

# Streaming Algorithms \*

**Christian Sohler**

Institute of Computer Science I

University of Bonn

Germany

`sohler@informatik.uni-bonn.de`

\*This is a very preliminary version that does not yet contain references.



# Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	An Example: Determine the Number of Users of a Search Engine . . . . .	7
1.1.1	A Different View of Insertion-Only Data Streams . . . . .	8
1.2	More Streaming Models . . . . .	12
1.2.1	The Sliding Window Model . . . . .	12
1.2.2	Dynamic Data Streams . . . . .	14
1.3	Outline . . . . .	17
<b>2</b>	<b>Uniform Random Sampling</b>	<b>19</b>
2.1	Reservoir Sampling . . . . .	19
2.2	Tools for the Analysis of Randomized Algorithms . . . . .	21
2.2.1	Markov Inequality . . . . .	21
2.2.2	Chebyshev Inequality . . . . .	22
2.2.3	Chernoff Bounds . . . . .	23
2.3	The Heavy Hitter Problem . . . . .	25
2.3.1	Analysis Using Chebyshev inequality . . . . .	26
2.3.2	Analysis Using Chernoff Bounds . . . . .	28
2.4	Clustering Data Streams . . . . .	29
2.4.1	Overview of the Analysis . . . . .	30
2.4.2	Analysis of the Random Sampling Approach . . . . .	31
2.5	Sampling in Dynamic Data Streams . . . . .	32
2.5.1	The Data Structure . . . . .	32
2.5.2	The Algorithm . . . . .	33
2.5.3	Analysis . . . . .	34
<b>3</b>	<b>Hashing and Random Projections</b>	<b>35</b>
3.1	Ideal Hash Functions . . . . .	35
3.2	Count-Min Sketches . . . . .	36
3.2.1	The Data Structure . . . . .	37
3.2.2	Analysis . . . . .	37
3.3	The Johnson-Lindenstrauss Lemma and Second Frequency Moments . . . . .	38
3.3.1	The choice of the $r_i$ . . . . .	39
3.3.2	Second Frequency Moments . . . . .	41
3.4	Universal Hash Functions . . . . .	42
3.4.1	Constructing Universal Hash Functions . . . . .	42

3.5	Working with Universal Hash Functions . . . . .	43
3.5.1	Independence of Random Variables . . . . .	44
3.5.2	When Universal Hash Functions are not Sufficient. . . . .	45
3.6	An Improved Algorithm for the Distinct Elements Problem . . . . .	46
3.6.1	New Idea . . . . .	46
3.6.2	The Algorithm . . . . .	46
3.6.3	Analysis . . . . .	46
3.7	Classes of k-wise Independent Hash Functions . . . . .	49
3.7.1	Construction of a Class of k-wise Independent Hash Functions . . . . .	49
3.7.2	k-wise Independent Random Variables . . . . .	50
3.8	Estimating the Frequency Moments . . . . .	52
3.8.1	Estimating $F_2$ . . . . .	52
3.9	Higher Frequency Moments . . . . .	56
<b>4</b>	<b>The Merge and Reduce Principle</b>	<b>61</b>
4.0.1	Description of Operations . . . . .	61
4.0.2	An Example . . . . .	61
4.0.3	A Streaming Algorithm Using Merge and Reduce . . . . .	62
4.1	A Streaming Algorithm for k-Median Clustering via Coresets . . . . .	62
4.1.1	From Clusterings to Coresets . . . . .	63
4.1.2	A Coreset Construction . . . . .	64
4.2	Coresets for High Dimensional Point Sets . . . . .	65
<b>5</b>	<b>Streaming Algorithms via Embeddings into Tree Metrics</b>	<b>71</b>
5.1	Embeddings into Trees . . . . .	71
5.1.1	Construction of a 2-HST for P . . . . .	72
5.2	Minimum Spanning Tree Cost . . . . .	73
5.2.1	Approximation of $cost(T)$ in Dynamic Data Streams . . . . .	74
5.3	Cost of Minimum Weighted Matching . . . . .	74
5.4	Facility Location Cost . . . . .	77
5.4.1	Reduction to HSTs . . . . .	77
5.4.2	Approximating the Cost of Facility Location in HSTs . . . . .	78
5.4.3	Analysis . . . . .	80
<b>6</b>	<b>Density Sampling</b>	<b>83</b>
6.1	Density sampling . . . . .	84
6.1.1	A Coreset for the 1-Median Problem . . . . .	84
6.1.2	Developing the Streaming Algorithm . . . . .	86
6.1.3	Coreset Construction . . . . .	88
6.1.4	Overview of the Analysis . . . . .	88
6.1.5	Analysis . . . . .	88
6.1.6	1-Median in Dynamic Data Streams . . . . .	91

<b>7</b>	<b>Lower Bounds</b>	<b>93</b>
7.1	Distinct Elements cannot be solved Deterministically . . . . .	93
7.1.1	There is no Deterministic $c^*$ -Approximation Algorithm for Distinct Elements . . . . .	94
7.2	Lower Bounds for Randomized Algorithms . . . . .	96
7.2.1	Lower Bounds for Randomized Computation of the Second Frequency Moment . . . . .	97



# 1 Introduction

Maybe the most significant change in computer systems in the last 20 years was caused by the interconnection of computers, in particular, via the Internet. This new technology allows us now to communicate with other people all around the world using, for example, email, VoiceOverIP, or ICQ. We can also access all kind of information via the World Wide Web. However, these new possibilities also raise new questions. How can we search in collections of documents as large as the World Wide Web? How can we monitor internet traffic to quickly find out about malicious behaviour like spreading worms or viruses? Search engine providers offer a wide variety of tools to search in the web. These tools are already very effective and the building ground of a multi-billion dollar industry. However, there are also many unsolved problems. For example, if we are searching for words that have multiple meanings like 'Jaguar', then we are getting as a result a mix of web pages about the car and the animal. Searching for images and movies is only possible, if users have provided the right textual descriptions. Searching the web for local information as, for example, 'pubs in bonn' is not yet very well supported. It may be an interesting service to allow queries for English (Chinese, Russian,...) webpages using German keywords. Ideally, the search engine could provide automatic translations of the content. In summary, there are still many ways to improve the current technology.

From the point of view of an algorithms person the above setting raises the general question how we can deal with data sets too large to be stored completely (like network traffic or the World Wide Web). Often these huge data sets occur in the form of data streams, for example, streams of packets at a router or streams of queries at a search engine. In this course we will develop algorithmic methods to deal with these kind of problems.

## 1.1 An Example: Determine the Number of Users of a Search Engine

Imagine we run a query engine and we are interested in the number of users that use it. How can we obtain an estimate for this number? Obviously, there is no direct way of counting different users, because people do not have unique IDs like a user name or so. However, it is easily possible to find out the IP address of the computer from which the query is submitted. IP addresses can change and different users may use the same computer, but still the number of different IP addresses gives us a good estimate of the number of different users of our query engine.

Our first step will be the development of a model of computation for this scenario. We will assume that our search engine is run on a single computer. Queries arrive sequentially and at high rate. With each query there is an IP associated and IPs come from a fixed universe. Thus, we can model our scenario as follows. Our input consists of a sequence  $\sigma = (\sigma_1, \dots, \sigma_n)$  of

items that come from a fixed universe  $\mathcal{U}$ , i.e. the items correspond to IP addresses. We can view the stream as a sequence of insertions of items  $\sigma_1, \dots, \sigma_n$  in some set  $Z$ . For this reason, we also call such a data stream an *insertion-only stream*. In our case, we are interested in the *number of distinct items in this sequence*. In a standard model of computation we can solve this problem easily by first sorting the sequence and then counting the number of distinct items with one linear scan over the data. However, sorting requires random access to the data. This is only possible if the data is stored in main or secondary memory. But in our case this is hardly possible. A search engine must handle so many queries that we cannot even store one IP for each query.<sup>1</sup> Thus, we need a different approach. We will trade the space used by our algorithm against accuracy, i.e. we will try to *approximate* the number of distinct items rather than computing it exactly. Thus our algorithm is supposed to maintain a small summary of the data already seen and this small summary should easily fit into the main memory of a standard PC, because we might want to maintain different statistics at the same time, which altogether must be kept in main memory. Thus the interesting question is what approximation can we guarantee under our space restriction or, equivalently, how much space is required, if we want to guarantee a certain approximation quality (the latter formulation is what we typically look at in the theory community).

Besides space efficiency, we also have to ensure that we can quickly process each data item. Otherwise, our algorithm will significantly slow down the search engine, which is something that must be avoided by all means. We can summarize that we are looking for an algorithm that

- (a) processes data items in order of arrival,
- (b) uses small space, and
- (c) has fast per-item processing time.

Depending on the application different values for space and per-item-processing time may be tolerable. For example,  $O(\sqrt{n})$  space will suffice for many applications, but ideally we only require  $\log^{O(1)} n$  bits of memory. Thus, we will typically require that a streaming algorithm uses a polylogarithmic amount of memory in the length of the stream. Most of the time we will measure memory requirements in bits. In some situations however, it will be more convenient to talk about memory cells and assume that memory cells contain real numbers. In a similar way to space complexity, we require that the per-item processing time is  $\log^{O(1)} n$ . If an algorithm requires more space or processing time we discuss this in detail.

### 1.1.1 A Different View of Insertion-Only Data Streams

We now want to introduce a different view on insertion-only data streams. This view will be helpful in many situations and show analogies to other areas such as dimensionality reduction via the Johnson-Lindenstrauss Lemma.

Our items come from a fixed universe  $\mathcal{U}$  and we can view our stream as insertions of items from  $\mathcal{U}$  into a multiset  $Z$ . For simplicity we will assume that  $\mathcal{U} = \{1, \dots, |\mathcal{U}|\}$ . We can

---

<sup>1</sup>It should be noted that this is an idealized version of reality. Indeed, search engine providers maintain query logs that contain the IP of every query. However, these logs are huge and cannot be accessed quickly, so that sorting such a log by IP addresses is not really an option.

describe the current status of set  $Z$  as a  $|\mathcal{U}|$ -dimensional *frequency vector*  $f = (f_1, \dots, f_{|\mathcal{U}|})$  that contains for each item in  $\mathcal{U}$  its multiplicity in  $Z$ . For example, when we consider the stream  $a, a, d, c, c, a$  with items from the set  $\mathcal{U} = \{a, b, c, d\}$  we can view the current status of the stream as a 4-dimensional vector  $f = (3, 0, 2, 1)$ , where the first entry corresponds to the number of occurrences of item  $a$ , the second entry to the number of occurrences of  $b$ , and so on. Having this view of the *current status of a stream* the arrival of an item translates into an increment of one of the entries in this vector. In our problem we are interested in the support of the set  $Z$  or, equivalently, the number of non-zero entries of  $v$ . We denote this number sometimes as

$$F_0 = \sum_{1 \leq i \leq |\mathcal{U}|} f_i^0,$$

using the convention the  $0^0 = 1$  and call it the *0-th-frequency moment* of the stream. Obviously, the 0-th frequency moment is the number of distinct items in the stream. The  $k$ -th frequency moment is then defined similarly as

$$F_k = \sum_{1 \leq i \leq |\mathcal{U}|} f_i^k.$$

The 1-st frequency moment is simply the number of items in the stream  $n$ . The frequency moments of a stream belong to the best studied problems in the area of data streaming. Right now, we will focus on the 0-th frequency moment, i.e. the number of distinct elements in the stream. However, later in this course we will also learn how to approximate higher order frequency moments. Obviously, it is easy to maintain the 1-st frequency moment using a single counter with  $O(\log n)$  bits of memory.

Our new view of things suggest that we should work with the set  $\mathcal{U}$  rather than individual items. This turns out to be quite helpful. For simplicity, let us first assume that the items (not their frequencies) in the stream are chosen uniformly at random from  $\mathcal{U}$ . In other words, we are given a frequency vector and for each entry we choose the corresponding element uniformly at random from  $\mathcal{U}$ . The sequence itself is given by an adversary. How can we learn about the number of distinct items in this case? We have that the non-zero entries are distributed uniformly in our frequency vector  $f$ . We know that  $f$  is  $|\mathcal{U}|$ -dimensional and has  $F_0$  non-zero entries. Thus we can expect that the smallest non-zero dimension in  $f$  is around  $|\mathcal{U}|/F_0$  (we will quantify this more precisely later). Thus, if we remember the index  $i$  of the smallest non-zero entry in  $f$  we can output  $\tilde{F}_0 \approx |\mathcal{U}|/i$  as an estimator for the number of distinct elements.

The remainder of this section is devoted to making the above idea more precise. We start by calculating the expected minimum of  $n$  numbers chosen uniformly at random from the interval  $[0, 1]$ . We first determine the distribution of the minimum value of the  $n$  random points. To do this, we recall that a continuous distribution is described by a *probability density function*  $f$  over its domain (in our case the interval  $[0, 1]$ ). We also recall that for a random value  $X$  chosen from  $[0, 1]$  according to our probability density function  $f$  we have

$$\Pr[X \leq k] = \int_0^k f(x) \, dx.$$

We are choosing points uniformly at random from  $[0, 1]$ , so our probability density function is the constant 1-function, i.e.  $f(x) = 1$  for all  $x \in [0, 1]$ .

We would now like to determine the probability density function of the minimum value of  $n$  points chosen uniformly at random from  $[0, 1]$ . Let  $g$  denote the probability density function of this distribution. What is the value of  $g(x)$ ? We know that  $x$  is the minimum value among the  $n$  points, iff there is a point at  $x$  and all  $n - 1$  remaining points have a at least  $x$ . Since any of the  $n$  points can attain the minimum value at  $x$  we obtain

$$g(x) = n \cdot f(x) \cdot \left( \int_x^1 f(x) \, dx \right)^{n-1} = n \cdot (1-x)^{n-1} .$$

The expected value of  $X$  is then given by

$$\begin{aligned} \mathbf{E}[X] &= \int_0^1 x \cdot g(x) \, dx \\ &= \int_0^1 x \cdot n \cdot (1-x)^{n-1} \, dx \\ &= \left[ -\frac{(1-x)^n \cdot (nx+1)}{n+1} \right]_0^1 \\ &= \frac{1}{n+1} . \end{aligned}$$

Our next step is to show that  $X$  is concentrated around its expected value, i.e. with probability at least  $7/10$  we have  $1/(4n) < X < 4/n$ . We first calculate the probability that  $X$  is at least  $4/n$ .

$$\begin{aligned} \mathbf{Pr}[X \geq 4/n] &= \int_{4/n}^1 n \cdot (1-x)^{n-1} \, dx \\ &= \left[ -(1-x)^n \right]_{4/n}^1 \\ &= \left(1 - \frac{4}{n}\right)^n \\ &\leq 1/e^4 \\ &\leq 1/20 \end{aligned}$$

Then we need the probability for the event  $X \leq 1/(4n)$ .

$$\begin{aligned} \mathbf{Pr}[X \leq 1/(4n)] &= \int_0^{1/(4n)} n \cdot (1-x)^{n-1} \, dx \\ &= \left[ -(1-x)^n \right]_0^{1/(4n)} \\ &= 1 - \left(1 - \frac{1}{4n}\right)^n \\ &\leq 1 - 3/4 \\ &= 1/4 \end{aligned}$$

Since  $1 - (1/20 + 1/4) \geq 7/10$  it follows

**Lemma 1.1.1** *Let  $Y_1, \dots, Y_n$  be  $n$  random numbers drawn independently and uniformly at random from  $[0, 1]$ . Let  $X = \min_{1 \leq i \leq n} Y_i$ . Then  $E[X] = 1/(n + 1)$  and*

$$\Pr[1/(4n) \leq X \leq 4/n] \geq 7/10 .$$

Now we want to implement the above observation as follows. Our algorithm selects a random function  $h : \mathcal{U} \rightarrow [0, 1]$ , such that every element from  $\mathcal{U}$  is mapped to a location chosen independently and uniformly at random from  $[0, 1]$ . The algorithm maintains the minimum value  $m$  of the  $h(\sigma_i)$  seen so far. Upon arrival of an item  $\sigma_i \in \mathcal{U}$  in the data stream  $h(\sigma_i)$  is computed and compared to the current value of  $m$ . If  $h(\sigma_i)$  is smaller than  $m$  we set  $m$  to the value  $h(\sigma_i)$ . Otherwise, we keep our value of  $m$ . In the end we output  $1/m$  as an estimate for the number of distinct items. We summarize the algorithm below.

DISTINCTELEMENTS( $\sigma$ )

1. Choose function  $h$  such that every element from  $\mathcal{U}$  is mapped to a location chosen independently and uniformly at random from  $[0, 1]$
2.  $m = \infty$
3. **for each** item  $\sigma_i$  in the input stream **do**
4.     **if**  $h(\sigma_i) < m$  **then**  $m = h(\sigma_i)$
5.     **output**  $\tilde{F}_0 = 1/m$

From our discussion above it follows that with probability at least  $7/10$  we have  $F_0/4 \leq \tilde{F}_0 \leq 4 \cdot F_0$ . Hence, our algorithm computes with probability  $7/10$  a 16-approximation of the number of distinct elements in the stream. The algorithm processes each item in constant time. Ignoring the space required to store  $h$  it uses  $O(1)$  memory cells.

**Theorem 1** *Algorithm DISTINCTELEMENTS computes a 16-approximation of the number of distinct elements in a data stream. It processes each item in  $O(1)$  time and uses  $O(1)$  space plus the space required to store the function  $h$ .*

**Discussion.** We have developed a first data streaming algorithm that already contains many of the ingredients that are used in this area. First of all, the algorithm is randomized. Randomization plays a crucial role in most data streaming results. However, also a surprisingly large number of deterministic data streaming algorithms exist. One of the main difficulties in data streaming lies in a problem that we have hidden in the choice of the function  $h$ . The first problem is that we map into the continuous space  $[0, 1]$ . We can easily deal with this problem by discretizing the space. The bigger problem lies in the assumption that  $h$  maps each element from  $|\mathcal{U}|$  uniformly and independently to a number in  $[0, 1]$  (or in our discretization of this interval). To store such a function  $h$  we would need  $|\mathcal{U}|$  memory cells. Since  $|\mathcal{U}|$  is typically larger than  $n$ , we need more space than necessary to store the whole input stream! Even if we replace  $[0, 1]$  by some discrete space  $[0, \dots, M]$  and we choose our function  $h$  such that it maps each element from  $\mathcal{U}$  independently and uniformly to  $[0, \dots, M]$  we require  $\Omega(|\mathcal{U}| \log M)$  bits. This follows from the fact that we can choose any total function from  $\mathcal{U}$  to  $[0, \dots, M]$  and there are  $M^{|\mathcal{U}|}$  many of these functions and so to address them we need at least  $\log(M^{|\mathcal{U}|}) = \Omega(|\mathcal{U}| \log M)$

bits. One of the main technical difficulties in data streaming is to show that one can choose  $h$  from a smaller set of functions such that  $h$  can be represented using only a small number of bits. However, there are general results saying that in most cases this is only a technical obstacle and it is enough to consider functions with ideal properties like  $h$  has. In the first part of this course we will therefore ignore these issues and assume that we have access to these ideal functions.

## 1.2 More Streaming Models

Besides the insertion-only model, there are a number of other models for data streaming that model different application scenarios. In this course we will consider two more models: The *sliding window model* and *dynamic data streams*.

### 1.2.1 The Sliding Window Model

In the last section we have developed an algorithm to approximate the number of users of a search engine. The streaming algorithm maintains an approximation of the overall number of users over the whole stream. In some situations, however, we are only interested in the number of users in a certain time-window, for example, during the last 10 days. Such queries cannot be answered with our previous algorithm. Since this type of question occurs in many scenarios, we will formulate a streaming model for it. We will assume that items occur at roughly the same rate in our stream, such that computing a statistic over a certain time window is (essentially) equivalent to computing a statistic over the last  $N$  items seen in the stream. Thus, we are given access to an input stream of items and we are, for example, interested in the number of distinct items among the last  $N$  item seen. This model is called the *sliding window model*.

Formally, in the sliding window model the input is a stream  $\sigma = (\sigma_1, \dots, \sigma_n)$  of items from a universe  $U$ . We are interested in maintaining statistics of the last  $N$  items  $\sigma_{n-N+1}, \dots, \sigma_n$ . Similarly to the insertion-only model we have sequential access to the stream and, typically,  $\log^{O(1)} n$  space.

**A Basic Counting Problem.** We consider the following basic problem. We are given a binary stream, i.e.  $U = \{0, 1\}$  and we would like to approximate the number of ones in the sliding window of the stream. While this problem can be trivially solved for insertion-only streams with a single counter that uses  $O(\log n)$  bits, the problem in the sliding window model is that when the sliding window moves we do not know whether a zero or one leaves the window. It can be shown that one needs  $\Omega(n)$  space to count the number of ones in the window *exactly*. The motivation to study the basic counting problem is that it provides a very basic tool and that the algorithmic technique of the algorithm we develop below is used quite frequently for algorithms in the sliding window model. Besides, the algorithm may be used to count the number of certain events (for example, packet losses) in a certain period of time.

**Notation.** We call an element  $\sigma_i$  of the data stream *active*, if it is in the sliding window, i.e. within the last  $N$  elements seen. The *time stamp* of an element is its position from the right

among all elements that appeared to so far in the stream, i.e. the current element has time stamp one.

**Idea behind the Algorithm.** Our algorithm maintains a histogram for the active ones in the stream. A histogram is a collection of non-intersecting buckets that cover the sliding window. For each bucket we maintain the number of ones inside it and the time stamp of the rightmost element within the bucket. If the time stamp of a bucket is at least  $N + 1$  then all its ones are outside of the sliding window. Thus we do not need it any more and delete it.

Let  $C_i$ ,  $1 \leq i \leq m$  denote the number of ones in the  $i$ -th bucket sorted by time stamp. We will also call  $C_i$  to be the *size of bucket*  $i$ . We define  $\tilde{N}_1 = \frac{C_m}{2} + \sum_{1 \leq i \leq m} C_i$  to be our estimation for the number of ones inside the sliding window. What is the error of this estimator? The only bucket that may contain ones that are not inside the sliding window is bucket  $m$ . Since our buckets cover the sliding window and since bucket  $m$  is the only bucket that may contain ones outside of the window, the additive error of our estimator is at most  $\pm C_m/2$ .

**A  $(1 + \epsilon)$ -Approximation Algorithm.** We now would like to design a subdivision into buckets that guarantees that our additive error translates into a multiplicative approximation guarantee of  $(1 \pm \epsilon)$ .

We know that the number of elements inside the sliding window is at least

$$1 + \sum_{i=1}^{m-1} C_i ,$$

because the buckets 1 to  $m - 1$  are completely contained in the sliding window and bucket  $m$  has at least a single one within the window. Therefore, the algorithm makes a relative error of at most

$$\frac{C_m}{2 \cdot (1 + \sum_{i=1}^{m-1} C_i)} .$$

We want to select the bucket sizes in such a way that the relative error is at most  $\epsilon$ , i.e.

$$\frac{C_m}{2 \cdot (1 + \sum_{i=1}^{m-1} C_i)} \leq \epsilon .$$

Since it may happen that the current stream is followed by a stream consisting only of zeros, we have to ensure that the above property is satisfied for every bucket  $j$  in the stream that contains more than a single one (if a bucket contains only a single one, this one is contained inside the sliding window until the bucket expires and so the bucket does not induce any error), i.e.

$$\frac{C_j}{2 \cdot (1 + \sum_{i=1}^{j-1} C_i)} \leq \epsilon . \tag{1.1}$$

To achieve this, we maintain the following invariant.

(i)  $C_1 \leq C_2 \leq \dots \leq C_m$

(ii)  $C_j \in \{1, 2, 4, \dots, 2^{m'}\}$  for some  $m' = O(\log N)$

(iii) Every bucket size except for the size of the last bucket occurs either  $\lceil 1/\epsilon \rceil$  or  $\lceil 1/\epsilon \rceil + 1$  times.

Let  $C_j = 2^r$  denote the size of the  $j$ -th bucket. If  $r = 0$  there is nothing to prove and so we can assume  $r \geq 1$ . Our invariant implies that there are at least  $\lceil 1/\epsilon \rceil$  buckets of sizes  $1, 2, 4, \dots, 2^{r-1}$  with index smaller than  $j$ . This implies that

$$\begin{aligned} 2 \cdot \left(1 + \sum_{i=1}^{j-1} C_i\right) &\geq 2 \cdot \left(1 + \lceil 1/\epsilon \rceil (2^r - 1)\right) \\ &\geq \frac{2^{r+1} - 2}{\epsilon} \\ &\geq 2^r / \epsilon \end{aligned}$$

since  $r \geq 1$ . This implies inequality 1.1 since  $C_j = 2^r$ .

We next analyze the maximum number of buckets created by the algorithm. Clearly, the maximum bucket size is  $N$ . Therefore, we can have at most  $\log N$  different bucket sizes. For each bucket size we can have at most  $\lceil 1/\epsilon \rceil + 1$  buckets. Therefore, the overall number of buckets is at most  $O(\log N/\epsilon)$ . For each bucket we store its size and its time stamp. Both values are at most  $N$  and so  $O(\log N)$  bits suffice to represent them. Overall, our algorithm requires  $O(\log^2 N/\epsilon)$  space.

We summarize our algorithm below.

#### 0-1-COUNTING

1. Update the time stamps of the buckets
2. if the time stamp of the last bucket exceeds  $N$ , delete this bucket.
3. If the new element is a 0, ignore it. Otherwise, create a new bucket of size 1 and with time stamp 1.
4. Traverse the list of buckets in increasing order of time stamps. If there are  $\lceil 1/\epsilon \rceil + 2$  buckets of the same size, then join the two oldest (with respect to time stamp) to create a new bucket of the next larger size. This may lead to a cascade of such mergers.

An update operation takes  $O(\log N/\epsilon)$  time since we have to update the time stamp of every bucket. Similarly, an estimate for the number of ones can be computed in  $O(\log N/\epsilon)$  time.

**Theorem 2** *Algorithm 0-1-COUNTING computes a  $(1+\epsilon)$ -approximation for the 0-1-Counting Problem. It uses  $O(\log^2 N/\epsilon)$  bits of memory and performs updates and answers queries in  $O(\log N/\epsilon)$  time.*

### 1.2.2 Dynamic Data Streams

A third data streaming scenario that occurs in some situations is that of dynamic data streams. A dynamic data stream consists of INSERT and DELETE operations of items from a universe  $U = \{1, \dots, |U|\}$  into the current (multi)set of items  $Z$ . Each item may occur with multiplicity upto  $M$ . We will assume that dynamic data streams are *consistent*, i.e. no item is inserted more than  $M$  times and no item is deleted when no copy of it is present in  $Z$ . Dynamic data

streams can be used to model most kind of update operations. For example, our universe could be a discretization of the plane and items correspond to moving objects. We can update the position of an object by first deleting it at its old position and then reinserting the object at its new position. In a similar way, we can model updates in large database systems or financial transactions.

Similar to the insertion-only model and the sliding window model, we are looking for algorithms that process the stream (sequence of insert and delete operations) sequentially and that uses space polylogarithmic in the size of the universe and  $M$ .

Like for the sliding window model, we will also consider a basic problem that has no direct applications but occurs in important other problems as a subroutine. We would like to find a data structure that can be used to output the current set  $Z$ , if  $Z \leq k$  for some constant  $k$  known in advance. If  $Z > k$  the algorithm outputs this as well. We call this problem the  $k$ -set problem. We restrict our attention to the case  $M = 1$ , i.e. every item occurs at most one time in  $Z$ . The difficulty is that we may have a stream that first inserts many items into  $Z$  and then deletes most of them. Our goal is to find out, which items remain.

Let us consider the case  $k = 1$ , which is easy. We can simply count the number of items currently in  $Z$  by maintaining a single counter  $C$ , which is increased with every insert operation and decreased with every delete operation. To obtain the element it is sufficient to maintain the linear sum  $L$  of all inserted elements. Thus we can summarize the algorithm as follows.

INSERT( $q$ )

1.  $C = C + 1$
2.  $L = L + q$

DELETE( $Q$ )

1.  $C = C - 1$
2.  $L = L - q$

REPORT

1. **if**  $C = 0$  **output** <<'No item in current set.'; **return**
2. **if**  $C > 1$  **output** <<'More than one item in current set.'; **return**
3. **output** <<'Current set is ' <<  $L$

What can we learn from this simple example? First of all, we observe that in this case insert and delete are inverse operations of eachother. This will be the case for every streaming algorithm for dynamic data streams considered in this paper. In fact, to the best of our knowledge every data streaming algorithm for dynamic data streams has this property and it seems to be reasonable to conjecture that for every natural problem that can be solved in this model there is an algorithm that uses insert and delete operations that are inverse to eachother. The advantage of using operations that are inverse to eachother is that one can ignore the difficult case sketched in the beginning of this section, i.e. we do not care whether our stream consists of many insertions followed by many deletions or whether it is an insertion-only stream of the

remaining items. This will be very convenient for the analysis, because it will suffice to deal with the simpler insertion-only case.

Now we turn our attention towards the more general problem of finding  $k$  items. Our algorithm will use the same trick as for the  $F_0$ -approximation algorithm for the insertion-only model. We will view the current set  $Z$  as a characteristic vector  $f$  whose entries are from  $\{1, \dots, M\}$ , i.e. in our case  $f$  is a binary vector. Similar as in the case  $k = 1$  we can keep track of the number  $F_0$  of non-zero entries in  $f$ . Thus, if  $F_0 > k$  we can simply output this and so we just have to focus on the case  $F_0 \leq k$ . We cannot immediately use our previous attempt because in this case our linear sum consists of  $k$  distinct values and we do not know how it decomposes. So we are using the following trick. We subdivide  $U$  randomly into  $\ell = \Theta(k^2)$  sets  $U_1, \dots, U_\ell$ . This makes it quite unlikely that any of these sets contains 2 or more items. Thus, if we apply our approach on the sets  $U_i$  we obtain a streaming algorithm. The only problem is how to store the subdivision of  $U$ . If we store it explicitly, we require  $|U| \log k$  bits of memory. Therefore, we will again use a random function  $h$  that maps  $U$  to  $\{1, \dots, \ell\}$  and we define our sets  $U_i = h^{-1}(i)$ , i.e.  $U_i = \{u \in U : h(u) = i\}$ . We remark that storing  $h$  will also require  $|U| \log k$  bits, but as we will see later in Chapter 3 this space can be significantly reduced.

Now assume that at a fixed point of time we have  $k$  items in our current set  $Z$ . What is the probability that  $h$  maps two of them to the same set  $U_i$ ? Let us fix  $Z$  and two items  $x, y \in Z$ . Clearly,

$$\Pr[h(x) = h(y)] = 1/\ell .$$

Now, the union bound implies

$$\Pr[\exists x, y \in Z : h(x) = h(y)] \leq \sum_{x, y \in Z, x \neq y} \Pr[h(x) = h(y)] \leq \binom{k}{2} \cdot \frac{1}{\ell} \leq \frac{k^2}{2 \cdot \ell} .$$

If we choose  $\ell \geq 5 \cdot k^2$  we get with probability at least  $9/10$  that all elements from the current  $Z$  map to different locations. Therefore, if we apply our algorithm for one element to each of the sets  $U_i$  we obtain a solution for the  $k$ -set problem (with  $M = 1$ ). The algorithm maintains the function  $h$  and  $\ell$  counters  $C_1, \dots, C_\ell$ . and another  $\ell$  counter  $L_1, \dots, L_\ell$ .

INSERT( $q$ )

1.  $C_{h(q)} = C_{h(q)} + 1$
2.  $L_{h(q)} = L_{h(q)} + q$

DELETE( $Q$ )

1.  $C_{h(q)} = C_{h(q)} - 1$
2.  $L_{h(q)} = L_{h(q)} - q$

REPORT

1. **if**  $\exists i : C_i > 1$  **then output**  $\ll$  'Algorithm fails.'; **return**
2. **else**
3.     **output**  $\ll$  'Current set is ';
3.     **for**  $i = 1$  **to**  $\ell$  **do**
3.         **if**  $C_i = 1$  **then output**  $\ll$   $L_i$

Since each element of the universe is at most inserted once into our current set  $Z$ , we know that  $C_i \leq |U|$  and  $L_i \leq |U|^2$ . It follows that each counter requires  $O(\log |U|)$  bits of memory.

**Theorem 3** *There is a streaming algorithm for the  $k$ -set problem that at any point of time with probability at least  $9/10$  reports the current set  $Z$ , if  $|Z| \leq k$ . The algorithm uses  $O(k^2 \cdot \log |U|)$  bits of memory plus the space required to store  $h$ .*

In contrast to the algorithm for  $F_0$ -approximation in insertion-only data streams, the algorithm above recognizes when it makes an error and outputs a failure message.

## 1.3 Outline

So far, we have introduced the three different streaming models that will be used in this course: Insertion-only streams, the sliding window model, and dynamic data streams. Our next step is to develop the basic techniques that are used in this area. The first such technique is uniform random sampling from the elements of the stream and will be discussed in Chapter 2. We will use uniform sampling in the context of the heavy hitter problem, i.e. the problem of finding elements that occur very often in the data stream. In this context we will also introduce some analytical tools like Markov inequality, Chebyshev inequality and Chernoff-Hoeffding bounds. Then we will apply these techniques to the analysis of random sampling in the context of clustering. This will provide us with a first algorithm to cluster data streams.

In Chapter 3 we discuss another important approach to data streaming, namely the use of hash functions and random projections. We will again consider insertion-only streams as operations on a  $|U|$ -dimensional vector  $f$  and map this vector to some other space using a function  $h$ . For example, when we want to approximate the second frequency moment  $F_2$  we map  $f$  randomly to a low dimensional space. The Johnson-Lindenstrauss Lemma tells us that the second frequency moment of this projection can be used as an approximation for  $F_2$ .

In Chapter 4 we consider an important paradigm that can be used in many situations to develop deterministic data streaming algorithms. If we can find an algorithm that computes a small space representation of an input set such that (i) the union of two such representations is again a small space representation and (ii) the we can apply our algorithm to this union again to reduce the size of the representation, then the merge and reduce paradigm essentially gives us immediately a streaming algorithm for this problem.



