The Count-Min-Sketch and its Applications

Jannik Sundermeier

Abstract In this thesis, we want to reveal how to get rid of a huge amount of data which is at least difficult or even impossible to store in local memory. Our dataset consists of events connected to their frequencies. We are interested in several queries concerning our dataset. Since we cannot store the whole dataset, there is provably some approximation needed. The Count-Min Sketch is a data structure which allows us to store a sublinear approximation of our original set. Furthermore, the Count-Min Sketch is a very easy to implement data structure which offers good estimation guarantees. In this thesis, we want to explain the structure of the Count-Min Sketch and proof the guarantees for some basic query types.

1 Problem Setup

In this thesis, we will explain and analyze the Count-Min Sketch. Section 1.1 introduces the initial situation where we want to make use of the Count-Min Sketch. Section 1.2 deals with some variants of our general model and section 1.3 introduces several query types we want to get aware of using the Count-Min Sketch.

Jannik Sundermeier
Universität Paderborn, Warburger Str. 100, 33098 Paderborn, e-mail: janniksu@mail.upb.de

1.1 The Scenario

The information given in this thesis is mainly based on [1]. Whenever we make use of another source, you will find a quote.
The goal of the Count-Min Sketch is to handle a huge dataset which is at least difficult or even impossible to store in local memory. We can imagine this dataset as a set, consisting of events combined with their amount of occurrences. The setup is the data stream scenario. Hence, we have n different events and our set consists of tuples (i, a) with $i \in \{1, \cdots, n\}$ and an amount $a \in \mathbb{N}$. We can interpret this dataset as a vector $a = (a_1, \cdots, a_n)$. An entry a_i determines how often event i has happened in our distributed system. Initially all values of our vector are set to zero. Furthermore, our dataset obtains continuously updates. Updates are given as tuples (i, c). This means, that we have to look at the vector entry a_i and add the amount c to that entry. All other entries are not modified. Since we cannot store the whole dataset, we need some approximation of the information contained. A sketch data structure allows us to store an approximation of the original dataset sublinear in space. We take the definition for a sketch out of [2].

Definition 1. An f(n)-sketch data structure for a class Q of queries is a data structure for providing approximate answers to queries from Q that uses $O(f(n))$ space for a data set of size n, where $f(n) = n^{\epsilon}$ for some constant $\epsilon < 1$.

The set of queries Q will be introduced in section 1.3. All in all, we are looking for a sketch data structure which allows us to store a sublinear approximation of our original vector a. To solve this problem, we will introduce the Count-Min Sketch which is a data structure that solves the problem.

1.2 Update Semantics

The dataset we are observing receives permanently updates. According to the application field of our data structure, different update types might be plausible. In some cases it seems to be useful or even naturally, that the updates are strictly positive. Referring to that update semantics, we will call it the **cash-register case**.
If we want to allow negative updates, we will talk about the **turnstile case**. There are two important variations of the turnstile case to consider. The first one is called the **general case**. The general case has no further restrictions to the updates. The second case is called the **non-negative case**. Negative updates are allowed, but the application somehow ensures that the vector entries of the original vector cannot become negative.

1.3 Query types

We want to be able to compute different query types according to our vector a. Hereinafter we introduce the queries we are interested in.

1.3.1 Point Queries

The very basic point query is interested in approximating the value of a specific a_i. Given a value i, an approximated value for a_i is returned.

1.3.2 Range Queries

A range query computes the sum of a given range of values. Given the values l and r, the following sum is (approximately) computed: $\sum_{i=l}^{r} a_i$. Consider that there is a certain order on the indices of the original vector. For example, you want to analyze the regional allocation of visitors of your website. So each entry of your original vector represents the number of visitors of your website in a certain city. Assume, that the entries from 1000 to 5000 are from European cities. By range querying for the range [1000,5000] you can compute the amount of website visitors from Europe.

1.3.3 Inner Product Queries

An inner product query computes (an approximation) of the inner product of our vector a and a given vector b. An inner product of two vectors a and b is defined as follows: $a \odot b = \sum_{i=1}^{n} a_i b_i$.
Inner Product Queries might be reasonable in a database scenario. Computing the join size of two relations is a task query optimizers of database systems regularly have to solve. Related to our sketch, we assume without loss of generality, that the attribute values of the two joined relations a and b are integers in the range $[1, n]$. Now we can represent our relations with vectors a and b. A value a_i determines, how many tuples we have with i in the first relation. This is defined similar for vector b. Using sketches, we can estimate the join size of a and b in the presence of continuously updates, so that items can be removed or added to the relations at every time.

1.3.4 ϕ-quantiles

The ϕ-quantiles of our original vector are defined as follows: At first, we have to sort our vector entries by their frequencies. The ϕ-quantiles consist of all indices with rank $k\phi\|a\|_1$ for $k = 0, \cdots, \frac{1}{\phi}$. In order to approximate the ϕ-quantiles, we accept all indices i with $(k\phi - \epsilon)\|a\|_1 \leq a_i \leq (k\phi + \epsilon)$ for some specified $\epsilon < \phi$.

1.3.5 Heavy hitters

The ϕ-heavy hitters of our dataset are the events with highest frequencies. We can describe the set of ϕ-heavy hitters as follows: $H = \{a_i | i \in \{1, \cdots, n\} \text{ and } a_i \geq \phi\|a\|_1\}$. Because we cannot compute exact values, we try to approximate the ϕ-heavy hitters by accepting any i such that $a_i \geq (\phi - \epsilon)\|a\|_1$ for some $\epsilon < \phi$.
The problem of finding heavy hitters is very popular, because we can imagine various applications. For instance, we are able to detect the most popular products of a web-shop, for example Amazon. Besides, we are able to detect heavy TCP flows. Here, we can interpret a as the amount of packets passing a switch or a router. This might be useful for identifying denial-of-service attacks.

2 Preliminaries

In the following section, we will introduce the formal preliminaries used in later parts of this thesis. At first we need a general theorem about expectation. Sometimes we want to analyze the expectation of a sum of random variables. Linearity of expectation (section 2.1) helps us to compute this expectation. Additionally, we need two important important results of the probability theory,

namely Markov's inequality (section 2.2) and Chernoff bounds (section 2.3), to proof our results. Afterwards we will introduce pairwise-independent hash functions (section 2.4) which is a family of hash functions used here.

2.1 Linearity of Expectation

Linearity of expectation allows us to compute the expectation of a sum of random variables by computing the sum of the individual expectations.

Theorem 1. *Let $X_1, X_2, \cdots, X_n$ discrete random variables and $X = \sum_{i=1}^{n} X_i$. Then it holds*

$$E[X] = E\Big[\sum_{i=1}^{n} X_i\Big] = \sum_{i=1}^{n} E[X_i].$$

2.2 The Markov inequality

To estimate our probabilistic results, we need Markov's inequality.

