^{m_h} w_h X_{i,h}.$$

$w_h X_{i,h}$ is bounded, has mean zero and is independent of the other variables. Thus, due to Hoeffding's inequality:

$$P(|\text{Err}| > \varepsilon n) \leq 2 \exp \frac{-\varepsilon^2 n^2}{\sum_{h=1}^{H} \sum_{i=1}^{m_h} w_h^2}.$$

Setting $w_h = 2^{h-1}$ and $k = 1/\varepsilon\sqrt{\log 1/\delta}$ will keep the error probability bounded by δ.

The following improvement by Karnin *et al*. [85] suggested to:

1. use exponentially decreasing size of the compactor,
2. keep only top $O(\log 1/\varepsilon)$ top compactors,
3. keep the size of the top $O(\log\log 1/\delta)$ compactors fixed,
4. replace the top $O(\log\log 1/\delta)$ compactors with GK sketch [64].

(1) and (2) dropped the space down to $O(\frac{1}{\varepsilon}\sqrt{\log\frac{1}{\varepsilon}})$, (3) pushed it further to $O(\frac{1}{\varepsilon}\log^2\log\frac{1}{\varepsilon})$, and (4) led to an optimal $O(\frac{1}{\varepsilon}\log\log\frac{1}{\varepsilon})$. The authors also provided the matching lower bound. Note, the last solution is not mergeable due to the use of GK [64] as a subroutine.

While (3) and (4) lead to the asymptotically better algorithm, its implementation is quite complicated for application purposes and mostly are of a theoretical interest. For these reasons, in the current chapter, we build upon the KLL algorithm of [85] using only (1) and (2).

In [85], the authors show that if the size of the compactor decreases as $k_h = c^{h-h}k$ for $c \in (0.5, 1)$, then $\sum_{h=1}^{h}\sum_{i=1}^{m_h} w_h^2 \leq \frac{n^2/k^2}{2c^2(2c-1)}$. As previously, applying Hoeffding inequality results in:

$$P(|\text{Err}| > \varepsilon n) \leq 2\exp\left(-C\varepsilon^2 k^2\right) \leq \delta,$$

where $C = 2c^2(2c-1)$. Setting $k = O(1/\varepsilon\sqrt{\log 1/\varepsilon})$ lead to the desired approximation guarantee for all $O(1/\varepsilon)$ quantiles with constant probability. Note that the smallest meaningful compactor has size 2, thus the algorithm will require

$$k(1 + c + \ldots + c^{log_{1/c}k}) + O(\log n) = \frac{k}{1-c} + O(\log n)$$

compactors, where the last term is due to the stack of compactors of size 2. The authors suggested replacing that stack with a basic sampler, which picks one item out of every $2^{w_{H-log_{1/c}k}}$ updates at random and logically is identical but costs only $O(1)$ words. The resulting space complexity is $O(\frac{1}{\varepsilon}\sqrt{\log\frac{1}{\varepsilon}})$. We provide the pseudocode for the core routine in Algorithm 13.

Algorithm 13 Core routines for KLL algorithm [85]

```
1: function KLL.UPDATE(item)
2:     if SAMPLER(item) then KLL[0].APPEND(item)
3:     for h = 1 ... H do
4:         if LEN(KLL[h] ≥ k_h) then KLL.COMPACT(h)
5:     end for
6: end function
7: function KLL.COMPACT(h)
8:     KLL[h].SORT(); rb = RANDOM({0,1});
9:     KLL[h + 1].APPEND(KLL[h][rb : : 2])
10:    KLL[h]= []
11: end function
```

4.3 Our Contribution

In the effort of obtaining a practical method, during the chapter we maintain a convenient API for the algorithm where the only parameter it requires is the memory limit, as opposed to knowing the values of different parameters such ε, δ in advance. Our aim is to obtain the best possible guarantee given that memory limit. Such optimization (minimizing error under fixed memory) is widely used in practical applications. Below we present a series of modifications that improve the approximation error both in theory and practice. We then compare all of them experimentally in Section 4.4.

Lazy compactions We suggest all the compactors to share the pool of allocated memory and perform compaction only when the pool is fully saturated. This way each compaction will be performed on the larger set of items, and the total number of compactions will be lower. Each compaction introduces fixed ammount of error thus total error introduced will be lower. Algorithm 14 gives the formal lazy-compacting algorithm, and Figure 4.3 visualizes its advantage: in vanilla KLL all compactors having fewer items than their individual capacities, in lazy KLL this is not enforced due to sharing the pool of memory.

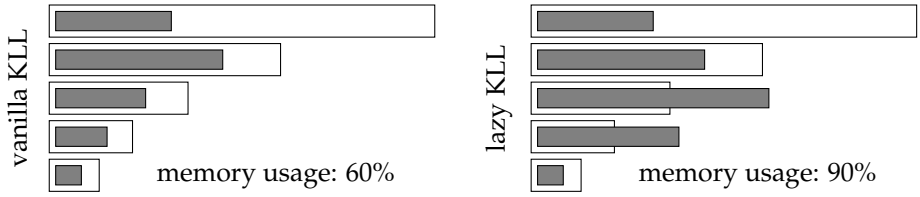


FIGURE 4.3: Compactor saturation: vanilla KLL vs. lazy KLL

Algorithm 14 Update procedure for lazy KLL

```
1: function KLL.UPDATE(item)
2:     if SAMPLER(item) then KLL[0].APPEND(item); itemsN++;
3:     if itemsN > sketchSpace then
4:         for h = 1 ... H do
5:             if LEN(KLL[h]) ≥ k_h then
6:                 KLL.COMPACT(h); break;
7:             end if
8:         end for
9:     end if
10: end function
```

	ii	*io*	*oi*	*oo*	
even → *odd*	0	$-w$	$+w$	0	w.p. 1/2
odd → *even*	0	$+w$	$-w$	0	w.p. 1/2

TABLE 4.1: Possible outcomes for the rank query q.

Reduced randomness Instead of flipping a coin during each compaction, we suggest only doing it for every other compaction, and for others use the opposite to the previous choice. This way, each coin flip defines 2 consecutive compactions: w.p. $\frac{1}{2}$ it is even $\to$ odd ($e \to o$), and w.p. $\frac{1}{2}$ it is odd $\to$ even ($o \to e$). Consider all possible options for a fixed rank query q in two consecutive compactions: io(inner-outer), oi, ii, oo (depicted in the Table 4.1). Clearly, in expectation every two compactions introduce 0 error, additionally, we conclude that total variance introduced while processing the stream is twice smaller compared to vanilla KLL, as in vanilla KLL each compaction introduces an unbiased error bounded in absolute value by w, and in our modification the same variance is introduced by every *two* compactions.

Equally Spread Error Recall that in the analysis of all compactor based solutions [104, 2, 85, 143] during a single compaction we can distinguish two types of rank queries: inner queries, for which some error is introduced, and outer queries, for which no error is introduced. That being said, the existing algorithms do not exploit the fact that outer

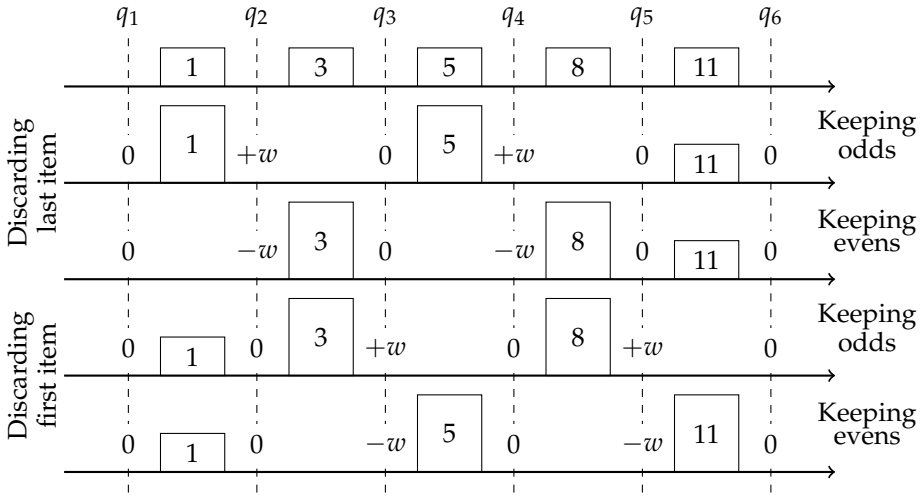


FIGURE 4.4: Compaction with an equally spread error: every query $q_{2,3,4,5}$ is either inner or outer equiprobably.

queries do not present an error, and an adversarial stream might be arranged such that a particular query is internal in all compactions. We suggest to probabilistically "spread" an error of each compaction among all queries. Suppose k is odd, on each compaction we flip a coin and then either compact the items with indices 0 to $k-1$ or 1 to k equiprobably. This way query q w.p.$\frac{1}{2}$ is outer and no error is introduced; w.p. $\frac{1}{4}$ is inner and introduced error is $-w$; and w.p.$\frac{1}{4}$ is inner with error $+w$. Thus, the estimator is still unbiased but the variance is cut in half. We note that the same analysis applies for two consecutive compactions using the reduced randomness improvement discussed in Section 4.3: The configuration (*ii*,*io*,*oi*,*oo*) of a query in two consecutive compactions described in Table 4.1 will now happen with equal probability. Consequently we have the same distribution for the error: 0 w.p. $\frac{1}{2}$, $+w$ w.p. $\frac{1}{4}$ and $-w$ w.p. $\frac{1}{4}$, meaning that the variance is cut in half compared to its worse case analysis without the error-spreading improvement. Thus, if we apply both modifications total variance should become 4 times smaller. The algorithm including the error-spreading improvement is described in Figure 4.4.

Sweep-compactor The error bound for all compactor based algorithms follows from the property that every batch of $k/2$ compressions is disjoint. In this section we suggest to compress pairs one at time, while making sure that every batch of *at least* $k/2$ compression is disjoint. Compacting a single pair takes constant time; hence we reduce the worst-case update time from $O(1/\varepsilon)$ to $O(\log(1/\varepsilon))$. Additionally, for some data streams the disjoint batch size is strictly larger than $k/2$ resulting in a reduction in the overall error.

The modified compactor operates in phases we call *sweeps*. It maintains the same buffer as before and an additional threshold θ initialized as $-\infty$ [1]. As previously, the items in the buffer are stored in non-decreasing sorted order. When a pair must be compacted

[1]Notice that $-\infty$ is still defined in the comparison model

algorithm chooses the pair of two smallest items larger than θ; after the compaction algorithm updates θ to have the value of the largest item of the compacted pair.

If no such pair exists (due to θ being too large) we set θ as $-\infty$, thereby starting a new sweep. Figure 4.5 demonstrates a single sweep, and the pseudocode for the modified compactor is provided in Algorithm 15.

Algorithm 15 Sweep compaction procedure

```
1: function KLLSWEEP.COMPACT(h)
2:     KLL[h].SORT()
3:     i* = argmin_i (KLL[h][i] ≥ KLL[h].θ)
4:     if i* == None then i* = 0;
5:     KLL[h].θ = KLL[i* + 1];
6:     KLL[h].POP(i*+ RANDBIT());
7:     return KLL[h].POP(i*)
8: end function
```

We note two important properties of a sweep in a buffer that can hold k (or $k+1$) items; the proof of these properties is immediate. The first is that the overall number of pairs compacted in each sweep is *at least* $k/2$. The second is that the intervals of all the pairs in a single sweep do not intersect. These two properties together result in the exact same guarantee as the ordinary compactor in the worst-case. However, notice that for an already sorted stream the modified compactor performs only a single sweep, hence in this scenario the resulting error would not be a sum of n/k i.i.d. error terms, each of magnitude $\pm w$ but rather a single error term of magnitude $\pm w$. Though this extreme situation may not happen very often, it is likely that the data admits some sorted subsequences and the average sweep would contain more than $k/2$ pairs. We demonstrate this empirically in our experiments.

Weighted stream extension Consider the stream of updates (a_i, w_i), where each item a_i comes with a weight w_i. After feeding the stream into a data structure, it is queried with

a quantile ϕ, and should report an item x such that:

$$(\phi - \varepsilon)W \leq \sum_{i:a_i<x} w_i \leq (\phi + \varepsilon)W,$$

where $W = \sum_i w_i$ is the total weight of the entire stream.

The naive approach is to transform the input stream S into the stream of unit updates S' and apply one of the algorithms described earlier. However, this might cause update time $O(w_i \log 1/\varepsilon)$ to become very poor if the item's weight is very large, for instance, $w_i = 2^n$.

In the current chapter we suggest two algorithms with improved time performance: **base2update** with $O(\log^2 1/\varepsilon)$ worst-case update time and **base2compactor** with $O(\log 1/\varepsilon)$ worst-case update time. Both algorithms have the same space complexity as KLL.

We can tweak the naive approach to make use of the inner structure of the compactor based algorithms. Instead of breaking the weighted updates into a unitary updates, one

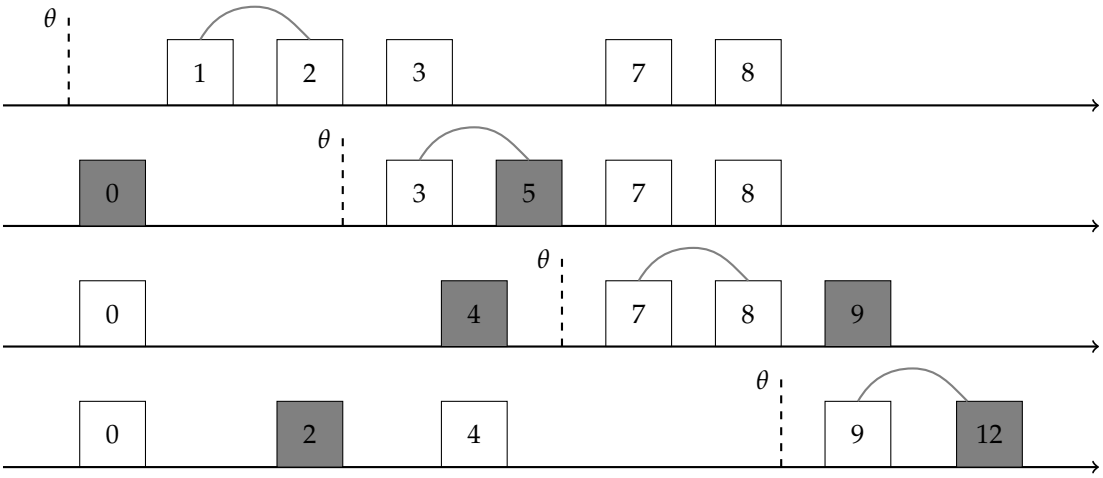


FIGURE 4.5: Example of one full sweep in 4 stages, each stage depicts pair chosen for the compaction, updated threshold θ and new items arrived (shadow bars).

can break it into the power-of-two weights:

$$w_i = \sum_{i=1}^{\log w_i} \Big(\frac{w_i}{2^i} \mod 2 \Big).$$

Note that this operation can be performed very efficiently, because the weight is typically stored in binary representation. Such breakdown guarantees that each update in S would not cause more than $O(\log w_m)$ updates in S', where w_m is the largest weight of the updates. Later we will show how to push it to only $O(\log 1/\varepsilon)$ updates independent of w_m. Thus $|S'| \leq |S| \log w_m$. Recall that all compactor based algorithms have a weight $w_h = 2^h$ assigned to the compactor at the level h. Therefore, instead of adding unitary updates in the lowest compactor, we will distribute our power-of-two weighted updates among corresponding compactors (including feeding into the sampler, if needed). The process is depicted on the Figure 4.6. The algorithm's update procedure is presented in the Algorithm 16.

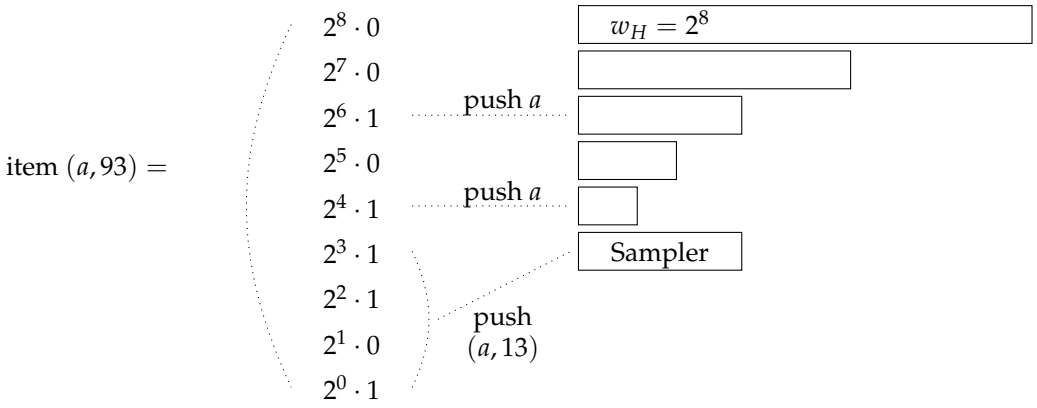


FIGURE 4.6: Intuition behind base2update algorithm

Theorem 44. *Algorithm base2update processes stream of weighted updates and outputs all ε-approximate quantiles with high probability, works in a space $O(1/\varepsilon \log^{1/2} 1/\varepsilon)$ and has an update time $O(\log^2 1/\varepsilon)$.*

Algorithm 16 Base2update update procedure

```
1: function KLL.PUSHITEMS((a, w))
2:     for h = log w − log 1/ε ... log w do
3:         if w^h = w/2^{h−1} mod 2 then
4:             KLL[h].APPEND(a)
5:             Compact if needed
6:         end if
7:     end for
8: end function
9: function KLL.UPDATE((a, w))
10:    if w < 2^{H+1} then
11:        KLL.PUSHITEMS((a, w))
12:    end if
13:    if w ∈ [2^{H+1}, 1/ε2^H] then
14:        KLL.PUSHITEMS((a, w))
15:    end if
16:    if w > 1/ε2^H then
17:        h* = argmax_h (2^{H+h} 1/ε < w)
18:        delete bottom h* compactors
19:        add h* empty compactors on the top
20:        KLL.PUSHITEMS((a, w))
21:    end if
22: end function
```

Proof. Algorithm 16 classifies all updates by its weight into three groups:

1. small — $w < 2^{H+1}$
2. medium — $w \in [2^{H+1}, \frac{1}{\varepsilon}2^H]$x
3. large — $\exists h^* \geq 0 : w > \frac{1}{\varepsilon}2^{H+h^*}$

For small updates its weight can be represented as $\log w_i$ updates with power of 2 weights. Note that the algorithm discards all except the largest $O(\log 1/\varepsilon)$ of them. In doing so it for each update introduces to total error additionaly at most $\sum_i \varepsilon w_i = \varepsilon W$. Then the algorithm adds every update with weight 2^h to the level h compactor directly. This operation does not introduce any error until any compaction procedure is performed.

For large updates its weight is too large to include into compactor. However, it makes the content of the bottom h^* compactor impact negligible compared with w_i. Thus we can delete bottom h^* compactors, and create new h^* compactors on the top. Similarly to the case described earlier, we discard at most εw_i of weight, thus in total it could introduce up to εW error. After new h^* compactors are created, a large update can be classified as a small one and can be processed accordingly.

On the other side medium updates are not large enough to let us discard bottom compactors. As a result, we could not create new compactors. Additionally, it could possibly require up to $O(1/\varepsilon)$ updates to be pushed on the top layer, which pushes the update time exponentially. To avoid such loss in the update time, the compactor would need to support efficient insertion of many identical items at a time: this can be achieved if we allow to store the number of items next to the item itself. As we only need to do it for the items which appear at least twice, this modification should not hurt the space complexity.

Note that in all three cases the error introduced is bounded by $3\varepsilon W$: εW for discarding bottom weights in decomposition of w_i, εW for discarding bottom h^* compactors, εW

from running the KLL algorithm. Each case requires up to $O(\log 1/\varepsilon)$ updates for KLL. We will use sweep-compactor KLL as a core of the base2update algorithm, then the total update time is $O(\log^2 1/\varepsilon)$ and the space complexity $O(1/\varepsilon \log^{1/2} 1/\varepsilon)$. □

The Base2compactor algorithm does not require any stream transformation. To explain the idea behind it we first introduce a weighted compactor and a weighted pair compression. Suppose you are given two pairs (a, w_a) and (b, w_b), such that $a < b$, however the data structure can store only one item. Due to the limitations of the comparison model, as described in section 4.2, the only option is to pick either a or b. Base2compactor chooses a w.p. $\frac{w_a}{w_a+w_b}$ and b w.p. $\frac{w_b}{w_a+w_b}$, assigns weight $w_a + w_b$ for the chosen item and drop the other one. For $q \notin [a, b]$ this operation does not introduce any error, however, for $q \in [a, b]$ the introduced error is w_b w.p. $\frac{w_a}{w_a+w_b}$ and $-w_a$ w.p. $\frac{w_b}{w_a+w_b}$. The expectation of the error is zero and direct computation shows that it's variance is $w_a w_b$. To carefully control the variance we introduce the weighted compactor as an array of pairs $\{(a_i, w_i)\}_{i=1}^{k_h}$ such that $w_i \in [w, 2w)$ (w is a characteristic of given compactor), and the compaction procedure is similar to the unweighted case:

1. sort the array using a_i as an index,
2. break the array into pairs of neighbors $(a_j, w_j), (a_{j+1}, w_{j+1})$
3. compress each pair, using procedure described above

The whole process is depicted in the Figure 4.7. The algorithm's updated procedure for a given input item (a_i, w_i) pushes it into a compactor with w s.t. $w_i \in [w, 2w)$, and then performs a compaction if needed. Note that the input of the compactor on the level with weight w have weights in the range $[w, 2w)$ and output of the compaction on the same level has items with a weight in the range $[2w, 4w)$ thus it can be pushed into the following compactor on the next level.

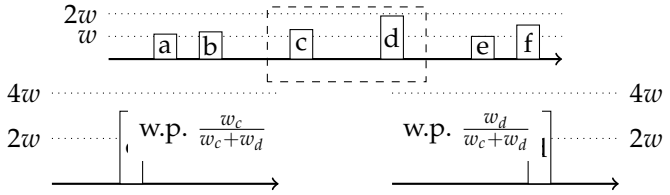


FIGURE 4.7: Compressing pair in the weighted compactor

The analysis for this extension is in the same key as for unitary updates. Let $X_{i,h}$ be a random variable which indicates the sign of the error introduced during the i-th compaction on the h-th level, and let it be equal to zero if no error is introduced. Note that during each weighted compaction when all items have the weights within the range $[w, 2w)$, maximum absolute error is bounded by $2w$, thus total error introduced is

$$\text{Err} = \sum_{h=1}^{H} \sum_{i=1}^{m_h} 2w_h X_{i,h}.$$

Note that this value differs from the one in the analysis of vanilla KLL, only by a factor of 2, thus all the previous derivations made for unitary updates would still hold with one constant adjustment:

$$P(|\text{Err}| > \varepsilon W) \leq 2 \exp\left(-\frac{C}{4}\varepsilon^2 k^2\right) < \delta.$$

To reach the same approximation guarantees with the same probability of fail, one need to set up $k_{new} = 2k_{old}$, i.e. this algorithm will use space twice as much compared to naive implementations. Additionally it requires to store weight for each item explicitly, which might cause up to another factor of two in space complexity (depends on the memory requirement to store one item).

To process any weight updates we suggest to use the same technique as in base2update algorithm. Then base2compactor would require at most $O(1)$ updates to KLL to process each weighted update from the initial stream. Thus if we use sweep-compactor KLL as a

core algorithm we will get the following theorem.

Theorem 45. *Algorithm base2compactor processes stream of weighted updates and outputs all ε-approximate quantiles with high probability, works in a space $O(1/\varepsilon \log^{1/2} 1/\varepsilon)$ and has an update time $O(\log 1/\varepsilon)$*

Note that the importance of the improvement in update time depends a lot on the application, in particular on the relation between n and $1/\varepsilon$, and is debatable. Lets call a weighted stream "good" if it is long enough ($n >> 1/\varepsilon$) and all updates have moderate weights (there is no extreme situations, when weight of the update is $O(1)$ of the total weight seen so far). For "good" streams, as soon as sampling rate grows significantly larger than the typical weight of the update, worst-case update time for KLL-ls (lazy sweep compactors) becomes $O(\log 1/\varepsilon)$ and amortized time drops to $O(1)$.