## 2 Uniform Random Sampling

In this chapter we will consider the first important technique in the area of data streaming: Uniform Random Sampling. The idea is very simple. If we cannot store a data set, we simply pick a small subset of the data uniformly at random and analyze this subset. The main questions that we will address is how to pick a uniform random sample from a data stream and for which problems we give provable guarantees using the random sampling approach.

### 2.1 Reservoir Sampling

The first problem we consider is how to maintain a random sample in the insertion-only model. We will first develop a method known as *reservoir sampling* that can be used to maintain a random sample  $S \subseteq Z$  of size  $s$  of the current set  $Z$ . This method is used for sampling without repetition. If we would like to take a random sample of size  $s'$  with repetition, then we can start  $s'$  parallel instances of the above algorithm with  $s = 1$ .

The basic idea of reservoir sampling is simple. We start taking the first  $s$  vertices of the stream as our sample set. Clearly, at this point of time the only sample of  $s$  vertices from  $s$  elements is to take all of them. When the next item  $\sigma_i$  arrives we calculate the probability that this element is in the sample set. This probability for this event is obviously  $s/i$ . If the new element is in the sample set, it must replace an old one. To preserve the uniform distribution, we simply replace an element chosen uniformly at random from the sample.

RESERVOIRINSERT( $\sigma_i, s$ )

S: Current sample set

s: sample size

1. **if**  $i < s$  then  $S = S \cup \sigma_i$
2. **else**
3.     Choose a number  $x$  uniformly at random from  $[0, 1]$
4.     **if**  $x \leq s/i$  **then**
5.         Choose  $y \in S$  uniformly at random
6.          $S = S \setminus \{y\} \cup \{\sigma_i\}$

To formally prove the correctness, we have to show that for any fixed set  $R$  of  $s$  elements from  $\sigma_1, \dots, \sigma_n$  we have  $\Pr[S = R] = 1/\binom{n}{s}$ . We first prove this for the case  $s = 1$ . In this case we have  $R = \{\sigma_i\}$  for some  $1 \leq i \leq n$ . We have  $R = \{\sigma_i\}$  after all elements have been processed, iff  $\sigma_i$  has been chosen as a sample element when it is processed and no other element that appears later in the stream has also been selected. The probability that  $\sigma_i$  is selected is  $1/i$ .

The probability that the  $j$ -th element (for  $j > i$ ) is not selected is  $j - 1/j$ . Hence,

$$\Pr[S = R] = \frac{1}{i} \cdot \prod_{j=i+1}^n \frac{j-1}{j} = \frac{1}{i} \cdot \frac{i}{n} = \frac{1}{n} .$$

Now let us consider the general case. The proof is by induction over the length of the stream. The base case is a stream of length  $n = s$ . In this case, we have  $S = R$  with probability  $1 = s/n$ . Now assume that after  $n - 1$  elements for every fixed set  $R$  we have

$$\Pr[S = R] = \frac{1}{\binom{n-1}{s}} = \frac{s! \cdot (n-1-s)!}{(n-1)!} .$$

Now we want to prove that after processing  $\sigma_n$  we have for each set  $R \subseteq \{\sigma_1, \dots, \sigma_n\}$  of cardinality  $s$  a probability of  $\frac{s! \cdot (n-s)!}{n!}$ . We distinguish between the cases (a)  $\sigma_n \notin R$  and (b)  $\sigma_n \in R$ .

**Case (a).** Let us use  $X$  to denote the event that  $S = R$  after item  $\sigma_{n-1}$  has been processed. By the above discussion we have  $\Pr[X] = \frac{s! \cdot (n-1-s)!}{(n-1)!}$ . Let us use  $Y$  to denote the event that after processing of  $\sigma_n$  we have  $S = R$ . Clearly,  $Y$  happens only if  $X$  happens and item  $\sigma_n$  is not taking into the sample. Therefore, we have

$$\Pr[Y] = \Pr[X] \cdot \Pr[Y|X] = \frac{s! \cdot (n-1-s)!}{(n-1)!} \cdot \Pr[Y|X] .$$

Next we observe that  $\Pr[Y|X] = 1 - s/n$ , since  $Y$  happens conditioned on  $X$ , iff  $\sigma_n$  is not taken into the sample set. Therefore, we have

$$\Pr[Y] = \frac{s! \cdot (n-1-s)!}{(n-1)!} \cdot \frac{n-s}{n} = \frac{s! \cdot (n-s)!}{n!} = \frac{1}{\binom{n}{s}} .$$

**Case (b).** Let us call  $X$  the event that  $R \cap \{\sigma_1, \dots, \sigma_{n-1}\} \subseteq S$  after processing item  $\sigma_{n-1}$  and let us call  $Y$  the event that  $S = R$  after processing item  $\sigma_n$ . Again we observe that  $Y$  only happens, if  $X$  happens and so we obtain

$$\Pr[Y] = \Pr[X] \cdot \Pr[Y|X] .$$

To determine  $\Pr[X]$  we observe that in this case  $s - 1$  elements of  $S$  are fixed after  $\sigma_{n-1}$  has been processed. Thus only one element of  $S$  is not fixed and there are  $n - 1 - (s - 1)$  choices for this element. By induction hypothesis, each of these choices has a probability of  $1/\binom{n-1}{s-1}$  and the corresponding events are disjoint. Thus, we obtain

$$\Pr[X] = \frac{n-s}{\binom{n-1}{s}} = \frac{s! \cdot (n-1-s)! \cdot (n-s)}{(n-1)!} = \frac{s! \cdot (n-s)!}{(n-1)!} .$$

To determine  $\Pr[X|Y]$  we observe that this event happens, iff  $\sigma_n$  is chosen to be in  $S$  and the only element of  $S$  that is not in  $R$  is evicted from  $S$ . Therefore,

$$\Pr[X|Y] = \frac{s}{n} \cdot \frac{1}{s} = \frac{1}{n} .$$

Finally, we obtain

$$\Pr[Y] = \frac{s! \cdot (n-s)!}{(n-1)!} \cdot \frac{1}{n} = \frac{1}{\binom{n}{s}} .$$

We summarize our findings in the following theorem.

**Theorem 4** *We can compute a sample of  $s$  elements taken uniformly at random with or without repetition from  $\{\sigma_1, \dots, \sigma_n\}$  in the insertion-only data streaming model. Both algorithms (sampling with and with repetition) use  $O(s)$  memory cells. A single element is processed in  $O(1)$  time for samples without repetition and in  $O(s)$  time for samples with repetition. Returning a reference on the sample set requires  $O(1)$  time.*

**Proof :** The space requirements and running time follow immediately from the algorithms' descriptions.  $\square$

## 2.2 Tools for the Analysis of Randomized Algorithms

In this section we discuss a few basic inequalities for the analysis of randomized algorithms: Markov inequality, Chebyshev inequality, and Chernoff/Hoeffding bounds. We will apply Chebyshev inequality and Chernoff/Hoeffding bounds to analyze the quality of our streaming algorithm for the relaxed heavy hitter problem.

### 2.2.1 Markov Inequality

One of the most basic inequalities in probability theory is Markov inequality. Essentially, Markov inequality states that if the average of  $n$  non-negative numbers is  $\mu$ , then there can be no more than  $n/k$  numbers that are bigger than  $k \cdot \mu$ .

**Theorem 5 (Markov inequality)** *Let  $X$  be a non-negative random variable. Then we have for  $k > 0$ :*

$$\Pr[X \geq k] \leq \mathbf{E}[X]/k .$$

**Proof :**

$$\begin{aligned} \Pr[X \geq k] &= \sum_{x \geq k} \Pr[X = x] = \frac{1}{k} \cdot \sum_{x \geq k} k \cdot \Pr[X = x] \\ &\leq \frac{1}{k} \cdot \sum_{x \geq k} x \cdot \Pr[X = x] \leq \frac{1}{k} \cdot \sum_{x \geq 0} x \cdot \Pr[X = x] = \frac{\mathbf{E}[X]}{k} \end{aligned}$$

$\square$

Setting  $k = \ell \cdot \mathbf{E}[X]$  we obtain

**Corollary 2.2.1** *Let  $X$  be a non-negative random variable. Then we have for  $k > 0$ :*

$$\Pr[X \geq \ell \cdot \mathbf{E}[X]] \leq 1/\ell .$$

A typical application of the above corollary in the context of randomized algorithms is as follows. Suppose we have an algorithm with expected running time  $O(n) \leq cn$  for some constant  $c$ . Since the running time of an algorithm is a non-negative random variable, we can immediately apply Corollary 2.2.1 and get that with probability at least  $9/10$  the running time of our algorithm is at most  $10cn$ .

## 2.2.2 Chebyshev Inequality

While Markov inequality only depends on the expectation of a random variable, we can get better bounds, if we take the variance of the random variable into account. This is often done using Chebyshev inequality, which is proved using Markov inequality.

**Theorem 6 (Chebyshev-Ungleichung)** *Let  $X$  be a random variable. Then we have for  $k > 0$*

$$\Pr[|X - \mathbf{E}[X]| \geq k] \leq \frac{V[X]}{k^2} .$$

**Proof :**

$$\Pr[|X - \mathbf{E}[X]| \geq k] = \Pr[(X - \mathbf{E}[X])^2 \geq k^2]$$

From Markov inequality it follows that:

$$\leq \frac{\mathbf{E}[(X - \mathbf{E}[X])^2]}{k^2} = \frac{V[X]}{k^2} .$$

□

A typical application of Chebyshev inequality is as follows. Suppose we take a sample of  $s$  from a stream of  $n$  non-negative numbers  $\{\sigma_1, \dots, \sigma_n\}$  independently and uniformly at random. Let  $M = \max_{1 \leq i \leq n} \sigma_i$ . Let  $X_i$  denote the  $i$ -th random number taken. Then we have

$$\mathbf{Var}[X_i] = \mathbf{E}[X_i^2] - \mathbf{E}[X_i]^2 \leq \mathbf{E}[X_i^2] \leq M^2 .$$

Since the  $X_i$  are pairwise independent we get

$$\mathbf{Var}\left[\sum_{1 \leq i \leq s} X_i\right] \leq s \cdot M^2 .$$

Then Chebyshev inequality gives us

$$\Pr\left[\left|\sum_i X_i - \mathbf{E}\left[\sum_i X_i\right]\right| \geq \epsilon \cdot s \cdot M\right] \leq \frac{s \cdot M^2}{\epsilon^2 \cdot s^2 M^2} = \frac{1}{\epsilon^2 \cdot s} .$$

Therefore, if we choose  $s \geq \frac{1}{\delta \cdot \epsilon^2}$  we obtain with probability  $1 - \delta$  that our sum deviates from its expectation by at most  $\epsilon s M$ . Thus, the average value of our sum deviates at most  $\epsilon M$  from its expectation.

### 2.2.3 Chernoff Bounds

**Theorem 7 (Chernoff Ungleichung)** Let  $X_1, \dots, X_n$  be independent 0 – 1-random variables and let  $0 < \epsilon < 1$ . Then we have

$$\Pr\left[\sum_{i=1}^n X_i \geq (1 + \epsilon) \cdot \mathbf{E}\left[\sum_{i=1}^n X_i\right]\right] \leq e^{-\frac{1}{3}\epsilon^2 \cdot \mathbf{E}\left[\sum_{i=1}^n X_i\right]} . \quad (2.1)$$

und

$$\Pr\left[\sum_{i=1}^n X_i \leq (1 - \epsilon) \cdot \mathbf{E}\left[\sum_{i=1}^n X_i\right]\right] \leq e^{-\frac{1}{2}\epsilon^2 \cdot \mathbf{E}\left[\sum_{i=1}^n X_i\right]} \quad (2.2)$$

**Proof :** We first prove Inequality 2.1. Let  $h > 0$  be a value that is optimized later.

$$\Pr\left[\sum_{i=1}^n X_i \geq (1 + \epsilon) \cdot \mathbf{E}\left[\sum_{i=1}^n X_i\right]\right] = \Pr\left[h \cdot \sum_{i=1}^n X_i \geq h \cdot (1 + \epsilon) \cdot \mathbf{E}\left[\sum_{i=1}^n X_i\right]\right] .$$

By monotonicity of the e-function we obtain

$$\Pr\left[h \cdot \sum_{i=1}^n X_i \geq (1 + \epsilon) \cdot h \cdot \mathbf{E}\left[\sum_{i=1}^n X_i\right]\right] = \Pr\left[e^{h \cdot \sum_{i=1}^n X_i} \geq e^{h \cdot (1 + \epsilon) \cdot \mathbf{E}\left[\sum_{i=1}^n X_i\right]}\right] .$$

Now we can apply Markov inequality and get

$$\Pr\left[e^{h \cdot \sum_{i=1}^n X_i} \geq e^{h \cdot (1 + \epsilon) \cdot \mathbf{E}\left[\sum_{i=1}^n X_i\right]}\right] \leq \frac{\mathbf{E}\left[e^{h \cdot \sum_{i=1}^n X_i}\right]}{e^{h \cdot (1 + \epsilon) \cdot \mathbf{E}\left[\sum_{i=1}^n X_i\right]}} \quad (2.3)$$

Before we continue with the proof, we have to determine  $\mathbf{E}\left[e^{h \cdot \sum_{i=1}^n X_i}\right]$ . By independence of the  $X_i$  we have

$$\begin{aligned} \mathbf{E}\left[e^{h \cdot \sum_{i=1}^n X_i}\right] &= \mathbf{E}\left[\prod_{i=1}^n e^{h \cdot X_i}\right] \\ &= \prod_{i=1}^n \mathbf{E}\left[e^{h \cdot X_i}\right] \end{aligned}$$

Now let  $p_i$  denote the expected value of  $X_i$ , which is equivalent to  $\Pr[X_i = 1]$  since the  $X_i$  are 0 – 1–random variables.

$$\begin{aligned} \prod_{i=1}^n \mathbf{E}\left[e^{h \cdot X_i}\right] &= \prod_{i=1}^n (p_i \cdot e^h + (1 - p_i)) \\ &= \prod_{i=1}^n (1 + p_i(e^h - 1)) \\ &\leq \prod_{i=1}^n e^{p_i(e^h - 1)} \\ &= e^{\sum_{i=1}^n p_i \cdot (e^h - 1)} \\ &= e^{\sum_{i=1}^n \mathbf{E}[X_i] \cdot (e^h - 1)} \\ &= e^{\mathbf{E}\left[\sum_{i=1}^n X_i\right] \cdot (e^h - 1)} , \end{aligned}$$

where the inequality follows from  $1 + x \leq e^x$  for all  $x$ . Together with 2.3 we get

$$\begin{aligned} \Pr\left[\sum_{i=1}^n X_i \geq (1 + \epsilon) \cdot \mathbf{E}\left[\sum_{i=1}^n X_i\right]\right] &\leq e^{-h \cdot (1+\epsilon) \mathbf{E}[\sum_{i=1}^n X_i]} \cdot e^{\mathbf{E}[\sum_{i=1}^n X_i](e^h - 1)} \\ &= e^{-(1+h \cdot (1+\epsilon) - e^h) \cdot \mathbf{E}[\sum_{i=1}^n X_i]} \end{aligned}$$

Now we set  $h = \ln(1 + \epsilon)$  as this value minimizes the right-hand side of the above inequality. We get

$$\begin{aligned} \Pr\left[\sum_{i=1}^n X_i \geq (1 + \epsilon) \cdot \mathbf{E}\left[\sum_{i=1}^n X_i\right]\right] &\leq e^{-(1 + \ln(1+\epsilon)) \cdot (1+\epsilon) - e^{\ln(1+\epsilon)} \cdot \mathbf{E}[\sum_{i=1}^n X_i]} \\ &= e^{(\epsilon - (1+\epsilon) \cdot \ln(1+\epsilon)) \cdot \mathbf{E}[\sum_{i=1}^n X_i]} \end{aligned}$$

We use the Taylor series expansion of  $\ln(1 + \epsilon)$ , which is  $\ln(1 + \epsilon) = \sum_{i \geq 1} (-1)^{i+1} \frac{\epsilon^i}{i}$ . It follows

$$\begin{aligned} (1 + \epsilon) \cdot \ln(1 + \epsilon) &= (1 + \epsilon) \cdot \sum_{i \geq 1} (-1)^{i+1} \frac{\epsilon^i}{i} \\ &= \sum_{i \geq 1} (-1)^{i+1} \cdot \frac{\epsilon^i}{i} + \epsilon \cdot \sum_{i \geq 1} (-1)^{i+1} \cdot \frac{\epsilon^i}{i} \\ &= \epsilon - \sum_{i \geq 2} (-1)^i \cdot \frac{\epsilon^i}{i} + \sum_{i \geq 1} (-1)^{i+1} \cdot \frac{\epsilon^{i+1}}{i} \\ &= \epsilon - \sum_{i \geq 2} (-1)^i \cdot \frac{\epsilon^i}{i} + \sum_{i \geq 2} (-1)^i \cdot \frac{\epsilon^i}{i-1} \\ &= \epsilon + \sum_{i \geq 2} (-1)^i \epsilon^i \left( \frac{1}{i-1} - \frac{1}{i} \right) \end{aligned}$$

For  $0 < \epsilon < 1$  we obtain by ignoring higher order terms

$$(1 + \epsilon) \cdot \ln(1 + \epsilon) > \epsilon + \frac{\epsilon^2}{2} - \frac{\epsilon^3}{6} \geq \epsilon + \frac{\epsilon^2}{3} .$$

Finally, we obtain

$$\begin{aligned} \Pr\left[\sum_{i=1}^n X_i \geq (1 + \epsilon) \cdot \mathbf{E}\left[\sum_{i=1}^n X_i\right]\right] &\leq e^{(\epsilon - (1+\epsilon) \cdot \ln(1+\epsilon)) \cdot \mathbf{E}[\sum_{i=1}^n X_i]} \\ &\leq e^{(\epsilon - (\epsilon + \epsilon^2/3)) \cdot \mathbf{E}[\sum_{i=1}^n X_i]} \\ &= e^{-\frac{\epsilon^2 \cdot \mathbf{E}[\sum_{i=1}^n X_i]}{3}} \end{aligned}$$

This shows Inequality 2.1. To prove Inequality 2.2 we proceed similarly.

Again, let  $h > 0$  be a value that is optimized later.

$$\begin{aligned}
\Pr\left[\sum_{i=1}^n X_i \leq (1 - \epsilon) \cdot \mathbf{E}\left[\sum_{i=1}^n X_i\right]\right] &= \Pr\left[-\sum_{i=1}^n X_i \geq -(1 - \epsilon) \cdot \mathbf{E}\left[\sum_{i=1}^n X_i\right]\right] \\
&= \Pr\left[-h \cdot \sum_{i=1}^n X_i \geq -h \cdot (1 - \epsilon) \cdot \mathbf{E}\left[\sum_{i=1}^n X_i\right]\right] \\
&= \Pr\left[e^{-h \cdot \sum_{i=1}^n X_i} \geq e^{-h \cdot (1 - \epsilon) \cdot \mathbf{E}\left[\sum_{i=1}^n X_i\right]}\right] \\
&\leq \frac{\mathbf{E}\left[e^{-h \cdot \sum_{i=1}^n X_i}\right]}{e^{-h \cdot (1 - \epsilon) \cdot \mathbf{E}\left[\sum_{i=1}^n X_i\right]}}
\end{aligned}$$

Similarly, as in the proof of Inequality 2.1 we obtain

$$\mathbf{E}\left[e^{-h \cdot \sum_{i=1}^n X_i}\right] = e^{\mathbf{E}\left[\sum_{i=1}^n X_i\right](e^{-h} - 1)}$$

and

$$\Pr\left[\sum_{i=1}^n X_i \leq (1 - \epsilon) \cdot \mathbf{E}\left[\sum_{i=1}^n X_i\right]\right] \leq e^{(e^{-h} + (1 - \epsilon)h - 1) \cdot \mathbf{E}\left[\sum_{i=1}^n X_i\right]}$$

We set  $h = \ln(1/(1 - \epsilon))$  and get

$$e^{(e^{-h} + (1 - \epsilon)h - 1) \cdot \mathbf{E}\left[\sum_{i=1}^n X_i\right]} = e^{(-\epsilon - (1 - \epsilon) \cdot \ln(1 - \epsilon)) \cdot \mathbf{E}\left[\sum_{i=1}^n X_i\right]}$$

Using the Taylor series expansion we get

$$\ln(1 - \epsilon) = -\sum_{i \geq 1} \frac{\epsilon^i}{i}.$$

Multiplying this with  $(1 - \epsilon)$  gives

$$(1 - \epsilon) \cdot \ln(1 - \epsilon) = \sum_{i \geq 1} \frac{\epsilon^{i+1}}{i} - \sum_{i \geq 1} \frac{\epsilon^i}{i} = -\epsilon + \sum_{i \geq 2} \frac{\epsilon^i}{i-1} - \sum_{i \geq 2} \frac{\epsilon^i}{i} > -\epsilon + \epsilon^2/2.$$

Finally, we get

$$\Pr\left[\sum_{i=1}^n X_i \leq (1 - \epsilon) \cdot \mathbf{E}\left[\sum_{i=1}^n X_i\right]\right] \leq e^{-\frac{\epsilon^2 \cdot \mathbf{E}\left[\sum_{i=1}^n X_i\right]}{2}}.$$

□

## 2.3 The Heavy Hitter Problem

An element of the universe  $U$  is called a  $\lambda$ -heavy hitter in an insertion-only data stream of length  $n$ , if it appears at least  $\lambda n$  times in the stream. In the heavy hitter problem we are interested to report the set of  $\lambda$ -heavy hitters in the stream. One motivation to study the heavy hitter problem

is the search for 'hot items' in streams, for example, heavily traded stocks in streams of financial transactions. Another example is the problem of detecting spreading viruses in network traffic. Here we can assume that a spreading virus uses a big share of the overall number of packets sent and so we will see many similar packets in the stream.

As we will see later in the lower bounds section, the heavy hitter problem posed as above cannot be solved using  $o(n)$  space. Therefore, we will define the following relaxed form of the heavy hitter problem.

**Problem 2.3.1 (Relaxed Heavy Hitter Problem)** *The  $\epsilon$ -relaxed  $\lambda$ -heavy hitter problem is to find a set  $H \subseteq U$  such that*

- (i)  $U$  contains every  $\lambda$ -heavy hitter,
- (ii)  $U$  contains no item that appears less than  $(\lambda - \epsilon)n$  times in the stream.

We will now consider the following simple approach to the heavy hitter problem. We maintain a random sample of  $s$  elements from the stream chosen independently and uniformly at random with repetition. Then

REPORTHEAVYHITTER( $\lambda, \epsilon$ )

S: Set of  $s$  random elements chosen independently and uniformly at random with repetition

1. Report all elements that occur more than  $(\lambda - \epsilon/2)s$  times in  $S$

In the next section we will introduce the necessary tools to analyze our algorithm and perform a first analysis.

### 2.3.1 Analysis Using Chebyshev inequality

We apply a similar analysis to the one above to the heavy hitter algorithm. Let us fix a stream  $\sigma = (\sigma_1, \dots, \sigma_n)$  and let us consider an arbitrary  $x \in U$  and assume that  $x$  occurs  $\lambda^*n$  times in  $\sigma$  for some  $\lambda^* \in [0, 1]$ . We will first determine the probability that  $x$  is reported by our algorithm. Let  $X_i$  denote the indicator random variable for the event that the  $i$ -th element from our random sample is  $x$ . Since the  $i$ -th element of the sample is taken uniformly at random from the stream, we know that  $\Pr[X_i = 1] = \lambda^*$ . It immediately follows that

$$\mathbf{E}[X_i] = 0 \cdot \Pr[X_i = 0] + 1 \cdot \Pr[X_i = 1] = \lambda^* .$$

Linearity of expectation implies that

$$\mathbf{E}\left[\sum_{i=1}^s X_i\right] = \lambda^* s .$$

Plugging these values into the analysis from Section 2.2.2 (using  $M = 1$ ) we obtain

$$\Pr\left[\left|\sum_i X_i - \mathbf{E}\left[\sum_i X_i\right]\right| \geq \epsilon/2 \cdot s\right] \leq \frac{1}{4\epsilon^2 \cdot s} .$$

Now let us consider the case that  $x$  is a heavy hitter, i.e.  $\lambda^* \geq \lambda$ . Choosing  $s = \frac{4}{\delta \cdot \epsilon^2}$  and using that  $\lambda^* \geq \lambda$  we obtain that

$$\begin{aligned}
\Pr\left[\sum_i X_i > (\lambda - \epsilon/2)s\right] &\geq \Pr\left[\sum_i X_i > \mathbf{E}\left[\sum_i X_i\right] - \epsilon s/2\right] \\
&= \Pr\left[\mathbf{E}\left[\sum_i X_i\right] - \sum_i X_i < \epsilon s/2\right] \\
&= 1 - \Pr\left[\mathbf{E}\left[\sum_i X_i\right] - \sum_i X_i \geq \epsilon s/2\right] \\
&\geq 1 - \Pr\left[\left|\sum_i X_i - \mathbf{E}\left[\sum_i X_i\right]\right| \geq \epsilon s/2\right] \\
&\geq 1 - \delta
\end{aligned}$$

the probability of reporting  $x$  is at least  $1 - \delta$ .

Next we consider the case that  $x$  is not a  $(\lambda - \epsilon)$ -heavy hitter, i.e.  $\lambda^* < \lambda - \epsilon$ . Choosing  $s = \frac{4}{\delta \cdot \epsilon^2}$  and applying a similar analysis to the one above we obtain that the probability of reporting  $x$  is at most  $1 - \delta$ .

Finally, we have to choose  $\delta$  to make sure that (a) every  $\lambda$ -heavy hitter is reported and (b) no element is reported that occurs less than  $(\lambda - \epsilon)n$  times in the stream. To prove this, we have to apply the following trick for the analysis. If two items in the stream appear less than  $(\lambda - \epsilon)n/2$  times in the stream, we replace the item that occurs fewer times by the item that occur more often in stream. Clearly, if the item is not reported after this procedure, then none of the original items is reported. The benefit of this procedure is, that every item that occurs less than  $(\lambda - \epsilon)n$  times occurs at least  $(\lambda - \epsilon)n/2$  times. Thus, we have at most  $2/(\lambda - \epsilon)$  different items in the stream. Therefore, we obtain by the union bound

$$\Pr[\text{The output of the algorithm is correct}] \geq 1 - (\delta \cdot 2/(\lambda - \epsilon)) .$$

Choosing  $\delta \leq \delta'(\lambda - \epsilon)/2$  gives an overall probability of error of at most  $\delta'$  and a sample size  $s = \frac{8}{\delta' \cdot (\lambda - \epsilon) \epsilon^2}$ .

**Theorem 8** *Algorithm REPORTHEAVYHITTER with parameter  $s = \frac{8}{\delta' \cdot (\lambda - \epsilon) \epsilon^2}$  outputs with probability  $1 - \delta'$  a set  $H$  that*

(a) *contains all  $\lambda$ -heavy hitters and*

(b) *contains no item that occurs less than  $(\lambda - \epsilon)n$  times in the stream.*

### 2.3.2 Analysis Using Chernoff Bounds

To analyze the heavy hitter algorithm via Chernoff bounds we can apply a similar analysis as above. To prove that the algorithm accepts a  $\lambda$ -heavy hitter  $x$ , we show that for  $\lambda^* \geq \lambda$  we have

$$\begin{aligned} \Pr\left[\sum_i X_i > (\lambda - \epsilon/2)s\right] &= \Pr\left[\sum_i X_i > \left(1 - \frac{\epsilon}{2\lambda}\right)\lambda s\right] \\ &\geq \Pr\left[\sum_i X_i > \left(1 - \frac{\epsilon}{2\lambda}\right)\lambda^* s\right] \\ &\geq \Pr\left[\sum_i X_i > \left(1 - \frac{\epsilon}{2}\right)\lambda^* s\right], \end{aligned}$$

since increasing the right hand side of the inequality decreases the probability of success. Now

$$\begin{aligned} \Pr\left[\sum_i X_i > \left(1 - \frac{\epsilon}{2}\right)\lambda^* s\right] &= \Pr\left[\sum_i X_i > \left(1 - \frac{\epsilon}{2}\right)\mathbf{E}\left[\sum_i X_i\right]\right] \\ &= 1 - \Pr\left[\sum_i X_i \leq \left(1 - \frac{\epsilon}{2}\right) \cdot \mathbf{E}\left[\sum_i X_i\right]\right] \\ &\geq 1 - e^{-\frac{1}{2}\left(\frac{\epsilon}{2}\right)^2 \cdot \mathbf{E}\left[\sum_i X_i\right]} \\ &\geq 1 - e^{-\frac{\lambda \cdot \epsilon^2 \cdot s}{8}} \end{aligned}$$

This implies that for  $s = \frac{8 \ln(\delta^{-1})}{\lambda \epsilon^2}$  we obtain a probability of at least  $1 - \delta$  for reporting  $x$ .

Now we consider the case that  $x$  is not a  $(\lambda - \epsilon)$ -heavy hitter, i.e.  $\lambda^* \leq \lambda - \epsilon$ . We can assume that  $\lambda^* = \lambda - \epsilon$ , since the probability of seeing too many elements in the sample is maximized when the individual probability of seeing an element is maximized. In this case we obtain

$$\begin{aligned} \Pr\left[\sum_i X_i < (\lambda - \epsilon/2)s\right] &= \Pr\left[\sum_i X_i < \left(1 - \frac{\epsilon}{2\lambda}\right)\lambda s\right] \\ &= \Pr\left[\sum_i X_i < \left(1 - \frac{\epsilon}{2(\lambda^* + \epsilon)}\right)(\lambda^* + \epsilon)s\right] \\ &= \Pr\left[\sum_i X_i < \lambda^* s + \epsilon s/2\right] \\ &= \Pr\left[\sum_i X_i < \left(1 - \frac{\epsilon}{2\lambda^*}\right)\lambda^* s\right] \\ &= \Pr\left[\sum_i X_i < \left(1 + \frac{\epsilon}{2\lambda^*}\right) \cdot \mathbf{E}\left[\sum_i X_i\right]\right] \\ &\leq \Pr\left[\sum_i X_i < \left(1 + \frac{\epsilon}{2}\right) \cdot \mathbf{E}\left[\sum_i X_i\right]\right] \\ &= 1 - \Pr\left[\sum_i X_i \leq \left(1 + \frac{\epsilon}{2}\right) \cdot \mathbf{E}\left[\sum_i X_i\right]\right] \\ &\geq 1 - e^{-\frac{1}{3}\left(\frac{\epsilon}{2}\right)^2 \cdot \mathbf{E}\left[\sum_i X_i\right]} \\ &\geq 1 - e^{-\frac{\lambda \cdot \epsilon^2 \cdot s}{12}} \end{aligned}$$

Hence, for  $s = \frac{12 \ln(\delta^{-1})}{\lambda \epsilon^2}$  we obtain a probability of not reporting  $x$  of at least  $1 - \delta$ . Following the analysis for the Chebyshev inequality we obtain

**Theorem 9** Algorithm REPORTHEAVYHITTER with parameter  $s = \frac{12 \ln(\frac{2}{(\lambda - \epsilon) \delta'})}{\lambda \epsilon^2}$  outputs with probability  $1 - \delta'$  a set  $H$  that

- (a) contains all  $\lambda$ -heavy hitters and
- (b) contains no item that occurs less than  $(\lambda - \epsilon)n$  times in the stream.

## 2.4 Clustering Data Streams

One approach to the analysis of large data sets is clustering. In clustering we partition a set of objects into subsets called *clusters*, such that, ideally, each subset contains similar objects and objects from different subset are dissimilar. A typical application of clustering is to get a smaller representation of the input data. Instead of dealing with the whole data set, we represent the data by taking one representative for each cluster.

One standard approach in clustering is to describe objects by a set of  $d$  real-valued *features*. For example, when we are monitoring internet traffic, we can easily transform information in the packet headers (source IP, destination IP, packet size, etc.) into real values and then obtain a clustering. It is likely that we can detect anomalies in network traffic by only looking at this clustering. A description of objects via feature vectors allows us to view each object as a point in a  $d$ -dimensional feature space. A natural measure for dissimilarity of points is then given by the Euclidean distance between these points.

Even in the Euclidean case there are many different way to formalize clustering. Here, we will consider the  $k$ -median clustering problem, which is defined as follows.

**Problem 2.4.1 (k-median)** Let  $P$  be a set of points in  $\mathbb{R}^d$ . Then the  $k$ -median problem is to find  $k$  points  $c_1, \dots, c_k \in \mathbb{R}^d$  such that

$$\sum_{p \in P} \min_{1 \leq i \leq k} d(p, c_i)$$

is minimized, where  $d(p, c_i) := \|p - c_i\|_2$ .

We define  $d(p, C) = \min_{c \in C} d(p, c)$  and  $cost(P, C) = \sum_{p \in P} d(p, C)$ . Using this notation the  $k$ -median problem is to find a set  $C$  of  $k$  points that minimizes  $cost(P, C)$ . We will sometimes call the points in  $C$  the centers of a clustering.

Our goal is to analyze the following simple algorithmic approach to clustering a point set  $P$  with respect to the  $k$ -median quality measure.

CLUSTERINGSAMPLES( $P, s$ )

1. Draw a random subset  $S \subset P$  of  $s$  points uniformly at random with repetition
2. Run an arbitrary  $\alpha$ -approximation algorithm for  $k$ -median on input  $S$
3. Output the computed centers as a solution for  $P$

First of all, we can easily apply this approach in data streaming. We know how to draw random samples and so we simply maintain a random sample and then compute a clustering for it. The main question is whether we can give bounds for the quality of this approach.

Unfortunately, in the general case, our algorithm can be arbitrarily bad for certain inputs. Consider, for example, an input that consists of a set  $Q$  of  $n - 1$  points that are close to each other and a single point  $r$ , which is far away from the rest. Since a small random sample typically does not contain  $r$ , every (approximate) solution for clustering the sample will be close to  $Q$ . However, if the distance between  $Q$  and  $r$  is large enough, the distance between  $r$  and the nearest center (which is close to  $Q$ ) will dominate the cost, i.e. the cost of the clustering depends on the distance between  $Q$  and  $r$ . However, the cost of an optimal clustering does not depend on this distance, since we can place one center at  $r$ . Hence, the gap between an approximate solution for a random sample of size  $o(n)$  and an optimal solution can be arbitrarily large with probability going to 1. One way to deal with this situation that models many applications nicely is to assume that all input points are contained in a ball  $B$  and then parametrize the analysis in terms of the diameter  $M$  of this ball.

