Estimating Frequencies and Finding Heavy Hitters

Jonas Nicolai Hovmand, 2011 3884

Morten Houmøller Nygaard, 2011 4582

Master's Thesis, Computer Science

June 2016

Main Supervisor: Gerth Stølting Brodal

Project Supervisor: Kasper Green Larsen

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

In recent years, vast amounts of data has called for new models where it can be efficiently handled. An example of such a model is the data stream model, where data arrives in a continuous stream and algorithms are characterized by having fast processing times without storing the data explicitly. The continuous stream can be thought of as a continuous sequence of elements which leads to the interesting problems of estimating the respective frequencies of each element and the problem of finding the most frequent elements (heavy hitters).

In this thesis we analyze several algorithms solving these two problems in the *Strict Turnstile Model*. For the problem of estimating frequencies we analyze the Count-Min Sketch and Count-Median Sketch and for the problem of finding heavy hitters we analyze solutions obtained from hierarchical structures using the sketches as black boxes. In addition to the analyses we look at the lower bounds of the solutions and show that only some of the algorithms are optimal in space usage, while none are optimal in update time.

For both problems we experiment with different variations of the analyzed algorithms and compare the results against each other. The most prominent findings of the experiments are that all solutions are able to handle tens or even hundreds of thousands of updates per second, while being able to provide good approximations, using only small space. Furthermore, we find that a new variation of the hierarchical data structure with constant-sized Count-Min Sketches is the fastest and most desirable solution, since it does not suffer from the same disadvantages as the other solutions.

"As human beings, we perceive each instant of our life through an array of sensory observations (visual, aural, nervous, etc). However, over the course of our life, we manage to abstract and store only part of the observations, and function adequately even if we can not recall every detail of each instant of our lives. We are data stream processing machines."

— Shan Muthukrishnan [28]

Acknowledgments

We wish to express out sincere gratitude to our project supervisor Kasper Green Larsen. He always took his time to answer our questions and he was a huge help in discussing problems, solutions, and literature. Without his guidance we would have had a difficult time, doing the work presented in this thesis.

A big thanks should also be addressed to Maria Arup, Kristian Frost, and Haakon Nergaard for providing valuable feedback on the thesis.

Last but not least, we would like to thank Kasper Sacharias Roos Eenberg for meticulously going through the thesis, questioning every second sentence, improving the quality while practicing his multi-coloring highlight/painting skills.

Jonas Nicolai Hovmand
Morten Houmøller Nygaard
Aarhus, Tuesday 14th June, 2016

Contents

Chapter 1

Introduction

Once upon a time, in the darkest corner of a basement, a small router needed to keep count of its most frequent visitors. The owner of the small router was running a fancy website, which at points were under more load than the servers could manage. He needed the small router to determine if and when someone was trying to do a Denial-of-Service attack (DoS attack) on his website. The small router handled millions of visits every minute, but it did not have enough memory to keep count of even a small fraction of the visitors. However, the small router's owner did not give in. He had heard about algorithms that were able to give good approximations of the most frequent visitors while only using a tiny bit of memory. Such algorithms were said to solve the approximate heavy hitters problem. Hence, the owner updated the software of the small router with an approximate heavy hitters algorithm in order to find the most frequent visitors. With the help of the new approximate heavy hitters algorithm, the small router was now able to figure out who was generating the most traffic. Based on the findings of the heavy hitters query from the small router, the owner of the fancy website could now take measures to stop visitors from performing DoS attacks that overloaded the servers. The router, its owner, and the fancy website lived happily ever after.

In a sequence of elements, heavy hitters are the elements that occur most frequently, for some definition of frequent. Consider a sequence of 100 elements and a frequency fraction defined as $1/20$ of the occurrences, then the heavy hitters are the elements with 5 or more occurrences in the sequence.

Generally, the heavy hitters problem is about counting numbers, but as the fairy tale above implies, we can count other things than numbers. In fact, we can count many different things by letting the elements represent what we want to count. It could be numbers, but it could also be search queries, IP addresses, database updates etc. As noted above, we also need to define what it means for an element to be frequent. The above example with elements in a sequence defined it as a fraction of the number of elements in the sequence. The fairy tale on the other hand, defined it as a fraction of the total traffic volume.

In an attempt to study the heavy hitters problem, we will also study the problem of determining the frequencies of elements. The heavy hitters problem is about knowing

which elements are frequent, but how do we determine the frequency of an element? The problem of estimating frequencies of elements is called the frequency estimation problem, and is used as a building block for a range of other problems including many heavy hitters solutions.

Generally, frequency estimation solutions cannot be kept exactly for problems such as the one presented in the fairy tale due to the amount of memory required to do so. Consequently, they only provide approximations.

Consider a sequence of 100 elements with 20 unique elements. Now let 7 children with 10 fingers each, count the occurrences of the unique elements. Using their fingers only, will they be able to look through the sequence once and tell us how many occurrences each of the unique elements had? The answer is probably no, but using a frequency estimation algorithm, one would be able to provide an answer, not too far from the actual frequencies.

Both problems are important in literature and has been studied extensively for many years. In this thesis we seek to study the theoretical upper bounds for solutions to both problems. Furthermore, we seek to compare the upper bounds with the theoretical lower bounds in order to say if the solutions are optimal. Finally, we seek to study the practical performance of the solutions, in order to determine which solutions are the most applicable ones in practice.

1.1 Background and related work

Over the last decades, there has been an extensive increase in generated data and more recently it has lead to new terms such as "Big Data" and "Data Stream". These terms heavily influence how we think, build, and manage many practical as well as theoretical problems of Computer Science. While the former term is a broad notion for a lot of problems handling extensive amounts of data, the latter is a specific form, where huge amounts of data is seen as a stream or a sequence of data arriving at high rate.

For streams with data arriving at high rate, algorithms are needed which use as little processing time as possible, in order to analyze and provide answers in real time. Furthermore, such algorithms should only use a sublinear – preferably polylogarithmic – amount of space in the whole data set, in order for it to be kept on machines such as embedded environments, routers, servers and computers. Such structures are denoted synopsis data structures by Gibbons and Matias [16] and fits the description of a lot of algorithms and data structures supporting the data stream model [28].

Two problems which are a part of the data stream model are the problem of estimating frequencies of elements and the problem of finding the most frequent elements i.e. the heavy hitters problem, where the first problem usually provides a solution to the latter.

Both problems have been studied extensively in literature, and can be traced all the way back to the 1980's [3, 15], where an algorithm for finding the majority element was proposed. A generalization of this problem, where all frequent elements for some definition of frequent must be found, was also proposed in the same period [26].

Building on the same ideas, the actual bounds of these algorithms were settled by Demaine et al. [13], Karp et al. [20]. In the same period of time, other solutions solving the same problem were suggested by Manku and Motwani [23], Metwally et al. [24] and common for all were the fact that they had very similar behavior and bounds.

Common for all solutions above, was also that counters were kept for elements in order to determine the frequencies and find the heavy hitters. Moreover, all algorithms function in such a way that only positive increments of the estimates is possible, making them all fail in a setting where increments and decrements of frequencies should be possible. Algorithms having such constraints are said to support the *Turnstile Model*, which is the data stream model, for which we wish to study solutions and provide experiments in.

Usually, other algorithms in the form of sketches are needed, in such cases. Charikar et al. [5] suggested the Count-Median Sketch which solves both the frequency estimation and the heavy hitters problem with estimation error guarantees according to the L_2-norm of the data. A few years later, Cormode and Muthukrishnan [9] suggested a similar yet different sketch, the Count-Min Sketch, which achieved solutions to the same problems, but with estimation errors according to the L_1-norm of the data.

Following the discovery of the Count-Min Sketch, a general misconception was made in literature [9], that the Count-Min Sketch should be better in theory and practice than the Count-Median Sketch, since the bounds of the two sketches differed in their original state. Later, empirical studies of the sketches [7] suggested that they were in fact much more comparable, if the bounds of the Count-Median Sketch were loosened.

Using sketches, the easiest solutions to the heavy hitters problem is to query all elements in the data set. Such a query will be slow whenever there exists a lot of elements as is usually the case in the data stream model. Consequently, different solutions structuring the data over a tree of sketches have been proposed in order to improve the query time [7, 9, 19]. Common for such solutions is that the improved query time is exchanged for a $\log m$ factor for both space and update time, where m is the amount of unique elements.

Recently a new algorithm proposed by Larsen et al. [22], observed that such a tree structure could be kept, where each level of the tree only maintained a constant sized sketch. This removes the $\log m$ factor of both the space usage and update time, while the query in general is improved in expectation. As far as we know, this new result has not been experimented with before, which makes it of great interest to see how this solution performs compared to the earlier solutions.

Comparing the known algorithms in practice is a good way to determine which algorithms performs the best. Still, algorithms could exists which were much better, if the upper bound of the best known algorithms do not match the lower bounds. For the approximate heavy hitters problem, Jowhari et al. [18] showed a space lower bound. Using this result, we will derive another lower bound for algorithms estimating frequencies in the *Turnstile Model*. The update time of sketches supporting point queries and approximate L_1 heavy hitters queries was studied by Larsen et al. [21], where it was found that no existing solution is tight for any of the problems. No lower bounds of the query time on each of the problems were found in the literature, but for the heavy hitters problem a

natural lower bound comes from the amount of heavy hitters present in the data.

To summarize, this thesis seeks to investigate and study different solutions to the frequency estimation and heavy hitters problems in the *Strict Turnstile Model* by analyzing their theoretical bounds and experimenting with the solutions.

The theoretical presentation of the solutions will give an overview of the solutions and a description of how they are constructed and compared. The experiments will show how the solutions perform in practice by comparing them in relation to each other and in relation to the theoretical bounds.

We will further look at lower bounds for the problems, which enable us to look at how close the theoretical bounds of the solutions are from the optimal bounds, and hence stating if those solutions experimented with, could be optimized further.

1.2 Overview

This thesis is divided in a theoretical part and an practical part. Chapters 3, 4, and 5 are theoretical followed by Chapter 6 with experiments.

In Chapter 2, a subset of necessary theoretical knowledge is presented. In Chapter 3, we present the frequency estimation problem and algorithms that solve it. Then in Chapter 4, we present the heavy hitters problem and how we can use frequency estimation to form approximate heavy hitters solutions. In Chapter 5, we study the lower bounds for the two problems and compare it with the bounds of the presented algorithms.

In Chapter 6 we experiment with both frequency estimation and heavy hitters algorithms. The experiments are compared with the theoretical bounds, in relation to each other, and with related work.

Finally, we present our conclusions in Chapter 7.

Chapter 2

Preliminaries

Before going into depth with the subject of this thesis, some frequently used notation and techniques will need to be explained. In the first section, we will explain some frequently used notation. Next, we will describe and show some probabilistic tools used to analyze randomized algorithms, followed by a short introduction to some special families of hash functions. Finally, we will introduce the data stream models on which we do our experiments and the word-RAM model, which is the model our analyses use.

2.1 Notation

To be able to write in a compact and clear way, we have chosen a few different notations which will be used throughout the thesis.

Whenever we refer to the log, we implicitly mean the $\log_2$, except when the base of the logarithm is explicitly stated.

We use two short-hand notations for sets, which are defined as follows:

$$[n]_0 = \{0, 1, \ldots, n-2, n-1\}$$
$$[n]_1 = \{1, 2, \ldots, n-1, n\}$$

2.2 Probabilistic Intuition

In this section we will introduce probabilistic lemmas that are useful when analyzing randomized algorithms.

Recall that the expected value of a real valued discrete random variable X taking values from a set S, is

$$\mathbb{E}[X] = \sum_{x \in S} \mathbb{P}[X = x]\, x$$

and the variance of X is

$$\mathrm{Var}(X) = \mathbb{E}\left[X^2\right] - \mathbb{E}[X]^2 = \sigma^2 \quad .$$

2.2.1 Linearity of Expectation

One of the most central lemmas of probability theory is Linearity of Expectation. Linearity of Expectation says that the expected value of a sum of random variables is equal to the sum of the individual expectations.

Formally, it is defined as the following lemma.

Lemma 1 (Linearity of Expectation)**.** *Let $X_1, \ldots, X_n$ be real valued discrete random variables and let $X = \sum_i X_i$ be the sum of the random variables. Then*

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_i X_i\right] = \sum_i \mathbb{E}[X_i]$$

and there is no requirement of $X_1, \ldots, X_n$ to be independent.

Linearity of Expectation is extremely useful in randomized algorithm analysis and the fact that the variables do not need to be independent makes it even more useful and widely applicable.

2.2.2 Union Bound

Another basic lemma is the Union Bound, which gives an upper bound on the probability of certain events. This is done by stating that the probability of one or more events to occur is no more than the probability of all of them to occur.

Lemma 2 (Union Bound)**.** *Let $E_1, \ldots, E_n$ be events. Then*

$$\mathbb{P}\left[\bigcup_i E_i\right] \leq \sum_i \mathbb{P}[E_i]$$

and there is no requirement of $E_1, \ldots, E_n$ to be independent.

Usually the Union Bound is used with *bad* events, seeking to bound the probability of some undesired events to happen. Hence, if each *bad* event has a small probability of occurring, then the sum of them will still be small, which is the intuition behind using the Union Bound.

2.2.3 Markov's Inequality

Markov's Inequality bounds how often an event can happen based on the expected value. This is done by bounding the frequency of events being off by a factor of t from the expected.

Lemma 3 (Markov's Inequality)**.** *Let X be a real valued non-negative discrete random variable, it then holds for any $t > 1$ that $\mathbb{P}[X > t\,\mathbb{E}[X]] < \frac{1}{t}$.*

The inequality is especially useful in situations where one wishes to bound the probability of a deviation from the expected value of a random variable, with more than a factor t.

2.2.4 Chebyshev's Inequality

Chebyshev's Inequality states that in any probability distribution most values are close to the mean. It states that no more than ϵ^{-2} of the distribution's values can be more than ϵ standard deviations (σ) away from the mean.

It is more or less equivalent to Markov's Inequality, except that Chebyshev's Inequality gives a 2-sided bound.

Lemma 4 (Chebyshev's Inequality)**.** *Let $X_1, \ldots, X_n$ be random variables, the sum of all variables be $X = \sum_i X_i$, the mean of the sum of all variables be $\mu = \mathbb{E}[X]$, and the variance of the sum of all variables be $\sigma^2 = Var(X)$. Then*

$$\mathbb{P}[|X - \mu| \geq t\sigma] \leq \frac{1}{t^2}$$

for any $t > 0$.

In general, a lot of distributions can be bounded tighter than the bound from Chebyshev's Inequality, but the advantage is that it works for all distributions.

2.2.5 Chernoff Bound

The Chernoff Bound, is able to give a tighter bound than the bounds provided by Chebyshev's Inequality and Markov's Inequality. The difference from those two is the requirement of all variables to be independent of each other.

Lemma 5 (Convenient Multiplicative Chernoff Bounds)**.** *Let $X_1, \ldots, X_n$ be independent random indicator variables, the sum of the variables $X = \sum_i X_i$, and the mean of the sum of the variables $\mu = \mathbb{E}[X]$. Then the convenient multiplicative form of the Chernoff Bound states that for any $0 < \delta < 1$*

$$\mathbb{P}[X \geq (1+\delta)\mu] \leq e^{-\frac{\delta^2\mu}{3}}$$
$$\mathbb{P}[X \leq (1-\delta)\mu] \leq e^{-\frac{\delta^2\mu}{2}}$$

and for any $\delta \geq 1$

$$\mathbb{P}[X \geq (1+\delta)\mu] \leq e^{-\frac{\delta\mu}{3}}$$

The above versions of the Chernoff Bound is quite loose, but often more convenient to use, as they are simpler and hence easier to work with. To be able to give a tighter bound we also use the multiplicative Chernoff Bound in its original form, as shown below.

Lemma 6 (Multiplicative Chernoff Bound)**.** *Let $X_1, \ldots, X_n$ be independent random indicator variables, the sum of the variables $X = \sum_i X_i$, and the mean of the sum of the variables $\mu = \mathbb{E}[X]$. Then the multiplicative form of the Chernoff Bound states that for $\delta > 0$*

$$\mathbb{P}[X > (1+\delta)\mu] < \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^\mu$$

2.3 Hash Functions

In this section we will introduce and show some of the qualities of the different families of hash functions used for randomized algorithms.

Generally, a hash function h is any function that can map data of arbitrary size into data of a fixed sized. In situations where a hash function maps data from a larger space into a much smaller space, they introduce a probability for two elements in the data to collide with each other, due to the Pigeonhole Principle. In such cases, it is often convenient to bound the probability of a collision, since this is an unwanted but inevitable feature of a hash function mapping to a smaller space.

There exists a lot of different hash functions and families of hash functions. Some focus on being reliable and secure, which make them applicable in areas such as cryptography, others merely focus on the guarantees in respect to collisions and are usually much simpler than those used in cryptography.

Next we will present a few different families of hash functions. Formally we want to map a universe U of arbitrary size into m bins i.e. perform a mapping $h : U \to [m]_0$.

2.3.1 c-universal

A c-universal family of hash functions are hash functions h_i, which are chosen uniformly at random from the family of hash functions $H = \{h_i : U \to [m]_0\}$, upholding

$$\forall x, y \in U, x \neq y : \mathop{\mathbb{P}}_{h_i \in H}[h_i(x) = h_i(y)] \leq \frac{c}{m}$$

That is, when choosing h_i uniformly at random from H, one is guaranteed that when applied to the hash function, the probability of a collision between two distinct elements in the universe U is no more than $\frac{c}{m}$, for some constant c.

In practice, such families of hash functions are easy to generate and can be used efficiently. In particular, one widely used family in both theory and practice is the family described by Carter and Wegman [4], defined as

$$H = \{h(x) = (((ax + b) \mod p) \mod m)\}$$

for $a \in [p-1]_1$, $b \in [p]_0$, and where $p \geq m$ is a prime.

This family of hash functions can be shown to be 1-universal i.e. collisions between two distinct elements can be shown to only happen with probability m^{-1}, when choosing a and b uniformly at random.

Moreover, storing such a hash function only requires constant amount of space, since the values of a and b can be stored in two words [27].

In practice a faster family of hash functions is the multiply-add-shift family [14], which is still 1-universal, but for which no modulo operations are needed. Assuming m is a power of two and γ is the bits in a machine word, choose a to be an odd positive integer for which it hold that $a < 2^\gamma$ and b to be a non-negative integer with $b < 2^{\gamma - M}$

where $M = \lceil \log m \rceil$ is the amount of bits used to represent m, the amount of bins. The family is defined as

$$H = \{h(x) = (((ax + b) \mod 2^{\gamma}) \div 2^{\gamma - M})\}$$

and is 1-universal. It can be implemented using only simple operations such as multiplication, addition, shifting, and subtraction, which in a C-like language is written as

```
return (a*x+b) >> (γ-M);
```

where `x` is the element that should be hashed to a smaller universe.

A multiply-add-shift implementation will be extremely fast, since any use of such a hash function will generally only require simple operations.

2.3.2 k-wise Independence

A k-wise independent family of hash functions are hash functions h_i, which are chosen uniformly at random from the family of hash functions $H = \{h_i : U \to [m]_0\}$, having

$$\mathop{\mathbb{P}}_{h_i \in H}[h_i(x_1) = a_1 \wedge h_i(x_2) = a_2 \wedge \ldots \wedge h_i(x_k) = a_k] \leq \frac{1}{m^k}$$

for any $a_1, \ldots, a_k \in [m]_0$, $x_1, \ldots, x_k \in U$ and $x_j \neq x_l : j, l \in [k]_1$.

In other words, when choosing h_i uniformly at random from H, one is guaranteed that the probability of hashing k distinct elements in the universe U to k specific values in $[m]_0$ is at most m^{-k}. That is, the probability of seeing a specific permutation $a_1, \ldots, a_k$ when hashing k elements.

The family for $k = 2$ is frequently used and the in following we will describe it in more detail.

Pair-wise Independence

Saying that a hash function is c-universal is a weaker guarantee than being k-wise independent. In fact it follows straight from the definitions that a pair-wise (2-wise) independent family of hash functions is also 1-universal, referred to as having strong universality, since

$$\mathop{\mathbb{P}}_{h_i \in H}[h_i(x_1) = a_1 \wedge h_i(x_2) = a_2] \leq \frac{1}{m^2}$$

for which choosing $a_1 = a_2$ can be analyzed using the Union Bound over all values of a_j

$$\begin{aligned}
\mathop{\mathbb{P}}_{h_i \in H}[h_i(x_1) = h_i(x_2)] &= \bigcup_{j \in [m]_0} \mathop{\mathbb{P}}_{h_i \in H}\left[h_i(x_1) = a_j \wedge h_i(x_2) = a_j\right] \\
&\leq \sum_{j=0}^{m-1} \mathop{\mathbb{P}}_{h_i \in H}\left[h_i(x_1) = a_j \wedge h_i(x_2) = a_j\right] \qquad \text{(Union Bound)}
\end{aligned}$$

$$\leq \sum_{j=0}^{m-1} \frac{1}{m^2}$$
$$= \frac{1}{m}$$

Hence, we have shown that a hash function h_i drawn uniformly at random from a pairwise independent family, is also 1-universal.

2.4 Computational Models

Computational models are a way of describing under which conditions a certain analysis is performed. They can define how the algorithms are allowed to interact with the data, or define a set of operations which are available for the algorithm to use.

This thesis uses three data stream models, which define how the input for the algorithms is shaped, and the word-RAM model which is used when analyzing the algorithms.

2.4.1 Data Stream Models

The data stream models, are models for which the algorithms must handle an input stream, i.e. a sequence of inputs $i_1, i_2, \ldots$ arriving sequentially. The input stream is as such an infinite sequence of inputs, but can also be viewed as a finite sequence of n inputs [28].

The input stream is generally not easy to reproduce, since the data arrives and is then forgotten. Hence, several passes over the data becomes difficult, since the data in that case must be stored. Instead, algorithms solving problems in the data stream models using single passes are preferable, since the data of the whole stream is not necessarily required to be stored.

In this thesis we will focus on two specific data stream models, namely the *Cash Register Model* and the *Turnstile Model*. The difference between the two models lies in the perception of what the input stream represents. Thinking of the inputs as tuples, $i_t = (s_j, c_t)$, containing an element, s_j, and a value, c_t, for $j \in [m]_1$, $t \in [n]_1$, both models want to sum the values of the updates, $v_j = v_j + c_t$.

The *Cash Register Model* constraints the value to be positive, $c_t \geq 0$. Handling inputs in the *Cash Register Model* then becomes much like using a cash register, summing multiple occurrences of element values to a counter over time.

The *Turnstile Model* is less strictly defined and lets c_t be both positive and negative. Handling inputs then becomes much the same job as the job of a turnstile on a busy train station, that is, keeping tap of the people entering and leaving the station.

There is a special case of the *Turnstile Model*, called the *Strict Turnstile Model*, for which the sum of the element counts are never allowed to be negative, $v_j \geq 0$ for all elements j. In the general *Turnstile Model* the sums are allowed to be negative, and compared to the *Cash Register Model*, where the update values are only allowed to be positive, the *Strict*

Turnstile Model allows the update values, c_j, to be negative, as long as the sum of the values for each element is not negative.

The different models have different applications in practice, but the *Turnstile Model* can be thought of as more general than the *Strict Turnstile Model* which again is more general than the *Cash Register Model*. As such one wishes to solve problems in the *Turnstile Model*, since they are then implicitly solved in the other models, but in practice it is often more viable to solve it in a weaker model to provide better bounds and faster algorithms for specific applications.

The important metrics of any algorithm in the data stream model, is the processing time per input item, the space used to store the data structure, and the time needed to compute the output function of the algorithm.

The general goal of algorithms in the data stream model [28] is to have input processing time, storage and computing time to be linearly – preferably poly-log – bounded by the number of unique elements, m, or the input size, n.

2.4.2 Word-RAM Model

The word-RAM model of computation is based on the Random Access Machine (RAM) which is an abstract machine. The RAM defines a set of simple operations which take one unit of time per bit. The word-RAM model allows for operations on words of γ bits in one unit of time.

The constant time operations include addition, subtraction, multiplication, comparisons, and jumps. Loops and subroutines are composed of multiple simple operations, and these need to be evaluated in composition with the rest. As the model is machine independent, it does not consider any cache or other differences in slow or fast memory lookups, and it offers an infinite memory. It gives equal status for all memory reads and writes, resulting in memory accesses in one unit of time.

We use the model when analyzing algorithms throughout this thesis. For algorithms we want to count the number of operations, and using this simple model, we are able to abstract the machine away and compare the different algorithms.

Chapter 3

Frequency Estimation

In this section, we will introduce and formally describe the problem of keeping count – frequencies – of elements in the data stream models. We will look at how the problem can be solved both exactly and approximately and will outline both deterministic and randomized solutions.

A difference between the deterministic and randomized solutions is that the deterministic solutions only work in the *Cash Register Model*, where only increments are allowed. Whereas some of the randomized solutions work in the *Turnstile Model*, which allows for increments and decrements.

Randomized structures are very flexible and we will analyze a special kind of these, called *sketches*. In common for all of the sketches we present, is that they work in the *Turnstile Model* and can be used as a black box for a wide range of problems [11] such as monitoring of networks and databases, estimating frequency moments, and more. In practice it could keep track of purchases for a store, or the frequency of traded stock shares of a given day, and an innumerable list of other counting tasks.

3.1 The Problem

The problem of finding frequencies of elements, also referred to as the *count tracking* problem [11], is as such not a hard problem to grasp. The two primary functions that every solution for the problem must have are `Update`(s_i, c_t) and `Query`(s_i). At time $t \in [n]_1$ the input tuple (s_i, c_t) is passed to the `Update` function, updating the count of the given element s_i with the value c_t, and the `Query` functions returns the frequency of the given element.

More formally, the problem can be described as: Given a set of elements $S = \{s_1, \ldots, s_m\}$ count the values of all elements s_i for $i \in [m]_1$, where updates come from a stream of n elements, such that at any time $t \in [n]_1$ the current count of an element can be obtained. Let $C_{j,t}$ be a count, indicating with what an element s_j should be updated with the value

c_t at time t, formally defined as

$$C_{j,t} = \begin{cases} c_t & \text{if } s_j = s_i \\ 0 & \text{o.w.} \end{cases} \qquad \text{for } \texttt{Update}_t(s_i, c_t)$$

for $j \in [m]_1$. A query at time t can then be defined as the function $f_t(s_j) = \sum_{k=1}^{t} C_{j,k}$.

A simple algorithm to keep count of $C_{j,t}$ will be to hold a counter for each of the m different elements. Updates could then be carried out by updating the counter for a given element and queries could be implemented by returning the count for the queried element. Such a solution would solve the count tracking problem exactly and would be trivial to implement in practice.

In the data stream model, and common for many streaming algorithms, the values of m and n are usually factors larger than the available main memory and hence, the trivial algorithm from above would have to store its counters in the much slower external memory, since $O(m)$ counters are stored. In such cases the slowdown from the need of external memory is so significant that the algorithm will not be practical.

It is quite easy to show, and somewhat intuitive, that any solution solving the count tracking problem exactly, has the same weakness as the trivial algorithm above, namely a space usage depending on the number of elements i.e. $\Omega(m)$ space is needed. The count tracking problem can be reduced to the simpler membership problem. Here, one is to determine if an element y exists in the set S of m elements. To answer this question exactly, the whole set S using $\Omega(m)$ words must be used as a consequence of the Pigeonhole Principle.

Even though the above states that no exact solution is possible without using linear space in the amount of elements, several solutions still exists for a relaxed version of the problem, where counts are allowed to differ up to some approximation factor, ϵ. Such approximation solutions only use sublinear space in m or n, making the solutions much more interesting in the streaming environment and in general as a black box.

Data structures enabling sublinear space are often referred to as synopsis data structures [16].

Definition 1 (Synopsis data structure)**.** *A synopsis data structure is generally denoted as a data structure that is substantially smaller than its base data set and having the following advantages over its non-synopsis equivalents:*

- *Fast Processing: A synopsis data structure has a space usage so that the whole data structure can be kept in main memory – maybe even the cache – making updates and queries on the data structure fast for even large values of m and n.*
- *Better System Performance: A synopsis data structure only uses a minor fraction of the main memory, leaving space for other data structures, which makes a synopsis data structure very suitable as a black box.*
- *Small surrogate: A synopsis data structure can function as an approximate replacement of the real data set, in cases where the data set is unavailable or too large to store and query efficiently.*

The low space usage of a synopsis data structure comes with a price. As described above we are no longer able to answer the problem exactly, but instead fall back to answering it approximately.

Generally the solutions to the approximate count tracking problem falls into two categories. Either they solve the problem deterministically with some approximation factor ϵ, or they solve the problem randomized, again with an approximation factor ϵ and a failure probability δ. Here, the approximation factor defines the absolute error allowed in the estimation of s_i, i.e. the error margin for which an estimate is accepted as a good estimate of s_i. The failure probability δ defines with what probability an estimate of s_i is bad, that is, the probability that the absolute error of an estimate of s_i is more than the maximum error of a good estimate, defined using the approximation factor ϵ.

The most popular deterministic solutions include the *Frequent* algorithm often referred to as the *Misra-Gries* algorithm [26, 13, 20], the *Lossy Counting* algorithm [23], and the *Space Saving* algorithm [24]. They all use the same idea of tracking a subset of the elements from the data stream and monitor the values of that subset, and they also only support $+1$ updates, i.e. c_t is assumed to be 1 for all updates. For problems where c_t can be greater than one, this can be handled by doing c_t updates with the same element and value one.

All of the above algorithms solves the approximate count tracking problem while guaranteeing an error of no more than $\epsilon \sum_{t=1}^{n} c_t$ from the true frequencies using $O\left(\epsilon^{-1}\right)$ space and $O\left(1\right)$ for updating and querying elements [7]. Another common thing about the algorithms is the fact that they do not support deletion of elements i.e. negative values of c_t. They all fall into the category of algorithms solving the approximate count tracking problem in the *Cash Register Model*.

A lot of applications, such as keeping tab on queries for specific elements in databases, need to be able to decrement the frequency of some elements if those are deleted from the data set. Such applications require a solution to the approximate count tracking problem in the *Turnstile Model*.

The most popular solutions for the problem in the *Turnstile Model* are sketches. These are based on randomization by their usage of hash functions, which introduce a certain failure probability, δ. More formally the two sketches presented in the following sections are able to provide estimates for s_i at time t, for which it holds that the absolute error is no more than $\epsilon \sum_{i=1}^{m} f_t(s_i)$ or $\epsilon \sqrt{\sum_{i=1}^{m} f_t(s_i)^2}$ with high probability. Hence, the sketches are able to bound the error margin according to the L_1 or L_2-norm of the provided data set.

3.2 Sketches

To give a clear introduction to the notion of sketches, we change the formal definition of the problem to be expressed in vectors. Let v be a vector with dimension m, where m is the number of distinct elements in the stream. At time t the vector is $v(t)^\intercal = [v_1(t) = f_t(s_1), \ \ldots, \ v_m(t) = f_t(s_m)]$, adopting the definition of the function $f_t(s_i)$ from above. Initially the vector will be the zero vector of dimension m. Updates (s_i, c_t) to the element

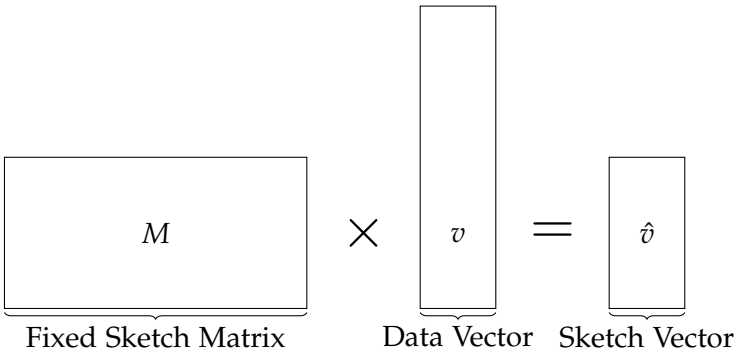


Figure 3.1: The relationship between v and $\hat{v}$ can be seen as a linear transformation of v and a fixed sketch matrix M yielding the sketch vector $\hat{v}$.

s_i is performed as $v_i(t) = v_i(t-1) + c_t$ and $\forall j \neq i : v_j(t) = v_j(t-1)$.