4.4 Experimental Results

Datasets. To study algorithm's properties we tested it on both synthetic and real datasets, with various of sizes, underlying distributions and orders. Note that approximation guarantees of all investigated algorithms do not depend on the data itself, however it depends on the order in which data set is fed into the algorithm. Surprisingly the worst-case is achieved when the dataset is randomly shuffled []. Therefore we will pay more attention to the randomly ordered data sets in this section, while also experimenting with the semi-random orders that resemble more to real-life applications. Due to the space limitations we could not possibly present all the experiments in the chapter. We encourage the readers to refer to the interactive stand while here we will only present the most interesting and relevant findings.

Our experiments were carried on two types of synthetic datasets. The length of the stream varies from 10^5 to 10^9 for both types of the datasets. In the first type, we fixed the distribution ($a_i \in \{1, \dots, n\}$ and each item appear exactly once) and varied the order of updates: randomly shuffled; sorted order (ascending and descending); semi-sorted (shuffle the entire array, for $i \in \{1 \dots \sqrt{n}\}$ sort locally i-th chunk of $\sqrt{n}$ items); zoom in order ($a_{2i} = i,\ a_{2i+1} = n - i$); zoom out order ($a_{2i} = n/2 + i,\ a_{2i+1} = n/2 - i$). In the second type, we fixed the order to be the random shuffle and checked the following distributions: uniform, Gaussian, mixture of Gaussians. Different variances and number of components were tested with the last two.

To evaluate the performance of KLL and all proposed modifications on the real data we used two datasets and to represent the advantage of the comparison based model both of them contain non-number updates: one contain strings and another one IP addresses:

(1) Anonymized Internet Traces 2015 (CAIDA).[138] The dataset contains anonymized passive traffic traces from the internet data collection monitor which belongs to CAIDA (Center for Applied Internet Data Analysis) and located at an Equinix data center in Chicago, IL. While the dataset has a variety of additional information provided with each packet, for simplicity we will consider only two fields, specifically IP of the source and IP of the destination, as one update for the algorithm. The comparison model is lexicographic. We evaluate the performance on the pieces of the dataset of different sizes: from 10^7 to 10^9. Note that evaluation of the CDF of underlying distribution for traffic flow lets optimize packet managing, thus CAIDA's datasets used widely for verifying different sketching techniques to maintain different statistics over the flow, and finding quantiles and heavy hitters specifically.

(2) Page view statistics for Wikimedia projects (Wiki) [119] The dataset contains counts for the number of requests for each page of the Wikipedia project during 8 months

of 2016. Data is aggregated by day, i.e. within each day data is sorted and each item is assigned with a count of requests during that day. Every update in this dataset is the title of a Wikipedia page. We will consider several modifications of the original dataset:

1. updates in the original order, while ignoring the weights,
2. updates are shuffled within each day, while ignoring the weights,
3. updates are shuffled within each day, while the weight indicates the number of appearances

Similarly to CAIDA dataset we will consider Wiki datasets of size from 10^7 to 10^9. In our experiments, one update is a string which contains the name of the page in Wikipedia, comparison model is lexicographic.

Implementation and evaluation detail All the algorithms and experimental settings are implemented in Python 3.6.3 and are publicly available via GitHub . The advantage of using scripting language is fast prototyping, and in our case it is a chance to quickly test all the new tweaks to the algorithm, and distribute concise and readable code inside the community thus encouraging others to try their own modifications. Time performance of the algorithm is not the subject of the research in the current chapter, and should be investigated separately, specifically sweep compactor KLL and algorithms for weighted quantiles, which theoretically improve the worst-case update time exponentially in $1/\varepsilon$. All algorithms in the current comparison are randomized, thus for each experiment the results presented are averaged over 50 independent runs. KLL and all suggested modifications are compared with each other and LWYC (algorithm Random from [77]). In [143] the authors carried the experimental study algorithms from [104, 105, 2, 64] and concluded that their own algorithm LWYC with the space complexity of $O(1/\varepsilon \log^{3/2} 1/\varepsilon)$ is

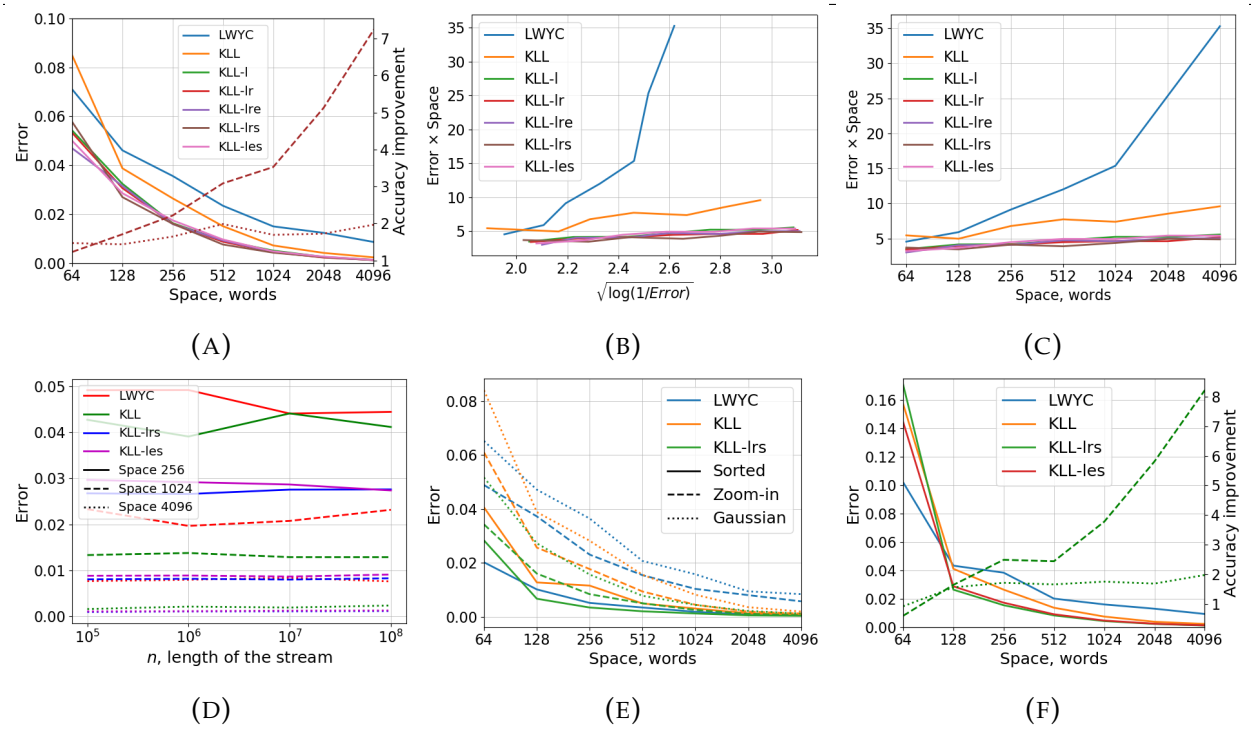


FIGURE 4.8: Figures 4.8a, 4.8b, 4.8c, 4.8e, 4.8f depict the trade-off between maximum error over all queried quantiles and space allocated to the sketch: figures 4.8a, 4.8c, 4.8b shows the results on the randomly ordered streams but in different axes, figure 4.8e shows the results for the sorted stream, stream ordered according to zoom-in pattern, and stream with Gaussian distribution, 4.8f shows the approximation ratio for CAIDA dataset. Figure 4.8d shows the trade-off between error and the length of the stream.

preferable to the others: better in accuracy than [64] and similar in accuracy compared with [105] while LWYC has a simpler logic, i.e. easier to implement. As mentioned earlier we compared our algorithms under a fixed space restrictions. In other words, in all experiments we fixed the space allocated to the sketch and evaluate the algorithm based on the best accuracy it can achieve under that space limit. We measured the accuracy as the maximum deviation among all quantile queries, otherwise known as the Kolmogorov-Smirnov divergence, widely used to measure the distance between CDFs of two distributions. Additionally, we measure the introduced variance caused separately

by the compaction steps and sampling. Its value can help the user to evaluate the accuracy of the output. Note that for KLL this value depends on the size of the stream, and is independent of the arrival order of the items. In other words, the guarantees of KLL are the same for all types of streams, adversarial and structured. Some of our improvements change this property; recall that the sweep compactor KLL, when applied to sorted input, requires only a single sweep, i.e. compaction, per layer. For this reason, in our experiments we found variance to be dependent not only on the internal randomness of the algorithm but also the arrival order of the stream items.

Results On the Figure 4.8a you can see the actual error for LWYC, KLL and its modifications. Note that majority modifications presented in the current chapter can be combined for better performance, due to the space limits we present only the following: KLL-l, KLL-lr, KLL-lre, KLL-lrs and KLL-les. For compactness, from here we use the following encoding: "l" for lazy, "r" for reduced randomness, "e" for spread error, and "s" for sweep compactor. Solid lines on the Figure 4.8a depicts actual error vs. space allocated to the sketch, dashed line depicts the improvement of KLL-lrs over the LWYC and dotted line — over the KLL. Improvement is measured as error(LWYC)/error(KLL-lrs) and error(KLL)/error(KLL-lrs) correspondingly.

First, we can see that all KLL-based algorithms provide the approximation ratio significantly better than LWYC as the space allocation is growing, which confirms theoretical guarantees, i.e. dependency of the space complexity on ε in both algorithms. In more details it is presented on the Figure 4.8b where the same results are plotted in axis (error $\times$ space vs. $\log^{1/2} 1/\text{error}$). Although the axis are not very useful for practical tuning and optimizing, they depict that theoretical upper bounds are quite tight: all KLL-based algorithms can be approximated with linear functions, while LWYC clearly shows the

polynomial growth. Figure 4.8b also makes it obvious that vanilla KLL has different constant compared to its modifications. Going back to the Figure 4.8a we can see that improvement of modified KLL over the vanilla version is approaching the factor of two, which agrees with the theoretical argument, that the variance of the total error introduced in KLL-lre is four times smaller than for KLL. Note that smaller allocated space induce higher portion of the final error that comes from sampling, thus larger sketch size would demonstrate larger improvement factor. Figure 4.8a shows that KLL-lre has up-to 7 times better performance than *LWYC* and the factor is growing with the space allocated to the sketch.

Similar experiments were carried on all synthetic and real datasets, on the Figure 4.8e we plotted results for sorted order, zoom-in order, and randomly shuffled items from Gaussian distribution. Note that most compactor-based algorithms show its best performance on the sorted order datasets. Figure 4.8e shows that on the sorted stream vanilla KLL performs worse compared to LWYC. However, the sweep-compactor modification helps to beat LWYC results even on the sorted data, i.e. sweep compactor based KLL-lrs is better than LWYC on all tested orders. Same figure shows significant overhead of proposed algorithms and modifications compared to LWYC on zoom-in ordered and Gaussian streams. Similar positive results were achieved on the real datasets, on the Figure 4.8f you can see that error introduced by KLL-lrs is twice smaller compared to KLL and 8 times smaller compared to LWYC. Although, theoretically none of the algorithms should depend on the length of the dataset, we verified this property in practice, the results can be seen on the Figure 4.8d.

4.5 Conclusion

We verified experimentally that the algorithm KLL proposed by Karnin *et al*. [85] has predicted asymptotic improvement over LWYC[143], previously showing the best results among the compactor based algorithms. We proposed three modifications to KLL with provably better constants in the approximation bounds. Experiments verified that improvement in approximation which is almost twice better compared to KLL and more than 8 times better compared to LWYC (and growing with the space allocated to the sketch). Moreover, worst-case update time for presented sweep-compactor based KLL is $O(\log 1/\varepsilon)$ which improves over the rest compactor based algorithms with $O(1/\varepsilon)$. Two algorithms proposed for the weighted streams improve over the naive extension from $O((\max w_i) \log 1/\varepsilon)$ to $O(\log 1/\varepsilon)$, while working with the same space complexity.

Chapter 5

Finding haloes in cosmological N-body simulations

5.1 Introduction

The main purpose of astrophysics as a field is to describe the universe and explain all of its observed properties. Cosmology in particular works with distribution of matter in the universe on the large scale. Specifics of the field made the intense computer simulations to be the only way to mimic the controlled experiments and therefore verify certain hypothesis on the evolution of matter distribution. In other words, simulations help to understand how matter organizes itself into galaxies, clusters of galaxies and large-scale structures (e.g. [133]). Therefore, a large amount of effort is spent on running simulations modeling representative parts of the universe in ever greater detail. On the high level, simulation operates with a set of particles and at each step iteratively computes first the gravitational force field caused by their interaction and then corresponding change in the position and velocity of each particle. The nature of the simulations made it computationally and memory intensive, pushing the requirements to the hardware beyond the

capabilities of many researchers, just hosting a single snapshot for the state-of-the art simulations [7, 124] requires tens of Terabytes of memory. Moreover, even just storing such a large snapshot is not only expensive but also challenging. The crucial step in the analysis of the simulations is to locate mass concentration called "haloes" [92], where galaxies would be expected to form. This step is important to connect theory and observations as galaxies are the most observable objects that trace the large-scale structure, but their precise spatial distribution is only established through these simulations. Finding haloes in the output of the simulations allows astronomers to compute important statistics, e.g., mass functions [87].

Although from an astronomical perspective the concept of a "halo" is fairly well understood, the mathematical definition of halos in a simulation varies among simulation and analysis methods. For instance, [122] defines it as mass blobs around the density peaks above some thresholds; [47] defines it as the connected components of the distances graph on the particles. A definition that does not use the density, instead uses particle crossings [54]. Many algorithms have been developed to find haloes in simulations. The algorithms vary widely, even conceptually. The lack of agreement upon a single definition of a halo makes it difficult to uniquely compare the results of different halo-finding algorithms. Nevertheless, [91] evaluated 17 different algorithms and compared them using various criteria, and found broad agreement between them, but with many differences in detail for ambiguous cases. Friends-of-Friends algorithm (FoF) [47] is often considered to be a standard approach, as it was among the first used, and is simple conceptually. The drawbacks of FoF include that the simple density estimate can artificially link physically separate haloes together, and the arbitrariness of the linking length. A halo-finding comparison project [91] evaluated the results of 17 different halo-finding algorithms; further analysis appeared in [92].

Although there are a large number of algorithms and implementations [93, 61, 122, 90, 134, 144, 114, 47], all of them are generally computationally intensive, often requiring all particle positions and velocities to be loaded in memory simultaneously. In fact, most are executed during the execution of the simulation itself, requiring comparable computational resources. However, in order to understand the systematic errors in such algorithms, it is often necessary to run multiple halo-finders, often well after the original simulation has been run. Also, many of the newest simulations have several hundred billion to over a trillion particles [7, 124], with a very large memory footprint, making post-processing analysis unfeasible unless using supercomputers of the same size that created the simulations in the first place. In the current chapter we investigate a way to apply streaming algorithms as halo finders, and compare the results to those of other algorithms participating in the Halo-Finding Comparison Project. Given that state-of-the-art cosmological simulations operate with over a trillion particles, compared to offline algorithms that require the input to be entirely in memory, streaming model provide a way to process the data by making just one or a small number of passes over it, using only megabytes of memory instead of terabytes.

Section 5.2 is based on work done in collaboration with Liu Z., Yang L., Neyrinck M., Lemson G., Szalay A., Braverman V., Budavari T., Burns R., and Wang X. [102]. In that work we investigate a novel connection between the problem of finding the most massive halos in cosmological N-Body simulations and the problem of finding heavy hitters in data streams. We have built a halo finder based on the implementation of the Count-Sketch algorithm [39] and Pick-and-Drop sampling [26]. The halo finder successfully locates most ($> 90\%$) of the top $k \approx 1000$ largest haloes in a dark matter simulation and can be run on the machines with relatively modest computing resources.

However, in [102], all experiments were running on relatively small data streams with

at most 10^9 items. One of the reasons for that was the rather poor *time* performance of the underlying algorithms, which would cause every experiment to take more than week to run. In section 5.3, which is based on [81], work done in collaboration with Liu Z., Yang L., Kumar S., Neyrinck M., Lemson G., Szalay A., Braverman V., Budavari T., we improve the implementation and push the number of halos' centers to be found to $\sim 10^4 - 10^5$. Our tool needs less than 5 minutes to find the top $3 \cdot 10^5$ heavy cells on a dataset with 10^{10} particles. Compared to previous results [102], which required more than 8 hours, it's more than a $100\times$ improvement. This dataset consists of a snapshot of the Millennium dataset [133] and we use a grid of 10^{11} cells in our algorithm for approximation of the density field, which can be used further for astrophysical analysis. We port the entire Count-Sketch infrastructure into the GPU and thus make the tool significantly outperform the previous approach. In our analysis, we carefully investigate the trade-off between memory and the quality of the result.

In [102] authors reduced the halo-finding problem to the problem of finding the top-k densest cells in a regular mesh. This reduction shows that these densest cells are closely related to space with the heaviest halos. In [81] we consider another possible application, that of determining statistics on "excursion sets". Kaiser [83] investigated the clustering properties of the regions with a density higher than the average in Gaussian random fields. He showed that such regions cluster more strongly than those with lower overdensities and the strength of this effect increases with the density threshold. He used this as an explanation of the observed stronger clustering of galaxy clusters compared to the clustering of the galaxy distribution itself. Bardeen et al. [13], refined this argument, focusing on the peaks of the density fields—the locations where galaxies and clusters are expected to form. This biased clustering phenomenon can be examined in an evolved

density field by filtering regions in the dark matter distribution field, based on their density. This is equivalent to examining the "heavy hitters" in the counts-in-cells. We expect the randomized algorithms to not be exact, and it is interesting to investigate how this affects the clustering measure.

5.2 Streaming Algorithms for Halo Finders

This section is based on [102].

5.2.1 Methodology

Streaming and heavy hitters. As it was defined in Chapter 3.2, a data stream $D = D(n, m)$ is an ordered sequence of objects $p_1, p_2, \ldots, p_n$, where $p_j = 1 \ldots m$. The elements of the stream can represent any digital object: integers, real numbers of fixed precision, edges of a large graph, messages, images, web pages, etc. In the practical applications, both n and m may be very large, and we are interested in the algorithms with $o(n + m)$ space. A streaming algorithm is an algorithm that can make a single pass over the input stream. The above constraints imply that a streaming algorithm is often a randomized algorithm that provides approximate answers with high probability. In practice, these approximate answers often suffice.

We investigate the results of cosmological simulations where the number of particles will soon reach 10^{12}. Compared to offline algorithms that require the input to be entirely in memory, streaming algorithms provide a way to process the data using only megabytes memory instead of gigabytes or terabytes in practice.

Further we investigate the connection between problems of finding haloes and heavy hitters, thus for the sake of completeness, we review the heavy hitter problem and heavy hitter algorithms first introduced in 3.2.

For each element i, its frequency f_i is the number of its occurrences in D. The k^{th} frequency moment of a data stream D is defined as $F_k(D) = \sum_{i=1}^{m} f_i^k$. We say that an element is "heavy" if it appears more times than a constant fraction of some L_p norm of the stream, where $L_p = (\sum_i f_i^p)^{1/p}$ for $p > 1$. In this section, we consider the following heavy hitter problem.

Problem 1. *Given a stream D of n elements, the ε-approximate (ϕ, L_p)-heavy hitter problem is to find a set of elements T:*

- $\forall i \in [m], f_i > \phi L_p \implies i \in T.$
- $\forall i \in [m], f_i < (\phi - \varepsilon) L_p \implies i \notin T.$

We allow the heavy hitter algorithms to use randomness; the requirement is that the correct answer should be returned with high probability. The heavy hitter problem is equivalent to the problem of approximately finding the k most frequent elements. Indeed, the top k most frequent elements are in the set of (ϕ, L_1)-heavy hitters in the stream, where $\phi = \Theta(1/k)$. There is a $\Omega(1/\varepsilon^2)$ trade-off between the approximation error ε and the memory usage. Heavy hitter algorithms are building blocks of many data stream algorithms ([30, 80]).

We treat the cosmological simulation data from [91] as a data stream. To do so, we apply an online transformation that we describe in the next section.

Data Transformation. In a cosmological simulation, dark matter particles form structures through gravitational clustering in a large box with periodic boundary conditions

representing a patch of the simulated universe. The box we use [91] is of size 500 Mpc/h, or about 2 billion light-years. The simulation data consists of positions and velocities of 256^3, 512^3 or 1024^3 particles, each representing a huge number of physical dark-matter particles. They are distributed rather uniformly on large scales ($\gtrsim$ 50 Mpc/h) in the simulation box, clumping together on smaller scales. A halo is a clump of particles that are gravitationally bound.

To apply the streaming algorithms, we transform the data. We discretize the spatial coordinates so that we will have a finite number of types in our transformed data stream. We partition the simulation box into a grid of cubic cells, and bin the particles into them. The cell size is chosen to be 1 Mpc/h as to match a typical size of a large halo; there are thus 500^3 cells. This parameter can be modified in practical applications, but it relates to the space and time efficiency of the algorithm. We summarize the data transformation steps as follows.

- Partition the simulation box into grids of cubic cells. Assign each cell a unique integer ID.
- After reading a particle, determine its cell. Insert that cell ID into the data stream.

Using the above transformation, streaming algorithms can process the particles in the same way as an integer data stream.

Heavy Hitters and Dense Cells. For a heavy-hitter algorithm to save memory and time, the distribution of cell counts must be very non-uniform. The simulations begin with an almost uniform lattice of particles, but after gravity clusters them together, the density distribution in cells can be modeled by a lognormal PDF ([41], [89]):

$$P_{LN}^{(1)}(\delta) = \frac{1}{(2\pi\sigma_1^2)^{1/2}} \exp\left\{-\frac{[\ln(1+\delta)+\sigma_1^2/2]^2}{2\sigma_1^2}\right\}\frac{1}{1+\delta}, \tag{5.1}$$

where $\delta = \rho/\bar{\rho} - 1$ is the overdensity, $\sigma_1^2(R) = \ln[1+\sigma_{\rm nl}^2(R)]$, and $\sigma_{\rm nl}^2(R)$ is the variance of the nonlinear density field in spheres of radius R. Our cells are cubic, not spherical; for theoretical estimates, we use a spherical top-hat of the same volume as a cell.

Let N be the number of cells, and P_c be the distribution of the number of particles per cell. The L_p heaviness ϕ_p can be estimated as

$$\phi_p \approx \frac{P_{200}}{(N\langle {P_c}^p\rangle)^{1/p}}, \tag{5.2}$$

where P_{200} is the number of particles in a cell with density exactly $200\bar{\rho}$. This density threshold is a typical minimum density of a halo, coming from the spherical-collapse model. We theoretically estimated $\sigma_{\rm nl}$ for the cells in our density field by integrating the nonlinear power spectrum (using the fit of [132], and the cosmological parameters of the simulation) with a spherical tophat window. The grid size in our algorithm is roughly 1.0 Mpc (500^3 cells in total), giving $\sigma_{\rm nl}(\text{Cell}) \approx 10.75$. We estimated $\phi_1 \approx 10^{-6}$ and $\phi_2 \approx 10^{-3}$, matching order-of-magnitude with the measurement of the actual density variance from the simulation cells. These heaviness values are low enough to presume that a heavy-hitter algorithm will efficiently find cells corresponding to haloes.

Streaming Algorithms for Heavy Hitter Problem. The above relation between the halo-finding problem and the heavy hitter problem encourages us to apply efficient streaming algorithms to build a new halo finder. Our halo finder takes a stream of particles, performs the data transformation described above and then applies a heavy hitter algorithm

to output the approximate top k heavy hitters in the transformed stream. These heavy hitters correspond to the densest cells in the simulation data as described earlier. In our first version of the halo finder, we use Count-Sketch algorithm [39] and Pick-and-Drop Sampling [26].

The Count-Sketch Algorithm. For a more generalized description of the algorithm, please refer to [39]. For completeness, we summarize the algorithm as follows. The Count-Sketch algorithm uses a compact data structure to maintain the approximate counts of the top k most frequent elements in a stream. This data structure is an $r \times t$ matrix M representing estimated counts for all elements. These counts are calculated by two sets of hash functions: let $h_1, h_2, \ldots, h_r$ be r hash functions, mapping the input items to $\{1, \ldots, t\}$, where each h_i is sampled uniformly from the hash function set H. Let $s_1, s_2, \ldots, s_r$ be hash functions, mapping the input items to $\{+1, -1\}$, uniformly sampled from another hash function set S. We can interpret this matrix as an array of r hash tables, each containing t buckets.