For the analysis we will use the following lemma, which we are not going to prove. It states that if we have a set of functions that assigns values to elements from a set  $X$  and if the function values are bounded, then the average function value can be approximated by the average function values of a random sample of sufficient size.

**Lemma 2.4.2** *Let  $F$  be a finite set of functions from  $X$  to  $\mathbb{R}$  with  $0 \leq f(x) \leq M$  for all  $f \in F$  and  $x \in X$ . Let  $S = x_1, \dots, x_s$  be a sequence of  $s$  objects that are drawn independently and uniformly from  $X$  and let  $\epsilon > 0$ . Then*

$$\Pr[\exists f \in F : \left| \frac{\sum_{x \in X} f(x)}{|X|} - \frac{\sum_{x \in S} f(x)}{|S|} \right| \geq \epsilon] \leq \delta,$$

if

$$s \geq \frac{M^2}{2\epsilon^2} \cdot \left( \ln |F| + \ln \frac{2}{\delta} \right).$$

### 2.4.1 Overview of the Analysis

We will use Lemma 2.4.2 to analyze the quality of random samples with respect to the  $k$ -median problem. We observe that, once the positions of the cluster centers are fixed, the cost of every point (the distance to the nearest cluster center) is also fixed. We will use this observation to apply Lemma 2.4.2. To do so, we set  $X = P$  and define for every set  $C$  of  $k$  centers in  $B$  a function  $f_C(x) = d(x, C)$ . Since  $B$  has diameter  $M$  and the centers and points are in  $B$ , we know that  $0 \leq f_C(x) \leq M$  for all  $x \in P$  and all  $f_C$ . We remark that it suffices to consider centers from  $B$  since if we have centers outside of  $B$  we can improve them by projecting them onto  $B$ . We also assume that our  $\alpha$ -approximation algorithm computes centers in  $B$ .

We would like to apply Lemma 2.4.2, but the set of functions  $f_C$  is not finite. To overcome this problem, we discretize the solution space by putting a grid over the ball  $B$ . If this grid is fine enough, it will suffice to consider only solutions on this grid. In particular, if the grid has cell width  $Z$  then for each  $c \in B$  we have a grid point in distance  $Z \cdot \sqrt{d}/2$ . Let  $C$  be an arbitrary set of centers in  $B$  and let  $C'$  be the set of centers obtained by moving each center from

C to the closest grid point. The difference between the cost of C and C' is at most  $n \cdot Z \cdot \sqrt{d}/2$  by the triangle inequality.

## 2.4.2 Analysis of the Random Sampling Approach

We will prove the following theorem.

**Theorem 10** *Let  $s \geq \frac{9M^2\alpha^2}{2\epsilon^2} \cdot (\ln(\lceil \frac{3\sqrt{d}M}{\epsilon} \rceil^{dk} + 1) + \ln \frac{2}{\delta})$ . Then the solution L computed by algorithm CLUSTERINGSAMPLES satisfies with probability  $1 - \delta$ :*

$$\text{cost}(P, L) \leq \alpha \cdot \text{cost}(P, \text{Opt}) + \epsilon n ,$$

where *Opt* denotes the cost of an optimal solution for P.

**Proof :** We use a grid with cell width  $Z = \frac{\epsilon}{3\sqrt{d}}$ . Let G be the set of grid points in B. Define  $f_C : P \rightarrow \mathbb{R}$  with  $f_C(p) = d(p, C)$  for  $C \subseteq \mathbb{R}^d$ . Let  $F' = \{f_C | C \subseteq G, |C| = k\}$  and  $F = F' \cup \{f_{\text{Opt}}\}$ , where  $f_{\text{Opt}}$ . It follows that  $|F| \leq (\lceil \frac{M}{Z} \rceil)^{dk} + 1$ . We apply Lemma 2.4.2 with error parameter  $\epsilon/(3\alpha)$  and hence  $s \geq \frac{9M^2\alpha^2}{2\epsilon^2} (\ln |F| + \ln \frac{2}{\delta})$  and obtain

$$\Pr[\exists f_C \in F : |\frac{\sum_{x \in P} f_C(x)}{|P|} - \frac{\sum_{x \in S} f_C(x)}{|S|}| \leq \frac{\epsilon'}{3\alpha}] \leq \delta .$$

Using that  $\sum_{x \in P} f_C(x) = \text{cost}(P, C)$  and  $\sum_{x \in S} f_C(x) = \text{cost}(S, C)$  we get

$$\Pr[\exists f_C \in F : |\frac{\text{cost}(P, C)}{|P|} - \frac{\text{cost}(S, C)}{|S|}| \leq \frac{\epsilon'}{3\alpha}] \leq \delta$$

and hence

$$\Pr[\forall f_C \in F : |\frac{\text{cost}(P, C)}{|P|} - \frac{\text{cost}(S, C)}{|S|}| \leq \frac{\epsilon'}{3\alpha}] > 1 - \delta .$$

Now let  $\mathcal{A}$  be an arbitrary  $\alpha$ -Approximationsalgorithmus and L be the solution computed by  $\mathcal{A}$  on input S. We know from above that

$$\frac{\text{cost}(S, \text{Opt})}{s} < \frac{\text{cost}(P, \text{Opt})}{n} + \frac{\epsilon}{3\alpha} \tag{2.4}$$

with probability at least  $1 - \delta$ . Hence, we know that an  $\alpha$ -approximation algorithms finds a solution L satisfying

$$\text{cost}(S, L) \leq \alpha \frac{\text{cost}(P, \text{Opt})}{n} + \frac{\epsilon}{3} .$$

Now, let *Opt* be an optimal set of k centers and let  $C_{\text{bad}} \subseteq B$  be an arbitrary solution with  $|C_{\text{bad}}| = k$  and  $\text{cost}(P, C_{\text{bad}})/n > \alpha \cdot \text{cost}(P, \text{Opt})/n + \epsilon$ . Let  $C_{\text{bad}}^*$  be the grid solution

obtained by moving the centers from  $C_{\text{bad}}$  to the nearest grid point. We know that

$$\begin{aligned}
\frac{\text{cost}(S, C_{\text{bad}})}{s} &\geq \frac{\text{cost}(S, C_{\text{bad}}^*)}{s} - \frac{\epsilon}{6} \\
&= \frac{\text{cost}(P, C_{\text{bad}}^*)}{s} - \frac{\epsilon}{6} - \left( \frac{\text{cost}(P, C_{\text{bad}}^*)}{s} - \frac{\text{cost}(S, C_{\text{bad}}^*)}{s} \right) \\
&\geq \frac{\text{cost}(P, C_{\text{bad}}^*)}{s} - \frac{\epsilon}{6} - \left| \frac{\text{cost}(P, C_{\text{bad}}^*)}{s} - \frac{\text{cost}(S, C_{\text{bad}}^*)}{s} \right| \\
&> \frac{\text{cost}(P, C_{\text{bad}}^*)}{n} - \frac{\epsilon}{2} \\
&\geq \frac{\text{cost}(P, C_{\text{bad}})}{n} - \frac{2\epsilon}{3} \\
&> \alpha \cdot \frac{\text{cost}(P, \text{Opt})}{n} + \epsilon/3
\end{aligned}$$

with probability  $1 - \delta$ , where the fourth inequality follows from inequality 2.4 and  $\alpha \geq 1$ . Hence, with probability  $1 - \delta$  no solution  $C_{\text{bad}}$  is returned by the algorithms. It follows that

$$\frac{\text{cost}(P, L)}{n} \leq \alpha \cdot \frac{\text{cost}(P, \text{Opt})}{n} + \epsilon ,$$

which implies

$$\text{cost}(P, L) \leq \alpha \cdot \text{cost}(P, \text{Opt}) + \epsilon n .$$

□

## 2.5 Sampling in Dynamic Data Streams

We will now consider the problem of sampling from dynamic data streams. First of all, we have to define what it means to sample from a dynamic data stream. Recall that a dynamic data stream can be viewed as a sequence of insertions and deletions into a multiset of objects  $Z$ . Here, we will only consider the case that  $M = 1$ , i.e. each object occurs at most once in  $Z$ . In this scenario we are interested in obtaining a random sample from  $Z$ . We cannot apply reservoir sampling because it may happen that the object currently in the sample is deleted. Then we are left with no sample object (and we have stored no information about the objects already processed). Another problem is similar to the case of the  $k$ -set data structure studied in Chapter 1. It may happen that there is a long sequence of insertions and almost all of these items are later deleted. We will use a similar trick as in Chapter 1 to overcome this problem.

### 2.5.1 The Data Structure

We will use  $Z$  to denote the current set of items.  $Z$  will be a set of items, i.e. we do not allow multiple occurrences of a single item ( $M = 1$ ). Our data structure is parametrized by a failure probability  $\delta$ . It supports three operations:

- $\text{INSERT}(x)$ :  $Z = Z \cup \{x\}$ , where  $x \in U$  and it is assumed that  $x \notin Z$ .

- **DELETE(x)**:  $Z = Z - \{x\}$ , where it is assumed that  $x \in Z$ .
- **SAMPLE()**: An item chosen uniformly at random from  $Z$  is returned or the algorithm outputs 'failed'.

To choose a set of  $s$  items uniformly at random with repetition, we can run  $O(s)$  instances in parallel and select the union of the first  $s$  instances that did not return 'failed'.

## 2.5.2 The Algorithm

For simplicity let us first assume that  $|Z|$  is known to the algorithm in advance. Then the main idea is very simple and kind of similar to the one used for the  $k$ -set structure. We use a fully random function  $h : U \rightarrow \{1, \dots, |Z|\}$  (see next chapter for a more detailed discussion) and apply the  $k$ -set data structure for  $k = 1$  developed in Chapter 1 to  $h^{-1}(1)$ , i.e. to all elements mapped to 1. If only one element is mapped to 1, then it is chosen uniformly at random from all elements in  $Z$ , because  $h$  is fully random. As we will see later, if we know  $|Z|$  or a good approximation of it our algorithm will return a random element with constant probability. However, if there are many more elements in  $Z$  than we have guessed, it is very likely that more than one element is mapped to 1 and so our algorithm fails. Similarly, if  $|Z|$  is much smaller than our guess, it is quite likely that no element is mapped to 1 and the algorithm fails with high probability. To overcome this problem we run one instance of our algorithm for every guess  $2^i$  for  $0 \leq i \leq \lceil \log U \rceil$ . Then we check all instances in order of increasing  $i$ . We return the sample of the first instance that did not fail or we return 'failed', if all instances failed. We refer with  $h_i : U \rightarrow \{1, \dots, 2^i\}$  to the function corresponding to the guess  $|Z| = 2^i$  and  $UE_i$  to the corresponding 1-set data structure (**UE** is used to abbreviate 'unique element').

**INSERT(x)**

1. **for**  $i = 0$  **to**  $\lceil \log U \rceil$  **do**
2.     **if**  $h_i(x) = 1$  **then**
3.          $UE_i$ .**INSERT(x)**

**DELETE(x)**

1. **for**  $i = 0$  **to**  $\lceil \log U \rceil$  **do**
2.     **if**  $h_i(x) = 1$  **then**
3.          $UE_i$ .**DELETE(x)**

For the **SAMPLE** procedure we modify **REPORT** in such a way that it reports 'failed', if there is not exactly one element present in the data structure.

**SAMPLE()**

1. **for**  $i = 0$  **to**  $\lceil \log U \rceil$  **do**
2.     **if**  $UE$ .**REPORT**  $\neq$  failed **then**
3.         **RETURN**  $UE_i$ .**REPORT**

### 2.5.3 Analysis

We show that with constant probability the algorithm return a sample. By the above discussion it suffices to show that there exists  $i$  such that  $|h_i^{-1}(1)| = 1$ . Let us consider  $i = \lfloor \log |Z| \rfloor$ , i.e.  $2^i \leq |Z| < 2^{i+1}$ . For  $i = 0$  we have  $|Z| = 1$  and only need to maintain a single element. Thus, our algorithms works correctly. Given  $Z$  and  $i \geq 1$ , the probability that exactly one element is mapped to 1 is at least

$$\Pr[|h_i^{-1}(1)| = 1] = \frac{|Z| \cdot (2^i - 1)^{|Z|-1}}{(2^i)^{|Z|}} = \frac{|Z|}{2^i} \cdot \left(1 - \frac{1}{2^i}\right)^{|Z|-1} \geq \left(1 - \frac{1}{2^i}\right)^{2^{i+1}-1} \geq \left(1 - \frac{1}{2^i}\right)^{2^{i+1}} \geq 1/e^2 .$$

We summarize our result below.

**Theorem 11** *Algorithm SAMPLE returns with probability at least  $1/e^2$  an element chosen uniformly at random from the current set of items  $Z$  in the dynamic data stream model. If the algorithm does not report an item, it returns an error message 'failed'. The algorithm uses  $O(\log^2 |U|)$  space.*

## 3 Hashing and Random Projections

In this chapter we will take a closer look at random functions and discuss how much space is needed to represent them. The first question to address is how to choose a random function from a universe  $\mathcal{U}$  to a set  $[m] := \{0, \dots, m-1\}$ . For this purpose, we usually define a class  $\mathcal{H}$  of functions  $h : \mathcal{U} \rightarrow [m]$  and pick a function from  $\mathcal{H}$  uniformly at random. The difficulty is to define a set of functions  $\mathcal{H}$  such that

- a random  $h \in \mathcal{H}$  can be represented in small space,
- $h$  is reasonably close (in some well-defined mathematical sense) to a random function from the set of all functions from  $\mathcal{U}$  to  $[m]$  (ideal hash functions),
- the process of sampling  $h$  is efficient, and
- $h$  can be efficiently evaluated.

### 3.1 Ideal Hash Functions

We start with the definition of an ideal hash function.

**Definition 3.1.1** *A set  $\mathcal{H}$  of functions  $h : \mathcal{U} \rightarrow [m]$  is called an ideal class of hash functions, if a randomly chosen  $h \in \mathcal{H}$  satisfies for any  $x, y_1, \dots, y_\ell \in \mathcal{U}$  with  $y_i \neq y_j$  and  $x \neq y_i$ ,  $1 \leq i, j \leq \ell$ , and  $z, r_1, \dots, r_\ell \in [m]$*

$$\Pr[h(x) = z \mid \bigwedge_{i=1}^{\ell} h(y_i) = r_i] = \frac{1}{m} .$$

The intuitive benefit of ideal hash functions is that no matter 'how much we know' about  $h$ , if we do not know the image of an element  $x \in \mathcal{U}$ , it will be distributed uniformly at random in  $[m]$ . This simplifies the analysis of many algorithms significantly. So, why do we not always use ideal hash functions in our algorithms? The reason is that the only set of functions that is an ideal class of hash functions is the set of all functions from  $\mathcal{U}$  to  $[m]$ .

**Lemma 3.1.2** *The only set of functions that is a class of ideal hash functions is the set  $\mathcal{H}_{all}$  of all functions from  $\mathcal{U}$  to  $[m]$ .*

**Proof:** We first show that  $\mathcal{H}_{all}$  is an ideal class of hash functions. This can be seen as follows. Fixing  $\ell$  values of the functions in  $\mathcal{H}_{all}$  leaves us with  $m^{|\mathcal{U}|-\ell}$  different functions. Out of these functions  $m^{|\mathcal{U}|-\ell-1}$  map  $x$  to  $z$ . Hence,

$$\Pr[h(x) = z \mid \bigwedge_{i=1}^{\ell} h(y_i) = r_i] = \frac{m^{|\mathcal{U}|-\ell-1}}{m^{|\mathcal{U}|-\ell}} = \frac{1}{m} .$$

Now assume that  $\mathcal{H}$  is a class of ideal hash functions and some function  $h^* : \mathcal{U} \rightarrow [m]$  is not present in  $\mathcal{H}$ . Now let us consider a random function  $h \in \mathcal{H}$ . Let  $k$  be the smallest index such that there exists  $h \in \mathcal{H}$  with  $h(i) = h^*(i)$  for all  $k \leq i \leq m$ . Since  $h^* \notin \mathcal{H}$  we conclude  $k > 1$  and obtain

$$\Pr[h(k-1) = h^*(k-1) \mid \bigwedge_{i=k}^m h(i) = h^*(i)] = 0 ,$$

which contradicts our assumption that  $\mathcal{H}$  is an ideal class of hash functions. □

So we know that the only set of ideal hash functions is the set all all functions from  $\mathcal{U}$  to  $[m]$ . Why is it a bad ideal to use this set in our algorithms? The reason is that there are many functions from  $\mathcal{U}$  to  $[m]$ , which implies that storing one function requires much space.

**Lemma 3.1.3** *The number of functions from  $\mathcal{U}$  to  $[m]$  is  $m^{|\mathcal{U}|}$ .*

**Proof :** Each element from  $\mathcal{U}$  is mapped to a value between 1 and  $m$ . Hence, the overall number of such mappings is  $m^{|\mathcal{U}|}$ . □

Using  $x$  bits of memory we can encode  $2^x$  different functions. This implies

**Corollary 3.1.4** *Storing an arbitrary function from the set of all functions from  $\mathcal{U}$  to  $[m]$  requires  $|\mathcal{U}| \cdot \log(m)$  bits of memory.*

Thus, if we wanted to store an arbitrary function from the set of all functions from  $\mathcal{U}$  to  $[m]$  we require more bits of memory than the size of our universe! Obviously, this way too much and we have to find a way to reduce the space requirements.

## 3.2 Count-Min Sketches

In this section we will consider the problem of approximating the distribution of the elements given in a dynamic data stream. A simple strategy is to partition the universe  $\mathcal{U} = \{1, \dots, M\}$  into  $k$  buckets each containing  $M/k$  elements from  $\mathcal{U}$ . For each bucket we maintain a counter that stores the number of elements in the bucket. One problem of this approach is that most elements might be contained in a single bucket. Then we do not know get much information using our approach.

For example, it will not be possible to get information about the median of the data using the above approach. In the following we will develop a data structure to find approximate medians and so-called approximate  $\Phi$ -quantiles of the data.  $\Phi$ -quantiles can be viewed as a generalization of the median. The median of  $n$  is the element with rank  $n/2$ ; the  $\Phi$ -quantiles are all elements whose rank is a multiple of  $\Phi n$ .

**Definition 3.2.1 ( $\Phi$ -quantile)** *Given a multiset of  $n$  elements from a universe  $\mathcal{U} = \{1, \dots, M\}$  we call an element with rank  $\lceil k\Phi M \rceil$ ,  $k = 1, \dots, 1/\Phi$ , a  $\Phi$ -quantile.*

It is not possible to compute  $\Phi$ -quantiles exactly in the data stream setting. Therefore, we aim at computing a relaxation of  $\Phi$ -quantiles.

**Definition 3.2.2** *An element with rank between  $\lfloor (k \cdot \Phi - \epsilon) \cdot M \rfloor$  and  $\lceil (k \cdot \Phi + \epsilon) \cdot M \rceil$  for  $\epsilon < \Phi$  is called a relaxed  $\Phi$ -quantile.*

### 3.2.1 The Data Structure

In the following we describe a data structure, which is called a *count-min sketch*. For a given approximation parameter  $\epsilon$  and error probability  $\delta$  it uses  $d = \lceil \frac{1}{\delta} \rceil$  random functions  $h_1, \dots, h_d : \mathcal{U} \rightarrow \{1, \dots, W\}$ , where  $W = \lceil \frac{e}{\epsilon} \rceil$ .

We further need a 2-dimensional array  $count[1, 1], \dots, count[d, W]$ .  $count[i, j]$  stores the number of elements that are mapped by  $h_i$  to  $j$ , i.e.  $\sum_{x: h_i(x)=j} |h_i^{-1}(x)|$ .

We use the following operations to insert or delete an element.

INSERT( $x$ )

1. **for**  $j = 1$  **to**  $d$  **do**
2.  $count[j, h_j(x)] = count[j, h_j(x)] + 1$

DELETE( $x$ )

1. **for**  $j = 1$  **to**  $d$  **do**
2.  $count[j, h_j(x)] = count[j, h_j(x)] - 1$

It is easy to see that the data structure needs  $Wd$  memory cells plus the space to store the  $h_i$ 's.

A query to our data structure asks for the multiplicity of an element  $x$ . We know that each hash function  $h_j$  distributes the elements randomly. Further, all occurrences of  $x$  are mapped to  $h_j(x)$  and stored in  $count[j, h_j(x)]$ . Thus, in a typical situation  $count[j, h_j(x)]$  will count all occurrences of  $x$  plus approximately  $M/W$  other 'random' elements. Therefore, we can safely take the minimum of all *count* values.

QUERY( $x$ )

1. **return**  $\tilde{f}_x = \min_{1 \leq j \leq d} count[j, h_j(x)]$

Recall that  $f_x$  denotes the number of occurrences of element  $x$  in the current set.

### 3.2.2 Analysis

We will give the following bounds for the quality of approximation of a count-min sketch.

**Theorem 12** *For the output value  $\tilde{f}_x$  computed by the count-min sketch we have  $f_x \leq \tilde{f}_x$  and, with probability  $1 - \delta$ ,  $\tilde{f}_x \leq f_x + \epsilon M$ , where  $M$  is the number of elements in the current set. Insertion, Deletion, and Query of elements can be performed in  $O(\ln(1/\delta))$  time and the data structure uses  $O(\ln(1/\delta)/\epsilon)$  space plus the space required to store the hash functions  $h_j$ .*

**Proof :**

For  $x, y \in \mathcal{U}$  with  $x \neq y$  let  $I_{j,x,y}$  be the indicator random variable for the event  $h_j(x) = h_j(y)$ . We have

$$\mathbf{E}[I_{j,x,y}] = \Pr[h_j(x) = h_j(y)] = 1/W \leq \epsilon/e .$$

Let  $Y_{j,x}$  be the random variable for the number of elements other than  $x$  that are mapped to  $h_j(x)$ .

$$Y_{j,x} = \sum_{y \in U, y \neq x} I_{j,x,y} \cdot f_y .$$

Our construction ensures that

$$\text{count}[j, h_j(x)] = f_x + Y_{j,x} .$$

Thus, it follows that

$$\min \text{count}[j, h_j(x)] \geq f_x .$$

Further, we have

$$\mathbf{E}[Y_{j,x}] = \mathbf{E} \left[ \sum_{y \in U, y \neq x} I_{j,x,y} \cdot f_y \right] \leq \frac{\epsilon}{e} \cdot M .$$

It follows that

$$\begin{aligned} \Pr[\tilde{f}_x > f_x + \epsilon M] &= \Pr[\forall j : \text{count}[j, h_j(x)] > f_x + \epsilon M] \\ &= \Pr[\forall j : f_x + Y_{j,x} > f_x + \epsilon M] \\ &= \Pr[\forall j : Y_{j,x} > \epsilon \cdot \mathbf{E}[Y_{j,x}]] \\ &= (\Pr[Y_{j,x} > \epsilon \cdot \mathbf{E}[Y_{j,x}]])^d \\ &< e^{-d} \\ &\leq \delta \end{aligned}$$

□

### 3.3 The Johnson-Lindenstrauss Lemma and Second Frequency Moments

In this section we will prove a result that in the first place seems to have nothing to do with streaming algorithms. We consider a point set in a high dimensional space and show that there always exists a mapping (embedding) of this point set in a smaller dimensional space such that all pairwise distances between the points are approximately the same as in the source space. The result is typically known as the Johnson-Lindenstrauss theorem:

**Theorem 13 (Johnson-Lindenstrauss)** *Let  $P$  be a set of  $n$  points in  $\mathbb{R}^d$ ,  $0 < \epsilon < 1$  and  $c$  be a sufficiently large constant. Then there exists an embedding  $\pi : P \rightarrow \mathbb{R}^{c \log n / \epsilon^2}$ , such that the following inequality holds for all  $p, q \in P$ :*

$$(1 - \epsilon) \cdot \|p, q\|_2 \leq \|\pi(p), \pi(q)\|_2 \leq (1 + \epsilon) \cdot \|p, q\|_2 .$$

Thus we can reduce the dimensionality of a point set from  $d$  to  $O(\log n / \epsilon^2)$ , if we allow a distortion of upto  $(1 \pm \epsilon)$  in the pairwise distances between points. The proof of the theorem is algorithmic, i.e. it provides a method to construct the embedding explicitly.

The connection to streaming is as follows. If we apply the theorem to a single point  $f = (f_1, \dots, f_d)$  and 0, then it tells us that we can approximate

$$\sqrt{\sum_{i=1}^d f_i^2}$$

upto a factor of  $(1 \pm \epsilon)$  using space  $O(1/\epsilon^2)$  plus the space for the embedding. As we will see the proof will even provide a stronger result, namely that we can approximate

$$\sum_{i=1}^d f_i^2 = F_2$$

upto a factor of  $(1 \pm \epsilon)$ . Thus, we can approximate the second frequency moment using the Johnson-Lindenstrauss embedding.

To prove the Johnson-Lindenstrauss theorem, we project the point set  $P$  to a randomly selected subspace. This is done using a random projection  $\pi : \mathbb{R}^d \rightarrow \mathbb{R}^k$ ,  $k \leq d$ . One can construct such an embedding by choosing  $k$  random vectors  $r_i$  from  $\mathbb{R}^d$  independently according to a suitable distribution. For  $v \in \mathbb{R}^d$  we consider the following mapping

$$\pi(v) = (v^T \cdot r_1, \dots, v^T \cdot r_k)^T.$$

### 3.3.1 The choice of the $r_i$

Let  $r_i = (r_i^1, \dots, r_i^d)^T$ . Then we pick each entry  $r_i^j$  from  $r_i$ ,  $1 \leq j \leq d$ , uniformly at random from a Gaussian distribution with expectation 0 and variance 1, i.e.

$$\Pr[x \leq \ell] = \int_{-\infty}^{\ell} \varphi(x) dx$$

with

$$\varphi(x) = \frac{1}{\sqrt{2 \cdot \pi}} \cdot e^{-\frac{1}{2} \cdot x^2}.$$

If we use  $A$  to denote the  $k \times d$  matrix, whose  $k$  rows are the vectors  $r_1, \dots, r_k$ , then we can write  $\phi(v) = Av$ . Hence,  $\phi$  is a linear function and thus satisfies  $\pi(v_1 - v_2) = \pi(v_1) - \pi(v_2)$  for all  $v_1, v_2 \in \mathbb{R}^d$ . It suffices to prove that for all  $p, q \in P$  the length of the vector  $v := p - q$  is approximated by the length of  $\pi(v) = \pi(p - q)$ . Since we would like to approximate the length of this vector with relative error, we can assume that  $\|v\|_2 = 1$  since  $\pi$  is linear. We want to prove that with probability  $1 - \frac{1}{3n^2}$  the length of  $\pi(v)$  (after scaling) approximates the length of  $v$  upto a multiplicative error of  $1 \pm \epsilon$ . In fact, we prove a stronger result, namely, that we get a  $(1 + \epsilon)$ -approximation for the square of the Euclidean distance. The proof of the lemma below is quite similar to the proof of Chernoff bounds.

**Lemma 3.3.1** *Let  $v \in \mathbb{R}^d$  be a unit vector and  $\pi : \mathbb{R}^d \rightarrow \mathbb{R}^k$  be defined as above. Then we have for  $0 < \epsilon < 1$  and  $k \geq c \cdot \log n / \epsilon^2$*

$$\Pr[(1 - \epsilon) \leq \frac{\|\pi(v)\|_2^2}{k \cdot \|v\|_2^2} \leq (1 + \epsilon)] \geq 1 - \frac{1}{3n^2},$$

where  $c$  is a sufficiently large constant.

**Proof :** Let  $v = (v^1, \dots, v^d) \in \mathbb{R}^d$  with  $\|v\|_2^2 = \|v\|_2 = 1$ . We consider the random variables  $X_i = v \cdot r_i = \sum_{j=1}^d v^j \cdot r_i^j$ . By the properties of the normal distribution we have that  $v^j \cdot r_i^j$  is normally distributed with expectation 0 and variance  $(v^j)^2$ . The sum of independent normal distribution is again a normal distribution whose expectation and variance is the sum of the expectation and variance of the former distributions. Hence,  $X_i$  is normally distributed with expectation  $\mathbf{E}[X_i] = 0$  and variance  $\mathbf{Var}[X_i] = \sum_{j=1}^d (v^j)^2 \cdot \mathbf{Var}[r_i^j] = \sum_{j=1}^d (v^j)^2 = 1$ . Now, let  $Y = \|\pi(v)\|_2^2$  be a random variable for the square of the length of  $\pi(v)$ . We have  $Y = \sum_{i=1}^d X_i^2$ . Furthermore,

$$\begin{aligned}
\mathbf{E}[Y] &= \mathbf{E}\left[\sum_{i=1}^k X_i^2\right] \\
&= \sum_{i=1}^k \mathbf{E}[X_i^2] \\
&= \sum_{i=1}^k (\mathbf{Var}[X_i] + (\mathbf{E}[X_i])^2) \\
&= \sum_{i=1}^k \mathbf{Var}[X_i] \\
&= k
\end{aligned}$$

Next, we prove that  $Y$  is sharply concentrated.

$$\Pr[Y \geq (1 + \epsilon)k] = \Pr[e^{sY} \geq e^{s(1+\epsilon)k}] \quad (3.1)$$

$$\leq \mathbf{E}[e^{sY}] / e^{s(1+\epsilon)k} \quad (3.2)$$

$$= e^{-s(1+\epsilon)k} \cdot \mathbf{E}\left[\prod_{i=1}^k e^{sX_i}\right] \quad (3.3)$$

$$= e^{-s(1+\epsilon)k} \cdot (\mathbf{E}[e^{sX_i^2}])^k, \quad (3.4)$$

where equality 3.1 holds for all  $s > 0$ , inequality 3.2 follows from Markov's inequality and equality 3.3 follows from  $Y = \sum_{i=1}^k X_i^2$ . Furthermore, we know that the  $X_i$  are distributed according to the normal distribution. Every  $X_i$  has density function  $\varphi(t) = \frac{1}{\sqrt{2\pi}} \cdot e^{-\frac{1}{2}t^2}$ . From this we conclude

$$\begin{aligned}
\mathbf{E}[e^{sX_i^2}] &= \frac{1}{\sqrt{2\pi}} \cdot \int_{-\infty}^{\infty} e^{st^2} \cdot e^{-t^2/2} dt \\
&= \frac{1}{\sqrt{2\pi}} \cdot \int_{-\infty}^{\infty} e^{-t^2(1-2s)/2} dt \\
&= \frac{1}{\sqrt{2\pi}} \cdot \frac{1}{\sqrt{1-2s}} \cdot \int_{-\infty}^{\infty} e^{-z^2/2} dz \\
&= \frac{1}{\sqrt{1-2s}},
\end{aligned}$$

where the second inequality from below follows from the assumption  $s < 1/2$  by substitution of  $z = t^2 \cdot \sqrt{1 - 2s}$ . We get

$$\Pr[Y \geq (1 + \epsilon)k] \leq \frac{e^{-s(1+\epsilon)k}}{(1 - 2s)^{k/2}} = \left(\frac{e^{-s(1+\epsilon)}}{\sqrt{1 - 2s}}\right)^k$$

This is minimal for  $s = \epsilon/(2(1 + \epsilon))$ . We get

$$\begin{aligned} \Pr[Y \geq (1 + \epsilon)k] &\leq \left(\frac{e^\epsilon}{1 + \epsilon}\right)^{-k/2} \\ &= \exp\left(-\frac{k}{2}(\epsilon - \log(1 + \epsilon))\right) \\ &\leq \exp\left(-\frac{k}{2}\left(\frac{\epsilon^2}{2} - \frac{\epsilon^3}{3}\right)\right) \\ &\leq \frac{1}{6n^2} \end{aligned}$$

für  $k \geq \frac{2 \log(6n^2)}{\frac{\epsilon^2}{2} - \frac{\epsilon^3}{3}}$ . A similar proof shows that

$$\Pr[Y \leq (1 - \epsilon)k] \leq 1/(6n^2)$$

holds. Hence, the lemma follows.  $\square$

The proof of theorem 13 now follows from the union bound and the fact, that we have at most  $\binom{n}{2} \leq n^2$  pairwise distances between points in  $P$ , i.e.

$$\begin{aligned} &\Pr[\forall p, q \in P : (1 - \epsilon) \cdot D_{\ell_2}(p, q) \leq D_{\ell_2}(\pi(p), \pi(q)) \leq (1 + \epsilon) \cdot D_{\ell_2}(p, q)] \\ &\geq 1 - \sum_{p, q \in P} 1 - (1 - 3/n^2) \\ &\geq 2/3 . \end{aligned}$$

This proves theorem 13, because we have non-zero probability that our random process yields an embedding that satisfies the properties in the theorem. Hence, such an embedding exists.

### 3.3.2 Second Frequency Moments

As already pointed out, we can use the Johnson-Lindenstrauss Theorem directly to obtain a streaming algorithm to approximate the second frequency moment. This algorithm uses space  $O(1/\epsilon^2)$  plus the space required to store the embedding. In our case, we need  $\Omega(d)$  space to store the embedding, which translates to space bigger than the universe in the streaming setting. This is, of course, too large to be of practical use. However, we will see later, that we can define a similar embedding that requires less randomness and still offers the same guarantees as the above embedding. This new embedding can be stored in small space.

## 3.4 Universal Hash Functions

In our algorithms we always assume that the function  $h$  maps an arbitrary fixed element  $x \in \mathcal{U}$  to a uniformly distributed value in  $[m]$ . Therefore, the minimum requirement is to ensure that for any  $y \in [m]$  we have

$$\Pr[h(x) = y] = 1/m .$$

However, it quickly turns out that this requirement is not enough for our purposes. For example, we could define the set to be the set of constant functions, i.e.

$$\mathcal{H} = \{h : \mathcal{U} \rightarrow [m] \mid \exists c \in [m] \forall x \in \mathcal{U} h(x) = c\} .$$

The problem is that in this example each element is mapped to the same value from  $[m]$ , which will not be helpful for our purposes.