Theorem 2. *For a non negative random variable X and for any $a > 0$, it holds:*

$$Pr[X \geq a] \leq \frac{E[X]}{a}.$$

2.3 Chernoff bound(s)

For one analysis, we will need the following Chernoff bound:

Theorem 3. *Let $X_1, \cdots X_n$ a sequence of independent Bernoulli experiments with success probability p. Let $X = \sum_{i=1}^{n} X_i$ with expectation $\mu = np$. Then, for every $\delta > 0$ it holds:*

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

2.4 Pairwise-Independent Hash Functions

For our sketch, we need the notion of pairwise independent hash functions. They are useful, because the probability of a hash collision is kept small and the mapping is independent from the hashed data.

Definition 2. Let $n, m \in \mathbb{N}, n \geq m$. A family of functions $H \subseteq \{h | h : [N] \to [M]\}$ is called **a family of pairwise independent hash functions**, if for all $i \neq j \in N$ and $k, l \in M$ $Pr[h(i) = k \wedge h(j) = l] = \frac{1}{M^2}$.

We remark here that another name for this class of hash functions is "strongly 2-universal". This depends on the following fact: Let h from a family of pairwise independent hash functions.

$$\text{For } i \neq k, Pr[h(i) = h(k)] \leq \frac{1}{M}$$

This inequality results directly from the definition given above. In the following we want to introduce a concrete family of pairwise independent hash functions, we can use for examples in later parts of this thesis.

Example 1 *The following exemplary construction is taken out of [4]. For further interest you can find a proof there, which proofs that the given construction indeed is a family of pairwise-independent hash functions.*
Let $U = \{0, 1, 2, \cdots, p^k - 1\}$ *and* $V = \{0, 1, 2, \cdots, p - 1\}$ *for some integer* k *and prime* p. *Now, we rewrite every element of the universe* U *with* $\bar{u} = (u_0, \cdots, u_{k-1})$ *such that* $\sum_{i=0}^{k} u_i p^i = u$. *Then, for every vector* $\bar{a} = (a_0, a_1, \cdots, a_{k-1})$ *with* $0 \leq a_i \leq p - 1, 0 \leq k \leq k - 1$ *and for any value* b *with* $0 \leq b \leq p - 1$, *let*

$$h_{\bar{a},b}(u) = \Big(\sum_{i=1}^{k} a_i u_i + b \Big) \mod p.$$

Consider the family

$$H = \{h_{\bar{a},b} | 0 \leq a_i, b \leq p - 1 \text{ for all } 0 \leq i \leq k - 1\}.$$

H is a family of pairwise-independent hash functions.

3 The Count-Min Sketch

In this section, we will introduce the Count-Min Sketch. First, we will explain a general idea of the Count-Min Sketch (section 3.1) and then we will define it formally (section 3.2). Afterwards, we will give an example of how it works (section 3.3).

3.1 Idea

The goal of the Count-Min Sketch is to store an approximation of the vector a sublinear in space. Therefore, it is not possible to store each entry of the original vector. A shorter vector of size $w < n$ ($a' = (a'_1, a'_2, ..a'_w)$) is needed. Using a hash function $h : \{1, ..., n\} \to \{1, ..., w\}$ enables us to map the indices of the original vector a to the new vector a'. This means, that an update (i,c) will not be executed by adding c to a_i, but by adding c to $a'_{h(i)}$.

Example 2 *We assume that we have a vector a of length 7. More precisely, we assume that we have an exemplary vector $a = (2, 0, 6, 3, 0, 1, 4)$. The used hash function is here $h_1 : \{0, ..., 9\} \to \{0, ..., 3\}, x \rightarrowtail x \mod 4$. The resulting vector a' would be: $a' = (2, 1, 10, 3)$. Since $h_1(3) = h_1(7) = 3$ we get an overestimate on the entries 1 and 7 (just as one example). With the hash-function h_1 the probability that an arbitrary x maps to 1 ($\frac{3}{10}$) is higher than the probability that an arbitrary x maps to 3 ($\frac{1}{5}$).*

To improve the previous seen results, we can use hash functions that provide certain guarantees, for example a function out of a family of pairwise independent hash functions as defined in section 2.4. Thus, the probability for an overestimate is smaller than before. But there are still possibilities to improve the results.
Instead of storing just one vector a' we can store store a certain amount of smaller vectors $a'_1, ..., a'_d$. Each vector a'_i gets an own randomly chosen hash function h_i out a family of pairwise independent

hash functions. On update, each vector is updated. To estimate the value of a_i, we compute the minimum of the estimates of every a'_i. Intuitively, the minimum of all estimates seems to be a good estimate, because at least hash collisions happened (if our updates are always positive). In the following section, we describe the concrete construction.

3.2 Construction

The construction is defined as follows. We need two parameters ϵ and δ. We want to guarantee, that the error of our estimate is within a factor of ϵ with probability $1 - \delta$. With the parameters ϵ and δ, we choose $d = \lceil ln(\frac{1}{\delta}) \rceil$ and $w = \lceil \frac{e}{\epsilon} \rceil$. The Count-Min Sketch is represented by a two-dimensional array $count[d][w]$. Initially, each entry of the sketch is set to zero. Additionally, we choose d hash functions randomly from a family of pairwise independent hash functions. Each hash functions is assigned to exactly one row of our sketch. On update (i, c) the procedure works as follows:

$$\text{For each } j \text{ set } count[j, h_j(i)] \text{ to } count[j, h_j(i)] + c.$$

Therefore, we can interpret each row of the sketch as its own approximation of the original vector a. For our queries, we can somehow compute the estimation of our sketch that seems to be closest to the original value. Figure 3.2 shows a visualization of the Count-Min Sketch.

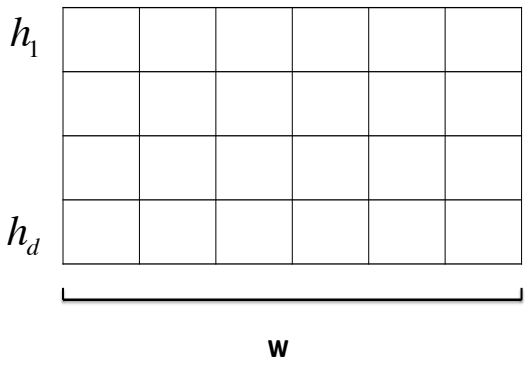

Fig. 1 A visualization of the Count-Min Sketch. $h_1, \cdots, h_d$ represent the chosen hash functions for the corresponding row. Each hash function maps an index of the original vector to an index in the row.

It seems to be remarkable, that the size of the sketch itself does not depend on the input size. Indeed, the size of the sketch is always constant in n. However, this is not the whole truth. To achieve reasonable results, you have to choose your parameters ϵ and δ dependent on your input size. For example $w > n$ does not make sense, because your sketch will be larger than your input size. Furthermore, we have to store the hash functions. Usually we can do this using $O(log(n))$ space. At the latest, when analyzing this circumstance, we see that there is a dependency on the input size. But basically, if you want to achieve better error guarantees, you will have to store larger vectors. If you want to increase the probability to stay in a certain bound, you will have to store more vectors.