There are two operations on the Count-Sketch data structure. Denote $M_{i,j}$ as the j^{th} bucket in the i^{th} hash table:

- $Add(M, p)$: For $i \in [1, r]$, $M_{i,h_i[p]} += s_i[p]$.
- $Estimate(M, p)$, return $median_i\{h_i[p] \cdot s_i[p]\}$

The *Add* operation updates the approximate frequency for each incoming element and the *Estimate* operation outputs the current approximate frequency. To maintain and store the estimated k most frequent elements, CountSketch also needs a priority queue data structure. The pseudocode of Count-Sketch algorithm is presented in Figure 5.1. More details and theoretical guarantees are presented in [39].

```
1: procedure CountSketch(r, t, k, D)                    ▷ D is a stream
2:     Initialize an empty r × t matrix M.
3:     Initialize an min-priority queue Q of size k
4:     (particle with smallest count is on the top).
5:     for i = 1, ..., n and p_i ∈ D do
6:         Add(M, p_i);
7:         if p_i ∈ Q then
8:             P_i.count++;
9:         else if Estimate(M, p_i) > Q.top().count then
10:            Q.pop();
11:            Q.push(p_i);
12:        end if
13:    end for
14:    return Q
15: end procedure
```

FIGURE 5.1: Count-Sketch Algorithm

The Pick-and-Drop Sampling Algorithm. Pick-and-Drop Sampling is a sampling-based streaming algorithm to approximate the heavy hitters. To describe the idea of Pick-and-Drop sampling, we view the data stream as a sequence of r blocks of size t. Define $d_{i,j}$ as the j^{th} element in the i^{th} block and $d_{i,j} = p_{k(i-1)+j}$ in stream D. In each block of the stream, Pick-and-Drop sampling will pick one random sample and record its remaining frequency in the block. The algorithm maintains a sample with the largest current counter and drops previous samples. The pseudocode of Pick-and-Drop sampling [26] is given in Figure 5.2 and we need the following definitions in Figure 5.2. For $i \in [r], j, s \in [t], q \in [m]$ define:

$$f_{i,q} = |\{j \in [t] : d_{i,j} = q\}|, \tag{5.3}$$

$$a_{i,s} = |\{j^* : s \leq j^* \leq t, d_{i,j^*} = d_{i,s}\}|. \tag{5.4}$$

The detail implementation is in Section 5.2.2.

procedure PICKDROP(r, t, λ, D)
  Sample S_1 uniformly at random on $[t]$.
  $L_1 \leftarrow d_{1,S_1}$,
  $C_1 \leftarrow a_{1,S_1}$,
  $u_1 \leftarrow 1$.
  for $i = 2, \ldots, r$ **do**
    Sample S_i uniformly at random on $[t]$.
    $l_i \leftarrow d_{i,S_i}$, $c_i \leftarrow a_{i,S_i}$
    if $C_{i-1} < \max(c_i, \lambda u_{i-1})$ **then**
      $L_i \leftarrow l_i$,
      $C_i \leftarrow c_i$,
      $u_i \leftarrow 1$
    else
      $L_i \leftarrow L_{i-1}$,
      $C_i \leftarrow C_{i-1} + f_{i,L_{i-1}}$,
      $u_i \leftarrow q_{i-1} + 1$
    end if
  end for
  return $\{L_r, C_r\}$
end procedure

FIGURE 5.2: Pick-and-Drop Algorithm

5.2.2 Implementation

Simulation Data. The N-body simulation data we use as the input to our halo finder was used in the halo-finding comparison project [91] and consists of various resolutions (numbers of particles) of the MareNostrum Universe cosmological simulation [63]. These simulations ran in a 500 h^{-1}Mpc box, assuming a standard ΛCDM (cold dark matter and cosmological constant) cosmological model.

In the first implementation of our halo finder, we consider two halo properties: center position and mass (the number of particles in it). We compare to the the fiducial offline algorithm FoF. The distributions of halo sizes from different halo finders are presented in Fig. 5.3.

Since our halo finder builds upon on the streaming algorithms of finding frequent

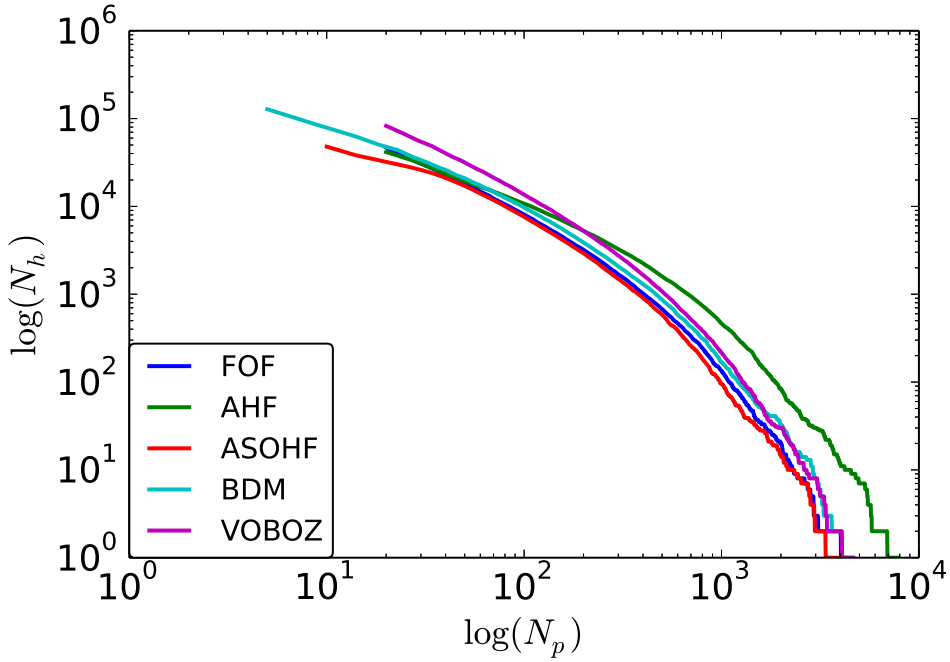


FIGURE 5.3: Halo mass distribution of various halo finders.

items, the algorithms need to transform the data as described in section 5.2.1 — dividing all the particles into different small cells and label each particle with its associated cell ID. For example, if an input dataset contains three particles p_1, p_2, p_3 and they are all included in a cell of ID $= 1$, then the transformed data stream becomes $1, 1, 1$. The most frequent element in the stream is obviously 1 and thus the cell 1 is the heaviest cell overall.

Implementation Details. Our halo finder implementation is written using C++ with GNU GCC compiler 4.9.2. We implemented Count-Sketch and Pick-and-Drop sampling as two algorithms to find heavy hitters.

Count-Sketch-based Halo Finder. There are three basic steps in the Count-Sketch algorithm, which returns the heavy cells and the number of particles associated with them. (1) Allocate memory for the CountSketch data structure to hold current estimates

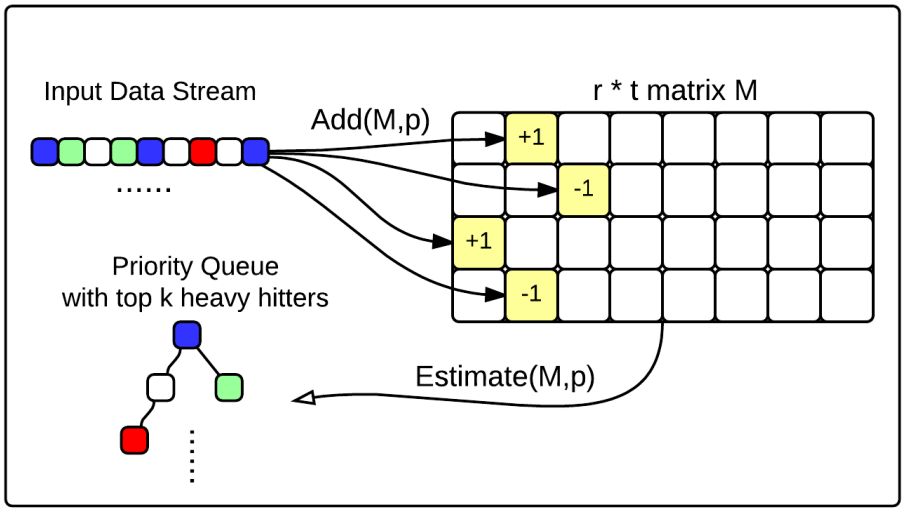


FIGURE 5.4: Count-Sketch Algorithm

of cell frequencies; (2) use a priority queue to record the top k most frequent elements; (3) return the positions of the top k heavy cells.

The Count-Sketch data structure is an $r \times t$ matrix. Following [42], we set $r = \log(\frac{n}{\varepsilon})$ and t to be sufficiently large (>1 million) to achieve an expected approximation error $\varepsilon = 0.05$. We build the matrix as a 2D array with $r \times t$ 0's. For each incoming element in the stream, an *Add* operation has to be executed and an *estimate* operation needs to be executed only when this element is not in the queue.

Pick-and-Drop-based Halo Finder. In the Pick-and-Drop sampling based halo finder, we implement a general hash function H: $\mathbb{N}^+ \rightarrow \{1, 2, \ldots, ck\}$, where $c \geq 1$, to gain the probability of success to approximate the top k heaviest cells. We apply the hash function H on every incoming element and put the elements with the same hash value together such that the original stream is divided into ck smaller sub-streams. Meanwhile, we initialize ck instances of Pick-and-Drop sampling so that each PD instance will process

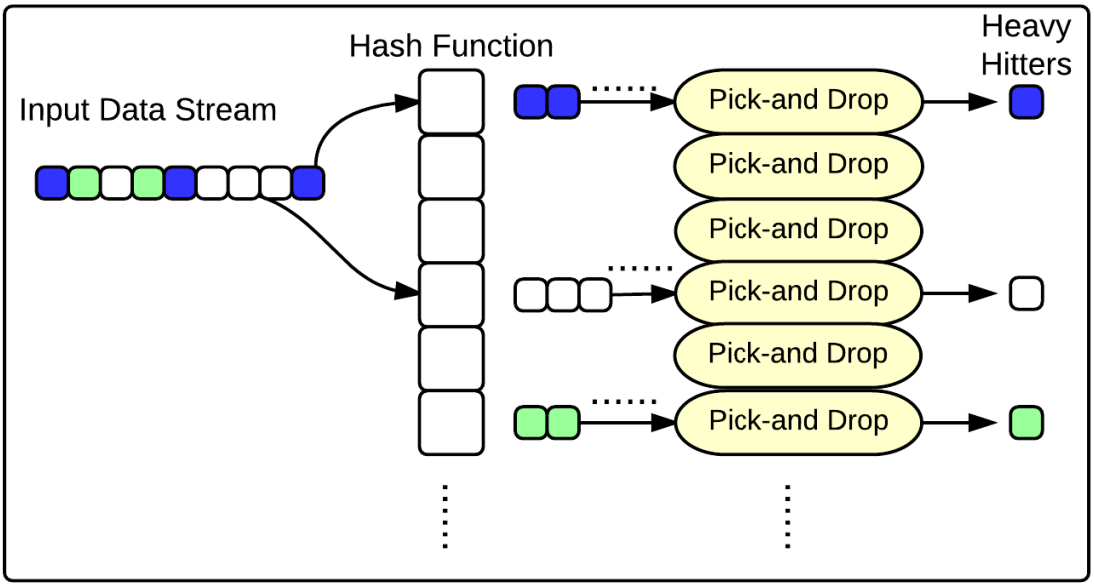


FIGURE 5.5: Pick-and-Drop Sampling

one sub-stream. The whole process of approximating the heavy hitters is presented in Figure 5.5. In this way, the repeated items in the whole stream will be distributed into the same sub-stream and they are much heavier in this sub-stream. With high probability, each instance of Pick-and-Drop sampling will output the heaviest one in each of the sub-streams, and in total we will have ck output items. Because of the randomness in the sampling method, we will expect some of inaccurate heavy hitters among the total ck outputs. By setting a large c, most of the actual top k most frequent elements should be inside the ck outputs (raw data).

To get precise properties of haloes, such as the center, and mass, an offline algorithm such as FoF [47] can be applied to the particles inside the returned heavy cells and their neighbor cells. This needs an additional pass over the data but we only need to store a small amount of particles to run those offline in-memory algorithms. The whole process

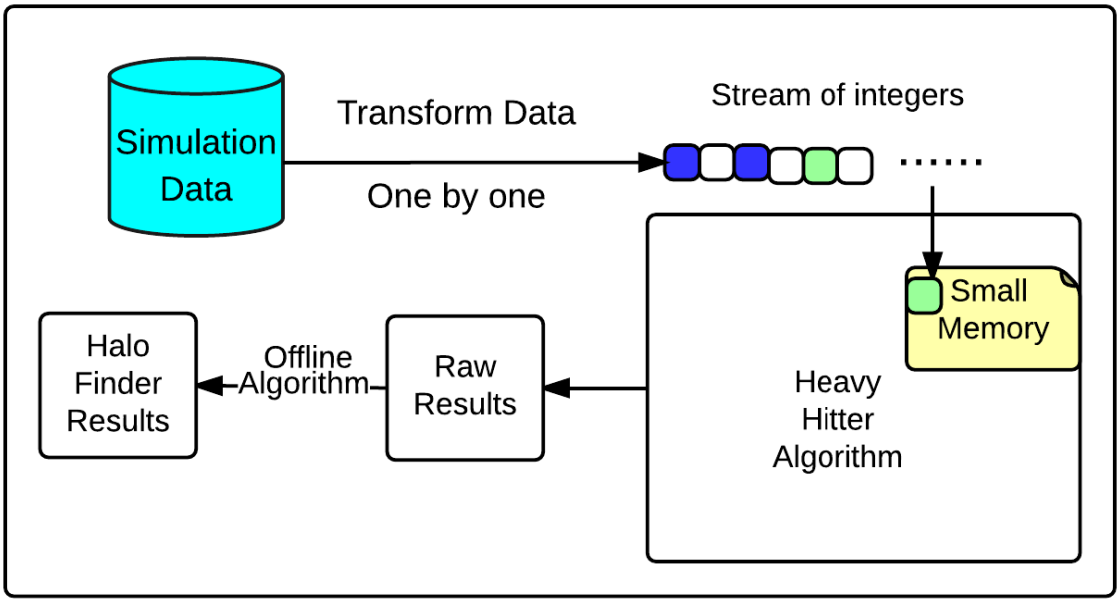


FIGURE 5.6: Halo Finder Procedure

of the halo finder is represented in Figure 5.6, where heavy hitter algorithms can be regarded as a black box. That is, any theoretically efficient heavy hitter algorithms could be applied to further improve the memory usage and practical performance.

Shifting Method. In the first pass of our halo finder, we only use the position of a heavy cell as the position of a halo. However, each heavy cell may contain several haloes and some of the haloes located on the edges between two cells cannot be recognized because the cell size in the data transformation step is fixed. To recover those missing haloes, we utilize a simple shifting method:

- Initialize 2^d instances of Count-Sketch or Pick-and-Drop in parallel, where d is the dimension. Our simulation data reside in three dimensions, so $d = 3$.

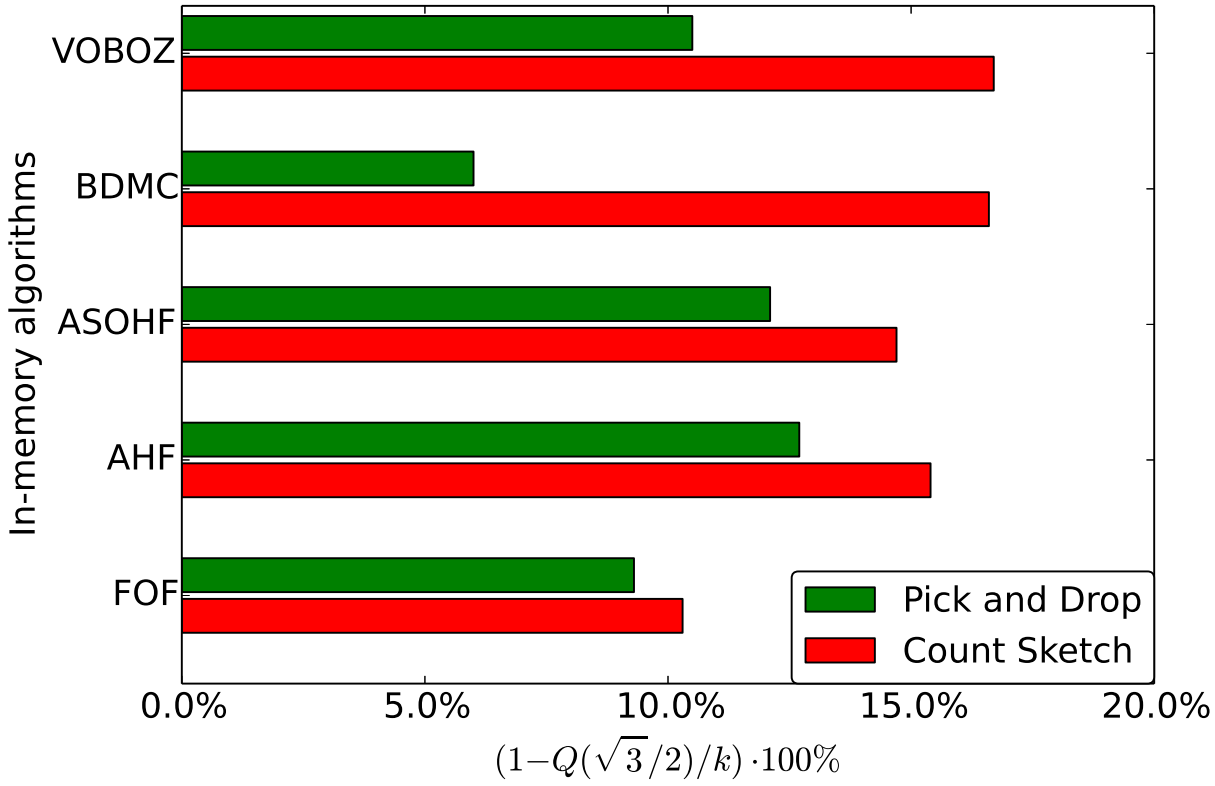


FIGURE 5.7: Measures of the disagreement between PD and CS, and various in-memory algorithms. The percentage shown is the fraction of haloes farther than a half-cell diagonal ($0.5\sqrt{3}$ Mpc$/h$) from PD or CS halo positions.

- Move all the particles to one of the 2^d directions with a distance of 0.5 Mpc$/h$ (half of the cell size). In each of the 2^d shifting processes, assign a Count-Sketch/Pick-and-Drop instance to run. By combining the results from 2^d shifting processes, we expect that the majority of the top k largest haloes are discovered. All the parallel instances of the CountSketch/Pick-and-Drop are enabled by OpenMP 4.0 in C++.

5.2.3 Evaluation

To evaluate how well streaming based halo finders work, we mainly focus on testing it in the following four aspects:

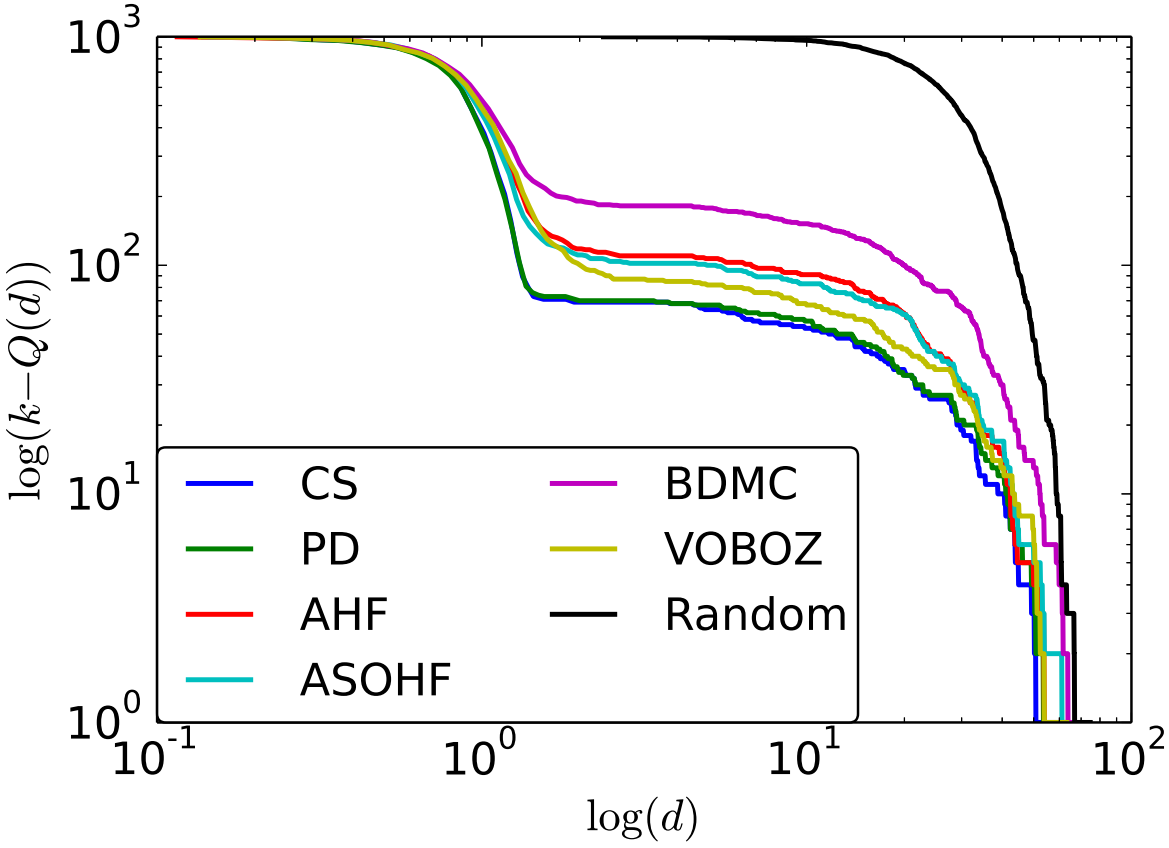


FIGURE 5.8: The number of top-1000 FoF haloes farther than a distance d away from any top-1000 halo from the algorithm of each curve.

- **Correctness:** Evaluate how close are the positions of top k largest haloes found by the streaming-based algorithms to the top k largest haloes returned by some widely used in-memory algorithms. Evaluate the trade-off between selection k and quality of result.

- **Stability:** Since streaming algorithms always require some randomness and may produce some incorrect results, we want to see how stable are streaming based heavy hitter algorithms are.

- **Memory Usage:** Linear memory space requirement is a "bottleneck" for all offline algorithms, and it is the central problem that we are trying to overcome by applying streaming approach. Thus it is significantly important to theoretically or experimentally estimate the memory usage of Pick-and-Drop and Count-Sketch algorithms.

In the evaluation, all the in-memory algorithms we choose to compare were proposed in the Halo-Finding Comparison Project [91]. We test against the fiducial FOF method, as well as four others that find density peak:

1. **FOF** by Davis et al.[47]
   "Plain-vanilla" Friends-of-Friends.

2. **AHF** by Knollmann & Knebe [93]
   Density peaks search with recursively refined grid

3. **ASOHF** by Planelles & Quilis. [122]
   Finds spherical-overdensity peaks using adaptive density refinement.

4. **BDM** [90], run by Klypin & Ceverino "Bound Density Maxima" – finds gravitationally-bound spherical-overdensity peaks.

5. **VOBOZ** by Neyrinck et al [114]

   "Voronoi BOund Zones" – finds gravitationally bound peaks using a Voronoi tessellation.

Correctness. As there is no agreed-upon rule how to define the center and the boundary of a halo, it is impossible to theoretically define and deterministically verify the correctness of any halo finder. Therefore a comparison to the results of previous widely accepted halo finders seems to be the best practical verification of a new halo finder algorithm. To compare the outputs of two different halo finders we need to introduce some formal measure of similarity. The most straightforward way to compare them is to consider one of them H as a ground truth, and another one E as an estimator. Among this the FOF algorithm is considered to be the oldest and the most widely used, thus in our initial evaluation we decided to concentrate on the comparison with FOF. Then the most natural measure of similarity is number of elements in H that match to elements in output of E. More formally we will define "matches" as: for a given θ we will say that center $e_i \in E$ matches the element $h_i \in E$ if $dist(e_i, h_i) \leq \theta$, where $dist(\cdot, \cdot)$ is Euclidean distance. Then our measure of similarity is:

$$Q(\theta) = Q(E_k, H_k, \theta) = |\{h_i \in H_k : \min_{e_j \in E_k} dist(h_i, e_j) < \theta\}|,$$

where k represents the top k heaviest halos.