We will need a slightly stronger definition.

**Definition 3.4.1** *A set  $\mathcal{H}$  of functions from  $\mathcal{U}$  to  $[m]$  is called a universal (or pairwise independent) class of hash functions, if for every two elements  $x, y \in \mathcal{U}$ ,  $x \neq y$ , and every  $z, r \in [m]$  we have*

- $\Pr[h(x) = z] = 1/m$
- $\Pr[h(x) = z \mid h(y) = r] = 1/m$  .

Thus, the above definition implies that for a hash function that is chosen from a universal class of hash functions, the events  $h(x) = z$  and  $h(y) = r$  are independent for  $x \neq y$ . This can in some cases be used together with Chebyshev's inequality. This is because Chebyshev's inequality only requires us to know the variance of a random variable. If the random variable can be expressed as a sum of pairwise independent random variables, then we can easily compute an upper bound on the variance using the expectation of the random variables. Before we consider such an application in detail, we will discuss a specific construction for universal hash functions.

### 3.4.1 Constructing Universal Hash Functions

A simple way to construct universal hash functions is to assume that  $m = p$  for some prime number  $p$  and to view  $[p]$  together with addition and multiplication modulo  $p$  as the finite field  $\mathbb{F}_p$ . We will choose  $p$  large enough and assume  $\mathcal{U} \subseteq [p]$ . The idea is to pick a random linear function in  $\mathbb{F}_p$ , i.e.  $\mathcal{H}_{univ} = \{h : [p] \rightarrow [p] \mid h(X) = aX + b \pmod p \text{ with } a, b \in [p]\}$ .

Let us verify that  $\mathcal{H}_{univ}$  is a universal class of hash functions. To do so, we first prove that for different choices of parameters the functions from  $\mathcal{H}_{univ}$  are different.

**Lemma 3.4.2** *Let  $a_1, b_1, a_2, b_2 \in [p]$ . Let  $h_1, h_2 : [p] \rightarrow [p]$  be defined as  $h_1(X) := a_1X + b_1 \pmod p$  and  $h_2(X) := a_2X + b_2 \pmod p$ , respectively. Then*

$$h_1 = h_2 \Leftrightarrow a_1 = a_2 \wedge b_1 = b_2$$

**Proof :** Since  $\mathbb{F}_p$  is a field we can write

$$a_1X + b_1 = a_2X + b_2 \pmod p \Leftrightarrow (a_1 - a_2)X = b_2 - b_1 \pmod p .$$

The latter is true for all  $X$ , iff  $a_1 = a_2$  and  $b_1 = b_2$ . □

The lemma implies that choosing a function uniformly at random from  $\mathcal{H}_{univ}$  is equivalent to choosing  $a$  and  $b$  uniformly at random from  $[p]$  and setting  $h(X) = aX + b \pmod p$ . Now it is easy to show that  $\mathcal{H}_{univ}$  is universal.

$$\Pr[h(x) = z] = \Pr[ax + b = z \pmod p] = \Pr[ax = z - b \pmod p] .$$

Fixing the random choice for  $b$  we obtain that  $ax = z - b \pmod p$ , iff  $a = (z - b) \cdot x^{-1} \pmod p$ . Hence, for one out of  $p$  choices we have  $h(x) = z \pmod p$ . Thus,

$$\Pr[h(x) = z] = 1/p .$$

It remains to show that  $\Pr[h(x) = z \mid h(y) = r] = 1/p$ . To do so, we consider

$$\Pr[h(x) = z \wedge h(y) = r] = \Pr[ax + b = z \pmod p \wedge ay + b = r \pmod p] .$$

We have  $ax + b = z \pmod p \Leftrightarrow b = z - ax \pmod p$  and hence  $ay + z - ax = r \pmod p \Leftrightarrow (y - x)a = r - z \pmod p \Leftrightarrow a = (r - z) \cdot (x - y)^{-1} \pmod p$ . Finally,  $b = z - (r - z) \cdot (y - x)^{-1} \pmod p$ . Hence, exactly one choice of  $a$  and  $b$  satisfies  $h(x) = z$  and  $h(y) = r$  simultaneously. This implies that  $\Pr[h(x) = z \wedge h(y) = r] = 1/p^2$ . Therefore,

$$\Pr[h(x) = z \mid h(y) = r] = \frac{\Pr[h(x) = z \wedge h(y) = r]}{\Pr[h(y) = r]} = \frac{1/p^2}{1/p} = 1/p .$$

Hence,

**Lemma 3.4.3**  $\mathcal{H}_{univ}$  is a universal class of hash functions.

Since it suffices to choose  $a$  and  $b$  uniformly at random from  $p$  and consider the function  $aX + b \pmod p$  it is easy to verify that we can store every such function using  $O(\log(p))$  bits and we are able to efficiently evaluate them.

One disadvantage of our approach is that we cannot construct hash functions into arbitrary sets  $[m]$ . This can be usually be circumvented by finding a number  $p$  which is only a small constant time  $m$ . Similar construction are known, if  $\mathbb{F}_q$  is a field with characteristic  $q = p^k$ , for example, if  $q = 2^k$ . For simplicity, we will assume in the following that we can construct a universal hash function from  $\mathcal{U}$  to  $[m]$  for any natural number  $m$  and any finite universe  $\mathcal{U}$ .

## 3.5 Working with Universal Hash Functions

One central question when we want to analyze algorithms that use universal hash functions is which events and which random variables are independent and which are not. Let us consider the class  $\mathcal{H}_{univ}$  described above using  $p = 97$ . Further let us define two events  $A$  and  $B$ .  $A$  denotes the event that a fixed  $x \in \mathcal{U} = [p]$  is mapped to  $\{0, \dots, 9\}$ .  $B$  denotes the event that a

fixed  $y \neq x \in U$  is mapped to the interval  $\{10, \dots, 19\}$ . Are  $A$  and  $B$  pairwise independent? Does it make a difference whether or not the relevant intervals intersect?

To prove that the two random variables are independent, we have to show that

$$\Pr[A \mid B] = \Pr[A] .$$

If  $\Pr[B] \neq 0$  this can be reformulated as

$$\Pr[A \cap B] = \Pr[A] \cdot \Pr[B] .$$

We further have

$$\Pr[A \cap B] = \sum_{i=0}^9 \sum_{j=10}^{19} \Pr[h(x) = i \wedge h(y) = j] = \frac{100}{97^2} .$$

Since  $\mathcal{H}_{univ}$  is universal, we know that  $\Pr[h(x) = i] = 1/97$ . Hence,  $\Pr[A \cap B] = \frac{100}{97^2}$ .

Further, we have

$$\Pr[A] = \sum_{i=0}^{10} \Pr[h(x) = i] = \frac{10}{97}$$

and, similarly,

$$\Pr[B] = \frac{10}{97} .$$

Hence, we have

$$\Pr[A \cap B] = \frac{100}{97^2} = \Pr[A] \cdot \Pr[B] .$$

Thus the events  $A$  and  $B$  are independent. Looking at the proof again one can see that the situation does not change when the two intervals overlap. Thus, even in this case the two events are independent.

### 3.5.1 Independence of Random Variables

Recall that two random variables  $X, Y$  are independent, if all possible pairs of events  $X = x$  and  $Y = y$  are independent. Hence, if we define  $X$  to be the indicator random variable for the event  $A$  and  $Y$  the indicator random variable for the event  $B$ , we obtain that  $X$  and  $Y$  are independent, because we know that

$$\Pr[X = 1 \mid Y = 1] = \Pr[X = 1]$$

and by symmetry

$$\Pr[Y = 1 \mid X = 1] = \Pr[Y = 1] .$$

These inequalities imply that

$$\Pr[X = 0 \mid Y = 1] = 1 - \Pr[X = 1] = \Pr[X = 0]$$

and

$$\Pr[Y = 0 \mid X = 1] = 1 - \Pr[Y = 1] = \Pr[Y = 0] .$$

Since

$$\Pr[X = 1] = \Pr[Y = 0] \cdot \Pr[X = 1 | Y = 0] + \Pr[Y = 1] \cdot \Pr[X = 0 | Y = 1]$$

we have

$$\begin{aligned} \Pr[X = 1 | Y = 0] &= \frac{\Pr[X = 1] - \Pr[Y = 1] \cdot \Pr[X = 1 | Y = 1]}{\Pr[Y = 0]} \\ &= \frac{\Pr[X = 1] - \Pr[Y = 1] \cdot \Pr[X = 1]}{\Pr[Y = 0]} \\ &= \frac{\Pr[X = 1] - \Pr[Y = 1] \cdot \Pr[X = 1 | Y = 1]}{1 - \Pr[Y = 1]} \\ &= \Pr[X = 1] . \end{aligned}$$

In a similar way we can prove  $\Pr[Y = 1 | X = 0] = \Pr[Y = 1]$ . Finally, we can apply the above technique to show  $\Pr[X = 0 | Y = 0] = \Pr[X = 0]$  and  $\Pr[X = 1 | Y = 0] = \Pr[X = 1]$ . Hence, indicator random variables are independent, if the corresponding events are independent.

Another way to prove that the random variables are independent for the case of uniform hashing is to follow a proof similar to that showing that events A and B are independent. (In)dependence of random variables is a big source for errors and recognizing dependencies may be very difficult. Such an elementary proof offers a safe way to verify the assumption of (pairwise) independence.

### 3.5.2 When Universal Hash Functions are not Sufficient.

In the algorithm for random sampling in dynamic data streams we used hash functions to make sure that with constant probability a single element of the current set maps to the value 0. Can we show that for this purpose universal hash functions suffice? Let us denote by C the event that exactly one element is mapped to 1. We have

$$\begin{aligned} \Pr[C] &= \sum_{x \in \mathcal{U}} \Pr[h(x) = 1 \text{ and } \bigwedge_{y \in \mathcal{U} \setminus \{x\}} h(y) \neq 1] \\ &= \sum_{x \in \mathcal{U}} \Pr[h(x) = 1 \text{ and } \bigwedge_{y \in \mathcal{U} \setminus \{x\}} \bigvee_{i=1}^{m-1} h(y) = i] \\ &= \sum_{x \in \mathcal{U}} \Pr[h(x) = 1 \text{ and } \bigvee_{i_y \in \{1, \dots, m-1\}} \bigwedge_{y \in \mathcal{U} \setminus \{x\}} h(y) = i_y] \\ &= \sum_{x \in \mathcal{U}} \sum_{i_y \in \{1, \dots, m-1\}} \Pr[h(x) = 1 \text{ and } \bigwedge_{y \in \mathcal{U} \setminus \{x\}} h(y) = i_y] \end{aligned}$$

Although we have reduced the original event to a sum of simpler events we still have the problem that the event  $h(x) = 1$  and  $\bigwedge_{y \in \mathcal{U} \setminus \{x\}} h(y) = i_y$  depends on all values of  $h(x')$  for  $x' \in \mathcal{U}$  rather than on two of them. This indicates that the random variables may be not independent. Technically, we have the problem that we cannot apply the definition of universal hash functions since it involves only the outcome of two values from  $\mathcal{U}$ .

## 3.6 An Improved Algorithm for the Distinct Elements Problem

We will now revisit the distinct elements problem from Chapter 1. Recall that the number of distinct items in a stream  $\sigma_1, \dots, \sigma_n$  is denoted as  $F_0$  and that the basic idea of the algorithm from Chapter 1 is to map the frequency vector of the stream to the interval  $[0, 1]$ . We will now replace the function  $h$  with a universal hash function that maps the universe to some discrete interval  $[0, m]$ , where  $m = |\mathcal{U}|^3$ .

In comparison to our first algorithm we will also add another improvement that will allow us to obtain a  $(1 + \epsilon)$ -approximation of the number of distinct elements.

### 3.6.1 New Idea

We use  $D = \{v_1, \dots, v_{F_0}\}$  to denote the set of distinct items in the stream. If we look at the  $t$ -th smallest values obtained by applying  $h$  to the set  $D$ , we expect that these values are approximately mapped into  $\{0, \dots, \lceil tm/F_0 \rceil - 1\}$ . If we use  $\kappa$  to denote the  $t$ -th smallest hash value then typically  $\kappa \approx tm/F_0$ , which implies  $F_0 \approx tm/\kappa$ . Therefore, our new algorithm stores the  $t = \lceil 96/\epsilon^2 \rceil$  smallest hash values and uses  $\tilde{F}_0 = tm/\kappa$  as an approximation, where  $\kappa$  is the position of the  $t$ -th smallest element in  $h(D)$ .

### 3.6.2 The Algorithm

A description of the algorithm is given below. It uses a balanced binary search tree to maintain the  $t$  smallest hash values. Besides insert and delete and search the tree is supposed to support the operations `max` and `sizeof`, which return the maximum values stored in the tree and the number of values stored in the tree, respectively.

IMPROVEDDISTINCTELEMENTS( $\sigma, \epsilon$ )

1. Choose function  $h : [\mathcal{U}] \rightarrow [m]$ ,  $m = |\mathcal{U}|^3$ , uniformly at random from universal class of hash functions
2.  $t = \lceil 96/\epsilon^2 \rceil$
3. Erzeuge leeren Suchbaum  $T$
4. **for each** item  $\sigma_i$  in the input stream **do**
5.     **if**  $h(\sigma_i) \notin T$  **then**
6.         **if** `sizeof(T) < t` **then** `insert(T, h( $\sigma_i$ ))`
7.         **if** `max(T) > h( $\sigma_i$ )` **then** `insert(T, h( $\sigma_i$ )); delete(T, max(T))`
7.     **output**  $\tilde{F}_0 = mt/\max(T)$

### 3.6.3 Analysis

Our first step is to prove that with probability at least  $1 - 1/m$  the hash functions  $h$  maps no two elements from  $D$  to the same values. Since  $h$  is from a universal class of hash functions we have for fixed  $x, y \in D \subseteq \mathcal{U}$

$$\Pr[h(x) = h(y)] = \sum_{i=1}^m \Pr[h(x) = i \wedge h(y) = i] = 1/m .$$

Let  $X_{xy}$  denote the indicator random variable for the event that  $x$  and  $y$  are mapped to the same location. It follows immediately that

$$\mathbf{E}[X_{xy}] = 1/m .$$

The overall expected number of collisions is given by

$$\mathbf{E}\left[\sum_{\{x,y\}\subseteq D} X_{xy}\right] = \sum_{\{x,y\}\subseteq D} \mathbf{E}[X_{xy}] \leq |D|^2/m \leq |U|^2/m = 1/|U| .$$

However, a collision occurs only if  $\sum_{\{x,y\}\subseteq D} X_{xy} \geq 1$ . By Markov inequality we have

$$\Pr\left[\sum_{\{x,y\}\subseteq D} X_{xy} \geq |U| \cdot \mathbf{E}\left[\sum_{\{x,y\}\subseteq D} X_{xy}\right]\right] \leq \frac{\mathbf{E}\left[\sum_{\{x,y\}\subseteq D} X_{xy}\right]}{|U| \cdot \mathbf{E}\left[\sum_{\{x,y\}\subseteq D} X_{xy}\right]} = \frac{1}{|U|} .$$

Hence, with probability  $1 - 1/|U|$  no collision occurs.

Our goal is to ensure that the output value of the algorithm satisfies

$$(1 - \epsilon) \cdot F_0 \leq \tilde{F}_0 \leq (1 + \epsilon) \cdot F_0 .$$

Using  $\tilde{F}_0 = mt/\kappa$  this translates into

$$\frac{1}{1 + \epsilon} \cdot \frac{mt}{F_0} \leq \kappa \leq \frac{1}{1 - \epsilon} \cdot \frac{mt}{F_0} .$$

The latter inequality follows from

$$(1 - \epsilon/2) \cdot \frac{mt}{F_0} \leq \kappa \leq (1 + \epsilon) \cdot \frac{mt}{F_0} .$$

Now we observe that if the first inequality is not satisfied then there must be at least  $t$  values from  $h(D)$  that are mapped to  $\{0, \dots, \lfloor (1 - \epsilon/2) \cdot \frac{mt}{F_0} \rfloor - 1\}$ . The probability that a single value  $x \in D$  is mapped to  $\{0, \dots, \lfloor (1 - \epsilon/2) \cdot \frac{mt}{F_0} \rfloor - 1\}$  is at most  $(1 - \epsilon/2) \cdot \frac{mt}{F_0} / m = (1 - \epsilon/2) \cdot \frac{t}{F_0}$ . If  $t \geq F_0$  then all distinct elements will be stored in our tree provided they hash to different numbers (which happens with probability at least  $1 - 1/|U|$ ). Therefore, we can assume  $t < F_0$  in the remainder of the proof. For each  $v_1, \dots, v_{F_0}$  let  $X_i$  denote the indicator random variable for the event that  $v_i$  is mapped to  $\{0, \dots, \lfloor (1 - \epsilon/2) \cdot \frac{mt}{F_0} \rfloor - 1\}$ . We have  $\mathbf{E}[X_i] \leq (1 - \epsilon/2) \cdot \frac{t}{F_0}$  and hence

$$\mathbf{E}\left[\sum_{i=1}^{F_0} X_i\right] = \sum_{i=1}^{F_0} \mathbf{E}[X_i] \leq F_0 \cdot (1 - \epsilon/2) \cdot \frac{t}{F_0} = (1 - \epsilon/2)t$$

We also have  $\mathbf{Var}[X_i] \leq \mathbf{E}[X_i]$  since  $X_i$  is a 0-1-random variable. By pairwise independence of the  $X_i$  we obtain

$$\mathbf{Var}\left[\sum_{i=1}^{F_0} X_i\right] = \sum_{i=1}^{F_0} \mathbf{Var}[X_i] \leq \sum_{i=1}^{F_0} \mathbf{E}[X_i] \leq (1 - \epsilon/2)t .$$

Chebyshev's inequality tells us that

$$\begin{aligned}
\Pr\left[\sum_{i=1}^{F_0} X_i \geq t\right] &\leq \Pr\left[\sum_{i=1}^{F_0} X_i \geq \frac{\epsilon t}{2} + \mathbf{E}\left[\sum_{i=1}^{F_0} X_i\right]\right] \\
&\leq \Pr\left[\left|\sum_{i=1}^{F_0} X_i - \mathbf{E}\left[\sum_{i=1}^{F_0} X_i\right]\right| \geq \epsilon t/2\right] \\
&\leq \frac{\mathbf{Var}\left[\sum_{i=1}^{F_0} X_i\right]}{(\epsilon t/2)^2} \\
&\leq \frac{4}{\epsilon^2 t} \leq 1/16 .
\end{aligned}$$

for our choice of  $t$ . Hence, with probability  $1 - 1/16 - 1/|\mathcal{U}|$  the first inequality is satisfied.

The approach to the second inequality is similar. We give it below for sake of completeness.

The second inequality ( $\kappa \leq (1 + \epsilon) \cdot \frac{mt}{F_0}$ ) is satisfied if at most  $t$  values from  $h(D)$  are mapped to  $\{0, \dots, \lceil(1 + \epsilon) \cdot \frac{mt}{F_0}\rceil - 1\}$  (assuming that no collision occur). In this case, the probability that a single value  $x \in D$  is mapped to  $\{0, \dots, \lceil(1 + \epsilon) \cdot \frac{mt}{F_0}\rceil - 1\}$  is at least  $(1 + \epsilon) \cdot \frac{t}{F_0}$ . Again, if  $t \geq F_0$  then all distinct elements will be stored in our tree provided they hash to different numbers (which happens with probability at least  $1 - 1/|\mathcal{U}|$ ). Therefore, we can assume  $t < F_0$  in the remainder of the proof.

For each  $v_1, \dots, v_{F_0}$  let  $X_i$  denote the indicator random variable for the event that  $v_i$  is mapped to  $\{0, \dots, \lceil(1 + \epsilon) \cdot \frac{mt}{F_0}\rceil - 1\}$ . We have  $\frac{2t}{F_0} \geq \mathbf{E}[X_i] \geq (1 + \epsilon) \cdot \frac{t}{F_0}$  and hence

$$\mathbf{E}\left[\sum_{i=1}^{F_0} X_i\right] = \sum_{i=1}^{F_0} \mathbf{E}[X_i] \geq F_0 \cdot (1 + \epsilon) \cdot \frac{t}{F_0} = (1 + \epsilon)t$$

We also have  $\mathbf{Var}[X_i] \leq \mathbf{E}[X_i]$  since  $X_i$  is a 0-1-random variable. By pairwise independence of the  $X_i$  we obtain

$$\mathbf{Var}\left[\sum_{i=1}^{F_0} X_i\right] = \sum_{i=1}^{F_0} \mathbf{Var}[X_i] \leq \sum_{i=1}^{F_0} \mathbf{E}[X_i] \leq 2t .$$

We obtain

$$\begin{aligned}
\Pr\left[\sum_{i=1}^{F_0} X_i \leq t\right] &\leq \Pr\left[\sum_{i=1}^{F_0} X_i \leq \mathbf{E}\left[\sum_{i=1}^{F_0} X_i\right] - \epsilon t\right] \\
&\leq \Pr\left[\left|\sum_{i=1}^{F_0} X_i - \mathbf{E}\left[\sum_{i=1}^{F_0} X_i\right]\right| \geq \frac{\epsilon t}{2}\right] \\
&\leq \frac{\mathbf{Var}\left[\sum_{i=1}^{F_0} X_i\right]}{(\epsilon t/2)^2} \\
&\leq \frac{8}{\epsilon^2 t} \leq 1/8 .
\end{aligned}$$

for our choice of  $t$ . Hence, with probability  $1 - 1/8 - 1/|\mathcal{U}|$  the second inequality is satisfied.

We summarize our analysis.

**Theorem 14** Algorithm IMPROVEDDISTINCTELEMENTS computes an estimator  $\tilde{F}_0$  for the number of distinct elements in a data stream such that with probability at least  $3/4$  using  $O(1/\epsilon^2)$  memory cells and with update time  $O(\log(1/\epsilon))$ .

**Proof :** If  $|\mathcal{U}| < 16$  then we count the number of distinct elements exactly. Otherwise, the theorem follows from our analysis and the fact that insert, delete and search can be performed in  $O(\log n)$  time in a balanced search tree of  $n$  elements. Also, the maximum element can be computed in constant time.  $\square$

## 3.7 Classes of $k$ -wise Independent Hash Functions

While for some applications universal hash functions suffice, for others we will need stronger properties of randomness. We extend the notion of universal (pairwise independent) classes of hash functions to  $k$ -wise independent classes.

**Definition 3.7.1** A class of functions  $\mathcal{H}$  from  $\mathcal{U}$  to  $[m]$  is called  $k$ -wise independent, if for every set of elements  $x, x_1, x_2, \dots, x_\ell \in \mathcal{U}$ ,  $\ell \leq k$ ,  $x_i \neq x_j$  for  $i \neq j$ , and every  $r, r_1, r_2, \dots, r_\ell \in [m]$  we have

- $\Pr[h(x) = r] = 1/m$
- $\Pr[\bigcap_{i=1}^{\ell} h(x_i) = r_i] = \frac{1}{m^\ell}$  .

The definition of classes of  $k$ -wise independent hash functions allows us to smoothly move from universal hashing to ideal hashing by 'adding more and more randomness'. We will see that for some applications in streaming  $k$ -wise independent hash functions with  $k$  being a constant greater than 2 are required.

### 3.7.1 Construction of a Class of $k$ -wise Independent Hash Functions

We use a construction which is a straightforward generalization of universal hashing. We again consider a finite field  $\mathbb{F}_p$  for  $p \geq k$  and consider  $\mathcal{U}$  as a subset of  $\mathbb{F}_p$ . Then we define  $\mathcal{H}_k$  to be the set of all polynomials from  $\mathbb{F}_p$  to  $\mathbb{F}_p$  of degree at most  $k$ , i.e.  $\mathcal{H}_k = \{h_a : [p] \rightarrow [p] \mid h_a(X) = \sum_{i=0}^{k-1} a_i X^i, a_i \in [p]\}$ . To select a random element in  $\mathcal{H}_k$  we proceed similar as in the case of universal hashing and pick the elements  $a_i$  uniformly at random from  $\mathbb{F}_p$ .

Now we prove that  $\mathcal{H}_k$  is  $k$ -wise independent.

$$\Pr[h_a(x) = r] = \Pr\left[\sum_{i=0}^{k-1} a_i x^i = z \pmod{p}\right] = \Pr\left[a_0 = z - \sum_{i=1}^{k-1} a_i x^i \pmod{p}\right] .$$

Fixing the random choice for  $a_1, \dots, a_{k-1}$  we obtain that  $z - \sum_{i=1}^{k-1} a_i x^i$  is a constant  $c \in \mathbb{F}_p$ . Hence,  $\sum_{i=0}^{k-1} a_i x^i = z \pmod{p}$ , iff  $a_0 = c$ . Obviously, this happens with probability  $1/p$ . Thus,

$$\Pr[h(x) = z] = 1/p .$$

It remains to show that for every set of elements  $x_1, x_2, \dots, x_\ell \in \mathcal{U}$ ,  $\ell \leq k$ ,  $x_i \neq x_j$  for  $i \neq j$ , and every  $r_1, r_2, \dots, r_\ell \in [p]$  we have  $\Pr[\bigcap_{i=1}^{\ell} h_a(x_i) = r_i] = \frac{1}{p^\ell}$ . To prove this, we have to consider the  $a_i$  as variables in the following system of equations. We will first assume that  $\ell = k$ . Since the  $x_i$  are fixed, this is a system of linear equations.

$$\begin{aligned} a_0 + a_1x_1 + a_2x_1^2 + \dots + a_{k-1}x_1^{k-1} &= r_1 \\ &\vdots \\ a_0 + a_1x_k + a_2x_k^2 + \dots + a_{k-1}x_k^{k-1} &= r_k \end{aligned}$$

This system has a unique solution as can be seen when we rewrite it in the matrix form.

$$\begin{pmatrix} 1 & x_1 & x_1^2 & \dots & x_1^{k-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{k-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_k & x_k^2 & \dots & x_k^{k-1} \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{k-1} \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_k \end{pmatrix}.$$

The matrix above is a Van der Monde matrix and has full rank since  $x_i \neq x_j$  for  $i \neq j$ . Hence there is only a single solution to this system. It follows that

$$\Pr\left[\bigcap_{i=1}^k h_a(x_i) = r_i\right] = \frac{1}{p^k}.$$

It remains to deal with the case  $k < \ell$ . This can be done by fixing the random choices of  $a_{k-1}, \dots, a_\ell$ . In a similar way as above we obtain a system of linear equations of rank  $\ell$  that has a unique solution. Hence,

$$\Pr\left[\bigcap_{i=1}^{\ell} h_a(x_i) = r_i\right] = p^{k-\ell} \cdot \frac{1}{p^k} = \frac{1}{p^\ell}.$$

Hence,

**Lemma 3.7.2**  $\mathcal{H}_k$  is a  $k$ -wise independent class of hash functions.

### 3.7.2 $k$ -wise Independent Random Variables

Recall that a set of events  $A_0, \dots, A_{n-1}$  is called  $k$ -wise independent, if for every subset  $I \subseteq [n]$ ,  $|I| \leq k$ , we have

$$\Pr\left[\bigcap_{i \in I} A_i\right] = \prod_{i \in I} \Pr[A_i].$$

A set of random variables  $X_0, \dots, X_{n-1}$  is called  $k$ -wise independent, if for every subset  $I \subseteq [n]$ ,  $|I| \leq k$ , and for every  $r_1, \dots, r_k \in \mathbb{R}$  we have

$$\Pr\left[\bigcap_{i \in I} X_i = r_i\right] = \prod_{i \in I} \Pr[X_i = r_i].$$

Let us consider a class  $\mathcal{H}$  of  $k$ -wise independent hash functions from  $\mathcal{U}$  to  $[p]$ . Let  $h$  be chosen uniformly at random from  $\mathcal{H}$ . Then we can view  $h(x)$  as a random variable. It follows immediately that the  $h(x)$  are  $k$ -wise independent.

We first show the following lemma, which will be useful in many situations.

**Lemma 3.7.3** *Let  $X_0, \dots, X_{n-1}$  be a set of  $2k$ -wise independent random variables. Let  $Y_i = X_{i_1} \cdot X_{i_2} \cdots X_{i_k}$ ,  $1 \leq i \leq m$  be a set of monomials of degree  $k$  such that no  $X_j$  appears in two distinct  $Y_i$  and no  $X_j$  appears more than once in any  $Y_i$ . Then the  $Y_i$  are pairwise independent.*

**Proof :** Wlog. we prove that  $Y_1$  and  $Y_2$  are pairwise independent. We have to show that  $\Pr[Y_1 = r \cap Y_2 = z] = \Pr[Y_1 = r] \cdot \Pr[Y_2 = z]$  for all  $r, z \in \mathbb{R}$ . We have

$$\Pr[Y_i = r \cap Y_j = z] = \Pr\left[\prod_{i=1}^k X_{1,i} = r \cap \prod_{i=1}^k X_{2,i} = z\right].$$

Now let us consider the sets  $R = \{(r_1, \dots, r_k) \mid \prod_i r_i = r\}$  and  $Z = \{(z_1, \dots, z_k) \mid \prod_i z_i = z\}$ . We get

$$\begin{aligned} \Pr\left[\prod_{i=1}^k X_{1,i} = r \cap \prod_{i=1}^k X_{2,i} = z\right] &= \Pr\left[\left(\bigcup_{r \in R} \bigcap_{i=1}^k X_{1,i} = r_i\right) \cap \left(\bigcup_{z \in Z} \bigcap_{i=1}^k X_{2,i} = z_i\right)\right] \\ &= \Pr\left[\bigcup_{r \in R, z \in Z} \bigcap_{i=1}^k (X_{1,i} = r_i \cap X_{2,i} = z_i)\right] \\ &= \sum_{r \in R, z \in Z} \Pr\left[\bigcap_{i=1}^k (X_{1,i} = r_i \cap X_{2,i} = z_i)\right] \end{aligned}$$

Now we can apply that our random variables are  $2k$ -wise independent and obtain

$$\begin{aligned} \sum_{r \in R, z \in Z} \Pr\left[\bigcap_{i=1}^k (X_{1,i} = r_i \cap X_{2,i} = z_i)\right] &= \sum_{r \in R, z \in Z} \prod_{i=1}^k \Pr[X_{1,i} = r_i] \cdot \Pr[X_{2,i} = z_i] \\ &= \left(\sum_{r \in R} \prod_{i=1}^k \Pr[X_{1,i} = r_i]\right) \cdot \left(\sum_{z \in Z} \prod_{i=1}^k \Pr[X_{2,i} = z_i]\right) \\ &= \Pr\left[\prod_{i=1}^k X_{1,i} = r\right] \cdot \Pr\left[\prod_{i=1}^k X_{2,i} = z\right] \\ &= \Pr[Y_1 = r] \cdot \Pr[Y_2 = z] \end{aligned}$$

□

The above lemma can be used to obtain the following useful corollary.

**Corollary 3.7.4** *Let  $X_0, \dots, X_{n-1}$  be  $2k$ -wise independent random variables. Let  $S = \sum_i Y_i$  with  $Y_i = X_{i_1} \cdot X_{i_2} \cdots X_{i_k}$  with  $i_j \in [n]$  and such that no random variable appears twice in the  $Y_i$ . Then  $\mathbf{Var}[S] \leq \sum_i \mathbf{Var}[Y_i]$ .*

**Proof :** Follows immediately from the fact that the  $Y_i$  are pairwise independent.  $\square$

For example, we might want to apply Chebyshev's inequality to a random variable that can be expressed as a sum of degree  $k$  monomials of basic non negative random variables. If we can make sure that the basic random variables are  $(2k)$ -wise independent, then we can again obtain an upper bound on the variance using the expectation of the basic random variables.

## 3.8 Estimating the Frequency Moments

In this section we discuss how to approximate the frequency moments  $F_k$  for  $k \geq 2$ . We have already seen that we can approximate  $F_2$  using the Johnson-Lindenstrauss Theorem, if we ignore the space required to store the random matrix  $A$  that defines the random projection. Now we will see how this can be achieved using 4-wise independent hash functions. In the following we will consider universe  $U = \{1, \dots, n\}$  and we use  $f_i$  to denote the number of occurrences of element  $i \in U$  in the data stream.

### 3.8.1 Estimating $F_2$

The basis of our algorithm is a procedure that computes a value  $X$  whose expected value is  $F_2$  and whose variance is relatively small. In order to achieve a good concentration we run this procedure  $s_1 = 16/\epsilon^2$  number of times and compute the average value. This way, we achieve that with constant probability our algorithm computes a  $(1 \pm \epsilon)$ -approximation of  $F_2$ . To boost the confidence probability to  $1 - \delta$  we then invoke  $s_2 = 96 \log(1/\delta)$  instances of the latter algorithm and take the median of the output values. More details will be given at the end of the analysis.

We first define our procedure to compute  $X$ . We sample a function  $h$  from a class of 4-wise independent hash functions from  $\{1, \dots, n\}$  to  $\{-1, 1\}$ . To define  $X$  we first need to define a random variable  $Z$  as follows

$$Z = \sum_{i=1}^n h(i) \cdot f_i .$$

Note that we can maintain  $Z$  easily. If an element  $x$  arrives we simply have to perform the assignment  $Z = Z + h(x)$ . Finally, we define  $X = Z^2$ .