3.3 Example

We want to illustrate the construction and the update procedure with an example. Set $n = 8, w = 3$ and $d = 3$. Now, we choose 3 hash functions from the family of pairwise independent hash functions described in Example 1. $h_1(u) = (1u_0 + 2u_1 + 1) \mod 3$, $h_2(u) = (2u_0 + u_1) \mod 3$ and $h_3(u) = (2u_0 + u_1 + 1) \mod 3$. Consider the update $(6, 2)$. At first, we have to compute the base 3 representation of 6. $6 = 0 * 3^0 + 2 * 3^1$. Thus, we rewrite 6 using the vector $(0, 2)$. Now, we can compute the update for each row:

- $h_1(6) = (1 * 0 + 2 * 2 + 1) \mod 3 = 2$
- $h_2(6) = (2 * 0 + 2 * 2 + 0) \mod 3 = 1$
- $h_3(6) = (2 * 0 + 1 * 2 + 1) \mod 3 = 0$

Consider a second update (5,7). $5 = 2 * 3^0 + 1 * 3$, so we rewrite 5 with the vector $(2, 1)$.

- $h_1(5) = (1 * 2 + 2 * 1 + 1) \mod 3 = 2$
- $h_2(5) = (2 * 2 + 1 * 1 + 0) \mod 3 = 2$
- $h_3(5) = (2 * 2 + 1 * 1 + 1) \mod 3 = 0$

The following figure 3.3 visualizes the sketch after executing both updates. Note that $h_1(6) = h_1(5) = 2$ and $h_3(6) = h_3(5) = 0$. Hence, we have an overestimation on the values for indices 5 and 6 concerning to hash functions h_1 and h_3. If we were interested in point querying for index 5 or 6, we would look at the estimation of every row. Exemplary, we will do this for index 5. In the first row, we see $count[1, h_1(5)] = 9$. The estimation of the second row is $count[2, h_2(5)] = 7$ and the estimation of the third row is $count[3, h_3(5)] = 9$. We should take the estimation of h_2, because no hash collision happened in that row. Indeed, computing the minimum of all estimations will be the procedure we use. In section 4.1 we will analyze the error guarantees we could achieve with that procedure.

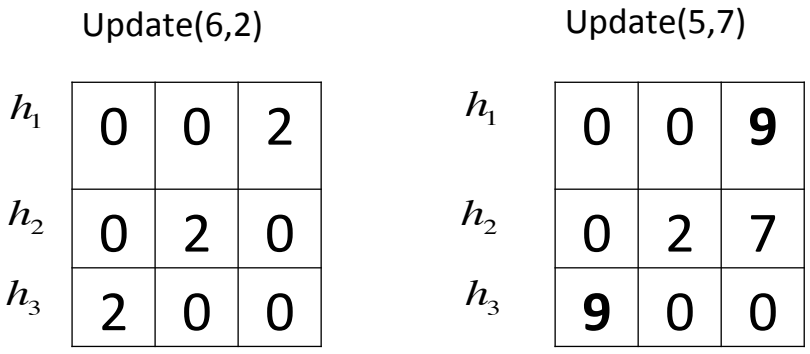

Fig. 2 This figure visualizes the resulting sketch of the example. The left part visualizes the sketch after the update (6,2). The right parts shows the resulting sketch after both updates (6,2) and (5,7)

4 Query Estimation

In this chapter, we want to analyze the estimations for our queries described in section 1.3. Section 4.1 deals with estimating a point query, section 4.2 is about the Inner Product Query and the topic of section 4.3 is Range Queries. Afterwards, we briefly want to give an overview on the estimations of ϕ-quantiles (section 4.4) and heavy hitters (section 4.5). In the whole chapter we make us of the L1-norm of a vector. The L1-norm of a vector $a = (a_1, a_2, \cdots, a_n)$ is defined as $\|a\|_1 = \sum_{i=1}^{n} a_i$.

4.1 Point Query

In the following, we will focus on point queries as defined in section 1.3. Therefore, we will analyze how to estimate a point query using a Count-Min Sketch and afterwards we will proof error guarantees we could achieve. We will start with an estimation for the non-negative case.

4.1.1 Non-Negative Case

By intuition, we have to find that estimation for a_i with the least amount of hash-collisions, because every hash-collision to the index i leads to an overestimation. More formally, we can compute our estimation ($\hat{a_i}$) for a_i as follows:

$$\hat{a_i} = min_j count[j, h_j(i)]$$

In other words, we look at the estimation for a_i in every row j of our sketch and subsequently compute the minimum of all estimations. The succeeding theorem is about the error guarantees which we can achieve using the described estimation for a point query.

Theorem 4. *The estimate $\hat{a_i}$ offers the following guarantees:*

1. $a_i \leq \hat{a_i}$
2. *With probability at least* $1 - \delta$: $\hat{a_i} \leq a_i + \epsilon\|a\|_1$.

Proof of Theorem 4: The idea of the proof is to analyze the expected error for one row of the sketch. Afterwards, we can use that knowledge to show that it is unlikely that the error in every row exceeds the introduced bound.

Proof of part 1:
$a_i \leq \hat{a_i}$ results directly from our update assumption. Since our updates cannot be negative, always the correct value a_i is contained in our estimation. The only thing can happen is, that we overestimate the correct value for a_i due to hash-collisions.

Proof of part 2:
We define $\widehat{a_{i,j}}$ as $count[j, h_j(i)]$ which is the estimation for a_i in row j. Because of our observations of part 1 we can say, that $count[j, h_j(i)] = a_i + X_{i,j}$. So it consist of the correct value a_i and an error term, which depends on the chosen hash function j and our estimation index i.
For analyzing the error term, we need an indicator variable $I_{i,j,k}$, that indicates if there is a hash collision of the index i and a different index k concerning the hash function j:

$$I_{i,j,k} = 1 \leftrightarrow h_j(i) = h_j(k) \text{ for } i \neq k \text{ and } I_{i,j,k} = 0 \text{ otherwise.}$$

Because we are interested in the expected error term for a row j, we are interested in the expectation of our indicator variable. As our indicator variable is a binary random variable, the expectation of $I_{i,j,k}$ is the probability that $I_{i,j,k} = 1$:

$$E[I_{i,j,k}] = Pr[h_j(i) = h_j(k)] \overset{(*)}{=} \frac{1}{range(h_j)} = \frac{1}{\frac{e}{\epsilon}} = \frac{\epsilon}{e}.$$

$(*)$ holds, because h_j is chosen from a family of pairwise independent hash functions, which is especially a family of universal hash functions. Using the introduced indicator variable, we are able to express our error term $X_{i,j}$ formally:

$$X_{i,j} = \sum_{k=1}^{n} I_{i,j,k} * a_k$$

Now, we want to analyze the error term.

$$E[X_{i,j}] = E[\sum_{k=1}^{n} I_{i,j,k} * a_k] \overset{(**)}{\leq} \sum_{k=1}^{n} a_k E[I_{i,j,k}] \overset{(***)}{=} \|a\|_1 \frac{\epsilon}{e}$$