As described earlier, representing v explicitly is infeasible in the data stream model and instead we try to solve the problem of approximating v, yielding the approximate sketch vector $\hat{v}$.

The two different sketches that will be covered thoroughly later in this chapter, are both categorized as *linear sketches*. A *linear sketch* is defined as a sketch that can be seen as a linear transformation of the input data. Such a sketch comes with a certain set of properties: An update to the sketch is handled independent of the state of the structure, i.e. irrespective of updates in the past, and that implicitly work in the *Turnstile Model*.

Looking at an example e.g. if we were to model a vector of discrete frequencies summarized by a histogram, then the sketch of this $\hat{v}$ can be expressed as a linear transformation of the vector v with a fixed sketch matrix M, as illustrated in Figure 3.1 [6]. Any update to the vector v can be carried out by changing a single row in the vector v, whereas $\hat{v}$ can be found by multiplying the fixed sketch matrix M with the new representation of the vector v.

In practice, the fixed sketch matrix M, used to make the linear transformation, would require space greater than that required to store v. To handle this, most sketches instead use hash functions to generate a linear transformation with a smaller memory footprint.

Since the sketch produced by the linear transformation only take up a small fraction of the space needed to store v, the resulting vector $\hat{v}$ naturally becomes an approximation of the true vector v, which as for the counting solutions presented earlier, makes it interesting to bound errors of an approximation according to some ϵ, $0 < \epsilon \leq 1$.

Next, we will in detail describe two of the most common frequency based sketches, namely the Count-Min Sketch and the Count-Median Sketch, both supporting point queries for the estimated frequency $\hat{v}_i$ of a specific element s_i.

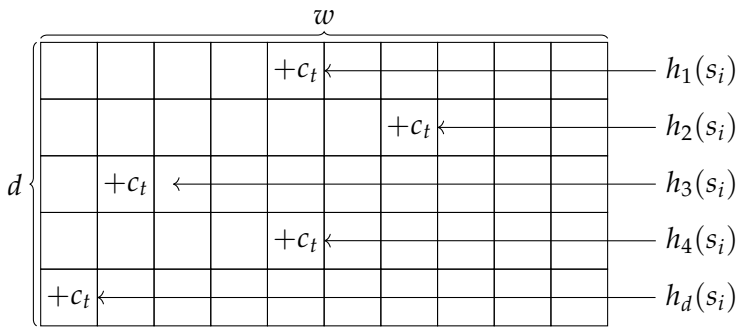


Figure 3.2: The matrix V with $b = 2$, $\epsilon = 0.20$ and $\delta = 0.05$. Element s_i is updated with the value c_t. This is done by finding its corresponding column in each row of the matrix using the hash functions.

3.3 Count-Min Sketch

The Count-Min Sketch [9] is a frequency based sketch, named after the two basic operations: Counting elements and finding the minimum. Given the parameters ϵ and δ, where ϵ is the approximation factor, and δ the probability of failure with respect to the approximation factor, a sketch is generated, which returns the approximate frequency $\hat{v}_i$ when queried, for which it holds that for any $i \in [m]_1$, $\hat{v}_i$ is no more than $\epsilon \|v\|_1$ larger than v_i with probability $1 - \delta$. Here $\|v\|_1 = \sum_{i=1}^{m} v_i$ is the L_1-norm of the true frequency vector v.

The sketch uses a matrix $V \in \mathbb{R}^{d \times w}$ where the amount of rows, d, will be called the depth and the amount of columns, w, will be called the width. Initially all entries are zeroed and the dimensions of the matrix are defined as $w = \lceil b/\epsilon \rceil$ and $d = \lceil \log_b \delta^{-1} \rceil$, where the base of the logarithm, b, can be chosen freely for all $b > 1$.

The linear transformation is performed by choosing d hash functions $h_1, \ldots, h_d$ independently, from a family of hash functions defined as $h_i : [m]_1 \to [w]_1$ for all i. In the original paper [9], the hash functions are chosen from a pair-wise independent family, however as will be shown in the analysis of the algorithm in Subsection 3.3.1, the requirement can be loosened to hash functions from a c-universal family for some constant c, by appropriately scaling w.

Each hash function is associated with a row in V. When an update is performed, the matrix is updated once for each row in V, that is, if element s_i is updated with value c_t at time t, all entries $V[j, h_j(s_i)]$ += c_t are updated for all $j \in [d]_1$, making updates in a specific row independent of updates to the other rows, by the independence of the hash functions. This is illustrated in Figure 3.2. The cost of such an update is only related to the depth d of the matrix, where constant work is used for each row, since an invocation of a hash function can generally be performed in $O(1)$ time (see Section 2.3), and updating a single entry in a row is also a constant operation, yielding $O(d)$ time in total.

The space used by the structure is the space occupied by the matrix V, that is $w * d$

words and the space used for the d hash functions, which for a specific family of c-universal hash functions are only a constant amount of memory per row [27]. This will be shown to be the optimal space usage of the problem in Chapter 5.

More generally, any family of c-universal hash functions could be used, while still maintaining the same bounds for the data structure, as long as the hash function can be stored in constant space and as long c is a constant.

As a final note about the Count-Min Sketch, it is argued in the original paper that choosing $b = e$ where e is the base of the natural logarithm, minimizes the space usage of the sketch [9]. Choosing $b = e$ changes the depth to $d = O\left(\ln \delta^{-1}\right)$, implies that the space and update bounds changes to $O\left(\epsilon^{-1} \ln \delta^{-1}\right)$ words and $O\left(\ln \delta^{-1}\right)$ time, respectively.

Several different types of queries can be performed on the Count-Min Sketch. The most important query for the work of this thesis and the reason why the parameter w and d are chosen as they are, is the point query $\mathcal{Q}(s_i)$ that returns the approximate frequency $\hat{v}_i$ of an element s_i.

3.3.1 Point Query

A point query $\mathcal{Q}(s_i)$ on the Count-Min Sketch, is defined as a query returning the estimated frequency $\hat{v}_i$ of element s_i. How close the estimated frequency is to the true frequency v_i, is defined according to the approximation factor ϵ and the failure probability δ.

A point query for element s_i is carried out by looking up all buckets $\hat{v}_{i,j} = V[j, h_j(s_i)]$ for $j \in [d]_1$. Since each row is build from a different hash function h_j, the estimates in each row are very likely to be different. Furthermore since we are in the *Strict Turnstile Model* and updates are carried out by addition of the values c_t, the error of each row is one-sided, meaning that $\hat{v}_{i,j} \geq v_i$ for all rows j. Hence, taking the minimum of all buckets $\min_j \hat{v}_{i,j}$ gives the closest estimate $\hat{v}_i$ generated by the sketch.

In the following theorem we will bound the exact guarantees given by the Count-Min Sketch for such a point query.

Theorem 1 (Count-Min Sketch point query guarantees)**.** *Using the Count-Min Sketch data structure it holds for the estimated frequency $\hat{v}_i$, that*

1. $v_i \leq \hat{v}_i$, *and*
2. *with probability* $1 - \delta : \hat{v}_i \leq v_i + \epsilon \|v\|_1$.

Proof. Let $I_{i,j,k}$ indicate if a collision occurs when two distinct elements are applied to the same hash function, for which

$$I_{i,j,k} = \begin{cases} 1 & \text{if } i \neq k \wedge h_j(s_i) = h_j(s_k) \\ 0 & \text{otherwise} \end{cases}$$

for all i, j, k where $i, k \in [m]_1$ and $j \in [d]_1$.

The expected amount of collisions can be derived from the choice of the family of hash functions. The probability of collision for a c-universal is by definition $\mathbb{P}\left[h_j(s_i) = h_j(s_k)\right] \leq \frac{c}{w}$ for $i \neq k$. The expectation of $I_{i,j,k}$ then becomes:

$$\begin{aligned}
\mathbb{E}\left[I_{i,j,k}\right] &= \mathbb{P}\left[h_j(s_i) = h_j(s_k)\right] \\
&\leq \frac{c}{w} && (c\text{-universal}) \\
&\leq \frac{c}{\left\lceil \frac{b}{\epsilon} \right\rceil} && (\text{Substitute } w) \\
&\leq \frac{\epsilon c}{b}
\end{aligned}$$

Let $X_{i,j}$ be the random variable $X_{i,j} = \sum_{k=1}^{m} I_{i,j,k} v_k$ for $i \in [m]_1$, $j \in [d]_1$, and from the independent choices of the hash functions. $X_{i,j}$ then expresses all the additional mass contributed by other elements as a consequence of hash function collisions. Since v_i is non-negative per definition of the *Strict Turnstile Model*, it must also hold that $X_{i,j}$ is non-negative. By the construction of the array V of the Count-Min Sketch data structure, an entry in the array is then $V[j, h_j(s_i)] = v_i + X_{i,j}$. This implies that item 1 is true, since $\hat{v}_i = \min_j V[j, h_j(s_i)] \geq v_i$.

Proving item 2 requires further work. Observe that the expected collision mass for $i \in [m]_1$, $j \in [d]_1$, and the constant c can be defined as:

$$\begin{aligned}
\mathbb{E}\left[X_{i,j}\right] &= \mathbb{E}\left[\sum_{k=1}^{m} I_{i,j,k} v_k\right] \\
&= \sum_{k=1}^{m} v_k \mathbb{E}\left[I_{i,j,k}\right] && (\text{Linearity of Expectation}) \\
&\leq \frac{\epsilon c}{b} \sum_{k=1}^{m} v_k && (\text{Substitute } \mathbb{E}\left[I_{i,j,k}\right]) \\
&= \frac{\epsilon c}{b} \left\|v\right\|_1 && (3.1)
\end{aligned}$$

Using this, we can calculate the probability that $\hat{v}_i > v_i + \epsilon \left\|v\right\|_1$, i.e. the probability that the estimate of the frequency is larger than the expected error bound introduced by approximation. This is reasonably simple to do, since taking the minimum of all the estimated frequencies for an element s_i from each of the rows, gives us the closest estimate to the true frequency. For this estimate to fail ($\hat{v}_i > v_i + \epsilon \left\|v\right\|_1$) it must by the definition of the minimum have failed for all rows. This is what is expressed in the following:

$$\begin{aligned}
\mathbb{P}\left[\hat{v}_i > v_i + \epsilon \left\|v\right\|_1\right] &= \prod_j \mathbb{P}\left[V[j, h_j(s_i)] > v_i + \epsilon \left\|v\right\|_1\right] && (3.2) \\
&= \prod_j \mathbb{P}\left[v_i + X_{i,j} > v_i + \epsilon \left\|v\right\|_1\right] && (\text{Substitute})
\end{aligned}$$

$$
\begin{aligned}
&= \prod_j \mathbb{P}\left[X_{i,j} > \frac{b}{c}\mathbb{E}\left[X_{i,j}\right]\right] && \text{(Substitute (3.1))}\\
&< \prod_j \frac{c}{b} && \text{(Markov's Inequality)}\\
&= c^d b^{-d} \leq c^d \delta
\end{aligned}
$$

Note that the product of the probabilities from (3.2) is possible as the hash functions h_j are all independent of each other. If we rescale w with the constant c, that is $w = \lceil bc/\epsilon \rceil = O\left(\epsilon^{-1}\right)$, and redo the analysis in this theorem we end up removing the c constant throughout the proof, giving us that $\mathbb{P}\left[\hat{v}_i \leq v_i + \epsilon \left\|v\right\|_1\right]$ is $1-\delta$, which proves item 2. □

A point query of the Count-Min Sketch data structure is guaranteed to answer within a certain error range of the correct frequency with a certain probability while only using $d = O\left(\log_b \delta^{-1}\right)$ time answering the query, since a point query is simply carried out by a hashing a value and visiting an index for each of the d rows and returning the estimated frequency of the minimum one. Choosing $b = e$ will consecquently change the time bound of the query to $O\left(\ln \delta^{-1}\right)$.

3.4 Count-Median Sketch

The Count Sketch data structure Charikar et al. [5] also referred to as the Count-Median Sketch has without doubt inspired the creation of the Count-Min Sketch, described in Section 3.3. The data structures are similar and only differ in a few important points.

Given the parameters ϵ and δ, where ϵ bounds the fraction of error allowed for each element, and δ the probability of failing to uphold this, a matrix $V \in \mathbb{R}^{d \times w}$ with the rows and columns denoted as the width and depth. Initially, all entries are zeroes and the dimensions are defined as $w = \lceil k/\epsilon^2 \rceil$ and

$$
d = \left\lceil \frac{\ln\left(\delta^{-1}\right)}{\frac{1}{6} - \frac{1}{3k}} \right\rceil
$$

for some constant $k > 2$ determining the error probability of a specific bucket.

The sketch represented by V is then able to guarantee that for a point query $\mathcal{Q}\left(s_i\right)$ an estimate of the frequency $\hat{v}_i$ of an element s_i can be returned, such that the absolute error of $\hat{v}_i$ is no more than $\epsilon \left\|v\right\|_2$ from the true frequency v_i with probability $1-\delta$. Here $\left\|v\right\|_2 = \sqrt{\sum_{i=1}^{m} v_i^2}$ is the L_2-norm of the input data.

For each row j in V a pair of independent hash functions (h_j, g_j) are associated. Let the hash functions $h_1, \ldots, h_d$ and $g_1, \ldots, g_d$ create the linear transformation of the input vector v to its estimate $\hat{v}$, defined as $h_j : [m]_1 \to [w]_1$ and $g_j\left[m\right]_1 \to \{-1, 1\}$. It will then suffice to draw the hash functions h_j from a c-universal family using more or less the same argument as in Theorem 1. That is, a c-universal hash function suffices as long as w is scaled by the constant c, giving $w = \lceil kc/\epsilon^2 \rceil$. For the g_j hash functions, a tighter

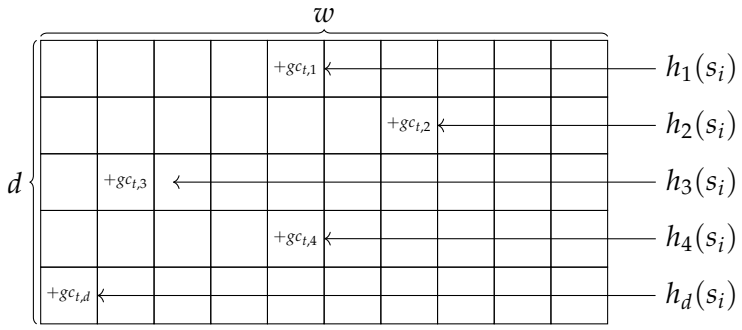


Figure 3.3: The matrix V. Element s_i is updated with the value $gc_{t,j} = g_j(s_i)\, c_t$ for each of the j rows in V. This is done by finding its corresponding column in each row of the matrix using the hash functions.

guarantee is needed for the family, namely that g_j is chosen from a pair-wise independent family (see Section 2.3).

At time t, when element s_i should be updated with value c_t each row of V is updated as follows: $V[j, h_j(s_i)] \mathrel{+}= g_j(s_i)\, c_t$ for all $j \in [d]_1$. The updates of the rows will happen independent of each other, because of the independence of the hash functions h_j and g_j. In Figure 3.3, an illustration of such an update is shown.

As for the Count-Min Sketch, the cost of the update is only dependent on the number of rows in V. This comes from the fact that for each update two invocations of two hash functions, a multiplication, and finally an addition is performed, yielding constant time for each of the d rows, resulting in $d = O\left(\ln \delta^{-1}\right)$ running time.

The space used by the data structure is the size of the matrix

$$|V| = w\, d = \left\lceil \frac{k}{\epsilon^2} \right\rceil \left\lceil \frac{\ln\left(\delta^{-1}\right)}{\left(\frac{1}{6} - \frac{1}{3k}\right)} \right\rceil = O\left(\epsilon^{-2} \ln \delta^{-1}\right)$$

words to store the estimates and the amount of space used to store the hash functions, which can be done in constant space [27] for each of the d rows, adding an extra $O(d)$ words of space. This space usage is shown to be optimal in order to provide estimated frequencies with errors according to the L_2-norm in Chapter 5.

As for the Count-Min Sketch, the Count-Median Sketch can be queried in several different but interesting ways. For the purpose of this thesis it suffices to present and analyze the point query $\mathcal{Q}(s_i)$, which returns the approximated value $\hat{v}_i$ for element s_i.

3.4.1 Point Query

For a point-query $\mathcal{Q}(s_i)$ of the Count-Median Sketch data structure, an approximate value $\hat{v}_i$ of the true frequency v_i will be returned. Since the use of the sign in the update leads to additions and subtractions of the true frequency, it is not possible to take the minimum value over the rows of V. Instead of taking the minimum, we will show that taking the median of the d estimated frequencies will suffice to provide the guarantees

stated earlier. To query for the estimated frequencies in the Count-Median Sketch, one has to reverse the sign – the effect of multiplying the value from g_j – that is applied when performing an update.

In the lemma that follows, the expected error and variance of a specific bucket $\hat{v}_{i,j} = V[j, h_j(s_i)]$ is found. These will lead to the theorem further below, stating the precise bound for the Count-Median Sketch with respect to ϵ, δ and a constant k.

Lemma 7 (Count-Median Sketch expected bucket error)**.** *The expected error – where the error is thought of as the mass contributed from colliding elements – in a bucket $V[j, h_j(s_i)]$ is 0, while the variance of the same bucket is $\frac{c\|v\|_2^2}{w}$.*

Proof. Let $I_{i,j,k}$ be the indicator variable indicating if a collision occurs for two distinct elements when applied to the same hash function, defined as

$$I_{i,j,k} = \begin{cases} 1 & \text{if } i \neq k \wedge h_j(s_i) = h_j(s_k) \\ 0 & \text{otherwise.} \end{cases}$$

for $i, j, k \mid i, k \in [m]_1, j \in [d]_1$. The expected amount of collisions can then be derived since h_j is chosen from a c-universal family, which by definition has probability $\leq c/w$ for two distinct elements to collide.

$$\begin{aligned} \mathbb{E}\left[I_{i,j,k}\right] &= \mathbb{P}\left[h_j(s_i) = h_j(s_k)\right] \\ &\leq \frac{c}{w} && (c\text{-universal}) \end{aligned}$$

Let $X_{i,j} = \sum_{k=1}^{m} I_{i,j,k} v_k g_j(s_k)$ be the random variable describing the mass of all elements that collide with s_i for hash function h_j. Hence, we can associate $X_{i,j}$ with the content of a bucket, since we can rewrite $V[j, h_j(s_i)] = X_{i,j} + g_j(s_i)v_i$.

If we expect over $X_{i,j}$ we get the expected collision mass introduced in each bucket of the sketch:

$$\begin{aligned} \mathbb{E}\left[X_{i,j}\right] &= \mathbb{E}\left[\sum_{k=1}^{m} I_{i,j,k}\, v_k\, g_j(s_k)\right] \\ &= \sum_{k=1}^{m} \left(\mathbb{E}\left[I_{i,j,k}\right]\; v_k\, \mathbb{E}\left[g_j(s_k)\right]\right) && \text{(Linearity of Expectation)} && (3.3) \\ &= \frac{c}{w} \sum_{k=1}^{m} \left(v_k\, \mathbb{E}\left[g_j(s_k)\right]\right) && \text{(Substitute } \mathbb{E}\left[I_{i,j,k}\right]\text{)} \\ &= 0 * \frac{c}{w} \sum_{k=1}^{m} v_k && \left(\mathbb{E}\left[g_j(s_k)\right] = \frac{1}{2} * -1 + \frac{1}{2} * 1 = 0\right) && (3.4) \\ &= 0 \end{aligned}$$

In expectation, the error of each bucket is then canceled out due to the use of the g_j hash functions. Note that (3.3) comes from the fact that the hash functions g_j and h_j are independent of each other, and (3.4) holds since g_j is chosen from a pair-wise independent family of hash functions.

Furthermore, one can expect over the variance of $X_{i,j}$, to see how much the estimated error will variate from the expected value of $X_{i,j}$.

$$
\begin{aligned}
\mathrm{Var}\left(X_{i,j}\right) &= \mathbb{E}\left[X_{i,j}^2\right] - \mathbb{E}\left[X_{i,j}\right]^2 \\
&= \mathbb{E}\left[X_{i,j}^2\right] - 0 && \text{(Substitute } \mathbb{E}\left[X_{i,j}\right]\text{)} \\
&= \mathbb{E}\left[\left(\sum_{k=1}^{m} I_{i,j,k}\, v_k\, g_j(s_k)\right)^2\right] \\
&= \mathbb{E}\Bigg[\sum_{k=1}^{m} \left(I_{i,j,k}\, v_k\, g_j(s_k)\right)^2 \\
&\quad + \sum_{k' \neq k} I_{i,j,k}\, v_k\, g_j(s_k)\, I_{i,j,k'}\, v_{k'}\, g_j(s_{k'})\Bigg] \\
&= \mathbb{E}\left[\sum_{k=1}^{m} \left(I_{i,j,k} v_k g_j(s_k)\right)^2\right] \\
&\quad + \sum_{k' \neq k} \mathbb{E}\left[I_{i,j,k} I_{i,j,k'}\right]\, v_k\, \mathbb{E}\left[g_j(s_k)\right]\; v_{k'}\, \mathbb{E}\left[g_j(s_{k'})\right] && (3.5) \\
&= \mathbb{E}\left[\left(\sum_{k=1}^{m} I_{i,j,k}^2\, v_k^2\, g_j(s_k)^2\right)\right] && \left(\mathbb{E}\left[g_j(s_k)\right] = 0\right) \\
&= \mathbb{E}\left[\left(\sum_{k=1}^{m} I_{i,j,k}^2\, v_k^2\right)\right] && \left(g_j(s_k)^2 = 1^2 \vee (-1)^2 = 1\right) \\
&= \mathbb{E}\left[\left(\sum_{k=1}^{m} I_{i,j,k}\, v_k^2\right)\right] && \text{(Definition of } I_{i,j,k}\text{)} \\
&= \sum_{k=1}^{m} \left(\mathbb{E}\left[I_{i,j,k}\right]\; v_k^2\right) && \text{(Linearity of Expectation)} \\
&= \frac{c}{w} \sum_{k=1}^{m} v_k^2 && \text{(Substitute } \mathbb{E}\left[I_{i,j,k}\right]\text{)} \\
&= \frac{c\, \|v\|_2^2}{w}
\end{aligned}
$$

Where (3.5) comes from Linearity of Expectation, the fact that h_j and g_j are chosen independently of each other, and because g_j is pair-wise independent. □

From Lemma 7 we get that each bucket $V[j, h_j(s_i)]$ is expected to hold the value $v_i\ g_j(s_i)$ and not have a variance of more than $c\,\|v\|_2^2/w$ from the expected value.

To actually be able to get a meaningful output of a query to a specific bucket, the sign of a specific element has to be reversed. This will enable us to get the expected frequency

of every bucket along with the expected variance. Moreover, to bound the approximation factor and failure probability of the entire query algorithm, the analysis has to take into account, finding the median of all the d buckets associated with a specific element. In the following theorem all of this will be handled.

Theorem 2 (Count-Median Sketch point query guarantees)**.** *Using the Count-Median Sketch data structure it holds for the estimated frequency $\hat{v}_i$, that with probability $1-\delta$: $|\hat{v}_i - v_i| \leq \epsilon \|v\|_2$ for depth $d = \frac{\ln\left(\delta^{-1}\right)}{\left(\frac{1}{6}-\frac{1}{3k}\right)} = O\left(\ln \delta^{-1}\right)$ and width $w = \frac{ck}{\epsilon^2} = O\left(\epsilon^{-2}\right)$.*

Proof. We denote the output of the algorithm $\hat{v}_i = \text{median}_j V[j, h_j(s_i)] g_j(s_i)$ and further denote a specific bucket of a query $\hat{v}_{i,j} = V[j, h_j(s_i)] g_j(s_i)$. First, lets compute the expectation of the output of each bucket.

$$\begin{aligned}
\mathbb{E}\left[\hat{v}_{i,j}\right] &= \mathbb{E}\left[V[j, h_j(s_i)] g_j(s_i)\right] \\
&= \mathbb{E}\left[\left(X_{i,j} + v_i g_j(s_i)\right) g_j(s_i)\right] && \text{(Substitute } V[j, h_j(s_i)]\text{)} \\
&= \mathbb{E}\left[X_{i,j} g_j(s_i) + v_i\right] && \left(g_j(s_i)^2 = 1\right) \\
&= \mathbb{E}\left[X_{i,j}\right] \mathbb{E}\left[g_j(s_i)\right] + v_i && (3.6) \\
&= v_i && \text{(Lemma 7)}
\end{aligned}$$

where (3.6) comes from the fact that $g_j(s_i)$ is independent of the hash function $h_j(s_i)$ from the $X_{i,j}$ variable and furthermore because of it being independent of the choices of the other sign hashes $g_j(s_k)$ likewise defined in the $X_{i,j}$ variable. Next, the variance of each bucket can be calculated:

$$\begin{aligned}
\text{Var}\left(\hat{v}_{i,j}\right) &= \mathbb{E}\left[\hat{v}_{i,j}^2\right] - \mathbb{E}\left[\hat{v}_{i,j}\right]^2 \\
&= \mathbb{E}\left[\hat{v}_{i,j}^2\right] - v_i^2 && \text{(Substitute } \mathbb{E}\left[\hat{v}_{i,j}\right]\text{)} \\
&= \mathbb{E}\left[\left(V[j, h_j(s_i)] g_j(s_i)\right)^2\right] - v_i^2 && \text{(Substitute } \hat{v}_{i,j}\text{)} \\
&= \mathbb{E}\left[\left(\left(X_{i,j} + v_i g_j(s_i)\right) g_j(s_i)\right)^2\right] - v_i^2 && \text{(Substitute } V[j, h_j(s_i)]\text{)} \\
&= \mathbb{E}\left[\left(X_{i,j} g_j(s_i) + v_i\right)^2\right] - v_i^2 && \left(g_j(s_i)^2 = 1\right) \\
&= \mathbb{E}\left[\left(X_{i,j} g_j(s_i)\right)^2 + 2(X_{i,j} g_j(s_i) v_i) + v_i^2\right] - v_i^2 \\
&= \mathbb{E}\left[X_{i,j}^2 + 2(X_{i,j} g_j(s_i) v_i) + v_i^2\right] - v_i^2 \\
&= \mathbb{E}\left[X_{i,j}^2\right] + 2(\mathbb{E}\left[X_{i,j}\right] \mathbb{E}\left[g_j(s_i)\right] v_i) + v_i^2 - v_i^2 && (h_j(s_i),\, g_j(s_i) \text{ are independent}) \\
&= \mathbb{E}\left[X_{i,j}^2\right] + v_i^2 - v_i^2 && \left(\mathbb{E}\left[g_j(s_i)\right] = 0\right) \\
&= \frac{c\|v\|_2^2}{w} && \text{(Lemma 7)}
\end{aligned}$$

The final step is to ensure that large deviations from the mean in any bucket, does not happen very frequently. This can be done using Chebyshev's Inequality

$$\mathbb{P}\left[|\hat{v}_{i,j} - v_i| \geq \epsilon' \sqrt{\frac{c\left\|v\right\|_2^2}{w}}\right] \leq \frac{1}{\epsilon'^2} \Rightarrow \mathbb{P}\left[|\hat{v}_{i,j} - v_i| \geq \epsilon' \frac{\sqrt{c}\left\|v\right\|_2}{\sqrt{w}}\right] \leq \frac{1}{\epsilon'^2}$$

$$\Rightarrow \mathbb{P}\left[|\hat{v}_{i,j} - v_i| \geq \epsilon \left\|v\right\|_2\right] \leq \frac{c}{w\epsilon^2} \qquad (\epsilon' = \frac{\sqrt{w}\epsilon}{\sqrt{c}})$$

and choosing $w = \frac{ck}{\epsilon^2}$ gives that a bucket deviates from the mean with more than the variance, with probability at most k^{-1}.

Having bounded the error on each bucket of the sketch, we are able to bound the whole sketch by bounding the probability over the median of all the buckets that are associated with a frequency estimate $\hat{v}_i$. The median is the middle element in a sorted set, hence for the median to have a large deviation from the mean, it most hold that $d/2$ of the buckets deviates with at least as much. Let $Y_{i,j}$ be an indicator variable indicating if a bucket estimate $\hat{v}_{i,j}$ is off by more than allowed, defined as

$$Y_{i,j} = \begin{cases} 1 & \text{if } |\hat{v}_{i,j} - v_i| > \epsilon \left\|v\right\|_2 \\ 0 & \text{otherwise} \end{cases}$$

Let $Y_i = \sum_{j=1}^{d} Y_{i,j}$ be the total number of failed buckets over all rows, for element s_i. Since we have $j \in [d]_1$ and all of the d rows are independent of each other in the sketch, it must hold that $\mathbb{E}\left[Y_i\right] \leq d/k$ using a Union Bound, which gives us:

$$\begin{aligned}
\mathbb{P}\left[|\hat{v}_i - v_i| > \epsilon \left\|v\right\|_2\right] &\leq \mathbb{P}\left[Y_i \geq \frac{d}{2}\right] \\
&\leq \mathbb{P}\left[Y_i \geq \frac{k}{2}\mathbb{E}\left[Y_i\right]\right] && \text{(Substitution)} \\
&\leq \mathbb{P}\left[Y_i \geq \left(1 + \left(\frac{k}{2} - 1\right)\right)\mathbb{E}\left[Y_i\right]\right] && (3.7) \\
&\leq e^{-\frac{(\frac{k}{2}-1)\mathbb{E}\left[Y_i\right]}{3}} && \text{(Chernoff Bound mult. form)} \\
&\leq e^{-\frac{(\frac{k}{2}-1)\frac{d}{k}}{3}} \\
&\leq e^{-(\frac{d}{6} - \frac{d}{3k})} \\
&\leq e^{-d(\frac{1}{6} - \frac{1}{3k})} && (3.8)
\end{aligned}$$

Here (3.7) requires that $k > 2$, which is also required in (3.8). Choosing $d = \frac{\ln\left(\delta^{-1}\right)}{\left(\frac{1}{6} - \frac{1}{3k}\right)}$ gives us:

$$\mathbb{P}\left[|\hat{v}_i - v_i| > \epsilon \left\|v\right\|_2\right] \leq \delta$$

which proves this theorem. □

The point query has a running time proportional to the depth of the sketch. As for the update procedure of Count-Median Sketch, two invocations of a hash function, a multiplication, and an addition is required at each row. All these operations are constant and add up to $O(d)$ time. Furthermore the median of the d estimates must be found, which can be done in linear $O(d)$ time [2], yielding a total of $O(d) = O\left(\ln \delta^{-1}\right)$ running time for the point query overall.

3.5 Summary

Wrapping up, we have visited the problem of count tracking, formally defined the problem, and stated it in two different models namely the *Cash Register Model* and *Turnstile Model*.

Generally, in the data stream model it is not possible to solve the problem exactly in acceptable time and with a practical space consumption. For the approximate version of the same problem in the *Cash Register Model* many deterministic and randomized algorithms have been proposed, which solves the problem using small space and fast queries.

In the *Turnstile Model* fewer solutions exist. Two of those have been thoroughly introduced and analyzed, namely the Count-Min and Count-Median Sketches.