We compare the output of both streaming-based halo finders to the output of in-memory halo finders. We made comparisons for the $256^3, 512^3$ and 1024^3-particle simulations, finding the top 1000 and top 10000 heaviest hitters. Since the comparison results in all cases were similar, the figures presented below are for the smallest dataset, and $k = 1000$.

On the Figure 5.7 we show for each in-memory algorithm the percentage of centers that were not found by streaming-based halo finder. We can see that both the Count-sketch and Pick-and-drop algorithms missed no much more than 10 percent of the haloes in any of the results from the in-memory algorithms.

To understand if 10 percent means that two halo catalogs are close to each other or not, we will choose one of the in-memory algorithms as a ground truth and compare how close are the other in-memory algorithms. Again, we choose FOF algorithm as a ground truth. The comparison is depicted in Fig. 5.8. From this graph you can see that outputs of Count-sketch and Pick-and-drop based halo finders are closer to the FOF haloes, then other in-memory algorithms. It can be easily explained, as after finding heavy cells we apply the same FOF to the heavy cells and their neighborhoods, thus output should always have similar structure to the output of in-memory FOF on the full dataset. Also from this graph you can see that each line can be represented as a mix of two component, one of which is the component of random distribution. Basically it means that after distance of $\sqrt{3}/2$ all matches are the same if we just put bunch of points at random.

The classifier is using a top-k to select the halo candidates. Figure 5.9 shows how sensitive the results are to the selection threshold of $k = 1000$. It shows several curves, including the total number of heavy hitters, the ones close to an FoF group – we can call these true positive (TP) – and the ones detected, but not near an FoF object (false positives FP). From the figure, it is clear that the threshold of 1000 is close to the optimal detection threshold, preserving TP and minimizing FP. This corresponds to a true positive detection rate (TPR) of 96% and a false positive detection rate of 3.6%. If we lowered our threshold to $k = 900$, our TPR drops to 91% but the FPR becomes even lower, 0.88%.

These tradeoffs can be shown on a so-called ROC-curve (receiver operating characteristic), where the TPR is plotted against the FPR. This shows how lowering the detection

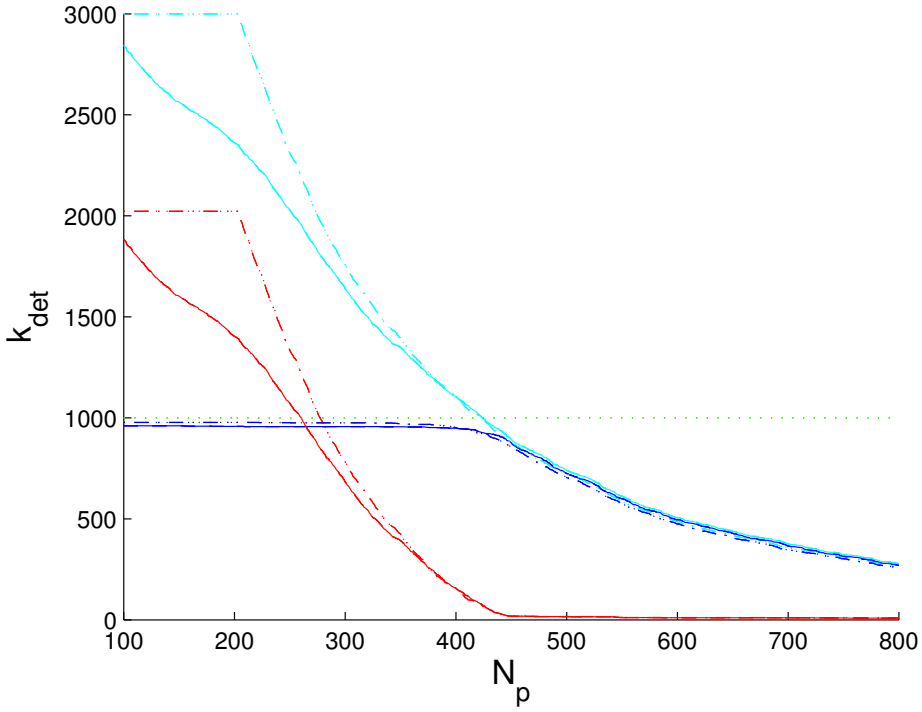


FIGURE 5.9: Number of detected halos by our two algorithms. The solid lines correspond to (CS) and the dashed lines to (PD). The dotted line at $k = 1000$ shows our selection criteria. The x axis is the threshold in the number of particles allocated to the heavy hitter. The cyan color denotes the total number of detections, the blue curves are the true positives (TP), and the red curves are the false positives (FP).

threshold increases the true detections, but the false detection rate increases much faster. Using the ROC curve, shown below we can see the position of the $k = 1000$ threshold as a circle and the $k = 900$ as a square.

Finally, we should also ask, besides the set comparison, how do the individual particle cardinalities counted around the heavy hitters correspond to the FoF ones. Our particle counting is restricted to neighbouring cells, while the FoF is not, so we will always be undercounting. To be less sensitive to such biases, we compare the rank ordering of the two particle counts in the two samples. The rank 1 is assigned to the most massive objects in each set.

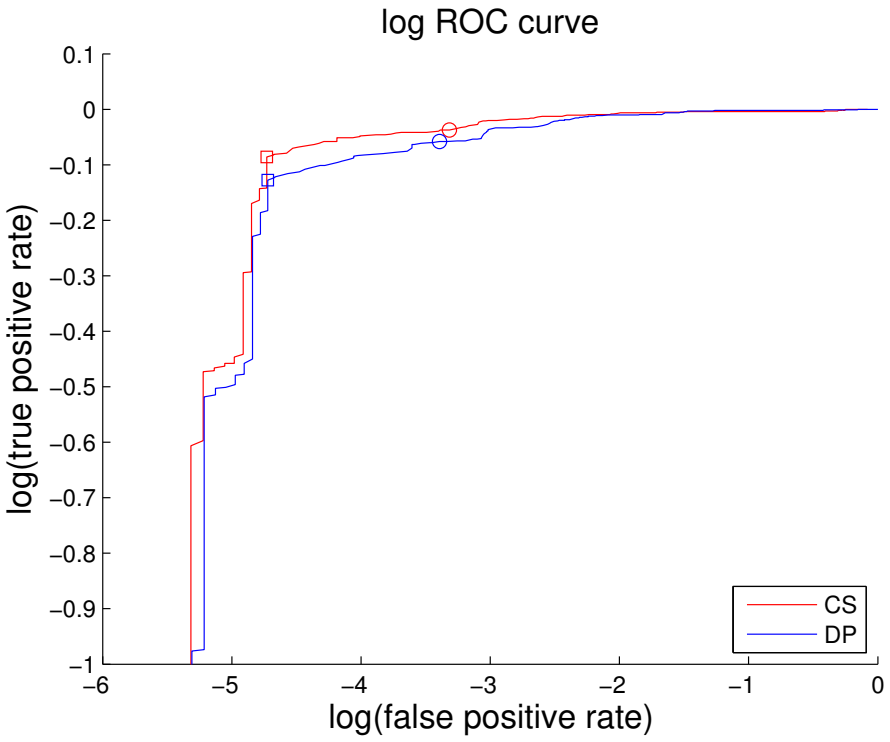


FIGURE 5.10: This ROC curve shows the tradeoff between true and false detections as a function of threshold. The figure plots TPR vs FPR on a log-log scale. The two thresholds are shown with symbols, the circle denotes 1000, and the square is 900.

Stability. As most of the streaming algorithms utilize randomness, we estimate how stable our results are compared to the results from a deterministic search. In the deterministic search algorithm, we find the actual heavy cells by counting the number of particles inside each cell; we perform the comparison for the smallest dataset containing 256^3 particles. To perform this evaluation we run 50 instances of each algorithm (denoting the outputs as $\{C^i_{cs}\}_{i=1}^{50}$ and $\{C^i_{pd}\}_{i=1}^{50}$). We also count the number of cells of each result that match the densest cells returned by the deterministic search algorithm C_{ds}. The normalized number of matches will be $\rho^i_{pd} = \frac{|C^i_{pd} \cap C_{ds}|}{|C_{ds}|}$ and $\rho^i_{cs} = \frac{|C^i_{cs} \cap C_{ds}|}{|C_{ds}|}$ correspondingly. Our experiment showed:

$$\mu(\rho^i_{cs}) = 0.946,\ \sigma(\rho^i_{cs}) = 2.7 \cdot 10^{-7}$$

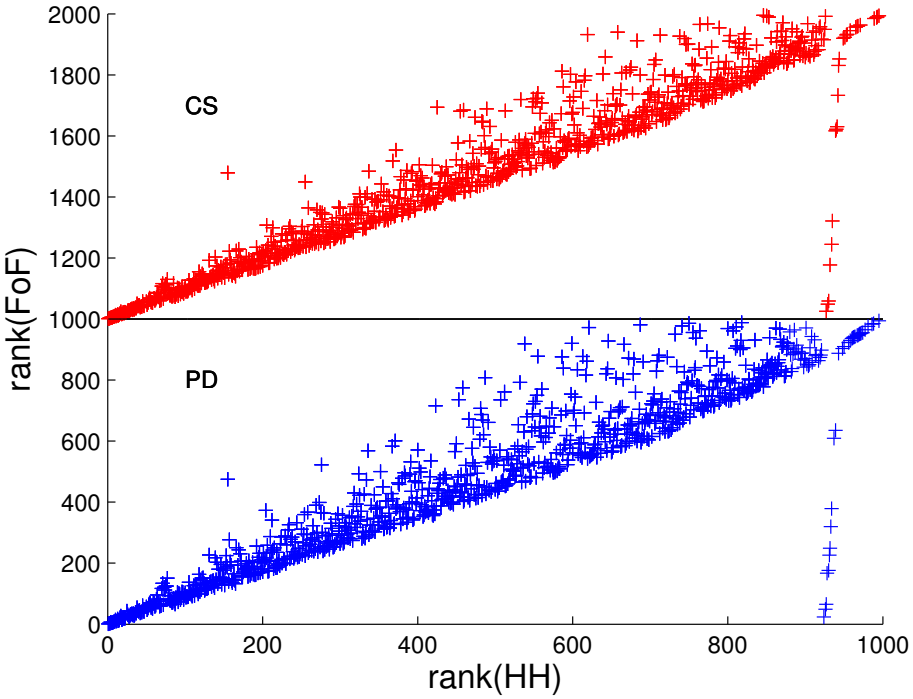


FIGURE 5.11: The top 1000 heavy hitters are rank-ordered by the number of their particles. We also computed a rank of the corresponding FoF halo. The linked pairs of ranks are plotted. One can see that if we adopted a cut at $k = 900$, it would eliminate a lot of the false positives.

$$\mu(\rho^i_{pd}) = 0.995,\ \sigma(\rho^i_{pd}) = 6 \cdot 10^{-7}$$

That means that the approximation error caused by randomness is very small compared with error caused by transition from overdense cells to haloes centers. This fact can also be easily caught from the Fig. 5.12. On that figure, you can see that shaded area below and above red and green lines, which represents the range of outputs among 50 instances, is very thin. Thus the output is very stable.

Memory Usage Low memory usage is one of the most significant advantages of streaming approaches comparing with current halo finding solutions. To the best of our knowledge even for the problem of locating top 1000 largest haloes in the simulation data with

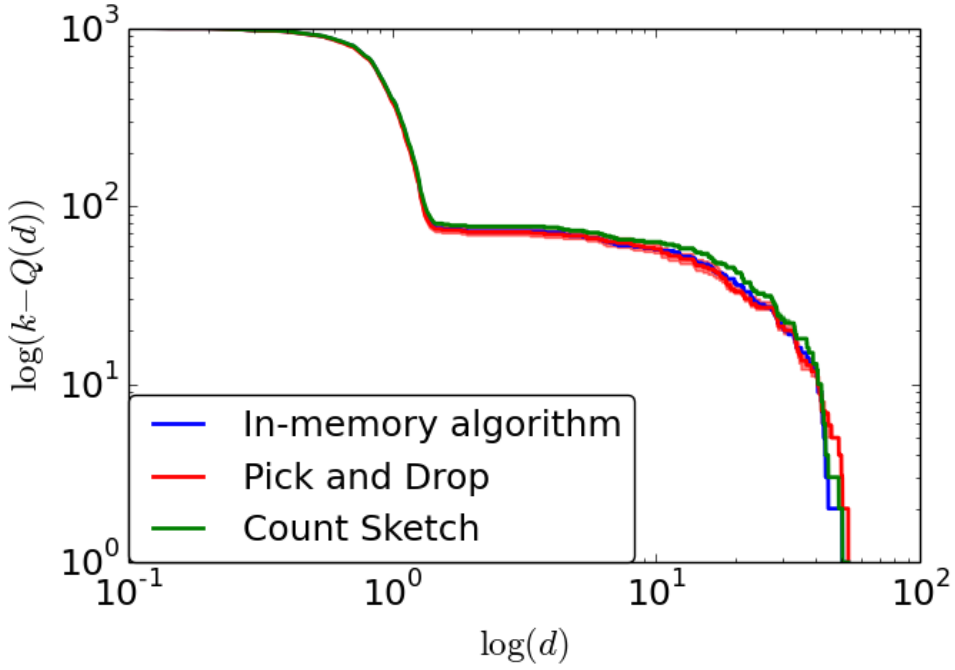


FIGURE 5.12: Each line on the graph represents the top 1000 halo centers found with Pick-and-Drop sampling, Count-Sketch, and in-memory algorithms, as described in section 5.2.2. The comparison with FOF is shown in Fig.5.8. The shaded area (too small to be visible) shows the variation due to randomness.

1024^3 particles, there is no way to run other halo finder algorithms on a regular PC since 1024^3 particles already needs $\approx$ 12GB memory to only store all the particle coordinates; a computing cluster or even supercomputer is necessary. So, the application of streaming techniques introduces a new direction on the development of halo-finding algorithms.

To find top k heavy cells Count-sketch theoretically requires following amount of space:

$$O(k \log \frac{n}{\delta} + \frac{\sum_{q'=k+1}^{m} f_{q'}^2}{(\varepsilon f_k)^2} log \frac{n}{\delta}),$$

where $1-\delta$ is probability of success, and ε is an Q_k estimation error, where Q_k is the frequency of k-th heaviest cell. It is worth mentioning that in application to the heavy

cells search problem the second term is the dominating one. The first factor in the second term represents the linear dependency of memory usage on the heaviness of the top k cells. Thus we can expect linear memory usage for small data-set. But as a dataset grows dependence becomes logarithmic if we assume the same level of heaviness. Experiments verify this observation, as for small data set with 256^3 particles algorithm used around 900 megabytes, while for the large dataset with 1024^3 particles, the amount of space increased up to almost 1000 megabytes, which is insignificant compared with dataset size increase. Thus if we can assume almost constant heaviness of the top k cells, then memory grows logarithmically with dataset size, that is why such an approach is scalable for even larger datasets.

Experimentally for this particular dataset Pick-and-drop algorithm showed much better performance in terms of memory usage than Count-sketch. The actual usage of memory was around 20 megabytes for a small dataset with 256^3 particles and around 30 megabytes for the large dataset with 1024^3 particles.

5.3 Scalable Streaming Tools for Analyzing N-body Simulations

This section is based on [81].

5.3.1 Methodology

In this section, we introduce our methods for efficiently analyzing cosmological datasets. First, we recap the concept of streaming and explain how the problem of estimating density statistics can be approached from the perspective of finding frequent items in the

stream. Then we go over several crucial details of the heavy hitter algorithm named Count Sketch [39].

In this section, we work on two cosmological N-body simulations with 10^{10} and 3×10^{11} particles, respectively, resulting in several terabytes of data. In this setting, typical approaches for finding halos that require loading data into memory become inapplicable on common computing devices (e.g. laptop, desktop, or small workstations) for post-processing and analysis. In contrast, a streaming approach makes the analysis of such datasets feasible even on a desktop by lowering the memory footprint from several terabytes to less than a gigabyte.

Much of the analysis of cosmological N-body simulations focuses on regions with a high concentration of particles as it was shown in Section 5.2. By putting a regular mesh on the simulation box, we can replace each particle with the ID of the cell it belongs to [102]. Then using streaming algorithms we can find the k most frequent cells, i.e. cells with the largest number of particles (see Figure 5.13). Such statistics are very useful for analyzing a spatial distribution of particles on each iteration of the simulation, as shown in 5.2 and as we will show in the current section. One might think that this approach is too naive and just keeping a counter for each cell would provide the exact solution with probability 1, which is much better than any streaming algorithm can offer. However, under the assumption that particles are not sorted in any way, the naïve solution would increase memory usage to terabytes even for the mesh with only 10^{12} cells in it.

Finding frequent elements is one of the most studied problems in streaming settings, moreover, it is often used as a subroutine in many other algorithms [99, 60, 80, 36, 110]. For the sake of completeness we repeat the problem definition in current section with additional details. The frequency (or count) of the element i is the number of its occurrences in the stream S: $f_i = |\{j | s_j = i\}|$. We will call element i as (α, ℓ_p)-heavy if $f_i > \alpha \ell_p$ where

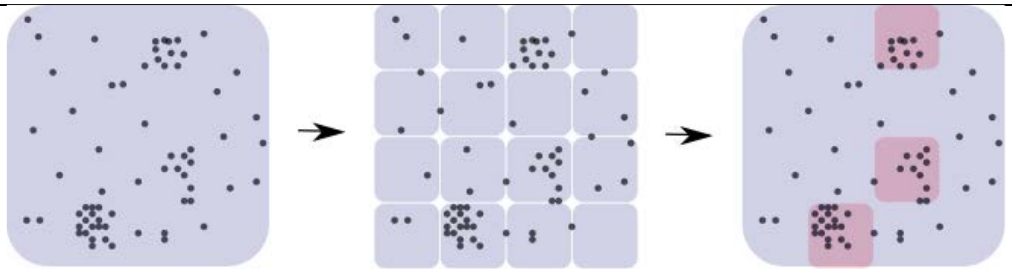


FIGURE 5.13: Finding approximate dense areas with the help of a regular mesh and a streaming solution for finding the top k most frequent items in the stream.

$\ell_p = \left(\sum_j f_j^p\right)^{1/p}$. An approximate scheme for the problem is the following:

Problem 1 (Heavy Hitter)**.** *Given a stream S of m elements the ε-approximate (α, ℓ_p)-heavy hitter problem is to find a set of elements T, such that:*

- $\forall i \in [n], f_i > \alpha \ell_p \rightarrow i \in T$
- $\forall i \in [n], f_i < (\alpha - \varepsilon)\ell_p \rightarrow i \notin T$

Note that ε in the definition above serves as slack for the algorithm to output some items which are not (α, ε)-heavy hitters, but are "ε close" to them. Typically smaller input ε would cause the algorithm to use more memory. Finding the k most frequent items in the stream is the same as finding all (α_k, ℓ_1)-heavy hitters, where α_k is the heaviness of the k-th most frequent item. Note that being ℓ_2-heavy is a weaker requirement than being ℓ_1-heavy: every ℓ_1-heavy item is ℓ_2-heavy, but the other way around it is not always the case. For example, consider the stream where all n items of the dictionary appear only once. To be found in such a stream, a ℓ_1-heavy hitter needs to appear more than εn times for some constant ε, while an ℓ_2-heavy hitter needs to appear just $\varepsilon\sqrt{n}$. Catching an item that appears in the stream significantly less often is more difficult, thus finding all ℓ_2-heavy hitters is more challenging than all ℓ_1.

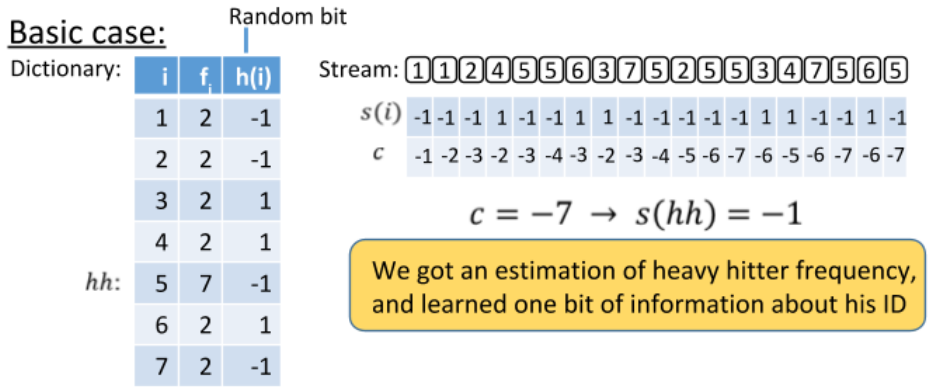


FIGURE 5.14: Count Sketch subroutine on an example stream: each non-heavy item appears twice, heavy hitter (5) appears 7 times, a random $+1/-1$ bit is assigned to each item, the algorithm maintains the sum of the random bits, and the final sum is an unbiased estimator of the heavy hitter frequency having the same sign as its random bit

.

The problem of finding heavy-hitters is well studied and there are memory optimal algorithms for ℓ_1 [109, 110] and ℓ_2[36] heavy hitters, of which we are most interested in the latter. Here we will describe a Count Sketch algorithm [36] which finds $(2\varepsilon, \ell_2)$-heavy hitters $O(1/\varepsilon^2 \log^2(mn))$ bits of memory.

Count-Sketch Algorithms. Consider a simplified stream with only one heavy item i', and every other item i appears in the stream only once. Let $h : [n] \Rightarrow \{-1, +1\}$ be a hash function which flips a $+1/-1$ coin for every item in the dictionary $i \in [n]$. If we will go through the stream $S = \{s_1, \ldots, s_m\}$ and maintain a counter $c = c + h(s_j)$, then at the end of the stream, with high probability, c will be equal to the contribution of i': $h(i')f_{i'}$ plus some small noise, while the majority of non-heavy contributors will be canceled by each other. The absolute value of c can be considered as an approximation of the heavy item's frequency. At the same time, the sign of c coincides with the random bit assigned to heavy

items; thus, it helps us to reveal the ID of the heavy hitter by eliminating from consideration all items of the opposite sign. Simply repeating the experiment $t = O(\log n)$ times in parallel will reveal the entire ID of the heavy item. However, if the number of repetitions would be significantly smaller, we will face the problem of collisions, i.e. there will be items with the same random bits as the heavy item in all experiments. Thus we end up with many false positives due to the indistinguishability of those items from the heavy hitter. An example stream is depicted in Figure 5.14. If our stream has k heavy hitters, all we need to do is randomly distribute all items of the dictionary into $b = O(k)$ different substreams. Then with high probability none of the $O(k)$ substreams will have more than one heavy hitter in it, thus for each stream we can apply the same technique as before. On figure 5.15 you can see the high-level intuition of both ideas described above:

1. Bucket hash to distribute items among b substreams (buckets)
2. Sign hash to assign random bit to every update
3. Row of b counters to maintain the sum of random bits
4. t instances to recover the IDs.

Thus for each item update we need to calculate t bucket hashes (specifying which substream/bucket this item belongs to) and t sign hashes. We then update one counter in each row of the Count Sketch table, which is t counters. In total the algorithm requires $t \cdot b = O(\log n)$ counters, which in turn requires $O(\log^2 n)$ bits of memory. For simplicity, here and later, we assume that $\log m = O(\log n)$, i.e. in our application mesh size is at most polynomially larger than the number of particles in the simulation).