By using linearity of expectation $(**)$, the definition of the L_1 norm $(***)$ and applying $E[I_{i,j,k}] \leq \frac{\epsilon}{e}$, we are able to show that the expected error term in a certain row j is less or equal to $\frac{\epsilon}{e}\|a\|_1$.
After analyzing the error term for one row of our sketch, we have to analyze the error for our whole estimation. We can apply Markov inequality to show that the probability that the estimation of every row is worse than $\epsilon\|a\|_1$ is less or equal to δ. As already mentioned, our estimation is worse than $\epsilon\|a\|_1$ if the error term of every row is worse than $\epsilon\|a\|_1$, because we compute the minimum of all possible estimations. It follows that

$$Pr[\hat{a_i} - a_i > \epsilon\|a\|_1] = Pr[\forall j : X_{i,j} > \epsilon\|a\|_1].$$

Since we have chosen our hash functions independently from each other, we can interpret each row (or each smaller vector) as its own random experiment. Therefore, we can simply multiply the probabilities of each row.

$$Pr[\forall j : X_{i,j} > \epsilon\|a\|_1] = \prod_{i=1}^{d} Pr[X_{i,j} > \epsilon\|a\|_1].$$

As a last step, we can apply Markov's inequality, because for the non-negative case, it holds $X_{i,j} \geq 0$.

$$\prod_{i=1}^{d} Pr[X_{i,j} > \epsilon\|a\|_1] \leq \prod_{i=1}^{d} Pr[X_{i,j} \geq \epsilon\|a\|_1] \overset{\text{Markov}}{\leq} \prod_{i=1}^{d} \frac{E[X_{i,j}]}{\epsilon\|a\|_1}.$$

Substituting the expectation of our error term by the term we already have calculated, we finally proof the theorem.

$$\prod_{i=1}^{d} \frac{E[X_{i,j}]}{\epsilon\|a\|_1} = \prod_{i=1}^{d} \frac{\frac{\epsilon}{e}\|a\|_1}{\epsilon\|a\|_1} = \prod_{i=1}^{d} \frac{1}{e} = e^{-d} = e^{-\lceil ln(\frac{1}{\delta})\rceil} \leq e^{-ln(\frac{1}{\delta})} = \delta.$$

Finally, we have shown that - in the non-negative case - the error of our estimation is less or equal to $\epsilon\|a\|_1$ with a probability of $1 - \delta$. $\square$

4.1.2 General Case

After analyzing the non-negative case, we are interested in possible negative updates, which is the analysis of the general turnstile case. For this case, our estimation works as follows:

$$\hat{a_i} = median_j count[j, h_j(i)]$$

For this estimation, we can show, that the following theorem is true:

Theorem 5. *With probability* $1 - \delta^{\frac{1}{4}}, a_i - 3\epsilon\|a\|_1 \leq \hat{a_i} \leq a_i + 3\epsilon\|a\|_1$

Proof of Theorem 5: Similar to the proof for the non-negative case, we analyze the expected error term of a certain row j. The only difference to the non-negative case is, that our error term $X_{i,j}$ could become negative. Thus, we can write our estimation for row j as $count[j, h_j(i)] = a_i + X_{i,j}$. In the following, we will analyze the absolute value of our error term $X_{i,j}$, because we can apply Markov's inequality then.

$$E[|X_{i,j}|] = E[|\sum_{k=1}^{n} I_{i,j,k} * a_k|] \leq \sum_{k=1}^{n} |a_k * E[I_{i,j,k}]| \leq \sum_{k=1}^{n} |a_k| * \tfrac{\epsilon}{e} = \|a\|_1 * \tfrac{\epsilon}{e}$$

Actually, we can use the same estimation for our error term as before. Applying Markov inequality again, we can show that with probability of at least $\frac{7}{8}$ our error term is within the bounds of Theorem 5.

$$Pr[|X_{i,j}| > 3\epsilon\|a\|_1] \overset{\text{Markov}}{\leq} \tfrac{E[|X_{i,j}|]}{3\epsilon\|a\|_1} \leq \tfrac{\frac{\epsilon}{e}\|a\|_1}{3\epsilon\|a\|_1} = \tfrac{1}{3e} \leq \tfrac{1}{8}$$

Consider the random experiment of taking a certain row j and look up, whether the error term of that row fits to our bounds or not. Due to the independent choice of our hash functions, we can interpret this random experiment as Bernoulli experiment with success probability $p = \frac{1}{8}$ (because we want to analyze, that it is unlikely that many estimations exceed our bounds). The expectation μ of our experiment is $\mu = d * p = \frac{d}{8}$. We need at least $\frac{d}{2}$ of such low probability events to assure, that the median does not fit to our bounds. With that knowledge, we can apply Chernoff bounds and finally proof the theorem. Denote $S_j = 1$ if the expected error of row j does not fit to our bounds and 0 otherwise.

$$Pr[\sum_{j=1}^{d} S_j > (1+3)\tfrac{d}{8}] \leq (\tfrac{e^4}{(1+3)^{(1+3)}})^{\frac{d}{8}} = (\tfrac{e^{\frac{3}{8}}}{2})^{ln(\frac{1}{\delta})} = \tfrac{(\frac{1}{\delta})^{\frac{3}{8}}}{(\frac{1}{\delta})^{ln(2)}} = \delta^{ln(2)-\frac{3}{8}} < \delta^{\frac{1}{4}}$$

Eventually, it holds

$$Pr[a_i - 3\epsilon\|a\|_1 \leq \hat{a_i} \leq a_i + 3\epsilon\|a\|_1] > 1 - \delta^{\frac{1}{4}} \quad \square$$

4.2 Inner Product Query

In this section, we will analyze how to estimate an inner product query $a \odot b$ for two vectors a and b. We simply consider the strict turnstile case. Four our estimation, we need to sketches $count_a$ and $count_b$. The first sketch represents vector a and the second one vector b. Both sketch systems have to share the same hash functions and the same parameters ϵ and δ to achieve reasonable estimations. The idea of the estimation is the same idea as we used for point queries. For each row of our sketch (or for each chosen hash function), we compute the inner product query using that specific hash function. Afterwards, we compute the minimum of all estimations:

$$\widehat{a \odot b} = min_j \sum_{k=1}^{w} count_a[j,k] * count_b[j,k]$$

The following theorem deals with the error guarantees we can achieve using the estimation procedure described above.

Theorem 6. *The estimation $\widehat{a \odot b}$ offers the following guarantees:*

1. $a \odot b \leq \widehat{a \odot b}$
2. *With probability* $1-\delta : \widehat{a \odot b} \leq a \odot b + \epsilon \|a\|_1 \|b\|_1$

Proof of Theorem 6: The idea of the proof is again similar to the proof of Theorem 3. First, we analyze the error term of a certain row j and afterwards show that it is unlikely that the error term of every row exceeds our bounds.

