DATA STREAM ALGORITHMS

by

Yihui Tang

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY

School of Computer Science

at

CARLETON UNIVERSITY

Ottawa, Ontario
January, 2008

Your file Votre référence
ISBN: 978-0-494-36795-7
Our file Notre référence
ISBN: 978-0-494-36795-7

Abstract

In recent years, there has been a dramatic growth of interest in developing algorithms for massive data sets. In particular, the *data stream* model has received a lot of attention. Many applications that deal with massive data, such as Internet traffic analysis and database mining, motivate this data stream model. In the data stream model, the data is treated as sequences, and the only feasible way of accessing data is through sequential access. This is a more appropriate computational model than the classic *random access machine (RAM)* model when dealing with massive data sets for which random access is expensive or even impossible. Due to these constraints, many data stream algorithms are randomized and/or compute only an approximation of the exact answer. Designing such data stream algorithms often involves trade-offs between time, space and accuracy.

In this thesis we study several problems in this computational model. The first result is on estimating the frequency of each element in a data stream using a small amount of memory. Previous research on this topic has been focused on the algorithms that compute probabilistic results. Our study shows that an element's frequency can be estimated with guaranteed accuracy and we give a near optimal trade-off between space and accuracy in Chapter 3. Then we study the problem of quickly answering range mode and range median queries, which are two of the most important statistics of a data set. We propose the first non-trivial solutions to the approximate versions of the problem, first on one dimensional arrays (Chapter 4), then on matrices in higher ($d \geq 2$) dimensional space (Chapter 5). Finally, we study one of the earliest data stream problems, namely, sorting large data sets stored on tapes with limited internal memory. There is a gap of a factor of 4 in previous results on the lower bounds and upper bounds. We close this gap in Chapter 6. We also derive the first probabilistic lower bound for the problem.

Acknowledgements

I would like to thank my thesis advisers: Professor Evangelos Kranakis for his continuous support and encouragement, whose guidance going far beyond thesis research, gives a special meaning to the word ``adviser''; Professor Prosenjit Bose, whose passion for scholarly pursuits and sense of humor makes the lab a much more enjoyable place to be; last but certainly not least, Professor Pat Morin, who is a seemly inexhaustible source of knowledge and ideas, always has the answers to my questions even before I understand what my questions are really about, truly a role model for anyone who strives for academic excellence.

Table of Contents

List of Tables

List of Figures

List of Frequently Used Notations

The following is a list of the commonly used notations. The first entry is the symbol itself, followed by its meaning or name (if any). In addition, some standard symbols whose definition and usage should be clear from the text.

Symbol and Operations	Meaning
$f_x\{M\}$	Number of occurrences (frequency) of x in set M
F_k	The k-th frequency moment, $F_k = \sum_x f_x^k$
F_0	The number of distinct elements
F, F_∞	The frequency of the most frequent element, $F = F_\infty = \max\{f_x\}$
$EQ\{x, y\}$	Equality function
$DISJ(x, y)$	Set disjointness function
$\tilde{x}$	An approximate of number x
$\tilde{O}(f(m))$	$f(m)(\frac{1}{\epsilon})^{O(1)} \log^{O(1)} mn \log 1/\delta$
$\lambda(k, \cdot)$	The $\lfloor k/2 \rfloor$-th level of the primitive recursive hierarchy
C_k	The k-th mode curve
$d(C_i, C_j)$	Distance between C_i and C_j
$QR_{(x_1,y_1)(x_2,y_2)}$	Query range bounded by (x_1, y_1) and (x_2, y_2)
$\overrightarrow{ab}$	Directed distance from a to b
$\overline{ab}$	Distance between a and b

Chapter 1

Introduction

Although the concept of making a few passes over the data for performing computation can be traced back much earlier when tapes were the main storage medium (Knuth [1998]), the *Data Stream Model* of computation has only received much attention in recent literature (Alon et al. [1996], Manku et al. [1998], Henzinger et al. [1999], Datar et al. [2002], Demaine et al. [2002], Arasu and Manku [2004]). In this model, the input to an algorithm is a sequence of data items that may only be read in sequential order; no random access of the data is allowed. Furthermore, the amount of memory used by the algorithm must be small (typically $o(n)$, where n is the length of the input data stream, which can be very large or even unbounded).

The data stream model fits well with the constraints of computing on massive data sets, where the size of the input data is too large to fit entirely into main memory. Instead, the data must be placed on secondary storage devices such as disks or tapes. There is a large overhead cost (seek time) to access these storage devices, and their designs are optimized for sequential access performance (Hennessy and Patterson [2007]). Random accesses to data stored on these devices are very expensive if not entirely impossible, as a result, sequential retrieval is the only practical way of accessing such very large data sets. The data stream model is also a good model for computations that must be carried out on the data that is generated continuously in quantities too large to be stored (*e.g.*, millions of packets passing through a high-speed network router every second).

1.1 Motivation

Most modern algorithm research assumes a simple *Random Access Machine (RAM)* model of computation. In this model, a computer system consists of a CPU that

executes instructions and a memory system that holds instructions and data. Furthermore, the memory system is a linear array of bytes, and the CPU can access an arbitrary memory location in a single step. While this is an effective model as far as it goes, a number of recent technological developments motivate our study of the data stream model.

- *Ability to generate high volumes of data continuously.*

  Several applications naturally generate data streams. In telecommunications, for example, millions of call records are generated everyday. Also, the ubiquitous high-speed Internet and the rapidly decreasing cost of computing and storage have scaled up the rate at which data is generated tremendously in various streams: browser click streams, user queries received by a search engine, electronic financial transaction records, just to name but a few. The ever increasing ability of generating massive data streams challenges the effectiveness of the RAM model.

- *Need for sophisticated real-time analysis of data.*

  With traditional data feeds, all the data of interest are archived in a data warehouse before complex tasks such as statistics collection, trend analysis, and fraud detection can be performed off-line on the collected data. However, with the automatically generated data streams arising from various applications, such as network management, financial data analysis, sensor networks, it becomes more time critical to detect intrusion, fraud, emerging trends, anomalous activities, etc. Which means the computation needs to be done constantly and in real time in order to keep pace with the rate of data generation and accurately reflect rapidly changing characteristics of the data.

There are also applications where non-streaming data is treated as a stream due to performance constraints. In data mining applications, for example, the volume of data stored on disk is often so large that it is only possible to make one pass (or perhaps a small number of passes) over the data. The objective is to perform the required computations with a single scan (or a small number of scans) of the data,

using only a bounded amount of memory that is much smaller than the size of the entire data set.

The increasing need to process large data sets poses some unique challenges that have not been faced before. The storage methods are the major issue here. Most of the data sets reside on (magnetic or optical) disks or tapes as these are the most economical way to store large volumes of data; these devices naturally produce data streams. The technology of these devices is optimized for sequential access. Random access is typically expensive. This is especially true for tape drives that have been for a long while considered as backup media of choice for large data sets. In traditional computer science research, an algorithm is considered efficient if it runs in small (polynomial) time and requires small (linear) space. For large data sets, efficient algorithms must run in linear or even sublinear time (that is, look at only a subset of the data and thus avoid scanning through all the data) and must use sublinear space (that is, avoid using the additional storage size comparable to the input size).

Another challenging aspect of processing over data streams is that while the length of a data stream may be unbounded, making it impractical or undesirable to store the contents of the entire stream, for many applications it is still important to retain some ability to execute queries that reference past data. In order to support queries of this sort using a bounded amount of storage, it is necessary to devise techniques for storing summary or synopsis information about previously seen portions of data streams. Generally there is a trade-off between the size of the summaries and the ability to provide precise answers to queries involving past data.

1.2 The Data Stream Model

The data stream model is proposed in response to the need for dealing with massive data sets in many practical applications. It abstracts the requirement that data be processed on-the-fly by a machine that has very little internal memory. In this model, an input $x_1, \ldots, x_n$ is written as a sequence of data items on an input tape. Computation is done by a machine with small local memory making one or a small number of sequential passes on the tape.

A *data stream algorithm* is an algorithm that computes some function f over a

data stream, it may access the data in a sequential manner only. Every retrieval of the input values $x_1, \ldots, x_n$ is called "a pass". More formally:

- $f : X^n \to Y$, defines the function to be computed.
- X and Y are arbitrary sets (typically small, such as $\{0, 1\}$).
- n: the number of elements in the stream (usually very large, possibly unbounded).
- Given $x \in X^n$, every $x_i \in x$ is called "an element".
- The input data are accessed in sequential order.
- The order of the elements in the stream is not controlled by the algorithm.
- The goal is to compute $f(x)$, or an approximation of $f(x)$, given $x \in X^n$.

The efficiency of a data stream algorithm is measured in terms of the size of the internal memory, the number of passes, and the time it spends on performing the computation. In the past few years, algorithms for many important problems have been developed in this model. Many of these data stream algorithms have the following flavor: they use space sublinear in the size of the input, they are randomized, and they produce only an approximation of the correct answer after one or a small constant number of passes over the data.

1.2.1 Sliding Window Model

For many applications in which data takes the form of continuous data streams, it is both impractical and unnecessary to process anything but the most recent data. For example, a phone company may only need to examine a call record once, and operate on a "window" of recent call records to update customer billing information for the past month. Other examples of such applications include network monitoring and traffic measurements, financial transaction logs, etc. All these applications view recently arrived data as more important than those a long time back (see Demaine et al. [2002], Estan and Varghese [2002], Venkataraman et al. [2005]). This preference

for recent data is referred to as the *sliding window model* in which the queries are answered regarding only the most recently observed W data elements (Datar et al. [2002]). Suppose x_i is the current data element, then the window consists of elements $x_{i-W+1}, x_{i-W+2}, \ldots, x_{i-1}, x_i$. New data elements arrive at each time instant and old elements expire after W time steps, and the portion of data that is relevant is the set of the most recent W data elements. To answer queries that may come at any time, we need to maintain the statistics about the window in a continuous fashion, in other words, updates may be necessary at every time instant. The *sliding window* refers to the window of active data elements at any time instant. A *sliding-window algorithm* is an algorithm that computes some function over the window for each time instant.

There are two common types of sliding windows: *counter-based windows*, which maintain the last W data elements seen at any time instant and *time-based windows*, which include only those items which have arrived in the last T time units.

If the entire window fits in main memory, answering queries over sliding window is simple. For example, to find all items that occur more than the given threshold, we maintain the count of each distinct item in the window and update the counters as new items arrive and old items expire. To know which item is to expire, it is necessary to remember the arrival time of each data item. Unfortunately, it is not unusual that the data elements arrive so fast that there is not enough storage space for all the data, even for the data in one window. In this case, the window must somehow be summarized and an answer must be approximated on the basis of the available summary information.

1.2.2 Single Pass vs. Multiple Passes

Intuitively, the rigid restriction of data stream algorithms to a single pass over the data limits its potential utility. To circumvent this restriction, algorithms using multiple passes were discussed in the literature (e.g., Munro and Paterson [1980], Alon et al. [1996], Henzinger et al. [1999]). In these mostly database related applications, multiple passes, although not desirable, may be tolerable if the algorithms are not to make too many passes over the data. The *pass-efficient model* has been proposed by Drineas and Kannan [2003] as a more flexible version of the streaming model.

The pass efficient model allows for multiple sequential passes over the input (ideally a small, perhaps constant, number of passes), and a small amount of extra space. While processing each element of the data stream, the algorithm may only use computing time that is independent of the size of the data set (n), but after each pass, it is allowed more computing time (typically $o(n)$). In this model of computation, we are most concerned with two resources: *the number of passes* and *additional storage space.*

Although data stream algorithms are a quite recent phenomenon, the history of multi-pass algorithms can be traced back much earlier (Knuth [1998]). The seminal paper of Munro and Paterson [1980] studied the tradeoffs between the number of passes and space for the classical sorting and selection problems. In particular, they gave a selection algorithm that requires only $\lceil 1/\delta \rceil$ passes and $O(n^{\delta} \log^{2-2\delta} n)$ space for an arbitrary fixed constant $0 < \delta < 1$; they also provided an almost matching lower bound.

More recently, algorithms that are allowed to make multiple (but a small number of) passes over the input with limited working memory have also been studied for geometric problems (Agarwal et al. [2003], Suri et al. [2004], Chan and Chen [2005]) among others (Pagter and Rauhe [1998], Bar-Yossef et al. [2002], Drineas and Kannan [2003], Suri et al. [2004], Feigenbaum et al. [2004, 2005], Govindaraju et al. [2005]). The restriction in the data stream model that input elements are read sequentially in a few passes is attractive for such a wide range of applications because of the lower I/O overhead. An interesting result is by Chan and Chen [2005]. In this paper, the authors investigated multi-pass algorithms in computational geometry, and they discovered that although in the one-pass model one cannot usually obtain exact algorithms and must turn toward approximation algorithms, in the multi-pass model there are geometric problems that can be solved exactly.

1.2.3 Complexity Measures

Designing algorithms which efficiently solve data processing problems has been a focus of computer science since its very beginning. However, the definition of what constitutes an "efficient" algorithm changes drastically when one attempts to process

terabytes of data or more. For example, a linear running time, which for many years was considered to be an ultimate efficiency goal, could easily mean that the algorithm can take several days to finish. Alternatively, an algorithm which requires linear space could easily become unacceptable if the required amount of memory is not available; this applies to cases when the continuously generated data is much too voluminous even to store or archive.

Given an input of massive data sets, the algorithm should take little time to process each data item and should use little space in comparison to the input size. This is especially important for many data stream applications where the data is generated online and real time response is highly desirable. In the data stream model, we are mainly concerned with the following three performance measures of a data stream algorithm.

- *Space.* The algorithms are to use as little memory as possible. If memory was unlimited, the data stream model would not be necessary: given an unlimited memory, one can store the entire input in RAM and retrieve it in a random access manner. Ideally, the amount of memory will be poly-logarithmic in the input size (i.e., $O(\log^c n)$, for some constant $c \geq 0$) or even $\Theta(1)$.

- *Number of passes.* Since the model usually deals with extremely large data sets, even a single pass over the complete data set will take a long time. Preferably, the algorithm would finish within a (small) constant number of passes. When the data is generated "on the fly", only a single pass is ever possible.

- *Running time.* As little as possible. Any algorithm that requires significantly super-linear computation time is practically impossible to solve on these inputs. Ideally, the running time would be quasilinear in the input size (meaning $O(n \log^c n)$, for some constant $c \geq 0$) or even linear meaning in average only constant time is spent on each item.

1.3 Data Stream Algorithms vs. External Memory Algorithms

The availability of large data sets, often on the order of terabytes, and also the growing disparity between the fast increasing capacity and the slowly improving random access

time of magnetic disks (Hennessy and Patterson [2007]), have led to an interest in computational models suitable for large data sets. Two such models are *external memory model* (Vitter [2001]) and *data stream model*, the latter being more restrictive than the former. In the external memory model, there are two levels in the memory hierarchy: a (small but fast) cache and a (big but slow) disk. The disk is partitioned into blocks of size B, and each read or write is of an entire block. Also, the cache has size M and holds $\frac{M}{B}$ blocks, while the disk holds a much greater number of blocks. The CPU can access the cache quickly, while there is a large overhead cost to access the disk (seek time). The running time of an algorithm is therefore dominated by the disk I/O and the main goal in external memory algorithms design is to minimize the number of memory transfers (reads and writes to disk) the algorithm makes.

Although the external memory model has been proposed as an effective model for dealing with large data sets that cannot fit in RAM, it is not applicable when the data set in question is so huge (possibly even infinite in the case of online streams) that it cannot be stored on disks. Even when the data set can be stored in its entirety, the following two examples illustrate the similarities and differences between the external memory model and the data stream model.

- *Scanning*

  We have an array of N elements (numbers) stored on disk and want to compute the sum of these numbers. By reading sequentially one block of B elements at a time, there would be $\lceil \frac{N}{B} \rceil$ memory transfers. Obviously this algorithm is optimal in both the data stream model and the external memory model.

- *Searching*

  B-tree (Bayer [1971]) is a very efficient data structure for searching in the external memory model, in fact, it is asymptotically optimal as only $O(\frac{\log N}{\log(B+1)})$ block transfers are needed per search. This is no longer true in the data stream model, where using B-tree for searching requires multiple random accesses to blocks and therefore, multiple passes ($O(\frac{\log N}{\log(B+1)})$ passes in the worst case), although much fewer than that if we access one item at a time. On the other hand, sequential search has optimal performance in the data stream model (single pass,

$O(1)$ memory, $O(1)$ processing time per element) despite its poor performance ($O(\frac{N}{B})$ block transfers) in the external memory model.

1.4 Communication Complexity

Due to the many constraints imposed by the data stream model, it is not surprising that some problems do not have efficient solutions in the data stream model.

One of the most powerful methods for proving such limits in the data stream model is that of communication complexity, and most lower bounds for data stream algorithms are obtained by reducing the problem to an appropriate communication complexity problem. For example, the seminal paper of Alon et al. [1996] is one of the first to use communication complexity to prove lower bounds on data stream algorithms.

Here we give a very rudimentary introduction to communication complexity, a comprehensive treatment of the subject can be found in the book by Kushilevitz and Nisan [1997].

1.4.1 Basic Model

Communication complexity aims at studying the amount of communication bits that has to be exchanged between the participants of a system in order to perform certain tasks. It was introduced by Yao [1979], who first investigated the following problem. Suppose there are two players with unlimited computing power, Alice and Bob. Alice possesses an n-bit string x and Bob another n-bit string y. Neither knows the other's input, and the goal is for one of them (say Bob) to compute a function $f(x, y)$ with the least amount of communication between them[1].

Given the function they are to compute, what Alice and Bob have to do – before they know their inputs x, y – is agree upon a protocol P (which depends only on f) for communication.

[1]Not to confuse with information theory, where an algorithm is given messages that have to be transmitted over a noisy channel, and the goal is to find a coding scheme so that the messages can be transmitted robustly with minimum communication overhead. To study the communication complexity of a problem, the channel is assumed to be clear and the main task for the participants is to determine what message to send and in what order.

At each stage, the protocol P must determine which player sends a bit of communication next. To avoid contention, this must depend solely on the bits communicated so far as this is the only knowledge common to both Alice and Bob. In addition, if it is Alice's turn to communicate a bit, the protocol must specify what she sends, which depends on the communication so far as well as on Alice's input x. Similarly, if it is Bob's turn, the protocol must specify what he sends; this again solely depends on the communication so far and on y, his input. In the end, the protocol must specify the value of $f(x, y)$.

The cost of a protocol P on input (x, y) is the number of bits communicated by P on input (x, y). The cost of a protocol P is the worst case cost of P over all inputs (x, y), that is, the maximum number of bits exchanged between Alice and Bob. The complexity of f is the minimum cost of any protocol that computes f.

Note that here we are not concerned about the amount of computation, or the size of the computer memory used. These assumptions help us concentrate on the core issue of communication required for distributed computations. Despite its apparent simplicity, this basic model turns out to already capture many of the fundamental issues related to the complexity of communication and that results proved in this model can be often extended to more complicated scenarios.

Of course Alice can always send her whole n-bit string to Bob, who then computes the function f, but the idea here is to find clever ways of calculating f with less than n bits of communication.

Next are two classic communication complexity problems and proofs of their respective communication complexities. As we will see, the same techniques used in the proofs are also applicable to proving the lower bounds of a wide range of data stream problems.

1.4.2 Equality Function

The first example concerns the communication complexity of the *Equality Function.*

$$f(x, y) = \begin{cases} 0 & : \quad x \neq y \\ 1 & : \quad x = y \end{cases}$$

Although communication can happen in both directions between Alice and Bob,

to simplify analysis, we will assume a simple one-way version of this game: Alice must send Bob a sequence of bits, and then Bob must send back the one-bit answer.

Note that we can identify any function $f : \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}$ with a $2^n \times 2^n$ boolean matrix F. Every row of this matrix corresponds to a possible input value for Alice; every column represents a possible input value for Bob. For example, if f is the equality function, then F is the identity matrix.

Lemma 1. *(Kushilevitz and Nisan [1997]) (One way communication complexity of the equality function) In the worst case, Alice must send at least n bits in order for Bob to correctly compute the equality function $f(x, y)$.*

Proof. Every row of the matrix F is different. If Alice transmits $k < n$ bits, then Bob can only divide the rows of F into $2^k < 2^n$ equivalence classes. In other words, there are at least two rows x and x' which Bob cannot distinguish. Suppose Bob's input $y = x$, then he will transmit the same value for the inputs (x, x) and (x', x), so the protocol is incorrect. □

This proof introduces an important lower bound technique called *fooling pair*: two inputs that cannot be distinguished by a protocol that ends too soon. Since there is an obvious n-bit algorithm– Alice transmits her whole string to Bob–we have the exact one-way communication complexity of n bits for the equality function.

It is easy to see that Lemma 1 is a special case of the following more general result.