We have shown that the Count-Min Sketch uses $O\left(\epsilon^{-1} \ln \delta^{-1}\right)$ words of space to provide a data structure where both updates and queries have a cost of $O\left(\ln \delta^{-1}\right)$ time. The Count-Min Sketch gives an L_1-norm guarantee according to the input data where frequency estimates $\hat{v}_i$ in the sketch is within an additive factor of $\epsilon \|v\|_1$ of the true frequency v_i with probability $1 - \delta$. Moreover, the guarantee is one-sided because $\hat{v}_i \geq v_i$.

The Count-Median Sketch provides a better guarantee, since it guarantees that $\hat{v}_i$ is within an additive factor of $\epsilon \|v\|_2$ of the true frequency v_i with probability $1 - \delta$. This guarantee is significantly stronger in most cases as $\|v\|_2 \leq \|v\|_1$. The cost of this guarantee is significantly larger, since the Count-Median Sketch requires $O\left(\epsilon^{-2} \ln \delta^{-1}\right)$ words of space to support updates and queries in the same time as Count-Min Sketch.

As such, the sketches can be hard to compare since they for most cases provide different error guarantees. However, we will still try to do experiments with both structures in Section 6.2. In the same section, we will show a proof which states that the Count-Min Sketch and Count-Median Sketch is in fact comparable provided the same amount of space.

As neither of the sketches is dependent on m or n, they both serve as good theoretical synopsis data structures. Generally, such structures are applicable in many situations, one of which is the approximate heavy hitters problem in the *Turnstile Model*. This problem uses a solution to the count tracking problem in the *Turnstile Model* as a black box in order to solve the heavy hitters problem, hence both sketches are interesting for such a problem.

Chapter 4

Heavy Hitters

In this chapter, we describe the problem of finding heavy hitters in the data stream model, also known in the literature as the problem of finding the most popular items, frequent items, hot items, elephants, and iceberg queries.

Different solutions to the problem of finding heavy hitters are used in a wide range of problems, such as finding IP addresses by some metric, e.g. packet size at AT&T [12], indexing web searches at Google [29], or finding hot items in relational databases to only name a few. Hence, it is an important matter to reason and experiment with solutions to the problem in order to support such applications with the best possible solutions.

The problem is usually specified in relation to a norm of the input data, which defines what it means to be frequent. This is usually according a threshold factor ϕ of either the L_1 or L_2-norm of the input data. The wide specification of the problem and the fact that it is stated in the data stream model naturally provides a lot of different algorithms, solving the problem according to the *Cash Register Model*, the *Strict Turnstile Model*, the general *Turnstile Model*, or multiple of them.

While solutions in the *Cash Register Model* have been widely studied, fewer solutions exist in the *Turnstile Model*. The main focus of this thesis will be on solving the appropriate L_1 heavy hitters problem in the *Strict Turnstile Model*. The problem in this model is of great interest, since a lot of different applications need to be able to both insert and delete elements, while having the constraint that frequencies in general never become negative. We do not touch the problem for the general *Turnstile Model*, but solutions to the general problem has been provided in the literature using ideas such as Group Testing [7, 10] and cluster-preserving clustering [22].

In the following section we will introduce and formally describe the problem and its variants. Then we will go into detail of how to solve the approximate L_1 heavy hitters problem in the *Cash Register Model*. In the remaining sections we will return to the *Strict Turnstile Model*, starting with a general description of a hierarchical structure and then detailed descriptions of different algorithms using the hierarchical structure to solve the approximate L_1 heavy hitters problem.

4.1 The Problem

Finding heavy hitters in a data stream, is the problem of finding those elements that appear most frequently, for some definition of "frequent" that depends on the norm of the input data.

More formally for a data stream of size n with m unique elements, updates are tuples (s_i, c_t) where $i \in [m]_1$ and $c_t \in \mathbb{R}$ is the update value, at time $t \in [n]_1$. The task is to keep count of the updates of the set of elements $S = \{s_1, \ldots, s_m\}$ by storing the frequencies of the elements as a vector $v(t)^\intercal = [v_1(t) = f_t(s_1), \ldots, v_m(t) = f_t(s_m)]$ where $f_t(s_i)$ is defined as a function determining the frequency at any time t of an element s_i.

An update from time $t-1$ to time t is then carried out by changing a single index in the vector, namely $v_i(t) = v_i(t-1) + c_t$, while the rest of the indices stay the same i.e. $\forall j \neq i, j \in [m]_1 : v_j(t) = v_j(t-1)$.

The goal is to find all elements, for which it holds that their frequency $v_i(t) \geq \phi \|v(t)\|$ where $0 < \phi \leq 1$ is a parameter determining the fraction of the norm, for which an element is said to be frequent. In short, we are trying to find elements which are classified as heavy hitters by having a frequency $v_i(t) \geq \phi \|v(t)\|$.

An algorithm solving such a problem needs to support the following two operations: `Update(`s_i`,` c_t`)` and `Query()`. The `Update` operation will modify some auxiliary data structure in order to support the `Query` operation in returning all elements which are heavy hitters.

The exact solution to the problem could be solved by storing a counter for each unique element s_i and return those elements that exceed the $\phi \|v(t)\|$ threshold. This solution is the same as using the exact frequency algorithm mentioned in Section 3.1 as a black box, to find the heavy hitters. Thus, finding the exact heavy hitters is the task of passing through the data stream once, and query every single element in the exact frequency algorithm.

The problem with an exact counting solution in practice, is that n and m in the data stream often are of such size that keeping count of for example m elements would exceed the amount of internal memory available, forcing one to use external memory such as hard disks, which would invoke a tremendous slowdown.

Ideally, one would wish to support the heavy hitters problem as stated above, but using sublinear space in the amount of unique elements m in the data stream. Exactly solving the heavy hitters problem can be proven to require at least linear space in m, as shown below.

Consider the set of elements $S = \{s_1, \ldots, s_m\}$ and an initial data stream of $n = m$ elements where each of the elements $s_i \in S$ appears in the stream exactly once. Now add $\phi \|v(t)\| - 1$ of the same element s_j for $j \in [l]_1$, $l > m$ to the data stream. At time t, if $s_j \notin S$ then s_j is not a heavy hitter since its frequency must be $\phi \|v(t)\| - 1$. On the other hand if $s_j \in S$ then its frequency must be $\phi \|v(t)\|$, making it a heavy hitter. This example can be seen as the Membership Problem i.e. is s_j a member of the set S? To answer such a question $\Omega(m)$ space must be used as a consequence of the Pigeonhole Principle.

For any algorithm to guarantee that only the elements s_i with frequency $v_i \geq \phi \|v(t)\|$ are output, at least $\Omega(m)$ space must be used since any less would result in either some

of the heavy hitters not being returned or some non-heavy hitter elements to be included in the resulting set.

As the heavy hitters problem cannot be solved exactly without using linear space, other approaches need to be taken when going sublinear as is the case for algorithms in the data stream model. Weakening the constraints of the definition by allowing a small error enables approximation of the solution. Such approximations can use less space as the exact solution is not required.

In an approximation setting, one still seeks to guarantee that the algorithm finds all heavy hitters, that is all elements s_i with frequency $\phi \|v(t)\|$ and above, but also limits the number of other – non-heavy hitter elements – to be returned.

Limiting other elements is done by defining the guarantee on the basis of not including elements with a frequency less than an approximation factor ϵ. That is, for $\epsilon < \phi$ no elements with a frequency less than $(\phi - \epsilon) \|v(t)\|$ is returned with high probability, $1 - \delta$. The approximation factor ϵ, then represents the allowed absolute error of the estimated frequency of any specific element s_i and δ the probability of failing to uphold this.

For such an approximation several algorithms that achieve these requirements have been proposed using sublinear space in the amount of unique elements. Common for most of these algorithms are that they use frequency estimation algorithms (Chapter 3) as a black box to enable the approximation guarantees.

The following sections will cover different solutions to the approximate heavy hitters problem. First in the *Cash Register Model* and then three different algorithms in the *Strict Turnstile Model*, each obtaining different space usages, update, and query times.

4.2 Cash Register Model

In the *Cash Register Model* the solution to the approximate heavy hitters problem is generally easier than in the *Turnstile Model*. This is due to the fact that only increments are allowed in the *Cash Register Model*.

The constraint of only having positive insertions is extremely powerful, since in general one only has to store the amount of potential heavy hitter candidates $\approx \phi^{-1}$ at any time. This allows for less space usage and better running times for both update and queries.

The main focus of this thesis is on the *Strict Turnstile Model* and thus, this section only briefly mentions deterministic and randomized solutions for the approximate L_1 heavy hitters problem in the *Cash Register Model*. As a consequence, the solutions and requirements of the solutions will only be touched upon briefly, which implies that any deeper understanding of the algorithms must be obtained from the references.

4.2.1 Deterministic

In the *Cash Register Model*, deterministic algorithms exist, which guarantee that all heavy hitters are returned and no elements s_j with frequency $v_j(t) < (\phi - \epsilon) \|v(t)\|_1$ is returned.

These are basically the same deterministic solutions as mentioned in Section 3.1 i.e. the `Frequent` algorithm also known as the `Misra-Gries` algorithm [26], the `Lossy Counting` algorithm [23], and the `Space Saving` algorithm [24], to mention a few.

What all of the algorithms above use is that the frequency of an element s_i at time $t \leq n$, will always be below or equal to the frequency at time n when the stream ends. Hence, they all hold only a subset of size ϵ^{-1} of the highest frequency elements at any time t and at time n return those elements in the subset for which their estimated frequency $\hat{v}_j(n) \geq (\phi - \epsilon) \|v(n)\|_1$. Such solutions will only require $O\left(\epsilon^{-1}\right)$ words of space while updates are performed in $O(1)$ time and queries in $O\left(\epsilon^{-1}\right)$ time.

4.2.2 Randomized

Randomized solutions also exist for the *Cash Register Model*. One such solution uses the Count-Min Sketch as a black box. What is observed in Cormode and Muthukrishnan [9] is the fact that updates only ever increase the estimated frequency $\hat{v}_i$ in a sketch and more generally the L_1-norm. As such, whenever an estimate exceeds the heavy hitter threshold it must be due to it being a heavy hitter or because the sketch has overestimated the frequency. Note that the L_1-norm that is part of the threshold, can be maintained at all times since it is simply the sum of all updated values.

For all updates to the sketch, one checks the estimated frequency to see if it exceeds the threshold. If so, the value is put into a Min Heap containing all potential heavy hitters. Furthermore the minimum element of the Min Heap is checked if still being larger than the threshold, if not a Delete-Min operation is performed.

When a heavy hitters query is performed the Min Heap can be scanned and all elements s_i having an estimated value $\hat{v}_i(t) \geq \phi \|v(t)\|_1$ are returned as heavy hitters. Properly scaling the failure probability provided to the sketch: $\delta = \log \frac{\|v(t)\|_1}{\delta'}$ ensures that with probability $1 - \delta'$, no elements s_j with frequency $v_j(t) < (\phi - \epsilon) \|v(t)\|_1$ is output.

This heavy hitters algorithm uses $O\left(\epsilon^{-1} \log\left(\|v(t)\|_1/\delta'\right)\right)$ words of space, supporting updates in $O\left(\log\left(\|v(t)\|_1/\delta'\right)\right)$ time and queries in $O\left(\phi^{-1}\right)$ time.

Compared to the *Turnstile Model*, the L_1-norm and estimated frequencies of elements are never decreased. If decrements was supported, then decreasing an element's frequency would also imply a decrease of the L_1-norm. For the randomized solution above, such a change could cause that elements not currently stored in the heap, would become heavy hitters. Such elements would not be detectable in other ways than scanning all m elements, making the use of the heap irrelevant. Hence, for a Count-Min Sketch to solve the approximate L_1 heavy hitters problem in the *Strict Turnstile Model*, one could simply maintain the same sketch as for the randomized solution, but sacrifice the query time, which would become $O(m)$.

To obtain better query times another structuring of the estimates is needed to support the strict or general *Turnstile Model* with an acceptable query time, while still maintaining update time and space usage close to the randomized solution presented in this section.

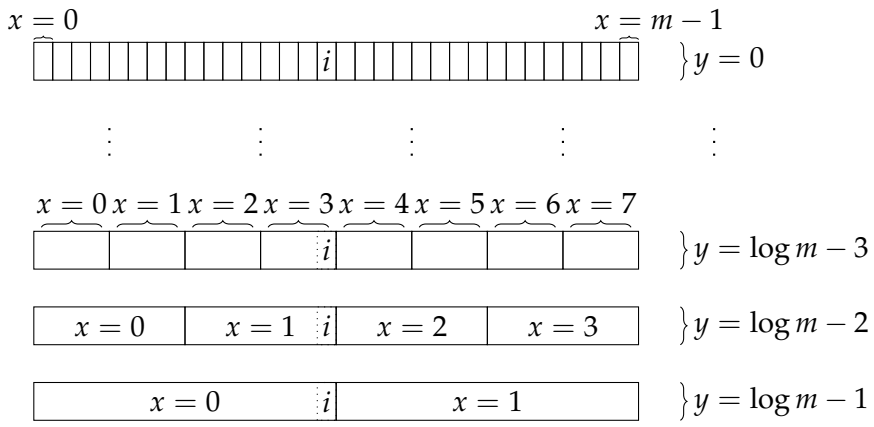


Figure 4.1: Illustration of the $[m]_1$ unique elements split into dyadic ranges, such that for each level y a sketch is held over $|x|$ buckets. Here the i'th element is shown through the hierarchy.

4.3 General Sketch Approach

This section describes some of the common ideas for sketch based approximate heavy hitters solutions in the *Strict Turnstile Model*. These ideas are used in the next sections where specific sketches are combined with the outlines of this section to get different approximate heavy hitters algorithms.

The sketch based solutions mentioned in the following sections all use the concept of dyadic ranges to create a hierarchical data structure that enables faster heavy hitters queries. Dyadic ranges are a way of dividing the data into several levels, each having the data separated into an increasing number of ranges. In our case, the ranges is split over the unique elements in the stream m and provide range sums of the frequencies of the elements in those ranges.

The dyadic ranges for elements in $[m]_1$ are defined as the sets of all ranges from $[x2^y + 1 \ldots (x+1)2^y]$ for all $y \in [\log m]_0$, $x \in \mathbb{N}^0$, and $(x+1)2^y \leq m$.

The idea of the dyadic ranges can also, and will in the remainder of this thesis, be referenced to as a tree. The analogy of a tree is where each y defines a new level with $y = 0$ being the level with the leafs and the height of the tree being $\log m$. The x values of a level are then seen as nodes. The nodes of a level is then indirectly connected with the nodes covering the same range at a level below. See Figure 4.1 for visual confirmation of this analogy.

For the heavy hitters problem a black box is associated with each of the $\log m$ levels of the dyadic ranges. For our case, this is in the form of a synopsis data structure (Definition 1) solving the frequency estimation problem, such as the Count-Min Sketch or Count-Median Sketch. The synopsis structures are used to make decisions on how to proceed when querying for heavy hitters, by holding information about the frequencies of the x ranges for each level of the tree. Since the x's on each level cover a certain range of the elements in the universe, the synopsis data structures provide an estimated range

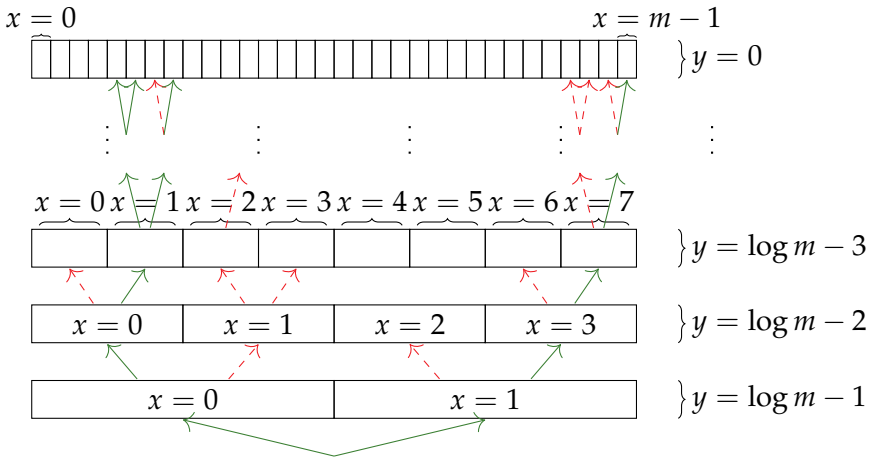


Figure 4.2: Example query of the hierarchical data structure to find the heavy hitters. An arrow represents a range that should be checked further. The solid (green) arrows represents *heavy* paths which should be followed in order to find all the heavy hitters. The dashed (red) arrows represent *light* paths, which – at some point – should be detected and stopped.

sum of the frequencies of the elements in that range.

Updates are performed for each of the synopsis data structures from the top of the tree to the leafs. Each sketch is updated with a frequency value $c_t \in \mathbb{R}$ associated with the x value of the dyadic range as the element. That is, replacing the element identifier i with the x that covers the specific element at each level. This is shown in Figure 4.1, where i is associated with $x = 0$ on level $y = \log m - 1$ and $x = 1$ on the next level, and so forth on the following levels.

When the structure is queried, we use the fact that the frequencies provided by the synopsis data structures for each x range, describe the sum of the frequencies of all elements covered by that range. The sum can then be compared to the heavy hitter threshold and if the sum is above the threshold, then that range is investigated on the level below, with a decreased y and new x ranges containing more concentrated sums. Such a query in the dyadic ranges can be done recursively using a parallel binary search or in an iterative stack-based approach.

An example of such a query is shown in Figure 4.2 where the arrows represent paths that need to be further investigated. The solid (green) arrows indicate a path with heavy hitters and the dashed (red) arrows indicate paths without heavy hitters. The paths without heavy hitters are necessary to investigate as the sums of the elements below are above the threshold but should be stopped as the ranges are further divided implying a lower range sum. The nodes that the arrows come from, can be described as *heavy* nodes whereas nodes with no outgoing arrows are *light* nodes.

The concept of *heavy* and *light* nodes are used to describe the mass/frequency of the nodes. For each node in the tree of dyadic ranges, a *heavy* node is defined as a node for which its mass exceeds the heavy hitter threshold of $\phi \|v\|_1$. All other nodes are defined

as *light* nodes.

The nodes of the tree each cover a range of elements in the universe, with the mass being the sum of the frequencies of the covered elements. For higher levels in the tree the frequencies are sums of the level below, resulting in increasing values as the level increases. This ensures that all *heavy* elements in the leafs are connected through other *heavy* internal nodes, all the way to the root. Likewise, the path from the root to a *heavy* leaf node – an element – is called a *heavy* path. This notion of *heavy* and *light* nodes will be used to explain some of the algorithms in the following sections.

As a consequence of the $\log m$ levels with synopsis data structures, the space usage of the hierarchical data structure will be in the order of $\log m$ times the space of such a synopsis data structure. Any update will have to update each of the $\log m$ levels. This will take time $\log m$ times the update time of the synopsis data structure.

Finally the query will have to traverse all $\log m$ levels where an upper bound of ϕ^{-1} queries – assuming not too many erroneous paths are followed – are performed on each level, since this is the upper bound of the amount of heavy hitters. This upper bound holds, since the range sums of the frequency vector are only divided into more nodes, this does not change the L_1-norm of the vector and hence, no more than ϕ^{-1} elements have a mass of more than $\phi \|v\|_1$ on each level. The time of such a query will hence be $O\left(\log m/\phi\right)$ times the time to perform a point query in the synopsis data structure.

Instead of having a binary tree of dyadic ranges, any branching factor $k \leq m$ of the tree could in general be chosen, as long as the ranges covering each level are chosen according to k instead of the dyadic power of two. Such a change would imply that the update time of the tree would be improved to $\log_k m$ times the update time of the synopsis data structure, while a trade-off would have to be made according to the query time, due to the branching factor resulting in extra nodes to check. The space would generally also benefit from an increasing branching factor.

In practice, for a hierarchical data structure as described above, one would substitute the synopsis data structures with exact counts for the first $l-1$ levels of the tree, where the l'th level is the level where an exact count consumes more memory than the synopsis data structure. Doing this would improve the guarantees of the queries since the traversal will be determined on exact range sums instead of estimates. Furthermore, a point query to the exact structure would generally be faster than one to a synopsis data structure.

In the next 3 sections, different sketches and uses of them will be presented to actually allow for finding approximate heavy hitters within the guarantees presented in Section 4.1, and using the hierarchical data structure presented in this section.

4.4 Hierarchical Count-Min Structure

A solution to the approximate L_1 heavy hitters problem in the *Strict Turnstile Model*, can be created using the Count-Min Sketch [9] from Section 3.3 as the synopsis data structure for the hierarchical data structure from Section 4.3.

The L_1 guarantee comes from the estimated frequency provided by a Count-Min Sketch according to the L_1-norm. Such a solution would in general provide very good

space, update and query bounds, which will be close to those stated earlier for the randomized solution in the *Cash Register Model*, as should be clear from the following theorem.

Theorem 3 (Hierarchical Count-Min Sketch structure bounds)**.** *Using* $O\left(\epsilon^{-1}\log\frac{\log m}{\delta'\phi}\log m\right)$ *words of space, updates on the hierarchical data structure can be carried out in* $O\left(\log\frac{\log m}{\delta'\phi}\log m\right)$ *time, and approximate* L_1 *heavy hitters queries in* $O\left(\log\frac{\log m}{\delta'\phi}\log m\right)$ *time, where*

1. *element* s_i *is output if* $v_i \geq \phi \|v\|_1$*, and*
2. *with probability* $1-\delta'$*, no element* s_i *with frequency* $v_i < (\phi-\epsilon)\|v\|_1$ *is output*

with a branching factor of the hierarchical data structure of $k = 2$*.*

Proof. As the underlying Count-Min Sketches never underestimate the frequencies by Theorem 1, (1.) is trivially satisfied for the last sketch of the dyadic ranges. For the intermediate sketches, the estimated frequencies are the sums of the frequencies of the elements spanned by the range, which likewise will not underestimate. Thus, all sketches satisfy (1.) enabling us to follow the *heavy* paths in the hierarchical data structure.

The number of queries performed on the dyadic ranges in each of the $\log m$ levels, is bounded by the maximum amount of true heavy hitters on each level. Since the L_1-norm of a vector does not change on each level, this is at most ϕ^{-1} queries for each of the $\log m$ levels. Due to the branching factor, at most a factor of $k = 2$ more point queries are carried out and when all estimates of those queries are correct with high probability, there is a total of $\frac{2\log m}{\phi}$ point queries.

Let E be the set of failed events E_i over the point queries above. For an event to fail it must hold that the estimated frequencies of the events is $\hat{v}_i > v_i + \epsilon\|v\|_1$. The probability of failure for any of the $\frac{2\log m}{\phi}$ point queries, can be expressed using the Union Bound.

$$\begin{aligned}
\mathbb{P}[E] &= \mathbb{P}\left[\bigcup_{i=1}^{\frac{2\log m}{\phi}} E_i\right] \\
&\leq \sum_{i=1}^{\frac{2\log m}{\phi}} \mathbb{P}\left[\hat{v}_i > v_i + \epsilon\|v\|_1\right] && \text{(Union Bound)} \\
&\leq \sum_{i=1}^{\frac{2\log m}{\phi}} \delta \\
&= \frac{2\log m}{\phi}\delta
\end{aligned}$$

Hence, scaling δ for all sketches of the tree to $\frac{\delta'\phi}{2\log m}$, gives us (2.).

The space usage, update time, and query time then follows directly, since a Count-Min Sketch with failure probability δ is associated with each of the $\log m$ levels. □

The binary tree of Theorem 3 can generally be changed to a k-ary tree. Such a change would improve the update time to $O\left(\log \frac{k \log_k m}{\delta' \phi} \log_k m\right)$ by trading off the query time, which would become $O\left(k \log \frac{k \log_k m}{\delta' \phi} \log_k m\right)$. The space of the structure would likewise be improved to $O\left(\epsilon^{-1} \log \frac{k \log_k m}{\delta' \phi} \log_k m\right)$ words.

4.5 Hierarchical Count-Median Structure

This section describes how the approximate L_1 heavy hitters problem is solved using the Count-Median Sketch [5] from Section 3.4 as the synopsis data structure for the hierarchical data structure in Section 4.3.

The Count-Median Sketch provides its estimates with guarantees according to the L_2-norm, which for all distributions is at least as good, and for many significantly stronger than for the L_1-norm, since $\sqrt{\|v\|_1} \leq \|v\|_2 \leq \|v\|_1$. As a consequence it is quite easy to use the Count-Median Sketch as we used the Count-Min Sketch in Section 4.4 to have an algorithm that finds approximate L_1 heavy hitters with very good error guarantees for each of the estimates.

To use the Count-Median Sketch a minor tweak must be made to fit the definition of an approximate L_1 heavy hitters algorithm. Since the Count-Median Sketch has a two-sided error (Theorem 2) i.e. it both underestimates and overestimates its frequencies, and thus, we must adjust the threshold for which an L_1 heavy hitters is expected, such that all true heavy hitters are always found. This and the analysis of the algorithm is carried out in the following Theorem 4.

Theorem 4 (Hierarchical Count-Median Sketch structure bounds)**.** *An approximate L_1 heavy hitters solution using Count-Median Sketches, uses $O\left(\epsilon^{-2} \ln \frac{\log m}{(\phi-\epsilon)\delta'} \log m\right)$ words of space, such that updates and queries on the hierarchical data structure can be carried out in $O\left(\ln \frac{\log m}{(\phi-\epsilon)\delta'} \log m\right)$ time and such that approximate L_1 heavy hitters queries report:*

1. *element s_i for which $v_i \geq \phi \|v\|_1$, and*
2. *with probability $1-\delta'$, no element s_i with frequency $v_i < \phi \|v\|_1 - 2\epsilon \|v\|_2$*

with a branching factor of the hierarchical data structure of $k = 2$.

Proof. Because of the two-sided error of the Count-Median Sketch, the threshold used to compare frequencies from the sketches needs to be adjusted. Such an adjustment is needed to compensate for underestimated frequencies, in order to support (1.).

Assume that element s_i with true frequency v_i is a heavy hitters, then $v_i \geq \phi \|v\|_1$ and it holds that

$$|\hat{v}_i - v_i| \leq \epsilon \|v\|_2 \;\Rightarrow\; \phi \|v\|_1 - \epsilon \|v\|_2 \leq \hat{v}_i \leq \phi \|v\|_1 + \epsilon \|v\|_2$$

with probability $1-\delta$ our estimated frequency is at most $\epsilon \|v\|_2$ less than the actual frequency. This implies that some heavy hitters will only be reported with probability $1-\delta$, which does not satisfy the approximate L_1 heavy hitters definition.

Changing the threshold for our sketch queries to $\phi \|v\|_1 - \epsilon \|v\|_2$ makes the definition hold up to a constant factor since all elements with true frequency $v_i > \phi \|v\|_1$ will be reported, while no elements with true frequency $v_i < \phi \|v\|_1 - 2\epsilon \|v\|_2 \leq (\phi - 2\epsilon) \|v\|_1$ will be reported with probability $1 - \delta$.

The change of the threshold when querying the tree enables us to use more or less the same arguments as in Theorem 3 to prove the theorem with only a few adjustments.

The estimations of the dyadic ranges now have errors according to the L_2-norm. Since the L_2-norm of different dyadic ranges of the same vector generally differs, the guarantee is generally different for each level of the tree. However, this is not a problem, since the L_2-norm of any of the dyadic ranges never exceeds the L_1-norm. Hence, all estimates are generally stronger than those provided by the Count-Min Sketch. By adjusting the threshold from above according to the L_2-norm of the dyadic range of a given level in the tree we get the exact guarantees of the theorem. Thus, we can still use the fact that if all queries does not fail, at most $\frac{2 \log m}{\phi}$ queries are carried out through the traversal of the tree.

But the above misses a detail. A change in the threshold requires a rescaling of δ, as the upper bound on the number of heavy hitters changes when the threshold changes. Now, at most $\frac{1}{\phi - \epsilon}$ heavy hitters exist at each level, which means that the amount of nodes visited on each level of the hierarchical data structure is at most two times that, i.e. $\frac{2}{\phi - \epsilon}$. Since there are $\log m$ levels where we perform at most that many point queries, we need to rescale the δ of all the sketches and by the same Union Bound argument as in Theorem 3 this gives us $\delta = \frac{(\phi - \epsilon)\delta'}{2 \log m}$.

We have shown (2.) as no elements with true frequency $v_i < (\phi - 2\epsilon) \|v\|_1$ will be returned in the resulting set with probability $1 - \delta'$. □

In practice, the L_2-norm of each of the levels in the tree will be hard to compute and one could instead choose the new threshold to be $(\phi - \epsilon) \|v\|_1$ which increases the amount of false positives for most distributions, but would be an approximate L_1 heavy hitters solution up to a constant factor. Empirical studies [6, 7] have shown, though, that the adjustment of the threshold generally is not needed in order to provide as good results as the solution from Theorem 3 in practice.

Compared to the previous section, this structure uses more space than the algorithm using Count-Min Sketches. The difference in space is because of the width where a Count-Min Sketch has a width of $O\left(\epsilon^{-1}\right)$ compared to $O\left(\epsilon^{-2}\right)$ of the Count-Median Sketch.

Such a difference in size will in practice make the Count-Min Sketch preferable since it will use less space than the Count-Median Sketch. In the experiments presented in Chapter 6 we will present a way to cope with this difference in space and error guarantees.

As with the solution using the Count-Min Sketch, this solution can also be changed to a k-ary tree instead of the binary presented above. Such a change will again provide a better space usage and update time by trading off with a worse query time.

As a final note, it was proven in Kane et al. [19], that the Count-Median Sketch and the

hierarchical data structure can be used to solve any approximate heavy hitters problem for norm $0 < p \leq 2$.

4.6 Hierarchical Constant-Min Structure

This section will describe a similar but slightly different approach than the Hierarchical Count-Min Structure from Section 4.4. The data structure is from Larsen et al. [22] and it also solves the approximate L_1 heavy hitters problem in the *Strict Turnstile Model*. The Count-Min Sketch is still used as a black box structure internally, but the analysis is changed from providing worst case guarantees for space, updates and queries to expected guarantees for queries, while maintaining worst case guarantees for space and updates.

The previous sections looked at the desired heavy hitters guarantee and then scaled the failure probability of the sketches to match these guarantees. This algorithm will instead choose a constant failure probability $\delta = 1/4$ for all internal sketches in the tree.

A constant probability of failure for each point query at each level of the hierarchical structure should produce more unwanted paths, since nodes in the tree will have a tendency to have a too heavily overestimated frequency. As a consequence, more nodes have to be queried in the tree.

Such unwanted paths may lead all the way down to the leafs, which requires a Count-Min Sketch with a stronger guarantee, than the constant ones, at the leaf level. Such a sketch is constructed by appropriately scaling the Count-Min Sketch according to the expected amount of queries performed on the leaf level. Thus, one is able to discard the wrong paths and incorrect reporting. The final scaled sketch will be kept externally from the tree, but we will reference to it as the bottom sketch. In Theorem 5 below, d is the depth of the internal sketches, d' is equal to the depth of the bottom scaled Count-Min Sketch, m is the amount of unique elements in the vector v and $\delta = 1/4$ is a constant used to calculate d, the depth of each of the internal sketches in the tree.

Theorem 5 (Hierarchical Constant-Min Structure bounds)**.** *Choosing δ to be a constant, $\delta = 1/4$, for all internal sketches imply that the depth of each sketch becomes constant and gives a data structure that solves the approximate L_1 heavy hitters problem using $O\left(\epsilon^{-1} \log m + \epsilon^{-1} d'\right)$ words of space, where updates on the hierarchical structure can be carried out in $O\left(\log m + d'\right)$ time and approximate heavy hitters queries can be performed, where*

1. *All elements s_i with frequency $v_i \geq \phi \|v\|_1$ are output, and*
2. *with probability $1 - \delta'$, no element s_i with frequency $v_i < (\phi - \epsilon) \|v\|_1$ is output*

using expected $O\left(\log(m)\, \phi^{-1} + d'\right)$ time. Where $d' = O\left(\log\left(\delta'^{-1}\phi^{-1}\right)\right)$ denotes the depth of the bottom sketch and the branching factor of the hierarchical data structure is $k = 2$.