All of the statements above can be proven formally [39]. Here we only show that using such a counter c provides us with an unbiased estimator $\hat{f}_i = c \cdot h(i)$ for the frequency of

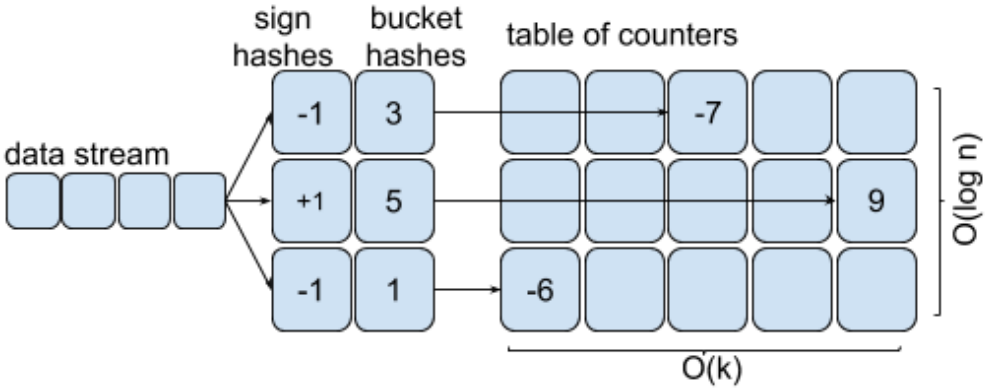


FIGURE 5.15: Count Sketch algorithm scheme: bucket hash to identify the counter to which we should add the sign hash. Repeat t times to recover the IDs.

the item i:

$$\forall i : E(c \cdot h(i)) = E\left(\sum_j f_j \cdot h(j)h(i)\right) = \sum_{i \neq j} E(f_j \cdot h(j)h(i)) + f_i = f_i,$$

where the last equality is due to the 2-independence of hashing $h(\cdot)$. However, the variance of such estimators might be quite large and depends mainly on the second frequency moment of the other items in the substream. At the same time we know that with high probability there is only one heavy hitter in each substream and we repeat the experiment $t = O(\log n)$ times. We take the median of those estimates, which reduces the variance and boost the probability for the final estimator to be within the approximation error from the real value. Summarizing, we have a data structure containing $b \times t = O(k) \times O(\log n)$ counters which maintain good estimates for the frequencies of the top k most frequent items, but we still have to find the values of their IDs. There are three approaches to do this:

1. **Count Sketch with Full Search(CSFS)**

   When all stream updates are processed we estimate the frequency of each possible

item in the dictionary $i \in [n]$ and find the top k most frequent.

pros: updates are fast and easy to run in parallel

cons: post-processing becomes very slow as the size of the dictionary grows

2. **Count Sketch with Heap(CSHe)**

   While processing each item, estimate its frequency and maintain the heap with the top k most frequent items.

   pros: post-processing takes zero time

   cons: updates require extra $\log k$ time-steps to update the heap

3. **Count Sketch Hierarchical(CSHi)**

   Maintain two sketches, the first one for stream of super-items $S' = \{s_j/1000\}$ and the second one for the initial stream $S = \{s_j\}$. When all stream updates are processed, we first estimate the frequency of each possible super-item $i \in [n/1000]$ in the dictionary of S' and find the top k most frequent super-items $K' = \{hh'_j\}_{j=1}^{k}$, then estimate the frequencies of all potentially heavy items $i \in [n]$ s.t. $i/1000 \in K'$ and find the top k most frequent items. This way we reduce the number of potentially heavy items to check. If necessary, more than 2 layers might be created.

   pros: post-processing is fast even for very large dictionaries

   cons: update time is ρ times slower and the algorithm uses ρ times more memory, where ρ is the number of layers.

CSFS contains set of $b \times t$ counters M, t hash functions $h_s : [n] \to \pm 1$ and t hash functions $h_b : [n] \to [b]$ which decides which counter in the t-th row element i corresponds to. In addition, CSHe contains the heap of pairs (item,frequency), and CSHi contains more than one sets of counters $\{M_i\}_{i=1}^{\rho}$. Let's define three following operations:

- Add(M, s_j):

  $$\forall i \in [t] : M_{i,h_{i,b}(s_j)} += h_{i,s}(s_j)$$

- Estimate(M, j):

  return *median* $\left(\left\{ M_{i,h_{i,b}(j)} \cdot h_{i,s}(j) \right\}_{i=1}^{t} \right)$

- UpdateHeap(H, j, $\hat{f}_j$):

  if $(j \in H)$: $\quad H[j] := \hat{f}_j$

  else if $(H.\text{top}().\hat{f} < \hat{f}_j)$: $\quad H.\text{pop}()$; $H.\text{push}(j, \hat{f}_j)$;

The Add() operation updates all the counters, Estimate() outputs current approximation for the frequency of the element j and UpdateHeap() maintains the top k most frequent items via updates of $(i, \hat{f}_i)$. The pseudo code for discussed functions is the following:

Similar construction is used in the algorithm Count Min Sketch [44]. The algorithm utilize the similar logic and the same size table of counters, however for each update it computes only one hash (to specify the bucket to be updated) rather than two in Count Sketch, and output the minimum over the estimates, rather than the median. Thus subroutines "Add" and "Estimate" are different:

- Add(M, s_j):

  $$\forall i \in [t] : M_{i,h_{i,b}(s_j)} += 1$$

- Estimate(M, j):

  return min $\left(\left\{ M_{i,h_{i,b}(j)} \cdot h_{i,s}(j) \right\}_{i=1}^{t} \right)$

We compare Count Sketch and Count Min Sketch experimentally. However the latter only finds ℓ_1 heavy hitters, so we expect it to be outperformed by Count Sketch.

Algorithm 17 Count Sketch with Full Search(CSFS)

procedure INITIALIZATION
    initialize $b \times t$ matrix of counters M with zeroes
end procedure
procedure PROCESSING THE STREAM
    for $s_i \in [m] = \{1, \ldots, m\}$ **do**
        Add(M,s_i)
    end for
end procedure
procedure QUERYING THE DATA STRUCTURE
    initialize a heap H of size k
    for $j \in [n]$ **do**
        $\hat{f}_j$ = Estimate(M, j);
        UpdateHeap(H ,j ,$\hat{f}_j$)
    end for
    for $i \in [k]$ **do**
        $(j, \hat{f}_j)$ = H.pop()
        return $(j, \hat{f}_j)$
    end for
end procedure

5.3.2 Implementation

Section 5.2 presented a halo finding tool using streaming algorithms that can be very useful even in systems with as low as 1GB memory. However, the running time of that tool was more than 8 hours on a desktop for a relatively small dataset. Here we provide a new algorithm based on an efficient GPU implementation. The core part of the halo finding tool relies on the implementation of the Count Sketch algorithm. All experiments in this section were carried out on the CPU Intel Xeon X5650 @ 2.67GHz with 48 GB RAM and GPU Tesla C2050/C2070.

Count Sketch Implementation. The data flow of the Count Sketch algorithm consists of 5 basic stages, i.e. for each item we need to do the following:

Algorithm 18 Count Sketch with Heap(CSHe)

```
1: procedure INITIALIZATION
2:     initialize b × t matrix of counters M with zeroes
3:     initialize a heap H of size k
4: end procedure
5: procedure PROCESSING THE STREAM
6:     for s_i ∈ [m] = {1, ..., m} do
7:         Add(M,s_i)
8:         f̂_j = Estimate(M, s_j)
9:         UpdateHeap(H ,s_j ,f̂_j)
10:    end for
11: end procedure
12: procedure QUERYING THE DATA STRUCTURE
13:    for i ∈ [k] do
14:        (j, f̂_j) = H.pop()
15:        return (j, f̂_j)
16:    end for
17: end procedure
```

1. Compute cell ID from XYZ representation of the particle
2. Compute t bucket hashes and t sign hashes
3. Update t counters
4. Estimate the current frequency for the item (find median of t updated counters)
5. Update the heap with current top-k if necessary

Below we consider different implementations of the Count Sketch algorithm with the argument to architectural decisions made:

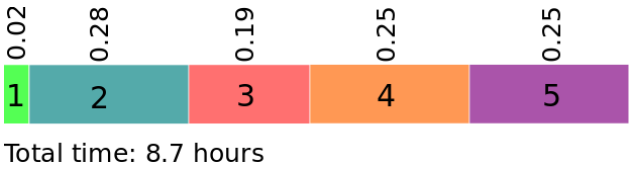
1. CPU:

   Purely CPU version of the Count Sketch has all five stages implemented on the CPU and described in details in [102]. As depicted below, it takes 8.7 hours to process one snapshot of all particles from the Millennium dataset. In the breakdown of

Algorithm 19 Count Sketch Hierarchical(CSHi)

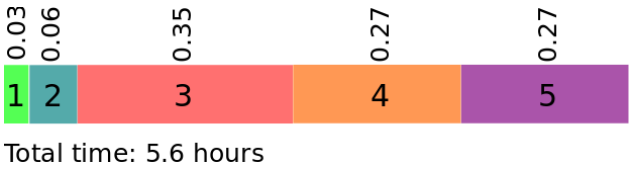
procedure INITIALIZATION
    initialize two $b \times t$ matrices of counters M_1 and M_2 with zeroes
end procedure
procedure PROCESSING THE STREAM
    for $s_i \in [m] = \{1, \ldots, m\}$ **do**
        Add(M_1,$s_i/1000$)
        Add(M_2,s_i)
    end for
end procedure
procedure QUERYING THE DATA STRUCTURE
    for $j \in [n/1000]$ **do**
        $\hat{f}_j$ = Estimate(M_1, j);
        if $\hat{f}_j > \theta_1$ **then**
            for $j' \in [1000j : 1000(j+1)]$ **do**
                $\hat{f}_{j'}$ = Estimate(M_2, j');
                if $\hat{f}_{j'} > \theta_2$ **then**
                    return $(j', f_{j'})$
                end if
            end for
        end if
    end for
end procedure

the profiler output below, where integer numbers denote the 5 stages of the Count Sketch algorithm and fractions show proportional amounts of time spent on that stage, we can see that the second stage is computationally the most expensive. The most straightforward improvement is to "outsource" this computation to the GPU. We implemented this idea, and we describe it further below.



2. CPU + hashes on GPU

   In this implementation, we are trying to "outsource" the most time intensive operation — calculating hashes. Recall that we need to compute $2t$ hashes. As long as t is a relatively small number (≤ 16), a naive parallelism which suggests computing all hashes for each particle in t parallel threads, will not provide a significant speedup due to the inability to saturate all cores (~ 2000) of the graphics card. Thus to improve performance even further, we need to make use of data parallelism, which assumes computing hashes for a batch of updates at the same time. Such an approach is straightforward due to the fact that computing hashes are identical operations required for all particles and those operations can be performed independently. As illustrated below, the GPU computes hashes almost for free, compared to stages 3,4 and 5, and total time drops by 35%. The next bottleneck is stage 3, during which the algorithm updates counters. Although it is just $2t$ increments or decrements, they happen at random places in the table of counters. This makes it impossible to use CPU cache and memory access becomes a bottleneck for the algorithm.



3. GPU + heap on CPU

   Updating counters (stage 3) and estimating current frequencies (stage 4) are two very connected stages. If we keep them together we can significantly save on the number of queries to the memory. Implementing a time efficient heap (stage 5) on the GPU is quite challenging, due to hardware features. Thus our next implementation takes advantage of the CPU for maintaining the heap, while doing all other

computations and storing the table of counters on the GPU. The basic data flow can be described as follows:

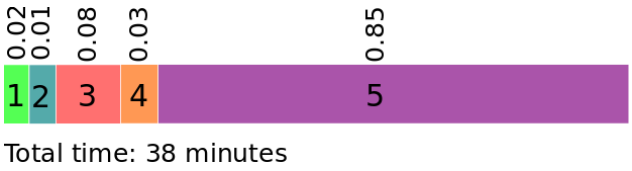
(a) CPU sends a batch of particles in XYZ representation onto GPU

(b) GPU processes all particles in parallel: compute cell ID, compute hashes, update counters and estimate frequencies

(c) GPU sends a batch of estimates back to the CPU

(d) CPU maintains heap with top k items using estimations from GPU

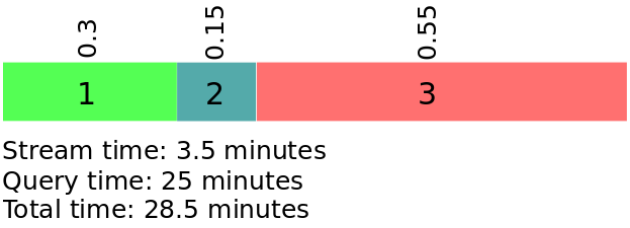
It can be seen below that adopting such an approach pushed the total time of the algorithm down to 38 minutes. In the breakdown of the profiler, one can see that updating the heap became a new bottleneck for the algorithm.



4. GPU without heap

While heap on the CPU is quite efficient, it still slows down the process quite seriously, especially when the top k gets larger and reaches 10^6. On large datasets this might cause many items to have an update time close to $\log k$. Moreover, keeping the heap on the CPU forces the GPU to send a lot of data back to the CPU. Avoiding this data transfer would improve the slowest memory operation by a factor of 2. Thus we decided to switch from Count Sketch with Heap (CSHe) to Count Sketch with Full Search (CSFS), both of which were broadly described in the previous section. The CSFS algorithm works in two modes: update mode, which encompasses

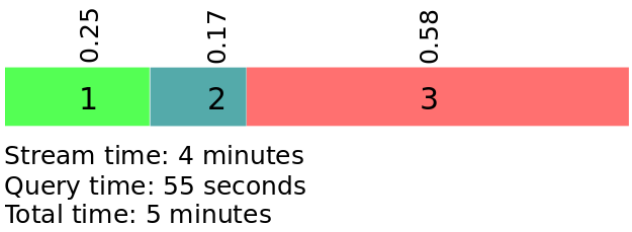
calculating hashes and updating counters, and estimate mode, which deals with estimating the frequency for all cells and emitting the top k. The CSFS algorithm is first invoked in update mode for the entire stream, and when it finishes, the generated table of counters is used as input to estimate mode. While in estimate mode, we still need to maintain the top k items and do it on the GPU. This can be done semi-dynamically by adding to the array all items which are larger than some threshold. Then, if we have more than k items, we will raise the threshold and rearrange elements in the array, deleting those items which do not satisfy the new threshold. If we grow the threshold geometrically we can guarantee that such "cleaning" step won't happen too often. Such an approach cannot be applied to the CSHe algorithm due to the possibility of two updates for the same cell. In the figure below, the stream time, which includes only the update mode, takes only 3.5 minutes, while the estimate mode takes 25 minutes. The time of the estimate mode, i.e. query time, linearly depends on the size of the mesh, due to the necessity to estimate the frequency for every cell in the mesh. For example, in the same experiment for the mesh with $5 \cdot 10^8$ cells, query time would be less than 10 seconds.



5. GPU hierarchy

   As it was already discussed in the previous section, one of the ways to decrease query time is to eliminate the full search and implement it as a search tree instead. In our case, the search tree (hierarchy) will contain only two layers. By grouping cells together we can find the heavy super-cells first (using a small mesh), then search for

heavy cells only inside heavy super-cells. We will merge cells by their IDs in the top layer with a dictionary size of $\sim 10^8$, find top $c \cdot k$ super-cells and find top k cells inside the selected heavy super-cells, where $c > 1$ is a small constant. As can be seen below, such an approach reduces query time from 25 minutes down to 55 seconds. However, it requires twice the amount of memory due to the need to store a table of counters for each layer. It can also be observed that time performance of the update mode gets worse, due to the necessity to calculate twice as many hashes and update twice as many counters. The total time of the algorithm is 5 minutes, which is very impressive for the size of the dataset and the mesh. The total performance improvement over the sequential CPU implementation is more than 100-fold.



Here we will briefly introduce the key architectural decisions in the implementation of the “GPU without heap” version of the algorithm. While it is not the most efficient implementation, it is easier to explain. At the same time, it makes it straightforward how to extend it to the "hierarchical" version. The graphical processor is a separate device that has many limiting features compared to the CPU. In this project, all our experiments leverage the CUDA platform to make use of the graphical processor’s capabilities. [117].

A GeForce GTX 1080 has 20 multiprocessors (SM) each with 128 cores (threads). CUDA introduced a block/thread approach, such that all computations are grouped into blocks, where one block is always implemented on only one SM. Within a block, we can specify how to share computation between threads. CUDA has three layers of memory:

1. **Global memory:** accessible within the device and conventionally is quite large (up to 8 GB). It is also the only type of memory that can be used to copy to or from RAM. At the same time, it is the slowest memory on the device.

2. **Shared memory:** accessible from within the block and shared among all threads of that block. Shared memory is ~ 10 times faster than global memory, however, it is very limited with $\sim 48 - 64$KB per SM.

3. **Registers:** there are 2^{15} 32-bit registers per SM. They are as fast as shared memory, but visible only to the thread.

Storing a table of counters for Count Sketch is possible only in global memory. Primarily, this is due to the large size of the counters $\sim$ 1GB. Secondly, counters are accessed in random order, which makes it impossible to store some localities in the shared memory. In our implementation, each block is in charge of exactly one update of the stream. In order to make an update, one needs to calculate $2t$ hash functions and update t counters, thus we distributed this work among t threads, each calculating two hashes and updating one counter.

Note that to avoid memory access conflicts we need to use atomic operations, which are present in CUDA. However we expect the number of conflicts not to be very large: while the typical width of the table is 10^7 counters and the maximum number requests is bounded by the number of GPU threads (which is ~ 2000 in our case), the probability of collision is negligible. In practice, we can see that using non-atomic operations would give us at most a 10%-fold gain in time performance. The pseudo code for each thread is presented in Algorithm 20.

After the stream is processed, we need to find the IDs of the heavy hitters. As described earlier, we need to find an estimation for each item in the dictionary. Here we

will use the same approach as for stream processing. Each block will be in charge of one cell. Each thread will be in charge of an estimation based on one row of Count Sketch counters. The procedure for each thread is described in Algorithm 20.

Algorithm 20 GPU thread code for Count Sketch

```
1:  procedure UPDATE(cellID)
2:      i = threadID;
3:      M[i, h_{i,b}(cellID)] += h_{i,s}(cellID)
4:  end procedure
5:
6:  procedure ESTIMATE(cellID)
7:      shared estimates[t];
8:      shared median;
9:      j = threadID;
10:     f̂ = M[i, h_{j,b}(cellID)] · h_{j,s}(cellID);
11:     estimates[j] = f̂;
12:     synchronize
13:     int above, below = 0;
14:     for i ∈ [t] do
15:         below += (estimates[i] < f̂)
16:         above += (estimates[i] > f̂)
17:     end for
18:     if above <= t/2 and below <= t/2 then
19:         median = f̂
20:     end if
21:     synchronize
22:     if j = 1 and median > θ then
23:         return median
24:     end if
25: end procedure
```

Note that we find the median using a very naive algorithm — for each item of the array check if it is a median by definition. That is thread i would be in charge of checking if the number of estimates smaller than estimates$[i]$ is equal to the number of estimates larger than estimates$[i]$, and reporting/recording the found media if so. This is one of the reasons why all estimates should be reachable by all threads, and thus should be stored in

the shared memory. While in sequential implementation this approach would take $O(t^2)$ time steps, here we use t parallel threads, ending up with time complexity of $O(t)$.

To boost the time performance even further we can apply sampling. However, one should not expect performance to improve linearly with the sampling rate, because of necessity to compute sampling hashes for all particles. The dependency of the time performance on the sampling rate is depicted in figure 5.16. From that graph, one can see that changing the subsampling rate from 8 to 16 is the last significant improvement in time performance.

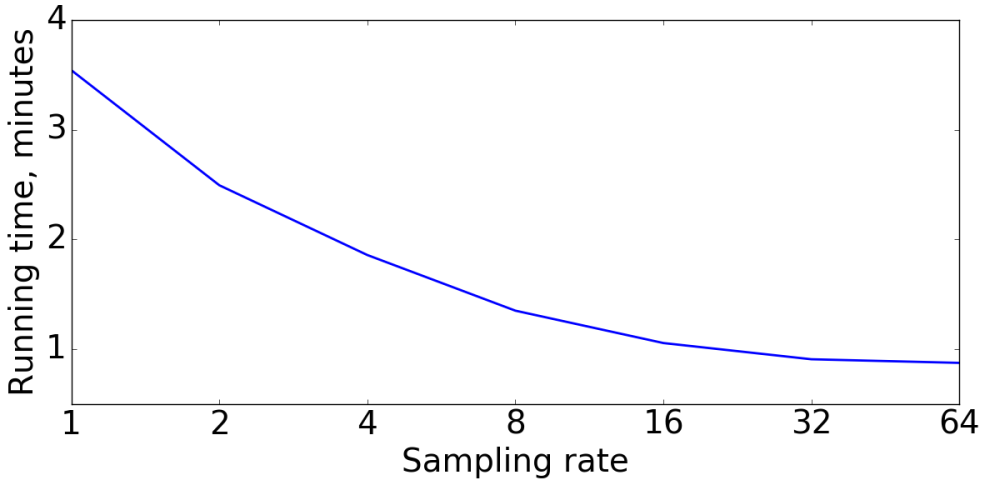


FIGURE 5.16: Dependency of time performance on sampling rate.

5.3.3 Evaluation

In this section, we present a tool which is capable of finding up to $10^5 - 10^6$ densest cells in state of the art cosmological simulations for an arbitrary sized regular mesh. Moreover, the proposed technique makes these procedures available even for the desktop or a small server. In this section, we evaluate this claim. We do this in two steps. In the first, which we call the *algorithmic evaluation* we compare the rank order produced by the heavy hitter

algorithm directly to the exact results. In the second, we perform a scientific evaluation and analyze what the effects are of the randomized nature of the approximate algorithm to various statistical measures of astrophysical interest; namely the tail end of the counts-in-cell distribution and the spatial clustering of excursion sets.

Evaluation Setup. For testing and evaluation, we use the Millennium dataset [96] with 10^{10} particles in a cube with side length 500 Mpc/h. The cell size in the grid is 0.1 Mpc/h, thus the total grid contains 1.25×10^{11} cells. Our goal is to find top 10^5 to 10^6 heaviest cells. Those numbers are important to understand some decisions in choosing the specific architecture of the implementation.

The data, originally stored in the GADGET[133] format, is reorganized, such that every 64 bits contains 3 coordinates for one particle. This reorganization helps to reduce the number of global memory writes inside the GPU. After such a reorganization the entire dataset weights in at 80 GB. One of the time performance bottlenecks in such settings is reading data from the hard drive. We implemented a parallel I/O system that includes 3 SSDs and 24 HDDs without data replication, and this way we reduced the pure I/O time from 15 minutes to 20 seconds. For comparison purposes all experiments were accomplished on two different hardware configurations:

1. AMD Phenom II X4 965 @ 3.4 GHz, 16 GB RAM, GPU GeForce GTX 1080.

2. Intel Xeon X5650 @ 2.67GHz, 48 GB RAM, GPU Tesla C2050/C2070.

Top-k Cells First, let's introduce different ways of finding the top k most frequent cells with exact counts. Given a set of particles in the simulation box and a regular grid of a fixed size we need to find k cells of the grid containing the largest numbers of particles, together with the IDs of those cells. The algorithm is required to return an estimate of

the number of particles in each cell. The most straightforward solution to this problem is to count the number of particles in each cell precisely. Such an exact algorithm might be described as follows:

1. Create a counter for every cell in the grid
2. While making a pass through the dataset, update the cell counters based on the position of the particles
3. Find k "heaviest" cells and return their IDs and exact counts

This solution breaks down in step 1 once the size of the mesh is too large to store all counters in memory.

It is possible to remove the memory problem at the expense of worsening time performance by making multiple passes over the data, as in the following algorithm. Assuming the memory is about a factor $1/\lambda$ of the total size of the grid:

1. Create a counter for every cell in the range $[i - n/\lambda, i]$, and use the basic algorithm above to find k "heaviest" cells in the range and call them $topK_i$.
2. Repeat previous step for all ranges $i \in \{n/\lambda, 2n/\lambda, \ldots, n\}$ and find top k "heaviest" cells in $\cup_i topK_i$

This multi-pass trick becomes unfeasible when the size of the mesh grows too large compared to the available memory, as it would take too many passes over the data. However, to evaluate how well our algorithm approximates the exact top k with counts, we do need to have exact counts. That was one of the reasons why [102] restricted themselves to relatively small meshes. In the current section, we show results for meshes of sizes 10^8, 10^{11} and 10^{12}. We will provide algorithmic evaluation only for 10^8, where the naive precise algorithm can be applied, and for 10^{11}, where we apply the trick described above

and do 20 passes over the dataset. For the mesh of size 10^{12}, algorithmic evaluation is more challenging and therefore only the scientific evaluation will be performed.

Evaluation of algorithm. In this section, all experiments are for the mesh size 10^{11}. Fig. 5.17 shows the distribution of exact cell counts for the top 10^7 cells.