Theorem 1. *(Kushilevitz and Nisan [1997]) If F has N distinct rows, then in the worst case, Alice must transmit exactly $\lceil \log N \rceil$ bits in order for Bob to correctly compute $F(x, y)$.*

1.4.3 Time-Space Tradeoffs for Turing Machines

In many models of computation it is possible to imagine communication between different stages of the computation. This communication is usually realized by one part of the computation leaving the computing device in a certain state and the other part of the computation starting from this state. The amount of information "communicated" this way can often be quantified as the "space" of this model; and the

number of times such a communication takes place relates to the "time" of the model. We then usually get time-space tradeoffs by utilizing the communication complexity lower bound. In this section, we show that using communication complexity results, it is possible to prove lower bounds for very simple problems in the one-head, one-tape Turing machine model. This version of the Turing machine has a read-only input tape of length n, with a single read head, and a work tape of unbounded length, with a single read/write head. The *time* taken by the Turing machine is as usual the number of state transitions; the *space* is the number of cells of the work tape that the machine accesses.

Theorem 2. *(Kushilevitz and Nisan [1997]) Let L be a language over alphabet $\Sigma = \{0,1\}$: $L = \left\{x\Sigma^{|x|}x \middle| x \in \{0,1\}^*\right\}$, i.e., the set of binary strings composed of three substrings: the first and the last substrings are identical, between which is an arbitrary string of the same length. Suppose there is a one-tape Turing machine that recognizes the language in time $T(n)$ and space $S(n)$. Then $T(n) \cdot S(n) = \Omega(n^2)$.*

Proof. Let M be a Turing machine that recognizes L in time $T(n)$ using space $S(n)$. Suppose M has q states. We can construct from M a communication protocol for the n-bit equality function with complexity $O(S(n)T(n)/n)$ as follows.

Given strings x and y of length n, Alice and Bob will simulate M with the input string $x\Sigma^n y$, each using only their own portion of the string. Alice begins by simulating M on the input string $x\Sigma^n?$. As soon as the input read head moves to the ? character, Alice transmits the current state and the content of the work tape ($S(n)$ bits) to Bob. Now Bob continues the simulation as though the input string is $?\Sigma^n y$, starting the read head at the first symbol of y. When Bob's simulation reads the symbol ?, Bob sends the current state and the contents of the current work tape back to Alice. If the Turing machine accepts, then Alice and Bob must have the same string.

The Turing machine must take at least n steps between any pair of transmissions. Thus, the total number of transmissions is at most $T(n)/n$. Each transmission consists of $S(n)$ bits. Thus, the total number of bits transmitted is $O\left(S(n)T(n)/n\right)$. On the other hand, we know that any protocol for the n-bit equality function requires at least n bits to be transmitted. It follows that $n = O\left(S(n)T(n)/n\right)$, or equivalently,

$S(n)T(n) = \Omega(n^2)$. □

Although the proof techniques may seem quite simple, this tradeoff lower bound is tight as this language can be recognized in linear time using linear space by copying the first half of the input to one of the read/write tapes and then checking that the first half matches the second half.

1.5 Probability Theory

Because of the constraints in the data stream model, it shall not be surprising that many data stream algorithms in the literature are randomized in nature, and the results they compute are often probabilistic and/or approximate. In this section we review some of the fundamental theorems in probability theory which are essential to the analysis of such data stream algorithms. For a more thorough treatment of the subject, please refer to Motwani and Raghavan [1995].

Markov Inequality

Theorem 3. *Let X be a random variable assuming only non-negative values. Then for all $t \in R^+$*

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

Equivalently,

$$\Pr[X \geq kE[X]] \leq \frac{1}{k}.$$

Chebyshev's Inequality

Theorem 4. *Let X be a random variable with expectation μ_X and standard deviation*[2] *σ_X. Then for any $t \in R^+$*

$$\Pr\left[|X - \mu_X| \geq t\sigma_X\right] \leq Pr\left[(X - \mu_X)^2 \geq t^2\sigma_X^2\right].$$

[2]For a random variable X with expectation μ_X, its variance σ_X^2 is defined to be $E[(X - \mu_X)^2]$. The *standard deviation* of X, denoted σ_X, is the positive square root of σ_X^2.

Chernoff Bound

Theorem 5. *Let $X_1, X_2, \ldots, X_n$ be independent Poisson trials*[3] *such that, for $1 \leq i \leq n$, $Pr[X_i = 1] = p_i$, where $0 < p_i < 1$. Then, for $X = \sum_{i=1}^{n} X_i, \mu = E[X] = \sum_{i=1}^{n} p_i$, and any $\delta > 0$*

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

1.6 Summary of Contributions

- In Chapter 3, we study the problem of how to reliably estimate the frequencies of frequent items in data streams. The previous research has been focused on algorithms that compute probabilistic results. We show that the frequencies can be estimated with deterministic accuracy linear to the number of internal memory locations. We also show that this is nearly optimal since there is an almost matching lower bound for this problem.

- We prove a lower bound on the expected accuracy of any randomized frequency estimation algorithm which is about 6 times that of the deterministic algorithm. This is contradictory to the previous belief that some randomized algorithm may be much more accurate than deterministic frequency estimation algorithms.

- In Chapter 4, we present the first non-trivial algorithms and data structures that can answer approximate range mode and range median queries quickly. Then we extend these results to higher dimensional ($d \geq 2$) space in Chapter 5.

- In Chapter 6 we study the classic data stream problem of sorting, which was first studied in Munro and Paterson [1980]. There is a gap of a multiplicative factor of 4 between the previous lower bound and upper bound. We close this gap by showing a matching lower bound on the number of passes any algorithm must make to sort n numbers stored on a tape using $m < n$ internal memory locations. We also present the first probabilistic lower bound on the average number of passes any sorting algorithm will have to make over the data given a random input.

[3]That is, X_i are independent random variables each takes 0 or 1 only.

Chapter 2

Related Work

2.1 Frequency Moments

One of the first results in data stream research is the paper by Alon et al. [1996]. In that influential paper, the authors give upper and lower bounds for the space complexity of data stream algorithms for the problem of estimating the frequency moments of a stream. In this problem, the data stream is a sequence of elements $S = (a_1, \ldots, a_n)$, where each element is a member of a set $\mathcal{M} = \{1, \ldots, m\}$. For each element $i \in \mathcal{M}$, its frequency f_i is the number of times it occurs in the sequence. For each $k \geq 0$, the k-th frequency moment of the stream S is defined as

$$F_k = \sum_{i=1}^{m} f_i^k.$$

The frequency moments of a data set provide useful statistics on the data, and are important features in the context of database applications. For example, it is claimed in Haas et al. [1995] that virtually all query optimization methods in relational and object-related database systems require a means for assessing the number of distinct values of an attribute in a relation, i.e., F_0 for the sequence consisting of the relation attribute.

Some special cases of particular interest are F_0, i.e., the number of distinct elements in the stream; F_1 the length of the stream; F_2 a measure of how uneven the distribution is [1]; and finally F_∞ is defined to be the number of occurrences of the most frequent element

$$F_\infty = \max_{1 \leq i \leq m} f_i.$$

A trivial solution for computing frequency moments would be that of storing a counter f_i for each element i, and add up all f_i^k at the end. This can give exact F_k

[1] Given fixed n, F_k is maximized when all elements are of the same type, i.e., $F_k = n^k$; F_k is minimized when all elements have the same frequency, i.e., $F_k = n^k/m^{k-1}$.

for any k, but one of the problems with this naive approach is that its space usage is $m \log n$ memory bits (n is the length of the stream, m is the number of the distinct elements in that stream), which makes it unsuitable for many practical applications that involve large data sets.

Can we do better than this? Unfortunately, with the obvious exception of F_1 which can be computed using $\log n$ bits, no algorithm can do with much less space. In fact, following the well known fact that the deterministic communication complexity of the equality function $EQ(x, y)$ whose value is 1 if and only if $x = y$, where x and y are m-bit numbers, is $\Theta(m)$, Alon et al. [1996] proved that any deterministic algorithm that computes F_k ($k \neq 1$) exactly must use at least $\Omega(m)$ space. This pessimistic lower bound holds true even for a relaxed version of the problem, that is, any deterministic algorithm that computes an approximation of F_k up to a constant relative error. In fact, it is shown in Alon et al. [1996] that any algorithm outputs a number $\tilde{F}_k$ such that $|\tilde{F}_k - F_k| \leq 0.1F_k$ would use at least $\Omega(m)$ bits of space.

The question now is what about randomized algorithms?

A randomized algorithm A (ϵ, δ)-approximates F_k if A outputs a number $\tilde{F}_k$ for which $\Pr[|\tilde{F}_k - F_k| > \epsilon F_k] < \delta$. The idea behind most of the randomized algorithms is to construct a random variable, which can be computed under the given space constraints, whose expected value is F_k, and has small variance.

Since the publication of et al.'s paper in 1996, the field of data stream algorithms has expanded greatly, but the problems they studied are still of great interest. In fact the most recent result is by Indyk and Woodruff [2005], which closes the final open problems left behind by Alon et al.'s paper. Next is a summary of the results concerning frequency moments.

2.1.1 Counting Distinct Elements over Data Streams (F_0)

The problem of computing the distinct elements in a data stream with low space is valuable in several applications. Database query optimizers can use F_0 to find the number of unique values in a database possessing a certain attribute, without having to sort all the values. It also has applications in networks: For example, an Internet router may want to gather the number of distinct destination addresses

passing through it, as a large number of requests from too many distinct sources (spoofed IPs) may indicate a denial of services attack.

As the obvious approach of maintaining the set of distinct elements seen so far would not work if we do not have enough space to store the complete set, the only hope for practical solutions lies in that we are willing to accept that the count may be in error, but limit the probability that the error is large. And we want to estimate the count in an unbiased way, that is, insensitive to the distributions of the data.

Flajolet and Martin [1985] designed an algorithm for approximating F_0 using $O(\log m)$ bits of memory. The idea is to pick a hash function h that maps each of the n elements to $\log m$ bits, uniformly. For each element a, let $r(a)$ be the number of trailing 0's in $h(a)$. Record R, which is the maximum $r(a)$ seen so far, and estimate 2^R.

However, the analysis in their original paper is based on the assumption that explicit families of hash functions with very strong random properties are available. A version of the Flajolet-Martin algorithm has been proposed by Alon et al. [1996] which can be implemented and analyzed using very simple linear hash functions. In the same paper, using a similar randomized communication complexity argument, it is proved that any randomized algorithm for approximating F_0 up to an additive error of $0.1F_0$ with probability at least 3/4 must use at least $\Omega(\log m)$ memory bits. In other words, the $O(\log m)$ bound in the Flajolet-Martin algorithm is tight.

2.1.2 Counting Large Numbers Using Small Memory (F_1)

This is probably the easiest to compute of all frequency moments. The straightforward solution of using a counter of $\log n$ bits for computing the length of a data stream of at most n elements is satisfactory for most applications. Nevertheless, this can be further reduced if high accuracy is not critical. Morris [1978] proposed a probabilistic approximate counting algorithm that counts up to n using only $O(\log \log n)$ bits of memory. The idea is to use a counter C of $\log \log n$ bits and probabilistically increment the value of the counter. The counter is set to 1 at the beginning. Then, in each incremental step, the actual value C is incremented by 1 with probability 2^{-C} and remains unchanged with probability $1 - 2^{-C}$. It can be shown that after n steps

C should contain a good approximation of $\log n$ with an expected constant relative error. A detailed analysis of this algorithm is presented in Flajolet and Martin [1985] which establishes good convergence properties of the algorithm and allows to quantify precisely space-accuracy tradeoffs.

In fact this algorithm achieves the optimal space complexity for any probabilistic algorithm that estimates F_1 with constant relative error. The length F_1 of the sequence can be any number up to n, therefore the final content of the memory should admit at least $\Omega(\log n)$ distinct values with positive probability, giving the lower bound of $\Omega(\log \log n)$ bits for any randomized algorithm for approximating F_1.

2.1.3 Higher Frequency Moments ($F_{k\geq 2}$)

In Alon et al. [1996], the authors present the first sub-linear space algorithm that (ϵ, δ)-approximates F_k for $k \geq 2$. For any given $k \geq 2, \epsilon > 0$ and $\delta > 0$, the randomized algorithm computes in one pass and using

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

memory bits [2], a number $\tilde{F}_k$ so that the probability that $\tilde{F}_k$ deviates from F_k by more than ϵF_k is at most δ. A special case is that of $k = 2$, for which a one-pass algorithm exists to (ϵ, δ)-approximate F_2 using

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

bits of memory. In the same paper, the authors also show an $\tilde{\Omega}(m^{1-5/k})$ lower bound for $k > 5$. These results are followed by a line of research on the frequency moments. In particular, the problem of estimating F_k was shown to have a space lower bound of $\tilde{\Omega}(m^{1-2/k})$ for any $k > 2$ (Bar-Yossef et al. [2002], Chakrabarti et al. [2003]) and better upper bounds of $\tilde{O}(m^{1-1/(k-1)})$ and $\tilde{O}(m^{1-2/(k+1)})$ have been proved more recently (Coppersmith and Kumar [2004], Ganguly [2004]).

The gap is finally closed in Indyk and Woodruff [2005]. In this paper, the authors present the first one-pass $\tilde{O}(m^{1-2/k})$-space streaming algorithm for estimating F_k for

[2] This is often written shorthanded as $\tilde{O}(1)$ in the literature. $f(m) = \tilde{O}(g(m))$ means $f(m) = g(m)(\frac{1}{\epsilon})^{O(1)} \log^{O(1)} mn \log 1/\delta$; the $\tilde{\Omega}$ notation is defined similarly.

any real $k > 2$, matching the space lower bound (up to $\tilde{O}(1)$ factors) for the problem. Departing from the earlier algorithms for estimating F_k which are all based on the idea of constructing a single estimator which equals F_k in expectation, and to have small variance, the approach used in Indyk and Woodruff [2005] is to divide the elements into groups S_i (of randomized boundaries based on frequencies), such that the elements in group S_i have frequency $\approx (1+\epsilon)^i$. The contribution of S_i to F_k can be approximately estimated by $s_i \times (1+\epsilon)^{ik}$, where s_i is the size of $S_i, i = 1, \ldots, \log_{1+\epsilon} n$. The estimator for F_k is obtained by summing up all estimated contributions

$$\tilde{F}_k = \sum_{i=1}^{\log_{1+\epsilon} n} s_i \times (1+\epsilon)^{ik}.$$

More recently, Bhuvanagiri et al. [2006] proposed an algorithm that uses groups of deterministic boundaries in order to simplify the analysis.

2.2 Frequent Elements Query

As mentioned earlier, a natural extension to the frequency moment problem is when $k = \infty$, in which case

$$F_\infty = \max_{1 \le i \le m} f_i,$$

that is the most frequently occurring item in the stream. A related and more general problem is that of finding the set of most frequent elements in the stream. This problem comes up in the context of many networks applications, one example is that of search engines, where the data streams in question are streams of queries sent to the search engine and we are interested in finding the most frequent queries in some period of time (Google Zeitgeist [3] and Charikar et al. [2002]). The data stream here is so large that any memory intensive solutions such as sorting the stream or keeping a counter for each distinct element would be infeasible, and moreover we can afford to make only one pass over the data.

A wide variety of heuristics for this problem have been proposed, all involving some combination of sampling, hashing, and counting (see Gibbons and Matias [1999] for a comprehensive survey on the subject). However, none of these solutions have

[3] http://www.google.com/press/zeitgeist.html

clean bounds on the amount of space necessary to produce good approximate lists of the most frequent items, instead, most of them have a space usage that depends on the distribution of the frequency of the items in the data streams. In fact, the only algorithm that has theoretical guarantees on its space usage is the straightforward sampling algorithm, in which a uniform random sample of the data stream is kept as a list of items plus a counter for each item. If the same object is added more than once, we simply increment its counter, rather than adding a new object to the list.

It is easy to see that an approximate answer for a popular items query can be readily obtained by sampling, since the sample size needed to obtain a desired answer quality can be predetermined from the frequency threshold. For example, if $p < 1$ is the prespecified threshold percentage, then by Chernoff bounds, any value whose frequency exceeds this threshold will occur at least $c/2$ times in a sample of size c/p with probability at least $1 - e^{-c/8}$.

Two variants of the naive sampling algorithm were given by Gibbons and Matias [1998]. The *concise samples* algorithm keeps a uniformly random sample of the data, but does not assume that we know the length of the data stream beforehand. Instead, it begins optimistically assuming that we can include elements in the sample with probability $\tau = 1$. As it runs out of space, it lowers τ until some element is evicted from the sample, and continues the process with this new, lower τ'. The idea behind the algorithm is that, at any point, each item is in the sample with the current threshold probability, which is to adapt to the input stream as it is processed. At the end of the algorithm, there is some final threshold τ_f, and the algorithm gives the same output as the sampling algorithm with this probability. However, the value of τ_f depends on the input stream in some complicated way, and no clean theoretical bound for this algorithm is available.

The other variant of the sampling algorithm is the so-called *counting sample* algorithm, which adds one more optimization based on the observation that so long as we are setting aside space for a count of an item in the sample anyway, we may as well keep an exact count for the occurrences of the item after it has been added to the sample. This change improves the accuracy of the counts of items, but does not change who will actually get included in the sample.

So far we have intentionally avoided specifying what exactly are frequent elements. Naturally, the set of frequent elements should include the most frequently occurring element in the data stream. But the following lower bound implies that even to find this most frequent one is hard.

The space complexity of approximating F_∞

Again the lower bound is obtained by reducing the data stream problem to an appropriate communication complexity problem. Suppose x and y are two subsets of $\{1, 2, \ldots, m\}$. The disjoint function $DISJ_m(x, y)$ is defined to be 1 if $x \bigcap y = \emptyset$ and 0 otherwise. Using the fooling set technique it is fairly straightforward to prove that any deterministic protocol for two parties possessing x and y respectively to compute $DISJ_m(x, y)$ must use $\Omega(m)$ bits of communication. The same lower bound holds even for the probabilistic version of the problem, *i.*e., at the end of the communication, the correct value of the function is output with probability at least $1 - \delta$ (Kalyanasundaram and Schnitger [1987]). The space complexity of approximating F_∞ follows easily from the above communication complexity of $DISJ$ function.

Lemma 2. *(Alon et al. [1996]) Any algorithm that outputs, given a sequence A of at most $2m$ elements of $\mathcal{M} = \{1, \ldots, m\}$, a number $\tilde{F}_\infty$ such that the probability that $\tilde{F}_\infty$ deviates from F_∞ by at least $F_\infty/3$ is less than δ, for some fixed $\delta < 1/2$, must use $\Omega(m)$ memory bits.*

If finding the most frequent element is hard, what about finding the set of the most frequent elements? The only hope seems to lie in solving some relaxed version of the problem. In fact, most of the existing literature deals with two notions of approximating the frequent elements problem. One is the so-called *Top-k queries* (Gibbons and Matias [1998]), in which the goal is to output a list of $l \geq k$ elements such that the k most frequent elements are guaranteed to be included in the list. As it turned out, even this approximate version of the frequent problem is hard. Suppose, for example, that the kth most frequent element has almost the same frequency as the $(l+1)$st most frequent element. Then it would be almost impossible to find only l elements that are likely to have the top k elements.

Another variant is to find a list of k elements such that every element i in the list has a relatively large frequency: $f_i \geq (1-\epsilon)f_k$ (assuming the elements are sorted according to their frequencies, $f_1 \geq f_2 \geq \ldots \geq f_m$). A somewhat stronger guarantee on the output is that every item i with $f_i > (1-\epsilon)f_k$ will be in the output list with high probability. This problem has been extensively studied in databases community, in which this is often called *iceberg queries* (Park et al. [1995], Fang et al. [1998], Charikar et al. [2002]). Not surprisingly, most of the existing algorithms for iceberg queries have been optimized for finite stored data and make multiple passes over the data. The basic idea is the following: in the first pass, a set of possible frequent elements (including all the actual frequent elements and possibly some other less frequent elements) are selected. In the second pass, exact frequencies for only those elements are maintained and the set of the most frequent elements are confirmed. When adapted to data streams, where only one pass is allowed, these algorithms failed to provide any prior guarantees on the quality of their output.

We have seen that the frequent element query has lower bound of $\Omega(m)$ memory bits. This holds even for an approximate version of the problem, that is, finding the frequent element(s) whose frequency is at least some constant $(1-\epsilon)$ times the frequency of the actual frequent element(s). $\Omega(m)$ space usage implies that in the worst case we might need to monitor every class, which is not much better than the naive algorithm of assigning a counter for each of the m classes. This is because in the worst case, even the most frequent element(s) may have very low frequency in the stream, in order to capture this frequent element with low frequency, or in the case of approximate algorithms, elements with ever lower frequencies, the algorithm must have very fine granularity and end up monitoring every element class. On the other hand, in most practical applications, people are only interested in finding those "significant" elements,

This shift of interests from relatively most frequent elements to absolutely frequent elements in the stream yields another version of the approximation, that is, we are to find those elements whose frequency is over a certain threshold.