Proof. The analysis is split in two. First, we will analyze the expected amount of work necessary in the hierarchical data structure and secondly, we will analyze the additional work introduced by the extra sketch on the leaf level.

To bound the expected amount of work performed in the hierarchical data structure, one needs to bound the amount of visited nodes i.e. sketch queries performed over the whole structure.

Since the tree of dyadic ranges is built such that a higher level of the tree contains the mass of the lower levels (Section 4.3), we are guaranteed that all *heavy* nodes are visited, performing the exact same traversal as the one in Section 4.4. By the definition of true heavy hitters there can exist at most ϕ^{-1} such nodes at each level – since the L_1-norm of the dyadic ranges stay the same – giving at most $\log(m)\ \phi^{-1}$ *heavy* nodes.

The crucial point is to bound the amount of *light* nodes visited when the sketches on each level allow errors with probability $\delta = 1/4$, that is, we will show that the visits to the *heavy* nodes will pay for the visits to the *light* ones.

We define the distance of the nearest *heavy* ancestor of a node j – $\texttt{nha}(j)$ – to be the amount of upwards traversals in the tree to find a *heavy* ancestor node, which implies for any node j that $1 \leq \texttt{nha}(j) < \log m$.

Let $I_{i,j}$ define an indicator variable, describing whether a node i is visited when having its nearest *heavy* ancestor j levels above, for $1 \leq i \leq 2m-1$ and $1 \leq j \leq \log m - 1$.

$$I_{i,j} = \begin{cases} 1 & \text{if node } i \text{ is visited and } \texttt{nha}(i) = j \\ 0 & \text{o.w.} \end{cases}$$

To bound the expectation of $I_{i,j}$ we note that the probability of the sketch on each level to overestimate a *light* node is $\delta = 1/4$. We further note that each level of the tree is independent by the definition of the dyadic ranges, and for a node i to be visited it must hold that all ancestors of i have been visited. As a consequence the probability of visiting node i is the product of the probability of visiting its nearest *heavy* ancestor, and the probability of visiting each of the *light* nodes on the path to the *heavy* ancestor. We therefore get

$$\mathbb{E}\left[I_{i,j}\right] \leq 1 * \prod_{j-1} \delta = \delta^{\texttt{nha}(i)-1} = \frac{1}{4}^{\texttt{nha}(i)-1}.$$

Let Y_i be the random indicator variable indicating if a node $1 \leq i \leq 2m-1$ is visited in the hierarchical data structure and let $Y = \sum_{i=1}^{2m-1} Y_i = \sum_i Y_i$. Since there exists at most ϕ^{-1} queries of *heavy* nodes on each of the $\log m$ levels in the whole structure and queries on *light* implies that such a node must have a *heavy* ancestor, we can bound the probability of reaching any node i by looking at the depth of node i in the tree.

$$\begin{aligned} \mathbb{E}[Y] &= \mathbb{E}\left[\sum_i Y_i\right] \\ &= \sum_i \mathbb{E}[Y_i] && \text{(Linearity of Expectation)} \\ &= \sum_{j=1}^{\log m - 1} \sum_{\{i \mid \texttt{nha}(i) = j\}} \mathbb{E}\left[I_{i,j}\right] && \text{(Redefinition of above)} \end{aligned}$$

$$
\begin{aligned}
&\leq \sum_{j=1}^{\log m - 1} \sum_{\{i \mid \mathtt{nha}(i) = j\}} \delta^{j-1} && \text{(Substitution)} \\
&\leq \sum_{j=1}^{\log m - 1} \frac{\log m}{\phi} 2^j \delta^{j-1} && (4.1) \\
&= \frac{\log m}{\phi} \sum_{j=1}^{\log m - 1} 2^j \delta^{j-1} \\
&= \frac{2 \log m}{\phi} \sum_{j=1}^{\log m - 1} 2^{j-1} \frac{1}{4}^{j-1} && \left(\text{Using the constant } \delta = \frac{1}{4}\right) \\
&= \frac{2 \log m}{\phi} \sum_{j=1}^{\log m - 1} \frac{1}{2}^{j-1} \\
&= O\left(\frac{\log m}{\phi}\right)
\end{aligned}
$$

where (4.1) comes from the fact that a node has 2^j descendants j levels below it and that the maximum number of heavy hitters at a level is ϕ^{-1} for all $\log m$ levels. The last step comes from the geometric series

$$
\sum_{j=1}^{\log m - 1} \frac{1}{2}^{j-1} \leq \sum_{j=0}^{\infty} \frac{1}{2}^{j} = 2
$$

and obtains a bound on the expected number of visited nodes when performing a heavy hitters query.

The second part of the analysis is to bound the time used on the extra Count-Min Sketch when all heavy hitter candidates are found. The extra sketch is needed since we at the final level of the tree will have introduced some extra candidates due to the constant error probability of all internal sketches in the tree.

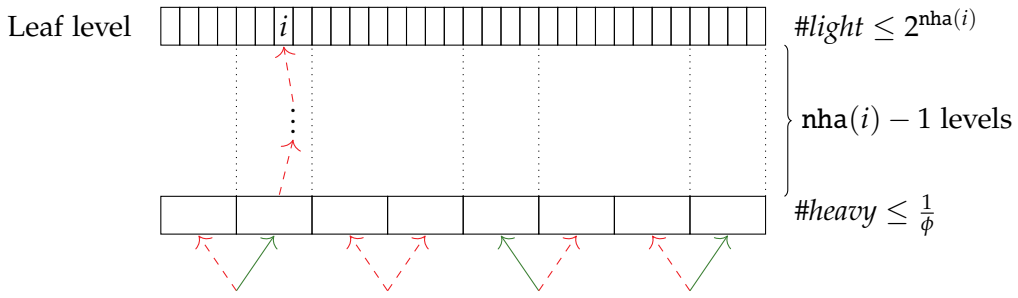


Figure 4.3: Illustration showing the line of thought from the point of view of leaf i, when arguing for the amount of candidates of a Hierarchical Constant-Min Structure's bottom sketch. The illustration shows the limits on the number of *heavy* and *light* nodes at the nearest ancestor level and the leaf level, respectively.

To bound the amount of candidates at the leaf level, we reuse some of the ideas from the analysis of the query above. Figure 4.3 shows a depiction of the line of thought. Consider a specific *light* leaf node i. For i to be a leaf candidate it must hold that all ancestor nodes of i have had estimated frequencies above the heavy hitter threshold, and $\texttt{nha}(i) - 1$ of these are *light* nodes by the definition of the nearest *heavy* ancestor. Furthermore, there can be at most ϕ^{-1} of such *heavy* ancestors that are $\texttt{nha}(i)$ levels above the leaf candidates and each of these can contribute with at most $2^{\texttt{nha}(i)}$ leaf queries, since this is an upper bound on the amount of leafs in a subtree rooted at such a *heavy* node.

Compared to the analysis of the queries over the whole tree, the analysis of queries in the bottom sketch is fixed to the leaf level. This removes the $\log m$ factor, as there can be only one level for which any leaf candidate i can have $\texttt{nha}(j)$ to a *heavy* node j in the tree. This also explains (4.2) of the equation below.

Let X_i be an indicator variable stating whether node i is a leaf candidate for $i = \{m, \dots 2m-1\}$ i.e. for i being a leaf node. Let $X = \sum_{i=m}^{2m-1} X_i = \sum_i X_i$. We can then expect over the amount of queries performed on the bottom sketch by looking at the expectation of leaf candidates.

$$
\begin{aligned}
\mathbb{E}[X] &= \mathbb{E}\left[\sum_i X_i\right] \\
&= \sum_i \mathbb{E}[X_i] && \text{(Linearity of Expectation)} \\
&= \sum_{j=1}^{\log m - 1} \sum_{i \in \{m,\dots,2m-1\} : \texttt{nha}(i) = j} \mathbb{E}\left[I_{i,j}\right] && \text{(Redefinition of above)} \\
&= \sum_{j=1}^{\log m - 1} \sum_{i \in \{m,\dots,2m-1\} : \texttt{nha}(i) = j} \delta^{j-1} && \text{(Substitution)} \\
&\leq \sum_{j=1}^{\log m - 1} \frac{1}{\phi} 2^j \delta^{j-1} && (4.2) \\
&= \frac{1}{\phi} \sum_{j=1}^{\log m - 1} \delta^{j-1} 2^j \\
&= \frac{2}{\phi} \sum_{j=1}^{\log m - 1} \delta^{j-1} 2^{j-1} \\
&= \frac{2}{\phi} \sum_{j=1}^{\log m - 1} \frac{1}{4}^{j-1} 2^{j-1} && \left(\delta = \frac{1}{4}\right) \\
&= \frac{4}{\phi}
\end{aligned}
$$

where the last step comes from the same geometric series as above.

From the expectation of X we get that the expected number of nodes visited in the leaf level are $\frac{4}{\phi}$. Still, the actual amount of nodes can be much higher, since this is only in expectation. By Markov's Inequality we can bound the amount of nodes visited with good probability.

$$\begin{aligned}\mathbb{P}\left[X > t\mathbb{E}\left[X\right]\right] < \frac{1}{t} &\Rightarrow \mathbb{P}\left[X > \frac{4t}{\phi}\right] < \frac{1}{t} \\ &\Rightarrow \mathbb{P}\left[X > \frac{8}{\delta'\phi}\right] < \frac{\delta'}{2} \qquad \left(t = \frac{2}{\delta'}\right)\end{aligned}$$

Hence, by Markov's Inequality we get that the probability of having more than $\frac{8}{\delta'\phi}$ leaf nodes to query is less than $\frac{\delta'}{2}$.

Next, we can bound the probability of having even a single estimation error in the bottom sketch, conditioned on the fact that no more than $\frac{8}{\delta'\phi}$ queries is performed.

Let A be the event that $X \leq t\mathbb{E}\left[X\right]$ and let B be the event that one or more estimation errors occur in the final sketch when A is upheld. We can then calculate the probability of B by doing a Union Bound over the maximal amount of queries with good probability conditioned from A.

$$\begin{aligned}\mathbb{P}\left[B|A\right] &= \bigcup_{i=1}^{\frac{8}{\delta'\phi}} \mathbb{P}\left[\hat{v}_i > v_i + \epsilon \left\|v\right\|_1\right] \\ &\leq \sum_{i=1}^{\frac{8}{\delta'\phi}} \mathbb{P}\left[\hat{v}_i > v_i + \epsilon \left\|v\right\|_1\right] \\ &\leq \sum_{i=1}^{\frac{8}{\delta'\phi}} \delta \\ &= \frac{8\delta}{\delta'\phi} \\ &= \frac{\delta'}{2} \qquad \left(\delta = \frac{(\delta')^2\phi}{16}\right)\end{aligned}$$

By scaling δ of the bottom sketch to $\frac{(\delta')^2\phi}{16}$ we add more rows to the sketch and get that with probability $\frac{\delta'}{2}$ none of the point queries on the sketch will fail.

The final thing to do is to bound the overall probability of a bad estimate to occur in the bottom sketch. Here we assume that if more than $\frac{8}{\delta'\phi}$ queries are performed on the bottom sketch, it will always fail. The probability of a bad estimate can then be stated as the probability of having more than $\frac{8}{\delta'\phi}$ nodes to query at the leaf level or the probability of the sketch to fail, when at most $\frac{8}{\delta'\phi}$ queries are performed.

$$\mathbb{P}\left[\neg A \cup (B \cap A)\right] \;=\; \mathbb{P}\left[\neg A\right] + \mathbb{P}\left[B|A\right]\mathbb{P}\left[A\right] \;\leq\; \frac{\delta'}{2} + \frac{\delta'}{2}\left(1 - \frac{\delta'}{2}\right) \;\leq\; \delta'$$

We then end up with a bottom sketch having depth $d' = \log \frac{16}{(\delta')^2 \phi}$, which with probability $1 - \delta'$ provides successful estimates of all the queries performed on it. This guarantees that with probability $1 - \delta'$, no elements s_i with estimates $v_i < (\phi - \epsilon) \|v\|_1$ is present in the resulting set of heavy hitters, which proves this theorem.

The space usage, update and query times then follow directly from the bottom sketch size $O\left(\epsilon^{-1}\ d'\right)$ and the hierarchical data structure size $O\left(\epsilon^{-1}\ \log m\right)$. □

Hence, we get an algorithm providing approximate L_1 heavy hitters with better space usage, update time, and expected query time compared to the algorithms described in the previous two sections. What is just as fortunate, is the fact that the algorithm only hides small constants in the big-O notation and as a consequence should perform well in practice.

The only downside to this solution is the fact that changing the branching factor k for the hierarchical data structure does not imply the same trade-offs as is present for the earlier algorithms. This is due to the fact that the constant δ must be scaled appropriately according to k in order for the algorithm to remain true.

4.7 Summary

This chapter mentioned how different deterministic and randomized algorithms solve the approximate L_1 heavy hitters problem in the *Cash Register Model*.

It was then shown and proved how the L_1 heavy hitters problem can be approximated in the *Strict Turnstile Model* using a hierarchical data structure and sketches solving the frequency estimation problem. Generally, such solutions have fast queries with the trade-off of having to store and update $\log m$ sketches. The Hierarchical Constant-Min Structure recently introduced in literature [22], was also analyzed and shown to be faster than earlier solutions using Count-Min Sketches and Count-Median Sketches, by removing a $\log m$ factor. It will be interesting to see how this structure compares to other data structures in practice.

Another interesting aspect of all the algorithms and data structures introduced in this section is how their bounds compare to the lower bounds. This comparison will be investigated next, in Chapter 5.

Chapter 5

Lower Bounds

In the following sections we present and prove space and update lower bounds for the approximate L_1 heavy hitters problem and the frequency estimation problem in the *Strict Turnstile Model*.

First, we prove a space lower bound for the problem of finding approximate heavy hitters. This lower bound is compared with the solutions presented in this thesis.

Next, we prove a space lower bound for the problem of frequency estimation using sketch data structures supporting point queries. Again, we will compare the space lower bound with the bounds of the previously presented algorithms.

Finally, we will present an update lower bound for the approximate L_1 heavy hitters problem and the frequency estimation problem supporting point queries. These lower bounds are also compared to the upper bounds of the presented algorithms.

5.1 Space Lower Bound for Approximate Heavy Hitters

The space lower bound of the heavy hitters problem follows from the augmented indexing problem. The augmented indexing problem is defined as a game as follows: Let l and k be positive integers. The first player, Alice, is given a symbol $y \in [k]_0^l$, while the second player, Bob, is given an integer $i \in [l]_0$ and symbols y_j for all $j < i$. The purpose of the game is for Bob to output the value of y_i when only receiving a single message from Alice. An important note here is that Alice has no knowledge of the integer i, Bob received.

We need the following lemma from Jowhari et al. [18], Miltersen et al. [25] to show the heavy hitters lower bound.

Lemma 8 (Augmented indexing problem bound)**.** *In any one-way protocol in the joint random source model with success probability at least* $1 - \delta > \frac{3}{2k}$*, Alice must send a message of size* $\Omega\left((1-\delta)l \log k\right)$.

Here the joint random source model states that all random choices are done with a public random string, known by all parties, and independent of the input. In the

following situation, this is equivalent to both parties using the same random seed for the hash functions.

Using the above lemma and reducing the augmented indexing problem to the heavy hitters problem, we are able to play the above game in such a way that a space lower bound on the heavy hitters problem can be shown.

Theorem 6 (Heavy hitters space lower bound)**.** *Let $p \in \{1,2\}$ define the L_p-norm and $\phi \in (0,1/2)$ define the threshold parameter. Any one-pass heavy hitters algorithm in the* Strict Turnstile Model *must use $\Omega\left(\phi^{-p}\log^2 m\right)$ bits of space where m defines the amount of elements used to form the frequency vector x, and the heavy hitters algorithm is guaranteed to only have a constant failure probability.*

Proof. Alice and Bob want to solve the augmented indexing problem. Alice is given the values $y_0,\ldots,y_{s-1} \in \left[2^t\right]_0$, while Bob is given an integer $i \in [s]_0$ and y_j for all $j < i$. In order for both to solve the problem, they will reduce the problem to the heavy hitters problem.

They now construct the vectors u and v as follows: for $b = (1-(2\phi)^p)^{-\frac{1}{p}}$, Alice creates the vector u, such that u consists of s concatenated blocks of size 2^t where the y_k'th index of the k'th block is equal to $\lceil b^{s-k}\rceil$ for $k \in [s]_0$ and all other values in the block are 0. As a consequence, the dimension of the vector u is $\dim(u) = s2^t$. Bob creates the vector v in the exact same way, except that the last $s-i$ blocks are set to 0 such that $\dim(v) = \dim(u) = s2^t$. This is done since $y_i,\ldots,y_{s-1}$ are unknown to Bob.

To solve the augmented indexing problem, Alice and Bob then uses a heavy hitters algorithm for the vector $x = u - v$, which implies that the heavy hitters algorithm must solve the problem in the *Turnstile Model*. More specifically it solves the problem in the *Strict Turnstile Model*, since x is a valid vector in this model due to $b > 1$.

First, Alice performs updates using the heavy hitters algorithm to bring the initial zero vector $x \in \mathbb{Z}^m$ to $x = u$. Then, she sends the final memory content to Bob enabling him to use the same random seeds for the hash functions and maintain a copy of the memory content. Bob performs another round of updates by subtracting the vector v from u, that is, he brings $x = u$ to $x = u - v$. Finally, Bob performs a heavy hitters query on x giving him the set of heavy hitters Q from where he returns the value $z \in \left[2^t\right]_0$ if the smallest index in Q is equal to $(i-1)2^t + z$.

If an error does not occur in the heavy hitters algorithm, then $z = y_i$. This is because the first $(i-1)2^t$ entries of x zeroes due to the subtraction of the vectors. The only entries with values are $(k-1)2^t + y_k$ for $i \le k < s$ where the index calculated from $k = i$ is the minimum one. We must show that our heavy hitters algorithm will find this element by showing, $\lceil b^{s-i}\rceil \ge \phi \left\|x\right\|_p$, that the value of the elements is greater than the threshold of the algorithm.

$$\begin{aligned}
\phi^p \left\|x\right\|_p^p &= \phi^p \sum_{j=i}^{s-1} \left\lceil b^{s-j}\right\rceil^p \\
&< (2\phi)^p \sum_{j=i}^{s-1} (b^{s-j})^p && (\forall a \ge 1 : \lceil a\rceil < 2a)
\end{aligned}$$

$$
\begin{aligned}
&= (2\phi)^p \sum_{j=0}^{s-i} (b^j)^p \\
&= (2\phi)^p \frac{1 - b^{p(s-i+1)}}{1 - b^p} && (b \neq 1) \\
&= (2\phi)^p \frac{b^{p(s-i+1)} - 1}{b^p - 1} \\
&= b^{p(s-i)} \frac{(2\phi b)^p}{b^p - 1} - \frac{(2\phi)^p}{b^p - 1} \\
&= b^{p(s-i)} - \frac{1}{b^p - 1} && ((2\phi)^p = -\frac{1}{b^p} + 1) \\
&< b^{p(s-i)} && (b^p > 1)
\end{aligned}
$$

Now taking the pth root gives us $\lceil b^{s-i} \rceil > \phi \|x\|_p$, which proves that the augmented indexing problem can be solved if the approximate L_p heavy hitters algorithm is correct.

Choosing $s = \lceil (2\phi)^{-p} \log m \rceil$ and $t = \lceil \log m / 2 \rceil$ gives for a large enough m, that the dimension of x is covered by m i.e. $s2^t < m$ and the weights of each element x_i remains less than m. For a heavy hitters algorithm with failure probability $\delta < \frac{1}{2}$, Lemma 8 will provide a lower bound on the communication protocol of $\Omega(st) = \Omega\left(\phi^{-p} \log^2 m\right)$. This proves the theorem, since the size of the message sent from Alice to Bob is the same as the memory size of the heavy hitters algorithm. □

From Theorem 6 we get a space lower bound on algorithms solving the heavy hitters problem in the *Strict Turnstile Model*. The lower bound is expressed using the threshold parameter ϕ but generally, ϕ can be replaced with the approximation factor ϵ when ϵ is chosen to be a constant factor less than ϕ. This is due to the definition of the approximate heavy hitters problem in Section 4.1, which states that with probability $1 - \delta$, no elements with frequency $v_i < (\phi - \epsilon) \|v\|_p$ are returned for $\epsilon < \phi$ and all elements with frequency $v_i \geq \phi \|v\|_p$ are returned. We are hence able to compare the lower bound with solutions studied in this thesis.

First, lets go back to the sketches from Chapter 3. Assuming that the L_p-norm for $p \in \{1, 2\}$ can be generated or approximated using no more space than the Count-Min Sketch or the Count-Median Sketch, both sketches use optimal space for depth $d = C \log m$, where $C > 1$ is a constant to guarantee $|\hat{v}_i - v_i| \leq \epsilon \|v\|_p$ for all $i \in [m]_1$. That is, all estimates are not worse than the approximation factor times the L_p-norm with failure probability $\delta = m^{-(C-1)} = O\left(\frac{1}{poly(m)}\right)$. On the other hand, the query time would be bad since all elements in $[m]_1$ would have to be queried to figure out if they are heavy hitters.

To cope with the query time the data structures in Chapter 4 were introduced for the approximate L_1 heavy hitters problem. Both solutions from Section 4.4 and Section 4.5 introduce an extra $\log m$ factor to the space and update time of the algorithm, since a sketch must be maintained over $\log m$ levels. Hence, the solution using Count-Min

Sketches only becomes sub-optimal in space, while the solution using Count-Median Sketch is even worse due to the width of those sketches. The solution from Section 4.6 with constant work for each of the $\log m$ levels is optimal in space and furthermore provides a good update and expected query time.

In the next section we will make further use of Theorem 6 to prove a lower bound on sketches supporting point queries.

5.2 Space Lower Bound for Frequency Estimation

For the approximate heavy hitters problem Theorem 6 shows a space lower bound of $\Omega\left(\phi^{-p}\log^2 m\right) = \Omega\left(\epsilon^{-p}\log^2 m\right)$ bits or $\Omega\left(\epsilon^{-p}\log m\right)$ words, when ϵ is chosen to be a constant smaller than ϕ. Using this theorem we are able to lower bound the amount of space required for a sketch supporting point queries.

Theorem 7 (Sketch space lower bound)**.** *Let $p \in \{1,2\}$ and δ be the failure probability of a sketch. Since a space lower bound exists for the approximate L_p heavy hitters problem in the* Strict Turnstile Model *on $\Omega\left(\epsilon^{-p}\log m\right)$ words with error probability $\delta' = 1/poly(m) < 1/2$, a space lower bound exists on sketch data structures supporting point queries in the* Strict Turnstile Model *on $\Omega\left(\epsilon^{-p}\log \delta^{-1}\right)$ words.*

Proof. Let $V_1, \ldots, V_t$ be t independent sketches supporting point queries with failure probability δ, each using S space. Further, let $\hat{v}_i(j)$ be the estimate of element s_i from V_j.

To solve the approximate L_p heavy hitters problem over a stream of m unique elements, let all t sketches run on a stream such that the weights of each element are bounded by m. We can then solve the problem by querying all s_i in all of the sketches and take the median of the estimates: $\text{median}_j\ \hat{v}_i(j)$ for $i \in [m]_1, j \in [t]_1$. This will imply that we return all elements with frequency $v_i \geq \phi \left\|v\right\|_p$ and no elements with frequency $v_i < (\phi - \epsilon)\left\|v\right\|_p$ with probability $1 - \delta'$, as will be shown next.

Let $Y_{i,j}$ be an indicator variable, indicating whether a specific query to a specific sketch failed, defined as:

$$Y_{i,j} = \begin{cases} 1 & \text{if } |\hat{v}_i(j) - v_i| > \epsilon \left\|v\right\|_p \\ 0 & \text{otherwise} \end{cases}, \qquad Y_i = \sum_{j \in [t]_1} Y_{i,j} = \sum_j Y_{i,j}$$

It then follows directly from Linearity of Expectation and the Union Bound over all t sketches that $\mathbb{E}\left[Y_i\right] = t\delta$. Since we take the median of the estimated frequencies returned by the t sketches, at least $\left\lceil \frac{t}{2} \right\rceil$ of them must fail in order for the heavy hitters query to fail.

$$\begin{aligned}
&\mathbb{P}\left[|\text{median}_j\ \hat{v}_i(j) - v_i| > \epsilon \left\|v\right\|_p\right] \\
&\leq \mathbb{P}\left[Y_i > \left\lceil \frac{t}{2} \right\rceil - 1\right] \\
&\approx \mathbb{P}\left[Y_i > \frac{\delta^{-1}}{2}\mathbb{E}\left[Y_i\right]\right] && \text{(Substitution)}
\end{aligned}$$

$$
\begin{aligned}
&= \mathbb{P}\left[Y_i > \left(1 + \left(\frac{\delta^{-1}}{2} - 1\right)\right)\mathbb{E}\left[Y_i\right]\right] \\
&< \left(\frac{e^{\frac{\delta^{-1}}{2}-1}}{\left(\frac{\delta^{-1}}{2}\right)^{\frac{\delta^{-1}}{2}}}\right)^{\mathbb{E}[Y_i]} && \text{(Chernoff Bound)} \\
&< \left(\frac{e^{\frac{\delta^{-1}}{2}}}{\left(\frac{\delta^{-1}}{2}\right)^{\frac{\delta^{-1}}{2}}}\right)^{\mathbb{E}[Y_i]} && \left(e^{\frac{\delta^{-1}}{2}-1} \geq 1 \wedge \left(\frac{\delta^{-1}}{2}\right)^{\frac{\delta^{-1}}{2}} \geq 1\right) \\
&= \left(\frac{1}{\left(\frac{\delta^{-1}}{2e}\right)^{\frac{\delta^{-1}}{2}}}\right)^{\mathbb{E}[Y_i]} \\
&= e^{-\ln\left(\frac{\delta^{-1}}{2e}\right)\frac{\delta^{-1}}{2}\mathbb{E}[Y_i]} \\
&= e^{-\ln\left(\frac{\delta^{-1}}{2e}\right)\frac{t}{2}} && \text{(Substitute } \mathbb{E}\left[Y_i\right]\text{)} \\
&= e^{-\left(\ln\delta^{-1} - \ln(2e)\right)\frac{t}{2}} \\
&< e^{-\left(\frac{t}{2}\ln\delta^{-1} - t\right)} \\
&= e^{-\left(\frac{\ln m}{2} - \frac{\ln m}{\ln\delta^{-1}}\right)} && \left(t = \frac{\log m}{\log\delta^{-1}} = \frac{\ln m}{\ln\delta^{-1}}\right) \\
&= e^{-\ln m\left(\frac{1}{2} - \frac{1}{\ln\delta^{-1}}\right)} \\
&= \frac{1}{m^{\left(\frac{1}{2} - \frac{1}{\ln\delta^{-1}}\right)}} \\
&= O\left(\frac{1}{poly(m)}\right)
\end{aligned}
$$

Hence, for an appropriate choice of δ and $t = \log m / \log \delta^{-1}$ sketches, the error probability on any of the m estimates is at most $O\left(\frac{1}{poly(m)}\right)$.

Looking at the space used to support the heavy hitters query, we use t sketches each with S space. Since we know the space lower bound of the approximate L_p heavy hitters problem in the *Strict Turnstile Model* we can set up an equation stating the minimal space usage of a point query.

$$
\begin{aligned}
tS \geq \epsilon^{-p}\log m \Rightarrow S &\geq \frac{\epsilon^{-p}\ \log m}{t} \\
&\Rightarrow S \geq \frac{\epsilon^{-p}\ \log m}{\frac{\log m}{\log\delta^{-1}}} \\
&\Rightarrow S \geq \epsilon^{-p}\ \log\delta^{-1}
\end{aligned}
$$

To solve the approximate heavy hitters problem using sketches that support point queries, it requires the sketches to use $\Omega\left(\epsilon^{-p} \log \delta^{-1}\right)$ words of space. □

Having proved the space lower bound for sketches supporting point queries with norm guarantees for both the L_1 and L_2-norm, we are able to compare it with the sketches presented in Chapter 3.

The Count-Min Sketch uses space $O\left(\epsilon^{-1} \log \delta^{-1}\right)$ words to support queries for which estimates are no worse than $\hat{v}_i \leq v_i + \epsilon \|v\|_1$ with probability at least $1 - \delta$. This clearly matches the terms of the lower bound from above and we can hence conclude that the Count-Min Sketch provides estimates with an error according to the L_1-norm in optimal space.

The Count-Median Sketch uses $O\left(\epsilon^{-2} \log \delta^{-1}\right)$ words to support queries for which estimates are no worse than $|\hat{v}_i - v_i| \leq \epsilon \|v\|_2$ with probability at least $1 - \delta$. This again matches the lower bound, as the Count-Median Sketch provides estimates with an error according to the L_2-norm in optimal space.

It is then easy to conclude that space-wise the sketches presented and analyzed in Chapter 3 are as good as they get for randomized structures. When it comes to the update times of the same sketches the bounds are not near as exciting, as we will see in the next section.

5.3 Update Lower Bounds

The first non-trivial update time lower bounds for randomized streaming algorithms in the *Turnstile Model* was provided in Larsen et al. [21]. In their work, only a certain restricted class of randomized streaming algorithms, namely those that are *non-adaptive* could be bounded. The definition of a *non-adaptive* randomized streaming algorithm is:

Definition 2 (Non-adaptive Randomized Streaming Algorithm)**.** *A non-adaptive randomized streaming algorithm, is an algorithm where*

- *it may toss random coins before processing any elements of the stream, and*
- *the words read from and written to memory are completely determined by the index of the updated element and the initially tossed coins, on any update operation.*

Such constraints imply that memory must not be read or written to based on the current state of the memory, but only according to the coins and the index.

Comparing the above definition to the sketches, a hash function chosen independently from any desired hash family can emulate these coins, enabling the update algorithm to find some specific words of memory to update using only the hash function and the index of the element to update.

This makes the *non-adaptive* restriction fit exactly with all of the *Turnstile Model* algorithms presented earlier. Hence, we can compare the update times of all those algorithms with the update time lower bound of the following theorems (Theorem 8 and Theorem 9) from Larsen et al. [21].

Theorem 8 (Point Query update lower bound)**.** *Any randomized non-adaptive streaming algorithm for point query must have worst case update time*

$$t_u = \Omega\left(\frac{\log \delta^{-1}}{\sqrt{\log(m)\log\left(\frac{eS}{t_u}\right)}}\right)$$

where S is the space of the algorithm.

Both the Count-Min Sketch and the Count-Median Sketch are *non-adaptive* and support point queries. Comparing the update times for both sketches of $O(d) = O\left(\log \delta^{-1}\right)$ with the lower bound from Theorem 8, we get that the update times of the sketches are only near-optimal. A gap still exists in the form of the denominator of the lower bound, which is not captured by any of the sketches. The extra gap seems hard to reach in the word-RAM model, especially while maintaining the space optimality we proved in Section 5.2.

Theorem 9 (Heavy hitters update lower bound)**.** *Any randomized non-adaptive streaming algorithm for L_1 heavy hitters must have worst case update time*

$$t_u = \Omega\left(\min\left\{\sqrt{\frac{\log \delta^{-1}}{\log\left(\frac{eS}{t_u}\right)}}\ ,\ \frac{\log \delta^{-1}}{\sqrt{\log(t_u)\log\left(\frac{eS}{t_u}\right)}}\right\}\right)$$

where S is the space of the algorithm.