SECONDFREQUENCYMOMENTS( $s_1, s_2$ )

1. **for**  $i = 1$  **to**  $s_2$  **do**
2.     **for**  $j = 1$  **to**  $s_1$  **do**
3.          $Z_{i,j} = 0$
4.     **while not** EOF **do**
5.         Let  $x$  be the next item from the stream
6.         **for**  $i = 1$  **to**  $s_2$  **do**
7.             **for**  $j = 1$  **to**  $s_1$  **do**
8.                  $Z_{i,j} = Z_{i,j} + h(x)$
9.      $Y_i = \frac{1}{s_1} \cdot \sum_{j=1}^{s_1} (Z_{i,j})^2$
10. **return** median( $\bigcup_{i=1}^{s_2} Y_i$ )

Now we proceed with the analysis. Our first step will be to show that  $X$  is an unbiased estimator for  $F_2$ , i.e.  $\mathbf{E}[X] = F_2$ . We use the fact that the  $h(i)$  are pairwise independent and have  $\mathbf{E}[h(i)] = 0$  for all  $i$ . We get

$$\begin{aligned}
 \mathbf{E}[X] &= \mathbf{E}\left[\left(\sum_{i=1}^n h(i) \cdot f_i\right)^2\right] \\
 &= \sum_{i=1}^n f_i^2 \cdot \mathbf{E}[h(i)^2] + 2 \cdot \sum_{1 \leq i < j \leq n} f_i \cdot f_j \cdot \mathbf{E}[h(i) \cdot h(j)] \\
 &= \sum_{i=1}^n f_i^2 \cdot \mathbf{E}[h(i)^2] + 2 \cdot \sum_{1 \leq i < j \leq n} f_i \cdot f_j \cdot \mathbf{E}[h(i)] \cdot \mathbf{E}[h(j)] \\
 &= \sum_{i=1}^n f_i^2 = F_2
 \end{aligned}$$

To compute the variance of  $X$  we also have to compute  $\mathbf{E}[X^2]$ .

$$\begin{aligned}
\mathbf{E}[X^2] &= \mathbf{E}\left[\left(\sum_{i=1}^n h(i) \cdot f_i\right)^2\right] \\
&= \sum_{i=1}^n f_i^4 \cdot \mathbf{E}[h(i)^2] + \binom{4}{2} \cdot \sum_{1 \leq i < j \leq n} f_i^2 \cdot f_j^2 \cdot \mathbf{E}[h(i)^2 \cdot h(j)^2] \\
&\quad + \binom{4}{3} \cdot \sum_{\substack{1 \leq i, j \leq n; \\ i \neq j}} f_i^3 \cdot f_j \cdot \mathbf{E}[h(i)^3 \cdot h(j)] \\
&\quad + \binom{4}{2} \cdot 2! \cdot \sum_{\substack{1 \leq i \leq n; \\ 1 \leq j < k \leq n; \\ i \neq j; i \neq k}} f_i^2 \cdot f_j \cdot f_k \cdot \mathbf{E}[h(i)^2 \cdot h(j) \cdot h(k)] \\
&\quad + 24 \cdot \sum_{1 \leq i < j < k < \ell \leq n} f_i \cdot f_j \cdot f_k \cdot f_\ell \cdot \mathbf{E}[h(i) \cdot h(j) \cdot h(k) \cdot h(\ell)] \\
&= \sum_{i=1}^n f_i^4 \cdot \mathbf{E}[h(i)^2] + 6 \cdot \sum_{1 \leq i < j \leq n} f_i^2 \cdot f_j^2 \cdot \mathbf{E}[h(i)^2] \cdot \mathbf{E}[h(j)^2] \\
&\quad + 4 \cdot \sum_{\substack{1 \leq i, j \leq n; \\ i \neq j}} f_i^3 \cdot f_j \cdot \mathbf{E}[h(i)^3] \cdot \mathbf{E}[h(j)] \\
&\quad + 12 \cdot \sum_{\substack{1 \leq i \leq n; \\ 1 \leq j < k \leq n; \\ i \neq j; i \neq k}} f_i^2 \cdot f_j \cdot f_k \cdot \mathbf{E}[h(i)^2] \cdot \mathbf{E}[h(j)] \cdot \mathbf{E}[h(k)] \\
&\quad + 4! \cdot \sum_{1 \leq i < j < k < \ell \leq n} f_i \cdot f_j \cdot f_k \cdot f_\ell \cdot \mathbf{E}[h(i)] \cdot \mathbf{E}[h(j)] \cdot \mathbf{E}[h(k)] \cdot \mathbf{E}[h(\ell)] \\
&= \sum_{i=1}^n f_i^4 + 6 \cdot \sum_{1 \leq i < j \leq n} f_i^2 \cdot f_j^2
\end{aligned}$$

We obtain that

$$\mathbf{Var}[X] = \mathbf{E}[X^2] - (\mathbf{E}[X])^2 = 4 \cdot \sum_{1 \leq i < j \leq n} f_i^2 \cdot f_j^2 \leq 2F_2^2 .$$

At this point we have defined a random variable  $X$  with expected value  $F_2$  and (relatively) small variance. We have seen that  $X$  can be computed easily in the streaming scenario. We only have to maintain a single counter, which requires  $O(\log m)$  bits of memory plus a 4-wise independent hash function which requires  $O(\log n)$  bits of memory. It remains to obtain a sharply concentrated estimator by repeating this approach and apply Chebyshev's inequality and then boost the confidence probability by computing the concentrated estimator several times in parallel and taking the median of the computations as a result.

Formally, we perform the above experiment  $s_1 \cdot s_2$  times to compute the random variable  $X_{i,j}$ , where  $1 \leq i \leq s_2$  and  $1 \leq j \leq s_1$ . We set  $Y_i = \frac{1}{s_1} \cdot \sum_{j=1}^{s_1} X_{i,j}$  to be the average value of

the  $X_{i,j}$ . Linearity of expectation implies that

$$\mathbf{E}[Y_i] = \mathbf{E}\left[\frac{1}{s_1} \cdot \sum_{j=1}^{s_1} X_{i,j}\right] = \frac{1}{s_1} \cdot \sum_{j=1}^{s_1} \mathbf{E}[X_{i,j}] = F_2 .$$

We also have

$$\mathbf{Var}\left[\frac{1}{s_1} \sum_{i=1}^{s_1} Y_i\right] = \frac{1}{s_1^2} \cdot \mathbf{Var}\left[\sum_{i=1}^{s_1} Y_i\right] = \frac{1}{s_1^2} \cdot s_1 \cdot \mathbf{Var}[Y_1] = \frac{F_2^2}{s_1}$$

by independence of the  $Y_i$ . By Chebyshev's inequality we get for each fixed  $i$

$$\mathbf{Pr}[|Y_i - F_2| \geq \epsilon F_2] = \mathbf{Pr}[|Y_i - \mathbf{E}[Y_i]| \geq \epsilon F_2] \leq \frac{\mathbf{Var}[Y_i]}{s_1 \cdot \epsilon^2 F_2^2} \leq \frac{2F_2^2}{\epsilon^2 \cdot s_1 \cdot F_2^2} = 1/8 .$$

Thus, with probability at least  $7/8$  we have for every fixed  $i$

$$(1 - \epsilon) \cdot F_2^2 \leq Y_i \leq (1 + \epsilon) \cdot F_2^2 .$$

Finally, we return the median of the  $Y_i$ 's. We will use Chernoff bounds to show that the probability that more than  $s_2/2$  of the  $Y_i$  do not satisfy the above inequality is very small. But if at least  $s_2/2$  satisfy the above inequality, then so does the median of them and hence our output is within the desired approximation bounds. For the analysis let  $E_i$  be the indicator random variable for the event that  $(1 - \epsilon) \cdot F_2^2 \leq Y_i \leq (1 + \epsilon) \cdot F_2^2$  does not hold. By our above discussion we have  $\mathbf{Pr}[E_i = 1] \leq 1/8$ . For our analysis we will assume  $\mathbf{Pr}[E_i = 1] = 1/8$ . We are interested whether  $\sum_{i=1}^{s_2} E_i > s_2/2$ . We get

$$\begin{aligned} \mathbf{Pr}\left[\sum_{i=1}^{s_2} s_2 E_i > s_2/2\right] &\leq \mathbf{Pr}\left[\sum_{i=1}^{s_2} E_i > \left(1 + \frac{1}{2}\right) \cdot \mathbf{E}\left[\sum_{i=1}^{s_2} E_i\right]\right] \\ &\leq e^{-\frac{s_2}{96}} \\ &\leq \delta \end{aligned}$$

using Chernoff bounds.

We can summarize our result in the following theorem.

**Theorem 15** *For every  $\epsilon > 0$  and every  $\delta > 0$  there exists a streaming algorithm that given a stream of  $m$  elements from  $\mathcal{U} = \{1, \dots, n\}$  computes a value  $\tilde{F}_2^2$  such that  $(1 - \epsilon) \cdot F_2^2 \leq \tilde{F}_2^2 \leq (1 + \epsilon) \cdot F_2^2$ . The algorithm uses*

$$O\left(\frac{\log(1/\delta)}{\epsilon^2} \cdot (\log n + \log m)\right)$$

*bits of memory. Each element in the sequence can be processed in  $O(\log(1/\delta)/\epsilon^2)$  time.*

### 3.9 Higher Frequency Moments

Now we will develop an algorithm for the estimation of higher frequency moments. We consider the same situation as for  $F_2$ . We are given a stream of items from a universe  $U = \{1, \dots, n\}$  and we let  $f_i$  denote the number of occurrences of item  $i$  in the stream. We are interested in approximating  $F_k = \sum_{i=1}^n f_i^k$  upto a factor of  $(1 \pm \epsilon)$ .

The main idea behind the streaming algorithm is to use sufficiently weighted random sampling. Let  $A$  denote the set of all items in the stream and  $m := F_1$  denote the number of items in  $A$ . Next let us take a random element  $x$  from  $A$ . For now we assume that we can take the random sample and compute the value  $X = m \cdot f_x^{k-1}$  in a single pass. This will be the output value of our algorithm. We will later see, how to circumvent this problem using a modification of our algorithm.

Since the probability to take an element  $x$  is  $f_x/n$  we obtain

$$\mathbf{E}[X] = \sum_{i=1}^n \frac{f_i}{F_1} \cdot F_1 \cdot f_i^{k-1} = F_k .$$

Our next step is to compute an upper bound on the variance of  $X$ . To obtain such a bound we compute  $\mathbf{E}[X^2]$ , which is an upper bound on  $\mathbf{Var}[X]$ .

$$\mathbf{Var}[X] \leq \mathbf{E}[X^2] = \sum_{i=1}^n \frac{f_i}{F_1} \cdot F_1^2 \cdot f_i^{2k-2} = F_1 \cdot F_{2k-1} .$$

In order to achieve a sharp concentration we repeat our experiment  $s_1$  times computing  $s_1$  random variables  $X_i$ ,  $1 \leq i \leq s_1$ . Our output value will be  $\frac{1}{s_1} \cdot \sum_{i=1}^{s_1} X_i$ . It is easy to verify that  $\mathbf{E}[\frac{1}{s_1} \cdot \sum_{i=1}^{s_1} X_i] = F_k$ . Since the random variables are independent we obtain

$$\mathbf{Var}\left[\frac{1}{s_1} \cdot \sum_{i=1}^{s_1} X_i\right] = \frac{1}{s_1^2} \sum_{i=1}^{s_1} \mathbf{Var}[X_i] \leq \frac{F_1 \cdot F_{2k-1}}{s_1}$$

Then we can apply Chebyshev's inequality and obtain

$$\begin{aligned} \Pr\left[\left|\frac{1}{s_1} \cdot \sum_{i=1}^{s_1} X_i - \mathbf{E}\left[\frac{1}{s_1} \cdot \sum_{i=1}^{s_1} X_i\right]\right| \leq \epsilon \cdot \mathbf{E}\left[\frac{1}{s_1} \cdot \sum_{i=1}^{s_1} X_i\right]\right] &\leq \frac{\mathbf{Var}\left[\frac{1}{s_1} \cdot \sum_{i=1}^{s_1} X_i\right]}{(\epsilon \cdot \mathbf{E}\left[\frac{1}{s_1} \cdot \sum_{i=1}^{s_1} X_i\right])^2} \\ &\leq \frac{F_1 \cdot F_{2k-1}}{\epsilon^2 \cdot s_1 \cdot F_k^2} \end{aligned}$$

We now need the following inequality.

#### Claim 3.9.1

$$F_1 \cdot \left(\sum_{i=1}^n f_i^{2k-1}\right) = \left(\sum_{i=1}^n f_i\right) \cdot \left(\sum_{i=1}^n f_i^{2k-1}\right) \leq m^{1-1/k} \cdot \left(\sum_{i=1}^n f_i^k\right)^2 .$$

**Proof :** We use  $f_{max} = \max_{1 \leq i \leq n} f_i$ . Then

$$\begin{aligned}
\left( \sum_{i=1}^n f_i \right) \cdot \left( \sum_{i=1}^n f_i^{2k-1} \right)^2 &\leq \left( \sum_{i=1}^n f_i \right) \cdot (f_{max}^{k-1} \cdot \sum_{i=1}^n f_i^k) \\
&= \left( \sum_{i=1}^n f_i \right) \cdot \left( (f_{max}^k)^{\frac{k-1}{k}} \right) \cdot \left( \sum_{i=1}^n f_i^k \right) \\
&\leq \left( \sum_{i=1}^n f_i \right) \cdot \left( \left( \sum_{i=1}^n f_i^k \right)^{\frac{k-1}{k}} \right) \cdot \left( \sum_{i=1}^n f_i^k \right) \\
&= \left( \sum_{i=1}^n f_i \right) \cdot \left( \sum_{i=1}^n f_i^k \right)^{2k-1/k} \\
&\leq m^{1-1/k} \cdot \left( \sum_{i=1}^n f_i \right)^{1/k} \cdot \left( \sum_{i=1}^n f_i^k \right)^{2k-1/k} \\
&\leq m^{1-1/k} \cdot \left( \sum_{i=1}^n f_i^k \right)^{1/k} \cdot \left( \sum_{i=1}^n f_i^k \right)^{2k-1/k} \\
&= m^{1-1/k} \cdot \left( \sum_{i=1}^n f_i^k \right)^2
\end{aligned}$$

□

It follows that for  $s_1 \geq 8 \cdot m^{1-1/k} / \epsilon^2$  we get that with probability at least  $7/8$  our algorithm gives a  $(1 \pm \epsilon)$ -approximation. Now, we can boost the probability in a similar way as for the  $F_2$  sketch by computing the median of the output values of  $s_2 = 96 \log(1/\delta)$  instances of the above algorithm.

A problem with the above approach is that we cannot easily compute the random sample and, at the same time, count how many times the sample elements occur in the stream. Therefore, we proceed in a slightly different way. We choose an element  $x = \sigma_j$  from the stream  $\sigma_1, \dots, \sigma_m$  uniformly at random (for example, using reservoir sampling). Then we count all occurrences of  $x$  in  $\sigma_j, \dots, \sigma_m$ . Let us call this value  $r$ . It is not hard to see that we can modify reservoir sampling to compute  $r$ .

Finally, we compute the random value

$$X = m \cdot (r^k - (r-1)^k) .$$

$F_k$ -APPROXIMATION( $\sigma_i$ )

1.  $m = 0$
2. **while not EOF do**
3.     Let  $\sigma$  be the next element in the stream
4.      $m = m + 1$
5.     **if**  $m = 1$  **then**  $x = \sigma$ ;  $r = 1$
6.     **else**
7.         **if**  $\sigma = x$  **then**  $r = r + 1$
8.         Choose a number  $y$  uniformly at random from  $[0, 1]$
9.         **if**  $y \leq 1/i$  **then**
10.              $x = \sigma$
11.              $r = 1$
12. **output**  $m \cdot (r^k - (r - 1)^k)$

In order to compute  $X$  we need  $O(\log n + \log m)$  space to maintain the element  $x$ , the number of its occurrences and the overall number of elements in the stream. We compute the expected value of  $X$ .

$$\begin{aligned} \mathbf{E}[X] &= \frac{m}{m} \sum_{i=1}^n \sum_{j=1}^{f_i} (j^k - (j-1)^k) \\ &= \sum_{i=1}^n f_i^k = F_k \end{aligned}$$

To get an upper bound on the variance we compute  $\mathbf{E}[X^2]$ .

$$\begin{aligned} \mathbf{E}[X^2] &= \frac{m^2}{m} \sum_{i=1}^n \sum_{j=1}^{f_i} (j^k - (j-1)^k)^2 \\ &\leq m \cdot \sum_{i=1}^n \sum_{j=1}^{f_i} (j^k - (j-1)^k) \cdot (j^k - (j-1)^k) \end{aligned}$$

Now observe for any  $a > b > 0$  we have

$$a^k - b^k = (a - b) \cdot \sum_{i=0}^{k-1} a^{k-1-i} \cdot b^i \leq (a - b) \cdot k \cdot a^{k-1}$$

and so

$$(j^k - (j-1)^k) \leq k \cdot j^{k-1} .$$

This implies

$$\begin{aligned}
\mathbf{E}[X^2] &\leq m \cdot \sum_{i=1}^n \sum_{j=1}^{f_i} k \cdot j^{k-1} \cdot (j^k - (j-1)^k) \\
&\leq m \cdot \sum_{i=1}^n k \cdot f_i^{2k-1} \\
&= k \cdot F_1 \cdot F_{2k-1}
\end{aligned}$$

Again, we run  $s_1$  parallel instances of the algorithm and compute the average  $Y$  of the computed values. A similar analysis as before yields for  $s_1 = 8km^{1-1/k}/\epsilon$  that

$$(1 - \epsilon) \cdot F_k \leq Y \leq (1 + \epsilon) \cdot F_k .$$

We can amplify the probability to  $(1 - \delta)$  using Chernoff bounds as in the section on estimation of the second frequency moments. We summarize our results.

**Theorem 16** *For every  $k \geq 1$  and every  $\epsilon, \delta > 0$  there is a randomized algorithm that given a stream  $\sigma_1, \dots, \sigma_m$  of elements from  $\mathcal{U} = \{1, \dots, n\}$  a value  $\tilde{F}_k$  such that*

$$(1 - \epsilon) \cdot F_k \leq \tilde{F}_k \leq (1 + \epsilon) \cdot F_k .$$

*The algorithm uses  $O(\frac{k \log(1/\delta)}{\epsilon^2} m^{1-1/k} (\log n + \log m))$  bits of memory and processes each update in  $O(\frac{k \log(1/\delta)}{\epsilon^2} m^{1-1/k})$  time.*



## 4 The Merge and Reduce Principle

In this chapter we will learn a technique that is particularly useful to design data streaming algorithms in the context of geometric data streams. The general idea is to design two operations REDUCE and MERGE and prove that these two operations satisfy two basic properties. Then this will yield a streaming algorithm. We will now describe both operations and then show how they can be used to design a streaming algorithm.

### 4.0.1 Description of Operations

Let us assume we consider a certain optimization problem on a ground set of items. The REDUCE operation takes as input a set of  $n$  items and produces as output a set of  $\log^{O(1)} n$  items such that the cost of any solution for the output set of items is a  $(1 + \epsilon)$ -approximation for the corresponding solution of the input set. In particular, this requires us to specify the notion of corresponding solution.

The MERGE operation takes as input two sets of items and outputs one set of items (typically the union of both sets). These sets may be results from the REDUCE operation. It must satisfy that the cost of a solution for the output set is the sum of the corresponding costs for the input sets.

### 4.0.2 An Example

Let us consider a geometric data stream, i.e. a stream of  $n$  points  $\{p_1, \dots, p_n\}$  from  $\mathbb{R}^d$ . As an optimization problem we will consider the  $k$ -median problem. In the  $k$ -median problem, a solution is a set of  $k$  centers  $C$  that minimizes the sum of distances of the points  $p_i$  to their nearest center in  $C$ . To solve the  $k$ -median problem over data streams we proceed with a two step procedure. In the first step we compute a summary of the input data using the MERGE and REDUCE paradigm. Then we solve the  $k$ -median problem on the summary using an arbitrary  $(1 + \epsilon)$ -approximation algorithm. Here, we will only discuss the first step and not deal with the design of approximation algorithms for the  $k$ -median problem. The input to the REDUCE operation will be a positively weighted points set whose point weights sum up to  $n$ . We can think of a point with weight  $w(p)$  as  $p$  occurring with multiplicity  $w(p)$ . An unweighted point set can be interpreted as a weighted point set, where each point has weight 1. The REDUCE operation will now get a weighted point set  $P$  with overall weight  $n$  and compute a weighted point set  $S$  of  $\log^{O(1)} n$  points, such that for every set of  $k$  centers (every solution) and up to a factor of  $(1 \pm \epsilon)$  the cost for  $S$  is approximately the cost for  $P$ .

A MERGE operation corresponds to take the union of two weighted sets  $P_1, P_2$ , since for any set of  $k$  centers the cost for  $P_1 \cup P_2$  equals the cost for  $P_1$  plus the cost for  $P_2$ .

### 4.0.3 A Streaming Algorithm Using Merge and Reduce

How can we use the operations REDUCE and MERGE to design a streaming algorithm. The idea is as follows. We are maintaining a set of  $\log n$  buckets  $B_i$ ,  $0 \leq i \leq \log n$ . Before we process the stream all buckets are empty. We take the first  $\ell$  elements from the stream and apply the REDUCE operation. Then we store the resulting set of items in bucket  $B_0$ . We continue by taking the next  $\ell$  items from the stream and apply the REDUCE operation to them. Now, bucket  $B_0$  already contains a set of points and so we apply the MERGE operation to MERGE the output of the REDUCE operation with the points from bucket  $B_0$ . The resulting set of items is stored in bucket  $B_1$  and bucket  $B_0$  is emptied. So, we can proceed by reading the next  $\ell$  points and storing their summary in bucket  $B_0$ . In general, we proceed as follows

STREAMINGMERGEANDREDUCE

```
while not EOF do
  Let P be the next set of  $\ell$  items from the stream
   $S \leftarrow \text{REDUCE}(P)$ 
   $i \leftarrow 0$ 
  while  $B_i \neq \emptyset$  do
     $S \leftarrow \text{REDUCE}(\text{MERGE}(S, B_i))$ 
     $B_i \leftarrow \emptyset$ 
     $i \leftarrow i + 1$ 
   $B_i \leftarrow S$ 
```

We observe that the points stored in  $B_i$  correspond to a summary of  $2^i \cdot \ell$  points in the input stream. Therefore,  $\log n$  buckets suffice for  $n$  input items. It has been obtained by applying  $i$  REDUCE operations. Therefore, the error of our summary is  $(1 \pm \epsilon)^i$ . The error of the summary stored in the largest non-empty bucket is at most  $(1 \pm \epsilon)^{\log n}$ . Replacing  $\epsilon$  by  $c \cdot \epsilon / \log n$  will guarantee an error of at most  $\epsilon$  for some constant  $c$ . However, we need to know  $n$  (or a rough approximation of  $n$ ) in advance to do so. Although in practice this is not a big problem as  $n$  is supposed to be large and we are fine with, say, an  $n^{10}$  approximation, we will also derive an algorithm that does not require  $n$  to be known in advance at all. In general, the quality of a MERGE-and-REDUCE algorithm depends on the quality of the MERGE and REDUCE operations used and will be analysed ad-hoc.

## 4.1 A Streaming Algorithm for $k$ -Median Clustering via Coresets

Our first application of the MERGE-and-REDUCE principle will be to a streaming algorithm for the  $k$ -median problem. We will use the idea sketched above in the example. We use the MERGE and REDUCE operations to maintain a small summary of the data and solve the  $k$ -median only on this summary. We now discuss how to maintain such a summary. The REDUCE operation will receive a positively weighted point set with a bound on the sum of weights and it computes a weighted set with few points that satisfies the same bound on the point weights

and approximates the input point set. The MERGE operation simply takes the union of the two input sets.

Since we are dealing with weighted point sets let us define the  $k$ -median problem for weighted points as follows. We are given a point set  $P$  with positive point weights  $w(p)$  for each  $p \in P$ . Our goal is to find a set  $C$  of  $k$  centers that minimizes

$$\sum_{p \in P} w(p) \cdot d(p, C) ,$$

where  $d(p, C) = \min_{c \in C} \|p - c\|_2$ .

To implement the REDUCE operation we seek to compute a small weighted point set that approximately has the properties as the input point set. Such a point set is also called a *coreset* and is formally defined as follows.

**Definition 4.1.1 (Coreset)** *Let  $P \subseteq \mathbb{R}^d$  be a weighted point set. A weighted point set  $S \subseteq \mathbb{R}^d$  is called  $(k, \epsilon)$ -coreset for a weighted point set  $P \subseteq \mathbb{R}^d$  under the  $k$ -median objective function, if for every set  $C$  of  $k$  centers we have*

$$(1 - \epsilon) \cdot \text{cost}(P, C) \leq \text{cost}(S, C) \leq (1 + \epsilon) \cdot \text{cost}(P, C) .$$

Our goal is to compute a coreset of the input points that arrive in the data stream. We then use an arbitrary  $(1 + \epsilon)$ -approximation to solve the problem on the coreset. The results will be a  $(1 + 4\epsilon)$ -approximation as we will see in the following. An optimal solution  $C_{Opt}$  for  $P$  has cost  $\text{cost}(S, C_{Opt}) \leq (1 + \epsilon) \cdot \text{cost}(P, C_{Opt})$  for  $S$ . Hence, the cost of an optimal solution for  $S$  is at most  $(1 + \epsilon) \cdot \text{cost}(P, C)$ . Now consider an optimal solution  $C'_{Opt}$  for  $S$ . We have  $\text{cost}(S, C'_{Opt}) \geq (1 - \epsilon) \cdot \text{cost}(P, C'_{Opt})$ . Since  $C'_{Opt}$  is optimal for  $S$  we have  $\text{cost}(S, C'_{Opt}) \leq \text{cost}(S, C_{Opt})$ . It follows that

$$\text{cost}(P, C'_{Opt}) \leq \frac{1}{1 - \epsilon} \text{cost}(S, C'_{Opt}) \leq \frac{1}{1 - \epsilon} \text{cost}(S, C_{Opt}) \leq \frac{1 + \epsilon}{1 - \epsilon} \cdot \text{cost}(P, C_{Opt}) ,$$

which is at most  $(1 + 4\epsilon) \cdot \text{cost}(P, C_{Opt})$  for  $\epsilon \leq 1/2$ . Replacing  $\epsilon$  with  $\epsilon/4$  we obtain a  $(1 + \epsilon)$ -approximation algorithm for the  $k$ -median problem. It remains to describe how to maintain a coreset in the streaming scenario.

### 4.1.1 From Clusterings to Coresets

The following coreset construction can be also viewed as a pre-clustering that uses more than  $k$  centers (but not too many) and has cost at most  $\epsilon \cdot \text{cost}(P, C_{Opt})$ . Once we have computed the pre-clustering we define the coreset to be the set of cluster centers and the weight of the points is the number of points inside each cluster. Let us assume we have a partition of the input space into clusters  $C_1, \dots, C_m$  and corresponding cluster centers  $c_1, \dots, c_m$  such that

$$\sum_i \sum_{p \in C_i} \|p - c_i\| \leq \epsilon \text{cost}(P, C_{Opt}) ,$$

where  $C_{Opt}$  is an optimal solution with  $k$  centers. We will show that the set  $\{c_1, \dots, c_m\}$  with  $w(c_i) = |C_i|$  is a  $(k, \epsilon)$ -coreset. In order to prove this we need the following lemma, which shows that moving a point by a distance  $D$  changes the distance to any point  $q$  by at most  $\pm D$ .

**Lemma 4.1.2** Let  $q \in \mathbb{R}^d$  be a point. Let  $p, p' \in \mathbb{R}^d$ . Then

$$\left| \|p - q\| - \|p' - q\| \right| \leq \|p - p'\| .$$

**Proof :** We have  $\|p - q\| \leq \|p' - q\| + \|p - p'\|$  and so  $\|p - q\| - \|p' - q\| \leq \|p - p'\|$ . Also,  $\|p' - q\| \leq \|p - q\| + \|p - p'\|$  and so  $\|p' - q\| - \|p - q\| \leq \|p - p'\|$ . This implies the lemma.  $\square$

Let us fix an arbitrary set  $C$  of  $k$  centers. Let  $\{c_1, \dots, c_m\}$  be as defined above. We can think of constructing this set by moving every point  $p \in C_i$  to  $c_i$ . By the lemma above, this changes its distance to every point in  $C$  by at most  $\|p - c_i\|$ . Hence, it also changes the distance to the nearest point in  $C$  by at most this value. Thus, the cost of solution  $C$  changes by at most  $\sum_i \sum_{p \in C_i} \|p - c_i\|$ , which is the cost of the pre-clustering. Since we have chosen the pre-clustering in such a way that  $\sum_i \sum_{p \in C_i} \|p - c_i\| \leq \epsilon \cdot \text{cost}(P, C_{opt}) \leq \epsilon \cdot \text{cost}(P, C)$  we know that

$$(1 - \epsilon) \cdot \text{cost}(P, C) \leq \text{cost}(S, C) \leq (1 + \epsilon) \cdot \text{cost}(P, C)$$

and so the set  $\{c_1, \dots, c_m\}$  is a coreset.

## 4.1.2 A Coreset Construction

In order to construct our coreset we pursue the approach sketched above and compute a pre-clustering. This pre-clustering is computed by partitioning the input space into  $m$  regions. For each region  $R_i$  we define  $C_i = P \cap R_i$ . The coreset is obtained by taking a certain point  $c_i$  from each  $R_i$  and assigning the weight  $|C_i|$  to it. The cost of clustering the  $C_i$  using the centers  $c_i$  will be at most  $\epsilon \cdot \text{cost}(P, C_{opt})$ .

Our first step to construct the pre-clustering is to compute a constant factor approximation  $A = \{a_1, \dots, a_q\}$  with  $q = O(k)$  and  $\text{cost}(P, A) \leq c \cdot \text{cost}(P, C_{opt})$  for some constant  $c$ . It is possible to compute such a clustering in  $O(nkd)$  time, but we will not discuss the algorithm here.

In order to construct the partitioning of the input space we put an exponential grid around each of the centers in  $A$ . For this purpose let  $P_i$  be the subset of  $P$  that is closest to  $a_i$ . Further let  $R = \text{cost}(P, A)/(c \cdot n) \leq \text{cost}(P, C_{opt})/n$  be a lower bound on the average cost of a point in an optimal solutions. We also have  $\|p - a_i\| \leq c \cdot n \cdot R$ , since  $\|p - a_i\| \leq \text{cost}(P, A)$ . To define the exponential grid let  $Q : i, j$  be an axis-aligned cube with side length  $R \cdot 2^j$  and center  $a_i$  for  $j = 0, \dots, M = \lceil 2 \log(cn) \rceil$ . Let  $V_{i,0} = Q_{i,0}$  and  $V_{i,j} = Q_{i,j} \setminus Q_{i,j-1}$ . Subdivide  $V_{i,j}$  using a grid with side length  $r_j = \epsilon \cdot R \cdot 2^j / 10(cd)$ . Let  $G_i$  denote the resulting exponential grid for  $V_{i,0}, \dots, V_{i,M}$ . The union of these cells defines our subdivision of the input space. We obtain the coreset by counting the number of points inside each grid cell and replacing them by an arbitrary point inside the cell weighted by the number of replaced points. Let  $S$  denote the resulting point set.

We first prove that  $S$  is small and then show that the cost of the constructed pre-clustering is small, which implies that  $S$  is a coreset.

### Lemma 4.1.3

$$|S| = O(|A| \cdot \log n / \epsilon^d) .$$

**Proof :**  $V_{i,j}$  is subdivided into at most  $(\frac{10cd}{\epsilon})^d$  cells. From  $0 \leq j \leq M = O(\log n)$  and  $1 \leq i \leq |A|$  it follows that  $|S| = O(|A| \cdot \log n / \epsilon^d)$ .  $\square$

**Theorem 17** *The weighted set  $S$  is a  $(k, \epsilon)$ -coreset for  $P$ .*

**Proof :** We only have to show that the cost for clustering the points inside the grid cells using the corresponding coreset point as a center is at most  $\epsilon \cdot \text{cost}(P, C_{Opt})$ . For a point  $p$  let us denote by  $c(p)$  the coreset point of the grid cell that contains  $p$ . We get

$$\begin{aligned} \sum_{p \in P} \|p - c(p)\| &= \sum_{p \in P, d(p, A) \leq R} \|p - c(p)\| + \sum_{p \in P, d(p, C_{Opt}) > R} \|p - c(p)\| \\ &\leq \frac{\epsilon}{10c} \cdot n \cdot R + \frac{\epsilon}{10c} \cdot \sum_{p \in P} d(p, C_{Opt}) \\ &\leq \frac{2\epsilon}{10c} \cdot \text{cost}(P, C_{Opt}) \\ &\leq \epsilon \text{cost}(P, C_{Opt}) \end{aligned}$$

$\square$

## 4.2 Coresets for High Dimensional Point Sets

The coreset construction from the last section is small with respect to  $n$ , but the size of the coreset and hence the space requirement of our algorithm depends exponentially on the dimension of the input space. In this section we will develop a new construction that has only linear dependence on the dimension. To obtain this result we need a new construction. We cannot hope for a pre-clustering with relatively few centers and with small cost compared to the cost of an optimal clustering solution. A simple counter example is the unit metric, where each pair of points has distance exactly 1. We can construct such a metric using  $n$  points in  $n$ -dimensional space. The Johnson-Lindenstrauss Lemma tells us that we can embed this  $n$ -dimensional point set in  $O(\log n / \epsilon^2)$  dimensions in such a way that every distance is preserved up to a factor of  $(1 + \epsilon)$ . Consider a clustering with  $m$  centers and the corresponding partition into clusters  $C_1, \dots, C_m$ . In each cluster we group the points into pairs leaving possible one point unmatched. By the triangle inequality, each pair has cost at least 1. Hence the overall cost of the clustering is at least  $n - m/2$ . Thus, for  $\epsilon \ll 1/2$  and  $m = o(n)$  there is no pre-clustering with cost  $\epsilon \cdot (P, C_{Opt})$ , where  $C_{Opt}$  denotes an optimal solution with  $k \ll m$  centers. Thus, we require a new idea.