Proof of part 1:
The first fact again results directly from our update assumption. The correct inner product $a \odot b$ is always contained in our solution because our updates cannot be negative.

Proof of part 2:
Initially, we look at the estimation of a fixed row j. We determine

$$(\widehat{a \odot b})_j = \sum_{i=1}^{n} a_i b_i + \sum_{\substack{p \neq q \\ h_j(p)=h_j(q)}} a_p b_q = a \odot b + \sum_{\substack{p \neq q \\ h_j(p)=h_j(q)}} a_p b_q$$

The second part is again our error term we have to analyze further. In the following we describe it using a random variable X_j. For the further analysis we rewrite our error term, using the indicator variable $I_{i,j,k}$ we have already used for analyzing point queries.

$$X_j = \sum_{\substack{p \neq q \\ h_j(p)=h_j(q)}} a_p b_q = \sum_{(p,q)} I_{i,j,k} a_p b_q$$

Now, we can proceed analyzing the expectation of our error term, using the expectation of our indicator variable and applying linearity of expectation.

$$E[X_j] = E[\sum_{(p,q)} I_{i,j,k} a_p b_q] = \sum_{(p,q)} a_p b_q E[I_{i,j,k}] \leq \sum_{(p,q)} a_p b_q \tfrac{\epsilon}{e}$$

To complete our analysis for one row, we need to know how many pairs of indices exist. We can analyze this by computing the Cartesian product of the index set with itself. In order to see that this is linked to the product of the L_1-norms, we rewrite the Cartesian product with a doubled sum.

$$\sum_{(p,q)} a_p b_q \tfrac{\epsilon}{e} = \tfrac{\epsilon}{e} \sum_{p=1}^{n} \sum_{q=1}^{n} a_p b_q = \tfrac{\epsilon}{e} \sum_{p=1}^{n} a_p * \sum_{q=1}^{n} b_q = \tfrac{\epsilon}{e} \|a\|_1 \|b\|_1$$

With the expected error term of one row, we show that it is unlikely that the error term of every row is greater than $\epsilon \|a\|_1 \|b\|_1$.

$$Pr[\forall j : X_j > \epsilon \|a\|_1 \|b\|_1] = \prod_{j=1}^{d} Pr[X_j > \epsilon \|a\|_1 \|b\|_1] \overset{\text{Markov}}{\leq} \prod_{j=1}^{d} \frac{\frac{\epsilon}{e} \|a\|_1 \|b\|_1}{\epsilon \|a\|_1 \|b\|_1} = \prod_{j=1}^{d} \tfrac{1}{e} \leq \delta$$

Eventually, it holds:

$$Pr[(\widehat{a \odot b}) \leq \epsilon \|a\|_1 \|b\|_1] > 1 - \delta \quad \square$$

As mentioned in section 1.3, we can use an Inner Product Query, to estimate the join size of two relations. With the previous analysis, we can directly conclude time and space complexity of the concrete application.

Corollary 1. *The join size of two relations on a particular attribute can be approximated up to* $\epsilon\|a\|_1\|b\|_1$ *with probability* $1-\delta$*, by keeping space* $O((\frac{1}{\epsilon})ln(\frac{1}{\delta}))$*. The time used for the estimate also can be described with* $O((\frac{1}{\epsilon})ln(\frac{1}{\delta}))$.

This follows, because the space used by our sketch is $d * w$. Plugging in our values for d and w, we get $\lceil ln(\frac{1}{\delta})\rceil * \lceil\frac{e}{\epsilon}\rceil$. Therefore, the statement about space complexity is true. Because we have to look at every sketch entry while computing the estimation, the same statement holds for time complexity as well.

4.3 Range Query

The following sections deal with the approximation of a range query defined in section 1.3. We start with two naive approaches of estimating a range query (section 4.3.1) and afterwards introduce a better approach using dyadic ranges (section 4.3.2). For the whole chapter we consider the non-negative turnstile case.

4.3.1 Naive approaches

The most naive approach to estimate a range query for a range $[l, r]$ is simply point querying for every index contained in the original range and afterwards summing up the estimates. It is easy to see, that in this case the error term increases linearly with the size of the range. Indeed the effective factor would be $r - l + 1$.
Another approach is to compute a special inner product query. For this version, we have to define a second vector x of size n, with $x_i = 1$ if $l \leq i \leq r$ and 0 otherwise. Computing the inner product of the vectors a and x will exactly compute the range query for the range $[l, r]$. But again, the error term increases linearly with the size of the range and we have the additional costs for creating a sketch for the second vector and executing the updates.

4.3.2 Computation with dyadic intervals

The following section is based on the basic usage of dyadic ranges introduced in [3]. The concrete procedure is described in [1]. To assure that the error term does not increase linearly to the size of the range but logarithmically, we need the notion of dyadic ranges.

Definition 3 (Dyadic Ranges). For $x \in \mathbb{N}$ and $y \in \mathbb{N}_0$, a dyadic range is an interval of the form:

$$D_{x,y} := [(x-1)2^y + 1, x2^y]$$

For better understanding and for a proof, we illustrate the dyadic ranges that are contained in an original interval $[1, n]$ (n is a power of two), using a binary tree. Figure 4.3.2 shows a visualization of that tree.
The following lemma and the related corollary tells us, why the usage of dyadic ranges could help us to keep the error term sublinear to the size of the range.

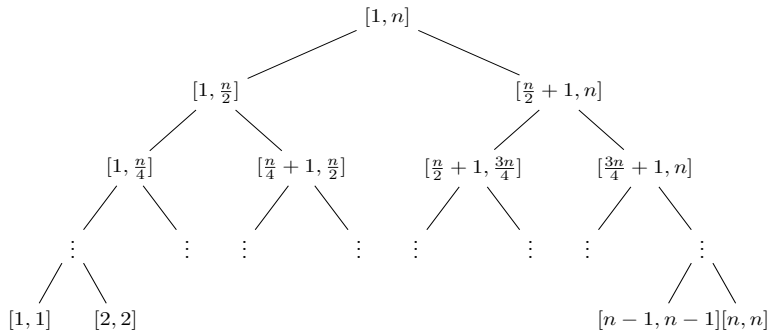

Fig. 3 This is a visualization of a dyadic tree. Each node of the tree represents a dyadic interval, which is part of the original interval $[1, n]$. The root represents the original interval. Each inner node has two children. The inner node is exactly halved by its children. Therefore, the left child of a node represents the first half of the interval and the right child the right half.

Lemma 1. *Let [l,r] = r a range. One every height of the tree of dyadic ranges, there are at most 2 dyadic ranges in a minimum cardinality dyadic cover C which covers the interval r.*

Since the tree of dyadic ranges is a balanced binary tree, we know that the height of that tree is $log(n) + 1$. This immediately implies the following corollary.

Corollary 2. *Each interval of size n can be covered by using at most $\lceil 2log(n) \rceil$ non-overlapping dyadic ranges.*

We want to use an example to clarify that this seems to be true and afterwards give a formal proof.