Manku and Motwani [2002] proposed two algorithms that compute the set of frequent elements in a single pass with prior error guarantees and for variable sized

streams. The algorithm accepts two user-specified parameters: a support threshold $s \in (0, 1)$, and an error parameter $\epsilon \in (0, 1)$ such that $\epsilon \ll s$. Let n denote the current length of the stream, i.e., the number of items seen so far. At any point of time, the algorithm will be asked to produce a list of items along with their frequencies. The answers produced by the algorithm will have the following properties:

1. All items whose frequency exceeds sn are output.

2. No item whose true frequency is less than $(s - \epsilon)n$ is output. There are no false positives.

3. Estimated frequencies are less than the true frequencies by at most ϵn.

The first algorithm, which they called *Sticky Sampling* algorithm, is a sampling based algorithm and the sampling rate increases logarithmically proportional to the size of the stream. It produces all items whose frequency exceeds s with probability at least $1 - \delta$ using at most $\frac{2}{\epsilon}\log(s^{-1}\delta^{-1})$ expected number of entries. Here the entries are of the form $(i, \tilde{F}_i)$, where $\tilde{F}_i$ estimates the frequency of element i so far. The second algorithm is a deterministic algorithm called *Lossy Counting* algorithm. It uses at most $\frac{1}{\epsilon}\log(\epsilon n)$ entries which grows logarithmically with n, the length of the stream.

We show in Chapter 3 a deterministic solution that can find the list of most frequent elements with guaranteed accuracy. The memory usage is linear to the desired accuracy and irrelevant to the length of the stream.

2.3 Range Aggregate Queries

In many applications, the items of interest are named by attributes. Many of these attributes have scalar values: *e.g.*, stock prices, source/destination IP addresses of Internet packets, etc. One natural way to query events of interest is to use range queries on these attributes. Some examples are: "What is the median price of the stock between 10:00am and 2:00pm" or "Find the most popular websites visited by users of our network in the past 24 hours."

Aggregates are not only an important class of queries in conventional database system applications but also serve as an important base for quantitative data mining

problems in many modern applications, such as financial market analysis and telecommunications. Approximate preprocessing of aggregates, thus has recently attracted a great deal of attention. Among which the most studied is the *median finding problem.*

Median Finding Problem

Given a set S containing n elements drawn from a totally ordered domain, the *median* of S is the minimum element that is greater than or equal to $\lfloor n/2 \rfloor$ elements of S. This is a special case of the *selection* problem which is to find the i-th order statistics of S, *i.e.,* the smallest element of S larger than or equal to $i-1$ elements of S. Obviously, the median is the $(\lfloor n/2 \rfloor + 1)$-st order statistics of S.

The selection problem is one of the most fundamental problems of computer science and it has been extensively studied. Selection is used as a building block in the solution of other fundamental problems such as sorting and finding convex hulls. However it was only shown in the early 70s by Blum et al. [1973] that the selection problem can be solved in $O(n)$ time. As $\Omega(n)$ time is clearly needed to solve the selection problem, the research on the selection problem since then has been primarily focused on counting the number of comparisons needed in the worst case to find the exact median.

A very natural setting for the selection problem is the *comparison model.* An algorithm in this model can access the input elements only by performing pairwise comparisons between them. The algorithm is only charged for these comparisons. All other operations are free. The comparison complexity of many selection problems is exactly known. It is clear, for example, that exactly $n-1$ comparisons are needed, in the worst case, to find the maximum or minimum of n elements. Exactly $n+\lceil \log n \rceil - 2$ comparisons are needed to find the second largest (or second smallest) element, and exactly $\lceil 3n/2 \rceil - 2$ comparisons are needed to find both the maximum and minimum elements of n elements (Knuth [1998]). However, the exact lower bound is much harder to get for the median. Blum et al. [1973] shows that selection of the k-th largest element out of n can be done using at most $5.43n$ comparisons. This paper also shows that at least $1.5n$ comparisons are required in the computation of the exact median. For an account of progress since then, see the survey by Paterson [1997].

The current best bounds are much tighter and are the product of sophisticated and deep analysis (see Paterson [1997], Dor and Zwick [1995, 2001]). The upper bound is $2.9423N$ comparisons (Dor and Zwick [1995]), and the lower bound $(2+\epsilon)N$, where ϵ is an extremely small constant of the order of 2^{-40} (Dor and Zwick [2001]), an extremely small number by most standards.

On the other hand, the idea of choosing an easily-computable approximate medians has shown to be extremely useful and been fruitfully applied in diverse disciplines such as computational statistics and databases.

Battiato et al. [2000] present an algorithm for the approximate median selection problem. With high probability, it returns a very close estimate of the true median. The running time is linear in the length n of the input. The algorithm makes fewer than $4n/3$ comparisons and $n/3$ exchanges on the average, and fewer than $3n/2$ comparisons and $n/2$ exchanges in the worst case. In fact, this is optimal (up to a constant factor) as computing an approximate median is shown to have an $\Omega(n)$ comparisons lower bound for any deterministic algorithm (Yao [1974]). However, this lower bound is easily beaten by resorting to randomization. The naive randomization algorithm, which outputs the median of a random sample of size $O(\frac{1}{\epsilon^2}\log\delta^{-1})$, uses a number of comparisons independent of n. For a comprehensive survey of this aspect of the literature, see the survey by Paterson [1997].

To be useful in the data stream model, a median finding algorithm must make as few passes as possible over the data and use as small amount of as possible memory. Pohl [1969] established that any deterministic algorithm that computes the exact median in one pass needs to store at least $n/2$ data elements. Munro and Paterson [1980] generalized this and showed that memory to store $\Theta(n^{\frac{1}{p}})$ elements is necessary and sufficient for finding the exact median in p passes. The same bound holds for any ϕ-quantile [4] for a constant ϕ. This result motivates a search for algorithms that produce approximate quantiles in a single pass with less memory.

First, Munro and Paterson [1980] developed a single pass algorithm for computing approximate quantiles. Jain and Chlamtac [1985] proposed a simple algorithm for computing quantiles in a single pass using only a constant amount of memory.

[4]The ϕ-quantile of a set of n numbers is the value that is greater than or equal to $\phi \times n$ members of the set.

Another one pass algorithm is proposed by Agarwal and Swami [1995]. The idea here is to adjust equi-depth histogram boundaries on the fly when they do not appear to be in balance. However, none of these algorithms provide *a-priori* guarantees on error. Alsabti et al. [1997] designed a quantile finding algorithm with error bounds. Manku et al. [1998] present another algorithm for computing approximate quantiles in a single pass, the algorithm provides explicit guarantees on error, and works for arbitrary value distributions and arrival distribution of the dataset.

We study in Chapter 4 and Chapter 5 the approximate median and approximate mode query problems in the sliding window model. In other words, the goal is not to achieve the optimal performance for one individual query, but the overall performance for a large number of queries on different subsets. By reusing the same precomputed median and mode for overlapping subsets, the algorithms achieve much faster query time using small amount of extra storage space.

Chapter 3

Frequency Estimation in Data Streams

We consider the problem of approximating the frequencies of frequently occurring elements in a data stream of length n using a memory of size $m \ll n$[1]. We show that when some data item a occurs $\alpha \times n$ $(0 \leq \alpha \leq 1)$ times in a stream of length n, the FREQUENT algorithm of Demaine et al. [2002], which was also independently discovered by Karp et al. [2003], can in fact be used to estimate a's frequency with an error of no more than $(1-\alpha)n/m$. We also give a lower bound of $(1-\alpha)n/(m+1)$ on the accuracy of any deterministic frequency estimation algorithm, which implies the FREQUENT algorithm is nearly optimal. Finally, we show that randomized algorithms can not be significantly more accurate since there is a lower bound of $\Omega((1-\alpha)n/m)$ on the expected accuracy of any randomized frequency estimation algorithm.

The problem of accurately maintaining frequency statistics in a data stream has applications in Internet routers and gateways, which must handle continuous streams of data that are much too large to store and process offline later. As an example, to implement fairness policies one might like to ensure that no user (IP address) of a router or gateway uses more than $\alpha = 1\%$ of the total available bandwidth. Keeping track of individual users' usage statistics would require (at least) one counter per user and there may be tens of thousands of users. However, the results in this chapter imply that, using only $m = 99$ counters, we can identify a set of users, all of which used more than 1% of the available bandwidth, although some users that used more than 1% of the bandwidth may be missing in this set, the set contains every user that uses more than 2% of the total bandwidth. If more accuracy is required, we could use $m = 990$ counters, and the threshold values become 1% and 1.1%, respectively. [2]

[1] A preliminary version of this chapter appeared in SIROCCO 2003 (Bose et al. [2003])

[2] The accuracy of counters, and the ability to prove that they are correct may be extremely useful, e.g., in proving that contractual obligations have been met.

3.1 Motivation

> *If we're keeping per-flow state, we have a scaling problem, and we'll be tracking millions of ants to track a few elephants.* -Van Jacobson, End-to-End Research meeting, June 2000.

We consider the problem of processing a data stream $x_1, \ldots, x_n$ of an unspecified number of *classes* in one pass. And we want to know how many elements belong to each class. This models the process of gathering statistics on Internet packet streams using a memory that is small relative to the number of classes (e.g., IP addresses) of packets. Network traffic measurements are crucial to the design, operation and control of the network. For example, such measurement information is essential for traffic engineering (e.g., rerouting traffic from congested links and upgrading overloaded links), monitoring (e.g., detecting denial-of-service attacks) and accounting.

The trend of ever-increasing link speed motivates three highly desirable performance criteria for high-speed network measurement (Estan and Varghese [2002], Cormode and Muthukrishnan [2004]): 1) a small amount of memory usage so it can be implemented in fast but usually very small SRAM due to its high cost; 2) a small number of memory accesses per packet to make on-line processing possible; and 3) scalability to a large key space size which can be billions of IP addresses and possibly much more with combinations of IP source-destination pairs, port numbers, etc.

The simple solution of keeping per packet class statistics is not viable because of the high cost of maintaining such data structures. On the other hand, it has been shown that in real world data distributions are often highly skewed, thus giving hope to more efficient solutions: although it is infeasible to accurately measure all classes on high speed networks, many applications can still benefit from accurately measuring only the few large classes (each accounts for more than a given threshold, say 1% of all packets). One can easily keep counters for a few large classes using a small amount of fast memory (SRAM). However, how does the device know which classes to track? If one keeps state for all classes to identify the few large classes, our purpose is defeated.

An ideal algorithm reports, at the end of the measurement interval, the class IDs and sizes of all classes that exceeded the threshold. A less ideal algorithm can fail in

three ways: it can omit some large classes (*false negative*), it can wrongly add some small classes to the report (*false positive*), and can give an inaccurate estimate of the counts of some large classes.

3.2 Applications

Our results for estimating the frequencies of large classes can potentially be used to solve the practical problems listed in this chapter with much better cost-effectiveness than existing solutions.

Real-time Traffic Monitoring

Network operators are constantly monitoring the traffic on their networks and looking for hot-spots in order to identify large traffic aggregates that can be rerouted to reduce congestion. Also, sudden increases in the traffic sent to certain destinations (the victims) may indicate an ongoing attack. For both traffic monitoring and attack detection, it may suffice to focus on large flows.

The monitoring infrastructure will have to do some computation on-line so as to minimize the buffer size and delay. The computation must be simple-at OC-768 speed ($\approx$ 40Gbps), a new packet arrives every 60 ns on average (assuming 300-byte packets). This allows only a few dozen instructions per packet on the fastest processor, or even fewer memory accesses to the fastest memory (today's SRAM has access times of about 10 ns).

Instead of collecting information about each packet, the network traffic measurement is usually done on a per flow basis to reduce the amount of information that needs to be stored and processed. A flow is a stream of packets that share some common attributes. The most common definition of a network flow is that classified by the 5-tuple in the packet header: *source IP address*, *destination IP address*, *source port number*, *destination port number*, *protocol type*. The assumption is that a flow corresponds to an application exchange. The packets that have the same 5 tuple are grouped together and the only information stored is these 5-tuples and the flow statistics such as the start/end time of the flow and the number of packets in the flow, etc. Doing so enough detailed information about the traffic is preserved while

the amount of resources used is much less than if the complete packets are copied and stored. Iannaccone et al. [2001] shows savings between 3 and 4 times are achieved by maintaining per-flow records.

The approach advocated by the standard body Real-Time Flow Measurement (RTFM) and the Working Group of the Internet Engineering Task Force (IETF) is to instrument routers to add flow meters at either all or selected input links (Brownlee et al. [1999]). Today's routers offer tools such as Cisco's NetFlow[3] that give flow level information about traffic.

As link speeds and the number of flows continue to increase, even keeping a counter for each flow is too expensive (using SRAM) or slow (using DRAM). A study done in as early as 1999 on large regional networks showed more than 1.7 million concurrent flows between end host pairs in a one hour period (Fang and Peterson [1999]). Cisco's NetFlow, which keeps its flow counters in DRAM, solves this problem by sampling: only sampled packets result in updates. But NetFlow sampling has problems of its own since it affects measurement accuracy.

Usage Based Accounting

Pricing for network usage has been subject to much discussion since the beginning of the commercialization of the Internet (Mackie-Mason and Varian [1993], Odlyzko [2001], Sun and Varaiya [2003]). There are three commonly adopted pricing schemes: 1) duration based (i.e., price based on user's time online); 2) usage based (x dollars per MB of data traffic); and 3) flat rate (i.e., a fixed price of y dollars per month no matter how long and how much the user uses the network).

The pricing model that measures subscribers' time online (in minutes or hours) is most popular at the time when dial-up is used by majority of the customers to access the Internet. It is essentially an extension of circuit-switched pricing models such as those used in the public telephone network and hence a technically effective pricing method most suitable for dial-up networks, where network capacity is determined based upon the number of simultaneous connections supported. Because dial-up networks are dependent upon circuit-switching to establish their connection to users,

[3]http://www.cisco.com

billing based on hours of usage is a fair and viable method for equating network costs to subscriber usage. However, today's dominant access technologies (DSL, Cable modem connection, etc.) no longer rely on circuit switching, instead, they all provide dedicated always on access, yielding the irrelevance of pricing based upon time online. It is natural for an ISP whose costs are now largely related to its bandwidth to shift from time based pricing to usage based pricing, and it has been shown that usage based pricing can improve overall utility. However, usage based pricing is not without its problem: it is not scalable because it is difficult to track all users's traffic at high speed. Because of its simplicity, flat rate pricing has been quickly adopted by many ISPs for broadband Internet access, who soon discovered that while the model of flat pricing has been warmly welcomed by the fast growing number of the high-speed, always on broadband service subscribers, it has left networks open for abuse by heavy bandwidth users who happily take advantage of the zero incremental cost of network usage. This has created a situation where the cost of providing service can surpass the revenue that is generated. Service providers are motivated to deter subscribers from using the network excessively under the flat-rate plan, some of the practices used by ISPs include prohibiting multiple computers from using the same access line or reducing available bandwidth for a subscriber if usage is too high, besides the added operating complexity, it risks discouraging the network usage by a majority of the customers. Although there is no simple solution to this problem, another possibility may lie in a hybrid scheme that we measure all aggregates that are above $z\%$ of the link; such traffic is subject to usage based pricing, while the remaining traffic is subject to flat rate pricing.

Fair Queueing

The fundamental task for a scheduling algorithm running at a router is to answer the following question: "Which packet gets sent out next?" The choice of which flow is the next one to use an outbound link has direct impact on the bandwidth allocation among flows and quality of services experienced by users. And one of the most important design criteria of scheduling algorithms is the fair allocation of bandwidth among flows.

One particular form of scheduling, called *Fair Queuing* (FQ), is widely adopted in computer networks and statistical multiplexing to allow several data flows to fairly share the link capacity. The advantage over conventional first in first out (FIFO) queuing, is that an ill-behaved flow (consisting of large data packets or bursts of many packets) will only punish itself and not other flows. More specifically, it achieves *max-min fairness*: no flow receives more than its request, r_i, all allocations converge to some value α such that all requests $r_i \leq \alpha$ are given a rate equal to r_i, while all inputs $r_i > \alpha$ are given a rate equal to α. Of course, the total bandwidth available C (the output links rate) is distributed amongst the n input flows such that

$$\sum_{i=1}^{n} \min(r_i, \alpha) = C.$$

Another important practical consideration in the design of scheduling algorithms concerns state management. The key concern here is that schemes that maintain per-flow state are not scalable, on the other hand, at a smaller time scale, scheduling mechanisms seeking to achieve max-min fairness need to detect and penalize flows sending above their fair rate. Unfortunately for FQ, although it achieves fair bandwidth allocations, it requires per-flow queueing and per-flow state management in the router, which is often prohibitively expensive.

Data Mining

One of the most notable applications of data mining is the so-called *Beer and Diapers* problem, which deals with association rules discovery (Agarwal et al. [1993]). Consider a supermarket that carries a large collection of items (a typical Wal-Mart super center sells more than 100,000 different items). It also runs a large database of customer purchasing records. Each record (often called *basket* in data mining literature) is a small set of the items that one customer buys together on one day. One question is to find sets of items that appear "frequently" in the baskets. This information on associations between purchased items can help the store's management to make decisions as to how to place merchandise on shelves and what to put on sale (e.g., run sale on beer to boost the sale of diapers).

Commonly the association finding algorithms attempt to find all sets of items which occur in at least a fraction α (e.g., 1%) of all purchase records. Sets whose

frequency exceeds α are called *frequent itemsets.* One observation is that if a set s is frequent, then any subset of s must also be, and most association finding algorithms attempt to exploit this fact. As an example, one of the most notable data ming algorithms is the so-called *A-priori* algorithm, which finds all frequent itemsets levelwise: (Suppose there are n baskets in the database.)

1. In the first pass, we find the items (set of size 1) that appear in at least $\alpha \times n$ baskets. Call this set L_1.

2. Pairs of items in L_1 become the candidate pairs for the second pass. The pairs in L_1 whose count reaches $\alpha \times n$ are the frequent itemsets of size 2, L_2.

3. The candidate triples are those sets $\{A, B, C\}$ such that all of $\{A, B\}$, $\{A, C\}$ and $\{B, C\}$ are in L_2. In the third pass, count the occurrences of triples and those with a count of at least $\alpha \times n$ are the frequent triples, L_3.

4. Proceed until the sets become empty. L_i is the frequent sets of size i.

The straightforward implementation of the A-priori algorithm would require there is enough main memory to count occurrences of all items in the first pass, all candidate pairs in the second pass, and all candidate triples in the third pass, etc. Although it could be argued that after the first pass there will unlikely be many candidate itemsets left to participate subsequent passes, leading to a memory footprint that is much smaller than that in the theoretical worst case scenario, still even the first pass would be computationally very expensive if we were to keep track of the counts of all hundreds of thousands different items.

Web Search Engine

Google publishes some of the most popular queries sent to its search engine on its Google Zeitgeist site. Ask.com, another search engine, also lists a list of the most popular search terms each week [4]. In Henzinger [2003] six open algorithmic problems were identified to have great importance in web search engines, one of which being finding queries with largest increase or decrease from one time period (e.g., a week)

[4]Ask.com IQ (Interesting Quotes): http://about.ask.com/en/docs/iq/iq.shtml

over the next. These top gainers and losers in search queries show interesting trends in user interests.

MapReduce (Dean and Ghemawat [2008]) is a programming model and an associated implementation used at Google for processing large amounts of data (such as billions of web pages) and generating keyed or indexed query results (such as the indexes used for Google's web search). Users specify a *Map* function that iterates over a list of independent elements and performs a specified operation on each element. Each element is mapped to a key-value pair and a subsequent *Reduce* function consolidates all intermediate key-value pairs sharing the same keys to single key-value pairs. Because each element is operated on independently and the original list is not being modified, it is very easy to perform a map operation in parallel. A reduce operation, on the other hand, takes a list and combines elements according to some algorithm. Since a reduce operation always ends up with a single answer, it is not as parallelizable as a map function, but the large number of relatively independent calculations means that reduce functions are still useful in highly parallel environments.

It is shown that many real world search engine tasks are expressible in this model and MapReduce aims at allowing programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system (i.e., Google clusters). For example consider the problem of counting the occurrences of each word in a large collection of search queries. The *Map* function maps each query to a list of words each with an associated count of occurrences ($< word, 1 >$). The *Reduce* function sums together all values for the same word and outputs a $< word, total\_count >$ pair.

Notice that all these applications have something significant in common. First, the amount of data is huge and keeping track of every data element is very expensive if not entirely impossible (billions of packets belonging to millions of flows, billions of search queries from millions of word combinations, etc.); on the other hand, for most practical applications the detailed knowledge of just the few "heavy hitters" is probably sufficient, for example, despite the large number of flows, a common observation found in many network measurement studies is that a small percentage of flows accounts for a large percentage of the traffic (Feldmann et al. [2001], Zhang

et al. [2002]). In particular, Zhang et al. [2002] shows that the 10% fastest flows account for between 30% and 90% of all bytes transfered. Second, although desirable, exact answers are not always necessary (e.g., how much would the marketing strategy be different if in fact 79% percent of diaper buyers will also buy beer instead of 80% as reported by the algorithm). In most cases, we are willing to trade a small amount of accuracy for much higher efficiency (lower infrastructure cost and/or less time).