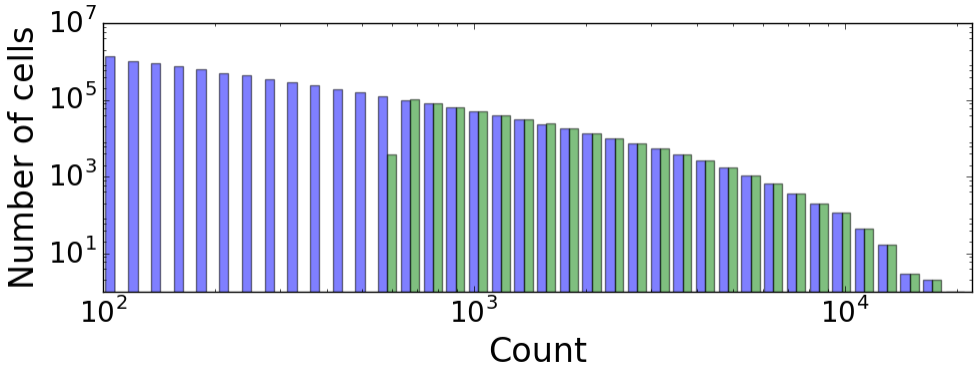


FIGURE 5.17: Cell density distribution for the top $0.5 \cdot 10^6$ cells found by Count Sketch (in green) and the top 10^7 cells found by exact counting (in blue).

Most experiments in this section use a Count Sketch with parameters $t = 5, b = 10^7$ and $k = 5 \cdot 10^5$. A motivation for these values will be provided later. To understand how well the Count Sketch approximates the exact counts and how well it reproduces the rank order, we determine how the relative error grows with the rank inside the top-k cells. To do so, for each cell i we find its count c_i and rank r_i in the output of an exact algorithm and its count $\hat{c}_i$ in the Count Sketch output. If cell i is not present in the Count Sketch output, i.e. not among its top k heaviest cells, we define $\hat{c}_i = 0$. Fig. 5.18a shows, in blue, the dependency on rank r_i of the relative error, defined as $|c_i - \hat{c}_i|/c_i$. Here we use a bin size of 100 in i for the averaging.

The relative error is shown in green and is determined for cells which were among the top k for both the exact and the Count Sketch counts. By ignoring the cells not found in the

Count Sketch, the relative error is artificially reduced. On the other hand, treating those cells as empty $\hat{c}_i = 0$ pushes the error rate up significantly. This overestimates the error compared to the count that might have been determined had the Count Sketch included those cells, for example by using a larger value of k.

As we can see, up to a rank of $\sim$ 250000 the algorithm works reliably and has quite low approximation error. However, at higher ranks the error grows rapidly. The main cause of this is the loss of heavy cells, rather than a bad approximation of the counts for the cells that were accepted by the Count Sketch. This is shown by the fact that the green line remains low.

Fig. 5.19 shows the same graphs, but now plotted against the over-density $\delta_i = (N_i - < N >)/ < N >$ in cells. This quantity is more meaningful from an astrophysical point of view compared to the rank. It shows that the errors are stable for a large range of over-densities, but very quickly shoot up near a threshold. That threshold depends on the size of the cell as the comparison in Fig. 5.19 shows. Note that the size of the cell for any specific dataset would influence the number of particles in the each of the top-k heavy cells and ℓ_2 norm of the stream.

There is a straightforward reason why the approximate algorithm loses so many heavy cells. Before we explain it we need point out three important facts. First, Fig. 5.20 shows that the absolute error is about constant for all cells. This can be understood from the theoretical arguments in [36], which state that all estimations have an additive approximation error of $\varepsilon\ell_2$, i.e. for each cell, error does not depend on the count, but only on the ℓ_2 norm of the entire dataset. Second, as we can see from Fig. 5.17, the number of cells is increasing exponentially with the count going down (this is a property of the mass function in the cosmological simulation). Third, for the cells with ranks near 250000, the actual count is $\sim$ 820, and for the cells with ranks near 500000 the actual count is $\sim$ 650.

While searching for the top-k cells using Count Sketch estimations we will face two types of errors:

type 1: false rejection of heavy cells caused by *underestimation* of the true count due to approximation error

type 2: false exclusion of heavy cells caused by *overestimation* of counts of cells below the top-K selection criterion.

We expect that having 250000 elements with counts between $\sim$ 650 and $\sim$ 820 with an average approximation error $\sim$ 80 (and ranging up to 250) will cause significant loss of heavy cells in the top k. We can see this in Fig. 5.18a, which also depicts the recovery rate. Thus we can conclude that the main cause for missing heavy cells in the output is the fact that many cells have counts which are relatively close to the approximation error of the algorithm. We have tested this conclusion by running the algorithm for larger cell sizes, with significantly larger expected counts. This increases the typical $|c_i - c_j|$ for cells i and j with small rank distance. The result of an experiment with a cell size of 1 Mpc/h is shown in Fig. 5.18b, which corroborates our hypothesis. The difference in the results for different mesh sizes is even more obvious in the relative error vs. exact count graphs in Fig. 5.21a and Fig. 5.21b. Note that closer to the cut-off threshold the algorithm is tending to overestimate the count rather then underestimate. This behaviour is reasonable due to the fact that only one-way error is passing the threshold test, while all items with underestimated counts are discarded by the algorithm.

We know that every ℓ_2 heavy hitters algorithm catches all ℓ_1 heavy items and some items which *are* ℓ_2-heavy but not ℓ_1-heavy. While asymptotically the space requirements for both algorithms are the same, the time performance for ℓ_1 algorithms is better in practice than for the ℓ_2 algorithms. It is therefore of interest to compare the two in the specific

application to our problem. To do so we compared the Count Sketch with the intuitively similar ℓ_1 Count Min Sketch algorithm. Fig. 5.22 shows that the approximation error differs significantly, with the Count Sketch algorithm giving much more accurate results.

As it was mentioned earlier, a random sampling of the particles before feeding them to the algorithm can significantly improve the time performance of the entire procedure. To investigate the influence of such sampling on the approximation error we carried out experiments comparing different sampling rates for mesh sizes 1 Mpc/h and 0.1 Mpc/h. From the figures 5.23 and 5.24 we can see that in both cases a sampling rate of $1/16$ still provides a tolerable approximation error. It is important to recall that the time performance does not scale linearly with the sampling rate due to the need to compute the sampling hash function for each element. This operation is comparable in time to processing the element through the entire data flow without skipping the Count Sketch.

The crucial advantage of the algorithm presented here compared to existing algorithms is the improvement in memory usage. Traditional algorithms often require complete snapshots to be loaded into memory, which for state-of-the-art cosmological simulations implies they cannot be analyzed on a small server or even one desktop. For a cell size of 1 Mpc/h and a box size of 500Mpc/h our mesh would contain only $1.25 \cdot 10^8$ cells which require only 500 MB for a naive algorithm and provides an exact solution. Such a low memory footprint makes the naive solution feasible even for a laptop. For a cell size of 0.1 Mpc/$h$with the same box size the memory requirements would be $1000\times$ larger and barely fit onto a mid-size server.

Next, we investigate the trade-off between time performance, memory requirement and approximation error in more detail. The Count Sketch data structure consists of $t \times b$ counters, which also sets the memory requirements. The graph in Fig. 5.25 shows the approximation error for different combinations of b and t.

The CS algorithm provides a tolerable error rate as long as $t \times b \geq 64 \cdot 10^6$, except for the case $(t, b) = (4, 16 * 10^6)$ which has too small of a t, causing a high rate of false positives; we will provide more details in the next paragraph. To better understand the spectrum of possible error rates, we consider the rates at rank 400000, where the frequencies of the cells are already quite low. For all combinations of the parameters, the algorithms start losing a significant fraction of correct heavy hitters near that value. The following table shows these error rates as a function of b and t.

t	16	12	8	16	12	16	8	12
b $/10^6$	16	16	16	8	8	4	8	4
Error	0.27	0.34	0.42	0.47	0.51	0.64	0.67	0.72

Algorithms with a similar space usage ($\propto t \times b$) have a similar error rate, but the solution with the larger number of rows is generally somewhat better. This can be easily explained using a theoretical argument: the largest portion of the relative error depicted is due to losing true heavy hitters; this happens due to the fact that the algorithm finds "fake" heavy hitters, and those push the true heavy hitters with a smaller frequency out of the top group. A fake heavy hitter can appear only if it collides with some other true heavy hitter in at least half of the rows. Thus, the expected number of collisions can be computed as a total number of different items n times the probability to have collided with at least one heavy hitter, which is k/b, and then this should happen in $t/2$ independent experiments. Therefore expected number of collisions is $n(k/b)^{t/2}$. From that dependency, we can see that under fixed $t \times b$, the larger value of t is always better. For example, if we want to find only one heavy hitter and minimize the space usage which is proportional to $t \times b$, then the most efficient way is to take $b = 2$ and $t = c \log n$. However, this would force us to increment or decrement $O(\log n)$ counters for each update, which is much slower, $O(\log_b n)$ with $b >> 2$. Theoretically running Count Sketch with

$t = 8$ would be twice slower than with $t' = 4$, due to the need to compute twice more hashes and increment twice more counters. In practice, we saw almost the same, mostly due to the fact that computing hashes and updating counters takes around 75% of the total running time in the "GPU hierarchy" implementation. From these examples, we can understand the nature of the space versus time trade-off, and we can see this behavior in the graph and in the table for the pairs $8 \times 16 \cdot 10^5$ and $16 \times 8 \cdot 10^5$, $16 \times 4 \cdot 10^5$ and $8 \times 8 \cdot 10^5$ and others. Note that increasing both b and t will provide better approximation and lower false positive rate, however increasing t would significantly push time performance and space (if we will keep b fixed) up, while increasing b will not influence the time performance but still push the memory usage. In all our experiments we were limited by the memory of the GPU, which for both devices was only 8GB.

Evaluation of the model. Here we will evaluate the quality of the model for two specific problems: finding halos and the analysis of excursion sets. To do so, we will try to solve the problem using Count Sketch and its ability to find top k densest cells in the simulation box.

In [102] authors showed a simple solution for using the heavy cells to find heavy clusters by making a second pass through the data set and storing locally the particles which belong to one of the heavy cells. This is possible because the number of particles in those heavy cells is much smaller than that of the entire data set, and we can even store them in main memory. This implies that any traditional in-memory algorithm can be applied offline. This scheme is illustrated in Fig. 5.26. In this section, we will not repeat the entire chain of this computation, but will simply check the number of halos contained in the top k cells.

It turns out that to find the centers of the 10^5 most massive halos we need to find the

$\sim 1.8 \cdot 10^5$ heavy cells, i.e. the centers of the top 10^5 most massive halos are contained in the top $\sim 1.8 \cdot 10^5$ heavy cells. Then running an offline in-memory halo finder, such as Friends of Friends [47] or any other halo finder of choice [91], on the particles located only inside the top $1.8 \cdot 10^5$ heavy cells (and it's immediate neighbours) will provide us with more precise halo centers and mass distribution for each halo. We emphasize on the fact that in current manuscript we find only the centers of the haloes and leave finding actual borders and mass distribution for future research. Hence the streaming approach can be considered as a sieve step, allowing us to efficiently remove the particles which are not in the largest halos from further consideration. The resulting filtered data set is significantly smaller in size, thus one can apply offline algorithms. In [102] we showed how to find 10^3 largest halos while working with a data set of size 10^9: find the top $2 \cdot 10^3$ heavy cells and run an offline algorithm on the particles that are located only inside the heavy cells.

Applying the same approach to find the top 10^6 heaviest cells in the Millennium data set containing 10^{10} particles would be challenging, but still manageable:

1. top 10^6 haloes contain $\sim 3.8 \cdot 10^9$ particles $=$ 45 GB;
2. top 10^5 haloes contain $\sim 2.5 \cdot 10^9$ particles $=$ 31 GB;
3. top 10^4 haloes contain $\sim 1.4 \cdot 10^9$ particles $=$ 16 GB;
4. top 10^3 haloes contain $\sim 0.8 \cdot 10^9$ particles $=$ 9 GB;

Thus we indeed can afford to run offline in-memory halo finder and locate $\sim 10^5$ haloes on a desktop or a small size server. At the first glance, it seems that the memory gain is not significant: initial dataset weights $\sim 90GB$, i.e. for the top $\sim 10^5$ haloes the gain is at most factor of 3 (factor of 5 for the top 10^4), at the cost of introduced approximation and non-zero probability of failure. However, initial dataset does not provide an option

of running offline halo finder sequentially in several passes on the machine with very low memory usage. Our filtering step provide an opportunity to run it on the machine with just 2GB of memory, the algorithm will require make more passes over the data, i.e. to find the top 10^4 haloes one will need to make ~ 10 passes while working under 2GB memory restriction.

We should take into account that number of particles in each halo is growing with the size of the data set. Additionally, using 10 times larger top will significantly increase the total number of particles one need to store in the memory, while applying offline algorithm. Hence, finding the top 10^5 haloes on the Millenium XXL dataset with $3 \cdot 10^{11}$ particles is less feasible as a low-memory solution:

1. top 10^6 haloes contain $\sim 31 \cdot 10^9$ particles $= 372$ GB;
2. top 10^5 haloes contain $\sim 9 \cdot 10^9$ particles $= 108$ GB;
3. top 10^4 haloes contain $\sim 2 \cdot 10^9$ particles $= 24$ GB;
4. top 10^3 haloes contain $\sim 0.4 \cdot 10^9$ particles $= 4.8$ GB.

From the list above we can see that to keep everything on the small server, proposed approach can help to find at most top 10^4 haloes in one extra pass or 10^5 haloes in $\sim 8 - 10$ passes, which we state as a result in the current section and keep the further improvement as a subject for future investigation. Note that compression level for the top 10^5 particles is 33 times (148 times for the top 10^4 haloes). However requirement to make more than 2 passes and utilize ~ 24 GB on the second pass is very restrictive and better techniques should be proposed for after-processing. Among the most straightforward solutions are sampling and applying streaming approach hierarchically for different cell sizes.

As described in the introduction, a connection can be made between the heavy hitters in the collection of grid cells and excursion sets of the density field. We want to determine

spatial clustering properties of these over-dense cells and determine if the algorithm by which this set is determined has an influence on the spatial statistics. To do this we have extracted the locations of heavy hitter grid cells in the Count Sketch result and determined their clustering properties using the two-point correlation function $\xi(R)$ [e.g. 120]. We compare this to the 2-pt function calculated on the cells in the exact excursion set. Adopted cell size is 0.1 Mpc/h. As the results in Fig. 5.27 show, for the over-densities that can be reliably probed with the streaming algorithm the exact and Count Sketch results are indistinguishable. The main deviations are due to discreteness effects for the smaller high-density samples. As an aside, we note that in Fig. 5.28, the higher-density cells cluster more strongly than the lower-density cells, as expected [83].

Millennium XXL. Running on the Millennium dataset, we could still find the "top-k" densest cells exactly using quite moderate time and memory. Even when the full density grid was too large to fit in memory, we could make multiple passes over the data and determine parts of the grid. Those experiments are necessary for evaluating the precision of the output from the randomized algorithm, but on our medium sized server they take about a day to complete.

In this section, we describe an experiment on the results of the Millennium XXL simulation [7], which contains around 300 billion particles, and hence is 30 times larger than the Millennium dataset. Its box size is 3 Gpc and we will use a cell size of 0.2 Mpc. Thus our regular mesh would contain $\sim 3.4 \cdot 10^{12}$ cells and need $13.5TB$ of RAM to be kept in memory completely (using 4-byte integer counters). While this is beyond the means of most clusters, our algorithm will be able to solve the problem with a memory footprint under 4GB while keeping the lapse time under an hour.

Before we describe some technical details of the experiment, we need to clarify the

process of evaluation, as we are now not able to produce exact counts in a reasonable amount of time. Hence, we consider only the following two ways for evaluating the accuracy of our results:

1. **From the size of the exact counts**

   While we can not determine the top-k most dense cells precisely, we can still make a second pass over the data and maintain counters for some subset of cells. We will use this opportunity to evaluate the approximation error from Count Sketch, but only for those particles which were output by the algorithm. Note that this verification is not as reliable as the one used earlier in this section because it does not catch any false negative items, i.e. the items which are supposed to be in the top-k, but were lost by the algorithm. But this way we can evaluate the approximation error, and get a preliminary estimation of the false positive rate.

2. **From astrophysics**

   Running a simulation with a larger number of particles provides us with more stable quantities. While we do not have any way to verify them precisely, we know that the spatial statistics should be more or less close to those from smaller size simulations. This evaluation is more qualitative than quantitative, but it will definitely be alarming if serious deviations are present.

We ran the "GPU hierarchical" version of the Count Sketch. We then made a second pass over the dataset where we determined the exact counts, restricted to the cells found in the first pass. While we do not know the cutoff frequency for the top-k, we can still approximately estimate the false positive rate: if all cells in the top-k output by Count Sketch have frequencies larger than 1700, then every item with a true frequency less than 100 would be considered as false positive. Initially, we set the same number of counters in

the Count Sketch table as before: 16 rows and 10^7 columns. However the result was quite noisy and had a very high rate of false positives: around 60000 had a frequency lower than 100, while the top-k frequency cutoff is around 1700. Then we ran Count Sketch with 24 rows, and the number of collisions dropped accordingly to approximately 800. The graph depicting relative error of the Count Sketch can be observed in Figure 5.29. It is evident that approximation error is more than twice that of the experiment with the Millennium dataset (refer to the Figure 5.18a). This can be explained by the size of the dataset, as long as algorithm's guaranteed approximation error is $\varepsilon\ell_2$, then with ℓ_2 for the XXL dataset the error is significantly larger. If the further astrophysics analysis will require better approximation error we can increase the width of the Count Sketch table.

In Fig. 5.30 we compare the two-point correlation function for all cells with $\delta \geq 20000$ for both Millennium and Millennium XXL. For the Millennium XXL result we use the Count Sketch, for the Millennium we use the exact over-densities, both in cells of size 0.2 Mpc. The XXL has a volume that is 216$\times$ the volume of the Millennium run and hence much better statistics. Nevertheless, the results are compatible with each other.

We ran the experiment on the small server with the following characteristics: Intel Xeon X5650 @ 2.67GHz, 48 GB RAM, GPU Tesla C2050/C2070. Our I/O was based on 4 Raid-0 volumes of 6 hard drives each. The total time for the I/O is 30 minutes. Due to the fact that I/O is implemented in parallel, if the time of the algorithm is higher than I/O, then I/O is "for free", this happens due to the fact that we can read a new portion of the data from the disk, while the GPU is still processing the previous portion. Our algorithm time on the Tesla card is 8 hours. In contrast, on the GTX1080, the estimated running time is expected to be less than an hour, which we calculated by running a small portion of the data. However, we were not able to carry out the entire experiment on the GTX1080, lacking a server with this card, and parallel I/O with a significant amount of storage.

For comparison, had we calculated the exact density field on a grid we would have required about 13.5TB of memory. Alternatively, on the machine with 48 GB RAM, we'd need about 280 passes over the data to calculate the exact field in chunks small enough to fit on the machine.

5.4 Conclusion

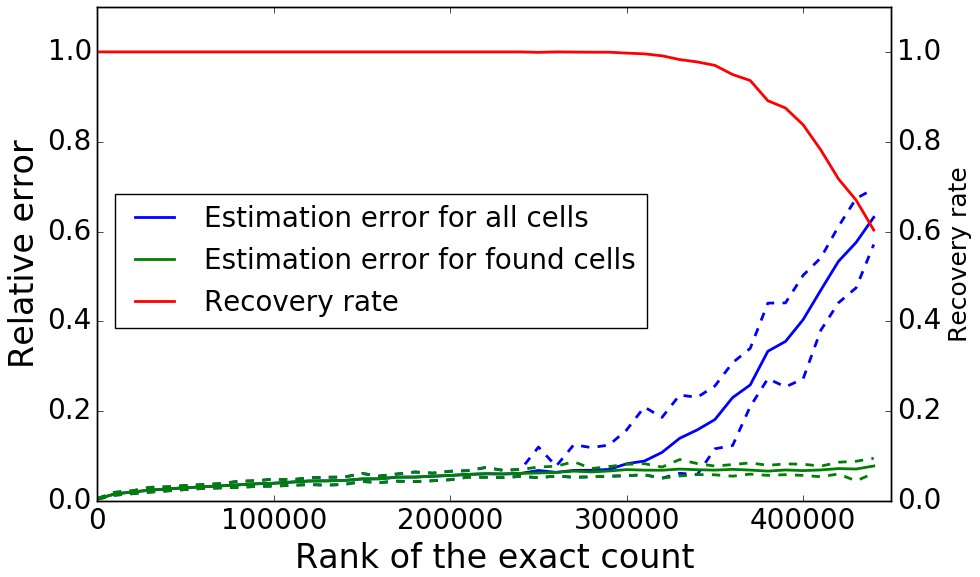
In Section 5.2 we found a novel connection between the problem of finding the most massive halos in cosmological N-Body simulations and the problem of finding heavy hitters in data streams. According to this link, we have built a halo finder based on the implementation of Count-Sketch algorithm and Pick-and-Drop sampling. The halo finder successfully locates most ($> 90\%$) of the k largest haloes using sub-linear memory. Most halo-finders require the entire simulation to be loaded into memory. But our halo finder does not and could be run on the massive N-body simulations that are anticipated to arrive in the near future with relatively modest computing resources. In section 5.2 we did not pay much attention to the time performance and all experiments were mainly focused on precision vs. memory trade-off. But both Count-sketch and Pick-and-drop Sampling can be easily parallelized further to achieve significantly better performance. The majority of the computation on Count-sketch is spent on the calculations of $r \times t$ hash functions. A straight forward way to improve the performance is taking advantage of the highly parallel GPU streaming processors to improve the performance of calculating a large number of hash functions.

In Section 5.3, we pushed the limits of these algorithms toward datasets with sizes up to $\sim 10^{12}$ particles, while still keeping all computations on a single server, or in some cases, even a desktop. To make this possible, we implemented the Count Sketch algorithm

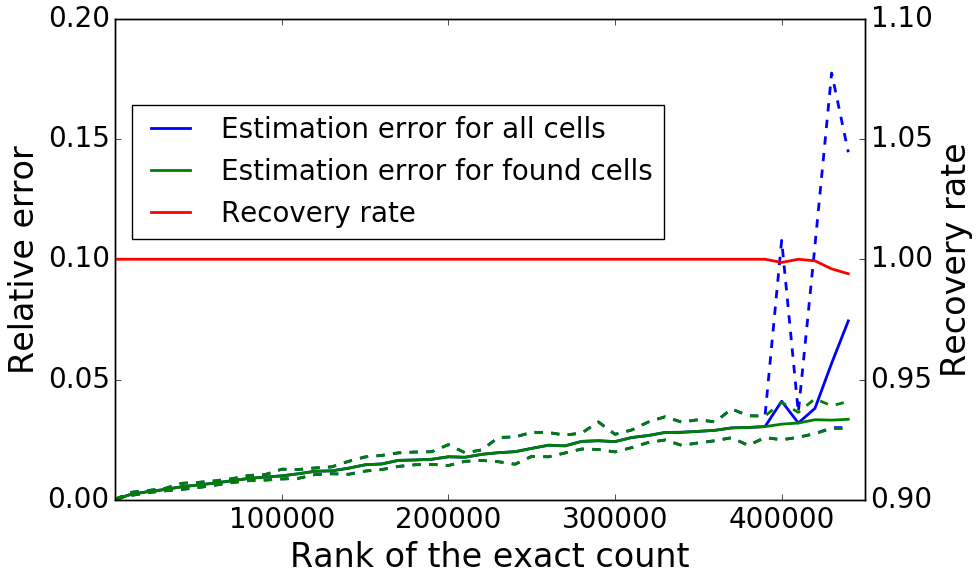
in a batch streaming setting and ported it to a graphics processor (GPU) using the CUDA environment. This approach significantly improves the time performance while using much less memory, enabling the possibility of processing very large datasets.

We have benchmarked several implementations, varying time, precision, and memory usage. We conclude that GPUs offer a perfect infrastructure for supporting the batch streaming model. Note that in the current project, while all experiments were carried out on a single GPU, we did not change the Count Sketch data structure. Thus, two or more sketches computed on different nodes, if merged, will approximate the cell counts for the combined stream of updates. Therefore, this approach can be used in distributed settings, where each node will have its own stream of updates and its own data sketch, and at the end all the sketches can be summed to find the heaviest cells. An implementation of this algorithm on distributed storage, using several GPUs, is crucial due to IO being the main bottleneck and will be considered in future work. Additionally, we will investigate the application of other classic streaming algorithms in a batch streaming model, on the GPU. Among other future directions, we are considering structure finding in 6D space, where each particle is described by its velocity and location; we are also considering hierarchical sketch-based clustering, to find the top-k heaviest cells in meshes of different sizes in parallel.