Example 3 *Figure 3 shows the dyadic tree of the interval $[1, 16]$. We want to cover the interval $i = [2, 12]$. The black marked nodes build a dyadic cover of the interval i. However, this cover is not a minimum cardinality dyadic cover of i. Note, that on height $log(n)+1$ three nodes are marked black. We are able to reduce the amount of black nodes on that height by replacing the nodes representing the intervals $[7, 7]$ and $[8, 8]$ by their parent node. After doing this, we could notice that on the height above are three nodes marked black. But again you can replace two black nodes sharing the same parent by replacing them with their parent which is marked in grey here. So, if we take the grey node to our solution and drop every node of the subtree below, we will have a minimum cardinality dyadic cover.*

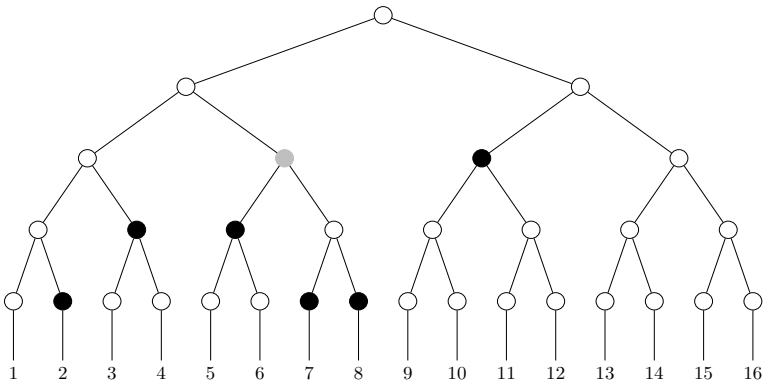

Fig. 4 This is a visualization of a dyadic tree which represents all dyadic intervals which are part of the original interval [1,16]. The black marked nodes build a dyadic cover of the interval $[2, 12]$.

We will proceed with the analysis of Lemma 1.
Proof of Lemma 1: Let C a set of non-overlapping dyadic ranges covering an interval $i = [l, r]$. Especially assume that C is not minimal and there is a height of the dyadic tree of i which contains at least 3 non-overlapping dyadic intervals of C. We start by looking at the lowest height h with at least 3 dyadic intervals of C. Let $left \in C$ (especially $left \subseteq [l, r]$) the interval with the lowest indices of h contained in C and analogous $right \in C$ (especially $right \subseteq [l, r]$) with the highest indices of h contained in C. Due to our assumption there has to be an interval $d \in C$ ($d \subseteq [l, r]$) of h whose indices are between the indices of $left$ and $right$. Let d_{sib} the sibling of d (this means, that they share the same parent node p).

Case 1: $d_{sib} = left$ or $d_{sib} = right$
In this case, we can replace d and d_{sib} directly by their parent p, because d_{sib} and d are both non overlapping dyadic ranges and $d, d_{sib} \subseteq [l, r]$. Denote further, that $d \cup d_{sib} = p$. Therefore, $p \subseteq [l, r]$. By replacing d and d_{sib} with p, we still cover the intervals d and d_{sib} but have reduced the cardinality of C.

Case 2: $d_{sib} \neq left$ and $d_{sib} \neq right$.
Without loss of generality, we assume that the indices of d_{sib} are smaller than the indices of d (the proof for larger indices works analogous). Determine that all indices of d_{sib} are larger than the indices of $left$ but smaller than the indices of d.

Case 2.1 $d_{sib} \in C$.
This sub-case is analogous to **Case 1**. Simply replace d and d_{sib} by their parent. The cardinality of C decreased but we still cover the original interval.

Case 2.2 $d_{sib} \notin C$
Due to our assumption the interval d_{sib} has to be covered by intervals of C. A further consequence is, that the interval d_{sib} has to be covered by children of d_{sib} because the intervals in C are non-overlapping and $d \in C$. Now replace all children c of d_{sib} with $c \in C$, d_{sib} itself and d with the parent of d and d_{sib} p. We still cover the original interval but we have reduced the cardinality of C by at least 1.

Since we have reduced the cardinality of C in both cases and we have reduced the amount of dyadic ranges on height h (which are part of C), this implies our Lemma. If we use the described

procedure to reduce the amount of dyadic ranges on height h until there are at most 2, we will look at height $h-1$ and execute the same procedure. If we do this for every height of our dyadic tree, we will have at most 2 dyadic ranges on each height of the tree. Since the height of our tree is $log(n)+1$, it follows that we can cover our original range $[l,r]$ by taking at most $\lceil 2log(n) \rceil$ dyadic ranges. □

Using the previously gained knowledge, we can explain the idea of estimating range queries using dyadic ranges. The core idea is to construct a data structure where we can answer range queries for dyadic intervals very easily. In that case we can answer our original range query by estimating the queries for the dyadic cover of the original interval and sum up the estimates. Now, we do not use only one sketch, but we use $log(n)+1$ sketches. Assume that n is a power of two (or increase n to the next power of two). Each sketch represents one height of the dyadic tree of the interval $[1,n]$. Denote the particular sketches with $count_0, \cdots count_{log(n)}$ with $count_i$ describing the sketch for height i. All sketches share the same hash functions and parameters ϵ and δ. On update (i,c) execute the following update procedure:

- At each height of the tree look at the interval $D_{x,y}$ with $i \in D_{x,y}$
- Use x as key for the hash functions and execute update (x,c) for count_y

Using this procedure, we can execute point queries for each dyadic range. The first step to answer a range query for an interval [l,r] is to compute a minimum cardinality dyadic cover C. Our estimation for the whole range query works as follows: Denote $a[l,r]$ as the exact result of a range query for an interval $[l,r]$ and $\widehat{a[l,r]}$ the estimation using the procedure above.

$$\widehat{a[l,r]} := \sum_{D_{x,y} \in C} \text{min}_j \text{count}_y[j, h_j(x)]$$

The subsequent theorem deals with the error guarantees we can achieve using the procedure described above.

Theorem 7. *The estimation $\widehat{a[l,r]}$ offers the following guarantees:*

1. $a[l,r] \leq \widehat{a[l,r]}$
2. *With probability at least* $1-\delta$: $\widehat{a[l,r]} \leq a[l,r] + 2\epsilon log(n)\|a\|_1$

Proof of Theorem 7: The ideas of this proof are still the same as for point and inner product queries. At the beginning, we look at the error term of a certain row j in our whole sketch system and analyze the expected error. Afterwards we use Markov's inequality to show that exceeding the bounds is very unlikely.

Proof of part 1:
We still consider the non-negative case. Hence, we cannot have negative updates and the point query of every dyadic interval always contains the exact solution plus possible error terms based on hash collisions. Since every point query for a dyadic interval cannot underestimate, the estimation for a dyadic cover cannot underestimate.