3.3 Previous Results

An early work, particularly relevant to this thesis, is the majority element algorithm by Fischer and Salzberg [1982]. In this early paper, the authors showed that, using one counter and making one pass through a data stream of n elements, it is possible to determine an element a such that, if any element occurs more than $n/2$ times, then it is a. On the other hand, there is no guarantee on a's actual frequency if there is no element occurs more than $n/2$ times.

Demaine et al. [2002] generalized Fischer and Salzberg's algorithm to an algorithm which they call FREQUENT. The same algorithm was also independently proposed in Karp et al. [2003]. The authors showed that, using m counters, the FREQUENT algorithm determines a set of m candidates that contain all elements that occur more than $n/(m+1)$ times. The output of FREQUENT is therefore a list of elements including all of these heavy users and possibly some light users. To determine all heavy users in one pass, it is pointed out in Karp et al. [2003] that we cannot do better than keeping a counter for each user, which requires $\Omega(c)$ memory, where c is the number of different users. In the case of Internet packet stream, the number of users (IP addresses) is substantially larger than m, the maximum number of candidates. Hence, the algorithm needs much more space than the size of the output. The difficulty with determining all heavy users is that at any point in time, many users may have nearly equal (small) number of occurrences, and therefore equal chances to become a heavy user later on, with no way of predicting which one(s) whose frequency will eventually surpass the threshold (*e.g.*, 1% of the total traffic) and become a heavy user, the algorithm must remember the exact count of each user at any point of time.

In applications such as network traffic measurement and billing, it is important

to not only identify all large flows but also to estimate the frequencies of these large flows. Manku and Motwani [2002] proposed two algorithms for computing frequencies of all large flows above a user-specified threshold. The *Sticky Sampling* algorithm is probabilistic, with probability $1 - \delta$, the algorithm identifies all elements whose true frequency exceeds a user specified threshold $s \in (0, 1)$ using at most $\frac{2}{\epsilon} \log(s^{-1}\delta^{-1})$ expected number of counters, where $\epsilon \in (0, s)$ is the maximum error in the estimated frequencies. The other algorithm called *Lossy Counting* is deterministic in the sense that it outputs all flows above the threshold. Regardless of the threshold s, it achieves the same accuracy ϵ using at most $\frac{1}{\epsilon} \log(\epsilon n)$ counters.

Other work on the particular problem of estimating frequencies in data streams includes Fang et al. [1998] which proposes heuristics to compute all values above a certain threshold. Several algorithms are proposed in Charikar et al. [2002] for computing the top k candidates under the Zipf distribution. Estan and Varghese [2002] attempt to identify a set of packet classes that are likely to contain the most frequently occurring packet classes, and give probabilistic estimates of the expected count value in terms of a user selected threshold.

3.4 Main Results

We are concerned with the accuracy of frequency estimation algorithms. More formally, we consider algorithms that process the stream $x_1, \ldots, x_n$ one element at a time. The algorithm has a memory of fixed-size and has access to $m(\ll n)$ integer counters, each of which can be labeled with a element class. If a counter is labeled with some element class a then we say that counter is *monitoring* a. While processing an element, the algorithm may modify its memory, perform equality tests on element classes, increment or decrement counters and change the labels of counters. However, other than comparing element classes and storing them as counter labels, the algorithm may not do any other computations on storage of element classes. After the algorithm completes, the *counter value* for a element class a is the value of the counter monitoring a. If no counter is monitoring a then the counter value for a is defined to be zero. A frequency estimation algorithm is *k-accurate* if, for any class a that appears c_a times, the algorithm terminates with a counter value $\tilde{c}_a$ for a that

satisfies

$$c_a - k \leq \tilde{c}_a \leq c_a \ . \tag{3.1}$$

In general, with m counters, no algorithm is better than $n/(m+1)$ accurate, this is because if in $m+1$ classes each occurs $n/(m+1)$ times then at least one of those classes will have a counter value of 0 when the algorithm terminates.

However, this argument breaks down when we consider the case when some particular packet class a occurs $c_a \geq \alpha n$ times, for some $\alpha > 1/(m+1)$. In this case, it may be possible for the algorithm to report the number of occurrences of a (and other elements) more accurately. We explore this relationship between accuracy and α. Our results are outlined in the next paragraph.

In Section 3.5 we show that the FREQUENT algorithm of Demaine et al. [2002] is in fact $(1-\alpha)n/m$-accurate, where αn is the number of times the most frequently occurring element class appears in the stream. In Section 3.6 we give a lower bound of $(1-\alpha)n/(m+1)$ on the accuracy of any deterministic frequency estimation algorithm and a lower bound of $(1-\alpha)\Omega(n/m)$ on the accuracy of any randomized frequency estimation algorithm. This latter result solves an open problem posed in Demaine et al. [2002] about whether randomized frequency estimation algorithms can be more accurate than deterministic ones. In Section 3.7 we summarize and conclude with open problems.

3.5 The *FREQUENT* Algorithm

The FREQUENT algorithm in Demaine et al. [2002] uses m counters. When processing an element x_i, the following rules are applied in order:

1. If there is a counter monitoring class x_i then increment that counter, otherwise
2. if some counter is equal to 0 then set that counter to 1 and have it monitor class x_i, otherwise
3. decrement all counters by 1.

A simple way to visualize this algorithm is to imagine a set of m buckets that hold colored balls. When a new ball arrives we either place it in the bucket that contains

balls of the same color (Case 1), place it in an empty bucket (Case 2) or discard one ball from every bucket as well as the new ball (Case 3).

To analyze the accuracy of this algorithm, we first provide a rough upper-bound on the accuracy and then use this upper-bound to bootstrap a better analysis. Let d be the number of times Case 3 of the algorithm occurs. No counter is ever less than 0, and each of Case 1 and Case 2 increments exactly one counter. Therefore, if C is the sum of all counters when the algorithm terminates, then

$$C = n - (m+1)d \geq 0 \ ,$$

so that

$$d \leq \frac{n}{m+1} \ .$$

It follows immediately that for any class a that occurs c_a times, the counter monitoring a has a value of at least

$$\tilde{c}_a \geq c_a - d \geq c_a - \frac{n}{m+1} \ .$$

Suppose that $c_a = \alpha n$ for some $\alpha \geq 1/(m+1)$. Now we can repeat the above argument, since we have just shown that

$$C = n - (m+1)d \geq \alpha n - \frac{n}{m+1} \ ,$$

so that

$$d \leq \frac{(1-\alpha)n}{m+1} + \frac{n}{(m+1)^2} \ .$$

and the value of $\tilde{c}_a$ satisfies

$$\tilde{c}_a \geq \alpha n - d \geq \alpha n - \left(\frac{(1-\alpha)n}{m+1} + \frac{n}{(m+1)^2} \right) \ .$$

In general, we can repeat the above argument k times to show that

$$\tilde{c}_a \geq \alpha n - \sum_{i=1}^{k} \frac{(1-\alpha)n}{(m+1)^i} - \frac{n}{(m+1)^{k+1}} \ .$$

In particular, as $k \to \infty$, we obtain

$$\tilde{c}_a \geq \alpha n - \frac{(1-\alpha)n}{m}.$$

Now, since $\tilde{c}_a$ is clearly never greater than c_a, we have

$$c_a - \frac{(1-\alpha)n}{m} \leq \tilde{c}_a \leq c_a \ .$$

In other words, the output $\tilde{c}_a$ is $(1-\alpha)n/m$-accurate.

Finally, we observe that the above analysis gives an upper-bound on d, and this gives an upper bound on the accuracy of the counter value for a. However, the upper bound on d also gives an upper bound on the accuracy of *any* counter, not just the counter for a. This implies our first result.

Theorem 6. *For any stream in which some element occurs at least αn times, the* FREQUENT *algorithm is $(1-\alpha)n/m$-accurate.*

3.6 Lower Bounds on Accuracy

In this section we give lower bounds on the accuracy of deterministic and randomized frequency estimation algorithms.

3.6.1 A Deterministic Lower Bound

Here we give a lower bound for deterministic frequency estimation algorithms by using an adversary argument. Our adversary builds two distinct streams that the algorithm cannot distinguish between.

Our adversary uses $m+2$ element classes and builds its streams in two parts (see Figure 3.1). The first part of both streams is of length $(1-\alpha)n$ and consists of the first $m+1$ element classes each occurring the same number of times, so that each class occurs $(1-\alpha)n/(m+1)$ times. At this point the two streams diverge. In the first stream, the adversary adds αn occurrences of the unique element class a of the $m+1$ first classes that is not being monitored by the algorithm after processing the first part of the stream. In the second stream, the adversary adds αn occurrences of the unique element class z that does not appear in the first part of the stream.

Observe that, since neither a nor z is stored in any of the algorithm's counters after processing the first part of the stream, the only information the algorithm obtains by reading the last element of the stream is that it is not being monitored. Therefore,

$$\begin{array}{ccc} abcd\cdots yabcd\cdots y & \cdots & abcd\cdots yaaaaaa\cdots a \\ \underbrace{abcd\cdots yabcd\cdots y}_{m+1 \quad m+1} & \cdots & \underbrace{abcd\cdots y}_{m+1}\underbrace{zzzzzz\cdots z}_{\alpha n} \end{array}$$

Figure 3.1: The adversary's two streams.

since the algorithm is deterministic, its counter value $\tilde{c}_a$ for a on the first stream will be equal to its counter value $\tilde{c}_z$ for z on the second stream. However, in the first stream a occurs

$$c_a = (1-\alpha)n/(m+1) + \alpha n$$

times and in the second stream, z occurs $c_z = \alpha n$ times. In order to be accurate at all the algorithm must terminate with a counter value

$$\tilde{c}_a = \tilde{c}_z \leq c_z = c_a - (1-\alpha)n/(m+1).$$

In other words, the algorithm is not better than $(1-\alpha)n/(m+1)$-accurate for the first stream.

Theorem 7. *For any deterministic algorithm, there exists a stream in which some element a occurs $c_a \geq \alpha n$ times, but the algorithm reports a value $\tilde{c}_a$ such that*

$$\tilde{c}_a \leq c_a - (1-\alpha)n/(m+1).$$

3.6.2 A Randomized Lower Bound

Next we give a lower bound for randomized algorithms. We do this by providing a probability distribution on input streams such that the expected accuracy of *any* deterministic algorithm on this distribution is at best $(1-\alpha)cn/m$. Since any randomized algorithm is just a probability distribution on deterministic algorithms, the lower bound therefore holds for randomized algorithms as well.[5] The distribution we use is a probabilistic version of our deterministic construction.

Our distribution uses two constants $1 < c_1 < c_2$ that will be specified later. Each stream of our distribution is a two part data stream made up of $c_2 m$ element classes. The first part of all streams is identical. As before, it is of length $(1-\alpha)n$, and it

[5]Technically, this is an application of *Yao's Principle* (Yao [1977]).

consists of the first $c_1 m$ element classes each occurring an equal number of times, so that each class occurs $(1-\alpha)n/c_1 m$ times. For the second part of the sequence, we select a element class uniformly at random from all $c_2 m$ classes and make that class occur αn times.

Let a be the element class chosen to make up the second part of the sequence. Immediately after the first part of the sequence has been processed by the algorithm, there are three cases to consider:

1. The algorithm has a counter that is monitoring a. Since the algorithm has only m counters, this happens with probability at most

$$p_1 \leq \frac{m}{c_2 m} = \frac{1}{c_2} \ ,$$

and the number of occurrences of a is

$$c_1 = (1-\alpha)n/c_1 m + \alpha n.$$

2. The algorithm does not have a counter monitoring a and a comes from the first $c_1 m$ element classes. This happens with probability at least

$$p_2 \geq \frac{(c_1 - 1)m}{c_2 m} = \frac{c_1 - 1}{c_2} \ ,$$

and the number of occurrences of a is also

$$c_2 = (1-\alpha)n/c_1 m + \alpha n.$$

3. The class a does not come from the first $c_1 m$ element classes (so the algorithm is not monitoring a). This happens with probability

$$p_3 = 1 - \frac{c_1}{c_2} \ ,$$

and the number of occurrences of a is $c_3 = \alpha n$.

Let $\tilde{c}_a$ be the value output by the algorithm for class a. Since we are proving a lower bound, we can assume that in Case 1, the algorithm answers with perfect accuracy, i.e.,

$$\tilde{c}_a = (1-\alpha)n/c_1 m + \alpha n.$$

However, if the algorithm is not monitoring class a (Cases 2 and 3) then it cannot distinguish between Cases 2 and 3. Since the algorithm is deterministic, it must output the same counter value $\tilde{c}_a$ in both cases. Therefore, the expected error made by the algorithm is at least

$$\begin{aligned}
\mathbf{E}\left[|\tilde{c}_a - c_a|\right] &\geq p_1 \times 0 + p_2 \times |\tilde{c}_a - c_2| + p_3 \times |\tilde{c}_a - c_3| \\
&\geq p_2 \times \left|\tilde{c}_a - \left(\frac{(1-\alpha)n}{c_1 m} + \alpha n\right)\right| + p_3 \times |\tilde{c}_a - \alpha n| \\
&= p_2 \times \left|x_a - \left(\frac{(1-\alpha)n}{c_1 m}\right)\right| + p_3 \times |x_a| \\
&\geq \frac{c_1 - 1}{c_2} \times \left|x_a - \left(\frac{(1-\alpha)n}{c_1 m}\right)\right| + \left(1 - \frac{c_1}{c_2}\right) \times |x_a|
\end{aligned}$$

where $x_a = \tilde{c}_a - \alpha n$. Setting $c_1 = 1 + \sqrt{2}/2$, $c_2 = 1 + \sqrt{2}$, and simplifying the above we obtain

$$\begin{aligned}
\mathbf{E}\left[|\tilde{c}_a - c_a|\right] &\geq \frac{\sqrt{2}}{2(1+\sqrt{2})} \times \left(\left|x_a - \left(\frac{(1-\alpha)n}{(1+\sqrt{2}/2)m}\right)\right| + |x_a|\right) \\
&\geq \frac{\sqrt{2}}{2(1+\sqrt{2})} \times \left(\frac{(1-\alpha)n}{(1+\sqrt{2}/2)m}\right) \\
&\geq 0.17157(1-\alpha)n/m.
\end{aligned}$$

Theorem 8. *For any randomized algorithm, there exists a stream in which some element a occurs $c_a \geq \alpha n$ times, but the algorithm has a counter value $\tilde{c}_a$ such that*

$$\mathbf{E}\Big[\,|c_a - \tilde{c}_a|\,\Big] \geq 0.17157(1-\alpha)n/m.$$

We observe that the proof of Theorem 8 extends to a slightly more powerful model in which the frequency estimation algorithm is allowed to periodically output class/value pairs of the form $(a, \tilde{c}_a)$ whose meaning is "a has occurred $\tilde{c}_a$ times" and the counter value for a is considered to be the last such value output. A similar model is used in Demaine et al. [2002] to study probabilistic packet streams. To see that the lower bound carries over, observe that the last such pair $(a, \tilde{c}_a)$ is either output before the second part of the stream begins, or after. In the latter case, the argument above shows that

$$\mathbf{E}\left[c_a - \tilde{c}_a\right] = \Omega\Big((1-\alpha)n/m\Big).$$

In the former case, the algorithm outputs the value $\tilde{c}_a$ without having seen the final αn occurrences of a. An argument similar to the one above shows that, in this case, there is a element class a such that

$$\mathbf{E}\left[c_a - \tilde{c}_a\right] = \Omega\left(\alpha n\right).$$

3.7 Conclusions

We have studied the problem of estimating the frequency of elements in a data stream using a small amount of memory. Frequency estimation in data stream models some of the fundamental problems in diverse research areas such as computer networks, data mining, and search engines, etc. Previous research has largely focused on variations of brute force methods which are costly for applications that often deal with large data sets; or approaches based on sampling and probabilistic inference and therefore do not have deterministic guarantee on the accuracy required by some applications. We propose a solution that uses a fixed number of counters to approximate the frequencies of elements in a data stream with guaranteed accuracy. We have proved that when some data element a occurs αn times in a stream of length n, then the FREQUENT algorithm (Demaine et al. [2002], Karp et al. [2003]) is $(1-\alpha)n/m$-accurate using m counters. We have also shown that this is nearly optimal for a deterministic algorithm as no deterministic algorithm is better than $(1-\alpha)n/(m+1)$-accurate. Finally, we have proved that randomized algorithms can not be significantly more accurate since any randomized algorithm has an expected accuracy of at least $(1-\alpha)\Omega(n/m)$.

Chapter 4

Approximate Range Queries

In this chapter[1], we consider data structures and algorithms for preprocessing a labelled list of length n so that, for any given indices i and j we can answer queries of the form: What is the mode or median label in the sequence of labels between indices i and j. One of the data stream applications of this problem is that of the queries within a sliding window. That is, the data set of interest is a subset of the data stream which may be of infinite length, and the goal is to construct a data structure so that queries within this subset can be answered quickly. Because of the overlapping nature of sliding windows, it is conceivable that an efficient solution may need to reuse previously constructed data structures and therefore avoid rebuilding the complete data structure from scratch for each sliding window. We give several results on approximate versions of this problem. Our first result is a data structure that uses $O\left(n/(1-\alpha)\right)$ space and can find in $O\left(\log\log_{\frac{1}{\alpha}} n\right)$ time [2] an element whose number of occurrences is at least α times that of the mode, for some user-specified parameter $0 < \alpha < 1$. Our second result shows that constant query time can be achieved for $\alpha = 1/2, 1/3$ and $1/4$, using storage space of $O(n \log n), O(n \log\log n)$ and $O(n)$, respectively. Finally, if the elements are comparable, approximate range median queries can be answered in constant time using a data structure of size $O\left(n/(1-\alpha)\right)$.

4.1 Motivation

Sociologists might be interested in studying which is the most popular name for new born babies in different time periods and how it changes as time passes. A marketing strategist may want to know what is the best selling product of different time periods and/or in different geographic areas. A tourist just arrived in a new city wants

[1] A preliminary version of this chapter appeared in STACS 2005 (Bose et al. [2005]).

[2] All logarithms are to base 2 unless otherwise specified.

to know what is the median price of the hotels that are within 5 blocks of the train station. A real estate developer looking for possible location for a new shopping center may find it useful information to know the median household income within certain driving distance along the highway or around the city. Database administrators are often motivated to provide quick response to users' queries. An obvious solution is to use servers that are powerful enough to handle queries *on demand* even during peak hours. That is, whenever a range query (such as what is the most frequent element or median element in that range) comes in, the data in that range are extracted and processed to find the answer to the query. The disadvantage of this straightforward solution becomes more apparent when there are too many range queries of large size (thereby increasing the processing time) and a real-time response is required. Two of the commonly used techniques for improving database performance are caching and preprocessing. But both suffer from the unpredictability of users' queries. In order to achieve meaningful speedup, it is often necessary to store answers to all possible range queries, resulting in a data structure of size quadratic in the size of the data. On the other hand, exact answers to range queries are not always necessary in the above mentioned applications as long as the errors are within a specified tolerance. This motivates us to study trade offs between space, time and accuracy in range queries.

4.2 Related Work

Let $A = a_1, \ldots, a_n$ be a list of elements of some data type. We wish to construct data structures on A, such that we can quickly answer *range queries.* These queries take two indices i, j with $1 \leq i \leq j \leq n$ and require computing $F(a_i, \ldots, a_j) = a_i \circ a_{i+1} \circ \cdots \circ a_{j-1} \circ a_j$. If the inverse of the operation "$\circ$" exists, then range queries have a trivial solution of linear space and constant query time. For example, if "$\circ$" is arithmetic addition (subtraction being its inverse), we precompute all the partial sums $b_i = a_1 + \cdots + a_i, i = 1, \ldots, n$, and the range query $F(a_i, \ldots, a_j) = a_i + \cdots + a_j$ can be answered in constant time by computing $b_j - b_{i-1}$. Yao (Yao [1982], see also Alon and Schieber [1987]) showed that if "$\circ$" is a constant time semigroup operation[3] (such as maximum or minimum) for which no inverse operation exists, and $a \circ b$ can be

[3]A semigroup operation is an associative binary operation.

computed in constant time then it is possible to answer range queries in $O(\lambda(k,n))$ time using a data structure of size $O(kn)$, for any integer $k \geq 1$. Here $\lambda(k,\cdot)$ is a slowly growing function at the $\lfloor k/2 \rfloor$-nd level of the primitive recursive hierarchy (Alon and Schieber [1987]). For example, $\lambda(2,n) = O(\log n)$, $\lambda(3,n) = O(\log\log n)$ and $\lambda(4,n) = O(\log^* n)$.