Though the emphasis in the entire chapter is on the technical application of these streaming algorithms in a new context, we showed, where possible, that these randomized algorithms provide results consistent with their exact counterparts. In particular, we can reproduce the positions of the most massive clusters and the two-point correlation function of highly non-linear excursion sets. The nature of these algorithms currently precludes the possibility of sampling the full density field or the full halo multiplicity function, though we are working on algorithms to at least approximate those statistics.



(A) Cell size = 0.1 Mpc/h



(B) Cell size = 1 Mpc/h

FIGURE 5.18: Relative error vs. rank for (a) cell size 0.1Mpc/h and (b) cell size 1Mpc/h. Each experiment was carried 20 times. Dashed lines depict the maximum and the minimum, while the solid line shows the average over those 20 runs for each rank value.

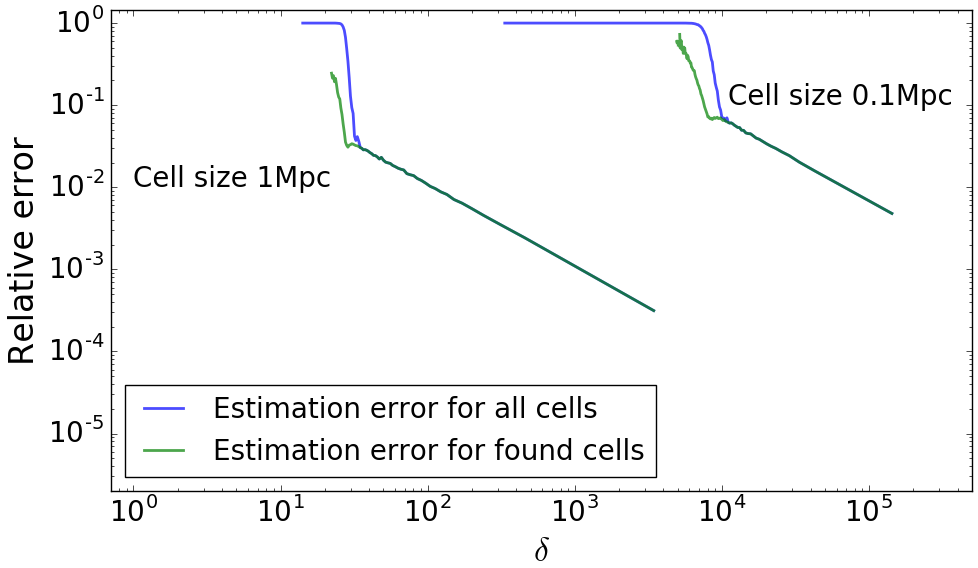


FIGURE 5.19: Relative error vs. δ of the cell

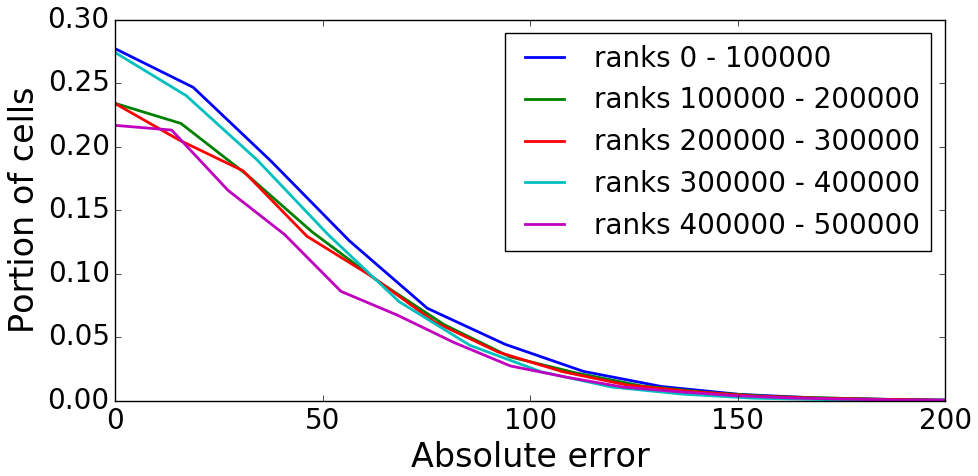
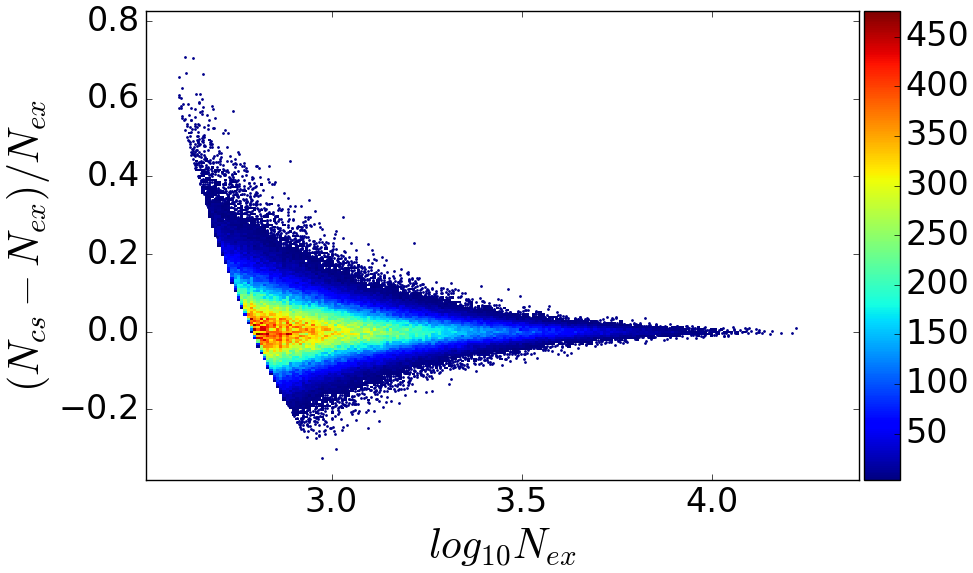
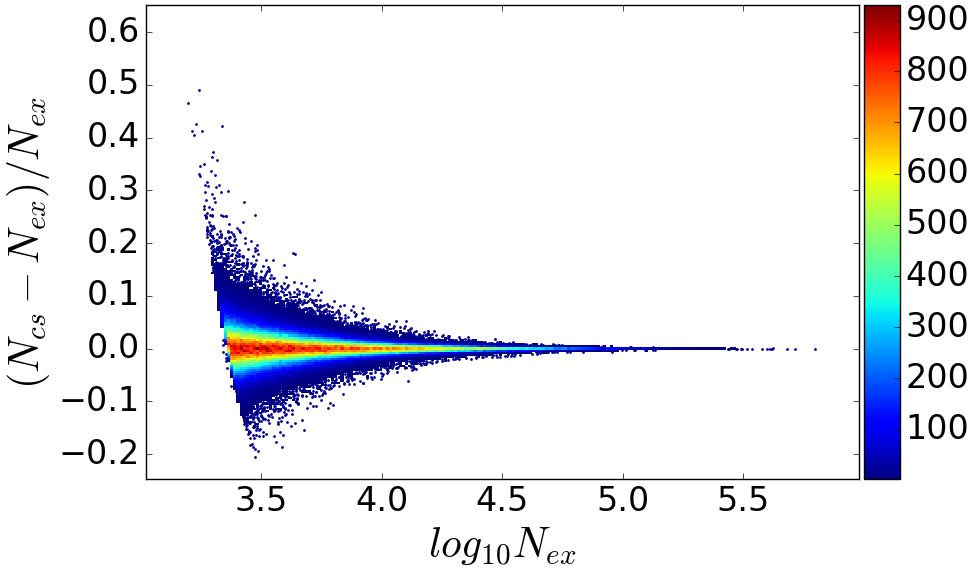


FIGURE 5.20: Distribution of absolute error for different ranks



(A) cell size = 0.1 Mpc/h



(B) cell size = 1 Mpc/h

FIGURE 5.21: Count distortion for the cell size = 0.1 Mpc/h on the top and for the cell size = 1 Mpc/h on the bottom.

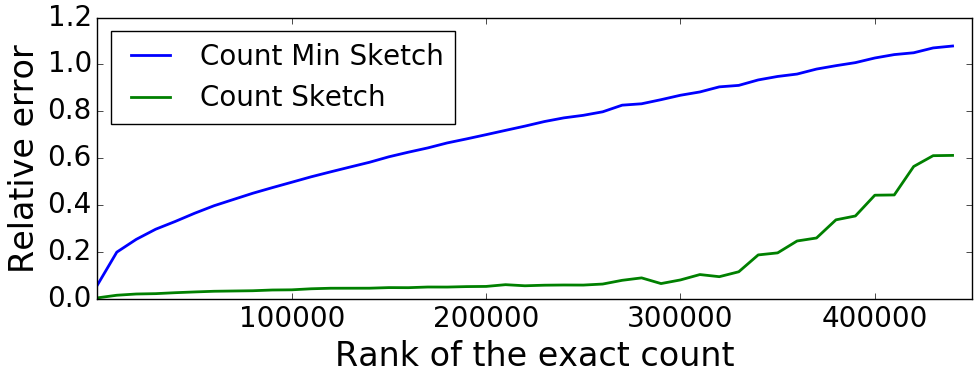


FIGURE 5.22: Relative error for the counts in the output of the Count Sketch algorithm and Count Min Sketch algorithm, cell size = 0.1 Mpc/h

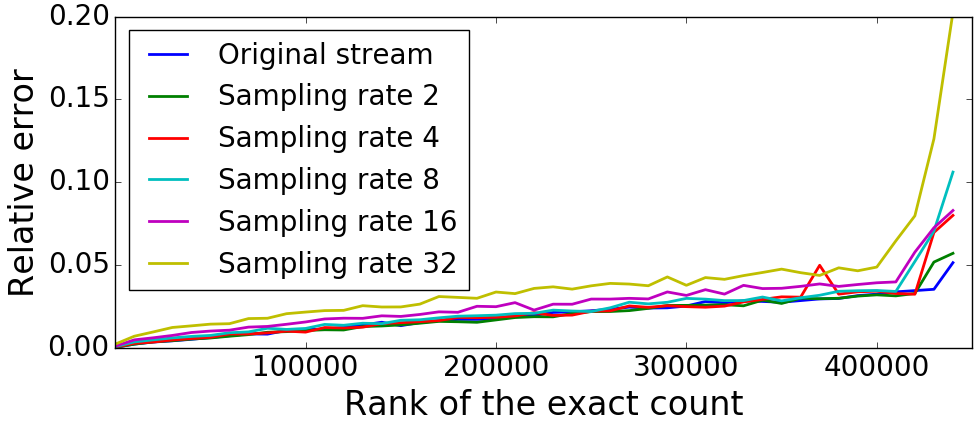


FIGURE 5.23: Relative error for the counts in the output of the Count Sketch algorithm with different sampling rates, cell size = 1 Mpc/h

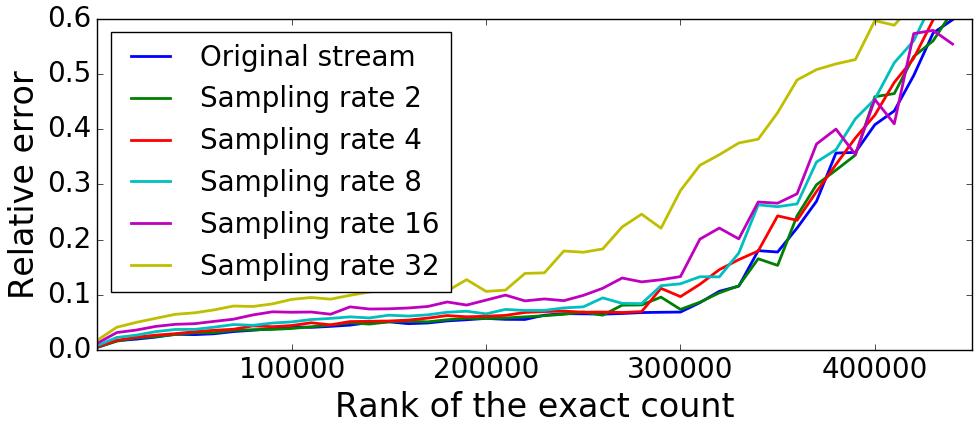


FIGURE 5.24: Relative error for the counts in the output of the Count Sketch algorithm with different sampling rates, cell size = 0.1 Mpc/h

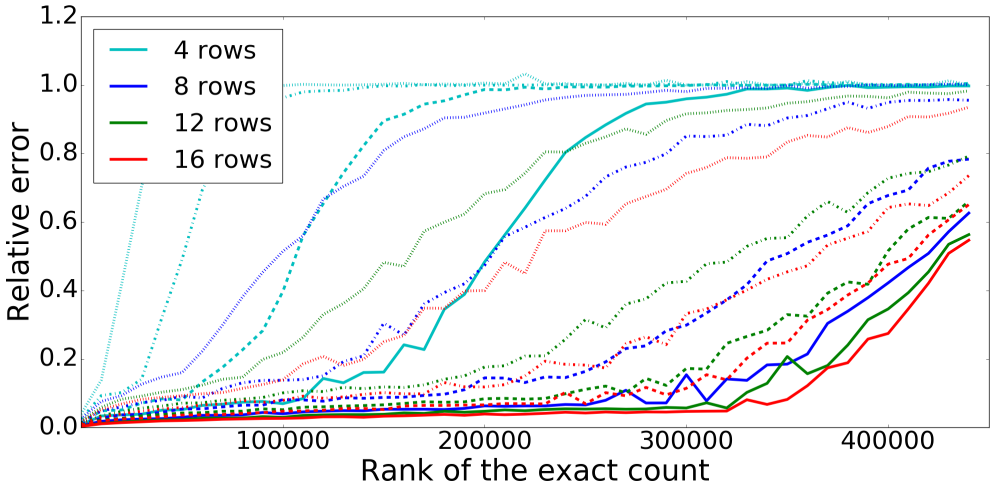


FIGURE 5.25: Relative error for the counts in the output of the Count Sketch algorithm with different internal parameters, cell size = 0.1Mpc. Color is the height of the CS table, and line type is the width of CS table: solid is $16 \cdot 10^6$, dashed is $8 \cdot 10^6$, dash-dotted is $4 \cdot 10^6$, and dotted is 10^6 columns

FIGURE 5.26: Finding halos from heavy cells exactly by running any offline in-memory algorithm on the subset of particles belonging to the top heaviest cells.

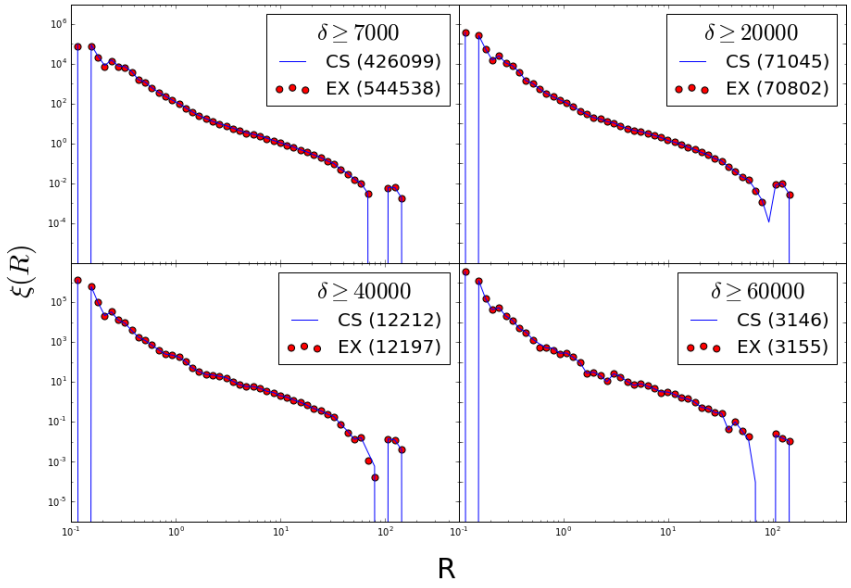


FIGURE 5.27: Comparison of the 2-point correlation functions of excursion sets determined using the exact counts and the Count Sketch results for 4 over-density levels. The numbers in parentheses indicate the number of cells that was found and used i the calculation of ξ. Clearly the results of applying the spatial statistic to the Count Sketch result is equivalent to that of the exact counts. The radius R is in the natural, co-moving units of the simulations, Mpc/h.

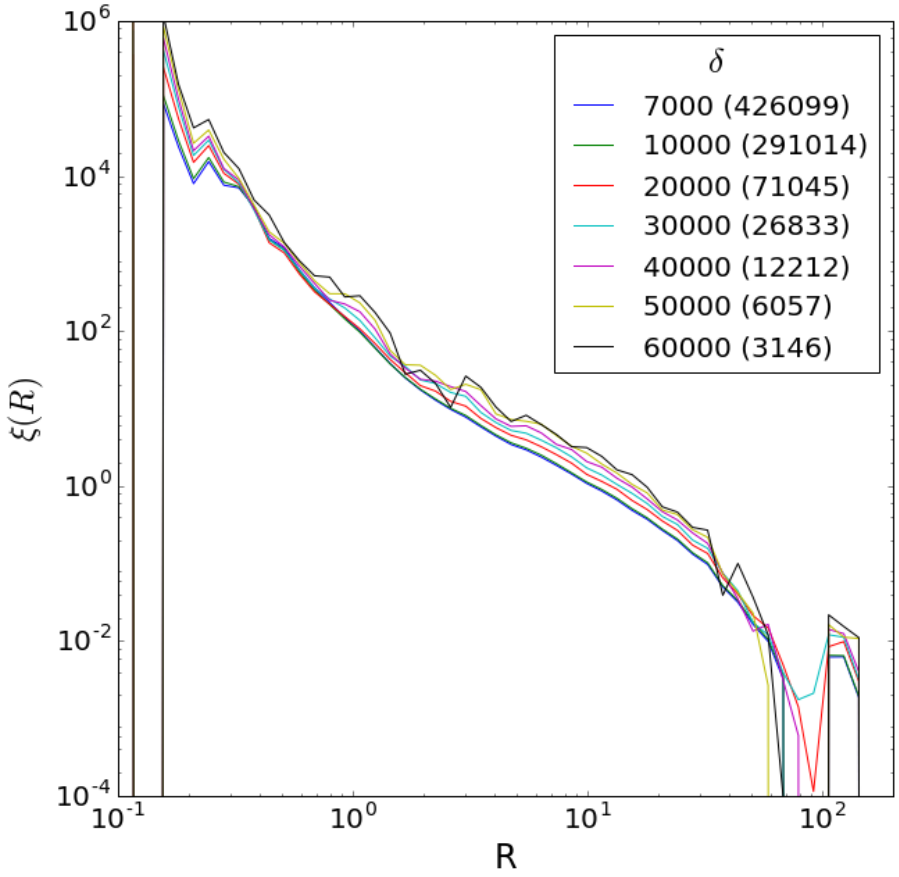


FIGURE 5.28: Two-point correlation functions of excursion sets, defined as sets of cells with a certain lower limit on the over-density. In this plot the results of the count-sketch algorithm for detecting heavy-hitters is used to determine the excursion sets. The number next to the line segments in the legend gives the over-density, the numbers in parentheses indicate the number of cells at that over-density. The radius R is in the natural, co-moving units of the simulations, Mpc/h.

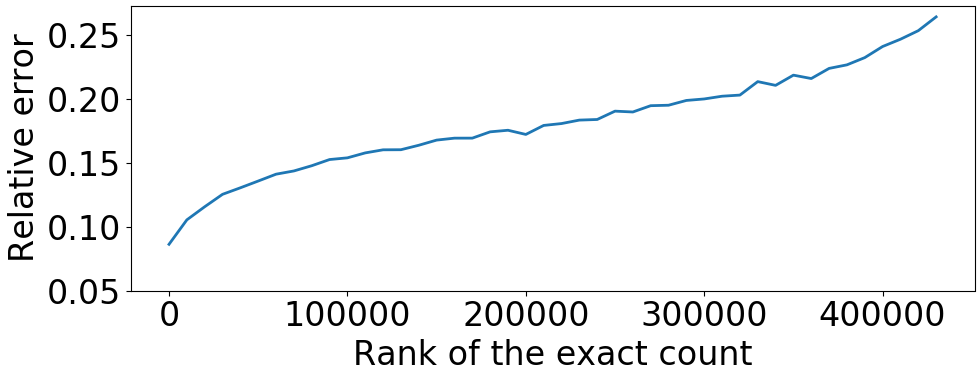


FIGURE 5.29: Relative error for the counts in output of Count Sketch algorithm for Millennium XXL dataset.

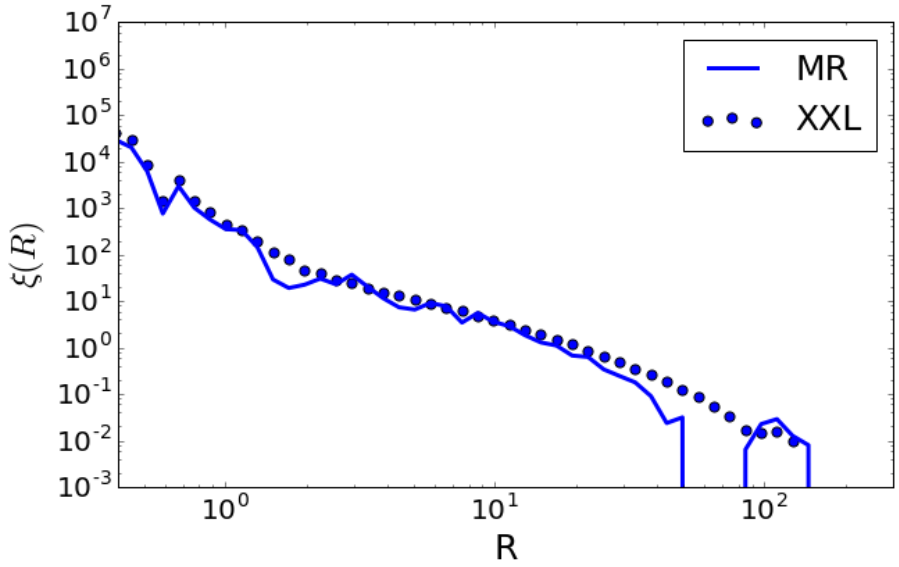


FIGURE 5.30: Comparison of CS 2-pt correlation function for excursion sets in 0.2 Mpc cells with $\delta \geq 20000$ for the XXL (dots) compared to the exact result for the Millennium run. The two results are compatible with each other, with deviations explained by discreteness effects in the much sparser Millennium result. The radius R is in the natural, co-moving units of the simulations, Mpc/h.

Bibliography

[1] Dimitris Achlioptas. "Database-friendly random projections: Johnson-Lindenstrauss with binary coins". In: *Journal of Computer and System Sciences* 66.4 (2003), pp. 671–687.

[2] Pankaj K Agarwal et al. "Mergeable summaries". In: *ACM Transactions on Database Systems (TODS)* 38.4 (2013), p. 26.

[3] Charu C Aggarwal. *Data streams: models and algorithms*. Vol. 31. Springer Science & Business Media, 2007.

[4] Rakesh Agrawal and Ramakrishnan Srikant. "Fast Algorithms for Mining Association Rules in Large Databases". In: *VLDB*. 1994, pp. 487–499.

[5] Zeyuan Allen-Zhu and Yuanzhi Li. "First efficient convergence for streaming k-pca: a global, gap-free, and near-optimal rate". In: *Foundations of Computer Science (FOCS), 2017 IEEE 58th Annual Symposium on*. IEEE. 2017, pp. 487–492.

[6] Noga Alon, Yossi Matias, and Mario Szegedy. "The space complexity of approximating the frequency moments". In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. ACM. 1996, pp. 20–29.

[7] RE Angulo et al. "Scaling relations for galaxy clusters in the Millennium-XXL simulation". In: *Monthly Notices of the Royal Astronomical Society* 426.3 (2012), pp. 2046–2062.

[8] Arvind Arasu and Gurmeet Singh Manku. "Approximate counts and quantiles over sliding windows". In: *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM. 2004, pp. 286–296.

[9] Eran Assaf et al. "Pay for a Sliding Bloom Filter and Get Counting, Distinct Elements, and Entropy for Free". In: *IEEE INFOCOM*. 2018.

[10] Khanh Do Ba et al. "Lower Bounds for Sparse Recovery". In: *CoRR* abs/1106.0365 (2011).