Using either of the sketches from Chapter 3 with depth $d = C\log m$ for $C > 1$ ensures that all m point queries succeed within the allowed error guarantee with probability $1 - O\left(\frac{1}{poly(n)}\right)$, while supporting updates in $O(\log m)$ time. Such an update time is generally good, but still off by a factor, compared to the lower bound from Theorem 9.

The algorithms from Chapter 4 using the hierarchical data structure and the sketches as black boxes are, in order to improve the query time, a factor of $\log m$ worse in update time, which makes their update time even further from optimal.

The final algorithm in Chapter 4 traded faster update speed for expected query time, to support updates in $O(\log m)$ time, making it comparable with the sketch solutions again. The upside to this algorithm compared to simply using a sketch, was the improvement in (expected) query time, using the same space as the sketches.

Generally, it seems hard to achieve optimal update time for the approximate L_1 heavy hitters problem without trading away the space usage. Such a trade is not necessarily wanted as several of the solutions presented are actually optimal in space.

To the knowledge of the authors of this thesis, no better algorithms for the approximate L_1 heavy hitters problem in the *Strict Turnstile Model* are known in the word-RAM model, and a gap thus exists between what is known and what should be possible to achieve, with respect to the update time of the algorithms.

5.4 Summary

In this chapter, we have compared the *Strict Turnstile Model* algorithms and data structures analyzed in earlier chapters according to the space and update lower bounds.

To solve the frequency estimation problem, the Count-Min Sketch and Count-Median Sketch both use optimal space but only have near-optimal update time.

Using the sketches to solve the heavy hitters problem, again gives optimal space, fast update time, but a bad query time due to the need for querying all elements.

The query time is improved by using the Hierarchical Count-Min Structure or the Hierarchical Count-Median Structurebut at the cost of an extra $\log m$ factor in update time and space usage. This makes the update time even further from optimal, while the space becomes a log factor from optimal.

The recent discovery of the Hierarchical Constant-Min Structure, deals with this trade-off and enables a structure with good expected query time, fast update time and optimal space usage.

Generally, it seems like there is a relation between space and update times, where improving one would hurt the other. Still, the Hierarchical Constant-Min Structure seems to be able to find a good trade-off, where the expected query times are improved significantly as well.

Chapter 6

Experiments

In this chapter we will describe and present experiments, which will test the *Strict Turnstile Model* algorithms and data structures described in Chapter 3 and Chapter 4. The experiments will mainly focus on testing the precision, space usage and running time of the algorithms in order to compare the theoretical parameters and bounds in practice.

In Section 6.1, we will present some general information about the testing environment, the test setup, the implementations of the different data structures, and the input data used for testing.

In Section 6.2, we will perform a range of experiments on the sketches mentioned in Chapter 3 and try to argue that the Count-Min Sketch and the Count-Median Sketch are comparable as a black box for other problems.

Finally, in Section 6.3, we will be doing experiments for the approximate L_1 heavy hitters problem, where several different implementations from Chapter 4 will be tested and compared against each other.

6.1 Implementation & Test Details

This section will describe the optimization and implementation details made in order for the algorithms to perform as good as possible. Furthermore, we will describe the setup in which the experiments are carried out in and more generally how the measurements are performed. Finally, the distribution used to generate the test data is discussed.

6.1.1 Implementation

All implementations were written in C (C99), which enabled us to have full control of the memory usage. We implemented two different sketches (the Count-Min Sketch and Count-Median Sketch) and 6 different approximate L_1 heavy hitters algorithms. The sketches are implemented in a generic framework which enable algorithms using a sketch to use an abstract sketch type and just specify which implementation (Count-Min Sketch or Count-Median Sketch) the sketch should have. As a consequence, any use

of sketches becomes a use of an abstract sketch object, for which the user defines the implementation.

Such an abstraction is a huge gain in being able to verify the correctness and increase the maintainability of the code, since less code has to be written. Just as important, the abstraction of sketches enables us to unit test the specific sketches, which again should provide a better guarantee of the correctness of the sketch implementations.

One possible downside of having designed the implementations as generic as possible, is that performance could be affected negatively, since extra function calls are introduced and since every generic structure allocates its own block of memory, that is not necessarily consecutively stored with other memory. Hence, a consequence of using the abstract sketch objects, could be extra cache misses.

For the sketch implementations, a big optimization is the placement of the constants (a and b, see Subsection 2.3.1) for the c-universal hash function. These are placed at the beginning of each row in the sketches. Such placement should generally imply fewer cache misses, since both the update and query algorithms of the sketches follow a specific pattern whenever a bucket is fetched or written to. First the constants are referenced in order to generate the hash, where after some element in the same row as the constant is referenced. Hence, for small sketches, the prefetcher of the cache, should be able to prefetch the hashed value, without causing a cache miss.

For all heavy hitters implementations using the hierarchical data structure with sketches as black boxes, an optimization is possible, since one can keep an exact count of the frequencies of ranges, at the first $l < \log_k m$ levels of the hierarchical data structure, where k is the branching factor. This is because the first levels of the tree covers large ranges, implying that only few nodes are present. The space usage from using a sketch at a level keeping track of for instance 4 ranges, is far greater than just keeping an exact counter for each of the ranges. The number of levels l for which the structure should hold exact counts is calculated by finding the level of the tree, for which holding an exact count of the ranges in that level becomes more expensive than having a sketch. This optimization is also mentioned by Cormode and Muthukrishnan [9] and used in the implementation from Cormode and Hadjieleftheriou [7].

A final optimization was performed for the Hierarchical Constant-Min Structure, where the sketches on each levels of the hierarchical data structure was changed to have a depth of 1. This implied that the width of the sketches should be scaled such that $w = O\left(\epsilon^{-1}\delta^{-1}\right)$, to allow each row of the sketches to provide a failure probability of $\frac{1}{b}$ i.e. in order to maintain the same guarantees as an unscaled Count-Min Sketch. The constant sized sketches could then be stored in a consecutively stored array, which should imply fewer cache misses.

6.1.2 Setup

All tests ran on a few dedicated test machines. The specifications of the machines are shown in Table 6.1 and they have the same specifications.

The machines have the latest version of Arch Linux with kernel version 4.4.5-1 installed, with a minimum of applications. The minimal install minimizes interference

CPU	Intel(R) Core™2 Duo CPU E8500 @ 3.16GHz
L1 Cache	32 KiB, 8-ways associative, 64 byte lines, Split
L2 Cache	6144 KiB, 24-ways associative, 64 byte lines, Unified
TLB	4 KiB pages, 512 entries, 4-ways associative
RAM	8GB 800MHz DDR2 SDRAM
HDD	Seagate Barracuda 7200.10 ST3160815AS 160GB 8MB Cache SATA 3.0Gb/s 3.5“

Table 6.1: Dell Optiplex 760 specifications.

from other applications when running the tests, and lowers the chance of context switching for the tests.

The code is compiled with GCC version 5.3.0 using `-O3`, `-march=native`, and `-NDEBUG` flags for optimization and removal of debugging code. Debugging and verification of correctness was done using the `Valgrind`[1] tool along with unit tests written using the `Criterion`[2] library.

Analyzing and optimizing specific parts of the code has been done using the `Cachegrind` and `Callgrind` tools available in `Valgrind`, along with the `Perf`[3] tool.

The code is stored in a git repository and includes a README and a Makefile. This combination allows for an easy way to fetch, compile, and reproduce the experiments. In addition to the compiled binaries, there are also a few Bash scripts with different test suites. Having the test suites in scripts, makes it easier to reproduce the set of tests and specify the parameters δ, ϵ, ϕ etc. that the algorithms use.

The implementation of all sketches and heavy hitters algorithms can be found at `https://github.com/mortzdk/heavy-hitters`, whereas a submodule enabling the collections of the PAPI events can be found at `https://github.com/mortzdk/libmeasure`.

6.1.3 Measurements

When measuring performance on a computer it is always difficult to ensure that nothing external have influenced the measurements. Other programs might require CPU time, memory etc. so in order to prevent this, the minimal install mentioned in the last section is an important factor.

Another way of minimizing external influence is by repeating the tests and using the mean values of the measurements. We make sure to repeat every measurement multiple times to get minimal external influence.

For the update algorithms, the amount of repetitions performed, is equal to the size of the stream n. This is feasible as we make sure that the streams are large i.e. at least several hundreds of thousands of updates are performed, from which we compute an average.

[1] `http://valgrind.org`
[2] `https://github.com/Snaipe/Criterion`
[3] `https://perf.wiki.kernel.org`

PAPI_TOT_CYC	Total cycles
PAPI_REF_CYC	Total cycles for constant clock rate
PAPI_TOT_INS	Instructions completed
PAPI_TLB_DM	Data translation lookaside buffer misses
PAPI_TLB_IM	Instruction translation lookaside buffer misses
PAPI_BR_NTK	Conditional branch instructions not taken
PAPI_BR_TKN	Conditional branch instructions taken
PAPI_BR_MSP	Conditional branch instructions mispredicted
PAPI_BR_PRC	Conditional branch instructions correctly predicted
PAPI_L1_DCA	L1 data cache accesses
PAPI_L1_DCM	L1 data cache misses
PAPI_L1_ICA	L1 instruction cache misses
PAPI_L1_ICM	L1 instruction cache misses
PAPI_L2_DCA	L2 data cache accesses
PAPI_L2_DCM	L2 data cache misses
PAPI_L2_ICA	L2 instruction cache misses
PAPI_L2_ICM	L2 instruction cache misses

Table 6.2: PAPI event descriptions.

When measuring the query algorithms we perform $10,000$ queries, and averages the measurements.

The measured time is real time and it is measured with the `clock_gettime` function using the `CLOCK_REALTIME` constant as an argument.

To get detailed information about the processor during executions of the update and query algorithms, we use the PAPI library (v. 5.4.3). The information available from the library is useful for a better understanding of which processor specific metrics that affect the running times of the algorithms.

Table 6.2 shows the list of the different events collected, together with a short description of the event. As we present the experiments, we will provide plots that use some of the events. Not all events are shown in the plots and experiments to follow, since only those that helps reasoning about the general running times of the algorithms are relevant to mention and show.

6.1.4 Zipfian Distribution

When testing algorithms, the test data used to do the tests is of great importance, since specific distributions of test data could make the algorithms perform better or worse.

Realistic distributions of data can often be characterized by the Zipfian, Pareto, or Power-law distributions, which essentially all are equivalent up to the choice of parameters. The zipfian distribution with parameter $\alpha \geq 0$ is a discrete distribution stating that the k'th largest frequency f_k has a frequency proportional to $k^{-\alpha}$. As a consequence, $\alpha = 0$ generates a uniform distribution whereas the larger α the more skewed the distribution gets.

The zipfian distributions has been shown to fit a wide variety of real world data, such as sizes of cities, word frequencies, citations of papers, web page access frequencies, and

Data Source	Zipfian Skewness (α)	Level of skew
	0	Uniform
	< 0.5	No/Light
	≥ 0.5	Moderate
Web page popularity	$0.7 - 0.8$	Moderate
FTP transmission size	$0.9 - 1.1$	Moderate/High
	≥ 1.0	High
Word use in English text	$1.1 - 1.3$	High
Depth of website exploration	$1.4 - 1.6$	High

Table 6.3: Examples of sources according to the α parameter of the Zipfian Distribution from Cormode and Muthukrishnan [8].

file transfer size and duration, to name a few [1]. Examples of real world data compared to the zipfian parameter α and the categorization of skewness of the distribution generated from the choice of α, can be found in Table 6.3.

Zipfian distributions are especially interesting with regards to the heavy hitters problem since this problem looks for frequencies which are significantly larger than the rest of the data. For increasingly skewed zipfian distributions the elements with such frequencies become more frequent, since only a few of the overall frequencies account for more of the total frequency. As a consequence, such elements should be easier to find with a reasonably chosen ϕ parameter.

Due to the property of being similar to real world data, we chose to do our experiments with artificially generated zipfian distributions, for different choice of α. The different elements of the distributions will be distributed uniformly at random over the universe of the data, such that elements with high frequencies are not necessarily neighbors. Earlier experiments have used the same type of distribution of data and showed that running the experiments on real world data gives the approximately same results as using zipfian distributed data [7, 8].

6.2 Sketches

In this section we will perform experiments on the Count-Min Sketch and Count-Median Sketch presented in Section 3.3 and Section 3.4.

Looking at the sketches from a theoretical point of view, the Count-Min Sketch and Count-Median Sketch provide different guarantees in the error they introduce, as a consequence of their approximations. The Count-Min Sketch guarantees a one-sided error where all estimates $\hat{v}_i \leq v_i + \epsilon \|v\|_1$ with failure probability δ. The Count-Median Sketch gives a potentially stronger guarantee, in that the absolute difference between all estimates and their true frequencies are $|\hat{v}_i - v_i| \leq \epsilon \|v\|_2$ with failure probability δ.

Another difference between the sketches is the space they use. The Count-Min Sketch has width $w = {}^{b}/_{\epsilon}$ and depth $d = \log_b \delta^{-1}$ for $b > 1$. The Count-Median Sketch has width $w = {}^{k}/_{\epsilon^2}$ and depth $d = {}^{\ln(\delta^{-1})}/\left(\frac{1}{6} - \frac{1}{3k}\right)$ for $k > 2$.

As a consequence the sketches can be quite hard to compare in practice, nonetheless we will try to do so.

In the next section we will test the sketches according to their theoretical bounds, that is, using whatever constants found from the theoretical analysis of the sketches. The constants b and k will be chosen to be equal i.e. $b = k = 4$ and the size of the universe – amount of unique elements – is chosen to be $m = 2^{26}$, which requires at least 512 MiB to create an exact solution by keeping a 64-bit counter for each element.

The choice of m comes from the fact that some of the results regarding the precision of the sketches, depend on being able to hold an exact solution in memory.

Finally the failure probability is chosen to $\delta = 2^{-18}$ since such a low probability typically is used for the heavy hitters problem due to scaling. This failure probability gives a depth of 9 for the Count-Min Sketch and a depth of 150 for the Count-Median Sketch.

Our tests should hopefully verify that the theoretical bounds and guarantees are upheld for both sketches. Moreover the tests should show the true difference in space and estimation errors for the two different sketches. Finally the tests should reveal the running times of the update and query algorithms and verify whether the Count-Min Sketch and Count-Median Sketch are applicable as synopsis data structures (Definition 1).

To compare the sketches even further we will repeat the experiments on the Count-Min Sketch and Count-Median Sketch data structures, giving them the exact same width and depth. This should clarify if we can allow to loosen the theoretical bounds of the Count-Median Sketch in order to obtain more or less equivalent guarantees to the Count-Min Sketch.

6.2.1 Theoretical bounds

In this subsection we will experiment with the Count-Min Sketch and Count-Median Sketch. Since the Count-Median Sketch has a significantly higher space usage than the Count-Min Sketch, due to different widths and larger constants in the depth (See Section 3.4), we will perform measurements for this sketch with two different settings, namely a Count-Median Sketch with the original theoretical bounds, and one without the constants for the depth, i.e. one with depth $d = \ln \delta^{-1}$.

Space

As the width of the Count-Median Sketch increases quadratically with ϵ, the width of the Count-Median Sketch will be significantly larger than the width of the Count-Min Sketch. Furthermore, the depth of the sketches differs in a considerable constant factor, which put together with the width should have the overall effect that the Count-Median Sketch will be using much more space than the Count-Min Sketch for most practical purposes.

Choose for example $\epsilon = 0.01$ and $\delta = 0.02$. The Count-Min Sketch will then use ≈ 0.0096 MiB while the Count-Median Sketch will use ≈ 15.04 MiB. Scaling the approximation factor with 10^{-1} to $\epsilon = 0.001$, yields for the Count-Min Sketch approximately

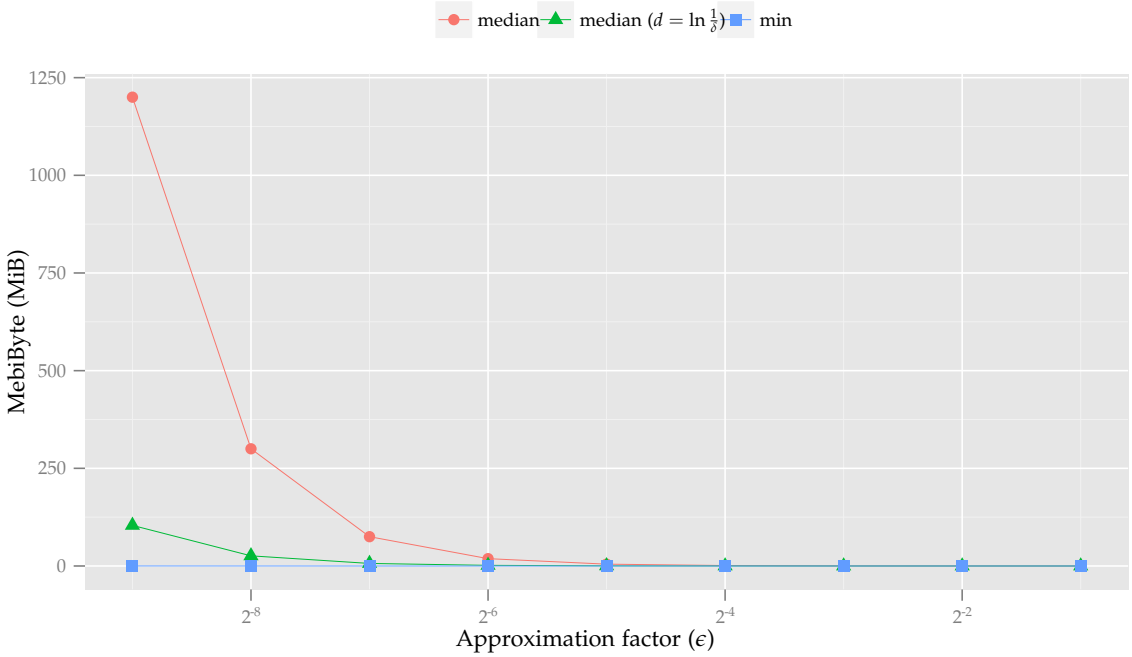


Figure 6.1: The space usage for $\delta = 2^{-18}$ of the Count-Min Sketch and the Count-Median Sketch with and without constants applied to calculate the depth.

0.096 MiB of space, while it will require approximately 1504 MiB of space for the Count-Median Sketch.

Looking at the space usage for the Count-Median Sketch with depth $d = \ln \delta^{-1}$, the space with the two approximation factors would be $\approx$ 1.28 MiB and $\approx$ 128 MiB, respectively. This is especially important for the latter setting as the size of the sketch is below the size of an exact solution.

From Figure 6.1 it can be observed that this is in fact the behavior of the space usage over different choices of ϵ. Here the space usage is plotted in $\log_2$ for an increasing ϵ and a fixed failure probability $\delta = 2^{-18}$. It is clear from the plot that as ϵ decreases, the difference in space usage between the Count-Min Sketch and the two different Count-Median Sketches increases significantly.

Comparing the sketches with holding an exact count over the frequencies, it is clear that the Count-Min Sketch uses significantly less space. The Count-Median Sketch with the true theoretical depth is another story. For $\epsilon < 2^{-8}$, the space usage of the sketch becomes larger than keeping an exact count, which at this point implies that the sketch is no longer useful. The Count-Median Sketch without the constants applied for the depth, saves a factor of 12 in space making the sketch applicable for all tested ranges of ϵ.

This experiment shows that the Count-Median Sketch and its estimation error guarantee according to the L_2-norm comes with a significant trade-off in space, implying an exact count of the frequencies would be just as useful for small parameter choices.

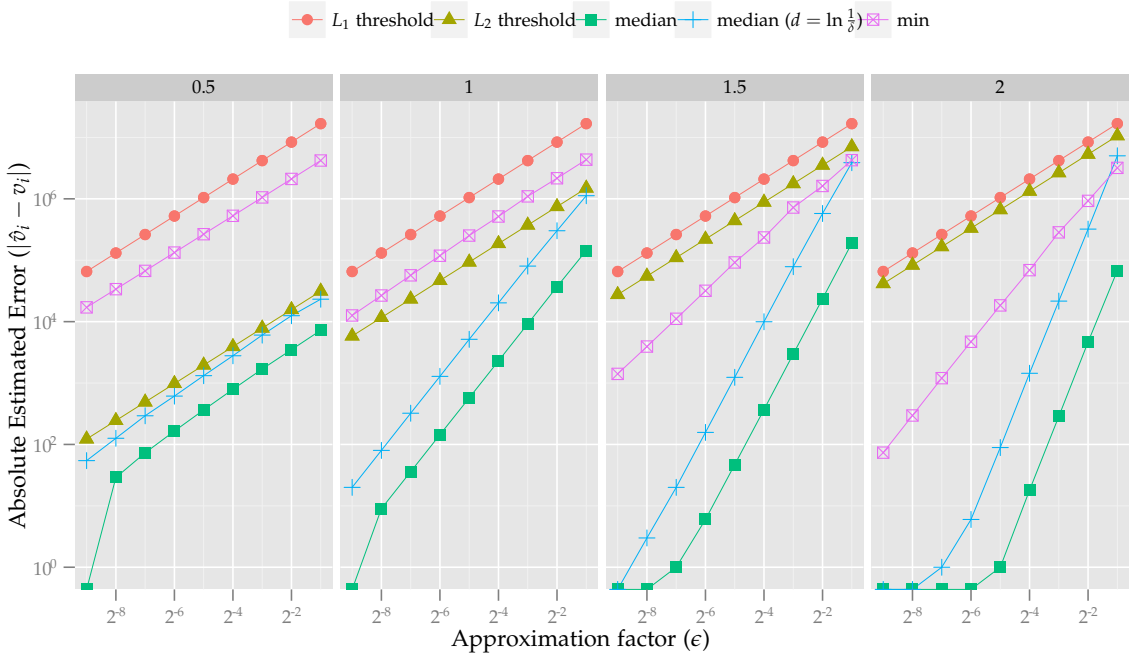


Figure 6.2: The absolute error of sketches over zipfian distributed data with parameter $\alpha = \{0.5, 1.0, 1.5, 2.0\}$. Here the L_1 and L_2 thresholds are shown in order to determine if the sketches uphold their guarantees. The Count-Median Sketch appears with theoretical constants and without ($d = \ln \delta^{-1}$).

Precision

The great difference in space usage should be compensated, when looking at the precision of the point queries. To test the precision, an experiment was created where the exact frequencies of all elements were kept, in order to calculate the absolute error $|\hat{v}_i - v_i|$ of the approximations. Such an experiment should reveal the error in each estimate and whether the failure probability is upheld.

Recall, from Section 3.3 that the Count-Min Sketch provides estimated frequencies $\hat{v}_i \leq v_i + \epsilon \|v\|_1$ and the Count-Median Sketch provides estimated frequencies $|\hat{v}_i - v_i| \leq \epsilon \|v\|_2$, both with probability $1 - \delta$. When querying the structures multiple times we expect the result to deviate with more than the approximation factor with probability δ. Therefore, the precision is measured as the largest absolute error less than the absolute errors of the δ elements with the largest absolute errors. The deviation of these δ elements are accepted to be more than the approximation factor.

For each element $i \in [m]_1$ the exact structure and the approximate structure were queried and the difference was calculated and stored in a list. Removing the δ m largest absolute errors from the list leaves a set of absolute errors, for which all must be within the approximation factor in order for the sketches to uphold their theoretical guarantee. For the Count-Min Sketch and the Count-Median Sketch it is $\epsilon \|v\|_1$ and $\epsilon \|v\|_2$,

respectively.

In Figure 6.2 the result of the experiment is shown. The y-axis contains the largest accepted absolute estimation which should be below the norm guarantees of the sketches. The absolute error is plotted on the y-axis in log_{10} and the x-axis is plotted in $\log_2$ and varies over the approximation factor ϵ. For a decreasing ϵ the structures use more space, while the acceptable absolute error of the estimates gets smaller. For both sketches, a fixed failure probability of $\delta = 2^{-18}$ was chosen. As mentioned in Subsection 6.1.4, the experiments were run on artificially created zipfian distributions, that act like most natural data for different settings of α (See Table 6.3). In Figure 6.2 the precision is plotted for zipfian distributions with different α parameters. The plots represent data being distributed almost uniformly for the smallest α to data that is highly skewed for the largest α.

From the plot it is clear that estimates provided by the Count-Median Sketch are significantly more accurate than those provided by the Count-Min Sketch. This is not really surprising as the Count-Median Sketch use a lot more memory than the Count-Min Sketch to provide the estimates. This consequently leads to the error guarantees being different for the two sketches, as found from the analysis in Chapter 3.

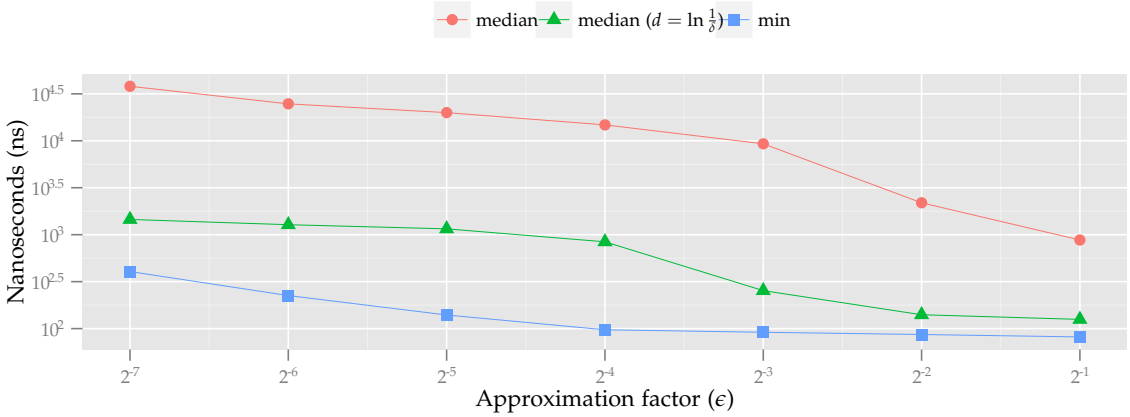
The relationship between the L_1 and L_2-norm is $\sqrt{\|v\|_1} \leq \|v\|_2 \leq \|v\|_1$, where uniformly distributed data make the L_2-norm go towards the $\sqrt{L_1}$ and highly skewed data the other way. This relationship between the norm guarantees, also seems to be present in Figure 6.2 where lower α values makes the difference between the precision of the two sketches larger and higher values of α the opposite.

When the data becomes more skewed the sketches become more accurate. This is seen from the difference between the absolute error and its respective threshold, where the distances in general increase when α increases. This can be explained by the fact that the larger α becomes the more skewed the data is, and less elements hold a larger portion of the total mass. For an increasing α and decreasing approximation factor, these elements will only collide with other *heavy* elements with less probability. The fact that the sketches become more precise as data is more skewed, is described and analyzed in Charikar et al. [5] and Cormode and Muthukrishnan [8] for both types of sketches. Here they show stronger space or threshold guarantees of the data structures for $\alpha > 1$.

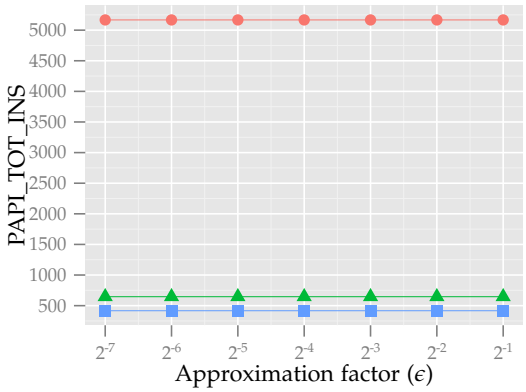
The most important thing to note from the plot is the fact that for all α it holds that the Count-Min Sketch is significantly below the L_1 threshold and the Count-Median Sketch are likewise significantly below the L_2 threshold. This observation verifies that both sketches uphold their theoretical guarantees, when they are used with their theoretical parameters.

For the Count-Median Sketch, it can further be observed that for our testing data, using depth $d = \ln \delta^{-1}$ still makes the sketch uphold the L_2 threshold, even though the estimates are somewhat worse than those of the theoretical correct one. In practice the Count-Median Sketch with $d = \ln \delta^{-1}$ seem more attractive since it has a significantly lower space usage, while still providing good frequency estimates with errors according to the L_2-norm.

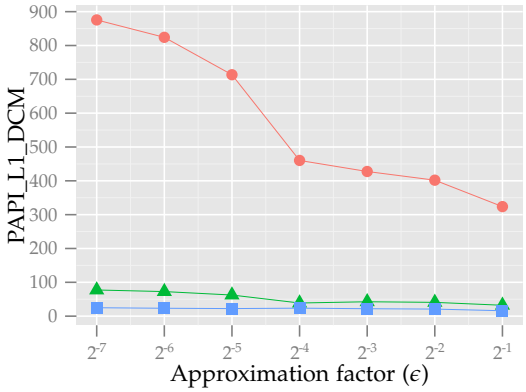
A final note about the precision of the sketches is that experiments were also performed for larger and smaller choices of δ. The results were similar and thus omitted.



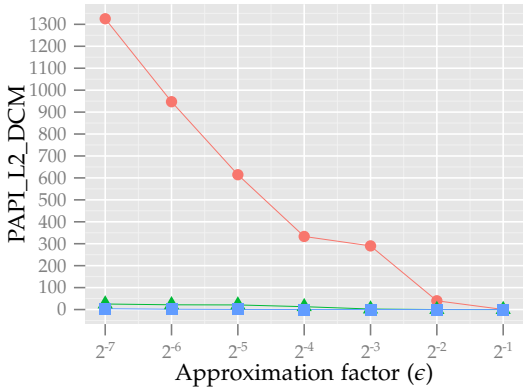
(a) Running time in nanoseconds



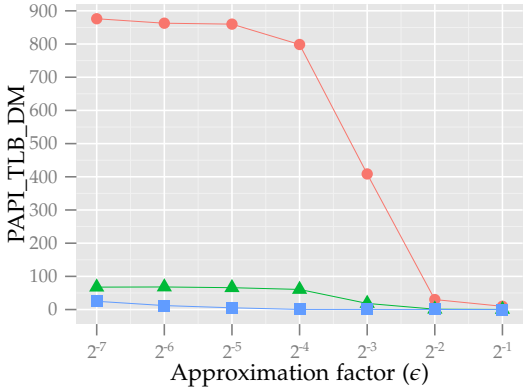
(b) Instructions Completed



(c) L1 Data Cache Misses



(d) L2 Data Cache Misses



(e) TLB Cache Misses

Figure 6.3: Update measurements where (a) shows the running time, (b) a plot of the amount of instructions carried out by an update operation, while (c-e) show the amount of different cache misses that occur during the update operation for the L1, L2, and TLB cache, respectively. All measurements are for data from a zipfian distribution with $\alpha = 1$.

Running times

The final set of experiments for the sketches was performed to measure the running times of the update and query algorithms. From the analysis of both sketches in Section 3.3 and Section 3.4, the theoretical bounds of both algorithms are $O(d)$ i.e. linear in the depth of the sketch.

For a fixed failure probability $\delta = 2^{-18}$, the analysis implies that the running times should be constant over all ϵ. However, this is not the case for either the update or query algorithm. In fact all measurements tend to grow in time as ϵ decreases, which should be clear from Figure 6.3a for the update algorithm where the y-axis is plotted in $\log_{10}$. The query algorithm have a very similar behavior, but we choose to omit plots and analysis of these, since the reasoning is equivalent to the update time.

Although this is not consistent with the analysis, one point is missing. When doing the analysis in the word-RAM model, the complexity of algorithms is measured in the amount of operations and memory accesses performed to solve a given problem. On real life machines a cache hierarchy is often present in the CPU. Such hierarchies are not part of the model, which implies that any delay in time caused by such hierarchies are not counted in the running time in the word-RAM model.

The machine that ran the measurements has two levels of cache, L1 and L2 cache, and the main memory (RAM), see Table 6.1. The L1 cache is the smallest but fastest cache, the L2 cache is a bit larger but slower, finally the main memory is even slower but much larger. One additional cache is present, namely the TLB cache which enables fast processing of virtual to physical addresses.