**Idea.** Our new construction combines two results that we have already proved. One of them is the fact that we can derive an additive error bound for a random sample of size  $s$ , if the points come from a space with diameter  $D$ . For our purposes we will assign weight  $n/s$  to every sample point. We can parametrize  $s$  in such a way that the additive error due to the sampling is at most  $\epsilon n D / c$ , where  $n$  is the size of the point set and  $\alpha$  is a constant. This is then combined with the charging argument of the previous coreset construction in the following way. We first

compute a (bi-criteria)  $\alpha$ -approximation  $A$  that uses  $O(k)$  centers. Then we put  $O(\log n)$  balls with exponentially increasing radius around these centers. These balls define a partition of the space into rings. Each ring has inner distance  $D$  from the center and outer distance  $2D$  for some value  $D$ . Therefore, every point in this ring contributes with at least  $D$  to the cost of the approximate solution. If we use  $m$  to denote the number of point in this ring, then the overall contribution of these points to the approximate solution is at least  $mD$ . But the additive error of the sample is at most  $\epsilon mD/c$ . Summing up over all rings gives an overall error of  $\sum \epsilon mD/c \leq \epsilon \text{cost}(P, A)/c$ . Choosing  $c = \alpha/2$  gives an error of at most  $\frac{\epsilon}{2} \text{Opt}$ . It remains to consider the error introduced by the (small) inner ball in the center of the set of rings. These balls are chosen small enough to guarantee that the overall error induced by the sampling is at most  $\frac{\epsilon}{2} \text{Opt}$ . Thus, summing up, the error for every set of centers is at most  $\epsilon \text{Opt}$ , which proves that the sample set is a coresets.

### The Algorithm.

HIGHDIMENSIONALCORESET( $P, s$ )

1. Compute  $\alpha$ -Approximation  $A = \{c_1, \dots, c_\ell\}$  with  $\ell = O(k)$  centers;  $\alpha = O(1)$
2. Let  $P_i \subseteq P$  be the set of points from  $P$  that are closest to  $c_i$
3. Let  $D = \text{cost}(P, A)/(\alpha n)$
4. **for**  $i = 1$  **to**  $k$  **do**
5.      $P_{i,0} = \{p \in P_i \mid d(p, c_i) \leq D\}$
6.     **for**  $j = 1$  **to**  $\log(\alpha n)$  **do**
7.          $P_{i,j} = \{p \in P_i \mid 2^{j-1} \cdot D \leq d(p, c_i) \leq 2^j \cdot D\}$
8.         Draw a random set  $S_{i,j}$  of  $s$  points from  $P_{i,j}$  uniformly at random with repetition
9.         Let the weight of the points in  $S_{i,j}$  be  $|P_{i,j}|/s$
10. **return**  $S = \bigcup_{i,j} S_{i,j}$

**Overview of the Analysis.** We have to show that for any set of centers  $C$  we get  $|\text{cost}(P, C) - \text{cost}(S, C)| \leq \epsilon \cdot \text{cost}(P, C)$ . In order to do so, we consider every  $P_{i,j}$  separately. Then we distinguish between two types of sets of centers. First we show that, if every center in  $C$  is far away from  $P_{i,j}$  then it does not matter which points from  $P_{i,j}$  are chosen as sample points, because their pairwise distance is small in comparison to their distance to  $C$ . It suffices that the sum of weights equals  $|P_{i,j}|$ . Then we consider a fixed set of sets of centers  $\mathcal{C}$ . We show that for every  $C \in \mathcal{C}$  with high probability  $|\text{cost}(P_{i,j}, C) - \text{cost}(S_{i,j}, C)|$  is small compared to the contribution of  $P_{i,j}$  to our approximate solution  $A$ . Then we argue that for each  $P_{i,j}$  we only have to consider a finite number of sets  $C$  to guarantee that  $|\text{cost}(P_{i,j}, C) - \text{cost}(S_{i,j}, C)|$  is small for all  $C$ . The argument is as follows. We already know that we only have to consider centers that are not too far away from  $P_{i,j}$ . To find a bound for the error of nearby centers we discretize the input space and consider only the solutions on the grid. If the grid is fine enough, then this will give us sufficiently precise bounds for every near solution.

**Any Set of Far Away Centers has Small Error.** We now want to show that we have small error for every set of centers  $C \subseteq \mathbb{R}^d$ . Our first observation is that if  $d(C, P_{i,j}) > 2^{j+3} \cdot D/\epsilon$

then the error induced by the sampling is small compared the  $cost(P_{i,j}, C)$ . In order to prove this, we will use the 'pre-clustering' argument from the previous coresets construction. We show that any clustering that uses the points  $S_{i,j}$  as cluster centers and assigns to  $S_{i,j}$  exactly  $|P_{i,j}|/s$  points has cost at most  $\epsilon \cdot cost(P_{i,j}, C)$ . Hence, the cost of clustering  $C$  is approximated within a factor of  $(1 + \epsilon)$ . Since  $|P_{i,j}|/s$  may not be integral, we allow to assign points fractionally.

**Claim 4.2.1** *Let  $Q$  be a point set in a ball with radius  $R$  and let  $C$  be a set of  $k$  centers with  $d(Q, C) > 8R/\epsilon$ . Then we can replace  $Q$  by an arbitrary positively weighted point set  $S \subseteq Q$  with overall weight  $|Q|$  and we have*

$$|cost(Q, C) - cost(S, C)| \leq \frac{\epsilon}{4} \cdot cost(P, C) .$$

**Proof :** We distribute the weight of every point in  $S$  in such a way that every point from  $Q$  gets an overall weight of exactly 1. This is always possible, since the overall weight of  $S$  is  $|Q|$ . For two points  $p \in S$  and  $q \in Q$  let  $w_{p,q}$  be the weight that  $q$  receives from  $p$ . Then we have

$$\begin{aligned} & \left| \sum_{q \in Q} D_{\ell_2}(q, C) - \frac{|P|}{s} \cdot \sum_{p \in S} D_{\ell_2}(p, C) \right| \\ = & \left| \sum_{q \in Q} D_{\ell_2}(q, C) - \sum_{p \in S} \sum_{q \in Q} w_{p,q} \cdot D_{\ell_2}(p, C) \right| \\ = & \left| \sum_{q \in Q} \sum_{p \in S} w_{p,q} D_{\ell_2}(q, C) - \sum_{p \in S} \sum_{q \in Q} w_{p,q} \cdot D_{\ell_2}(p, C) \right| \\ = & \left| \sum_{p \in S} \sum_{q \in Q} w_{p,q} D_{\ell_2}(q, C) - \sum_{p \in S} \sum_{q \in Q} w_{p,q} \cdot D_{\ell_2}(p, C) \right| \\ \leq & \sum_{p \in S} \sum_{q \in Q} |w_{p,q} (D_{\ell_2}(q, C) - D_{\ell_2}(p, C))| \\ \leq & \sum_{p \in S} \sum_{q \in Q} w_{p,q} \cdot R \\ \leq & |Q| \cdot R \\ \leq & \frac{\epsilon}{4} cost(Q, C) \end{aligned}$$

□

Using  $R = 2^j \cdot D$  and  $Q = P_{i,j}$  we have that in this case the sampling error is at most  $\pm \frac{\epsilon}{4} \cdot cost(P_{i,j}, C)$ .

Thus, we know that for every set of centers that has a large distance from  $P_{i,j}$  our sampling approach works well. It remains to prove that for every set of centers that is close to  $P_{i,j}$  we get small error. In order to obtain this result we first prove that for a finite set of sets of centers  $\mathcal{C}$  we have small error for all  $C \in \mathcal{C}$ . To prove that we can approximate the cost of every solution we discretize the space close to  $P_{i,j}$  and consider only solutions on this grid. We can restrict ourself to solutions where each point in  $C$  is relatively close to  $P_{i,j}$ . This is because at least one point of  $C$  must be close to  $P_{i,j}$  because otherwise we are in the first case. But then any center that is much further away cannot be closest center and is redundant for our purposes.

**A Fixed Set of Centers has Small Error.** We apply Lemma 2.4.2. Let us fix an arbitrary solution  $C$  and set  $P_{i,j}$ . We define  $f_C(p) = d(p, C) - d(P_{i,j}, C)$ . This implies that  $0 \leq f_C(p) \leq M$  for  $M = 2^{j+1} \cdot D$ , since the diameter of point set  $P_{i,j}$  is at most  $2^{j+1}D$  and because of the triangle inequality. Now let  $\mathcal{C} = \{C_1, C_2, C_3, \dots\}$  be a finite set of sets of centers. Further let  $F = \{f_C : C \in \mathcal{C}\}$ . Then we have for  $s \geq \frac{M^2}{2\lambda^2}(\ln|F| + \ln(2/\delta))$ :

$$\Pr[\exists f \in F : \left| \frac{\sum_{p \in P_{i,j}} f(p)}{|P_{i,j}|} - \frac{\sum_{p \in S_{i,j}} f(p)}{|S_{i,j}|} \right| \geq \lambda] \leq \delta .$$

Since the weight of each  $p \in S_{i,j}$  is  $|P_{i,j}|/|S_{i,j}|$  we get

$$\Pr[\exists f \in F : \left| \sum_{p \in P_{i,j}} f(p) - \sum_{p \in S_{i,j}} w(p)f(p) \right| \geq \lambda \cdot |P_{i,j}|] \leq \delta ,$$

where  $w(p)$  denotes the weight of  $p$ . It follows

$$\begin{aligned} & \Pr[\exists C \in \mathcal{C} : \left| \sum_{p \in P_{i,j}} f(p) - \sum_{p \in S_{i,j}} w(p)f(p) \right| \geq \lambda \cdot |P_{i,j}|] \leq \delta \\ &= \Pr[\exists C \in \mathcal{C} : \left| \text{cost}(P_{i,j}, C) - \text{cost}(S_{i,j}, C) \right| \geq \lambda \cdot |P_{i,j}|] \leq \delta \end{aligned}$$

Now we can set  $\lambda = \frac{\epsilon M}{16\alpha}$  and obtain

$$\Pr[\exists C \in \mathcal{C} : \left| \text{cost}(P_{i,j}, C) - \text{cost}(S_{i,j}, C) \right| \geq \frac{\epsilon}{16\alpha} \cdot |P_{i,j}| \cdot 2^{j+1}D] \leq \delta .$$

Hence,

**Lemma 4.2.2** *Let  $\mathcal{C}$  be a set of sets of centers and let  $0 < \delta, \epsilon$ . For  $s \geq \frac{256\alpha}{\epsilon^2}(\ln|\mathcal{C}| + \ln(2/\delta))$  we have*

$$\Pr[\exists C \in \mathcal{C} : \left| \text{cost}(P_{i,j}, C) - \text{cost}(S_{i,j}, C) \right| \geq \frac{\epsilon}{16\alpha} \cdot |P_{i,j}| \cdot 2^{j+1}D] \leq \delta .$$

**Any Set of Near Centers has Small Error.** The analysis for the case that at least one center is near to the input is similar to the analysis of uniform sampling in Chapter 2.4. We first discretize the input space by restricting ourselves to solutions on a grid with cell width  $\epsilon \cdot \text{Opt}/(4n\sqrt{d})$ . Let  $G$  be the set of grid points and  $G_{\text{eff}} = \{p \in G : d(p, P) \leq \frac{32}{\epsilon} \cdot \text{cost}(P, A)\}$ . Then we apply the analysis for a fixed set of centers. We first prove an upper bound on the cardinality of  $G_{\text{eff}}$ .

**Claim 4.2.3** *There is a constant  $c_d$  such that*

$$|G_{\text{eff}}| \leq c_d \cdot n^{d+1}/\epsilon^{2d} .$$

**Proof :** For each point from  $P$  there exists a  $d$ -dimensional cube with side length  $2 \cdot \frac{16}{\epsilon} \cdot \text{cost}(P, A)$  such that the union of these cubes contains all points from  $G_{\text{eff}}$ . Since the grid has cell width  $\epsilon \cdot \text{Opt}/(4n\sqrt{d})$ , there are at most  $c_d \cdot n^d/\epsilon^{2d}$  grid points in this cube for a constant

$c_d$  that depends on the dimension and because  $\alpha \leq \text{cost}(P, A)/\text{Opt}$  is a constant. The claim follows from  $|P| = n$ .  $\square$

Now consider an arbitrary solution  $C'$  such that at least one center has distance to  $P_{i,j}$  less than  $2^{j+3}D\epsilon$ . We snap all points from  $C'$  to their nearest grid point to obtain a solution  $C$ . By our choice of the grid width we move every point at most a distance  $\epsilon \cdot \text{Opt}/(4n)$ . Hence,  $|\text{cost}(P, C') - \text{cost}(P, C)| \leq \frac{\epsilon}{4} \cdot \text{Opt}$ . For every grid solution  $C$  we define  $C_{\text{eff}} = C \cap G_{\text{eff}}$ . We will show that we can restrict ourself on the solution  $C_{\text{eff}}$  since the points in  $P_{i,j}$  have no nearest neighbor in  $C \setminus C_{\text{eff}}$ . Since  $d(p, A) \leq \text{cost}(P, A)$  we have  $2^{j-1} \cdot D \leq \text{cost}(P, A) \Leftrightarrow D \leq \text{cost}(P, A)/2^{j-1}$  for all  $j$  with  $P_{i,j} \neq \emptyset$ . Hence,

$$d(P_{i,j}, C) \leq 2^{j+3}D/\epsilon \leq \frac{16}{\epsilon} \text{cost}(P, A) .$$

For every point  $p \in P_{i,j}$  we now get

$$d(p, C) \leq (2 + \frac{16}{\epsilon}) \text{cost}(P, A) \leq \frac{32}{\epsilon} \cdot \text{cost}(P, A)$$

because the diameter of  $P_{i,j}$  is at most  $\text{cost}(P, A)$ . It follows that no point from  $P_{i,j}$  has a nearest neighbor in  $C \setminus C_{\text{eff}}$ . Now we can apply the analysis for the case of a fixed set of set of centers. We define  $\mathcal{C} = \{C \subseteq G_{\text{eff}} : |C| \leq k\}$ . We have  $|\mathcal{C}| \leq |G_{\text{eff}}|^k \leq c_d^k \cdot n^{k(d+1)}/\epsilon^{2dk}$ . By Lemma 4.2.2 we have  $s \geq \frac{256\alpha}{\epsilon^2} (\ln |\mathcal{C}| + \ln(2/\delta))$

$$\Pr[\exists C \in \mathcal{C} : |\text{cost}(P_{i,j}, C) - \text{cost}(S_{i,j}, C)| \geq \frac{\epsilon}{16\alpha} \cdot |P_{i,j}| \cdot 2^{j+1}D] \leq \delta .$$

Now we set  $\delta = \frac{1}{3k(1+\log(\alpha n))}$  and write  $\mathcal{C}_{i,j}$  for the set  $\mathcal{C}$  that corresponds to  $P_{i,j}$ . We get

$$\Pr[\exists i, j \exists C_{\text{eff}} \in \mathcal{C}_{i,j} : |\text{cost}(P_{i,j}, C_{\text{eff}}) - \text{cost}(S_{i,j}, C_{\text{eff}})| \geq \frac{\epsilon}{16\alpha} \cdot |P_{i,j}| \cdot 2^{j+1}D] \leq \delta \cdot k \cdot (1 + \log(\alpha n)) ,$$

since there are at most  $k \cdot (1 + \log(\alpha n))$  sets  $P_{i,j}$ . For our choice of  $\delta$  we have error probability at most  $1/3$ .

Finally, we have to show that the overall error is small. We use the following claim.

**Claim 4.2.4** For  $j \geq 1$  we have

$$\text{cost}(P_{i,j}, A) \geq |P_{i,j}| \cdot 2^{j-1} \cdot D.$$

**Proof :** From the definition of  $P_{i,j}$  it follows that every point in  $P_{i,j}$  has distance at least  $2^{j-1} \cdot D$  from  $A$ . Hence, the claim follows.  $\square$

We obtain for  $j \geq 1$

$$\Pr[\exists C \in \mathcal{C} : |\text{cost}(P_{i,j}, C) - \text{cost}(S_{i,j}, C)| \geq \frac{\epsilon}{4\alpha} \cdot |P_{i,j}| \cdot \text{cost}(P, A)] \leq \delta .$$

For  $j = 0$  we have

$$\Pr[\exists C \in \mathcal{C} : |\text{cost}(P_{i,j}, C) - \text{cost}(S_{i,j}, C)| \geq \frac{\epsilon}{8\alpha} \cdot |P_{i,j}| \cdot D] \leq \delta .$$

Summing up the error for  $j \geq 1$  gives an overall error for any fixed set of centers  $C$  of at most  $\frac{\epsilon}{4\alpha} \cdot \text{cost}(P, A) \leq \frac{\epsilon}{4} \cdot \text{Opt} \leq \frac{\epsilon}{4} \cdot \text{cost}(P, C)$ . For the case  $j = 0$  the sum of error is at most  $\frac{\epsilon}{8\alpha} \cdot n \cdot \text{cost}(P, A) / (\alpha n) \leq \frac{\epsilon}{8} \cdot \text{Opt} \leq \frac{\epsilon}{8} \cdot \text{cost}(P, C)$ . Summing up the errors for the sampling and for considering only grid solutions we obtain an overall error of at most  $\epsilon \cdot \text{cost}(P, C)$ . Hence, the computed point set is a coresets. Plugging in our choice for  $\delta$  and the sizes of the  $\mathcal{C}_{i,j}$  we obtain

**Theorem 18** *For  $s \geq c \cdot \frac{k^2 d}{\epsilon^2} \cdot \log(nd/\epsilon)$  with probability at least  $2/3$  the set  $S$  is a  $(k, \epsilon)$ -coreset. The cardinality of  $S$  is  $O(k^3 \cdot \log^2 n \cdot d \log(nd/\epsilon)/\epsilon^2)$ .*

## 5 Streaming Algorithms via Embeddings into Tree Metrics

In this chapter we will develop algorithms for dynamic geometric data streams. We will learn a technique that can be used to reduce problems over such data streams to problems over simple statistics of high dimensional vector, i.e. problems like approximating  $F_0$  or  $F_2$ .

We first define dynamic geometric data streams. A dynamic geometric data stream is a sequence of INSERT and DELETE operations of points from a discrete space  $\{1, \dots, \Delta\}^d$ . As always when we consider dynamic streams we require that the stream is consistent, i.e. no points are removed that are not present in the current set of points and no points are inserted twice.

An example for an applications where dynamic geometric data streams naturally occur is a mobile ad-hoc network. Here we have a set of mobile devices (laptops, PDAs, cell phones, etc.) that form a communication network. We do not want to rely on base stations and instead form the network using only our mobile devices. Since most mobile devices have limited energy supply we would like to maintain an efficient network. In order to do this, each node has to broadcast its new position from time to time. If we implement such a broadcast as a DELETE of the old position of the device and an INSERT of its new position, we get a dynamic geometric data stream.

The algorithms we develop in this chapter are based on the following method. In a first step the current point set  $P$  (endowed with Euclidean metric) is probabilistically embedded into a tree such that for every point  $p, q \in P$  the expected distance in the tree is at most a factor  $O(d \log |P|)$  larger than the corresponding distance  $\|p - q\|_2$ . For many problem we can show that an optimal or near optimal solution on a tree has certain properties, from which we can approximate the cost of the solution by looking at certain statistics of the (new) metric space.

### 5.1 Embeddings into Trees

In this section we develop an embedding of a point set  $P$  into a tree metric that tries to guarantee that the distances between points in the original space do not differ very much from the corresponding distances in the tree. We need a few definitions.

**Definition 5.1.1 (Metric Embedding)** *Given two metric spaces  $(P, d)$  and  $(P', d')$ , a mapping from  $P$  to  $P'$  is called an embedding of  $(P, d)$  into  $(P', d')$ .*

**Definition 5.1.2 (Tree Metric)** *Let  $T$  be a tree with vertex set  $V$ . The corresponding tree metric  $(V, d_T)$  is defined by letting  $d_T(u, v)$  be the length of the (unique) path between  $u$  and  $v$ .*

**Definition 5.1.3 ( $\alpha$ -Probabilistic Approximation)** Let  $S$  be a family of metric spaces with point set  $P$  and let  $D$  be a distribution over  $S$ . Then  $(S, D)$  is an  $\alpha$ -probabilistic approximation of  $(P, d)$ , if

- every metric space in  $S$  dominates  $(P, d)$
- for all  $p, q \in P$  we have  $\mathbf{E}_{(P, d') \sim D}[d'(p, q)] \leq \alpha \cdot d(p, q)$ ,

where a metric  $(P, d')$  dominates  $(P, d)$ , if  $d'(p, q) \geq d(p, q)$  for all  $p, q \in P$ .

**Definition 5.1.4 (k-HST)** A rooted tree  $T$  is a  $k$ -hierarchically well-separated tree ( $k$ -HST), if

- the distances from every vertex to its children are identical
- pnever path from the root to a leaf the distances of subsequent edges drop by a factor of at least  $k$ .

### 5.1.1 Construction of a 2-HST for $P$

In the following we develop an algorithm that computes a probabilistic embedding of a point set  $P$  into a 2-HST. The distribution over tree metrics defined by this algorithm is an  $\alpha$ -probabilistic approximation of  $(P, d)$ . The idea of the embedding is simple. We use several randomly shifted grids  $G_i$  with cell width  $2^{i-1}$  for  $1 \leq i \leq \log \Delta$ . In grid  $G_0$  every point of  $P \subseteq \{1, \dots, \Delta\}^d$  is in a single cell. These points/cells are the leaves of our tree (level 0). In grid  $G_i$  all non-empty cells are interior nodes of the tree in level  $i$ . Each such node is connected to all nodes that correspond to the non-empty cells in grid  $G_{i-1}$ . The connecting edge has length  $\sqrt{d} \cdot 2^{i-1}$ .

2-HST-COMPUTATION( $P$ )

Let  $\alpha$  be a vector chosen uniformly at random from  $[0, \Delta]^d$ .

**for**  $i = 1$  **to**  $\log \Delta$  **do**

Let  $G_i$  be a grid with cell width  $2^{i-1}$  shifted by  $\alpha$

$T \leftarrow$  empty tree

Insert every point in  $P$  as a node of  $T$  at level 0

**for**  $i = 1$  **to**  $\log \Delta$  **do**

For each non-empty cell in grid  $G_i$  insert a node in level  $i$  of  $T$

Connect every node  $u$  in level  $i$  to the nodes in level  $i - 1$  whose corresponding grid cells are contained in the grid cell that corresponds to  $u$ .

Set the edge weight to  $\sqrt{d} \cdot 2^{i-1}$

Insert a root in level  $\log \Delta + 1$  of  $T$  and connect it to every node in level  $\log \Delta$  using an edge of length  $\Delta$ .

We first observe that  $(P, d_T)$  dominates  $(P, d)$ . Let us consider two point  $p, q$  and the coarsest grid  $G_i$  such that  $p$  and  $q$  are in different cells. Then  $d_T(p, q) > \sqrt{d}2^i$ . Further, we know that  $p$  and  $q$  are in the same cell in grid  $G_{i+1}$ . Hence their distance can be at most the diagonal of that cell, which is  $\sqrt{d} \cdot 2^i$ . Hence,  $(P, d_T)$  dominates  $(P, d)$ .

**Theorem 19** Let  $P$  be a set of point in the discrete space  $\{1, \dots, \Delta\}^d$  and let  $d(\cdot, \cdot)$  be the Euclidean distance. Then the distribution over 2-HSTs as computed by algorithm 2-HST-COMPUTATION is an  $O(d \log \Delta)$ -probabilistic approximation of the metric  $(P, d)$ .

**Proof :** We show for two points  $p, q$  that  $\mathbf{E}[d_T(p, q)] = O(d \log \Delta) \cdot d(p, q)$ . Let  $X_i$  be the indicator random variable for the event that  $p, q$  are in different grid cells in level  $i$ . Then we can write

$$d_T(p, q) = \sum_{i=0}^{\log \Delta} X_i \cdot \sqrt{d} \cdot 2^i .$$

Hence,

$$\mathbf{E}[d_T(p, q)] = \sum_{i=0}^{\log \Delta} \mathbf{E}[X_i] \cdot \sqrt{d} \cdot 2^i$$

by linearity of expectation. For  $d = 1$  we have

$$\Pr[X_i] \leq \frac{d(p, q)}{\sqrt{d} \cdot 2^{i-1}} .$$

For  $d > 1$  this implies

$$\Pr[X_i] \leq \sqrt{d} \cdot \frac{d(p, q)}{2^{i-1}} .$$

Hence,  $\mathbf{E}[X_i] \leq \sqrt{d} \cdot \frac{d(p, q)}{2^{i-1}}$ . We obtain

$$\mathbf{E}[d_T(p, q)] \leq \sum_{i=0}^{\log \Delta} \sqrt{d} \cdot \frac{d(p, q)}{2^{i-1}} \cdot \sqrt{d} \cdot 2^i = 2 \cdot d \cdot (1 + \log \Delta) \cdot d(p, q) = O(d \log \Delta) \cdot d(p, q) .$$

□

## 5.2 Minimum Spanning Tree Cost

Now we will use the previous result to develop a streaming algorithm that maintains the cost of a Euclidean minimum spanning tree of  $P$ , i.e. the minimum spanning of the metric  $(P, d)$ . For a weighted tree  $T$  we define  $cost(T)$  as the sum of the edge weights of tree. We will show that the expected cost of a 2-HST computed by the algorithm above is a  $O(d \log \Delta)$  approximation of the cost of the Euclidean minimum spanning tree.

Let  $T$  be a 2-HST computed by algorithm 2-HST-COMPUTATION. We show that there exists a spanning tree of  $(P, d_T)$  with cost at most  $2 \cdot cost(T)$ . We first transform  $T$  into a tree  $T'$  such that (a) the nodes of  $T'$  are the points in  $P$  (the leaves of  $T$ ), (b)  $(P, d_{T'})$  dominates  $(P, d_T)$ , and (c)  $cost(T') \leq 2 \cdot cost(T)$ . This can be done bottom-up by the following procedure. The invariant is that all subtrees rooted at level  $i$  consists of leaves only. Now let us consider a node at level  $i + 1$ . We choose one of its subtrees and connect all other subtrees to it using edges of length  $2 \cdot 2^i$  (instead of  $2^i$ ).

Now we replace the weight of every edge  $(u, v)$  in  $T'$  by  $d(u, v)$ . Let  $T''$  be the resulting tree. We have  $cost(T'') \leq cost(T') \leq 2 \cdot cost(T)$ . Further  $(P, d_{T''})$  dominates  $(P, d)$  and  $T''$  is a spanning tree for  $P$ .

**Lemma 5.2.1** *Let  $T^*$  be a minimum spanning tree for  $P$ . Then we have  $\mathbf{E}[\text{cost}(T)] = O(d \log \Delta) \cdot \text{cost}(T^*)$ .*

**Proof :** For each edge  $(u, v) \in T^*$  we have that the expected cost of the path connecting  $u$  and  $v$  in  $T$  is  $O(d \log \Delta) \cdot d(u, v)$ . The expected weight of all edges in  $E = \bigcup_{(u,v) \in T^*} d_T(u, v)$  is  $O(d \log \Delta) \cdot \text{cost}(T^*)$ . The graph  $(P, E)$  contains all edges of  $T$ .  $\square$

### 5.2.1 Approximation of $\text{cost}(T)$ in Dynamic Data Streams

Let us use  $n_i$  to denote the number of non-empty grid cells in grid  $G_i$ . We set  $n_0 = |P|$ .

**Lemma 5.2.2**

$$\text{cost}(T) = \sum_{i=0}^{\log \Delta} 2^i \cdot n_i .$$

Let  $G_i(p)$  be the cell in  $G_i$  that contains  $p$ . Then we use for each level  $i$  a data structure  $DE_i$  that counts the number of distinct non-empty cells in  $G_i$ .

INSERT( $p$ )

**for each** level  $i$  **do**

    insert cell  $G_i(p)$  in  $DE_i$

DELETE( $p$ )

**for each** level  $i$  **do**

    delete cell  $G_i(p)$  from  $DE_i$

Each distinct elements data structure requires  $O(\log \Delta)$  space.

**Theorem 20** *There is an algorithm that computes a  $O(d \log \Delta)$ -approximation of the weight of the Euclidean minimum spanning of the current point set in a dynamic geometric data stream. The algorithm requires  $O(d \log^2 \Delta)$  bits of memory.*

## 5.3 Cost of Minimum Weighted Matching

The next application of our technique will be to the minimum weighted matching problem. The goal is to find a perfect matching with minimum cost in the complete Euclidean graph, i.e. the complete graph over the point set  $P$  whose edges are weighted with the Euclidean distance between its vertices. We will assume that  $P$  is even.

For a given matching  $M$  we use  $\text{cost}(P, M) = \sum_{(p,q) \in M} d(p, q)$  to denote the cost of  $M$ . Like in the previous section we can probabilistically construct a 2-HST  $T$  for  $P$ . We use  $\text{cost}_T(P, M) = \sum_{(p,q) \in M} d_T(p, q)$  to denote the cost of  $M$  when the distance between two

points is given by the tree metric induced by  $T$ . We know that  $(P, d_T)$  dominates  $(P, d)$  and so we have  $cost_T(P, M) \geq cost(P, M)$ . Further we have

$$\begin{aligned}
\mathbf{E}[cost_T(P, Opt)] &= \mathbf{E}\left[\sum_{(p,q) \in M} d_T(p, q)\right] \\
&= \sum_{(p,q) \in M} \mathbf{E}[d_T(p, q)] \\
&\leq \sum_{(p,q) \in M} O(d \log \Delta) \cdot d(p, q) \\
&= O(d \log \Delta) \cdot cost(P, Opt) ,
\end{aligned}$$

where  $Opt$  denotes an optimal matching. Hence, we know that the cost of an optimal matching in  $(P, d_T)$  is a  $O(d \log \Delta)$ -approximations of an optimal matching in  $(P, d)$ . We will now study optimal matching in  $(P, d_T)$ .

**Claim 5.3.1** *The cost of an optimal minimum weighted matching of  $(P, d_T)$  is*

$$n + \sum_{i \geq 1} \sqrt{d} \cdot 2^{i-1} \cdot m_i ,$$

where  $m_i$  is the number of nodes  $v$  in level  $i$  of the tree such that the subtree rooted at  $v$  has an odd number of leaves.

**Proof:** We first prove that the cost of an optimal matching is at least  $n + \sum_{i \geq 1} \sqrt{d} \cdot 2^{i-1} \cdot m_i$ . We need  $n$  to connect all leaves of the tree to level 1. Let  $v$  be an interior node at level  $i$  with an odd number of leaves in the subtree  $T(v)$  rooted at  $v$ . At least one leaf of  $T(v)$  must be matched to a leaf outside of  $T(v)$ . This causes cost within  $T(v)$  of  $\sqrt{d} \cdot 2^{i-1}$ . Since we are not counting any cost twice, the lower bound follows.

Now we construct the upper bound, i.e. we give an algorithm that constructs a matching with cost  $n + \sum_{i \geq 1} \sqrt{d} \cdot 2^{i-1} \cdot m_i$ . We greedily match as many nodes  $v, w$  whose shortest path runs only through levels 0 and 1 of the tree. Then we match vertices whose shortest path run through levels 0 upto 2, and so on. The overall number of unmatched vertices per level is  $m_i$ . For these vertices we have to pay the cost of  $\sqrt{d} \cdot 2^{i-1}$  to connect them to the outside of the subtree (the cost upto level  $i$  has been payed in the previous rounds). Summing up the costs gives a matching of cost  $n + \sum_{i \geq 1} \sqrt{d} \cdot 2^{i-1} \cdot m_i$ .  $\square$

In order to develop a streaming algorithm for this problem we have to estimate the  $m_i$ . We do this by solving the following problem over data stream. We are given a vector  $x[1, \dots, M]$  and we would like to perform the operations INSERT, DELETE and ODDCOUNT, where INSERT is defined as  $x[i] = x[i] + 1$  and DELETE is defined as  $x[i] = x[i] - 1$ . In our scenario we will use  $M = \Delta^d$ . Thus, we can interpret  $x$  as characteristic vector over the input space and so over point set  $P$  corresponds to such a vector. We will assume that  $0 \leq x[i] \leq 1$  during the course of the algorithm. ODDCOUNT must return an approximation of the number OC of entries  $i$  such that  $x[i]$  is odd. We will develop a constant factor approximation algorithm for this problem.

We consider the following decision problem, which implies a constant factor approximation algorithm by running the decision algorithm for  $T = 1, 2, 4, \dots, \Delta^d$  in parallel.

**The Decision Problem.** Let  $T \in \{1, \dots, n\}$ , where  $n = |P|$ . We require that

- the algorithm accepts with probability greater than  $1/3$ , if  $OC > T$ , and
- the algorithm accepts with probability less than  $1/10$ , if  $OC < T/10$ .

Our algorithm just needs one bit of memory plus the space required to store the hash function.

INIT

Choose ideal hash function  $h : \{1, \dots, M\} \rightarrow \{1, \dots, T\}$   
 $s \leftarrow 0$

INSERT( $i$ )

**if**  $h(i) = 1$  **then**  $s \leftarrow s + 1 \pmod 2$

DELETE( $i$ )

**if**  $h(i) = 1$  **then**  $s \leftarrow s - 1 \pmod 2$

ODDCOUNT

**if**  $s = 1$  **then** accept  
**else** reject

**Claim 5.3.2** ODDCOUNT *satisfies the two conditions of the decision problem.*

**Proof :** Recall that every entry in  $x[i]$  is either 0 or 1 since  $x$  is a characteristic vector. Let us first consider the case that  $x$  has exactly  $T$  odd entries (ones). Let  $R$  be the set of the indices of these ones. We have

$$\Pr[h^{-1}(R) = 1] = \frac{T \cdot (T-1)^{T-1}}{T^T} = \frac{T}{T} \cdot \left(1 - \frac{1}{T}\right)^{T-1} .$$

For  $T \geq 2$  ( $T = 1$  always works) we get

$$1/2 \geq \left(1 - \frac{1}{T}\right)^{T-1} \geq 1/e .$$

If there are more than  $T$  ones, then we can fix the 'superfluous' ones and obtain that the probability of acceptance is between  $1/2$  and  $1/e$  or  $1 - 1/2$  and  $1 - 1/e$ , which is always greater than  $1/3$ .