[11] Brian Babcock et al. "Models and Issues in Data Stream Systems". In: *PODS*. Madison, Wisconsin, 2002.

[12] Ziv Bar-Yossef et al. "An information statistics approach to data stream and communication complexity". In: *Journal of Computer and System Sciences* (2004).

[13] James M Bardeen et al. "The statistics of peaks of Gaussian random fields". In: *The Astrophysical Journal* 304 (1986), pp. 15–61.

[14] Ran Ben Basat, Roy Friedman, and Rana Shahout. "Heavy Hitters over Interval Queries". In: *arXiv:1804.10740* (2018).

[15] Ran Ben-Basat et al. "Constant Time Updates in Hierarchical Heavy Hitters". In: *ACM SIGCOMM* (2017).

[16] Ran Ben-Basat et al. "Randomized Admission Policy for Efficient Top-k and Frequency Estimation". In: *IEEE INFOCOM*. 2017.

[17] Theophilus Benson, Aditya Akella, and David A. Maltz. "Network traffic characteristics of data centers in the wild". In: *ACM IMC*. 2010.

[18] Theophilus Benson et al. "MicroTE: Fine Grained Traffic Engineering for Data Centers". In: *ACM CoNEXT*. 2011, p. 8.

[19] Radu Berinde et al. "Space-optimal heavy hitters with strong error bounds". In: *Proceedings of the Twenty-Eigth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*. 2009, pp. 157–166.

[20] Kevin S. Beyer and Raghu Ramakrishnan. "Bottom-Up Computation of Sparse and Iceberg CUBEs". In: *SIGMOD*. 1999, pp. 359–370.

[21] Arnab Bhattacharyya, Palash Dey, and David P Woodruff. "An optimal algorithm for ℓ_1-heavy hitters in insertion streams and related problems". In: *ACM PODS*. 2016.

[22] Robert S Boyer and J Strother Moore. *A fast majority vote algorithm*. SRI International. Computer Science Laboratory, 1981.

[23] Robert S. Boyer and J. Strother Moore. "MJRTY: A Fast Majority Vote Algorithm". In: *Automated Reasoning: Essays in Honor of Woody Bledsoe*. 1991, pp. 105–118.

[24] V. Braverman. "Sliding window algorithms". In: *Encyc. of Algorithms, 2004* ().

[25] Vladimir Braverman, Ran Gelles, and Rafail Ostrovsky. "How to catch l2-heavy-hitters on sliding windows". In: *Theoretical Computer Science* (2014).

[26] Vladimir Braverman and Rafail Ostrovsky. "Approximating large frequency moments with pick-and-drop sampling". In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*. Springer, 2013, pp. 42–57.

[27] Vladimir Braverman and Rafail Ostrovsky. "Generalizing the Layering Method of Indyk and Woodruff: Recursive Sketches for Frequency-Based Vectors on Streams". In: *APPROX/RANDOM*. 2013.

[28] Vladimir Braverman and Rafail Ostrovsky. "Smooth histograms for sliding windows". In: *Foundations of Computer Science, 2007. FOCS'07. 48th Annual IEEE Symposium on*. IEEE. 2007, pp. 283–293.

[29] Vladimir Braverman, Rafail Ostrovsky, and Alan Roytman. "Zero-One Laws for Sliding Windows and Universal Sketches". In: *APPROX/RANDOM*. 2015.

[30] Vladimir Braverman et al. "An Optimal Algorithm for Large Frequency Moments Using $O(n^{1-2/k})$ Bits". In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*. 2014, pp. 531–544.

[31] Vladimir Braverman et al. "Beating CountSketch for heavy hitters in insertion streams". In: *ACM STOC*. 2016.

[32] Vladimir Braverman et al. "BPTree: an ℓ_2 heavy hitters algorithm using constant memory". In: *arXiv preprint arXiv:1603.00759* (2016).

[33] Vladimir Braverman et al. "Nearly Optimal Distinct Elements and Heavy Hitters on Sliding Windows". In: *arXiv preprint arXiv:1805.00212* (2018).

[34] Vladimir Braverman et al. "Streaming Space Complexity of Nearly All Functions of One Variable on Frequency Vectors". In: *ACM PODS*. 2016.

[35] Larry Carter and Mark N. Wegman. "Universal Classes of Hash Functions". In: *J. Comput. Syst. Sci.* 18.2 (1979), pp. 143–154.

[36] Amit Chakrabarti, Graham Cormode, and Andrew McGregor. "A near-optimal algorithm for estimating the entropy of a stream". In: *ACM Transactions on Algorithms (TALG)* 6.3 (2010), p. 51.

[37] Amit Chakrabarti, Subhash Khot, and Xiaodong Sun. "Near-optimal lower bounds on the multi-party communication complexity of set disjointness". In: *IEEE CCC*. 2003.

[38] Badrish Chandramouli et al. "Data stream management systems for computational finance". In: *Computer* 43.12 (2010), pp. 45–52.

[39] Moses Charikar, Kevin Chen, and Martin Farach-Colton. "Finding frequent items in data streams". In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2002, pp. 693–703.

[40] Tianqi Chen and Carlos Guestrin. "Xgboost: A scalable tree boosting system". In: *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. ACM. 2016, pp. 785–794.

[41] P. Coles and B. Jones. "A lognormal model for the cosmological mass distribution". In: *MNRAS* 248 (Jan. 1991), pp. 1–13.

[42] Graham Cormode and Marios Hadjieleftheriou. "Finding Frequent Items in Data Streams". In: *Proc. VLDB Endow.* 1.2 (Aug. 2008), pp. 1530–1541. ISSN: 2150-8097. DOI: 10.14778/1454159.1454225. URL: http://dx.doi.org/10.14778/1454159.1454225.

[43] Graham Cormode and Marios Hadjieleftheriou. "Methods for Finding Frequent Items in Data Streams". In: *J. VLDB* (2010).

[44] Graham Cormode and Shan Muthukrishnan. "An improved data stream summary: the count-min sketch and its applications". In: *Journal of Algorithms* 55.1 (2005), pp. 58–75.

[45] Graham Cormode et al. "Holistic aggregates in a networked world: Distributed tracking of approximate quantiles". In: *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM. 2005, pp. 25–36.

[46] Mayur Datar et al. "Maintaining stream statistics over sliding windows". In: *SIAM journal on computing* 31.6 (2002), pp. 1794–1813.

[47] Marc Davis et al. "The evolution of large-scale structure in a universe dominated by cold dark matter". In: *The Astrophysical Journal* 292 (1985), pp. 371–394.

[48] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. "Frequency Estimation of Internet Packet Streams with Limited Space". In: *ESA*. 2002, pp. 348–360.

[49] David J DeWitt, Jeffrey F Naughton, and Donovan A Schneider. "Parallel sorting on a shared-nothing architecture using probabilistic splitting". In: *Parallel and distributed information systems, 1991., proceedings of the first international conference on*. IEEE. 1991, pp. 280–291.

[50] Richard M. Dudley. "The sizes of compact subsets of Hilbert space and continuity of Gaussian processes". In: *J. Functional Analysis* 1 (1967), pp. 290–330.

[51] G. Einziger, B. Fellman, and Y. Kassner. "Independent counter estimation buckets". In: *IEEE INFOCOM*. 2015.

[52] Cristian Estan, Stefan Savage, and George Varghese. "Automatically Inferring Patterns of Resource Consumption in Network Traffic". In: *ACM SIGCOMM*. 2003.

[53] Cristian Estan and George Varghese. "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice". In: *ACM Trans. Comput. Syst.* 21.3 (2003), pp. 270–313.

[54] B. L. Falck, M. C. Neyrinck, and A. S. Szalay. "ORIGAMI: Delineating Halos Using Phase-space Folds". In: *ApJ* 754, 126 (Aug. 2012), p. 126. DOI: 10.1088/0004-637X/754/2/126. arXiv: 1201.2353.

[55] Min Fang et al. "Computing Iceberg Queries Efficiently". In: *VLDB*. 1998, pp. 299–310.

[56] Shir Landau Feibish et al. "Mitigating DNS random subdomain DDoS attacks by distinct heavy hitters sketches". In: *ACM/IEEE HotWeb 2017*.

[57] David Felber and Rafail Ostrovsky. "A randomized online quantile summary in O (1/epsilon* log (1/epsilon)) words". In: *LIPIcs-Leibniz International Proceedings in Informatics*. Vol. 40. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2015.

[58] Éric Fusy and Frécéric Giroire. "Estimating the number of active flows in a data stream over a sliding window". In: *ANALCO*. 2007.

[59] Pedro Garcia-Teodoro et al. "Anomaly-Based Network Intrusion Detection: Techniques, Systems and Challenges". In: *Computers and Security* (2009).

[60] Mina Ghashami et al. "Frequent directions: Simple and deterministic matrix sketching". In: *SIAM Journal on Computing* 45.5 (2016), pp. 1762–1792.

[61] Stuart PD Gill, Alexander Knebe, and Brad K Gibson. "The evolution of substructure—I. A new identification method". In: *Monthly Notices of the Royal Astronomical Society* 351.2 (2004), pp. 399–409.

[62] Parikshit Gopalan and Jaikumar Radhakrishnan. "Finding duplicates in a data stream". In: *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2009, pp. 402–411.

[63] S. Gottlöber and G. Yepes. "Shape, Spin, and Baryon Fraction of Clusters in the MareNostrum Universe". In: *ApJ* 664 (July 2007), pp. 117–122. DOI: 10.1086/517907. eprint: astro-ph/0703164.

[64] Michael Greenwald and Sanjeev Khanna. "Space-efficient online computation of quantile summaries". In: *ACM SIGMOD Record*. Vol. 30. 2. ACM. 2001, pp. 58–66.

[65] Michael B Greenwald and Sanjeev Khanna. "Power-conserving computation of order-statistics over sensor networks". In: *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM. 2004, pp. 275–285.

[66] Michael B Greenwald and Sanjeev Khanna. "Quantiles and equi-depth histograms over streams". In: *Data Stream Management*. Springer, 2016, pp. 45–86.

[67] Uffe Haagerup. "The best constants in the Khintchine inequality". In: *Studia Math.* 70.3 (1982), pp. 231–283.

[68] Jiawei Han, Jian Pei, and Yiwen Yin. "Mining Frequent Patterns without Candidate Generation". In: *SIGMOD*. 2000, pp. 1–12.

[69] Jiawei Han et al. "Efficient Computation of Iceberg Cubes with Complex Measures". In: *SIGMOD*. 2001, pp. 1–12.

[70] David Lee Hanson and Farroll Tim Wright. "A bound on tail probabilities for quadratic forms in independent random variables". In: *The Annals of Mathematical Statistics* 42.3 (1971), pp. 1079–1083.

[71] Hazar Harmouch and Felix Naumann. "Cardinality Estimation: An Experimental Survey". In: *J. VLDB* (2017).

[72] Nicholas J. A. Harvey, Jelani Nelson, and Krzysztof Onak. "Sketching and Streaming Entropy via Approximation Theory". In: *49th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 2008, pp. 489–498.

[73] Stefan Heule, Marc Nunkesser, and Alexander Hall. "HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm". In: *ACM EDBT*. 2013.

[74] Paul Hick. *CAIDA Anonymized 2014 Internet Trace, equinix-chicago 2014-03-20 13:55 UTC, Direction B.* http://www.caida.org/data/monitors/passive-equinix-chicago.xml.

[75] Christian Hidber. "Online Association Rule Mining". In: *SIGMOD*. 1999, pp. 145–156.

[76] Zengfeng Huang, Wai Ming Tai, and Ke Yi. "Tracking the Frequency Moments at All Times". In: *arXiv preprint arXiv:1412.1763* (2014).

[77] Zengfeng Huang et al. "Sampling based algorithms for quantile computation in sensor networks". In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM. 2011, pp. 745–756.

[78] Piotr Indyk. "Stable distributions, pseudorandom generators, embeddings, and data stream computation". In: *J. ACM* 53.3 (2006), pp. 307–323.

[79] Piotr Indyk, Eric Price, and David P. Woodruff. "On the Power of Adaptivity in Sparse Recovery". In: *IEEE 52nd Annual Symposium on Foundations of Computer Science (FOCS)*. 2011, pp. 285–294.

[80] Piotr Indyk and David Woodruff. "Optimal Approximations of the Frequency Moments of Data Streams". In: *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*. STOC '05. Baltimore, MD, USA: ACM, 2005, pp. 202–

208. ISBN: 1-58113-960-8. DOI: 10.1145/1060590.1060621. URL: http://doi.acm.org/10.1145/1060590.1060621.

[81] Nikita Ivkin et al. "Scalable streaming tools for analyzing N-body simulations: Finding halos and investigating excursion sets in one pass". In: *Astronomy and computing* 23 (2018), pp. 166–179.

[82] Hossein Jowhari, Mert Saglam, and Gábor Tardos. "Tight bounds for Lp samplers, finding duplicates in streams, and related problems". In: *PODS*. 2011, pp. 49–58.

[83] Nick Kaiser. "On the spatial correlations of Abell clusters". In: *The Astrophysical Journal* 284 (1984), pp. L9–L12.

[84] Daniel Kane, Raghu Meka, and Jelani Nelson. "Almost optimal explicit Johnson-Lindenstrauss families". In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*. Springer, 2011, pp. 628–639.

[85] Zohar Karnin, Kevin Lang, and Edo Liberty. "Optimal quantile approximation in streams". In: *Foundations of Computer Science (FOCS), 2016 IEEE 57th Annual Symposium on*. IEEE. 2016, pp. 71–78.

[86] Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. "A simple algorithm for finding frequent elements in streams and bags". In: *ACM Trans. Database Syst.* 28 (2003), pp. 51–55.

[87] Hannu Karttunen et al. *Fundamental astronomy*. Springer, 2016.

[88] Atul Kant Kaushik, Emmanuel S. Pilli, and R. C. Joshi. "Network Forensic Analysis by Correlation of Attacks with Network Attributes". In: *Information and Communication Technologies*. 2010.

[89] Issha Kayo, Atsushi Taruya, and Yasushi Suto. "Probability Distribution Function of Cosmological Density Fluctuations from a Gaussian Initial Condition: Comparison of One-Point and Two-Point Lognormal Model Predictions with N-Body Simulations". In: *The Astrophysical Journal* 561.1 (2001), p. 22. URL: `http://stacks.iop.org/0004-637X/561/i=1/a=22`.

[90] Anatoly Klypin and Jon Holtzman. "Particle-Mesh code for cosmological simulations". In: *arXiv preprint astro-ph/9712217* (1997).

[91] Alexander Knebe et al. "Haloes gone MAD: the halo-finder comparison project". In: *Monthly Notices of the Royal Astronomical Society* 415.3 (2011), pp. 2293–2318.

[92] Alexander Knebe et al. "Structure finding in cosmological simulations: the state of affairs". In: *Monthly Notices of the Royal Astronomical Society* 435.2 (2013), pp. 1618–1658.

[93] Steffen R Knollmann and Alexander Knebe. "AHF: Amiga's halo finder". In: *The Astrophysical Journal Supplement Series* 182.2 (2009), p. 608.

[94] Ilan Kremer, Noam Nisan, and Dana Ron. "On randomized one-round communication complexity". In: *Computational Complexity* 8.1 (1999), pp. 21–49.

[95] Michel Ledoux and Michel Talagrand. *Probability in Banach Spaces*. Vol. 23. Springer-Verlag, 1991.

[96] Gerard Lemson et al. "Halo and galaxy formation histories from the millennium simulation: Public release of a VO-oriented and SQL-queryable database for studying the evolution of galaxies in the LambdaCDM cosmogony". In: *arXiv preprint astro-ph/0608019* (2006).

[97] Yi Li and David P Woodruff. "A tight lower bound for high frequency moment estimation with small error". In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*. Springer, 2013, pp. 623–638.

[98] Zhenjiang Li et al. "Ubiquitous data collection for mobile users in wireless sensor networks". In: *INFOCOM, 2011 Proceedings IEEE*. IEEE. 2011, pp. 2246–2254.

[99] Edo Liberty. "Simple and deterministic matrix sketching". In: *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2013, pp. 581–588.

[100] Xuemin Lin et al. "Continuously maintaining quantile summaries of the most recent n elements over a data stream". In: *Data Engineering, 2004. Proceedings. 20th International Conference on*. IEEE. 2004, pp. 362–373.

[101] Zaoxing Liu et al. "One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: ACM, 2016, pp. 101–114. ISBN: 978-1-4503-4193-6. DOI: 10.1145/2934872.2934906. URL: http://doi.acm.org/10.1145/2934872.2934906.

[102] Zaoxing Liu et al. "Streaming Algorithms for Halo Finders". In: *e-Science (e-Science), 2015 IEEE 11th International Conference on*. IEEE. 2015, pp. 342–351.

[103] Gurmeet Singh Manku and Rajeev Motwani. "Approximate Frequency Counts over Data Streams". In: *PVLDB* 5.12 (2012), p. 1699.

[104] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G Lindsay. "Approximate medians and other quantiles in one pass and with limited memory". In: *ACM SIGMOD Record*. Vol. 27. 2. ACM. 1998, pp. 426–435.

[105] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G Lindsay. "Random sampling techniques for space efficient online computation of order statistics of large datasets". In: *ACM SIGMOD Record*. Vol. 28. 2. ACM. 1999, pp. 251–262.

[106] Raghu Meka. "A PTAS for computing the supremum of Gaussian processes". In: *FOCS*. IEEE. 2012, pp. 217–222.

[107] A. Metwally, D. Agrawal, and A. El Abbadi. "Efficient computation of frequent and top-k elements in data streams". In: *ICDT*. 2005.

[108] Rui Miao et al. "SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs". In: *ACM SIGCOMM*. 2017.

[109] Jayadev Misra and David Gries. "Finding repeated elements". In: *Science of computer programming* 2.2 (1982), pp. 143–152.

[110] Morteza Monemizadeh and David P Woodruff. "1-pass relative-error lp-sampling with applications". In: *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*. SIAM. 2010, pp. 1143–1160.

[111] Masoud Moshref et al. "DREAM: Dynamic Resource Allocation for Software-defined Measurement". In: *ACM SIGCOMM*. 2014.

[112] J Ian Munro and Mike S Paterson. "Selection and sorting with limited storage". In: *Theoretical computer science* 12.3 (1980), pp. 315–323.

[113] S. Muthukrishnan. "Data Streams: Algorithms and Applications". In: *Foundations and Trends in Theoretical Computer Science* 1.2 (2005). DOI: 10.1561/0400000002. URL: http://dx.doi.org/10.1561/0400000002.

[114] Mark C Neyrinck, Nickolay Y Gnedin, and Andrew JS Hamilton. "VOBOZ: an almost-parameter-free halo-finding algorithm". In: *Monthly Notices of the Royal Astronomical Society* 356.4 (2005), pp. 1222–1232.

[115] Ran Ben Basat Gil Einziger Roy Friedman Vladimir Braverman Nikita Ivkin Zaoxing Liu. "Monitoring the Network with Interval Queries". In: *In submission*. 2018.

[116] Noam Nisan. "Pseudorandom generators for space-bounded computation". In: *Combinatorica* 12.4 (1992), pp. 449–461.

[117] CUDA Nvidia. *Programming guide*. 2010.

[118] George Nychis et al. "An Empirical Evaluation of Entropy-based Traffic Anomaly Detection". In: *ACM IMC*. 2008.

[119] "Page view statistics for Wikimedia projects". In: (2016). URL: `https://dumps.wikimedia.org/other/pagecounts-raw/`.

[120] Phillip James Edwin Peebles. *The large-scale structure of the universe*. Princeton university press, 1980.

[121] Rob Pike et al. "Interpreting the data: Parallel analysis with Sawzall". In: *Scientific Programming* 13.4 (2005), pp. 277–298.

[122] Susana Planelles and Vicent Quilis. "ASOHF: a new adaptive spherical overdensity halo finder". In: *Astronomy & Astrophysics* 519 (2010), A94.

[123] Viswanath Poosala et al. "Improved histograms for selectivity estimation of range predicates". In: *ACM Sigmod Record*. Vol. 25. 2. ACM. 1996, pp. 294–305.

[124] Douglas Potter, Joachim Stadel, and Romain Teyssier. "PKDGRAV3: beyond trillion particle cosmological simulations for the next era of galaxy surveys". In: *Computational Astrophysics and Cosmology* 4.1 (2017), p. 2.

[125] Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. "DSK: k-mer counting with very low memory usage". In: *Bioinformatics* 29.5 (2013), pp. 652–653.

[126] Ashok Savasere, Edward Omiecinski, and Shamkant B. Navathe. "An Efficient Algorithm for Mining Association Rules in Large Databases". In: *VLDB*. 1995, pp. 432–444.

[127] Stuart Schechter, Cormac Herley, and Michael Mitzenmacher. "Popularity is everything: A new approach to protecting passwords from statistical-guessing attacks". In: *Proceedings of the 5th USENIX conference on Hot topics in security*. USENIX Association. 2010, pp. 1–8.

[128] Vyas Sekar et al. "LADS: Large-scale Automated DDoS Detection System." In: *USENIX ATC*. 2006.

[129] P Griffiths Selinger et al. "Access path selection in a relational database management system". In: *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. ACM. 1979, pp. 23–34.

[130] Nisheeth Shrivastava et al. "Medians and beyond: new aggregation techniques for sensor networks". In: *Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM. 2004, pp. 239–249.

[131] Haya Shulman and Michael Waidner. "Towards Forensic Analysis of Attacks with DNSSEC". In: *IEEE SPW*. 2014.

[132] R. E. Smith et al. "Stable clustering, the halo model and non-linear cosmological power spectra". In: *MNRAS* 341 (June 2003), pp. 1311–1332. DOI: 10.1046/j.1365-8711.2003.06503.x. eprint: astro-ph/0207664.

[133] Volker Springel. "The cosmological simulation code GADGET-2". In: *Monthly notices of the royal astronomical society* 364.4 (2005), pp. 1105–1134.

[134] PM Sutter and PM Ricker. "Examining subgrid models of supermassive black holes in cosmological simulation". In: *The Astrophysical Journal* 723.2 (2010), p. 1308.

[135] Michel Talagrand. "Majorizing Measures: The Generic Chaining". In: *The Annals of Probability* 24.3 (1996).

[136] Michel Talagrand. *Upper and Lower Bounds for Stochastic Processes: Modern Methods and Classical Problems*. Vol. 60. Springer Science & Business Media, 2014.

[137] *The CAIDA nonymized Internet Traces equinix-nyc 2018-03-15, Dir. A.*

[138] "The CAIDA UCSD Anonymized Internet Traces". In: (2015). URL: `http://data.caida.org/datasets/passive-2015/`.

[139] *The CAIDA UCSD Anonymized Internet Traces 2016 - January. 21st*. URL: `http://www.caida.org/data/passive/passive_2016_dataset.xml`.

[140] Mikkel Thorup and Yin Zhang. "Tabulation based 4-universal hashing with applications to second moment estimation". In: *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, New Orleans, Louisiana, USA, January 11-14, 2004*. 2004, pp. 615–624. URL: `http://dl.acm.org/citation.cfm?id=982792.982884`.

[141] Mikkel Thorup and Yin Zhang. "Tabulation-based 5-independent hashing with applications to linear probing and second moment estimation". In: *SIAM Journal on Computing* 41.2 (2012), pp. 293–331.

[142] Hannu Toivonen. "Sampling Large Databases for Association Rules". In: *VLDB*. 1996, pp. 134–145.

[143] Lu Wang et al. "Quantiles over data streams: an experimental study". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM. 2013, pp. 737–748.

[144] Martin White, Lars Hernquist, and Volker Springel. "The halo model and numerical simulations". In: *The Astrophysical Journal Letters* 550.2 (2001), p. L129.

[145] Li Yang et al. "CASE: Cache-assisted Stretchable Estimator for High Speed Per-flow Measurement". In: *IEEE INFOCOM*. 2016.

[146] Sen Yang, Bill Lin, and Jun Xu. "Safe Randomized Load-Balanced Switching By Diffusing Extra Loads". In: *ACM Meas. Anal. Comput. Syst., 2007* ().

[147] Ke Yi and Qin Zhang. "Optimal tracking of distributed heavy hitters and quantiles". In: *Algorithmica* 65.1 (2013), pp. 206–223.

Biography

Nikita Ivkin was born in Moscow, Russia. He obtained a B.S. and M.S. degrees in Applied Math from Moscow Institute of Physics and Technology in 2011 and 2013 correspondingly. In 2016 he earned M.S.E. degree from Johns Hopkins University.