Krizanc et al. [2003] studied the storage space versus query time tradeoffs for range mode and range median queries. These occur when F is the function that returns the mode or median of its input. Mode and median are two of the most important statistics (Manku et al. [1998], Battiato et al. [2000], Arasu and Manku [2004], Lin et al. [2004]). Given a set of n elements, a *mode* is an element that occurs at least as frequently as any other element of the set. If the elements are comparable (for example, real numbers), the *rank* of an element is its position in the sorted order of the input. For example, the rank of the minimum element is 1, and that of the maximum element is n. The ϕ-quantile is the element with rank $\lfloor \phi n \rfloor$. The 1/2-quantile is also called the *median.* Note the trivial solution of precomputing all the partial sums does not work for range mode or range median queries as no inverse exists for either operation. Yao's approach does not apply either because neither range mode nor range median is associative and therefore not a semigroup operation. Also, given two sets S_1 and S_2 and their modes (or medians), the mode (or median) of the union $S_1 \cup S_2$ cannot be computed in constant time. New data structures are needed for range mode and range median queries. Krizanc et al. [2003] gave a data structure of size $O(n^{2-2\epsilon})$ that can answer range mode queries in $O(n^\epsilon \log n)$ time, where $0 < \epsilon \leq 1/2$ is a constant representing the storage space query time tradeoff. For range median queries, they show that a data structure of size $O(n)$ can answer range median queries in $O(n^\epsilon)$ time and a faster $O(\log n)$ query time can be achieved using $O\left(\frac{n \log^2 n}{\log\log n}\right)$ space.

Here we consider the approximate versions of range mode and range median queries. We show that if a small amount of error is tolerable, range mode and range median queries can be answered much more efficiently in terms of storage space and query time. Given a sublist $S = a_i, a_{i+1}, \ldots, a_j$, an element is said to be an *approximate range mode* of S if its number of occurrences is at least α times that of the

actual mode of S, where $0 < \alpha < 1$ is a user-specified approximation factor. If the elements are comparable, the *median* is the element with rank $\lfloor (j-i+1)/2 \rfloor$ (relative to the sublist). An *α-approximate median* of S is an element whose rank is between $\alpha \times \lfloor (j-i+1)/2 \rfloor$ and $(2-\alpha) \times \lfloor (j-i+1)/2 \rfloor$, meaning the difference between the rank of the approximate range median and that of the actual range median is at most $(1-\alpha) \times \lfloor (j-i+1)/2 \rfloor$. Clearly, there could be several approximate range modes and medians.

We show that approximate range mode queries can be answered in $O(\log\log_{\frac{1}{\alpha}} n)$ time using a data structure of size $O(n/(1-\alpha))$. We also show that constant query time can be achieved for $\alpha = 1/2, 1/3$ and $1/4$ using storage space of size $O(n \log n)$, $O(n \log\log n)$ and $O(n)$, respectively. For approximate range median queries, we introduce a data structure that uses $O(n/(1-\alpha))$ space and can answer queries in constant time. We also study the preprocessing time required for the construction of these data structures.

It is well known that, given a list of n elements, its mode can be found in $O(n \log n)$ time by sorting the list, and its median can be found in linear time (Blum et al. [1973]). To the best of our knowledge, there is no previous work on approximate range mode or range median queries. Two problems related to range mode and range median queries are *frequent elements* and *quantile summaries* over sliding windows (Arasu and Manku [2004], Lin et al. [2004]). In the sliding window model, data takes the form of continuous data streams and queries are answered regarding only the most recently observed n data elements (Datar et al. [2002]). Lin et al. [2004] studied the problem of continuously maintaining quantile summaries over sliding windows. They devised an algorithm for computing approximate quantiles with an error of at most ϵn using $O\left(\frac{\log \epsilon^2 n}{\epsilon} + \frac{1}{\epsilon^2}\right)$ storage space in the worst case for a fixed window size n. For windows of variable size at most n (such as timestamp-based windows in which the exact number of arriving elements within a fixed time interval cannot be entirely predetermined), $O\left(\frac{\log^2 \epsilon n}{\epsilon^2}\right)$ storage space is used by their algorithm. Arasu and Manku [2004] improved both bounds to $O\left(\frac{1}{\epsilon} \log \frac{1}{\epsilon} \log n\right)$ and $O\left(\frac{1}{\epsilon} \log \frac{1}{\epsilon} \log \epsilon n \log n\right)$, respectively. They also proposed deterministic algorithms for the problem of finding all frequent elements (*i.e.*, elements with a minimum frequency of ϵn) using $O\left(\frac{1}{\epsilon} \log^2 \frac{1}{\epsilon}\right)$

and $O\left(\frac{1}{\epsilon}\log^2\frac{1}{\epsilon}\log\epsilon n\right)$ space for fixed and variable size windows, respectively.

4.3 Approximate Range Mode Queries

Given a list of elements $a_1, \ldots, a_n$ and an arbitrary element x in the list, $f_x(a_1, \ldots, a_n)$ is the frequency[4] of x in the list, and

$$F(a_1, \ldots, a_n) = \max_x f_x(a_1, \ldots, a_n)$$

is the number of occurrences of the most frequently occurring element in the list (*i.e.*, the mode[5]).

Similarly, given a list of elements $a_1, \ldots, a_n$ and an approximation factor $0 < \alpha < 1$, the *approximate range mode queries* problem can be specified formally as follows.
Input: Two indices i, j with $1 \leq i \leq j \leq n$.
Output: An element x in $a_i, \ldots, a_j$ such that $f_x(a_i, \ldots, a_j) \geq \alpha \times F(a_i, \ldots, a_j)$.

The idea behind our data structure is based on the observation that given a fixed left endpoint i of a query range, as the right endpoint j of the range moves from left to right, the approximate range mode changes much less frequently. In fact, the number of times the approximate range mode changes as j varies from i to n is at most $\log_{\frac{1}{\alpha}}(n-i)$. This is because the same element can be output as approximate range mode as long as no other element's frequency exceeds $1/\alpha$ times that of the current approximate range mode. When the actual mode's frequency has exceeded $1/\alpha$ times that of the approximate range mode, the approximate range mode is replaced and the actual mode becomes the new approximate range mode.

For example, given the list of 20 elements shown in Figure 4.1 and approximation factor $\alpha = 1/2$, b is an approximate range mode of $a_1, \ldots, a_9$ because b occurs 2 times in the range, while the actual mode a occurs 4 times in the same range. But this is no longer true for query $a_1, \ldots, a_{10}$, as the number of occurrences of b is still 2 while the actual mode a now occurs 5 times in the range ($f_b(a_1, \ldots, a_{10}) = 2 < \alpha \times f_a(a_1, \ldots, a_{10}) = 2.5$). In this case, either a or c ($f_c(a_1, \ldots, a_{10}) = 3$) becomes the new approximate range mode.

[4]We use the term frequency and number of occurrences interchangeably.
[5]There could be several modes, all have the same frequency.

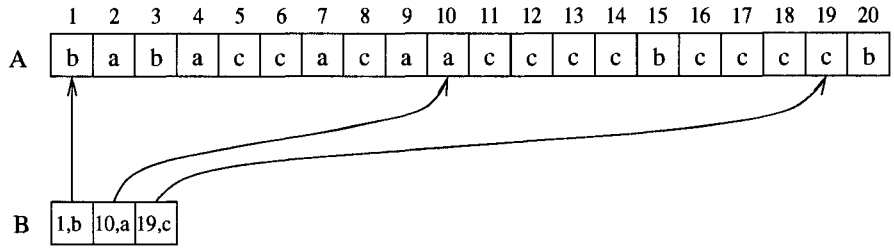

Figure 4.1: Given approximation factor $\alpha = 1/2$, a lookup table of size 3 is used for answering queries $a_1, \ldots, a_j$, $j = 1, \ldots, 20$.

Assume a is chosen to be the new approximate range mode, it remains a valid approximate range mode as the right endpoint of the query range proceeds until $j = 19$ at which point the actual mode c occurs 11 times ($f_c(a_1, \ldots, a_{19}) = 11$). Since no other element (neither a nor b) occurs more than or equal to half of the actual mode ($f_a(a_1, \ldots, a_{19}) = 5, f_b(a_1, \ldots, a_{19}) = 3$), c is now the only approximate range mode and remains so up to the end of the list. Since an approximate mode remains valid until another element occurs more than $1/\alpha$ times the current approximate range mode does, the number of approximate range modes that have to be stored is much less than the number of elements of the original list. As shown in the example of Figure 4.1, instead of storing the complete original array of 20 elements, a table of 3 approximate range modes suffices to answer all approximate range mode queries $a_1, \ldots, a_j, 1 \leq j \leq 20$.

Given a list of n elements $A = a_1, \ldots, a_n$ and an approximation factor α, all approximate range mode queries with a_1 being the left endpoint: $a_1, \ldots, a_j$ ($1 \leq j \leq n$) can be answered using $O(\log_{\frac{1}{\alpha}} n)$ storage space. The data structure is a lookup table

$$B = \{a_{c_1}, \ldots, a_{c_m}\}, 1 \leq c_1 < c_2 < \ldots < c_m \leq n.$$

in which we store m approximate range modes. The first entry is always a_1 ($c_1 = 1$). The second entry a_{c_2} is the first element in A that occurs $\lceil 1/\alpha \rceil$ times, *i.e.*,

$$f_{a_{c_2}}(a_1, \ldots, a_{c_2}) = \lceil 1/\alpha \rceil$$

and

$$f_{a_i}(a_1, \ldots, a_{c_2}) < \lceil 1/\alpha \rceil$$

for $\forall i \neq c_2$. In general, the kth entry in the table is the first element in A that occurs $\lceil 1/\alpha^{k-1} \rceil$ times as the right endpoint of the query range moves towards right. Note that a_{c_k} is an approximate range mode of $a_1, \ldots, a_j$ for any $c_k \leq j < c_{k+1}$ since a_{c_k} occurs at least $\lceil 1/\alpha^{k-1} \rceil$ times in $a_1, \ldots, a_j$

$$f_{a_{c_k}}(a_1, \ldots, a_j) \geq f_{a_{c_k}}(a_1, \ldots, a_{c_k}) = \lceil 1/\alpha^{k-1} \rceil$$

while no other element occurs more than $1/\alpha^k$ times in the same range

$$f_x(a_1, \ldots, a_j) < f_{c_{k+1}}(a_1, \ldots, a_{c_{k+1}}) = \lceil 1/\alpha^k \rceil.$$

The last approximate range mode in the table, a_{c_m}, occurs at least $\lceil 1/\alpha^{m-1} \rceil$ times in $a_1, \ldots, a_n$. It follows immediately that $m < \lfloor \log_{\frac{1}{\alpha}} n \rfloor + 1$ and the number of approximate range modes stored in the lookup table is at most $\lfloor \log_{\frac{1}{\alpha}} n \rfloor + 1$.

To answer approximate range mode query $a_1, \ldots, a_j$, binary search is used to find in $O(\log \log_{\frac{1}{\alpha}} n)$ time the largest c_k that is less than or equal to j and output a_{c_k} as the answer.

Lemma 3. *There is a data structure of size $O(\log_{\frac{1}{\alpha}} n)$ that can answer approximate range mode queries $a_1, \ldots, a_j$ $(1 \leq j \leq n)$ in $O(\log \log_{\frac{1}{\alpha}} n)$ time.*

An immediate application of Lemma 3 is a data structure for answering approximate range mode queries with arbitrary endpoints. The data structure is a collection of n lookup tables $(T_i, i = 1, \ldots, n)$, one table for each left endpoint. An auxiliary array of n pointers is used to locate a table in $O(1)$ time. A query $a_i, \ldots, a_j$ can be answered by first locating table T_i in $O(1)$ time, and then searching in T_i to find the approximate mode of $a_i, \ldots, a_j$, which takes $O(\log \log_{\frac{1}{\alpha}} n)$ time since T_i contains at most

$$O(\log_{\frac{1}{\alpha}}(n - i)) = O(\log_{\frac{1}{\alpha}} n)$$

approximate range modes.

Corollary 1. *There is a data structure of size $O(n \log_{\frac{1}{\alpha}} n)$ that can answer approximate range queries in $O(\log \log_{\frac{1}{\alpha}} n)$ time.*

4.3.1 An Improvement Based on Persistent Search Trees

We have seen that by maintaining a lookup table T_i of size $O(\log_{\frac{1}{\alpha}} n)$ for each left endpoint i $(1 \leq i \leq n)$ and using $O(n \log_{\frac{1}{\alpha}} n)$ total storage space for n lookup tables, any approximate range mode query $a_i, \ldots, a_j$ can be answered in $O(\log \log_{\frac{1}{\alpha}} n)$ time. Given a fixed left endpoint i, storing an answer for each right endpoint j is not necessary since the answer to the query changes less frequently as j varies. The approximate range modes of two query ranges with adjacent endpoints are unlikely to be very different. In this section, we pursue this idea and show that storage of a complete lookup table for each left endpoint is unnecessary because of the similarity between two tables with adjacent left endpoints.

To see how the approximate range mode changes gradually as the two endpoints of a query range move, we need a systematic way to keep track of the range within which the current approximate range mode remains a valid approximation of the actual mode and its number of occurrences in that range. As the query range changes, the frequency of the current approximate range mode may also change. Once it drops below a predetermined threshold value (f_{low}, the calculation of which will be discussed next), a new approximate range mode is chosen and the effective query range within which the new approximate range mode remains valid is also updated.

As shown in Table 4.1, each entry in the lookup table is a 5-tuple

$$(f_{low_r}, f_{high_r}, q_r, ans_r, f_{ans_r}).$$

The first entry in the first row is $[f_{low_1} = 1,\ f_{high_1} = 1]$, the complete table is constructed left to right and top down according to the following:

$$\begin{cases} f_{low_{r+1}} = f_{high_r} + 1 \\ f_{high_{r+1}} = \lceil f_{low_r}/\alpha \rceil + 1 \\ F(a_i, \ldots, a_{q_r}) = f_{high_r} \\ f_{ans_r} = f_{ans_r}(a_i, \ldots, a_{q_r}) \\ f_{low_r} \leq f_{ans_r} \leq f_{high_r} \end{cases}$$

To answer range query $a_i, \ldots, a_j$, we search in the ith table for r such that $q_r \leq$

$j < q_{r+1}$, and ans_r along with its frequency $f_{ans_r} = f_{ans_r}(a_i, \ldots, a_{q_r})$ is output as answer.

Frequency Range	Effective Query Range	Answer
...		
$[f_{low_r}, f_{high_r}]$	q_r	(ans_r, f_{ans_r})
$[f_{low_{r+1}}, f_{high_{r+1}}]$	q_{r+1}	$(ans_{r+1}, f_{ans_{r+1}})$
...		

Table 4.1: The lookup table used for answering approximate range mode queries.

As noted before, the ith table T_i corresponds to all the range queries with the same left endpoint i. Given an approximation factor α, $[f_{low_r}, f_{high_r}]$ are precomputed for $r = 1, \ldots, 2\lceil \log_{\frac{1}{\alpha}} n \rceil$ and remain the same for all tables. A counter is used for each element to keep track of its frequency as the right endpoint j varies. Given the fixed left endpoint i, as the right endpoint j proceeds, ans_r is the first element whose frequency in $a_i, \ldots, a_j$ reaches f_{high_r}, and q_{r+1} is the rightmost point up to which ans_r remains a valid approximate mode, *i.e.,* no other element's frequency is higher than f_{low_r}/α. Given a query $a_i, \ldots, a_j$ with $q_r \leq j < q_{r+1}$, ans_r is a valid approximate range mode since its frequency is at least f_{high_r} while no other element's frequency is higher than or equal to $f_{high_{r+1}} - 1 = \lceil f_{low_r}/\alpha \rceil$. To see how the subsequent tables are built based on T_i with minimum number of changes, the right endpoint of the query range is fixed, as the left endpoint of the query range proceeds, ans_r's frequency may decrease, but it remains a valid approximate range mode as long as $f_{ans_r} \geq f_{low_r}$ and it is copied to the next table along with a possibly smaller f_{ans_r} (Note that f_{ans_r} is needed only for bookkeeping purposes). The only time that ans_r must change for a table is when its frequency drops below f_{low_r}. At this point we update ans_r and the new approximate range mode is the first element whose frequency reaches f_{high_r} with respect to the current left endpoint of query range. The right endpoint of the effective query range q_r is also updated to reflect the change on the approximate range mode $(f_{ans_r}(a_i, \ldots, a_{q_r}) = f_{high_r})$.

Table 4.2 shows the data structure for answering approximate range mode queries on the same list as in Figure 4.1. For example, to look up the approximate range mode of $a_4, \ldots, a_{12}$, we search in T_4 and find the entry with the largest q_r that is

smaller than 12: $\{[4,5], 10, (a,4)\}$. This tells us that, in the sequence of $a_4, \ldots, a_{12}$, a occurs at least 4 times ($f_a(a_4, \ldots, a_{12}) \geq f_a(a_4, \ldots, a_{10}) = 4$) and no element occurs more than 8 times ($f_x(a_4, \ldots, a_{12}) \leq F(a_4, \ldots, a_{17}) - 1 = 8$).

T_i	T_1	T_2	T_3	T_4	T_5
a_i	b	a	b	a	c
$[1,1]$	$1, (\mathbf{b}, 1)$	$2, (\mathbf{a}, 1)$	$3, (\mathbf{b}, 1)$	$4, (\mathbf{a}, 1)$	$5, (\mathbf{c}, 1)$
$[2,3]$	$7, (\mathbf{a}, 3)$	$7, (a, 3)$	$7, (a, 2)$	$7, (a, 2)$	$8, (\mathbf{c}, 3)$
$[4,5]$	$10, (\mathbf{a}, 5)$	$10, (a, 5)$	$10, (a, 4)$	$10, (a, 4)$	$12, (\mathbf{c}, 5)$
$[6,9]$	$17, (\mathbf{c}, 9)$	$17, (c, 9)$	$17, (c, 9)$	$17, (c, 9)$	$17, (c, 9)$
$[10,13]$	$20, (\mathbf{c}, 11)$	$20, (c, 11)$	$20, (c, 11)$	$20, (c, 11)$	$20, (c, 11)$

T_i	T_6	T_7	T_8	T_9	T_{10}
a_i	c	a	c	a	a
$[1,1]$	$6, (\mathbf{c}, 1)$	$7, (\mathbf{a}, 1)$	$8, (\mathbf{c}, 1)$	$9, (\mathbf{a}, 1)$	$10, (\mathbf{a}, 1)$
$[2,3]$	$8, (c, 2)$	$10, (\mathbf{a}, 3)$	$10, (a, 2)$	$10, (a, 2)$	$13, (\mathbf{c}, 3)$
$[4,5]$	$12, (c, 4)$	$14, (\mathbf{c}, 5)$	$14, (c, 5)$	$14, (c, 4)$	$14, (c, 4)$
$[6,9]$	$17, (c, 8)$	$17, (c, 7)$	$17, (c, 7)$	$17, (c, 6)$	$17, (c, 6)$
$[10,13]$	$20, (c, 10)$	—	—	—	—

T_i	T_{11}	T_{12}	T_{13}	T_{14}	T_{15}
a_i	c	c	c	c	b
$[1,1]$	$11, (\mathbf{c}, 1)$	$12, (\mathbf{c}, 1)$	$13, (\mathbf{c}, 1)$	$14, (\mathbf{c}, 1)$	$15, (\mathbf{b}, 1)$
$[2,3]$	$13, (\mathbf{c}, 3)$	$13, (c, 2)$	$16, (\mathbf{c}, 3)$	$16, (c, 2)$	$18, (\mathbf{c}, 3)$
$[4,5]$	$14, (c, 4)$	$17, (\mathbf{c}, 5)$	$17, (c, 4)$	$19, (\mathbf{c}, 5)$	$19, (c, 4)$
$[6,9]$	$17, (c, 6)$	$19, (\mathbf{c}, 7)$	$19, (c, 6)$	—	—
$[10,13]$	—	—	—	—	—

T_i	T_{16}	T_{17}	T_{18}	T_{19}	T_{20}
a_i	c	c	c	c	b
$[1,1]$	$16, (\mathbf{c}, 1)$	$17, (\mathbf{c}, 1)$	$18, (\mathbf{c}, 1)$	$19, (\mathbf{c}, 1)$	$20, (\mathbf{b}, 1)$
$[2,3]$	$18, (c, 3)$	$18, (c, 2)$	$19, (\mathbf{c}, 2)$	—	—
$[4,5]$	$19, (c, 4)$	—	—	—	—
$[6,9]$	—	—	—	—	—
$[10,13]$	—	—	—	—	—

Table 4.2: An example showing the data structure for answering 1/2-approximate range mode queries on a list of 20 elements. Updates are in bold.

After T_1 is built, T_i ($i \geq 2$) is built based on T_{i-1} with necessary updates. The number of updates made is given by the following lemma.

Lemma 4. *If the rth row of the table is updated in T_i, then it does not need to be updated in T_k for any $i < k < i + 1/\alpha^{\lfloor r/2 \rfloor}$.*

Proof. When the rth row is updated in T_i, we set ans_r to be the first element such that $f_{ans_r}(a_i, \ldots, a_{q_r}) = f_{high_r}$. Its frequency f_{ans_r} is initially f_{high_r} in T_i. Although f_{ans_r} may decrease as i increases, ans_r does not need to be updated again until f_{ans_r} drops below f_{low_r}, which takes at least $f_{high_r} - (f_{low_r} - 1) = f_{high_r} - f_{high_{r-1}} = 1/\alpha^{\lfloor r/2 \rfloor}$ steps. □

Note that there are no more than $2\lceil \log_{\frac{1}{\alpha}} n \rceil$ rows in a table and every time we build a new table, the first row needs to be updated. Lemma 4 shows that the rth ($r \geq 2$) row changes no more than $\alpha^{\lfloor r/2 \rfloor} n$ times during the construction of all n tables. The total number of updates we have to make is given by the following theorem.