Whenever data is not present in a cache, a cache miss occurs which implies that it must be fetched from the next level in the cache hierarchy. As a consequence L1 cache misses occur when the data is not present in the L1 cache and must be fetched from the L2 cache, and L2 and TLB cache misses occurs when the data is not present, and must be fetched from RAM.

Looking at the CPU data for the three caches mentioned above in our measurements for the update algorithm and for $\alpha = 1$ reveals from Figure 6.3c, Figure 6.3d, and Figure 6.3e a pattern of the executions somewhat similar to the running time from Figure 6.3a.

When ϵ decreases, more cache misses occur, which cause the increase in the running time. The general reason behind the cache misses for all sketches is that memory accesses to the sketches, generally happens in "jumps", since one row is accessed at a time. The only thing that might reduce the amount of cache misses is if the prefetcher of the L1 cache, manages to prefetch a block of memory containing the next row in the sketch.

The L1 cache size is of size 32 KiB. This seems to imply that even for the smallest values of ϵ, the Count-Median Sketch with the true theoretical depth, suffers from several hundreds of cache misses, due to the fact that the depth of the sketch becomes $d = 150$.

The update algorithm for the Count-Median Sketch is as follows: For each row, four consecutively stored hash function constants, $a1$, $b1$, $a2$, $b2$, are fetched. Furthermore one specific memory cell in the row is fetched and written to, in order to perform an update.

For $\epsilon \geq 2^{-4}$ the widths of the sketches are generally of such size that the prefetcher

should be able to prefetch the four constants and the specific memory cell, since each row of the sketch does not get too large compared to the L1 data cache size.

For $\epsilon < 2^{-4}$ it is another story. Here each row of the sketch becomes greater than the full size of the L1 data cache and cache misses should in general occur for each memory access performed, since the prefetcher no longer can store the next row.

The same problem exists for the Count-Median Sketch with depth $d = \ln \delta^{-1}$. The difference in the running times, comes from the fact that the depth is only $d = 13$, which reduces the amount of cache misses by a factor of ≈ 12 for $k = 4$. The reduction in cache misses comes from the fact that fewer rows has to be visited in order for an update to be performed.

The Count-Min Sketch is less affected by L1 data cache misses, since the sketch in general is small and the width for most of the measured ϵ fits in the L1 data cache. A small increase is observed for $\epsilon \geq 2^{-4}$, where a row no longer fits a cache line, but the increase is not visible from the plot, since the increases for the other sketches are dominating. Still, the increase is indirectly visible in the running time, which begins to increase for $\epsilon < 2^{-4}$.

The L2 cache misses should show a similar behavior as the L1 data cache misses i.e. increase as the width of the sketch grows. This is also the behavior observed from Figure 6.3d. The closer the width of the sketches comes to the size of the L2 cache (6,144 KiB), the more L2 cache misses occur. The difference in the amount of L2 cache misses between the sketches, is again due to the depth of the sketches, i.e. more rows should be updated for the Count-Median Sketches.

The final kind of cache misses, the TLB data cache misses, occur whenever a memory access is performed and the address of the memory is not present in the TLB data cache. The page size of the test machines are 4 KiB, which implies that 4 KiB of consecutive memory is mapped to the same virtual address. Whenever memory is fetched and the TLB data cache is updated, the address of 4 KiB of consecutive memory around the requested memory is also stored.

For both Count-Median Sketches, the TLB data cache misses increase significantly as ϵ goes from 2^{-2} to 2^{-4}. At 2^{-4} the width of the sketches becomes 8 KiB which results in at least 1 TLB data cache miss for each depth of the sketches. As ϵ drops the misses grow slightly, until the point where every cache access gives a TLB cache miss. Again, the Count-Min Sketch is not too affected by TLB data cache misses, since it has a significantly smaller width and lower depth.

What we have established at this point is that the caching hierarchy of the CPU does influence the running time and in such ways that the plots of the caches and the running time have a similar curve. The question is then, if update algorithm do in fact uphold the bound from theory? To answer this we look at the amount of instructions performed. In the word-RAM model all simple instructions can be carried out in constant time. Since δ is fixed in these measurements, the amount of instructions are expected to be constant for all sketches. From Figure 6.3b we can indeed observe that this is true.

Here it is worth mentioning that the instructions actually drops for the Count-Median Sketch for the query algorithm as ϵ decreases, which is not visible since the plot was omitted. This can be explained by the fact that as ϵ decreases, the estimates generally

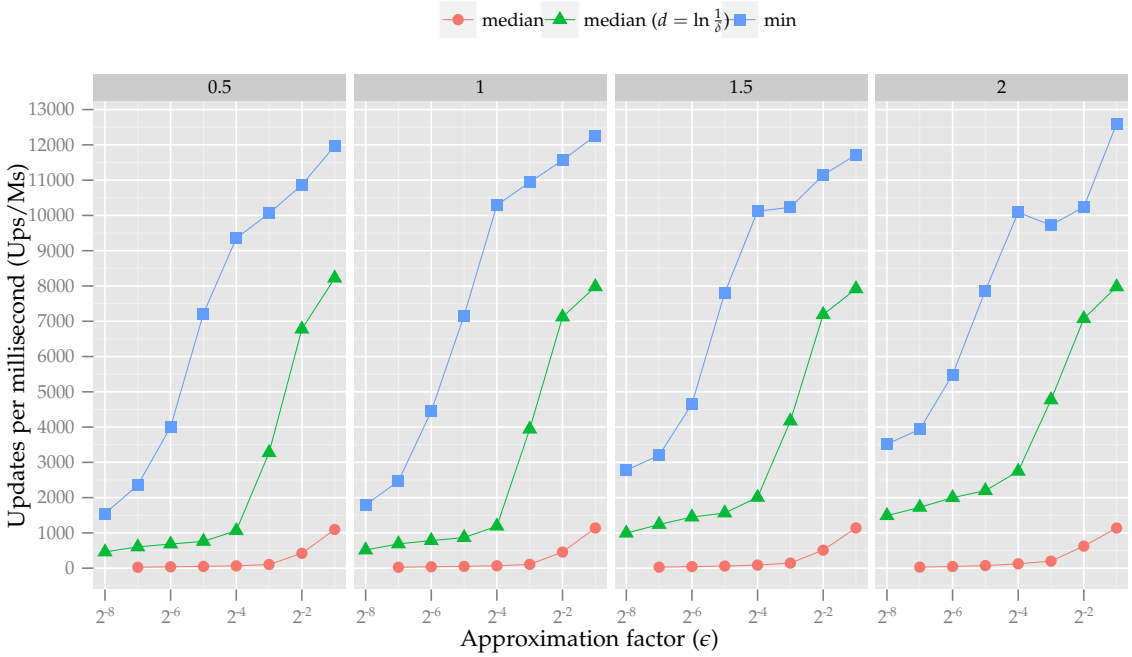


Figure 6.4: Updates per millisecond of the Count-Min Sketch and Count-Median Sketch, where the Count-Median Sketch is tested for different depths. For $\alpha = \{0.5, 1, 1.5, 2\}$.

become more equal over all rows. When the median algorithm is run for equal estimates, it decreases the amount of instructions, since branches can be skipped, as it does not matter which of the equal estimates to return.

Experiments where δ was varied were also performed, and it was observed that the instructions increased with a factor of d as expected by theory.

Based on the above observations we can say that the sketches follow the theoretical running times in the word-RAM model, but suffers from penalties, whenever cache misses occurs.

Having argued the relationship between the running times in theory and practice we can now actually look at the performance of the sketches. In Figure 6.4 and Figure 6.5 the operations per millisecond are plotted, ranging over different choices of ϵ.

From Figure 6.4 it is quite clear that the Count-Min Sketch is extremely fast, and can perform several thousands more operations than the Count-Median Sketches. For $\epsilon \geq 2^{-4}$ the amount of updates for the Count-Min Sketch only decrease slightly. For $\epsilon < 2^{-4}$ the decrease in amount of updates begins to be larger. This can be explained by the fact that more L1 cache misses occur due the width of the sketch becoming bigger than the size of the cache line.

Still, the Count-Min Sketch performs with several thousands of updates per milliseconds overall. The Count-Median Sketch with true theoretical depth, generally performs badly and as ϵ grows, it is only able to perform between $25 - 100$ updates per milliseconds. The reason for the big difference between the Count-Median Sketch and the

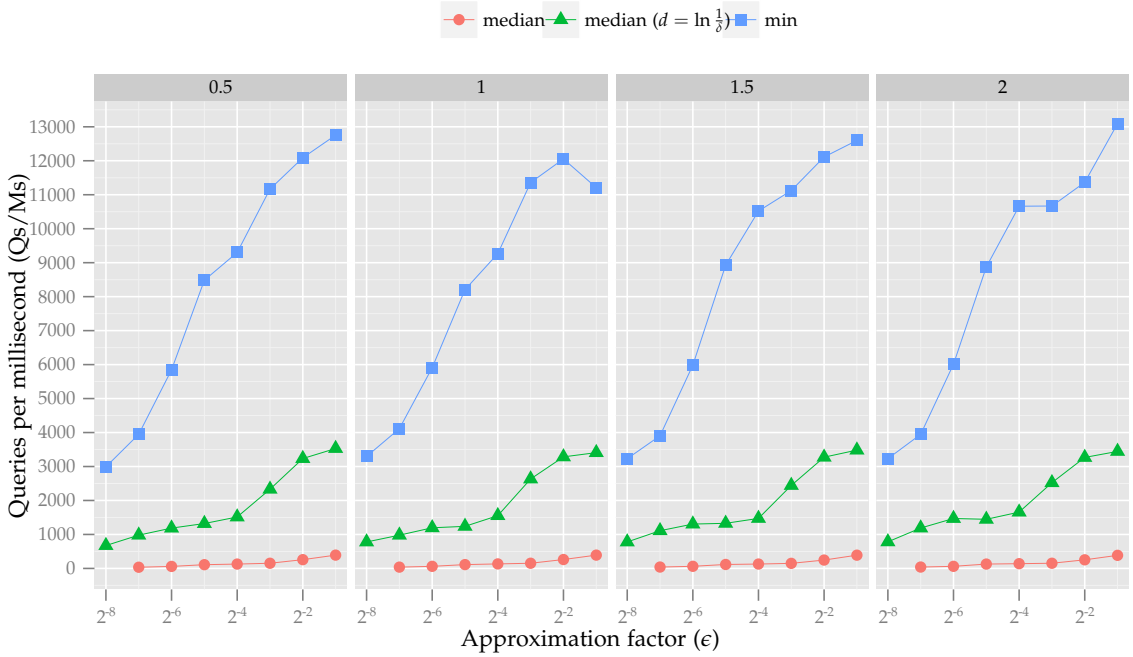


Figure 6.5: Queries per millisecond of the Count-Min Sketch and Count-Median Sketch, where the Count-Median Sketch is tested for different depths. For $\alpha = \{0.5, 1, 1.5, 2\}$.

Count-Min Sketch is the extra space usage, which in turn implies a lot more cache misses. The Count-Median Sketch with depth $d = \ln \delta^{-1}$ is less affected by the cache misses, but still performs significantly less updates per milliseconds than the Count-Min Sketch.

A final thing to note about the update times, is the fact that they are influenced by the skew of the distributions. As α grows, the data becomes more skewed, which implies that updates to the same element happens more frequently. Since the same element is updated more frequently, fewer cache misses in general happens, as the chance of having the memory or address stored in cache increases. A lot of real world data tends to be skewed, which impacts the sketches positively by making the update algorithms faster.

In Figure 6.5 the queries per millisecond of the sketches is shown. For the Count-Min Sketch this looks a lot like the updates, which is expected since the two algorithms are almost identical, except that no updates is performed, instead the minimum of all rows have to be found. The query throughput for the Count-Median Sketches are generally lower than the update throughput. This is due to the median algorithm generally being slower than for example finding the minimum, which influence the running time negatively.

The skew of the data does not seem to influence the running time of any of the query algorithms. The queries were performed for randomly chosen elements in the data, which means that the chance of the cache to reuse memory or addresses from old queries is almost non-existing.

As a final note, experiments with other choices of δ were also performed and what

could be observed about the running time was as in theory, that an increasing δ resulted in better running times whereas decreasing δ resulted in worse running times due to increasing the depth of the sketches.

From the experiments above, we conclude that the Count-Min Sketch is very applicable as a synopsis data structure, since it has a low memory footprint, is incredibly fast, and since the precision is within the bounds of theory. These features along with the fact that the sketch works as a good surrogate of the frequencies makes it a good synopsis data structure. Even though the Count-Median Sketch with true theoretical depth was shown to have a very good precision according to the guarantees from theory, it falls through when looking at the space usage and running times. The Count-Median Sketch with depth $d = \ln \delta^{-1}$ is in general more applicable as a synopsis data structure, since it upholds the bounds from theory for all data tested upon, while using acceptable space and having decent running times. We can thus conclude from the measurements that the Count-Min Sketch and the Count-Median Sketch with depth $d = \ln \delta^{-1}$ in practice do a good job approximating the frequencies of elements with an additive error within an approximation factor of the L_1 and L_2-norm, respectively.

All the results presented in this section have been for two different sketches providing different error guarantees. For this reason the comparisons made between the Count-Min Sketch and Count-Median Sketch is not really fair, since the sketches in general are used in two different contexts. In the next section we will experiment with the sketches, giving them the same depth and width. Such a change will enable a better and more fair comparison between the sketches.

6.2.2 Equal Space

This section will describe experiments with the Count-Min Sketch and the Count-Median Sketch of equal size. That is, configuring the sketches with equal width and depth, to obtain the same memory usage. The experiments will eliminate the advantages or disadvantages of using more memory, and specifically test the sketches with the main difference being the approach of choosing the minimum or median of the estimated frequencies and applying a sign or not.

Giving the two sketches equal width and depth will result in an almost equal memory usage for the two sketches, with the Count-Median Sketch using only slightly more space, to be able to change the sign of the updated values. The actual extra space for the Count-Median Sketch is two words more for each depth, which is a small and insignificant overhead. We therefore choose to say that these algorithms use equal space.

From Subsection 6.2.1 we found that the Count-Median Sketch provided estimates with an error guarantee according to the L_2-norm, even when the depth was chosen to be $d = \ln \delta^{-1}$. Using these results, we choose to ignore the constants in any future experiment, since the estimations were sound according to the L_2-norm at least for our data and since the space and running times generally improved by factors, simply by ignoring the constants.

Since the Count-Min Sketch should not be effected by the equal amount of memory, we expect the results of the Count-Min Sketch to be along the same as in Subsection 6.2.1,

whereas the Count-Median Sketch will use a lot less memory, which should infer different results for both precision, space and running times.

Changing the width of the Count-Median Sketch to be equal to the one of the Count-Min Sketch, should improve the running times of both the update and query algorithm. We in fact expect the running times of the algorithms to be quite close to those of the Count-Min Sketch. This is because the size of the two sketches will be equal, which should imply that both suffer equally from cache misses, which were shown to be the biggest reason for increased running times in the last section.

The Count-Median Sketch is still expected to be a little slower than the Count-Min Sketch, as it needs to evaluate an extra hash function to determine the sign of the index and then multiply it on to the update value. The query of the Count-Median Sketch is also expected to be a constant factor slower, since the median algorithm is generally thought to be slower than finding the minimum element in practice.

The precision of the Count-Median Sketch is also expected to change drastically, due to the decrease in space usage. In fact it is expected that the decrease in space implies that the error guarantee now is bounded according to the L_1-norm instead of the L_2-norm. This expectation comes Theorem 10 and from a similar proof in Gilbert and Indyk [17].

Theorem 10 (Count-Median Sketch L_1 guarantee)**.** *Using the Count-Median Sketch, one is able to bound guarantees according to the L_1-norm, choosing $w = O\left(\epsilon^{-1}\right)$ i.e. the absolute difference between all estimates and their true frequencies are $|\hat{v}_i - v_i| \leq \epsilon \|v\|_1$ with failure probability δ.*

Proof. To obtain a proof of this theorem the analysis of the Count-Median Sketch is done differently than earlier. Instead of only looking at the true frequency vector v, we also choose to look at a subset of the vector $v^{[\epsilon^{-1}]}$, which is defined as the vector v where the ϵ^{-1} largest frequencies – which we denote *heavy* frequencies – are zeroed.

For the Count-Median Sketch to provide estimates according to the L_1-norm, it must hold that each row of the Count-Median Sketch provides a L_1-norm guarantee for estimates with probability at least $1/2$. We can then use the same analysis as in Section 3.4 to bound the space, update and query time using an approximation factor ϵ and a failure probability δ, by taking the median of all rows. Let $\hat{v}_{i,j}$ denote the frequency estimate of element s_i in row j of the sketch. Further let E be the event that $|\hat{v}_{i,j} - v_i| > \epsilon \|v\|_1$, A be the event that the estimate $\hat{v}_{i,j}$ collide with a *heavy* frequency, and let $B = \neg A$ be the event that it does not.

The probability of a frequency estimate to fail in any row can then be stated as:

$$\mathbb{P}\left[E \mid A \cup B\right] = \mathbb{P}\left[E \mid A\right] + \mathbb{P}\left[E \mid B\right]$$

To argue that the Count-Median Sketch can deliver L_1-norm guarantees for its estimates we must show that the above happens with probability less than or equal to $1/2$.

$\mathbb{P}\left[E \mid A\right]$ comes from the fact that the mass of the ϵ^{-1}'th *heaviest* element in v can be at most $\epsilon \|v\|_1$. The probability for any frequency to collide with any one of the *heavy* frequencies can then be stated according to the Union Bound. Let $X_{i,k}$ be an indicator

variable indicating whether the estimated frequency of element s_i collides with *heavy* frequency v_k for any fixed row in the sketch. Since we use a c-universal hash function to decide the bucket of each elements frequency, we have that the probability of any two elements frequencies to collide is c/w where $c \geq 1$ is a constant.

$$\begin{aligned}
\mathbb{P}\left[E \mid A\right] &\leq \bigcup_{\{k \mid v_{k,j} \text{ is } \textit{heavy}\}} \mathbb{E}\left[X_{i,k}\right] \\
&\leq \sum_{\{k \mid v_{k,j} \text{ is } \textit{heavy}\}} \mathbb{E}\left[X_{i,k}\right] \\
&\leq \sum_{\{k \mid v_{k,j} \text{ is } \textit{heavy}\}} \frac{c}{w} \\
&= \frac{c}{\epsilon w} \\
&= \frac{1}{4} && \left(\text{Choosing } w = \frac{4c}{\epsilon}\right)
\end{aligned}$$

We thus have that a collision with one or more of the *heavy* frequencies happens with probability $1/4$ choosing $w = 4c/\epsilon$.

$\mathbb{P}\left[E \mid B\right]$ is then capturing the collisions of the remaining non-*heavy* frequencies. Here we can reuse the analysis of the expectation and variance of a single bucket for the vector $v^{[\epsilon^{-1}]}$ as done in Lemma 7 and Theorem 2. This give us:

$$\begin{aligned}
\mathbb{E}\left[\hat{v}_{i,j}\right] &= v_i \\
\text{Var}\left(\hat{v}_{i,j}\right) &= \frac{c\left\|v^{[\epsilon^{-1}]}\right\|_2^2}{w}
\end{aligned}$$

By Chebyshev's Inequality we can then bound the probability of any bucket to varie by more than $\sqrt{\epsilon}\left\|v^{[\epsilon^{-1}]}\right\|_2$, which we will show, is enough to state a guarantee according to the L_1-norm.

$$\begin{aligned}
\mathbb{P}\left[|\hat{v}_{i,j} - v_i| \geq \epsilon\sqrt{\frac{c\left\|v^{[\epsilon^{-1}]}\right\|_2^2}{w}}\right] \leq \frac{1}{\epsilon^2} &\Rightarrow \mathbb{P}\left[|\hat{v}_{i,j} - v_i| \geq \epsilon\frac{\sqrt{c}\left\|v^{[\epsilon^{-1}]}\right\|_2}{\sqrt{w}}\right] \leq \frac{1}{\epsilon^2} \\
&\Rightarrow \mathbb{P}\left[|\hat{v}_{i,j} - v_i| \geq \epsilon\left\|v^{[\epsilon^{-1}]}\right\|_2\right] \leq \frac{c}{w\epsilon^2} \\
&\Rightarrow \mathbb{P}\left[|\hat{v}_{i,j} - v_i| \geq \sqrt{\epsilon}\left\|v^{[\epsilon^{-1}]}\right\|_2\right] \leq \frac{c}{w\epsilon} \\
&\Rightarrow \mathbb{P}\left[|\hat{v}_{i,j} - v_i| \geq \sqrt{\epsilon}\left\|v^{[\epsilon^{-1}]}\right\|_2\right] \leq \frac{1}{4} \qquad (6.1)
\end{aligned}$$

where (6.1) comes from choosing $w = 4c/\epsilon$.

The final step is then to show that the above inequality does in fact state a guarantee according to the L_1-norm.

$$\sqrt{\epsilon}\left\|v^{[\epsilon^{-1}]}\right\|_2 = \frac{\left\|v^{[\epsilon^{-1}]}\right\|_2}{w^{\frac{1}{2}}} \qquad \left(w = O\left(\epsilon^{-1}\right)\right)$$

$$
\begin{aligned}
&= \frac{\sqrt{\sum_i (v_i^{[\epsilon^{-1}]})^2}}{w^{\frac{1}{2}}} \\
&< \frac{\sqrt{\left|\frac{\|v\|_1}{w}\right| * \left|\sum_i v_i^{[\epsilon^{-1}]}\right|}}{w^{\frac{1}{2}}} && (\forall v_i^{[\epsilon^{-1}]} < \epsilon \|v\|_1) \\
&= \frac{\sqrt{|\|v\|_1| * \left|\sum_i v_i^{[\epsilon^{-1}]}\right|}}{w} \\
&< \frac{\|v\|_1}{w} \\
&= \epsilon \|v\|_1
\end{aligned}
$$

From (6.1) we get that the probability of any bucket to provide frequency estimates above $\epsilon \|v\|_1$ is less than or equal to $1/4$ for $w = 4c/\epsilon$, when we are looking at the vector $v^{[\epsilon^{-1}]}$.

This proves the theorem since we have shown that:

$$
\begin{aligned}
\mathbb{P}[E \mid A \cup B] &= \mathbb{P}[E \mid A] + \mathbb{P}[E \mid B] \\
&\leq \frac{1}{4} + \frac{1}{4} \\
&= \frac{1}{2}
\end{aligned}
$$

which means that any specific bucket in the Count-Median Sketch will provide an estimate where the probability of $|\hat{v}_{i,j} - v_i| > \epsilon \|v\|_1$ is less than or equal to a half. Taking the median of estimates of each bucket over depth $d = O(\ln \delta^{-1})$ will then provide an L_1 error guarantee with failure probability δ by the same argument as in Theorem 2. □

As a consequence of giving the Count-Min Sketch and Count-Median Sketch equal amount of space, we expect that they perform equally fast and that their errors are within an additive approximation factor of the L_1-norm.

Precision

As can be seen in Figure 6.6 the precision of the Count-Median Sketch has changed drastically compared to the precision presented when using theoretical space. It no longer stays below the L_2 threshold for all distributions of data. In fact for $\alpha = \{0.5, 1\}$ the error is significantly higher than the L_2 threshold. This is as expected since we have reduced the width of the sketches from ϵ^{-2} to ϵ^{-1}. More important is whether the Count-Median Sketch stays below the L_1 threshold, as is expected by Theorem 10. This seems to be the overall case, but for $\alpha = 0.5$ and $\epsilon \leq 2^{-20}$ the error becomes greater than the allowed threshold. At this point the sketches reaches sizes near those of solving the problem exactly and the consequence of a having an error higher than the L_1 threshold becomes insignificant since one would choose the exact solution.

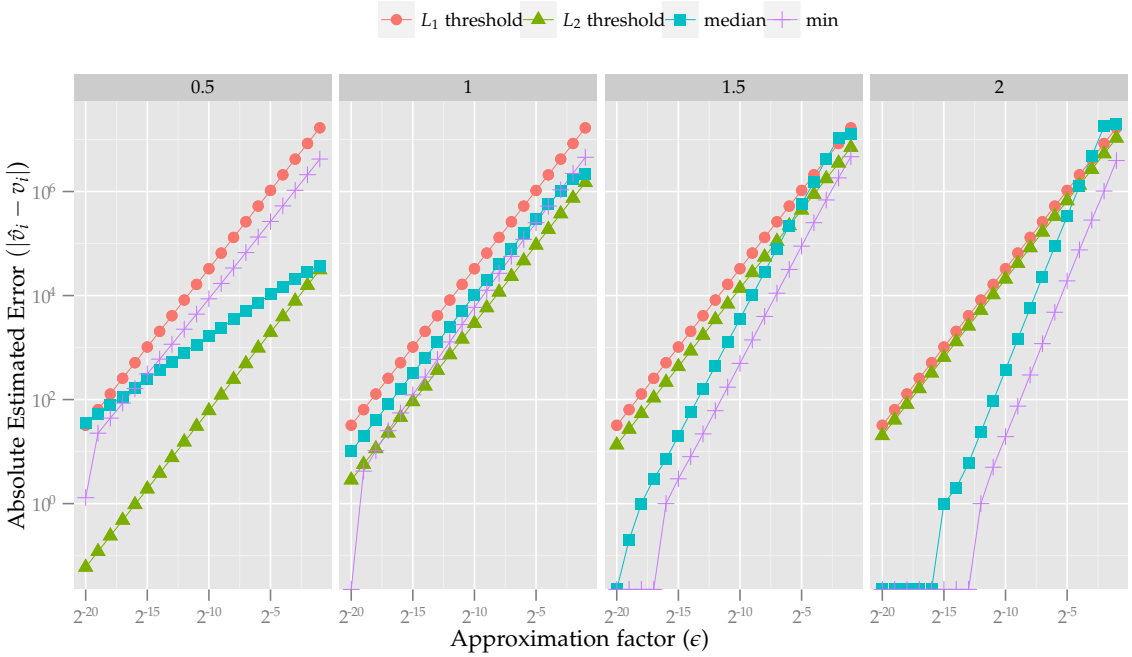


Figure 6.6: The precision of Count-Median Sketch and Count-Min Sketch on a zipfian distribution dataset with parameter $\alpha = \{0.5, 1.0, 1.5, 2.0\}$. The L_1 and L_2 thresholds are shown to be able to compare the sketches according to these.

For $\alpha = \{1.5, 2\}$ and $\epsilon = \{2^{-1}, 2^{-2}, 2^{-3}\}$ the error of the Count-Median Sketch is also a tiny fraction above the L_1 threshold. This is more surprising. One explanation of this behavior is that the depth is chosen according to the Count-Min Sketchs depth i.e. $d = \log_b \delta^{-1}$ whereas the theorem from above expects a depth of $d = \ln \delta^{-1}$. This implies that the Count-Median Sketch gets a depth of $d = 9$ for $\delta = 2^{-18}$, where it in theory should have $d = 13$. This explanation was tested in practice and showed to be true.

Overall we can conclude that the Count-Median Sketch does in fact uphold a L_1-norm guarantee, when choosing $w = O\left(\epsilon^{-1}\right)$. Comparing the precision of the two sketches, it is clear that for data with a near uniform distribution, the Count-Median Sketch provides smaller errors, whereas when the data becomes more and more skewed, the Count-Min Sketch provides the smallest error. This is not surprising since the Count-Median Sketch still has a relationship with the L_2-norm, which increases when the data becomes more skewed implying that the error increases as well.

Running times

The performance of the algorithms of the sketches can be seen in Figure 6.7 and Figure 6.8. Here the amount of operations per millisecond is shown for the update and query algorithms respectively. Overall both the update and queries of both sketches suffer from the same things as explained for the Count-Min Sketch in the last section.

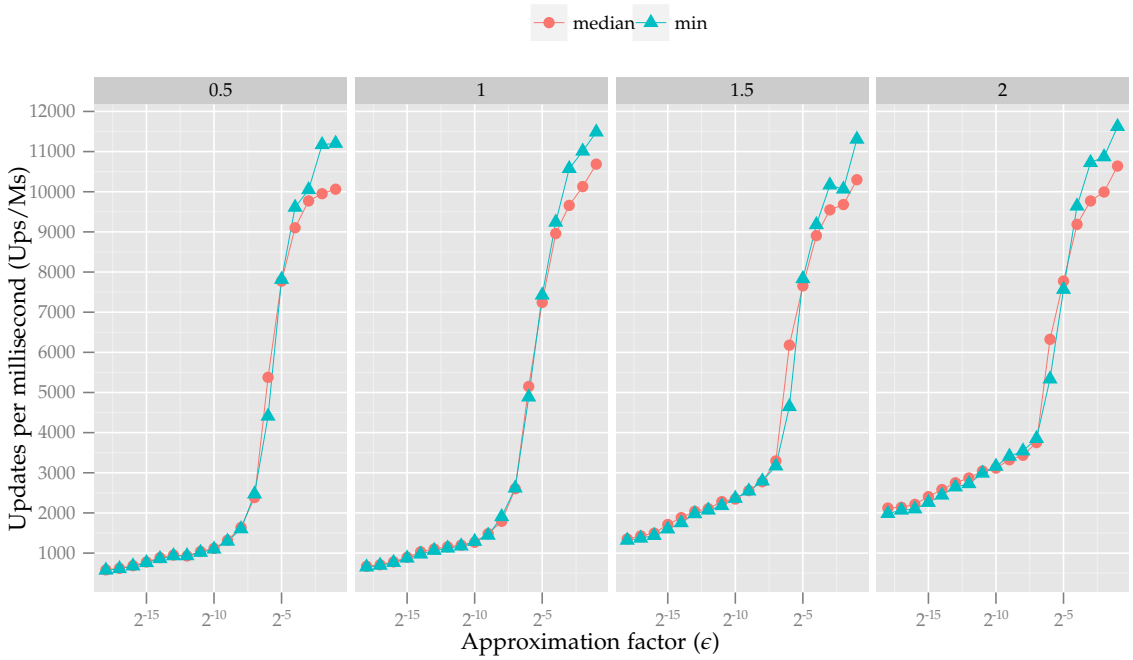


Figure 6.7: Updates per millisecond of Count-Min Sketch and Count-Median Sketch when the width w and depth d is fixed to be equal for both sketches.

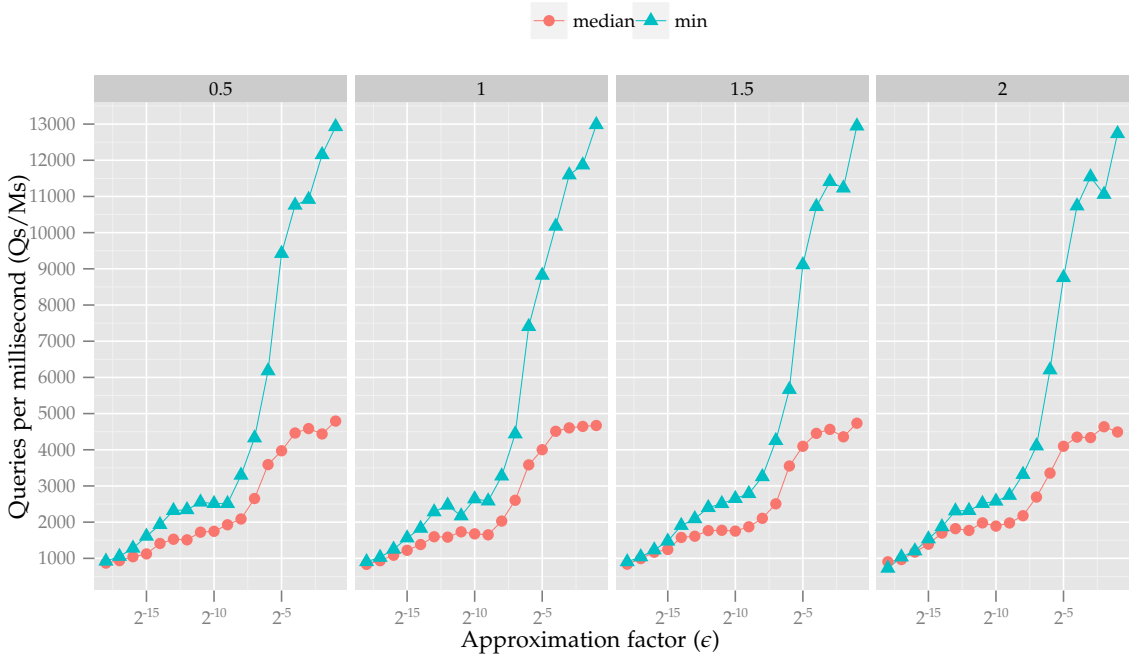


Figure 6.8: Queries per millisecond of Count-Min Sketch and Count-Median Sketch when the width w and depth d is fixed to be equal for both sketches.