To prove the second condition we can assume that the number of ones is less than  $T/10$ . Again, let  $R$  denote the set of indices of the ones in  $x$ . We have

$$\Pr[|h^{-1}(R)| \geq 1] \leq T/10 \cdot 1/T .$$

□

It remains to assemble things. We need to solve  $O(\log \Delta)$  instances of the ODDCOUNT problem (one for each level of the tree  $T$ ). To solve the approximation version of the problem we have to solve  $O(d \log \Delta)$  instances of the decision problem. To ensure that all these instances work correctly with constant probability, we need an error probability of at most  $1/O(d \log^2 \Delta)$ . This can be achieved by running  $O(\log d \log \log \Delta)$  copies of the ODDCOUNT data structure to ensure and accept, if at least  $1/2$  of the calls accept.

**Theorem 21** *The cost of a minimum weighted matching in a data stream can be approximated with  $O(d \log d \cdot \log^2 \Delta \cdot \log \log \Delta)$  bits plus the space required to store the  $O(d \log d \cdot \log^2 \Delta \cdot \log \log \Delta)$  hash functions.*

## 5.4 Facility Location Cost

Another fundamental problem over geometric data streams is the facility location problem. This problem models the following scenario. We have a set of customers at certain locations and these customers demand a certain commodity. We can open facilities at all locations to serve all customers. However, opening a facility has a certain cost  $f$ . So, it would be cheapest to open only one facility. But in this case, we have to ship the commodity to all the customers. This incurs another cost per customer, which is proportional to the distance of the customer to the nearest open facility. Our goal is to minimize the sum of the opening cost and the cost of serving the customers.

**Definition 5.4.1 (Facility Location Problem)** *We are given a metric space  $(P, d)$  and an opening cost  $f$ . Our goal is to find a set  $F \subseteq P$  of facilities such that*

$$\text{cost}_f(P, d, F) := f \cdot |F| + \sum_{p \in P} \min_{q \in F} d(p, q)$$

*is minimized. We define*

$$\text{cost}_f(P, d) := \min_{F \subseteq P} \text{cost}_f(P, d, F) .$$

In the streaming version of the problem we will again assume that the points come from the discrete space  $\{1, \dots, \Delta\}^d$  and the distance measure will be the Euclidean distance, i.e.  $d(p, q) = \|p - q\|_2$ . Thus, we consider only the Euclidean version of the facility location problem.

### 5.4.1 Reduction to HSTs

Like in the previous problems our first step is to reduce the Euclidean version of the problem to a problem on a 2-HST. We are using the same embedding into a tree metric as in the previous sections. Let  $d_T(p, q)$  denote the distance between  $p$  and  $q$  in this tree metric and let  $d(p, q) := \|p - q\|_2$  denote the distance between  $p$  and  $q$  in the Euclidean metric. By the properties of the embedding we know that  $d_T(p, q) \geq d(p, q)$ , i.e. the embedding is non-contracting. Further, we have  $\mathbf{E}[d_T(p, q)] = O(d \cdot \log \Delta) \cdot d(p, q)$ .

Now let us fix an optimal solution  $F_{\text{opt}}$  for an instance of the facility location problem. For each customer  $p$ , let us use  $r(p)$  to denote the nearest open facility. The cost of this solution is given by

$$\begin{aligned}
O(d \log \Delta) \cdot \text{cost}_f(P, d, F_{\text{opt}}) &= O(d \log \Delta) \cdot \left( f \cdot |F_{\text{opt}}| + \sum_{p \in P} \min_{q \in F_{\text{opt}}} d(p, q) \right) \\
&= O(d \log \Delta) \cdot \left( f \cdot |F_{\text{opt}}| + \sum_{p \in P} d(p, r(p)) \right) \\
&\geq f \cdot |F_{\text{opt}}| + O(d \log \Delta) \cdot \sum_{p \in P} d(p, r(p)) \\
&= f \cdot |F_{\text{opt}}| + \sum_{p \in P} \mathbf{E}[d_{\mathbb{T}}(p, r(p))] \\
&\geq f \cdot |F_{\text{opt}}| + \sum_{p \in P} \mathbf{E}[d_{\mathbb{T}}(p, r(p))] \\
&\geq f \cdot |F_{\text{opt}}| + \sum_{p \in P} \mathbf{E}[\min_{q \in F_{\text{opt}}} d_{\mathbb{T}}(p, q)] \\
&\geq \mathbf{E}[f \cdot |F_{\text{opt}}| + \sum_{p \in P} \min_{q \in F_{\text{opt}}} d_{\mathbb{T}}(p, q)] \\
&= \mathbf{E}[\text{cost}_f(P, d_{\mathbb{T}}, F_{\text{opt}})] .
\end{aligned}$$

Thus, we know that the expected cost with respect to  $d_{\mathbb{T}}$  of an optimal solution (w.r.t.  $d$ ) is at most  $O(d \log \Delta)$  times the optimal cost for  $d$ . Further we know that the embedding is non-contracting and so any solution for  $d_{\mathbb{T}}$  must be at least as expensive as the corresponding solution for  $d$ . Thus, we know that

$$\text{cost}_f(P, d) \leq \text{cost}_f(P, d_{\mathbb{T}}) \leq O(d \log \Delta) \cdot \text{cost}_f(P, d) .$$

### 5.4.2 Approximating the Cost of Facility Location in HSTs

Our next step is to find an approximation for the cost of facility location when the point set is given as a 2-HST  $\mathbb{T}$ . In the following we will assume that we are allowed to place facilities at the internal nodes of the tree. For any such solution, there is a solution of at most twice its cost that places all facilities at the leaves, i.e. points in  $P$ . This can be seen as follows. For every internal node  $v$  that has an open facility, we open a facility at one of the leaves of the subtree rooted at  $v$ . Since all leaves have the same distance to  $v$ , the distance to the nearest facility increases by at most 2. Thus, we open the same number of facilities and have at most twice the connection cost.

Let  $\mathbb{T}_i$  denote the set of nodes of  $\mathbb{T}$  at level  $i$ . Let  $|\mathbb{T}(v)|$  be the number of leaves of the subtree  $\mathbb{T}(v)$  rooted at  $v$ . We will show the following claim.

**Claim 5.4.2** *Let  $C$  be the minimum facility location cost for a 2-HST  $\mathbb{T}$ , where we allow to place facilities at internal nodes of the tree. Let*

$$Q = \sum_{i=1}^{\log \Delta} \sum_{v \in \mathbb{T}_i} \min |\mathbb{T}(v)| \cdot \sqrt{d} \cdot 2^{i-1}, f .$$

Then

$$C \leq Q + f \leq C \cdot \log \Delta + f .$$

**Proof :** To prove the first inequality we just have to show that there exists a solution with cost  $Q + f$ . Such a solution can be obtained as follows.

- One facility is opened at the root of the HST.
- For each node  $v \in T_i$  place a facility at  $v$ , if  $|T(v)| \cdot \sqrt{d}2^{i-1} \geq f$ .

The cost of the solution is  $Q + f$ . This can be seen as follows. We use  $f$  to pay for the facility at the root. At every internal node we either open a facility or  $|T(v)| \cdot 2^i$  equals the cost of using the edge for  $v$  to its parent for all points in the subtree  $T(v)$ . It follows that for each point we pay the cost to the nearest facility on the path to the root. Hence, the cost of the proposed solution is at most  $Q + f$ .

To prove the second inequality we only have to show that the cost for every level of the tree is at most  $C$ , i.e.

$$\sum_{v \in T_i} \min\{|T(v)| \cdot \sqrt{d}2^{i-1}, f\} \leq C .$$

Let  $F$  be a set of facilities that achieved cost  $C$ . Let us consider an arbitrary node  $v \in T_i$ . If the subtree  $T(v)$  contains an (open) facility from  $F$ , then this adds  $f$  to the cost of  $C$ . Otherwise, the nodes in  $T(v)$  must be connected to a facility outside of  $T(v)$ . This adds a cost of  $|T(v)| \cdot \sqrt{d} \cdot 2^{i-1}$ .  $\square$

It remains to design an algorithm to approximate  $Q$ . Our approach will be to approximate  $\sum_{v \in T_i} \min\{|T(v)| \cdot \sqrt{d} \cdot 2^{i-1}, f\}$  for each level  $i$  separately. In order to do so, it suffices to estimate  $\sum_{v \in T_i} \min\{|T(v)|, f/(\sqrt{d}2^{i-1})\}$ . If  $f/(\sqrt{d}2^{i-1})$  is smaller than 1 we can use an algorithm to estimate the number of distinct elements in a stream to estimate  $|T_i|$ . Otherwise, we can assume that  $f/(\sqrt{d}2^i)$  is integral as this changes the cost by at most a constant factor. Let us define  $M := f/(\sqrt{d}2^i)$ . Thus we would like to approximate  $\sum_{v \in T_i} \min\{|T(v)|, M\}$ , i.e. for each cell in level  $i$  we count the minimum of  $M$  and the number of points contained in the cell. The basic idea of the algorithm is very simple. If we select each element from the input stream with probability  $1/M$  then any cell that contains at least  $|T(v)|$  points will contain at least one point in expectation. Any cell that contains less than  $M$  points will contain a sample point with probability proportional to the number of points inside the cell. Thus, we can simply count the number of non-empty cells using a distinct elements data structure.

The approach is implemented as follows. We use a hash function  $h$  that maps the input points to  $\{1, \dots, M\}$  and we only consider operations of points that are mapped to 1. In other words, we are taking a sample from the input stream and select each point with probability  $1/M$ .

BCINSERT( $p$ )

**if**  $h(p) = 1$  **then**

determine the grid cell that contains  $p$

insert the grid cell into the distinct elements data structure

BCDELETE( $p$ )

**if**  $h(p) = 1$  **then**

    determine the grid cell that contains  $p$

    delete the grid cell from the distinct elements data structure

BCREPORT( $p$ )

  Let  $x$  be the number of elements reported by the distinct elements data structure

**return**  $Mx$

### 5.4.3 Analysis

It remains to show that  $Mx$  is a good estimator. Let  $K$  denote the number of distinct non-empty grid cells after the sampling, i.e. the value that is estimated by the distinct elements data structure.

**Claim 5.4.3** *Let  $p(k) = 1 - (1 - 1/M)^k$  be the probability that at least one element is selected, if we select each element from a set of  $k$  elements independently at random with probability  $1/M$ . We have*

$$\mathbf{E}[K] = \sum_{v \in T_i} p(|T(v)|) .$$

**Proof :**  $T(v)$  denotes the number of cells stored in subtree  $T(v)$ . □

**Claim 5.4.4** *For any  $k \geq 0$  we have*

$$\min\{k, M\}/2 \leq M \cdot p(k) \leq \min\{k, M\} .$$

**Proof :** To prove the first inequality observe that  $p(k)$  is concave for  $k \geq 0$ . Further we have  $p(0) = 0$  and  $p(M) = 1 - (1 - 1/M)^M > 1 - 1/e < 1/2$ . Hence,  $p(k) \geq k/2$  for  $0 \leq k \leq M$ , which proves the first inequality. The second inequality follows from  $p(k) \leq 1$  and  $p(k) \leq k/M$  by the union bound. □

The last two claims imply that

$$\sum_{v \in T_i} \min\{|T(v)|, M\}/2 \leq M \cdot \mathbf{E}[K] \leq \sum_{v \in T_i} \min\{|T(v)|, M\} .$$

Thus it remains to show that we also have a small variance. Then we can apply Chebyshev's inequality to obtain a sharp concentration by repeating the experiment a sufficient number of times.

$$\mathbf{Var}[K] \leq \sum_{v \in T_i} p(|T(v)|) = \mathbf{E}[K] .$$

If  $\sum_{v \in T_i} \min\{|T(v)| \cdot \sqrt{d} \cdot 2^{i-1}, f\} \geq M$  then  $\mathbf{E}[K] = \Omega(1)$ . In this case, we can repeat the experiment  $s$  times to obtain random variables  $K_1, \dots, K_s$ . Our output value will be

$$\frac{1}{s} \sum_{i=1}^s K_i$$

which has expectation  $\mathbf{E}[K]$  and variance at most  $\mathbf{E}[K]/s$ . By Chebyshev's inequality we get

$$\Pr\left[\left|\frac{1}{s} \sum_{i=1}^s K_i - \mathbf{E}\left[\frac{1}{s} \sum_{i=1}^s K_i\right]\right| \geq \frac{1}{2} \cdot \mathbf{E}\left[\frac{1}{s} \sum_{i=1}^s K_i\right]\right] \leq \frac{4 \cdot \mathbf{Var}[K]}{\mathbf{E}[K]^2} \leq \frac{4}{s \cdot \mathbf{E}[K]}$$

Thus, for  $s \geq 32 \log \Delta$  we get a confidence probability of  $1 - 1/(8 \log \Delta)$ . Hence, with probability  $7/8$  all data structures work for all levels  $i$ . Finally, we observe that for  $\sum_{v \in T_i} \min\{|T(v)| \cdot \sqrt{d} \cdot 2^{i-1}, f\} < M$  we can solve the problem exactly.

**Theorem 22** *There is a  $O(d \log^2 \Delta)$ -approximation algorithm for computing the weight of the minimum facility location problem of a dynamic geometric data stream. The space requirement of the algorithm is determined by  $O(\log^2 \Delta)$  independent copies of a distinct elements data structure plus the space for the hash functions.*



## 6 Density Sampling

In the next chapter we will consider a different technique to approximate problems over dynamic data streams: *Density Sampling*. Like embedding into tree metrics, density sampling obtains information about the input point set by considering nested grids with side length  $1, 2, \dots, \Delta$ . We will use  $G_i$  to denote a grid with side length  $2^i$ . In each grid we try to maintain the  $L$  cells that contain the largest number of points for some small value  $L$ . This means that in coarse grids we essentially maintain information about every non-empty cell. As the grids become finer the number of non-empty cells increases and we can only keep track of the cells that contain the most points. We call these cells *heavy cells*. In the streaming algorithm we will identify these heavy cells by using a random sample. The main trick will be to show that we can use samples much larger than the space requirement of our algorithm because all sample points will lie in a few distinct grid cells.

The number of sample points inside every cell tells us which cells are heavy and gives us a rough estimation about the number of points from  $P$  contained in it. Finally, we combine the cells and the corresponding estimates from the different grids to one approximation for the distribution of points. Here the idea is simple. If we have a cell in a larger grid that contains at least one heavy cell, we replace it by the corresponding  $2^d$  cells from the next finer grid.

In some way, density sampling tries to obtain as much information as possible from every grid layer and then combine this information in a single summary. As we will see later, this summary contains enough information to solve a number of interesting problems in the dynamic streaming model. The approach will be to construct a weighted point set by putting a point into each non-empty grid cell obtained by the density sampling. The weight of the point is the estimated number of points in the cell. For a number of problems, it will turn out that solving the problem on this new point set is approximately similar as solving it on the original point set. Thus, the new weighted point set is a coresset in the sense of Section 4.

For which problems we can apply density sampling and how are the details? For the first question, there is the following rule of thumb: A problem can be solved using density sampling, if

- (a) moving a point by a distance of  $D$  changes the cost of a solution by  $O(D)$ , and,
- (b) there exists a point  $q$  such that the cost of an optimal solution is  $\Theta(\sum_{p \in P} \|p - q\|_2)$ .

As we will see, condition (a) ensure that the quality of approximation will be good and condition (b) takes care of the space complexity. We remark that the characterization above is by far not complete. For example, we can apply density sampling to the  $k$ -means problem. However, it gives a good first intuition what is going on.

## 6.1 Density sampling

We first develop the density sampling algorithm and then show how it can be used to compute a coresets for the 1-Median problem. As usual in the dynamic geometric setting, let  $P$  be a point set from  $\{1, \dots, \Delta\}^d$ . We consider nested grids  $G_i$  with cell width  $2^i$  and assume that no input point lies on a grid point, i.e. in grid  $G_0$  each cell contains at most one point. The largest grid is  $G_{\log \Delta + 1}$ , where we assume that  $\Delta$  is a power of 2. We start with a version of the algorithm that cannot be implemented in the streaming scenario, because it uses too much space. Then we show how this algorithm can be implemented in streaming. The first version of the algorithm given below uses parameter  $L$  besides the point set  $P$ . This parameter is related to quality of approximation and space requirement of the algorithm and will be determined later.

DENSITY SAMPLING ( $P, L$ )

Let  $C$  be a bounding cube of  $P$  in grid  $G_{\log \Delta + 1}$

$i \leftarrow \log \Delta + 1; T_i \leftarrow 1; \mathcal{H}_i \leftarrow \{C\}$

**repeat**

Let  $\mathcal{C}_{i-1}$  be the set of cells of grid  $G_{i-1}$  that are contained in some cells of  $\mathcal{H}_i$

$T_{i-1} \leftarrow T_i$

**repeat**

for each cell  $C \in \mathcal{C}_{i-1}$  let  $n_C$  be the number of points in  $C$

Let  $\mathcal{H}_{i-1}$  be the set of cells in  $\mathcal{C}_{i-1}$  that contain at least  $T_{i-1}$  points

**if**  $|\mathcal{H}_{i-1}| > L$  **then**  $T_{i-1} \leftarrow 2 \cdot T_{i-1}$

**until**  $|\mathcal{H}_{i-1}| \leq L$

$\mathcal{L}_{i-1} \leftarrow \mathcal{C}_{i-1} \setminus \mathcal{H}_{i-1}$

$i \leftarrow i - 1$

**until**  $\mathcal{H}_i = \emptyset$  or  $i = 0$

The algorithm maintains the sets  $\mathcal{H}_i$  and  $\mathcal{L}_i$  and for each cell  $C \in \bigcup (\mathcal{H}_i \cup \mathcal{L}_i)$  it also maintains the value  $n_C$ . The union of this information is maintained as a summary of the data. We will call the cells in  $\mathcal{H}_i$  heavy cells and the cells in  $\mathcal{L}_i$  light cells.

**Claim 6.1.1** *The summary defined above can be stored  $O(\log \Delta \cdot L)$  space.*

**Proof :** In each grid we have at most  $L$  cells in  $\mathcal{H}_i$ . Therefore, in  $\mathcal{L}_{i-1}$  we have at most  $2^d \cdot L$  cells. Storing each cell requires  $O(1)$  space. Since we consider  $d$  to be constant the space bound follows.  $\square$

### 6.1.1 A Coresets for the 1-Median Problem

Let us recall that the 1-Median problem is to find the point  $q$  that minimizes  $cost(P, q) := \sum_{p \in P} \|p - q\|_2$ . Obviously, this problem satisfies conditions (a) and (b) from above. Also, recall that an  $\epsilon$ -coresets for the 1-Median problem is a set of points  $S$  with point weights  $w(p)$  for each  $p \in P$  such that for every point  $q \in \mathbb{R}^d$

$$(1 - \epsilon)cost(P, q) \leq cost(S, q) \leq (1 + \epsilon) \cdot cost(P, q) ,$$

where  $cost(S, q) := \sum_{p \in S} w(p) \cdot \|p - q\|_2$ .

We show that if we obtain the sets  $\mathcal{H}_i$  and  $\mathcal{L}_i$  using the above algorithm for sufficiently large value of  $L$  one can construct a coreset  $S$  from the maintained summaries in the following way.

If  $\mathcal{H}_0 \neq \emptyset$  then we have  $T_0 = 1$ , so  $\mathcal{L}_i = \emptyset$  for all  $i$ . Each cell in  $\mathcal{H}_0$  contains a unique grid point and we will use these grid points (with unit weight) as a coreset for  $P$ . In this case  $S = P$  and so  $S$  will clearly be a coreset.

If  $\mathcal{H}_0 = \emptyset$  then the set  $S$  is obtained by replacing every cell  $C \in \mathcal{L}_i$  with an arbitrary point inside  $C$  and assigning a weight of  $n_C$  to it is an  $\epsilon$ -coreset. We will interpret this process as moving each point in cell  $\mathcal{L}_i$  to the corresponding point from  $S$ . We have to show that the overall movement of points is at most  $\epsilon \cdot cost(P, q_{opt})$ , where  $q_{opt}$  is an optimal 1-Median for  $P$ . We need the following definition.

**Definition 6.1.2** *Let  $q_{opt}$  be an optimal 1-Median for  $P$ . A cell  $C$  in grid  $G_i$  is called distant, if it has distance from  $q_{opt}$  of more than  $2\sqrt{d} \cdot 2^i/\epsilon$ . Otherwise, it is called near.*

Next, we show that there are only few near cells.

**Claim 6.1.3** *There are at most  $(\frac{8\sqrt{d}}{\epsilon})^d$  near cells.*

**Proof :** There is a cube of side length  $8\sqrt{d}/\epsilon \cdot 2^i$  that contains all near cells. This cube has volume  $(8\sqrt{d}/\epsilon)^d \cdot 2^{id}$ . Each cell of grid  $G_i$  has side length  $2^i$  and volume  $2^{id}$ . Hence there can be at most  $(\frac{8\sqrt{d}}{\epsilon})^d$  near cells.  $\square$

Let  $\mathcal{L}_i^{near}$  be the subset of cells in  $\mathcal{L}_i$  that are near. Let  $\mathcal{L}_i^{distant}$  be the subset of cells in  $\mathcal{L}_i$  that are distant. For the analysis, we consider near and distant cells separately and show that the movement of points inside each type of cells is at most  $\frac{\epsilon}{2} \cdot cost(P, q_{opt})$ .

## Lower Bound

We recall that, by the triangle inequality, moving a point by a distance of  $D$  changes the cost of the 1-Median problem by at most  $D$ . Hence, moving the points to the center of a cell in grid  $G_i$  changes the cost of the 1-Median problem by at most  $\sqrt{d} \cdot 2^i \cdot T_i$ . Let  $i_{max}$  be a level such that  $\sqrt{d} \cdot 2^i \cdot T_i$  is maximized.

**Claim 6.1.4 (Lower Bound)** *Let  $q_{opt}$  be an optimal 1-Median for  $P$ . Then*

$$cost(P, q_{opt}) \geq T_{i_{max}} \cdot 2^{i_{max}-1} \cdot (L - (8\sqrt{d})^d/\epsilon^d)/\epsilon .$$

**Proof :** Since  $i_{max}$  is a level that maximizes  $\sqrt{d} \cdot 2^i \cdot T_i$ , we know that  $T_{i_{max}} > T_{i_{max}+1}$ . Hence, there are more than  $L$  cells in grid  $G_{i_{max}}$  that contain  $T_{i_{max}}/2$  points. At least  $(L - (8\sqrt{d})^d/\epsilon^d)$  of these cells must be distant and they contribute with  $T_{i_{max}}/2 \cdot 2^{i_{max}}/\epsilon$  to  $cost(P, q_{opt})$ . Hence the claim follows.  $\square$

## Near Cells

It remains to prove that the overall movement cost for points in near cells is at most  $T_{i_{\max}} \cdot 2^{i_{\max}-1} \cdot (L - \sqrt{d})8^d/\epsilon^d$  for  $L$  being large enough. We have

$$\sum_i |\mathcal{L}_i^{\text{near}}| \cdot \sqrt{d} \cdot 2^i \cdot T_i \leq (\log \Delta + 2) \cdot (8\sqrt{d}/\epsilon)^d \cdot \sqrt{d} \cdot 2^{i_{\max}} \cdot T_{i_{\max}} .$$

Using the lower bound from above we obtain that for  $L \geq 2(\log \Delta + 3) \cdot \sqrt{d} \cdot (8\sqrt{d}/\epsilon)^d = O(\log \Delta/\epsilon^d)$  we have

$$(\log \Delta + 2) \cdot \sqrt{d} \cdot (8\sqrt{d}/\epsilon)^d \cdot \sqrt{d} \cdot 2^{i_{\max}} \cdot T_{i_{\max}} \leq \frac{\epsilon}{2} \cdot \text{cost}(P, q_{\text{opt}}) .$$

## Distant Cells

For distant cells we use the fact that every point in a distant cell contributes at least  $2\sqrt{d} \cdot 2^i/\epsilon$  to the cost of an optimal solution. For a cell  $C$  let  $|C|$  denote the number of points in  $C$ . We obtain

$$\sum_i \sum_{C \in \mathcal{L}_i^{\text{distant}}} |C| \cdot \sqrt{d} \cdot 2^i \leq \frac{\epsilon}{2} \cdot \sum_i \sum_{C \in \mathcal{L}_i^{\text{distant}}} 2\sqrt{d} \cdot 2^i/\epsilon \leq \frac{\epsilon}{2} \cdot \text{cost}(P, q_{\text{max}}) .$$

Thus, the overall movement of points is at most  $\epsilon \cdot \text{cost}(P, q_{\text{opt}})$ . We can summarize our findings in the following theorem.

**Theorem 23** For  $L \geq 2(\log \Delta + 3) \cdot \sqrt{d} \cdot (8\sqrt{d}/\epsilon)^d$  the set  $S$  obtained by the above process is an  $\epsilon$ -coreset. The size of the coreset is  $O(\log \Delta \cdot L) = O(\log^2 \Delta/\epsilon^d)$ .

### 6.1.2 Developing the Streaming Algorithm

Our next step will be to develop a streaming version of the above algorithm. The basic idea is to identify the heavy cells using a random sample. However, a straightforward approach will not work. The problem is as follows. In the coarsest grids the threshold  $T_i$  can be as small as 1. In other words, every non-empty cell is heavy and so our sample must contain all the points. However, for problems satisfying condition (b) we can use a similar argument as in the analysis of the 1-median problem above to show that the overall number of points in distant cells is not too large, i.e. a properly sized random sample will only hit few of them. Hence, all but a very few sample points will lie in near cells. As a consequence, we can store all cells that contain a sample point. This is, what the following algorithm does.

To take a random sample from the data stream we use hash functions  $h_j$  that map the points to the number  $\{1, \dots, 2^j\}$  and consider all points that are mapped to 1 as sample points, i.e. using a hash function  $h_j$  corresponds to taking a sample  $S$  where each element is chosen with probability equal to  $1/2^j$ . Now let us consider grid  $G_i$ . To deal with the fact that the sample may be large, we only try to remember the different non-empty cells and the number of points inside them. Thus, whenever a point arrives that is contained in the sample set, we compute to which cell in  $G_i$  it belongs and then we insert this cell into a data structure that keep track

of the distinct cells and their count. For the latter purpose we use a  $k$ -set data structure (with parameter  $k := L'$ , therefore we call it  $L'$ -set data structure in the remainder of this chapter), that satisfies the following conditions.

- If the number of distinct elements is at most  $L'$  then the data structure return with high probability all distinct elements as well as their individual counts.
- If there are more than  $L'$  distinct elements, the data structure return 'failure' with high probability.

Using these two changes we can implement insertions and deletions as follows. To initialize our data structure we choose  $O(\log^2 \Delta)$  hash functions  $h_{i,j}$  and  $L'$ -set data structures  $\text{Set}_{i,j,L'}$  for  $0 \leq i \leq \log \Delta + 1$  and  $0 \leq j \leq d \log \Delta$ . Upon arrival of a point  $p$  we call the procedure **SAMPLINGINSERT** described below for all values of  $i$  and  $j$ .

**SAMPLINGINSERT**( $i, j, p$ )

```

if  $h_{i,j}(p) = 1$  then
  let  $C$  be the cell in grid  $G_i$  that contains  $p$ 
   $\text{Set}_{i,j,L'}$ .Insert( $C$ )

```

**SAMPLINGDELETE**( $i, j, p$ )

```

if  $h_{i,j}(p) = 1$  then
  let  $C$  be the cell in grid  $G_i$  that contains  $p$ 
   $\text{Set}_{i,j,L'}$ .Delete( $C$ )

```

Now we can also describe the streaming version of density sampling. The algorithm uses a value  $s_0$ , which is the smallest threshold at which we begin to use random sampling and which is assumed to be a power of 2. The value of  $s_0$  will be  $(\frac{\log \Delta}{\epsilon})^{O(1)}$ .

**STREAMINGDENSITYSAMPLING** ( $P, L', s_0$ )

```

Let  $C$  be a bounding cube of  $P$  in grid  $G_{\log \Delta + 1}$ 
 $j \leftarrow 0$ ;  $i \leftarrow \log \Delta + 1$ ;  $T_i \leftarrow 1$ ;  $\mathcal{H}_i \leftarrow \{C\}$ 
repeat
  while  $\text{Set}_{i,j,L'}$ .Report() = failure and  $j \leq d \log \Delta$  do
    if  $T_i < s_0$  then  $T_i \leftarrow 2T_i$ 
    else  $j \leftarrow j + 1$ 
  if  $\text{Set}_{i,j,L'}$ .Report() = failure then exit
  Let  $\mathcal{C}_{i-1}$  be the set of grid cells contained in some cell of  $\mathcal{H}_i$ 
   $\mathcal{D}_{i-1} \leftarrow \text{Set}_{i,j,L'}$ .Report()
  for each cell  $C \in \mathcal{C}_{i-1}$  do
    if  $C \in \mathcal{D}_{i-1}$  then  $n_C \leftarrow 2^j \cdot \text{Set}_{i,j}$ .Count( $C$ ) else  $n_C = 0$ 
    let  $\mathcal{H}_{i-1}$  be the set of cells in  $\mathcal{C}_{i-1}$  with  $n_C \geq 2^j \cdot T_i$ 
     $\mathcal{L}_{i-1} \leftarrow \mathcal{C}_{i-1} \setminus \mathcal{H}_{i-1}$ 
     $i \leftarrow i - 1$ 
     $T_i \leftarrow T_{i+1}$ 
until  $\mathcal{H}_i = \emptyset$  or  $i = 0$ 

```

### 6.1.3 Coreset Construction

The process to construct a coreset is a slightly modified version of the construction for the non-streaming version. If  $T_0 < s_0$  then for every cell we have the exact number of points. If  $L' \geq s_0 \cdot L$  then we can apply a similar analysis as in the previous section to show that the previous coreset construction gives an  $\epsilon$ -coreset.

Otherwise, we use the following modified construction. For each light cell  $C$  in set  $\mathcal{C}_i$  we check whether its parent cell  $H$  contains some heavy cell as a subcell, i.e. whether a subcell of  $H$  is contained in  $\mathcal{H}_i$ . If this is the case, we add a point in the center of  $C$  to the coreset and assign weight  $n_C$  to it. Otherwise,  $H$  contains no subcell in  $\mathcal{H}_i$ . In this case, we add a point in the center of  $H$  to the coreset and weight it with  $n_H$ .

This modification is needed, because it may happen that the value of  $j$  changes significantly when we move from  $i + 1$  to  $i$ . In this case, we approximate the points in  $C$  with only a relatively large additive error. In fact, this error may be much larger than  $\epsilon|H|$  and we do not know how to charge it in the analysis.

For the analysis we only prove that the change of cost introduced by the random sampling is relatively small, i.e. the difference between any point set that is within the sampling bounds and  $P$  is at most  $\epsilon \cdot \text{cost}(P, q_{opt})$ . Then we can apply our previous analysis to prove that the final point set is a coreset. We proceed similarly as in the non-streaming case and start with deriving a lower bound. After this, we continue the analysis by finding bounds for the change in cost in near cells and distant cells.

### 6.1.4 Overview of the Analysis

We now give a first overview of the analysis. It consists of five steps. We will show that with high probability

- every cell in  $\mathcal{H}_i$  has at least  $s_0 \cdot 2^j$  points,
- every cell with at least  $s_0 \cdot 2^{j+1}$  points is in  $\mathcal{H}_i$ ,
- the number of points in every cell in  $\mathcal{L}_i$  is approximated up to a relative error of  $\lambda \cdot s_0 \cdot 2^j$ ,
- if  $j$  is large enough, then with high probability for every cell that is hit by the sample set, we can determine the number of sample points in it, and
- if the above statements are satisfied, then the modified coreset construction gives a coreset.

### 6.1.5 Analysis

We will first show that for given  $j$ , if a cell contains a certain number of points, then it will be classified as heavy with high probability, i.e. the number of sample points in it is at least  $T = s_0$  (for smaller values of  $T$  we do not use sampling). We first observe that

**Lemma 6.1.5** Let  $S \subseteq P$  be a random sample, where each element is chosen with probability  $r = 1/2^j$ . Let  $\mathcal{C}$  be a cell with  $|\mathcal{C} \cap P| \geq s_0 \cdot 2^{j+1}$ . Then

$$\Pr[|S \cap \mathcal{C}| - \mathbf{E}[|S \cap \mathcal{C}|]| > \epsilon \mathbf{E}[|S \cap \mathcal{C}|] \leq \frac{1}{2\epsilon^2 \cdot s_0} .$$

**Proof :** Let  $X_p$  be the indicator random variable for the event that a given point  $p$  is taken into the sample set. Clearly,  $\mathbf{E}[X_p] = \frac{1}{2^j}$ . We have

$$\Pr[|S \cap \mathcal{C}| \geq s_0] = \Pr\left[\sum_{p \in \mathcal{C}} X_p \geq s_0\right] .$$

Since  $|\mathcal{C} \cap P| \geq s_0 \cdot 2^{j+1}$  we have

$$\mathbf{E}[|S \cap \mathcal{C}|] = |\mathcal{C} \cap P| \cdot \frac{1}{2^j} \geq 2 \cdot s_0 .$$

By pairwise independence of the  $X_p$  we have  $\mathbf{Var}[|S \cap \mathcal{C}|] = \mathbf{Var}[\sum_{p \in \mathcal{C}} X_p] \leq \frac{|\mathcal{C} \cap P|}{2^j}$ . Thus we can apply Chebyshev's inequality to obtain

$$\begin{aligned} \Pr[|S \cap \mathcal{C}| - \mathbf{E}[|S \cap \mathcal{C}|]| > \epsilon \cdot \mathbf{E}[|S \cap \mathcal{C}|] &\leq \frac{\mathbf{Var}[|S \cap \mathcal{C}|]}{\epsilon^2 \mathbf{E}[|S \cap \mathcal{C}|]^2} \\ &\leq \frac{1}{2\epsilon^2 s_0} \end{aligned}$$

□

**Corollary 6.1.6** Let  $S \subseteq P$  be a random sample, where each element is chosen with probability  $r = 1/2^j$ . Let  $\mathcal{C}$  be a cell with  $|\mathcal{C} \cap P| \geq s_0 \cdot 2^{j+1}$ . Then

$$\Pr[|S \cap \mathcal{C}| \geq s_0] \geq 1 - \frac{2}{s_0} .$$

A somewhat similar proof shows that the second statement is true with high probability.