Theorem 9. *The total number of updates we have to make is $O(n/(1-\alpha))$.*

Proof. Total number of updates $\leq n + \sum_{r=2}^{2\lceil \log_{\frac{1}{\alpha}} n \rceil} \alpha^{\lfloor r/2 \rfloor} n = O(\frac{n}{1-\alpha})$. □

Theorem 9 says that, the majority of the table entries can be reconstructed by referring to other tables. In other words, although n lookup tables are used to answer approximate range mode queries, many of them share common entries. A persistent search tree (Driscoll et al. [1989]) is used to store the tables efficiently. It has the property that the query time is $O(\log m)$ where m is the number of entries in each table, and the storage space is $O(1)$ per update. In the case of approximate range mode queries, although there are n lookup tables and each table can have as many as $2\lceil \log_{\frac{1}{\alpha}} n \rceil$ entries, many tables share the same entries and the number of different nodes in the persistent tree is $O(n/(1-\alpha))$, one for each update, and the query time for a node is $O(\log \log_{\frac{1}{\alpha}} n)$.

To build the search tree, we need to keep track of the frequency of each element as the query range varies. First we assign a list of counters, one for each distinct element in $O(n \log n)$ time by sorting. The idea presented in Demaine et al. [2002] leads to a data structure that supports retrieval of the highest counter value in constant time. More specifically, counters are stored in sorted order according to their value. To support fast increment or decrement of counters, which may require changes in the

total order, equal counters are coalesced into groups. The tables are built one row at a time. Suppose we are to fill the rth entries of all n tables. As the left endpoint i and the right endpoint j proceed, one of (a) and (b) will be executed with each increment in i or j.

(a) $f_{ans_r} = f_{ans_r}(a_i, \ldots, a_{q_r}) \geq f_{low_r}$. Increment i, copy the rth entry of the $(i-1)$th table into the rth entry of the ith table, update f_{ans_r} if necessary.

(b) $f_{ans_r} = f_{ans_r}(a_i, \ldots, a_{q_r}) < f_{low_r}$. The current approximate range mode's frequency has dropped below the threshold and a new approximate range mode is needed. Increment q_r and update (ans_r, f_{ans_r}) if necessary so that ans_r is the range mode of $a_i, \ldots, a_{q_r}$, repeat this until an element is found whose frequency is equal to f_{high_r}.

Because both (a) and (b) can be executed in constant time and combined they will be executed at most $2n$ times as both i and j go from 1 to n to fill each of the $O(n \log_{\frac{1}{\alpha}} n)$ rows, we obtain the following results.

Theorem 10. *There exists a data structure of size $O(n/(1-\alpha))$ that can answer approximate range mode queries in $O\left(\log\log_{\frac{1}{\alpha}} n\right)$ time, and can be constructed in $O\left((1+\frac{1}{\log(1/\alpha)})n \log n\right)$ time.*

4.3.2 Lower Bounds

Next we show that there is no faster worst case algorithm for computing the approximate range mode for any fixed approximation factor α. To see this, let A be a list of $n/\lceil 1/\alpha \rceil$ elements and $B = A \ldots A = b_1, \ldots, b_n$ a list of length n obtained by repeating A $\lceil 1/\alpha \rceil$ times. The problem of testing whether there exist two identical elements in A (also called *element uniqueness*) can be reduced to asking if the mode of B occurs more than $\lceil 1/\alpha \rceil$ times. In the case of approximate range mode query, the answer to query $b_1, \ldots, b_n$ is an element whose frequency is greater than 1 if and only if the actual mode of B occurs more than $\lceil 1/\alpha \rceil$ times.

In the algebraic decision tree model of computation, the running time of determining whether all the elements of A are unique is known to have a complexity of

$\Omega(n \log n)$ (Ben-Or [1983]). However, this problem can also be solved by doing a single approximate range mode query $b_1, \ldots, b_n$ after preprocessing B, which implies the same lower bound holds for approximate range mode queries.

Theorem 11. *Let $P(n)$ and $Q(n)$ be the preprocessing and query times, respectively, of a data structure for answering approximate range mode queries, we have $P(n) + Q(n) = \Omega(n \log n)$.*

On the other hand, $\Omega(n)$ storage space is required by any data structure that supports approximate range mode queries since the original list can be reconstructed by doing queries $(a_1, a_1), (a_2, a_2), \ldots, (a_n, a_n)$, regardless of what value α is.

4.3.3 Constant Query Time

Yao (Yao [1982], see also Alon and Schieber [1987]) showed that if a query $a_i, \ldots, a_j$ can be answered by combining answers of queries $a_i, \ldots, a_x$ and $a_{x+1}, \ldots, a_j$ in constant time, then $\Theta(n\lambda(k, n))$ preprocessing time and storage space is both necessary and sufficient to answer range queries in at most k steps[6]. We adapt the same approach to develop constant query time data structures for some special cases of approximate range mode queries. Namely, the approximation factor $\alpha = 1/k$ where k is some positive integer.

The following lemma says that, if we can partition the range $a_i, \ldots, a_j$ into k intervals and we know the mode of each interval, then one of these is an approximate range mode, for $\alpha = 1/k$.

Lemma 5. *If $\{B_1, \ldots, B_k\}$ is a partition of $a_i, \ldots, a_j$ then*

$$max_p F(B_p) \geq F(a_i, \ldots, a_j)/k.$$

Proof. By contradiction. Otherwise for any element x we have

$$\begin{aligned} f_x(a_i, \ldots, a_j) &= \sum_{p=1}^{k} f_x(B_p) \\ &\leq k \times max_p F(B_p) \\ &< F(a_i, \ldots, a_j). \end{aligned}$$

[6]See Chapter 4.2 for the definition of $\lambda(k, n)$.

□

Yao [1982] and Alon and Schieber [1987] gave an optimal scheme of using a minimum set of intervals such that any range $a_i, \ldots, a_j$ can be covered by at most k such intervals.

Lemma 6. *(Yao [1982], Alon and Schieber [1987]) There exists a set of $O(n\lambda(k,n))$ intervals such that any query range $a_i, \ldots, a_j$ can be partitioned into at most k of these intervals. Furthermore, given i and j, these intervals can be found in $O(k)$ time.*

Given Lemma 5 and Lemma 6, we immediately obtain a constant query time solution to approximate range mode queries with approximation factor $\alpha = 1/k$. By precomputing the mode of each interval, a query can be answered by first fetching the partition of the query range, which is a set of at most k intervals, and then outputting the one with the highest frequency among k modes of these intervals.

Theorem 12. *There exists a data structure of size $O(n\lambda(k,n))$ that can answer approximate range mode queries in $O(k)$ time, for $\alpha = 1/k$.*

The results in Theorem 12 can be further improved using a table lookup trick for $k \geq 4$. We partition the list into $n/\log n$ blocks of size $\log n$:

$$B_i = a_{(i-1)\log n+1}, \ldots, a_{i \log n}, i = 1, \ldots, n/\log n.$$

By Lemma 6, there exists a set of $O((n/\log n)\lambda(2, n/\log n)) = O(n)$ intervals such that any range with both endpoints at the boundaries of the blocks can be covered with at most 2 of these intervals. The exact modes of these intervals are precomputed. Inside every block, exact modes of 2 intervals are precomputed for each element, one interval is between the element and the beginning of the block and the other interval between the element and the endpoint of the block. Any query range that spans more than one block can be partitioned into at most 4 intervals. The first one is the (possibly partial) block in which the range starts; the last one is the (possibly partial) block in which the range ends and the other (at most) two intervals in between cover all the remaining blocks (if any). Of these intervals the modes are all precomputed, and the one with the highest frequency is a 1/4-approximation of the actual mode.

It remains to show that a query within a block can also be answered in $O(1)$ time. This is done by recursively partitioning the $\log n$ block into $\log n/\log\log n$ blocks of size $\log\log n$. The same method above is used to preprocess these blocks, and the result is a data structure of $O(n)$ size that can answer any query that spans more than one $\log\log n$-block in $O(1)$ time.

To answer queries within a $\log\log n$-block, a standard data structure trick (Gabow and Tarjan [1985]) of canonical subproblems is used. Note that we can normalize each block by replacing each element with the index of its first occurrence within the block. Because such index is a non-negative integer that is at most $\log\log n$ and each block consists of $\log\log n$ such values, there are at most $(\log\log n)^{\log\log n}$ different blocks. Among all $n/\log\log n$ blocks of size $\log\log n$, many are of the same type. Thus, preprocessing of each block is unnecessary, and storage space can be reduced by preprocessing a block once and reusing the results for all blocks of the same type. The data structure used is a $\log\log n \times \log\log n$ matrix that can answer range mode query in constant time. All the queries in blocks of the same type are done in the same matrix. There are at most $(\log\log n)^{\log\log n}$ possible matrices which require $O\left((\log\log n)^{\log\log n}(\log\log n)^2\right) = o(n)$ storage space.

Theorem 13. *There exists a data structure of size $O(n)$ that can answer approximate range mode queries in $O(1)$ time, for $\alpha = 1/4$.*

4.4 Approximate Range Median Queries

In this section, we consider approximate range median queries on a list of comparable elements $A = a_1, \ldots, a_n$. Given an approximation factor $0 < \alpha < 1$, our task is to preprocess A so that, given indices $1 \le i \le j \le n$, we can quickly return an element of $a_i, \ldots, a_j$ whose rank is between $\alpha \times \lfloor (j-i+1)/2 \rfloor$ and $(2-\alpha) \times \lfloor (j-i+1)/2 \rfloor$.

To simplify the presentation we assume $n = 2^d$ for some integer $d \ge 1$. Generalization to arbitrary n is straightforward. As an example shown in Figure 4.2, for a list of 16 elements, $d = 4$ levels of partitions are used. In level i ($i = 1, 2, 3, 4$), the list is partitioned into 2^i non-overlapping blocks of size $n/2^i$.

For each block up to $2\lceil 2\alpha/(1-\alpha) \rceil = 4$ medians are precomputed, each corresponding to a sublist with both endpoints at block boundaries up to $2\lceil 2\alpha/(1-\alpha) \rceil = 4$

blocks away. For example, associated with the $2nd$ block in level 3 are 4 medians, each corresponds to a union of up to 4 consecutive blocks: $1 = Median(B_{3_2})$; $10 = Median(B_{3_2} \bigcup B_{3_3})$; $6 = Median(B_{3_2} \bigcup \ldots \bigcup B_{3_4})$; $7 = Median(B_{3_2} \bigcup \ldots \bigcup B_{3_5})$. The idea behind our algorithm is that, if a query $a_i, \ldots, a_j$ spans many blocks, then the contribution of the first and last block is minimal and can be ignored. Instead, we could return the precomputed exact median of the union of the internal blocks as an approximate range median. On the other hand, since there are blocks of different sizes, we can choose a partition level so that $a_i, \ldots, a_j$ spans just enough blocks for the strategy above to give a valid approximation. This ensures that we do not have to precompute too many inter-block medians.

Note that a 1/2-approximate range median query that spans more than 4 complete blocks also spans at least 2 complete blocks in the next higher level and therefore can be answered in a higher level with sufficient accuracy. Range median queries are answered by looking in the level where the query range spans just enough number of complete blocks the exact range median of which is precomputed. For example, query $a_2, \ldots, a_{11}$ spans 4 complete level 3 blocks $(B_{3_2} \bigcup \ldots \bigcup B_{3_5})$ but only 1 complete level 2 block (B_{2_2}). Therefore, the $4th$ entry in the $2nd$ level 3 block:

$$T_{3_2}(4) = \text{M}edian(B_{3_2} \bigcup \ldots \bigcup B_{3_5}) = 7$$

whose rank in $a_2, \ldots, a_{11}$ is 4, is output as the approximate median, while the rank of the actual median is 5 in the sublist of 10 elements.

At the lowest level, $a_1, \ldots, a_n$ is partitioned into n blocks each consisting of a single element. We precompute for each $i = 1, \ldots, n$ all the medians of $a_i, \ldots, a_j$, for $i \leq j \leq i + 2\lceil 2\alpha/(1-\alpha) \rceil - 1$. This enables us to answer queries of length no more than $2\lceil 2\alpha/(1-\alpha) \rceil$ in $O(1)$ time using $O(n/(1-\alpha))$ space. To answer queries of length greater than $\lceil 2\alpha/(1-\alpha) \rceil - 1$, we search in a higher level where the query spans at least $\lceil 2\alpha/(1-\alpha) \rceil$ but no more than $2\lceil 2\alpha/(1-\alpha) \rceil$ full blocks. Suppose the query spans $\lceil 2\alpha/(1-\alpha) \rceil \leq c \leq 2\lceil 2\alpha/(1-\alpha) \rceil$ full blocks in level i, and l is the length of the query range, then we have $cn/2^i \leq l < (c+2)n/2^i$. The median of the union of these c blocks is precomputed and its rank in the query range is at least $cn/2^{i+1} \geq \alpha l/2$ and at most $cn/2^{i+1} + (l - cn/2^i) \leq (2-\alpha)l/2$, in other words, it is an α-approximate median of the query range.

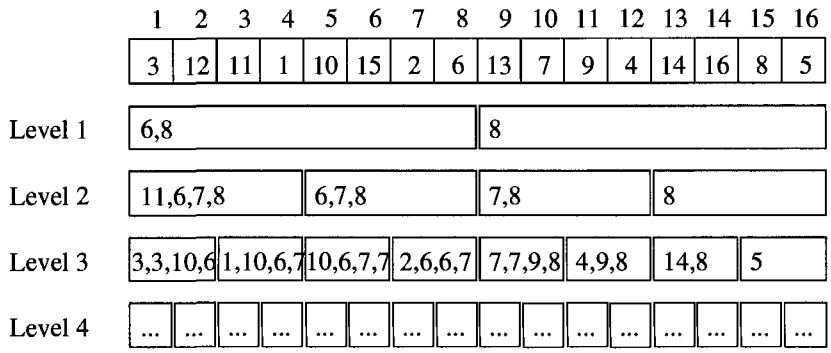

Figure 4.2: Data structure for answering $\frac{1}{2}$-approximate range median queries on a list of 16 elements.

In the next three subsections we give the preprocessing time, storage space and query time of our data structure for answering approximate range median queries.

4.4.1 $O(n \log n/(1-\alpha)^2)$ Preprocessing Time

We preprocess $A = a_1, \ldots, a_n$ and build d lookup tables as follows. To build T_i $(1 \leq i \leq d)$, we partition A into 2^i blocks each of size $n/2^i$: $B_{i_j} = a_{(j-1)\times n/2^i+1}$, $\ldots$, $a_{j\times n/2^i}$, $j = 1, \ldots, 2^i$. Each of T_i's 2^i entries $(T_{i_j}, j = 1, \ldots, 2^i)$ corresponds to a block B_{i_j} and contains a list of $2\lceil 2\alpha/(1-\alpha)\rceil$ inter-block range medians: $T_{i_j}(k) = Median(B_{i_j} \bigcup \ldots \bigcup B_{i_{j+k-1}})$, $k = 1, \ldots, 2\lceil 2\alpha/(1-\alpha)\rceil$. $B_{i_j} \bigcup \ldots \bigcup B_{i_{j+k-1}}$ is of length $kn/2^i$, and its median can be computed in $O(kn/2^i)$ time. There are $\log n$ tables to be computed. It follows that the total preprocessing time is

$$\begin{aligned} & \sum_{i=1}^{\log n} \sum_{j=1}^{2^i} \sum_{k=1}^{2\lceil \frac{2\alpha}{1-\alpha} \rceil} O\left(\frac{kn}{2^i}\right) \\ = \; & O\left(\frac{n \log n}{(1-\alpha)^2}\right). \end{aligned}$$

4.4.2 $O\left(n/(1-\alpha)\right)$ Storage Space

The data structure for answering approximate range median queries is a set of $\log n$ lookup tables. Each table T_i $(1 \leq i \leq \log n)$ has $O(2^i)$ entries and each entry is a list of at most $2\lceil 2\alpha/(1-\alpha)\rceil$ precomputed range medians, hence the total space needed to store all $\log n$ tables is

$$\sum_{i=1}^{\log n} O(2^i \alpha/(1-\alpha)) = O(n\alpha/(1-\alpha)) = O(n/(1-\alpha)).$$

4.4.3 $O(1)$ Query Time

Next we show how to compute an approximate range median of $a_i, \ldots, a_j$.

(a) Compute the length of the query $l = j - i + 1$, then locate table T_p in which to continue the search: $p = \lceil \log \frac{2\alpha n}{(1-\alpha)l} \rceil$.

(b) Compute $b_i = \lceil \frac{i2^p}{n} \rceil$ and $b_j = \lfloor \frac{j2^p}{n} \rfloor$. Since $p = \lceil \log \frac{2\alpha n}{(1-\alpha)l} \rceil < \log \frac{2\alpha n}{(1-\alpha)l} + 1 = \log \frac{4\alpha}{(1-\alpha)l}$, we have $2^p < \frac{4\alpha n}{(1-\alpha)l}$ and $b_j - b_i = \lfloor \frac{j2^p}{n} \rfloor - \lceil \frac{i2^p}{n} \rceil \leq \frac{(j-i)2^p}{n} \leq \frac{4(j-i)\alpha}{(1-\alpha)l} \leq \frac{4\alpha}{1-\alpha}$. In other words, $Median(B_{p_{b_i}} \bigcup \ldots \bigcup B_{p_{b_j}})$ is stored in a list to which a pointer is stored in $T_{p_{b_i}}$.

(c) Output $T_{p_{b_i}}(b_j - b_i) = Median(B_{p_{b_i}} \bigcup \ldots \bigcup B_{p_{b_j}})$ as the answer.

Because each of the three steps above takes $O(1)$ time, the total time required for answering the approximate range median query is $O(1)$.

Theorem 14. *There exists a data structure of size $O(n/(1-\alpha))$ that can answer approximate range median queries in $O(1)$ time, and can be built in $O(n \log n/(1-\alpha)^2)$ time.*

4.5 Discussion

The mode and median are two of the most important statistics of a data set. It is well known that an $O(n \log n)$ algorithm and a matching lower bound exists for finding the mode and an $O(n)$ algorithm and a matching lower bound for the median of a list of n comparable elements (Cormen et al. [2001]). A natural extension is that of range mode and range median. That is, given an array of objects, we wish to know what is the mode or median of an arbitrary subarray. One obvious solution is to compute the range mode or the range median "on-demand", that is, without any preprocessing, whenever a range query comes in, the subset is extracted and the standard mode or median finding algorithm is applied on the subset. At the other extreme of the

spectrum, we anticipate all possible queries and precompute and store all the range modes or range medians in a lookup table, in which case answering a range query becomes as simple as a table lookup operation. The first solution is the most space efficient as it requires no extra storage for anything other than the original set, but the query time can be as much as $\Theta(n \log n)$ for range mode and $\Theta(n)$ for range median. The second solution, on the other hand, can answer any query in constant time, but the space used to store the lookup table is $\Theta(n^2)$ for a list of n elements.

One observation is that if two query ranges are close to each other then their modes or medians are unlikely to be different, therefore we might be able to use the same answer for multiple range queries. We pursue this idea in Chapter 4 and show that if a small amount of inaccuracy in the answer is tolerable, it is possible to use a data structure of much smaller size than that would be needed to store all possible range query results. We propose data structures and algorithms for preprocessing a list of n data elements so that approximate range mode or range median queries can be answered much more efficiently in terms of storage space and/or query time. More specifically, given an approximation factor $0 < \alpha < 1$, an element whose number of occurrences is at least α times the actual mode of the query range is found in $O(\log_2 \log_{\frac{1}{\alpha}} n)$ time using a data structure of size $O(n/(1-\alpha))$, and an approximate range median can be found in constant time using a data structure of size $O(n/(1-\alpha))$.

Although much better than the quadratic space used by the brute force solution, the more-than-linear $O(n/(1-\alpha))$ space upper bound of the proposed data structure still seems unfortunate for large data sets. However, the actual space usage is expected to be much smaller for random sets. In fact, this can be seen as a variant of the famous *occupancy problem* (Feller [1968], Motwani and Raghavan [1995]), which states that if we are to uniformly randomly assign n balls (elements in the stream) to n bins (element classes), then with probability $1 - 1/n$, no bin has more than $\frac{e \ln n}{\ln \ln n}$ balls in it. With minimum efforts, this can be translated to that with high probability, no element appears more than $\frac{e \ln n}{\ln \ln n}$ times in a list of n random elements. In other words, there are at most

$$\log_{\frac{1}{\alpha}} \frac{e \ln n}{\ln \ln n} = O\left(\log_{\frac{1}{\alpha}} \ln n\right) = O\left(\frac{\ln \ln n}{\ln \frac{1}{\alpha}}\right)$$

approximate range modes with a fixed left endpoint in an array of length n, comparing to the upper bound $O(\log_{\frac{1}{\alpha}} n)$ in Lemma 3. In other words, even the simple implementation of storing a lookup table for each of the n possible left end points would use slightly more than linear space, and this is achieved without any optimization in Section 4.3.1.