The update throughput for both data structures are very similar. This is as expected, since the update algorithms of both structures are more or less equal, with the exception of the Count-Median Sketch calculating an extra sign for all updates. Surprisingly enough, the extra work does not seem to affect the running time significantly. It can in fact be observed that the Count-Median Sketchs update algorithm is faster than the Count-Min Sketch's for smaller choices of ϵ. From the data obtained from PAPI about cache misses and accesses, instructions, branch mispredictions, cycles etc. we were not able to derive, why this is the case. In fact they only show that the Count-Median Sketch use more instructions because of the extra sign calculation. Nonetheless, the difference in update throughput is in general so insignificant that we conclude they perform equally fast for $\epsilon < 2^{-4}$.

The query throughput shows a greater difference. This is mainly due to the fact that finding the median of the d estimated frequencies is more expensive than finding the minimum. Overall both sketches have queries supporting at least 1000 queries per millisecond.

Summing it up, we can conclude that the Count-Min Sketch and the Count-Median Sketch with width $w = O\left(\epsilon^{-1}\right)$ is indeed comparable, and that comparing the sketches according to space, precision and running times gives very similar results, where different data distributions determines which sketch performs the best. The only notable difference is in the running time of the query algorithms where the Count-Min Sketch in general seems to be faster than the Count-Median Sketch.

Common for both, as a consequence of their running times, precision, and space usage, is that they would be applicable as a black box mechanism to other algorithms e.g. the approximate heavy hitters problem.

6.2.3 Summary

In this section, we have experimented with two different sketches, namely the Count-Min Sketch and the Count-Median Sketch. At a first glance of the theoretical bound of both sketches, they did not seem very comparable, due to having different widths and different error guarantees. This was also shown to be true in Subsection 6.2.1, where it was found that, even though they both provided good data structures to estimate frequencies with errors according to L_1 or L_2, the differences in running time and space were significant.

We then showed that the Count-Median Sketch can in fact be shown to provide an error guarantee according to the L_1-norm, by changing the width to be equal to the width of a Count-Min Sketch. Experiments doing exactly this revealed that the two sketches are in fact comparable, and that they in general perform equally well for both the precision and the update algorithm. In practice the query algorithm of the Count-Min Sketch was shown to be faster than the Count-Median Sketchs query algorithm, likely due to it being easier to calculate the minimum compared to calculating the median.

Overall, both structures were small and fast, which make us conclude that they would both serve as good synopsis data structures and particularly in applications where a black box solution is needed for the frequency estimation problem.

6.3 Heavy Hitters

In this section we will describe experiments and results for 6 different implementations solving the approximate L_1 heavy hitters problem. First we will follow up on the last section, by testing the hierarchical data structure from Section 4.3 using Count-Min Sketches and Count-Median Sketches with equal width and depth. Next we will compare those results to the Hierarchical Constant-Min Structure solution from Section 4.6. All these solutions will then be compared to earlier results in the literature presented in Cormode and Hadjieleftheriou [7], specifically to their implementation of the hierarchical data structure using Count-Min Sketches. Finally we will further present results for hierarchical data structures using Count-Min Sketches and Count-Median Sketches with equal width and depth, but where the tree will be k-ary instead of binary. After analysing the performance of all solutions according to a single data distribution we will briefly have a section where the performance of all solutions are compared according to different aspects of skewness of the data distributions. Then we will briefly compare the space usage of all solutions, before a final section containing the summary of all results of the heavy hitters experiments will be given.

The heavy hitters experiments will generally look at three different aspects of each of the solutions. This is the precision of the query algorithm, the running times of the update and query algorithms and finally the space usage of the data structures used in the algorithms.

The precision of the query algorithms is tested by measuring special metrics of the resulting set. These metrics are denoted the **recall** and **false positives** of the resulting set. The recall is adopted from Cormode and Hadjieleftheriou [7] and is the percentage of true heavy hitters of the data, present in the resulting set. Such a metric was measured by knowing the exact frequency of a large portion of the highest frequencies of the data in advance, which enabled us to verify if all true heavy hitters were part of the resulting set obtained by the query algorithms.

The false positives of the resulting set, are the amount of elements which are not a true heavy hitters of the data. Such false positives can be present, since the frequency estimation algorithms used in the solutions, are all approximations. This implies that some error is introduced. The false positives metric can be derived from our experiments, due to the same reason as the recall can be derived.

Hence, from the above metrics, it is possible to measure if all true heavy hitters were found and furthermore measure the amount of the additional false heavy hitters that are returned. This is interesting as a measure of the precision of the algorithms, since it is important to know if the algorithms return the correct heavy hitters elements without providing too many extra elements. The metrics though only provide a good overview of the precision of the algorithms, when they are looked at in combination. A simple algorithm with 100% recall could just return all elements in the universe. Hence, the false positives are and important measure in order to determine that we did not return a lot of unwanted elements.

The ideal result of the above metrics come from comparing it according to an exact solution i.e. a solution keeping an exact count of all of the frequencies. A query on such

Parameter	Default Value	Description
ϕ	2^{-10}	Threshold factor
δ	2^{-2}	Failure probability
ϵ	$\frac{\phi}{2}$	Approximation factor
α	1	Zipfian data skew
m	$2^{31}-1$	Size of universe

Table 6.4: Default parameters chosen when nothing else is specified.

a data structure would result in a recall of 100% and 0 false positives. These numbers comes from the fact that all estimates are precise, which enables the query algorithm to choose precisely those that are true heavy hitters and no others. As argued in Section 4.1 such a solution is not reliable in practice when m becomes large, and one has to turn to approximation algorithms as those 6 solutions tested in this section to find the heavy hitters.

Still, the goal is the same. If the approximation algorithms can obtain a recall of 100% and 0 false positives, they can provide as good a result as an exact solution for the data tested. By the definition of an approximate L_1 heavy hitters algorithm, such an algorithm should in fact have 100% recall, since all true heavy hitters must be returned, where as the approximation is effectuated by allowing a relaxation of the false positives, such that no elements s_i with frequency $v_i < (\phi - \epsilon) \|v\|_1$ should be returned.

An important observation from the false positives metric for approximate solutions, is that returning false positives is not the same as saying that the algorithms failed to provide a resulting set of approximate L_1 heavy hitters. This is because the approximation algorithms are allowed to return some false positive by the definition above. An error of these algorithms only occurs whenever an element s_i with frequency $v_i < (\phi - \epsilon) \|v\|_1$ is present in the resulting set and this should by theory only happen with probability δ. To argue about whether the approximation algorithms actually do make errors, we also measured these errors, by checking whether the elements in the resulting set had a true frequency $v_i >= (\phi - \epsilon) \|v\|_1$.

Both the measurements of the precision and the running times will be run for artificially generated data from the zipfian distribution that varies over $\alpha = \{0.5, 1, 1.5, 2\}$, which means that the distributions of the data will go from almost uniform, towards a very skewed distribution. This should ensure that any effect resulting from the distribution of data should be visible in the experiments.

The measurements of the space usage will only be run for a single choice of $\alpha = 1$. This is because the space usage of any of the data structures are not affected by the distribution of the data.

General for all experiments in this section is that the size of the universe will be $m = 2^{31} - 1$. The default parameters are listed in Table 6.4, and is used unless otherwise specified.

In Figure 6.9 the PAPI measurements of the update algorithms is presented for $\alpha = 1$. These measurements will be used to compare and explain the running time between

each of the solutions throughout the next sections. The measurements for the query algorithms can be seen in Figure 6.10, but is generally of less interest, since queries in most cases will only be carried out a fraction of the times that updates do.

In Figure 6.11 the precision of all the solutions is plotted, likewise for $\alpha = 1$. These measurements will likewise be compared and explained throughout the next sections.

General for all plots are that *min* and *median* denotes the binary hierarchical solutions using Count-Min Sketches and Count-Median Sketches. The solution called *cormode* denotes the binary hierarchical solutions using Count-Min Sketches from Cormode and Muthukrishnan [8]. The Hierarchical Constant-Min Structure solution is denoted *const*, while the *k*-ary hierarchical solutions using Count-Min Sketches and Count-Median Sketches are denoted *kmin* and *kmedian* respectively.

In the next section we will follow up on the sketch experiments, by testing the same hierarchical data structure with two different black box solutions in the form of the Count-Min Sketch and the Count-Median Sketch.

6.3.1 Count-Min Sketch & Count-Median Sketch

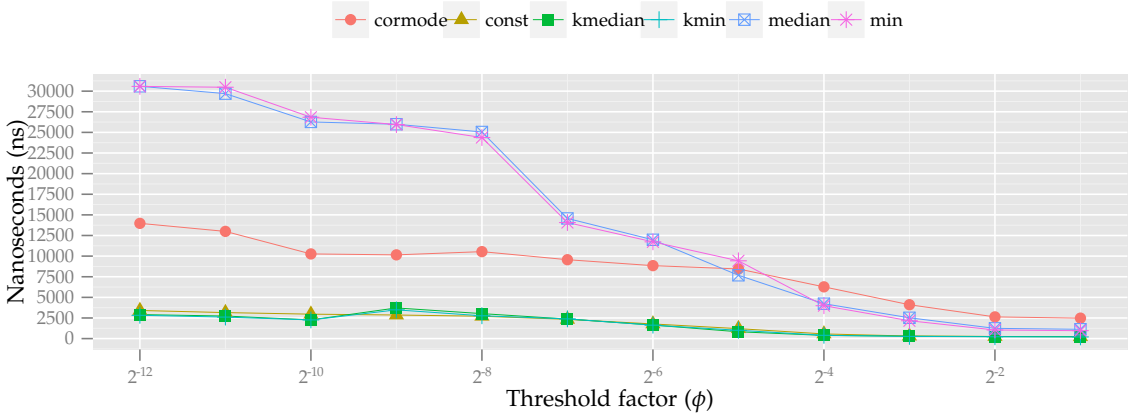
As was shown in Subsection 6.2.2, the Count-Median Sketch could provide guarantees according to the L_1-norm using the same space as the Count-Min Sketch. Changing the space of the Count-Median Sketch, according to the theoretical space of the Count-Min Sketch, provides us with two different frequency estimation algorithms that could be used in the hierarchical data structure from Section 4.3 to find approximate L_1 heavy hitters.

These two approximate L_1 heavy hitters data structures would be similar and almost use the same amount of memory. The precision of the algorithms is in theory similar, but it would be expected that the version using Count-Median Sketches would return more false positives, since the threshold of that algorithm would have to be adjusted, as described in Section 4.5. In the measurements from Figure 6.11, this is not be expected, since the threshold was not adjusted, due previous empirical studies [6, 7], saying that the version with Count-Median Sketches, should do approximately as well without any adjustments.

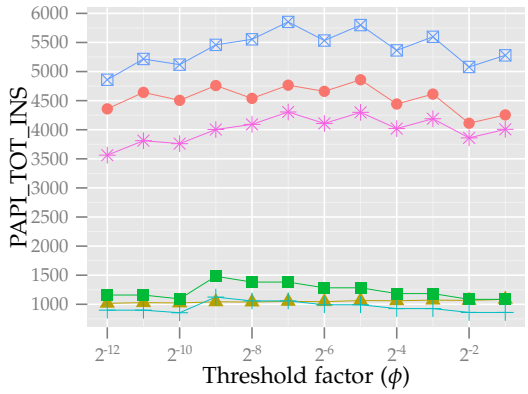
The running times of the query and update algorithms are expected to be to the results for the sketches in Subsection 6.2.2, as the extra overhead from updating and querying the hierarchical structure should be equal for both. So, we expect that the update algorithms perform equally well, while the query algorithm of the version with Count-Min Sketches should be faster than the Count-Median Sketches since calculating the median is more costly than finding the minimum in practice.

From Figure 6.9 and Figure 6.10 we can see that this is actually true in practice. Note that the hierarchical structure using Count-Min Sketches is denoted *min*, while the one using Count-Median Sketches is denoted *median*.

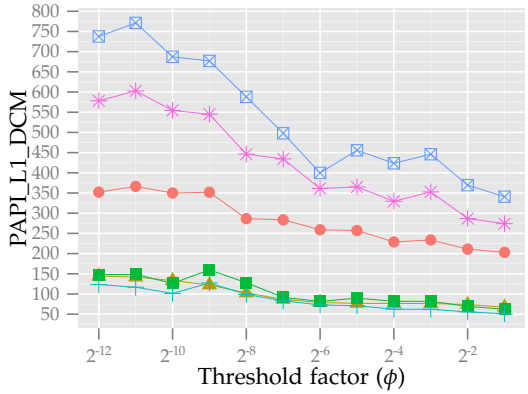
The running time of the update algorithms show that both solutions perform equally well over all ranges of ϕ. We saw a similar pattern for the sketches in Figure 6.7 where the amount of cache misses grew as ϵ decreased. Likewise, we get an increase in cache misses when ϕ decreases.



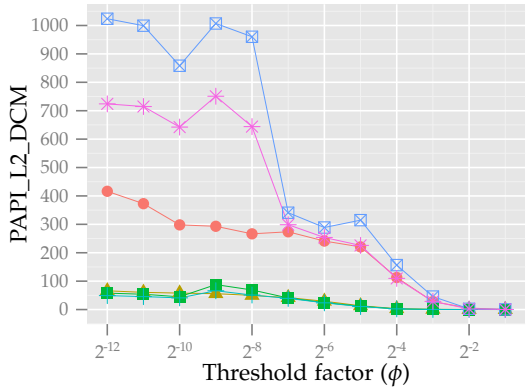
(a) Running time in nanoseconds



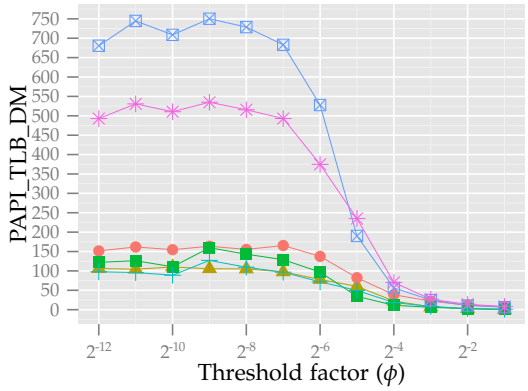
(b) Instructions Completed



(c) L1 Data Cache Misses

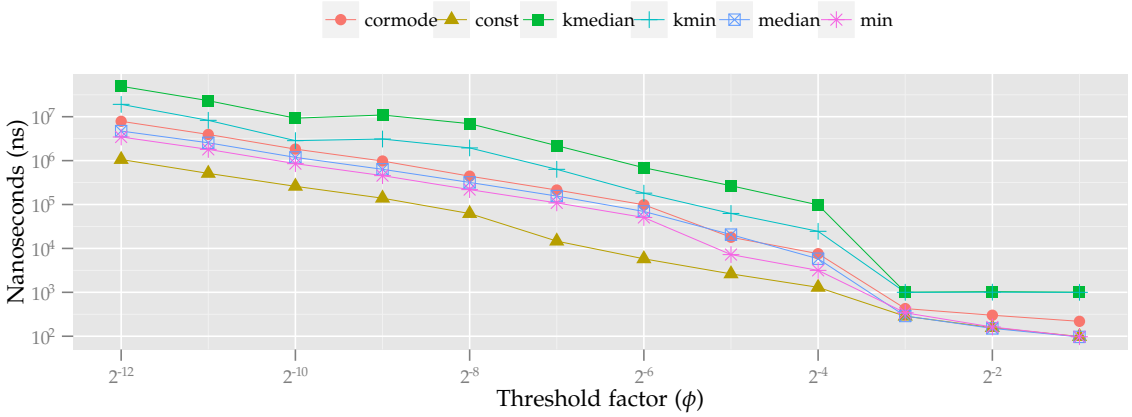


(d) L2 Data Cache Misses

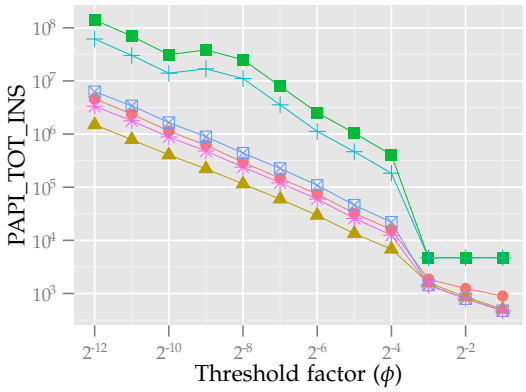


(e) TLB Cache Misses

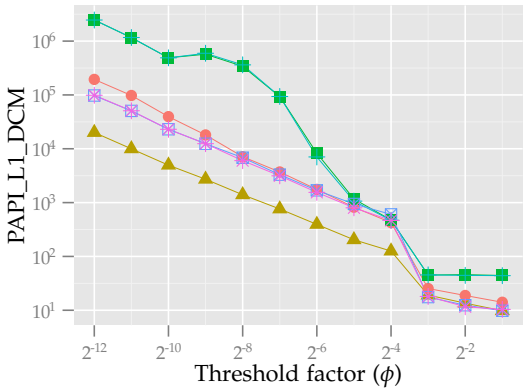
Figure 6.9: Update measurements where (a) shows the running time, (b) a plot of the amount of instructions carried out by an update algorithm, while (c-e) show the amount of different cache misses that occur during the update operation for the L1, L2, and TLB cache, respectively. All measurements are for data from a zipfian distribution with $\alpha = 1$.



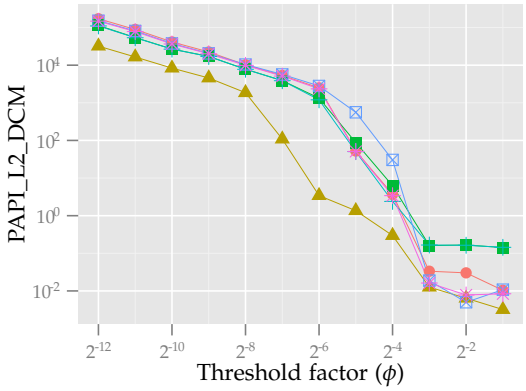
(a) Running time in nanoseconds



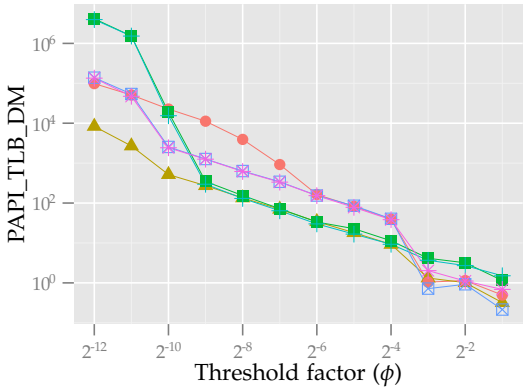
(b) Instructions Completed



(c) L1 Data Cache Misses

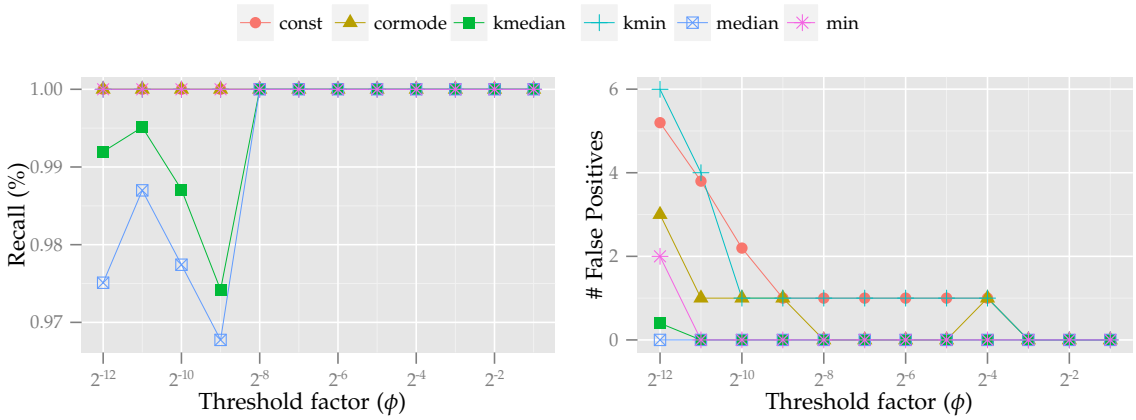


(d) L2 Data Cache Misses



(e) TLB Cache Misses

Figure 6.10: Query measurements where (a) shows the running time, (b) is a plot of the amount of instructions carried out by an query algorithm, while (c-e) show the amount of different cache misses that occur during the query operation for the L1, L2, and TLB cache, respectively. All measurements are for data from a zipfian distribution with $\alpha = 1$.



(a) Recall (%)

(b) # False Positives

Figure 6.11: Precision measurements of the heavy hitters query algorithms where (a) shows the recall i.e. percentage of true heavy hitters returned and (b) shows the amount of false positives returned in the resulting set.

For the TLB cache misses, the grow begins at $\phi = 2^{-4}$, this is due the fact that every sketch of the hierarchical tree begins to consume more space than the size of a page. As a consequence memory references to sketches begins to cause more than one TLB miss, and for $\phi \leq 2^{-7}$, all memory references to sketches tend to invoke a TLB cache miss. The L1 cache experiences a grow around $\phi = 2^{-6}$, at this point, one whole sketch cannot be stored in the L1 cache any more, which means that more L1 cache misses generally occur. Finally, the L2 cache misses also grows as ϕ and hence ϵ decreases.

The amount of instructions executed for an update, are expected to increase as ϕ decreases. This expectation comes from the fact that the decrease in ϕ should affect an increase in the depth of the internal sketches. For all choices of ϕ tested, the depths of the sketches are shown in Table 6.5. Here it can be observed that only every second decrease in ϕ affect the depth of the sketches.

From Figure 6.9b it can be observed that the amount of instructions generally does not increase as ϕ decreases. Instead the amount of instructions have a sawtooth behavior i.e. the amount of instructions increases and decreases for decreasing ϕ. An explanation for this behavior is that for every second decrease of ϕ, one more level will be using an exact count instead of a sketch. Updating the frequency exactly instead of in a sketch is significantly faster and requires less instructions, since only a single update is required, which is the reason we see the sawtooth behavior. Still, for all choices of ϕ the amount of instructions performed for both solutions are very similar, and can be thought of as almost constant.

Overall, we can conclude that the amount of instructions does not have any significant affect on the running time, and that the biggest influence of both update algorithms comes from the cache misses.

ϕ	$2^{-12}, 2^{-11}$	$2^{-10}, 2^{-9}$	$2^{-8}, 2^{-7}$	$2^{-6}, 2^{-5}$	$2^{-4}, 2^{-3}$	$2^{-2}, 2^{-1}$
d	10	9	8	7	6	5

Table 6.5: Depth of the Count-Min Sketches and Count-Median Sketches used for the heavy hitters solutions.

The query time of the two solutions show that the solution using the Count-Median Sketches is slower than the solution using Count-Min Sketches. This is exactly as expected, as a single Count-Median Sketch has a slower query than a Count-Min Sketch. The running time of a query increases as ϕ and ϵ decrease, which again is due to an increasing amount of cache misses.

Precisionwise, both solutions performs very well. From Figure 6.11 the recall and amount of false positives are shown. The solution using Count-Min Sketches has 100% recall as a consequence of always overestimating the frequencies while the amount of false positives is 0 for all choices of ϕ, except $\phi = 2^{-12}$ where 2 false positives are found. As noted in the initial description of the precision metrics, this is not the same a making an error, as looking further into the data of the experiments showed us that all returned elements was within $(\phi - \epsilon) \|v\|_1$.

The solution using the Count-Median Sketches has a 100% recall after $\phi = 2^{-9}$, and misses a few actual heavy hitters up until and before this point. The heavy hitters threshold of the implementation was not subtracted the potential $\epsilon \|v\|_2$ underestimate, implying that heavy hitters close to the threshold would potentially not be reported if they were underestimated. Still, almost all true heavy hitters are returned, and no false positives are included in the resulting set. Due to not having 100% recall, the implementation does not in a strict sense uphold the definition of a L_1 heavy hitters algorithm, but the precision is arguably almost as good as those that do uphold it.

An experiment with the solution using the Count-Median Sketch where the threshold was adjusted to $(\phi - \epsilon) \|v\|_1$ was also performed. This experiment resulted in a recall of 100%, but also an increase in the amount of false positives. No elements below the allowed $(\phi - \epsilon) \|v\|_1$ was found, but different data might. The adjusted threshold gave an extra overhead for the query time as more subtrees of the hierarchical structure are visited and more elements are returned.

Finally, to test the failure probability of the algorithms the precision tests were run several times with $\delta = 1/4$, implying that every fourth run should include an error. The observation of the experiment was that this was not the case as no errors were encountered for more than fifty runs. This result indicates that the failure probability of the heavy hitters algorithms is indeed an upper bound, much smaller in practice. Another reason for this result could be because the test data was generated using random weighted sampling which together with the random choices of hash functions imply that the behavior of the hash functions is truly random in practice.

6.3.2 Hierarchical Constant-Min Structure

The Hierarchical Constant-Min Structure mentioned in Section 4.6 is an approximate L_1 heavy hitters solution which to our knowledge has not previously been tested in practice.

This solution has lower space usage, faster update times, and expected faster query times than the two solutions tested in the previous section. Consequently, we expect it to outperform these solutions on both space and running times while having the same precision.

The expectation of the running time is built on the fact that, at each level of the tree, a constant sized sketch is kept. In our implementation of the algorithm, these are kept by storing one big consecutive block of memory containing the whole tree. This is an optimization compared to the solutions in the last sections, where an abstract sketch structure was kept for each level of the tree and where the memory of the sketch structures in general are not consecutively stored. The traversal of the tree for both update and query should due to the consecutively stored memory be very fast, since fewer cache misses should occur. Moreover since only a single entry per level of the tree is updated/queried compared to d entries for the earlier solutions, this should imply that the speed of the Hierarchical Constant-Min Structure should be considerably faster.

The update and query times of the Hierarchical Constant-Min Structure denoted *const* is shown in Figures 6.9, and 6.10 respectively. From the plots it is clear that the algorithm is also faster than the hierarchical structures using Count-Min Sketches and Count-Median Sketches in practice. This is because it performs less work, which can be seen from the number of instruction and because the amount of cache misses are significantly lower.

The precision of the Hierarchical Constant-Min Structure can be seen in Figure 6.11 and is very similar to the solutions presented earlier. Since the Hierarchical Constant-Min Structure internally uses Count-Min Sketches it has a recall of 100% for all choices of parameters. What is more interesting is that the solution generally seems more vulnerable to having false positives among the resulting set of the query algorithm. For $\phi \geq 2^{-4}$, one or more false positives are among the resulting set. This is likely due to the constant probability ($1/4$) of error for each estimate on all levels of the tree, which infers that extra queries are made to the bottom sketch. Due to the scaling of the bottom sketch, all estimates $v_i < (\phi - \epsilon) \|v\|_1$ is discarded, but a few extra false positives are found, due to the extra queries in the bottom sketch.

Overall this solution is a huge improvement in running time compared to the previous solutions, and the trade-off of having a few more false positives seems insignificant compared to the improved performance and lower space usage.

6.3.3 Cormode and Hadjieleftheriou

In the last two sections we have experimented with our own implementations. Since the heavy hitters problem is a heavily researched topic, others have also performed experiments.

To check how our own implementations performed compared to another implemen-

tation we also experimented with the hierarchical structure using Count-Min Sketches implemented in Cormode and Hadjieleftheriou [7]. We include the implementation in our own test runs, to make sure that the data is comparable, by running on the same setup.

Their findings for the hierarchical structure using Count-Min Sketches are that the space of the solution is small. As a consequence the update algorithm has a high throughput from 1500 to 2200 updates per millisecond ranging over $\phi \in [2^{-6}; 2^{-10}]$. They also measure precision by the recall of the algorithm and according to the number of true heavy hitters reported over the total number of reported elements. The recall is 100% for all experiments, since the Count-Min Sketches never underestimate frequencies. The second measure shows that as the data becomes more skewed, less and less false positives are returned in the resulting set. Still, over the range of $\phi \in [2^{-6}; 2^{-10}]$ the amount of true heavy hitters only account for between $73 - 87\%$ of the resulting set, which implies that some false positives are present. The problem with having provided these results in percentage is that, we cannot derive how many false positives this exactly is in practice and it is furthermore not mentioned whether any of the false positives are in fact errors.

The results from the article is not directly comparable with those provided in the earlier sections, since some crucial points are missing. In their testing setup, a fixed depth of 4 was chosen for all sketches in the hierarchy. This does not follow the theoretical depth of the sketches, which can be seen in Table 6.5, and this implies that the theoretical guarantees of the sketches are no longer true, as opposed to the implementations we provided in the earlier sections.

Furthermore, the hierarchical structure from their measurements is having a branching factor of 16 as opposed to the binary ones of our implementation. This will, as mentioned in the bottom of Section 4.3, lead to better space and update time with a trade-off on the query time.

Finally, from the code provided in Cormode and Hadjieleftheriou [7], it looks like the data they test on, is skewed in such a way that they only have elements with mass among the first 2^{20} elements, and not spread out on the full universe, of $2^{31} - 1$ elements. As a consequence only a fraction of the hierarchical tree structure will ever be visited. This could lead to improvements in running times, that actually would not be present, if the data would have been uniformly distributed among the whole universe, as is the case for our test data.

Next we will present the results of using their implementation with binary branching in our test setup. In Figure 6.11 the precision of the implementation can be seen. The implementation of the article is denoted *cormode*.

From Figure 6.11a the recall of the implementation can be observed. Here there is no real surprise, since the recall is a 100% over all runs, which is also expected since Count-Min Sketches are kept which never underestimate the frequencies. This is the same exact same results as for our hierarchical structure using Count-Min Sketches.

From Figure 6.11b, the number of false positives can be observed. Here it is notable that the number of false positives are just above our implementation for some choices of ϕ, with a difference of one element. This is certainly within a reasonable amount of false positives and probably significantly less than what is seen from the results of the article

[8]. None of the false positives are actual errors.

The running times of the query and update algorithm are expected to be more or less equivalent to our implementation as they have equal space, due to having sketches with equal depth and width.

Figure 6.9 shows among other things, the running time in nanoseconds per update. Here it can be observed that the implementation of the article has a faster running time than our implementation for most choices of ϕ. The faster running time, is a consequence of the *cormode* implementation having a lot less cache misses compared to our solution as can be observed in all the cache plots of the same figure.