**Lemma 6.1.7** Let  $S \subseteq P$  be a random sample, where each element is chosen with probability  $r = 1/2^j$ . Let  $\mathcal{C}$  be a cell with  $|\mathcal{C} \cap P| < s_0 \cdot 2^{j+1}$ . Then

$$\Pr[|S \cap \mathcal{C}| - \mathbf{E}[|S \cap \mathcal{C}|]| \geq \lambda \cdot s_0] \leq \frac{2}{\lambda^2 \cdot s_0} .$$

**Proof :** Let  $X_p$  be the indicator random variable for the event that a given point  $p$  is taken into the sample set. Again we have  $\mathbf{E}[X_p] = \frac{1}{2^j}$  and  $\Pr[|S \cap \mathcal{C}| \geq s_0] = \Pr[\sum_{p \in \mathcal{C}} X_p \geq s_0]$ . Since  $|\mathcal{C} \cap P| < s_0 \cdot 2^{j+1}$  we have

$$\mathbf{E}[|S \cap \mathcal{C}|] = |\mathcal{C} \cap P| \cdot \frac{1}{2^j} < 2 \cdot s_0 .$$

By pairwise independence of the  $X_p$  we have  $\mathbf{Var}[|S \cap \mathcal{C}|] = \mathbf{Var}[\sum_{p \in P \cap \mathcal{C}} X_p] \leq 2 \cdot s_0$ . Thus we can apply Chebyshev's inequality to obtain

$$\begin{aligned} \Pr[||S \cap \mathcal{C}| - \mathbf{E}[|S \cap \mathcal{C}|]| > \lambda \cdot s_0] &\leq \frac{2 \cdot s_0}{\lambda^2 \cdot s_0^2} \\ &\leq \frac{2}{\lambda^2 \cdot s_0} \end{aligned}$$

□

**Corollary 6.1.8** *Let  $S \subseteq P$  be a random sample, where each element is chosen with probability  $r = 1/2^i$ . Let  $\mathcal{C}$  be a cell with  $|\mathcal{C} \cap P| < s_0 \cdot 2^i$ . Then*

$$\Pr[||S \cap \mathcal{C}| - \mathbf{E}[|S \cap \mathcal{C}|]| \geq s_0] \leq \frac{2}{s_0} .$$

The two Lemmas above can be used to fix our choice of  $s_0$ . We would like to have that they apply to every cell in  $\mathcal{C}_i$ . We use the following claim.

**Claim 6.1.9** *If the data structures  $\text{Set}_{i,j,L'}$  work correctly then the number of cells in set  $\mathcal{H}_i$  is at most  $L'$ .*

**Proof :** If the data structures  $\text{Set}_{i,j,L'}$  work correctly, we know that if it does not return failure then our sample set intersects at most  $L'$  grid cells. Only these cells will obtain a value  $n_{\mathcal{C}} > 0$  and so only these cells can potentially be heavy cells. □

By the above claim, each  $\mathcal{C}_i$  contains at most  $2^d \cdot L'$  cells. Define  $\mathcal{C} = \bigcup_i \mathcal{C}_i$ . Clearly, we have  $|\mathcal{C}| \leq (2 + \log \Delta) \cdot 2^d \cdot L'$ . Lemma 6.1.6 has an error probability of  $1/(2\epsilon^2 s_0)$  and Lemma 6.1.7 has an error probability of  $2/(\lambda^2 \cdot s_0) \geq 2/s_0$  for  $\lambda < 1$  (which will hold for our choice of  $\lambda$ ). Hence, we can assume that the error probability for every cell and every lemma is at most  $2/(\lambda^2 \cdot \epsilon^2 \cdot s_0)$ . Summing up over all cells, the error probability is at most

$$2 \cdot |\mathcal{C}| \cdot 2/(\epsilon^2 \cdot \lambda^2 \cdot s_0) .$$

Hence, we have to use  $s_0 \geq \frac{(2+\log \Delta)L' \cdot 2^{d+8}}{\epsilon^2 \lambda^2}$  to guarantee that the probability that each lemma holds for all these cells is at least  $15/16$ .

**Lemma 6.1.10** *Let  $S \subseteq P$  be a random sample, where each element is chosen with probability  $r$ . Let  $\mathcal{C}$  be the set of cells in grid  $G_i$  that contain at least one point of  $S$ . Let  $L' \geq \sqrt{\frac{\ell}{\delta}} + 3^d + 1$  for some  $\ell > 0$  and let  $q \in \mathbb{R}^d$  be an arbitrary point. If  $\sum_{p \in P} \|p - q\|_2 \leq \ell \cdot \frac{2^i}{r}$  then with probability at least  $1 - \delta$  we have  $|\mathcal{C}| \leq L'$ .*

**Proof :** Let us fix an arbitrary point  $q \in \mathbb{R}^d$ . Let  $P' \subseteq P$  be the set of points with distance at least  $2^i$  from  $q$ . If  $\sum_{p \in P} \|p - q\|_2 \leq \ell \cdot \frac{2^i}{r}$  then  $|P'| \leq \frac{\ell}{r}$ . Let  $X_p$  be the random variable for the event that  $p \in P'$  is taken into the sample set. We have  $\mathbf{E}[X_p] = r$  and hence  $\mathbf{E}[\sum_{p \in P'} X_p] = \ell$ .

Since  $X_p$  is a 0–1–random variable, we have  $\mathbf{Var}[X_p] \leq \mathbf{E}[X_p]$ . By pairwise independence of the  $X_p$  we also have  $\mathbf{Var}[\sum_{p \in P'} X_p] \leq |P'| \cdot \mathbf{E}[X_p] \leq \ell$ . Using Chebyshev's inequality we get

$$\Pr \left[ \left| \sum_{p \in P'} X_p - \mathbf{E} \left[ \sum_{p \in P'} X_p \right] \right| \geq k \right] \leq \frac{\ell}{k^2} .$$

There are at most  $3^d$  cells with distance at most  $2^i$  from  $q$ . Hence, choosing  $k^2 = \ell/\delta$  gives for  $L' \geq \sqrt{\ell/\delta} + 3^d + 1$  that  $\Pr[|C| > L'] \leq 1 - \delta$ .  $\square$

### 6.1.6 1-Median in Dynamic Data Streams

We use  $L' \geq \sqrt{\frac{\ell}{\delta}} + 3^d + 1$  in the algorithm above for  $\delta = \frac{1}{16(1+d \log \Delta) \cdot (2+\log \Delta)}$  for some value of  $\ell$  to be determined later. For these choices we have by Lemma 6.1.10 that with probability at least  $15/16$ , for all  $i, j$ , if  $\sum_{p \in P} \|p - q_{opt}\|_2 \leq \ell \cdot \frac{2^i}{r}$  then  $|C| \leq L'$ . From now on let us assume that this event occurs. In this case we know that for all  $i, j$   $|C| > L'$  implies  $\ell \cdot \frac{2^i}{2^j} < \sum_{p \in P} \|p - q_{opt}\|_2$ , since in this case  $h_{i,j}$  is used to sample every element with probability  $1/2^j$ . In other words,  $\text{Set}_{i,j,L'} = \text{failure}$  implies that

$$\text{cost}(P, q_{opt}) > \ell \cdot 2^{i+j} .$$

For the analysis we will now assume that all procedures work within the specified bounds. This holds with probability at least  $7/8$ . We prove that for every center  $q$  for any point set  $P'$  that matches the bounds obtained via our sampling procedure we have

$$\left| \text{cost}(P, q) - \text{cost}(P', q) \right| \leq \epsilon \text{cost}(P, q) .$$

We extend the notion of near and distant to arbitrary centers as follows.

**Definition 6.1.11** *A cell  $C$  in grid  $G_i$  is called  $q$ -distant, if it has distance from  $q$  of more than  $2\sqrt{d} \cdot 2^i/\epsilon$ . Otherwise, it is called  $q$ -near.*

#### Lower Bound

To obtain a lower bound, we use that  $\text{Set}_{i,j,L'} = \text{failure}$  implies that  $\text{cost}(P, q_{opt}) > \ell \cdot 2^{i+j}$ . Let us define  $i_{\max}, j_{\max}, T_{\max}$  to be the values of  $i, j, T_i$  that maximizes  $2^{i+j} \cdot T_i$  during the course of the algorithm. We have that either  $j_{\max} = 0$  or  $j_{\max}$  increased by more than one when the algorithm moves to the current level  $i_{\max}$ . This implies that  $\text{Set}_{i_{\max}, j_{\max}-1, L'} = \text{failure}$  and hence

$$\text{cost}(P, q_{opt}) > \ell \cdot 2^{i_{\max}+j_{\max}-1} .$$

#### $q$ -Near Cells

Next we sum up the overall cost changes introduced by  $q$ -near cells. We have to show that this is at most  $\frac{\epsilon}{2} \cdot \ell \cdot 2^{i_{\max}+j_{\max}-1}$ . In the following we use  $\mathcal{L}_i^{\text{near}}$  to be the set of cells in  $\mathcal{L}_i$  that are  $q$ -near. We obtain

$$\sum_i |\mathcal{L}_i^{\text{near}}| \cdot \frac{2\sqrt{d} \cdot 2^i}{\epsilon} \cdot \lambda \cdot s_0 \cdot 2^j \leq (2+\log \Delta) \cdot \left(\frac{8\sqrt{d}}{\epsilon}\right)^d \cdot \sqrt{d} \cdot s_0 \cdot 2^{i+j+1} / \epsilon \leq \frac{\epsilon}{2} \text{cost}(P, q_{opt}) \leq \frac{\epsilon}{2} \cdot (P, q) .$$

$$\text{for } \ell \geq \frac{8\lambda(2+\log \Delta)s_0\sqrt{d}}{\epsilon^2} \cdot \left(\frac{8\sqrt{d}}{\epsilon}\right)^d .$$

## q-Distant Cells

It remains to sum up the error for the  $q$ -distant cells. Similarly to the non-streaming algorithm we will show that the change of cost is small in comparison with the contribution of the points. Here, the idea is slightly different because we also have to cope with errors in the estimated number of points rather than the movement of points.

We have to proceed by a case distinction. Let us consider a cell  $C$  in  $\mathcal{C}_{i-1}$  that is light. Let  $H \in \mathcal{H}_i$  be its parent cell. The first case is that at least one subcell of  $H$  is in  $\mathcal{H}_{i-1}$ . The second case is that all subcells of  $H$  are not in  $\mathcal{H}_{i-1}$ , i.e. they are light.

**Case (1).** We know that every cell with more than  $s_0 \cdot 2^{j+1}$  points is classified as heavy and every cell with less than  $s_0 \cdot 2^{j+1}$  points is approximated upto an additive error of  $\lambda \cdot s_0 \cdot 2^j$  by Lemma 6.1.7. Choosing  $\lambda < 1$  we know that every heavy cell has at least  $s_0 \cdot 2^j$  points. Thus, one of the subcells of  $H$  has at least  $s_0 \cdot 2^j$  points for the current value of  $j$ . Each point in this subcell contributes at least  $d(C, q) - \sqrt{d} \cdot 2^i$  to  $\text{cost}(P', q)$ , since  $C$  is a distant cell. Wlog. let us assume  $|P'| \geq |P|$  and let us consider an arbitrary subset  $P^* \subseteq P'$  of cardinality  $|P'| - |P|$ . The points in  $P^*$  contribute at most

$$\text{cost}(P^*, q) \leq \lambda s_0 \cdot 2^j \cdot (d(C, q) + \sqrt{d} 2^i) \leq \lambda \cdot s_0 \cdot 2^{j+1} \cdot d(C, q) .$$

This error is charged to the contribution of the points of  $P$  inside  $H$ , whose contribution is at least  $s_0 \cdot 2^j \cdot (d(C, q) - \sqrt{d} \cdot 2^i) \geq s_0 \cdot 2^{j-1} \cdot d(C, q)$ . Summing up over all cells in grid  $G_i$  we get an overall error of at most  $\lambda \cdot 2^{d+2} \cdot \text{cost}(P, q)$ . Hence, by summing up over all grids we get an overall error of at most

$$(2 + \log \Delta) \cdot \lambda \cdot 2^{d+2} \cdot \text{cost}(P, q) .$$

Hence, choosing  $\lambda \leq \frac{\epsilon}{8(2+\log \Delta)}$  gives an additive error of at most  $\epsilon/2 \cdot \text{cost}(P, q)$ .

**Case (2).** In Case (2) we use the number of points in  $H$  as an estimate. Since  $H$  is heavy, we know that the number of points inside is approximated within additive error  $\epsilon|H|$ . We further know that  $H$  contributes at least  $|H| \cdot d(H, q)$  to the cost of an optimal solution and this contribution is increased by at most  $(d(H, q) + \sqrt{d} 2^i) \cdot \epsilon|H| \leq 2d(H, q) \cdot \epsilon|H|$ . Hence, the overall change of contribution for each cell is at most  $2\epsilon$  times its current contribution. Thus, the overall change of cost is at most  $2\epsilon \cdot \text{cost}(P, q)$ .

The remaining analysis shows that moving the points inside a cell to its center increases the cost by at most another  $\epsilon \cdot \text{cost}(P, q)$ . Thus, the overall change of cost is at most  $3\epsilon \cdot \text{cost}(P, q)$ . Replacing  $\epsilon$  with  $\epsilon/3$  concludes the proof.

## 7 Lower Bounds

In this chapter we discuss how to prove lower bounds for streaming algorithms. In general, it is easier to prove a lower bound for a streaming algorithm than for a classical algorithm. The reason is that streaming algorithms work in a more restricted model of computation and the more restricted a model of computation is the harder it is to develop algorithms and the easier it is to prove lower bounds. In the streaming case it is clear that some problems cannot be solved in small space and with a single pass over the data. A simple example is the following problem. We are given a stream of  $n$  bits  $a_1, \dots, a_n$ . The problem is to design a deterministic streaming algorithm, i.e. an algorithm that uses  $\log^{O(1)} n$  space and that can answer the question 'What is the value of  $a_i$  for any index  $i \in \{1, \dots, n\}$ '. Clearly, we cannot design such an algorithm because it would imply that we can encode an arbitrary  $n$ -bit string using  $\log^{O(1)} n$  bits, which is impossible.

The arguments used in this section will always be as follows. If a certain streaming algorithm  $A$  exists, then we can design a compression algorithm that takes an arbitrary  $n$ -bit string  $y$  and compresses it to an  $o(n)$ -bit string  $\gamma$ , such that we can recover  $y$  from  $\gamma$ . Obviously, this is impossible and hence  $A$  cannot exist.

We will have to cope with two difficulties: approximation and randomization. Why do these two properties make it harder to prove lower bounds? Intuitively, both approximation and randomization make the output of an algorithm 'more fuzzy'. This makes it more difficult to exploit certain properties of a solution.

### 7.1 Distinct Elements cannot be solved Deterministically

In this section we develop our first lower bound. We will show that no deterministic algorithm can compute the number of distinct elements in a stream of elements from a universe  $U = \{1, \dots, n\}$  using  $o(n)$  space. Then we will prove that no deterministic algorithm with  $o(n)$  space can approximate the number of distinct elements within a certain (small enough) constant factor. This result together with our randomized algorithm to approximate the number of distinct elements shows that randomization is essential for some streaming problems.

**Theorem 24** *Any deterministic streaming algorithm that computes the number of distinct elements in a stream of elements from a universe  $U = \{1, \dots, n\}$  requires  $\Omega(n)$  bits of space.*

**Proof :** Assume we have a deterministic streaming algorithm  $A$  that computes the number of distinct elements in a stream of elements from a universe  $U = \{1, \dots, n\}$  exactly and that uses  $o(n)$  bits of memory. Now consider an arbitrary 0 – 1-vector  $y = (y_1, \dots, y_n)$  with exactly  $n/2$  ones. We will view  $y$  as a characteristic vector for elements from  $U$ . Thus, a vector  $y$  corresponds to a set of  $n/2$  elements from  $U$ . We can turn these elements into a stream by

sending them, for example, in increasing order. We will show how we could use algorithm  $A$  to compress the vector  $y$  into an  $o(n)$  bit string  $\gamma$  in such a way that we can recover  $y$  from  $\gamma$ . Since there are at least  $2^{n/c}$  choices for  $y$  for some constant  $c$ , we can encode any  $(n/c)$ -bit string as a unique vector  $y$ . Thus, we have designed an algorithm that compresses  $n/c$  bits to  $o(n)$  bits, which is a contradiction and so  $A$  cannot exist.

The idea of our 'compression algorithm' is quite simple. From  $y$  we construct a stream of  $n/2$  elements from  $U$  and run  $A$  on this stream. Then we store the current state  $s(y)$  of  $A$ . Since  $A$  uses  $o(n)$  bits of memory, we can store the state of  $A$  in a bitstring with  $o(n)$  bits. This short bitstring will be our compression  $\gamma$  of  $y$ . Next, we show how to 'decompress'  $\gamma$ . Let us consider an arbitrary  $0-1$  vector  $z$  with  $n/2$  ones. Again, this vector encodes a set with  $n/2$  elements from  $U$ , that can be easily turned into a stream. The idea is to run  $A$  on this stream starting in state  $s(y)$ . The final state of the algorithm is similar as if the algorithm was run on the stream corresponding to  $y$  followed by the stream corresponding to  $z$ . Hence, the number of distinct elements is  $n/2$ , if and only if  $y$  and  $z$  encode the same set of elements. Thus, having algorithm  $A$  and the state  $s(y)$  it suffices to try all values of  $z$  and check whether the number of distinct elements is  $n/2$  to recover  $y$ . Hence, algorithm  $A$  cannot exist.  $\square$

Our next step is to consider approximation algorithm. Here, we cannot immediately apply the idea from above because it requires us to distinguish between  $n/2$  and  $n/2 + 1$  distinct elements, which cannot be done by a constant factor approximation algorithm.

### 7.1.1 There is no Deterministic $c^*$ -Approximation Algorithm for Distinct Elements

We will now show that there is no deterministic  $c^*$ -approximation algorithm for some constant  $c^*$ , which will be determined later. The basic idea is to apply a similar approach as above, but instead of using  $y$  as a characteristic vector we first encode  $y$  using an error-correcting code.

**Definition 7.1.1 (Code)** A code  $C$  is an injective mapping from  $\{0, 1\}^n$  to  $\{0, 1\}^m$ . The set of codewords of  $C$  is given by  $\{a \in \{0, 1\}^m : \exists b \in \{0, 1\}^n \text{ such that } C(b) = a\}$ .

**Definition 7.1.2 (Hamming Distance)** The Hamming distance between two  $m$ -bit vectors  $a = (a_1, a_2, \dots, a_m)$ ,  $b = (b_1, b_2, \dots, b_m) \in \{0, 1\}^m$  is given as  $H(a, b) = \sum \mathbf{1}(a_i \neq b_i)$ , i.e. the number of values  $i$ ,  $1 \leq i \leq m$ , for which  $a_i \neq b_i$ .

**Definition 7.1.3** Let  $C$  be a code and  $W$  be its set of codewords.  $C$  is called  $c$ -error-correcting, if for every  $a \in \{0, 1\}^m$  there is at most one  $b \in W$  with  $H(a, b) \leq c$ .

Hence, consider a message  $m$  which is encoded by a  $c$ -error correcting code  $C$ . Assume that we send  $C(m)$  over some noisy channel such that due to the noise at most  $c$  bits are flipped. Let us call the resulting corrupted message  $C_e(m)$ . Then we know that  $C_e(m)$  is still closer to  $C(m)$  than to any other codeword and hence we can decode  $C_e(m)$  by moving to the nearest codeword. This process works, as long as we have at most  $c$  errors in the corrupted message.

How can we design an error correcting code? The most simple idea is to use repetition. Instead of sending every bits once, we send it three times. Clearly, the resulting code is 1-error-correcting. But this method of encoding does not seem to be very effective. In the following we prove the existence of much better error correcting codes.

**Claim 7.1.4** Let  $W \subseteq \{0, 1\}^m$  be a set of vectors with pairwise Hamming distance at least  $2c + 1$ . Let  $C$  be an injective mapping from  $\{0, 1\}^n$  to  $W$ . Then  $C$  is a  $c$ -error-correcting code.

**Proof :** Let us assume that  $C$  is not  $c$ -error-correcting. This means, that there exists a vector  $x \in \{0, 1\}^m$  and  $a, b \in W$  with  $H(a, x) \leq c$  and  $H(x, b) \leq c$ . Since the Hamming distance is a metric, it satisfies the triangle inequality. This implies  $H(a, b) \leq H(a, x) + H(x, b) \leq 2c$ , which is a contraction to the choice of  $W$ .  $\square$

Now we prove the existence of a good error correcting code. Our approach is based on the above claim and constructs the set of codewords  $W$  from  $\{0, 1\}^m$  incrementally. Once we have  $W$ , we set  $n = \lfloor \log |W| \rfloor$  and define  $C$  to be an arbitrary injective mapping from  $\{0, 1\}^n$  to  $W$ . The interesting question is how are  $m$  and  $n$  related.

We begin with  $X = \{0, 1\}^m$  as the set of candidates for our codewords and set  $W = \emptyset$ . Then we take an arbitrary codeword  $w \in \{0, 1\}^m$  and put it into  $W$ . We remove all words  $a \in X$  from  $X$  with  $H(x, w) \leq 2c$ . Then we take the next codeword from  $X$  and again remove all words within Hamming distance at most  $2c$ . We proceed until  $X$  is empty. Clearly, the words in  $W$  have pairwise Hamming distance at least  $2c + 1$  and so the resulting code is  $c$ -error-correcting, i.e. it receives an  $n$ -bit string and encodes it as an  $m$ -bit string that allows decoding in the presence of upto  $c$  errors. The question is, how large (small?) is  $n$ . The relation between  $n$  and  $m$  tells us, how much overhead is required for ensuring error tolerance.

To find out the size of  $n$  we have to compute a lower bound on the size of  $W$ . This is achieved as follows. We know that in the beginning  $X$  has  $2^m$  bit vectors. Each time when we add a new code words we eliminate all codewords within Hamming distance  $2c$  of it. If  $B$  denote the number of words within Hamming distance  $c$  from a vector  $x$ , then we know that  $W \geq 2^m/B$ . So, it remains to determine an upper bound for  $B$ .  $B$  is simply the number of words inside a Hamming ball of radius  $2c$  around a vector  $x$ . By symmetry, we can simply assume that  $x = (0, 0, \dots, 0)$ . Then a word is inside a Hamming ball of radius  $2c$ , if it has at most  $2c$  ones. The number of these words is given by

$$B = \sum_{i=0}^{2c} \binom{m}{i} \leq 2c \cdot \binom{m}{2c} \leq 2c \cdot \left(\frac{m \cdot e}{2c}\right)^{2c} .$$

In the following, we will define  $c = \frac{\epsilon \cdot m}{32}$ , i.e. we allow error linear in  $m$ . In this scenario, we can use at most  $\lfloor \log |W| \rfloor$  bits for our code. For simplicity, we will ignore the rounding errors. We have

$$\begin{aligned} \log W &\geq \log \frac{2^m}{B} \\ &= m - \log B \\ &\geq m - (2c \cdot \log \frac{m \cdot e}{2c} + \log(2c)) \\ &\geq m - 4c \cdot \log \frac{m \cdot e}{2c} \\ &\geq m/2 \end{aligned}$$

for our choice of  $c$ .

**Theorem 25** For any  $n$  there exists an  $\frac{e \cdot m}{32}$ -error-correcting code that requires  $m = O(n)$  bits per codeword.

Now we can use error correcting codes to obtain a space lower bound for approximating the number of distinct elements with a deterministic streaming algorithm. For the proof, we need an error correcting code  $C$ , whose codewords have exactly  $m/2$  ones. Since  $\binom{m}{m/2}$  is at least  $2^{m-\log m}$ , we can use a similar construction as above to prove that there exists an  $\frac{e \cdot m}{128}$ -error-correcting code whose codewords all have  $m/2$  ones (assuming  $m \geq 4$ ). We assume that we have such a code  $C$ . Let  $c^* = \frac{e}{128}$ . We assume that there is a deterministic algorithm  $A$  that takes a stream of elements from  $\mathbb{U} = \{1, \dots, m\}$ , uses  $o(m)$  bits of memory, and computes a factor  $(1 + c^*)$  approximation of the number of distinct elements in the stream, i.e. it computes a value  $\tilde{F}_0$  such that  $F_0 \leq \tilde{F}_0 \leq (1 + c^*)F_0$ . In order to compress an arbitrary string  $y \in \{0, 1\}^n$  into a small string, we apply  $A$  to the set of items encoded by  $C(y)$  and store the final state  $s(y)$  of  $A$  using  $o(m) = o(n)$  bits (since  $m = O(n)$ ). Using algorithm  $A$  we can now recover  $y$  from  $s(y)$  in the following way. We choose a string  $z \in \{0, 1\}^m$  with exactly  $m/2$  ones and apply  $A$  to the set of item encoded by  $z$  starting in state  $s(y)$ . Clearly, the output value is a  $(1 + c^*)$ -approximation for the number of distinct elements given by the union of the elements encoded by  $C(y)$  and  $z$ . If  $C(y) = z$  then there are  $m/2$  distinct elements and so the algorithm returns a number between  $m/2$  and  $m/2 + c^* \cdot m/2$ . On the other hand, if  $H(C(y), z) > \frac{e \cdot m}{128}$  then  $A$  computes a value of more than  $m/2 + c^* \cdot m/2$ . We try all choices for  $z$  until we find a vector  $z$  such that the algorithm outputs a number not bigger than  $m/2 + c^* \cdot m/2$ . Clearly, for this  $z$  we have  $H(C(y), z) \leq \frac{e \cdot m}{128}$ . Since  $C$  is  $\frac{e \cdot m}{128}$ -error-correcting we can obtain  $y$  from  $C(y)$ . Hence, algorithm  $A$  cannot exist.

**Theorem 26** There exists a constant  $c^*$  such that any deterministic streaming algorithm that approximates the number of distinct elements in a stream of elements from  $\{1, \dots, m\}$  upto a factor of  $(1 + c^*)$  requires  $\Omega(m)$  bits of memory.

## 7.2 Lower Bounds for Randomized Algorithms

In this section we will derive some lower bounds for randomized streaming algorithms. Our first step is to prove that there is no randomized algorithm that computes the number of distinct elements in a stream of elements from  $\mathbb{U} = \{1, \dots, m\}$  using  $o(m)$  space.

The general proof idea is similar to the lower bound proofs for deterministic algorithms. We will show that the existence of a streaming algorithm implies that we can compress an arbitrary  $n$ -bits string using  $o(n)$  bits. Let us assume that there exists a randomized algorithm  $A$  that compute the number of distinct elements in a stream of elements from  $\mathbb{U} = \{1, \dots, m\}$  exactly with error probability at most  $1/4$  using  $o(m)$  space. Running a constant number of copies of the algorithm and taking the median of the resulting output values we can amplify the success probability to an arbitrary constant. We will use the error correcting code  $C$  from the proof of the previous theorem and assume that the error probability of our algorithm is at most  $c^*/4$ . Given an arbitrary vector  $y \in \{0, 1\}^n$  let  $S_y \subseteq \mathbb{U}$  be the set of items that is described by the characteristic vector  $C(y)$ . For each set  $S_y$  let us define a unique stream  $s_{y,1}, \dots, s_{y,m/2}$  as the set  $S_y$  in increasing order. We feed this stream to algorithm  $A$  and store its state using

$o(m) = o(n)$  bits. We show that there exists a choice of random coins such that one can retrieve  $y$  from this internal state for a constant fraction of vectors  $y \in \{0, 1\}^n$ . This implies that we can store  $\Theta(n)$  bits using  $o(n)$  bits, a contradiction.

In order to retrieve  $y$  from the internal state  $i$ , we proceed as follows. For every  $j \in U$  we run algorithm  $A$  starting in state  $i$  on the stream consisting of the single element  $j$ . If the output value is  $m/2$  then we set  $z_j = 0$ . Otherwise, we set  $z_j = 1$ . For a fixed stream  $s_{y,1}, \dots, s_{y,m/2}, j$  we know that  $A$  has error probability at most  $c^*/4$  and so  $z_j = C(y)_j$  with probability at least  $1 - c^*/4$ . This implies that if we choose the  $j \in U$  uniformly at random and consider the stream  $s_{y,1}, \dots, s_{y,m/2}, j$  we still have success probability at least  $1 - c^*/4$ .

Now let  $p_i$  denote the probability that algorithm  $A$  started in state  $i$  on a random  $j \in U$  has an error. Since the overall probability of error for stream  $s_{y,1}, \dots, s_{y,m/2}, j$  is

$$\sum_i \Pr[i] \cdot p_i \leq c^*/4$$

we obtain that there exists a subset  $I$  of states with  $\sum_{i \in I} \Pr[i] \geq 1/2$  and  $p_i \leq c^*/2$  for all  $i \in I$ . For any state in  $I$  we have that the expected number of errors is at most  $c^*/2 \cdot m$ . By Markov inequality, with probability at least  $1/2$  we have less than  $c^* \cdot m$  errors. In this case, we can reconstruct  $y$  from  $z$ . Since this case occurs with probability at least  $1/4$  we know that there is a choice of random coin tosses which works for  $1/4$  of the vectors  $y$ . Hence,  $A$  cannot exist.

**Theorem 27** *Any randomized algorithm that computes the number of distinct elements in a stream of elements from a universe  $\{1, \dots, m\}$  with probability at least  $3/4$  requires  $\Omega(m)$  bits of memory.*

### 7.2.1 Lower Bounds for Randomized Computation of the Second Frequency Moment

In a similar way as above we can also prove that one cannot solve the following problem with a randomized streaming algorithm in  $o(m)$  space. We are given a stream of  $m/2 + 1$  elements, where the first  $m/2$  elements are distinct elements from a set  $U = \{1, \dots, m\}$  and the last element is an element  $i \in U$ . The goal is to output whether element  $i$  occurs among the first  $m/2$  elements or not. We will call this problem the indexing problem and reduce it to the computation of the second frequency moment (via an intermediate problem).

**Definition 7.2.1 (Gap Dot Product)** *In the gap product problem we are given access to a data streams  $\sigma_1, \dots, \sigma_n$  with pairs of elements  $\sigma_i = (u, j)$ , where  $u \in U = \{1, \dots, m\}$  and  $j \in \{1, 2\}$ . The stream encodes two multisets of elements. The multiset  $V = \{u : \sigma_i = (u, 1) \text{ for some } i\}$  and  $U = \{u : \sigma_i = (u, 2) \text{ for some } i\}$ . We use  $f_V$  and  $f_U$  denote the frequency vector of  $V$  and  $U$ , respectively. The gap dot product problem is to accept, if  $f_U \cdot f_V = 0$  and reject, if  $f_U \cdot f_V \geq \Delta$ . If  $0 < f_U \cdot f_V < \Delta$  the algorithm may accept or reject.*

We will prove that a special case of the gap dot product problem is hard.

**Lemma 7.2.2** *Let us restrict the dot product problem to instances, where  $\|f_V\|_2^2 = \|f_U\|_2^2 = m/2$  and where  $V$  only contains distinct elements. Then solving the dot product problem with probability at least  $3/4$  for  $\Delta = (\frac{m}{2})^{1/2}$  requires  $\Omega(m)$  space.*

**Proof :** We can reduce the indexing problem to the restricted dot product problem in the following way. Let us assume that algorithm  $A$  solves the dot product problem with probability at least  $3/4$ . Then we can use  $A$  to solve the indexing problem. The first  $m/2$  elements  $\sigma_1, \dots, \sigma_{m/2}$  are passed to  $A$  as pairs  $(\sigma_i, 1)$ . Then the last element  $i$  of the stream is read. For this element we append  $\sqrt{m/2}$  copies of the form  $(i, 2)$  to our stream. Since the first  $m/2$  elements are distinct, we have  $\|f_U\|_2^2 = \|f_V\|_2^2 = m/2$ . Now observe that  $f_U \cdot f_V = 0$ , if  $i$  does not occur among the first  $m/2$  elements and  $f_U \cdot f_V = \Delta$ , if it appears among the first  $m/2$  elements. Hence, the dot product problem decides the indexing problem. Thus algorithm  $A$  requires  $\Omega(m)$  space.  $\square$

To prove that the second frequency moment is hard to compute, we reduce the restricted gap dot problem to it. Assume we have an algorithm  $A$  that computes a  $(1 + \epsilon)$ -approximation to the second frequency moment for some value  $\epsilon = \frac{\epsilon}{\Delta}$ , i.e. a value  $\tilde{F}_2$  such that  $F_2 \leq \tilde{F}_2 \leq (1 + \epsilon) \cdot F_2$ . Given access to a stream for the restricted dot product problem, we compute the second frequency moment of the multiset  $V \cup U = \{u : \sigma_j = (u, 1) \text{ or } \sigma_j = (u, 2)\}$ . We use the equality

$$\|f_U + f_V\|_2^2 = \|f_U\|_2^2 + \|f_V\|_2^2 + 2 \cdot f_U \cdot f_V .$$

Since we know that  $\|f_U\|_2^2 = \|f_V\|_2^2 = m/2$  we have

$$\|f_U + f_V\|_2^2 = m + 2 \cdot f_U \cdot f_V .$$

Since we know that for the restricted gap dot problem we have  $\|f_U \cdot f_V\|_2^2 \in \{0, m/2\}$  we get that  $\|f_{U \cup V}\| \leq 3m/2$ . This implies that  $\|f_{U \cup V}\|$  is approximated with additive error at most  $\Delta/4 = \frac{3}{2\epsilon}$ . Hence, also  $f_U \cdot f_V$  is approximated upto an additive error of  $\pm \Delta/4$ , which solves the restricted dot product problem for approximation parameter  $\epsilon$  (we can accept, if the approximated value is at most  $\Delta/4$  and reject, if it is larger). This implies that  $A$  requires  $\Omega(m) = \Omega(1/\epsilon^2)$  space.

**Theorem 28** *Any randomized streaming algorithm that approximates the second frequency moment upto a factor of  $(1 + \epsilon)$  with probability at least  $3/4$  requires  $\Omega(1/\epsilon^2)$  space.*