Also, it shall be noted that a significant portion of the memory is used to store answers to small query ranges. Therefore, by not storing these answers, the space usage can be further reduced if we are willing to spend slightly more time computing queries of short ranges (on demand). This hybrid approach may be an attractive solution to mobile applications in which a mobile user's query is processed by the server if the query range is large; queries of short range will be processed by first downloading the data sets from the server and then processed on the mobile devices, thus achieving much reduced memory space usage on the server without imposing any significant mobile device computing or network transmission overheads.

Chapter 5

Approximate Range Queries in Higher Dimensional Space

In this chapter we extend the idea of using the same precomputed answer for adjacent approximate range queries to higher dimensional space. Here the input is a d-dimensional ($d \geq 2$) matrix[1]

$$A = \{a_{x_1,\ldots,x_d} | 1 \leq x_1 \leq n, \ldots, 1 \leq x_d \leq n\}$$

and an approximation factor $0 < \alpha < 1$. Given two entries in the matrix, $a_{i_1,\ldots,i_d}$ and $a_{j_1,\ldots,j_d}$, where $i_1 \leq j_1, \ldots, i_d \leq j_d$, the query range is the d-dimensional rectangle

$$QR_{(i_1,\ldots,i_d),(j_1,\ldots,j_d)} = \{a_{x_1,\ldots,x_d} | i_1 \leq x_1 \leq j_1, \ldots, i_d \leq x_d \leq j_d\}$$

The approximate range mode is an element $x \in QR_{(i_1,\ldots,i_d),(j_1,\ldots,j_d)}$ such that

$$f_x(QR_{(i_1,\ldots,i_d),(j_1,\ldots,j_d)}) \geq \alpha \times F(QR_{(i_1,\ldots,i_d),(j_1,\ldots,j_d)})$$

where $f_x(QR_{(i_1,\ldots,i_d),(j_1,\ldots,j_d)})$ is the frequency[2] of x in the rectangle $QR_{(i_1,\ldots,i_d),(j_1,\ldots,j_d)}$ and $F(QR_{(i_1,\ldots,i_d),(j_1,\ldots,j_d)}) = \max_x f_x(QR_{(i_1,\ldots,i_d),(j_1,\ldots,j_d)})$ is the number of occurrences of the most frequent element (*i.e., mode*[3]) in $QR_{(i_1,\ldots,i_d),(j_1,\ldots,j_d)}$. Similarly, the approximate range median is defined with respect to the set of all elements in the rectangle (range).

5.1 Approximate Range Queries in Two Dimensional Space

We first study approximate range mode queries in two dimensional space, here the input is an $n \times n$ matrix $A = \{a_{i,j} | 1 \leq i, j \leq n\}$ and a query range is the rectangle defined by two entries: the lower left corner a_{i_1,i_2} and the upper right corner a_{j_1,j_2}

[1] We assume square matrices to simplify presentation. The results obtained in this chapter can be generalized to non-square matrices.

[2] We use frequency and number of occurrences interchangeably throughout this chapter.

[3] Obviously, there could be several modes of a query range, but they all have the same frequency.

$(i_1 \leq j_1, i_2 \leq j_2)$. We show that we can construct a data structure on A such that given a query range $QR_{(i_1,i_2),(j_1,j_2)}$, we can quickly decide which of the input data elements is an approximate range mode or range median of the subset

$$\left\{ a_{x_1,x_2} \middle| i_1 \leq x_1 \leq j_1, i_2 \leq x_2 \leq j_2 \right\}.$$

The idea is to draw a sequence of *mode curves*

$$\left\{ C_k \middle| k = 1, \lfloor 1/\alpha \rfloor, \lfloor 1/\alpha^2 \rfloor, \ldots, \lfloor 1/\alpha^{\lfloor \log_{\frac{1}{\alpha}} n^2 \rfloor} \rfloor \right\}$$

each representing a new range mode

$$C_k = \left\{ (i,j) \middle| F\left(QR_{(0,0),(i,j)}\right) \geq k \wedge F\left(QR_{(0,0),(i,j-1)}\right) < k, i = 0, \ldots, n-1. \right\}.$$

Obviously C_1 is always the origin $(C_1 = (0,0))$, while other mode curves are specified by a set of entries in the matrix that represent the "envelopes" of range modes. As an example shown in Figure 5.1, C_4 is defined by 16 entries whose x-coordinate goes from 0 to 15

$$\begin{aligned} C_4 \;=\; & \{(0,4),(1,2),(2,2),(3,1),(4,1),(5,1),(6,0),(7,0), \\ & (8,0),(9,0),(10,0),(11,0),(12,0),(13,0),(14,0),(15,0)\}. \end{aligned}$$

Any rectangle with the origin being its lower left corner and its upper right corner above C_4 contains at least one element that occurs at least 4 times in the rectangle, while no rectangle whose upper right corner is below C_4 contains such elements. This immediately gives a naive solution to the range queries in two dimensional space. A set of $O(\log_{\frac{1}{\alpha}} n)$ mode curves are stored for each possible lower left corner, resulting a total storage space of $O(n^3 \log_{\frac{1}{\alpha}} n)$.

The $O(n)$ storage space for each such mode curve seems excessive for the objective of using a small data structure to quickly answer range queries. Next we show how a mode curve can be stored using less space. Then we show that it suffices to store a relatively small number of such mode curves as opposed to $O(\log_{\frac{1}{\alpha}} n)$ mode curves for every element. As a result, the total space usage is much smaller than that by the naive solution.

First, we prove that these mode curves are *monotonic*, *i.e.*, given any curve C_k and two points on it: $(x_1, y_1) \in C_k$ and $(x_2, y_2) \in C_k$, if $x_1 < x_2$ then we have $y_1 \geq y_2$.

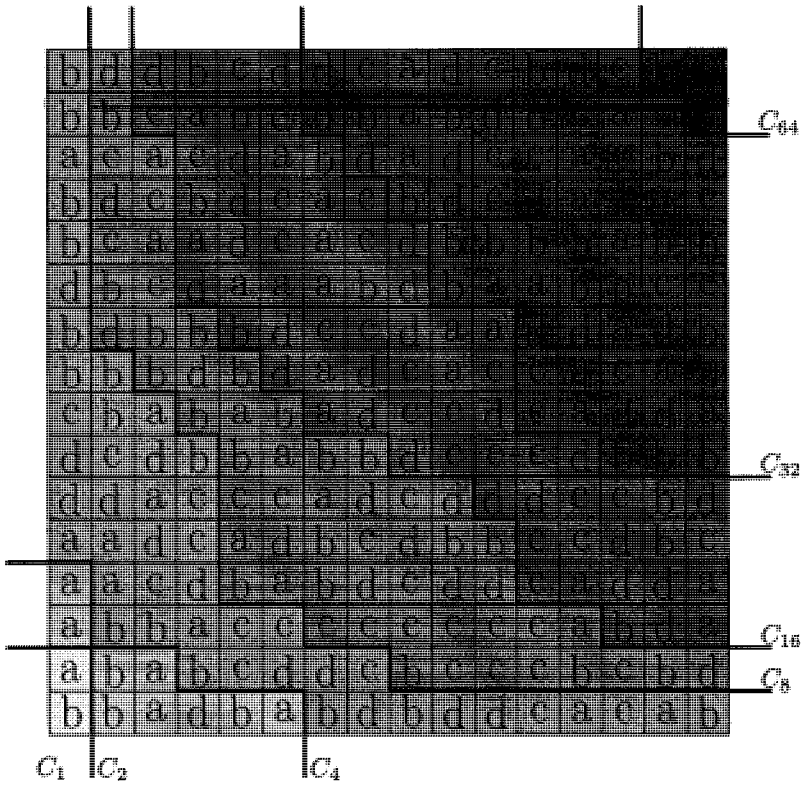

Figure 5.1: A 16 by 16 matrix partitioned by 6 mode curves.

Lemma 7. *The C_k are monotonic.*

Proof. By contradiction. Otherwise, suppose there exists a mode curve, say C_k that is not monotonic because there are two points $(x_1, y_1) \in C_k$ and $(x_2, y_2) \in C_k$ such that $x_1 < x_2$ and $y_1 < y_2$. Consequently we have

$$\begin{aligned} F\left(QR_{(0,0),(x_2,y_2-1)}\right) &\geq F\left(QR_{(0,0),(x_2,y_1)}\right) \\ &\geq F\left(QR_{(0,0),(x_1,y_1)}\right) \\ &= k. \end{aligned}$$

On the other hand, since $(x_2, y_2) \in C_k$, we have $F\left(QR_{(0,0),(x_2,y_2-1)}\right) < k$, a contradiction. □

Figure 5.1 illustrates that all the C_k are "step function" like. Given an arbitrary entry (x, y) between two adjacent mode curves C_{2^k} and $C_{2^{k+1}}$, the mode of the rectangle between $(0, 0)$ and (x, y) has frequency of at least 2^k but less than 2^{k+1}. On

the other hand, given an arbitrary entry in the matrix, there could be up to n^2 query ranges that have it as their lower left corner. A naive solution to the range mode queries is to pre-compute and store for this entry all the $O(n^2)$ range modes. The following lemma says that we can do much better in terms of storage space if we are willing to accept a small amount of inaccuracy in the answer.

Lemma 8. *There exists a data structure of size $O(n \log_{\frac{1}{\alpha}} n)$ that can answer approximate range mode queries on an $n \times n$ matrix with $(0,0)$ being the fixed lower left corner in $O(\log_2 \log_{\frac{1}{\alpha}} n)$ time.*

Proof. The frequency of any element in an $n \times n$ matrix is at most n^2, therefore at most $\log_{\frac{1}{\alpha}}(n^2)$ mode curves need to be stored for the matrix. Each of these curves can be uniquely specified by at most n points whose x-coordinates run from 1 to n. Therefore, the total storage space needed for storing these curves is $O(n \log_{\frac{1}{\alpha}} n^2) = O(n \log_{\frac{1}{\alpha}} n)$.

With these curves stored in a (at most) $\log_{\frac{1}{\alpha}} n$ row and n column table, in which the ith row corresponds to the ith curve $C_{\left(\frac{1}{\alpha}\right)^i}$, answering an approximate range mode query $QR_{(0,0),(x,y)}$ becomes a simple table lookup operation followed by a binary search in a sorted array: first the jth column is located, which can be easily proved to be already sorted, then the two curves are found between which (x,y) is located in $O(\log_2 \log_{\frac{1}{\alpha}} n)$ time. Suppose the two curves are $C_{\left(\frac{1}{\alpha}\right)^i}$ and $C_{\left(\frac{1}{\alpha}\right)^{i+1}}$, then we know that the frequency of the mode $F\left(QR_{(0,0),(x,y)}\right)$ is at least $\left(\frac{1}{\alpha}\right)^i$ but less than $\left(\frac{1}{\alpha}\right)^{i+1}$, with guaranteed frequency of at least α times that of the actual mode, the jth entry of $C_{\left(\frac{1}{\alpha}\right)^i}$ with its estimated frequency $\left(\frac{1}{\alpha}\right)^i$ is output as the approximate mode of $F\left(QR_{(0,0),(x,y)}\right)$. □

As an example, given the input matrix in Figure 5.1 and approximation factor $\alpha = 1/2$, the seven mode curves are stored in Table 5.1. Query $QR_{(0,0),(10,8)}$ is answered by first locating the 10th column, then finding that the y-coordinate of the query range is between the 10th entries of C_{16} and C_{32} $(5 < 8 \leq 10)$. The $(d, 16)$ is output as the approximate range mode of $QR_{(0,0),(10,8)}$ (note that the actual mode, c, occurs 22 times in the query range).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C_1	b,0	b,0	b,0	b,0	b,0	b,0	b,0	b,0	b,0	b,0	b,0	b,0	b,0	b,0	b,0	b,0
C_2	a,2	b,0	b,0	b,0	b,0	b,0	b,0	b,0	b,0	b,0	b,0	b,0	b,0	b,0	b,0	b,0
C_4	a,4	b,2	b,2	b,1	b,1	b,1	b,0	b,0	b,0	b,0	b,0	b,0	b,0	b,0	b,0	b,0
C_8	*,∞	b,9	b,8	b,7	b,3	b,3	b,2	b,2	b,1	b,1	b,1	b,1	b,1	b,1	b,1	b,1
C_{16}	*,∞	*,∞	b,14	b,10	b,9	b,8	b,7	b,7	b,6	b,6	d,5	c,3	c,3	c,2	c,2	c,2
C_{32}	*,∞	*,∞	*,∞	*,∞	*,∞	*,∞	b,14	b,13	b,12	b,10	b,10	c,7	c,7	c,6	c,6	c,6
C_{64}	*,∞	*,∞	*,∞	*,∞	*,∞	*,∞	*,∞	*,∞	*,∞	*,∞	*,∞	*,∞	*,∞	*,∞	b,15	b,14

Table 5.1: A lookup table of 7 rows and 16 columns is used for answering $\frac{1}{2}$-approximate range mode queries on the 16 by 16 matrix in Figure 5.1.

By storing a lookup table for each possible lower left corner of a query range in the matrix, we immediately have the first non-trivial solution to the approximate range mode queries in two dimensional space.

Theorem 15. *There exists a data structure of size* $O\left(n^3 \log_{\frac{1}{\alpha}} n\right)$ *that can answer approximate range mode queries on an* $n \times n$ *matrix in* $O\left(\log_2 \log_{\frac{1}{\alpha}} n\right)$ *time.*

Proof. The data structure is a set of n^2 lookup tables $T_{i,j}(1 \leq i \leq n, 1 \leq j \leq n)$, each corresponding to an entry of the matrix and can be stored using $O(n \log_{\frac{1}{\alpha}} n)$ space[4]. Pointers to these tables are stored in an array. To answer an arbitrary approximate range mode query $QR_{(x_1,y_1)(x_2,y_2)}$, first the pointer to T_{x_1,y_1} is located in $O(1)$ time, then we search in T_{x_1,y_1} for the appropriate approximate range mode, which can be done in $O(\log_2 \log_{\frac{1}{\alpha}} n)$ time as already shown in Lemma 8. □

5.1.1 An Improvement Based on Second Order Approximation

We have seen that by storing relatively few mode curves ($\log_{\frac{1}{\alpha}} n$ mode curves vs. n^2 entries in an $n \times n$ matrix), the approximate range mode can be found on the curve that is close to the query point, this works because the difference between the frequencies of the actual range mode and the approximate range mode is bounded by $\frac{1}{\alpha}$. The reduction in storage space is achieved at the cost of accuracy by using a nearby mode curve to approximate the actual mode. However, it still takes as much as $\Theta(n)$ space to explicitly specify a mode curve. Next we show that if $\alpha < 1/4$, the storage space can be further reduced by approximating these mode curves.

[4]Note that the size of the input is n^2.

The idea is based on the observation of the overall trend of increasing gaps between adjacent mode curves as they are further away from the origin. In fact, by choosing the distance measure between two adjacent mode curves appropriately, we can show that this distance increases exponentially.

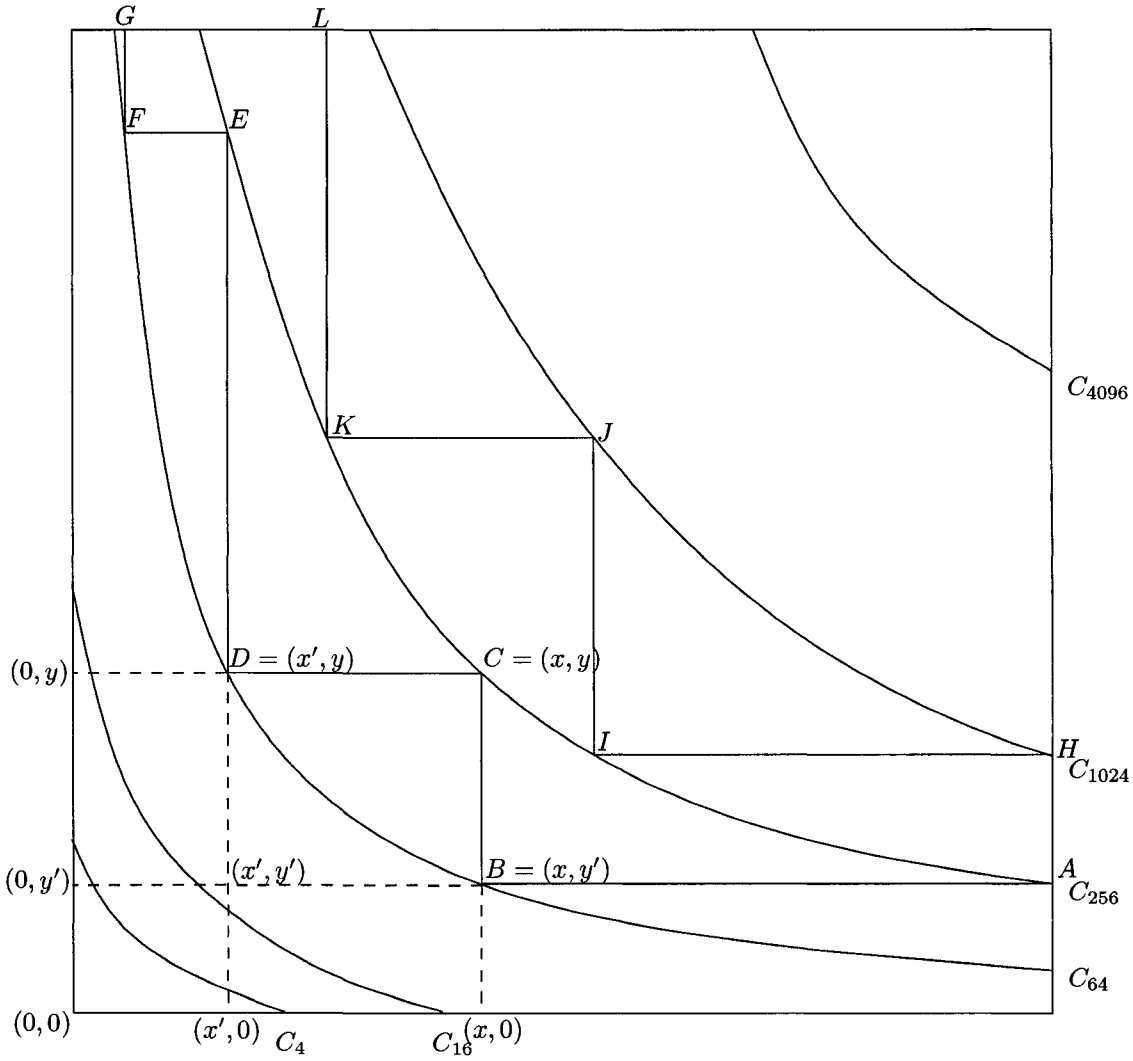

Figure 5.2: The "distance" between mode curves increases as moving away from the origin.