At first this might seem strange, since the implementations are similar. One point is missing, in our implementation, the sketches are actually generic structures created as an abstract sketch object, in the *cormode* implementation the sketches are implemented inline without any abstractions. The consequence of having the sketch abstraction is that every sketch allocates its own block of memory, which does not necessarily gets consecutively stored. The *cormode* implementation is on the other hand stored along with the hierarchical tree, which means that the memory block of the whole tree is consecutively stored. This has a huge influence with respect to cache misses, since the prefetcher of the CPU is actually more likely to prefetch the next memory that should be used as opposed to our solution, where it cannot know the next block to use due to them not being stored consecutively. What is also notable is that the running time of the update algorithm is no longer nearly as fast as the results presented in Cormode and Hadjieleftheriou [7]. This comes as a natural consequence of having the correct theoretical parameters, a lower branching factor and by not tangling with the skewness of the data.

Figure 6.10 shows the running times of the query algorithm. The overall behavior of the query times are very similar, for their and our implementations. Still, our implementation is a factor faster than theirs over all distributions of data, which can be explained by the fact that our query implementation was implemented iteratively, while their query implementation is recursively implemented, giving an overhead due to extra function calls and stack buildings.

Overall, the algorithms perform similarly, even though differences are observed in running times for both the query and update algorithm. What can be taken from this sections is the fact that all the implementations implemented by us, could in fact become even faster if the sketches of the structures were implemented inline instead of using the abstraction of a sketch. The experiments of Cormode and Hadjieleftheriou [7] show that even though the theoretical bounds of the algorithms are not followed, the algorithms are still precise with decent running times.

6.3.4 k-ary Hierarchical Structures

As mentioned in the earlier section, the experiments from Cormode and Hadjieleftheriou [7] were performed with a branching factor of 16. In general the hierarchical structure from Section 4.3 can be created for any branching factor, as long as the ranges in the tree is adjusted accordingly to the branching factor. In this section we will present results

for a k-ary hierarchical structure using Count-Min Sketches and Count-Median Sketches with branching factor $k = 256$. The choice of k was made inorder to achieve an update algorithm, which could be comparable in running time to the Hierarchical Constant-Min Structure solution in practice.

As noted in the theoretical sections, the implication of a k-ary hierarchical structure is that the space usage and update time improves, while the query time becomes worse as k rises. Hence, we expect that the practical results of the algorithms shows exactly this behavior compared to the binary solutions.

The running time of both the update and query algorithm can be seen in Figure 6.9 and Figure 6.10 respectively, where the structure using Count-Min Sketches is denoted *kmin* and the one using Count-Median Sketches is denoted *kmedian*.

From the plots of the running times of the update algorithms, it becomes clear that both the k-ary tree using Count-Min Sketches and Count-Median Sketch are significantly faster than their binary alternatives over all range of ϕ. This is a consequence of having fewer levels in the tree, implied by the branching factor. Fewer levels gives fewer cache misses and fewer instructions for the k-ary solutions compared to their binary counterparts.

The cache misses, instructions and running time of the update algorithm are in fact comparable to the Hierarchical Constant-Min Structure solution, which makes the solution applicable in practice, due to being able to handle hundreds of thousands of updates per seconds.

An odd thing about the update running times of the k-ary solutions are the drop in running time between $\phi = 2^{-9}$ and $\phi = 2^{-10}$. Since ϕ and hence ϵ decreases, the depth of the internal sketches increases. The reason for the drop in running time is found in the amount of sketches kept among the levels in the tree. The tree only consists of four levels due to its branching factor. Three of these are sketches up till $\phi = 2^{-9}$. At which point it becomes more feasible to hold an exact count of the second level and hence the tree is reduced to having only two sketches. This improves the running time of the update algorithm, since it is much cheaper to do a single memory access in an array than doing d accesses in a sketch.

What is gained in update speed should be lost in query speed, according to theory. From the running time of the query algorithm we can see that this is in fact true. The query time of both algorithms becomes significantly slower than any other algorithm, due to the fact that a lot of queries have to be carried out as a consequence of the branching factor. This implies a lot of extra instructions and L1 cache misses, since large parts of the tree have to be queried in order to find the heavy hitters. As was the case for the binary alternatives the k-ary tree with Count-Min Sketches has a faster query time than the one with Count-Median Sketches, this is again explained by the fact that computing the median takes more time than computing the minimum in practice. The query times are generally so much worse that any application using it in practice would have to consider how to handle the major delay that such a query operation would add. Comparing it to the Hierarchical Constant-Min Structure solution, the Hierarchical Constant-Min Structure is much faster, and would in any case be the preferable solution in practice according to the running times.

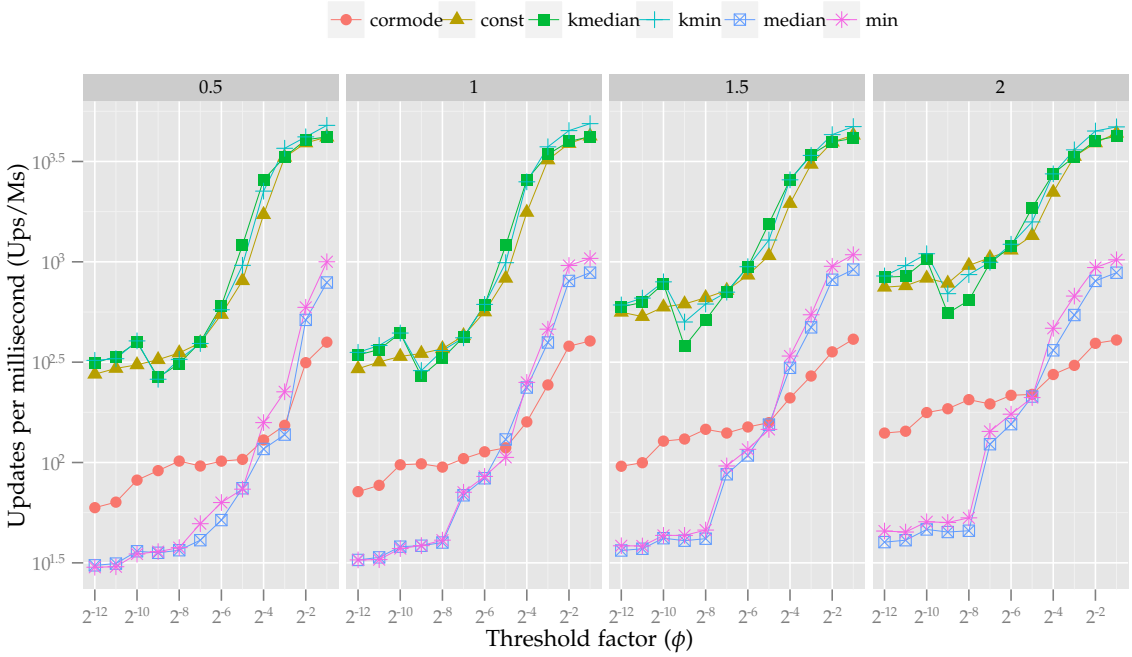


Figure 6.12: Updates per milliseconds shown for all heavy hitters solutions.

The precision of the k-ary trees can be seen in Figure 6.11. Since the solution using Count-Min Sketches does not underestimate its frequencies, it have a recall of 100%. The solution using Count-Median Sketches have the same problem as the binary solution using Count-Median Sketches i.e. it misses a couple percent of the true heavy hitters as ϕ decreases. The amount of false positives of the solution using Count-Min Sketches is generally the same as for the Hierarchical Constant-Min Structure solution, that is a few false positives is returned. For the solution using Count-Median Sketches the behavior is the same as the binary alternative, namely that no false positives are returned, except for a single one for $\phi = 2^{-12}$. Still, none of the false positives were actual errors. The solution with Count-Median Sketches was also tested with adjusted threshold $(\phi - \epsilon)\,\|v\|_1$, and generally showed the same results as for binary alternative described in Subsection 6.3.1. Overall the precision measurements shows that precisionwise both k-ary solutions are very applicable in practice.

6.3.5 Data Distributions

One point missing from all measurements above is the distribution of data. For all experiments above the data was zipfian distributed with parameter $\alpha = 1$. How does the algorithms behave speed- and precisionwise as the data gets distributed differently? To argue about that, the measurements from earlier sections was performed for $\alpha = \{0.5, 1, 1.5, 2\}$ i.e. for distributions ranging from almost uniform, to highly skewed.

From the experiments on the sketch implementations, we found that skewed distri-

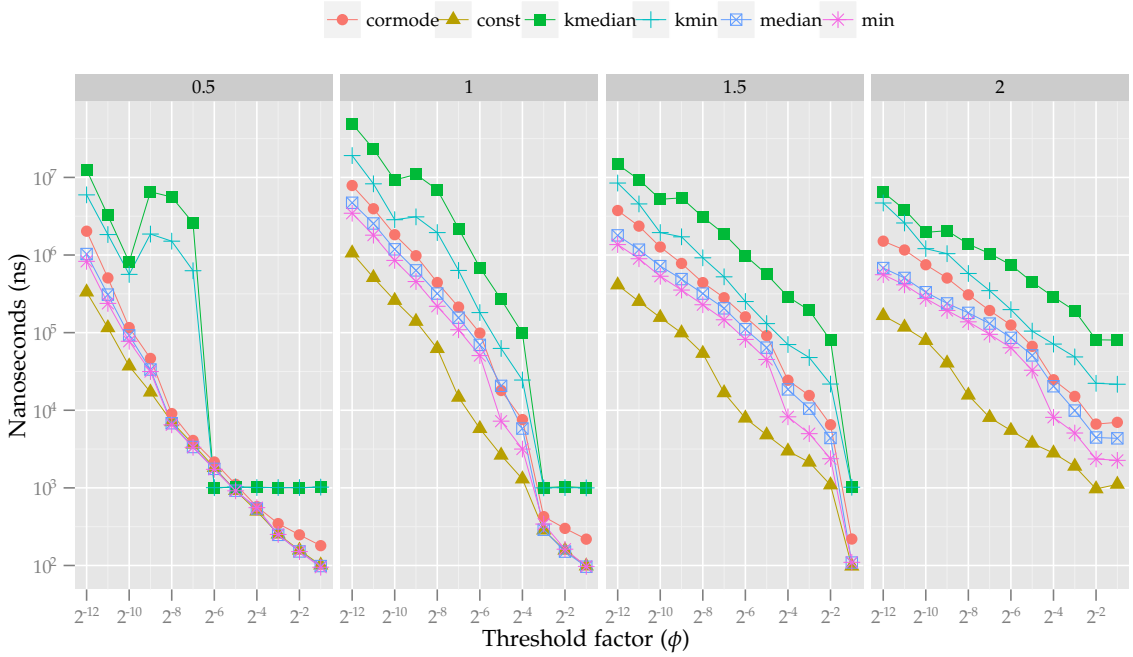


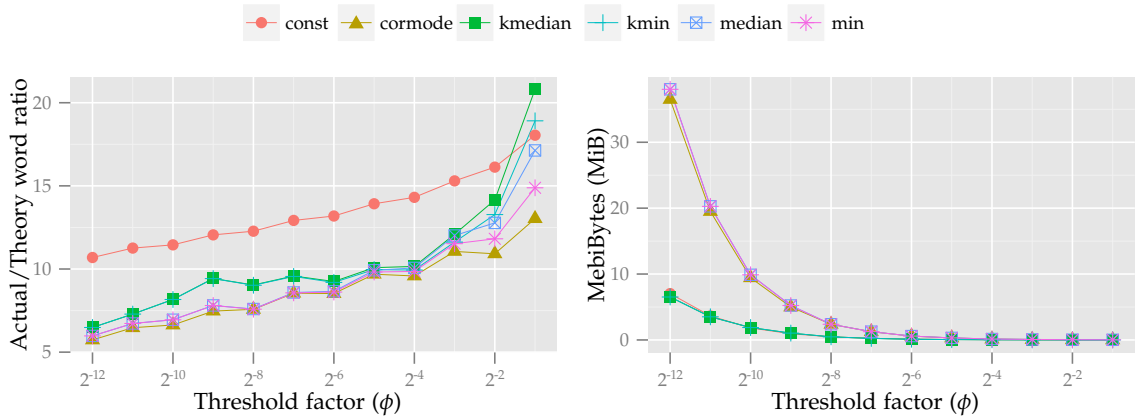
Figure 6.13: Running time for the query method in nanoseconds for all heavy hitters solutions.

butions affected the running time of the update algorithm of the sketches but not the query time. Since we use the sketch abstractions in all implementation it is only natural that skew should affect the update time of the heavy hitters solutions as well. Even though the query time of the sketches are not affected by the skew of data, the heavy hitters solutions actually are expected to be. This is due to the fact that, the more skewed data is, the more heavy hitters should be found for larger values of ϕ.

Hence, we should see that the query time for low values of α generally are faster, since almost no heavy hitters are found. As α grows, so should the amount of heavy hitters within the threshold of our experiments and this will have a negative effect on the running time of the query algorithm, since more queries have to be carried out throughout the trees.

From Figure 6.12 the updates per milliseconds over the different data distributions can be seen. Exactly as expected, the updates per milliseconds generally seems to increase as the data get more and more skewed. This is due to the fact that as data becomes more skewed, the probability of an element to be updated multiple times in a row increases, and hence the probability of having the memory present in at least the L2 cache increases. This is also what we see from the measurements, where the L1 cache misses stays approximately the same, whereas the L2 cache misses lowers significantly, as data becomes more skewed.

From Figure 6.13 the running times for the query algorithms are shown in nanoseconds. As noted there should occur changes in the query times as the data becomes more



(a) Word ratio

(b) Space usage

Figure 6.14: Space usage of the different algorithms where (a) shows the word ratio when the actual space usage is divided by the theoretical one, (b) shows the actual space usage in mebibyte (MiB), where *const* and *k{median,min}* are the lower ones.

skewed. This also seems to be the case. For $\alpha = 0.5$, no actual heavy hitters are found for any choice of ϕ, hence the traversal of the tree should in general stop for all solutions, before reaching the bottom of the tree. For $\alpha > 0.5$, heavy hitters begin to exist in the data for some of the ranges of ϕ. This implies that the heavy hitter paths of the trees have to be fully explored causing more queries on the tree. This is the main reason why the distribution of data implies a difference in query time.

With regards to precision, the distribution of data also has an impact. For $\alpha = \{0.5, 1.5, 2\}$ all solutions have 100% recall and no false positives. Only for $\alpha = 1$ the picture is different and this is the case that was analyzed for every solution in the earlier sections. The main reason for this is that for $\alpha = 1$, the most heavy hitters are present in the data, while a lot of the elements still have very similar frequencies. This seems to imply more false positives since the error introduced by the sketches, make the heavy hitters algorithm consider more elements.

Overall the skew of data seems to have an impact on the performance and precision of the solutions. The impact is though of such character that the overall conclusions from the analysis of the experiments on the individual algorithms from the last sections also is true for general distributions of data.

6.3.6 Space

The space usage for each of the algorithms is shown in Figure 6.14. A general thing to note about the space usage is that it is not affected by the distribution of the data, for which the algorithm it tested on, as was shown to be the case for both the update and query algorithm in the last section. The plots in Figure 6.14 are for data with $\alpha = 1$.

To verify that the space usage of the algorithms follow those of the theory, a plot of the space usage in words divided by theoretical big-O space usage is seen in Figure 6.14a. Generally the resulting ratio of each of the algorithms should as a consequence converge towards a constant value. From the plot this can be argued to be true, since the ratio seems to settle as ϕ decreases. The inclination can be explained by the extra amount of space used to store extra implementation specific structures, as well as the uneven change in the number of rows in the sketches, when the depth increases, versus the number of levels counted exactly.

Hence, we conclude that the practical space usage of the algorithms, generally follow the theoretical space. Next we look at the actual space usage of the algorithms.

What is interesting from Figure 6.14b is the fact that the algorithms can be split in two groups according to their space usage. The first group is the group of the hierarchical structures using the Count-Min Sketch and Count-Median Sketch denoted *min* and *median* and the solution from Cormode and Hadjieleftheriou [7] denoted *cormode*. These solutions use more memory than the other solutions since they theoretically keep a sketch for each of the levels in a binary tree, which for a universe of $m = 2^{31} - 1$, means 31 levels. In practice the counts of the first levels of the tree are kept exactly until the point where it in terms of space becomes feasible to use sketches. Since the width and depth of the sketches are kept equal for all three of these solutions, they end up using the same amount of space.

The second grouping consist of the Hierarchical Constant-Min Structure and the two k-ary hierarchical data structures using Count-Min Sketches and Count-Median Sketches. The latter algorithms use significantly less space, since the height of the tree is smaller because of the branching factor. With a branching factor of 256 the height of the tree is 4. The former, Hierarchical Constant-Min Structure, has a tree of height 31, but each of the levels only contain a sketch of constant size, plus a slightly larger sketch at the bottom, which is scaled appropriately to answer all queries performed on it. In practice the total amount of non-constant sized sketches becomes approximately the same for the Hierarchical Constant-Min Structure solution and the k-ary tree structure, which implies that the space usage also becomes approximately the same.

Comparing the groups, it is clear that the space usage of the second grouping increases at a much lower rate than the first grouping, as the heavy hitter threshold, ϕ, decreases. Still, the first grouping use less than 40 MiB and the second grouping less than 10 MiB for the choice of parameters to solve the approximate L_1 heavy hitters problem with very good precision. This space usage is an extreme improvement compared to solving the problem exactly, for which at least 16 GiB of space would be needed.

6.3.7 Summary

In this section we have performed several experiments for 6 different implementations of solutions solving the approximate L_1 heavy hitters problem.

In Section 6.2, we found that both the Count-Min Sketch and the Count-Median Sketch was applicable as a black box solution for the frequency estimation problem with error bounds on estimates according to the L_1-norm. As a consequence both sketches

were used in the hierarchical structure described in Section 4.3 to solve the approximate L_1 heavy hitters problem in the *Strict Turnstile Model*. They were furthermore compared to a solution from previous work [7], which showed to be faster than our implementations for the update algorithm, due to alignment of memory and slower for the query algorithm due to our iterative implementation opposed to their recursive one. Common for all where that the updates performed per millisecond were reasonably high, but still significantly lower than other solutions tested.

Furthermore they all used significantly more space than other solutions. One thing to take from the comparisons was the fact that implementing the sketches inline instead of using an abstraction of a sketch, implied faster algorithms.

Next experiments on the Hierarchical Constant-Min Structure was run, which to our knowledge never had been done before in practice. This structure theoretically had better bounds than any of the above solutions and also showed to be significantly faster in practice for both the update and query algorithms. Furthermore it used significantly less space, since only constant sized sketches were kept in the hierarchical structure. As such, this solution showed to be the most impressive structure to solve the approximate L_1 heavy hitters problem, and the only way to produce other structures that could be comparable in update time and space, but significantly worse in query time, was to change the branching factor of the hierarchical data structure significantly.

Doing this gave two more structures with very similar update running times compared to the Hierarchical Constant-Min Structure, but sadly the queries became significantly worse than any other structures.

The precision of all solutions was also tested. Here it was shown that all solutions performed very decent with respect to finding heavy hitters and not too many false positives. One of the reasons for this was that the solutions all used the theoretically correct parameters. An earlier practical experiment in literature [7] further enlighten that loosening the choices of parameters did not effect the precision that much in practice.

Finally the space usage of all solutions was shown and it was clear that all of them would be a great improvement opposed to finding heavy hitters using an exact count solution.

Overall, the Hierarchical Constant-Min Structure showed to perform the best and any streaming application with the need of a approximate L_1 heavy hitters solutions, should favor this solution, compared to the other tested.

Chapter 7

Conclusion

In this thesis we have analyzed and experimented with solutions to the approximate L_1 heavy hitters and frequency estimation problems in the *Strict Turnstile Model*. The conclusions for each of the algorithms are as follows.

For the frequency estimation problem, the Count-Min Sketch is optimal in space and has a near-optimal update time with an additive L_1-norm error. The experiments showed the Count-Min Sketch to be efficient at estimating frequencies with fast update and query times. The sketch has a low space usage and guarantees that frequencies are never underestimated.

The Count-Median Sketch with an additive two-sided L_2-norm error uses more space than the Count-Min Sketch, but is optimal in space and has a near-optimal update time. The experiments featured two versions with different depths, one with constants and one where we ignore constants. Removing the constants showed to decrease the space usage, update, and query time significantly, without sacrificing its error guarantee. We also showed that the Count-Median Sketch has a L_1-norm error guarantee by changing the dimensions of the sketch to be equal to those of the Count-Min Sketch. The experiments showed that the L_1-norm error guarantee made the Count-Median Sketch comparable with the Count-Min Sketch, with the differences being a slower query and the precision between the two sketches varying for different data distribution. To sum up, we showed that the Count-Median Sketch is able to provide guarantees according to the L_1-norm and L_2-norm with different space usages.

We conclude that the frequency estimation solutions are useful in practice as they are fast, provide good guarantees, and uphold their theoretical bounds.

For the heavy hitters problem, the Hierarchical Count-Min Structure ensures that all heavy hitters are found because of the underlying sketch. The solution is not optimal in space nor in update time. The experiments showed that the algorithm provided good results without including to many false positives, but with slow throughput. The results are good, but the update times were among the slowest we compared against.

The Hierarchical Count-Median Structure is neither optimal in space nor update time. The analysis showed that it used more space than the Hierarchical Count-Min Structure, but by using modified Count-Median Sketches the structure used space, and provided

a L_1-norm guarantee, equivalent to the Hierarchical Count-Min Structure. The experiments showed a precise solution with update times equivalent to the Hierarchical Count-Min Structure, but slower query times. Depending on how the threshold of the structure is adjusted it has a trade-off between finding all heavy hitters, but lots of false positives versus finding a few false positives, but potentially missing some heavy hitters.

The k-ary versions of the Hierarchical Count-Min Structure and Hierarchical Count-Median Structure obtains a smaller depth of the hierarchical structure. The experiments showed that the change in depth decreased the update time by more than a factor of 10 and decreased the space usage, compared to their binary counterparts. The disadvantage was a slower query. Consequently, increasing the branching factor is only desirable when it is acceptable to have a slow query.

The Hierarchical Constant-Min Structure provides an expected good query time, fast update time and optimal space usage. The experiments showed it to have the fastest query time while the update time and space usage were comparable to the k-ary solutions. Thus, this heavy hitters algorithm provides the overall best performance and is highly applicable for streaming settings with several hundreds of thousands updates per second.

Generally, we did not observe any errors for the heavy hitters solutions, which was unexpected as the solutions provide approximate results with a small probability of failure. Furthermore, all algorithms showed to provide precise results and to work as described in theory.

The results of the experiments with the recently introduced Hierarchical Constant-Min Structure are important as they show that this construction does indeed provide good results in practice. The experiments showed that the expected query time is indeed fast, and better than what is gained by using a k-ary solution with similar update times.

To conclude, this thesis is a collection of solutions in the *Strict Turnstile Model* to the frequency estimation problem and solutions for the heavy hitters problem using sketches. It provides an overview of the theory and bounds for each of the solutions, practical experiments with their implementations, and a theoretical and practical comparison of the solutions.

7.1 Future Works

Even though this thesis has studied the problem of estimating frequencies and finding heavy hitters extensively, some further work on the subject is still available. In the following we list some of the work, which could be interesting to address in the future.

Generic implementations may cause the algorithms to suffer from cache penalties. The code for the heavy hitters algorithms could be rewritten to handle the sketches specifically, as the code from Cormode and Hadjieleftheriou [7], which should remove the gaps between our implementation the implementation from Cormode and Hadjieleftheriou [7].

A L_2 heavy hitters algorithm using Count-Median Sketches could be interesting to measure in practice. An L_2 heavy hitters algorithm would be using quite a lot of space compared to the L_1 heavy hitters solutions, but it would still be interesting to see if such solutions would be efficient in practice.

The update time lower bound for both the heavy hitters problem and the frequency estimation problem was shown to have a gap between best known upper bounds and the lower bounds. Hence, an open problem exists whether it is possible to provide a matching upper bound for the update algorithms and if so, whether the space optimality can still be kept.

Real world data would be a good practical example to test on, to find out if heavy hitters from for example a DoS attack could be detected. It could also be interesting to simulate traffic in real time, to find out if the implementations would be fast enough for real world usage.

Tough data that would result in errors was never found. It might be difficult to find data that would make the algorithms fail with theoretical bounds. One way to improve space and running times of the algorithms, would be to loosen the parameters and hence leave the theoretical guarantees in order to find a trade-off where the algorithms performs well, while still finding all heavy hitters and not too many false positives and errors.

A tighter bound on the Count-Median Sketch could be done using Lemma 6. This would have reduced the theoretical depth of the Count-Median Sketch significantly.

Glossary

Delete-Min An operation that can be performed on a Min-Heap, which will delete and return the minimum element stored in the heap and heapify the remaining heap, in order for the heap to again uphold the heap property. 30

DoS attack A denial-of-service (DoS) attack is an attempt to make a machine or network resource unavailable to its intended users, such as to temporarily or indefinitely interrupt or suspend services of a host connected to the Internet. In the simples form the attack comes from the same IP address. . 1, 91

Min Heap A heap is a specialized tree-based data structure that satisfies the heap property: If A is a parent node of B then the key of node A is ordered with respect to the key of node B with the same ordering applying across the heap. For the sake of the min heap this ordering is ascending. 30

Pigeonhole Principle The pigeonhole principle states that if n items are put into m containers, with $n > m$, then at least one container must contain more than one item. 8, 14, 28

List of Tables

List of Figures

List of Theorems

Bibliography

[1] Radu Berinde, Piotr Indyk, Graham Cormode, and Martin J. Strauss. Space-optimal heavy hitters with strong error bounds. *ACM Trans. Database Syst.*, 35(4): 26:1–26:28, October 2010. ISSN 0362-5915. doi: 10.1145/1862919.1862923. URL http://doi.acm.org/10.1145/1862919.1862923.

[2] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, August 1973. ISSN 0022-0000. doi: 10.1016/S0022-0000(73)80033-9. URL http://dx.doi.org/10.1016/S0022-0000(73)80033-9.

[3] Robert S. Boyer and J. Strother Moore. Mjrty - a fast majority vote algorithm, 1982.

[4] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions (extended abstract). In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC '77, pages 106–112, New York, NY, USA, 1977. ACM. doi: 10.1145/800105.803400. URL http://doi.acm.org/10.1145/800105.803400.

[5] Moses Charikar, Kevin Chen, and Martin Farach-Colton. *Automata, Languages and Programming: 29th International Colloquium, ICALP 2002 Málaga, Spain, July 8–13, 2002 Proceedings*, chapter Finding Frequent Items in Data Streams, pages 693–703. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-540-45465-6. doi: 10.1007/3-540-45465-9_59. URL http://dx.doi.org/10.1007/3-540-45465-9_59.

[6] Graham Cormode. Sketch techniques for approximate query processing. In *Synposes for Approximate Query Processing: Samples, Histograms, Wavelets and Sketches, Foundations and Trends in Databases. NOW publishers*, 2011.

[7] Graham Cormode and Marios Hadjieleftheriou. Finding frequent items in data streams. *Proc. VLDB Endow.*, 1(2):1530–1541, August 2008. ISSN 2150-8097. doi: 10.14778/1454159.1454225. URL http://dx.doi.org/10.14778/1454159.1454225.

[8] Graham Cormode and Shan Muthukrishnan. *Summarizing and Mining Skewed Data Streams*, chapter 5, pages 44–55. doi: 10.1137/1.9781611972757.5. URL http://epubs.siam.org/doi/abs/10.1137/1.9781611972757.5.

[9] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, April 2005. ISSN 0196-6774. doi: 10.1016/j.jalgor.2003.12.001. URL http://dx.doi.org/10.1016/j.jalgor.2003.12.001.

[10] Graham Cormode and Shan Muthukrishnan. What's hot and what's not: Tracking most frequent items dynamically. *ACM Trans. Database Syst.*, 30(1): 249–278, March 2005. ISSN 0362-5915. doi: 10.1145/1061318.1061325. URL http://doi.acm.org/10.1145/1061318.1061325.

[11] Graham Cormode and Shan Muthukrishnan. Approximating data with the count-min sketch. *IEEE Software*, 29(1):64–69, Jan 2012. ISSN 0740-7459. doi: 10.1109/MS.2011.127.

[12] Graham Cormode, Theodore Johnson, Flip Korn, Shan Muthukrishnan, Oliver Spatscheck, and Divesh Srivastava. Holistic udafs at streaming speeds. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 35–46, New York, NY, USA, 2004. ACM. ISBN 1-58113-859-8. doi: 10.1145/1007568.1007575. URL http://doi.acm.org/10.1145/1007568.1007575.

[13] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Frequency estimation of internet packet streams with limited space. In *Proceedings of the 10th Annual European Symposium on Algorithms*, ESA '02, pages 348–360, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-44180-8. URL http://dl.acm.org/citation.cfm?id=647912.740658.

[14] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25 (1):19–51, October 1997. ISSN 0196-6774. doi: 10.1006/jagm.1997.0873. URL http://dx.doi.org/10.1006/jagm.1997.0873.

[15] Michael J. Fischer and Steven L. Salzberg. *Finding a Majority Among N Votes*. Defense Technical Information Center, 1982. URL https://books.google.dk/books?id=0aSUNwAACAAJ.

[16] Phillip B. Gibbons and Yossi Matias. Synopsis data structures for massive data sets. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '99, pages 909–910, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics. ISBN 0-89871-434-6. URL http://dl.acm.org/citation.cfm?id=314500.315083.

[17] Anna Gilbert and Piotr Indyk. Sparse recovery using sparse matrices. *Proceedings of the IEEE*, 98(6):937–947, June 2010. ISSN 0018-9219. doi: 10.1109/JPROC.2010.2045092.

[18] Hossein Jowhari, Mert Sağlam, and Gábor Tardos. Tight bounds for lp samplers, finding duplicates in streams, and related problems. In *Proceedings of the Thirtieth*

ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '11, pages 49–58, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0660-7. doi: 10.1145/1989284.1989289. URL http://doi.acm.org/10.1145/1989284.1989289.

[19] Daniel M. Kane, Jelani Nelson, Ely Porat, and David P. Woodruff. Fast moment estimation in data streams in optimal space. In *Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing*, STOC '11, pages 745–754, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0691-1. doi: 10.1145/1993636.1993735. URL http://doi.acm.org/10.1145/1993636.1993735.

[20] Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28(1):51–55, March 2003. ISSN 0362-5915. doi: 10.1145/762471.762473. URL http://doi.acm.org/10.1145/762471.762473.

[21] Kasper Green Larsen, Jelani Nelson, and Huy L. Nguyen. Time lower bounds for nonadaptive turnstile streaming algorithms. *CoRR*, abs/1407.2151, 2014. URL http://arxiv.org/abs/1407.2151.

[22] Kasper Green Larsen, Jelani Nelson, Huy L. Nguyen, and M. Thorup. Heavy hitters via cluster-preserving clustering. *ArXiv e-prints*, April 2016.

[23] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 346–357. VLDB Endowment, 2002. URL http://dl.acm.org/citation.cfm?id=1287369.1287400.

[24] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proceedings of the 10th International Conference on Database Theory*, ICDT'05, pages 398–412, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-24288-0, 978-3-540-24288-8. doi: 10.1007/978-3-540-30570-5_27. URL http://dx.doi.org/10.1007/978-3-540-30570-5_27.

[25] Peter Bro Miltersen, Noam Nisan, Shmuel Safra, and Avi Wigderson. On data structures and asymmetric communication complexity. In *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing*, STOC '95, pages 103–111, New York, NY, USA, 1995. ACM. ISBN 0-89791-718-9. doi: 10.1145/225058.225093. URL http://doi.acm.org/10.1145/225058.225093.

[26] Jayadev Misra and David Gries. Finding repeated elements. Technical report, Ithaca, NY, USA, 1982.

[27] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, NY, USA, 1995. ISBN 0-521-47465-5, 9780521474658.

[28] Shan Muthukrishnan. Data streams: Algorithms and applications. *Found. Trends Theor. Comput. Sci.*, 1(2):117–236, August 2005. ISSN 1551-305X. doi: 10.1561/0400000002. URL http://dx.doi.org/10.1561/0400000002.

[29] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming*, 13(4):277–298, 2005.