Proof of part 2:
Let us have a look at fixed row j of the sketch count_y querying for a dyadic range $D_{x,y}$. We can find the sum of all values a_i with $i \in D_{x,y}$ and a certain error term $X_{y,j,x}$ which depends on the chosen sketch y, the row j and the queried index x.

$$\text{count}_y[j, h_j(x)] = \sum_{i \in D_{x,y}} a_i + X_{y,j,x}$$

Using the indicator variable $I_{i,j,k}$ we have already used for point and inner product queries, we can rewrite our error term. The whole dyadic interval $D_{k,y}$ of every index k with $h_j(k) = h_j(k)$ is part of the error term if a hash collision occurs.

$$X_{y,j,x} = \sum_{k=1}^{n} I_{x,j,k} \sum_{i \in D_{k,y}} a_i$$

As a result, we are able to analyze the expected error using linearity of expectation and the expectation of our indicator variable.

$$\begin{aligned} E[X_{y,j,x}] &= E\Big[\sum_{k=1}^{n} I_{x,j,k} \sum_{i \in D_{k,y}} a_i\Big] = \sum_{k=1}^{n} I_{x,j,k} \sum_{i \in D_{k,y}} a_i \\ &\leq \sum_{k=1}^{n} \frac{\epsilon}{e} \sum_{i \in D_{k,y}} a_i \overset{(*)}{=} \frac{\epsilon}{e} \sum_{i=1}^{n} a_i = \frac{\epsilon}{e} \|a\|_1 \end{aligned}$$

(*) holds, because on every height of the dyadic tree, i is contained in exactly one dyadic interval $D_{x,y}$. For simplicity, we assume that we answer all queries using just one hash function j, because the following inequality holds:

$$\widehat{a[l,r]} := \sum_{D_{x,y} \in C} \min_j \text{count}_y[j, h_j(x)] \leq \min_j \sum_{D_{x,y} \in C} \text{count}_y[j, h_j(x)]$$

With this assumption we can bound the error term of our queries related to count_y using an additional random variable which represents the error term for the whole row j.

$$\begin{aligned} X_j &= \sum_{D_{x,y} \in C} X_{y,k,x} \\ E[X_j] &= E\Big[\sum_{D_{x,y} \in C} X_{y,k,x} = \Big] = \sum_{D_{x,y} \in C} E[X_{y,k,x}] \leq \sum_{D_{x,y} \in C} \frac{\epsilon}{e} \|a\|_1 \leq 2log(n) \frac{\epsilon}{e} \|a\|_1 \end{aligned}$$

Finally, we are able to analyze the overall error.

$$\begin{aligned} Pr[\min_j X_j > 2log(n)\epsilon\|a\|_1] &= \prod_{i=1}^{d} Pr[X_j > 2log(n)\epsilon\|a\|_1] \overset{\text{Markov}}{\leq} \prod_{i=1}^{d} \frac{E[X_j]}{2log(n)\epsilon\|a\|_1} \\ &\leq \prod_{i=1}^{d} \frac{2log(n)\frac{\epsilon}{e}\|a\|_1}{2log(n)\epsilon\|a\|_1} = \prod_{i=1}^{d} \frac{1}{e} \leq \delta \end{aligned}$$

Putting things together, we have finally shown:

$$Pr[\widehat{a[l,r]} > a[l,r] + 2log(n)\epsilon\|a\|_1] < 1 - \delta. \quad \square$$

To complete this chapter, we briefly want to analyze the space complexity of the introduced procedure. Since we need $log(n)+1$ different sketches we get $O(log(n)(\frac{1}{\epsilon})(ln(\frac{1}{\delta}))$ as a bound for the space complexity. If we further regard the space we need for our hash functions, we still get a bound of $O(log(n))$. All in all, the space complexity still stays sublinear in n which is exactly what we wanted to achieve.

4.3.3 Example

We consider the same basic setup as described in Example 3.3. $n = 8, d = 3$ and $w = 3$ and we use exactly the same hash functions h_1, h_2 and h_3. First, we build our basic sketch structure. Therefore we have to compute the dyadic tree of the interval $[1, 8]$. Figure 4.3.3 shows the set-up of our data structure.

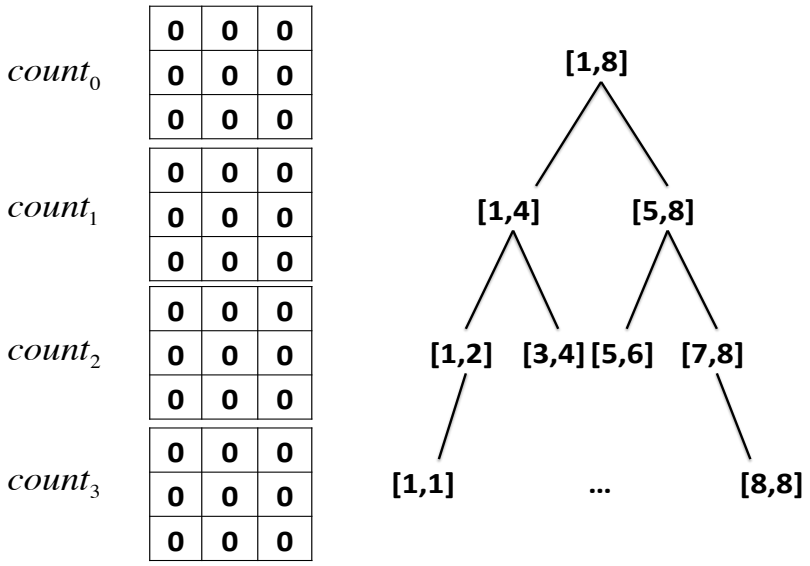

Fig. 5 The right part visualizes the dyadic tree for the interval $[1, 8]$. On the left side, we see the sketch structure for each height of the dyadic tree. This is the initial set-up, hence all values contained are set to 0.

Determine the update (4,3). $4 = 1 * 3^0 + 1 * 3^1$. Thus, we can rewrite 4 using the vector (1,1). For executing the update, we have to look at every row of our sketch.

- $count_0 : 4 \in [1, 8] = D_{1,3} \rightarrow$ execute update $(1, 3)$
- $count_1 : 4 \in [1, 4] = D_{1,2} \rightarrow$ execute update (1,2)
- $count_2 : 4 \in [3, 4] = D_{2,1} \rightarrow$ execute update (2,3)
- $count_3 : 4 \in [4, 4] = D_{4,0} \rightarrow$ execute update $(4, 3)$

Figure 4.3.3 illustrates the resulting sketch after executing the updates $(4, 3), (7, 4)$ and $(1, 3)$. We want to estimate a range query for the interval $[1, 5]$. Note, that the correct value $a[1, 5] = 3+3 = 6$. For our estimation we have to compute a minimum cardinality dyadic cover of $[1, 5]$ which is $C = \{[1, 4], [5, 5]\}$. At first, we execute a point query for the dyadic interval $[1, 4]$. $[1, 4] = D_{1,2}$ is part of the sketch $count_1$ so we have to point query for the index 1 using the sketch $count_1$. $\hat{a_1} = 6$. The second step is point querying for the dyadic interval $[5, 5]$. $[5, 5] = D_{5,0}$ is part of the sketch $count_3$ so we have to execute a point query for the index 5 using $count_3$. $\hat{a_5} = 3$. Summing up the results, we get $\widehat{a[1, 5]} = 6 + 3 = 9 > a[1, 5] = 6$. Due to hash collisions, the result of our estimation is larger than the real value.

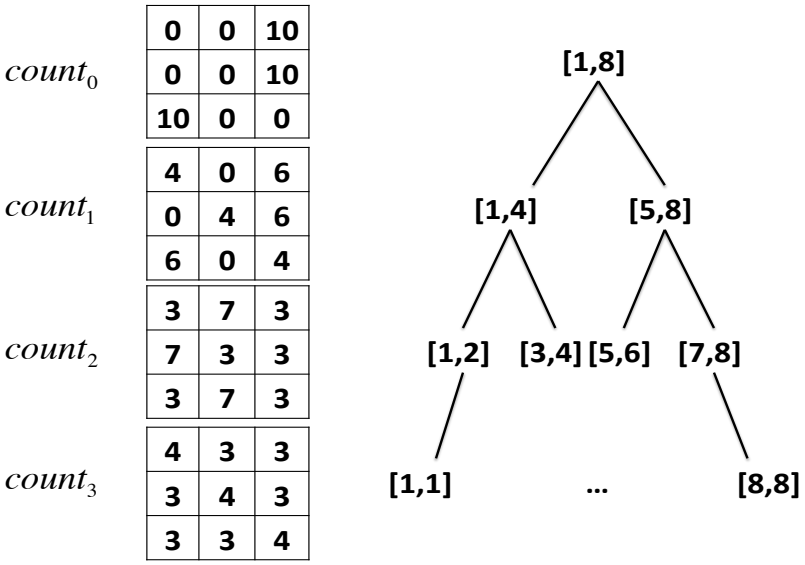

Fig. 6 The visualization of the data structure after executing the updates $(4,3),(7,4)$ and $(1,3)$.

4.4 ϕ-Quantiles

The authors of [1] describe two different approaches of computing the ϕ-quantiles. We will focus on the first approach and skip the approach which uses random subset sums, because both arrive at a similar runtime bound. We consider the non-negative turnstile case. Basically, computing ϕ-quantiles of our input stream can be reduced to a series of range queries. Consider the array $A = [a[1,1], a[1,2], \cdots, a[n,n]|$ containing all possible range queries with 1 as a lower bound. Executing a binary search on the array A, we can compute the smallest index r_k with $a[1,r_k] > k\phi a[1,n]$. Using the sketch system of section 4.3, we can reduce this to computing O(log(n)) point queries.
Using the described procedure, we can compute the ϵ-approximate ϕ-quantiles with probability at least $1-\delta$.

4.5 Heavy Hitters

We analyze the problem in both the cash-register case and the turnstile case. In the turnstile case, we assume the non-negative case. In this section, we omit the proofs of our guarantees. At least the ideas of the proofs can be looked up using the referenced literature.

4.5.1 Cash-Register Case

For estimating the heavy hitters of our original vector a, we need a second supporting data structure - a Min-Heap. Additionally, we have to keep track of the current value of $\|a\|_1$. On update (i,c), we have to increment the value of $\|a\|_1$ by c - additional to our basic update procedure. After updating, we execute a point query for index i. If the estimation $\hat{a_i}$ is above the threshold of $\phi\|a\|_1$ it is added to our heap. Periodically, we analyze whether to root of our heap (meaning the event with lowest count of our heap) is still above the threshold. If not, it is deleted from the heap. At the end of our input stream, we scan every element of our heap and all elements whose estimate is still above

$\phi\|a\|_1$ are output.
Analyzing the error guarantees, we could proof the following theorem:

Theorem 8. *Using the procedure described above, Count-Min Sketches offer the following guarantees:*

1. *Every index with count more than $\phi\|a\|_1$ is output*
2. *With probability $1-\delta$, no index with count less than $(\phi-\epsilon)\|a\|_1$ is output.*

4.5.2 Turnstile Case

If updates are allowed to be negative, we have to get aware of a two-sided error bound. In order to solve the problem, we make use of the setup we have introduced in section 4.3 for estimating Range Queries. Hence, we have one sketch for each height of the dyadic tree of our overall range. The update procedure stays the same. In order to find all heavy hitters, we execute a parallel binary search in our dyadic tree. All single items whose approximated frequency is still above the threshold are output. With a further analysis, we could show the following theorem:

Theorem 9. *Using the procedure described above, estimating the heavy hitters using Count-Min Sketches offer the following guarantees:*

1. *Every index with frequency at least $(\phi+\epsilon)\|a\|_1$ is output*
2. *With probability $1-\delta$ no index with frequency less than $\phi\|a\|_1$ is output.*

5 Summary / Conclusion

In this thesis we have introduced and analyzed the Count-Min Sketch extensive. Initially, we have seen that we can represent a set of events connected to their frequencies using a vector. Since we cannot store the whole vector, we need a technique to approximate the entries of the original data set. The basic idea is to map the original vector into a second vector of smaller size. Using hash functions from a family of pairwise-independent hash functions, we can map the indices from the original vector to the smaller vector and additionally, the probability of a hash collision is kept small. Storing not just one vector, but a certain amount of smaller vectors leads to the construction of the Count-Min Sketch.
Afterwards, we have analyzed the approximation techniques for several query types. For the estimation of point queries and inner product queries, we have seen that computing the minimum estimation of all rows j seems to be a good estimation (if updates cannot be negative). If negative updates are possible, we have proved that for point queries, computing the median offers good guarantees.
Furthermore, we have seen that estimating range queries with naive approaches cannot achieve good guarantees (or even better guarantees are possible). Therefore, we introduced a sketch structure using the notion of dyadic ranges. Our structure allows us to point query for dyadic ranges quite easily. Since we can cover each interval of size n using at most $2log(n)$ dyadic ranges, we could achieve an error term which does not increase linearly to the size of the range, but logarithmically.
Finally, we have roughly seen how to estimate heavy hitters and ϕ-quantiles, two applications which are very popular in the data stream scenario.

References

1. Cormode, Graham and S.Muthukrishnan "An improved data stream summary: the count-min sketch and its applications" Journal of Algorithms 55.1 (2005) 58-75.
2. Gibbons, Phillip B and Matias, Yossi. "Synopsis Data Structures for Massive Data Sets" In: " SODA '99 Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms", pp. 909-910.
3. Gilbert, Anna C., Kotidis, Yannis, S.Muthukrishnan and Strauss, Martin J.: "How to Summarize the Universe: Dynamic Maintenance of Quantiles" In: Proceeding VLDB '02 Proceedings of the 28th international conference on Very Large Data Bases, pp. 454-465.
4. Mitzenmacher, Michael, and Eli Upfal. Probability and computing: Randomized algorithms and probabilistic analysis. Cambridge University Press, 2005, pp. 324,325.