Definition 1. *The distance between two mode curves C_i and C_j $(i < j)$ is defined as:*

$$d(C_i, C_j) = \min_{(x,y)\in C_j \wedge (x',y)\in C_i \wedge (x,y')\in C_i} |x - x'| + |y - y'|$$

Lemma 9. *The distance as defined in Definition 1 between C_i and C_j is greater than $2\sqrt{j-2i}$ if $j > 2i$.*

Proof. Let $C = (x, y)$ be an arbitrary point on C_j, and its two projections on C_i are $B = (x, y')$ and $D = (x', y)$, as shown in Figure 5.2. Since C is on C_j, there must be an element that occurs at least j times in the rectangle $QR_{(0,0),(x,y)}$, i.e., $F(QR_{(0,0),(x,y)}) \geq j$. Similarly, we have $F(QR_{(0,0),(x',y)}) \geq i$ and $F(QR_{(0,0),(x,y')}) \geq i$. By definition of $F(.)$, we have:

$$\begin{aligned} F(QR_{(0,0),(x,y)}) &\leq F(QR_{(0,0),(x',y')}) + F(QR_{(x',0),(x,y')}) + F(QR_{(0,y'),(x',y)}) \\ &\quad + F(QR_{(x',y'),(x,y)}) \\ &\leq F(QR_{(0,0),(x',y')}) + F(QR_{(x',0),(x,y')}) \\ &\quad + F(QR_{(0,0),(x',y')}) + F(QR_{(0,y'),(x',y)}) \\ &\quad + F(QR_{(x',y'),(x,y)}) \\ &\leq F(QR_{(0,0),(x,y')}) + F(QR_{(0,0),(x',y)}) + F(QR_{(x',y'),(x,y)}). \end{aligned}$$

Thus,

$$\begin{aligned} F(QR_{(x',y'),(x,y)}) &\geq F(QR_{(0,0),(x,y)}) - F(QR_{(0,0),(x,y')}) - F(QR_{(0,0),(x',y)}) \\ &\geq j - 2i \end{aligned}$$

On the other hand, there are in total $|x-x'| \times |y-y'|$ elements (counting repetitions) in $QR_{(x',y'),(x,y)}$, therefore

$$F(QR_{(x',y'),(x,y)}) \leq |x - x'| \times |y - y'|.$$

It follows immediately that

$$|x - x'| + |y - y'| \geq 2\sqrt{|x - x'| \times |y - y'|} = 2\sqrt{j - 2i}.$$

□

Next we shall see how to use this fact to approximately store mode curves using a data structure of much smaller space. The idea is to draw a step function like polyline

between two adjacent mode curves, as a result, each mode curve except the first and the last one is bounded by two adjacent polylines. For example, in Figure 5.2, C_{256} is bounded by two polylines $\overline{ABCDEFG}$ and $\overline{HIJKL}$. On the other hand, $\overline{ABCDEFG}$ is bounded by C_{64} and C_{256} and it can be proved using Lemma 5.2 that it can be specified by at most $\frac{n}{8\sqrt{2}}+1$ intersections (A,C,E and G in Figure 5.2); similarly, $\overline{HIJKL}$ is bounded by C_{256} and C_{1024} and can be represented by a list of at most $\frac{n}{16\sqrt{2}}+1$ intersections (H, J and L). In general, given approximation ratio $\alpha < \frac{1}{2}$, the polyline between $C_{(\frac{1}{\alpha})^i}$ and $C_{(\frac{1}{\alpha})^{i+1}}$ can be specified by at most $\frac{n}{\sqrt{(\frac{1}{\alpha})^{i+1}-2(\frac{1}{\alpha})^i}}+1$ intersections, in other words, the space needed to store all such polylines (there are at most $\log_{\frac{1}{\alpha}} n^2$ of them) is given by

$$
\begin{aligned}
\sum_{i=1}^{\log_{\frac{1}{\alpha}} n^2} \left(\frac{n}{\sqrt{(\frac{1}{\alpha})^{i+1}-2(\frac{1}{\alpha})^i}} + 1 \right) &= \log_{\frac{1}{\alpha}} n^2 + \frac{n}{\sqrt{\frac{1}{\alpha}-2}} \times \sum_{i=1}^{\log_{\frac{1}{\alpha}} n^2} (\sqrt{\alpha})^i \\
&= 2\log_{\frac{1}{\alpha}} n + \frac{n}{\sqrt{\frac{1}{\alpha}-2}} \times \frac{1-(\sqrt{\alpha})^{\log_{\frac{1}{\alpha}} n^2+1}}{1-\sqrt{\alpha}} \\
&= 2\log_{\frac{1}{\alpha}} n + \frac{n}{\sqrt{\frac{1}{\alpha}-2}} \times \frac{1-\alpha^{\log_{\frac{1}{\alpha}} n} \times \sqrt{\alpha}}{1-\sqrt{\alpha}} \\
&= 2\log_{\frac{1}{\alpha}} n + \frac{n}{\sqrt{\frac{1}{\alpha}-2}} \times \frac{1-\frac{1}{n} \times \sqrt{\alpha}}{1-\sqrt{\alpha}} \\
&= O\left(\frac{n}{\sqrt{1-2\alpha}} \right).
\end{aligned}
$$

To answer an approximate range mode query $QR_{(0,0),(x,y)}$, we first locate the two polylines between which lies (x, y). Suppose these two polylines are bounded by $C_{(\frac{1}{\alpha})^i}$, $C_{(\frac{1}{\alpha})^{i+1}}$ and $C_{(\frac{1}{\alpha})^{i+2}}$ respectively, then we know that $F(QR_{(0,0),(x,y)})$ is at least $(\frac{1}{\alpha})^i$ but no more than $(\frac{1}{\alpha})^{i+2}$. Any point on the lower left polyline that is also dominated[5] by (x, y) can be output (along its associated frequency of $(\frac{1}{\alpha})^i$ or $(\frac{1}{\alpha})^{i+1}$) as the approximate range mode. By storing for each entry in the matrix the polylines that approximate mode curves using $O(\frac{n}{\sqrt{1-2\alpha}})$ space, we immediately have a data structure that can answer approximate range mode queries which is much more space efficient albeit at cost of being less accurate.

[5] $P_1(x_1, y_1)$ dominates $P_2(x_2, y_2)$ if and only if $x_1 \geq x_2$ and $y_1 \geq y_2$.

Theorem 16. *For $0 < \alpha < \frac{1}{4}$ there exists a data structure of size $O\left(\frac{n^3}{\sqrt{1-2\alpha}}\right)$ that can answer approximate range mode queries on an $n \times n$ matrix in $O\left(\log_2 \frac{n}{\sqrt{1-2\alpha}}\right)$ time* [6].

Proof. The data structure is a set of n^2 lookup tables each of size $O(\frac{n}{\sqrt{1-2\alpha}})$. Also pointers to these tables are stored in an array. To answer an approximate range mode query $QR_{(x_1,y_1),(x_2,y_2)}$, first the pointer to the table that corresponds to (x_1, y_1) is located in $O(1)$ time, then the approximate range mode of $QR_{(x_1,y_1),(x_2,y_2)}$ is found in this table, which takes $O(\log_2 \frac{n}{\sqrt{1-2\alpha}})$ time by the well known result in computational geometry on *point location problem* (Kirkpatrick [1983]). □

5.2 Approximate Range Median Queries in Higher Dimensional Space

In this section, we consider approximate range median queries in d-dimensional space ($d \geq 2$). Here the input is a d-dimensional matrix containing n^d comparable elements: $A = \{a_{x_1,\ldots,x_d} | 1 \leq x_1 \leq n, \ldots, 1 \leq x_d \leq n\}$. Given an approximation factor $0 < \alpha < 1$, our task is to preprocess A so that, given a query range $QR_{(i_1,\ldots,i_d),(j_1,\ldots,j_d)}$, we can quickly find an element in the query range $a_{x_1,\ldots,x_d}(i_1 \leq x_1 \leq j_1, \ldots, i_d \leq x_d \leq j_d)$ whose rank among all elements in the query range is between $\alpha \times \lfloor \frac{(j_1-i_1+1)\times\ldots\times(j_d-i_d+1)}{2} \rfloor$ and $(2-\alpha) \times \lfloor \frac{(j_1-i_1+1)\times\ldots\times(j_d-i_d+1)}{2} \rfloor$.

Our solution is based on a similar idea to that of the approximate range median queries on lists. If a query range covers many blocks, then the (possibly partial) blocks at the boundaries can be ignored as their contribution is minimal. As a result, we can return the precomputed range median of the union of the internal blocks as an approximate range median. The key question here is how to partition the input matrix into blocks of different sizes so that the number of precomputed/stored range medians is not excessively large when given an arbitrary query range, in the mean time, there is always a union of blocks that is close enough to the query range and its precomputed exact median sufficiently approximates the median of the query range.

[6] Note that the size of input is $O(n^2)$ and the naive solution of storing a mode for each possible query range would require $O(n^4)$ space.

5.2.1 $O\left(\frac{8^d n^d}{(1-\alpha^{\frac{1}{d}})^d}\right)$ Storage Space

To simplify the presentation we assume $n = 2^k$ for some integer $k \geq 1$. Generalization to arbitrary n is straightforward. As an example shown in Figure 5.3 in which a matrix has k levels of partitions in each dimension. At the ith level ($1 \leq i \leq k$), the n columns (rows) are partitioned into 2^i non-overlapping intervals of length $n/2^i$. In other words, there are in total $\sum_{i=1}^{\log n} 2^i = 2n-2$ (overlapping) intervals of various lengths. In each dimension, a *super-interval* is the union of up to $\frac{4\alpha^{\frac{1}{d}}}{1-\alpha^{\frac{1}{d}}}$ consecutive intervals of the same size (the reason for this will be clear next). A *super-block* in a d-dimensional matrix is a union of blocks whose boundary in each dimension is aligned with a super-interval at some partition level. Because each level-i interval in one dimension may have up to $\frac{4\alpha^{\frac{1}{d}}}{1-\alpha^{\frac{1}{d}}}$ associated super-intervals of sizes $n/2^i, 2 \times n/2^i, 3 \times n/2^i, \ldots, \frac{4\alpha^{\frac{1}{d}}}{1-\alpha^{\frac{1}{d}}} \times n/2^i$, respectively. The number of super-intervals in one dimension is given by

$$\sum_{i=1}^{\log_2 n} 2^i \times \frac{4\alpha^{\frac{1}{d}}}{1-\alpha^{\frac{1}{d}}} = \frac{(8n-8)\alpha^{\frac{1}{d}}}{1-\alpha^{\frac{1}{d}}}.$$

Each set of d super-intervals (one in each dimension) uniquely define a d-dimensional super-block. Therefore the number of these super-blocks whose range medians will be precomputed and stored is

$$\Theta\left(\left(\frac{(8n-8)\alpha^{\frac{1}{d}}}{1-\alpha^{\frac{1}{d}}}\right)^d\right) = \Theta\left(\frac{8^d n^d}{(1-\alpha^{\frac{1}{d}})^d}\right).$$

The data structure used to store these precomputed range medians is a set of lookup tables $T_{k_1 \ldots k_d} (1 \leq k_1, \ldots, k_d \leq \log_2 n)$. Each table $T_{k_1 \ldots k_d}$ corresponds to the set of all super-blocks whose boundary in each dimension is aligned with super-intervals at the same partition level. That is, we store in $T_{k_1 \ldots k_d}$ the exact medians of all super-blocks whose first dimensions are unions of up to $\frac{4\alpha^{\frac{1}{d}}}{1-\alpha^{\frac{1}{d}}}$ consecutive intervals of length $\frac{n}{2^{k_1}}$, and second dimensions unions of up to $\frac{4\alpha^{\frac{1}{d}}}{1-\alpha^{\frac{1}{d}}}$ consecutive intervals of length $\frac{n}{2^{k_2}}$, etc. It is easy to see that $T_{k_1 \ldots k_d}$ has $\left(\frac{n}{2^{k_1}} \times \frac{4\alpha^{1/d}}{1-\alpha^{1/d}}\right) \times \cdots \times \left(\frac{n}{2^{k_d}} \times \frac{4\alpha^{1/d}}{1-\alpha^{1/d}}\right)$ entries, to facilitate fast query, these precomputed range medians are organized in lexicographic order according to the starting interval and span in each dimension.

5.2.2 $O\left(\frac{4^d \alpha n^d \log_2^d n}{(1-\alpha^{\frac{1}{d}})^d}\right)$ Preprocessing Time

At level i partition of each dimension, there are 2^i intervals each of length $n/2^i$, and each interval has up to $\frac{4\alpha^{\frac{1}{d}}}{1-\alpha^{\frac{1}{d}}}$ associated super-intervals. Therefore, at each of the d dimension, an element can be part of up to

$$\log_2 n \times \frac{4\alpha^{\frac{1}{d}}}{1-\alpha^{\frac{1}{d}}}$$

super-intervals. It follows immediately that the number of d-dimensional super-blocks an element is part of is at most

$$\left(\log_2 n \times \frac{4\alpha^{\frac{1}{d}}}{1-\alpha^{\frac{1}{d}}}\right)^d.$$

The trivial linear time algorithm is used to compute the exact medians of the super-blocks that are aligned with these super-intervals. In other words, the running time spent on each element in the super-block is $O(1)$ in average. To compute the total running time, we note that the running time spent on an element depends on the number of the super-blocks which it is part of. Since there are n^d elements in the matrix, the total preprocessing time is given by

$$\begin{aligned}\text{Total preprocessing time} &= O\left(n^d \times \left(\log_2 n \times \frac{4\alpha^{\frac{1}{d}}}{1-\alpha^{\frac{1}{d}}}\right)^d\right) \\ &= O\left(\frac{4^d \alpha n^d \log_2^d n}{(1-\alpha^{\frac{1}{d}})^d}\right).\end{aligned}$$

5.2.3 $O(d)$ Query Time

To compute an approximate range median of $QR_{(i_1,\ldots,i_d),(j_1,\ldots,j_d)}$, first the length of the query range in each dimension is computed in $O(d)$ time

$$l_k = j_k - i_k + 1, k = 1, \ldots, d.$$

Next, the table $T_{x_1,\ldots,x_d}$ in which to continue the search is located:

$$x_k = \max\left\{i \middle| \frac{4\alpha^{1/d}}{1-\alpha^{1/d}} \geq \frac{l_k}{n/2^i} \geq \frac{2\alpha^{1/d}}{1-\alpha^{1/d}}\right\}.$$

This also takes $O(d)$ time and guarantees that in each dimension the query range spans at least $\frac{2\alpha^{1/\alpha}}{1-\alpha^{1/\alpha}}$ complete level i intervals. Now, compare the length of the union of these internal intervals and that of the query range which may contain at most two more partial intervals, one at each end

$$\begin{aligned} \frac{\lfloor \frac{l_k}{n/2^i} \rfloor \times \frac{n}{2^i}}{l_k} &\geq \frac{\frac{2\alpha^{1/d}}{1-\alpha^{1/d}}}{\frac{2\alpha^{1/d}}{1-\alpha^{1/d}} + 2}. \\ &= \alpha^{1/d}. \end{aligned}$$

On the other hand, the query range spans no more than $\frac{4\alpha^{1/d}}{1-\alpha^{1/d}}$ intervals in each dimension, in other words, the union of these intervals is in fact a super-interval which in turn defines a super-block whose exact median is precomputed. Since the length of this super-block in each dimension is at least $\alpha^{1/d}$ times that of the actual query range, it follows immediately that the size of the super-block is at least α times that of the actual query range and so are the frequencies of their respective range medians.

Finally, to locate the precomputed range median in $T_{x_1,\ldots,x_d}$, it suffices to compute for each dimension the first interval and the number of the intervals covered by the query range. Which can be easily shown to take $O(d)$ time.

Theorem 17. *Given a d-dimensional ($d \geq 2$) input matrix that is of length n in each dimension, and approximation factor $0 < \alpha < 1$, there is a data structure of size $O\left(\frac{8^d n^d}{(1-\alpha^{\frac{1}{d}})^d}\right)$, which can be constructed in $O\left(\frac{4^d \alpha n^d \log_2^d n}{(1-\alpha^{\frac{1}{d}})^d}\right)$ time, and can answer approximate range median queries in $O(d)$ time.*

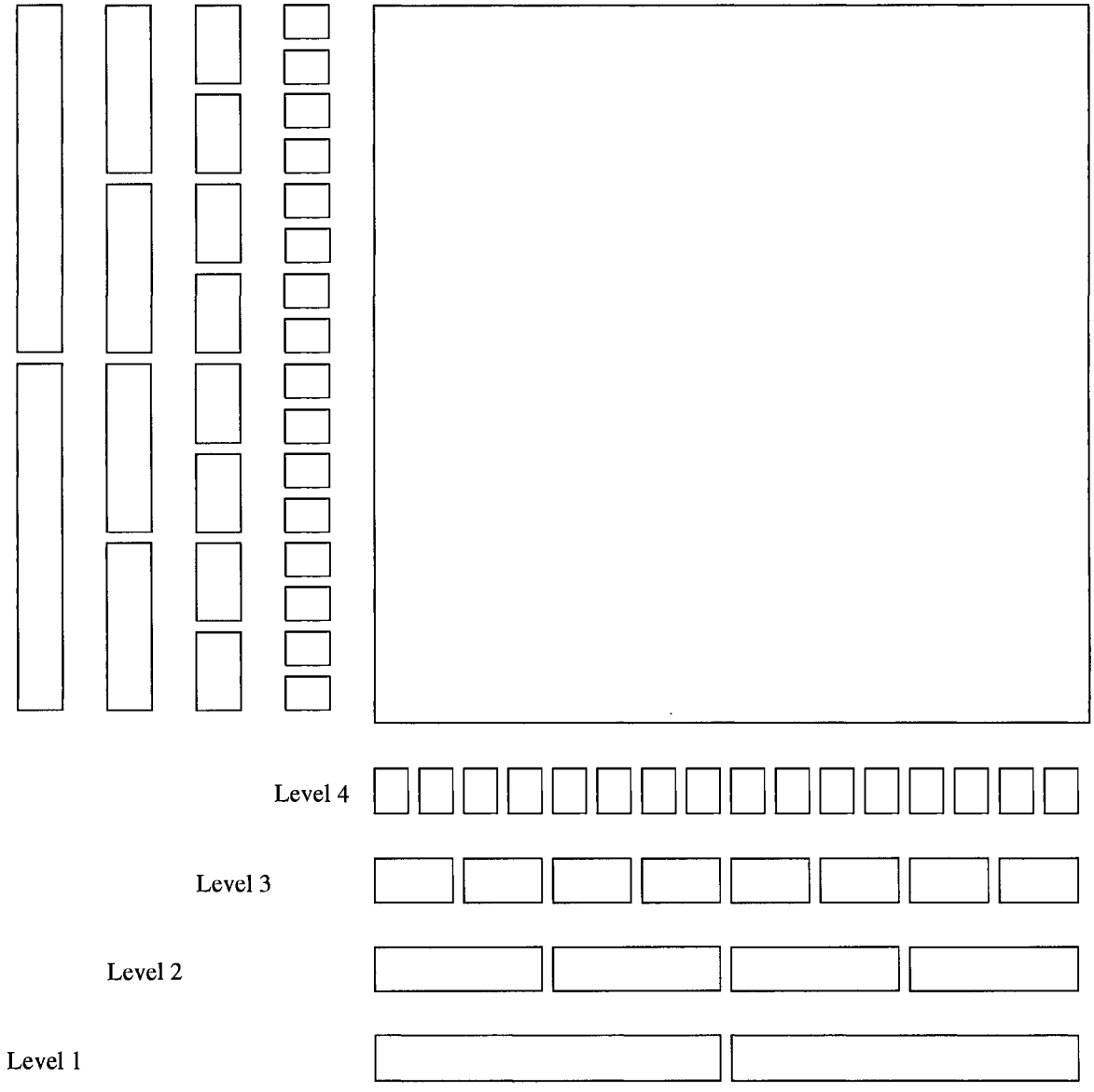

Figure 5.3: The input 16 by 16 matrix ($n = 16$) is divided into $2n - 2$ (overlapping) intervals in each dimension.

Chapter 6

Sorting on Tape

One of the earliest research results in the data stream model is the paper by Munro and Paterson [1980]. In this seminal paper, the authors studied the problem of sorting data elements stored on tapes using limited internal memory.

When data are stored on tapes, the only practical way of accessing these data is through sequential scan. Munro and Paterson [1980] studied the trade-offs between the amount of internal memory and the number of passes over the input data stored on tape. They proved a lower bound on the number of passes required as a function of the amount of memory. They also proposed an algorithm that achieves an upper bound that is 4 times the lower bound. In this chapter, we first close this gap by presenting a matching lower bound. Then we study a less restricted model for sorting data stored on tapes. We show that in this new model, no algorithm can do much better than the naive algorithm given the same amount of internal memory as there is a probabilistic lower bound on the number of passes any algorithm will have to make, which is one third of that by the naive algorithm.

6.1 Introduction

Suppose we want to sort a set of n distinct data elements stored on a one-way read-only tape. Initially the memory is empty and the read head is placed at the beginning of the tape. After each pass the tape is rewound to this starting position with no reading permitted during the rewind. During a pass, the element under the read head can be read into one of $m(< n)$ locations in internal random access memory and a comparison can be made between any two elements in internal memory. Furthermore, we assume that the cost of comparisons within internal memory is negligible compared to the I/O cost. In other words, the overall running time of sorting these data is dominated by the I/O cost, which is proportional to the number of passes.

When the internal memory is smaller than the size of the input data, several passes will be required and the goal is to make as few passes as possible.

6.2 The Simple Model

We start with a simple model in which there is no storage for the output, and the rank of an element is implicitly specified by the order in which the element is output. This is the same model used in Munro and Paterson's original paper (Munro and Paterson [1980]) in which a lower bound of $\lfloor \frac{n}{4m} \rfloor$ passes is established using an argument as follows: to sort a list of n elements, one has to make a minimum of $\lfloor \frac{n}{2} \rfloor$ comparisons, one comparison between the $(2i-1)$-th and the $2i$-th largest element for each $i \in \{1, 2, \ldots, \frac{n}{2}\}$. On the other hand, the adversary can construct the following input: all odd-ranked elements are placed in the first half of the tape and all even-ranked elements in the second half. Given this input, to make as many comparisons as possible in every pass, the best any algorithm can do is to carry m odd-ranked elements forward to "meet" their even-ranked neighbors, and m even-ranked elements backward when rewinding to meet their odd-ranked neighbors. It follows immediately that at most $2m$ comparisons can be made in one pass (which in fact includes one forwarding followed by one rewinding of the tape) and therefore $\lfloor \frac{n}{4m} \rfloor$ passes are required to sort and output n data elements. This is essentially the communication complexity lower bound if Alice has the first half of the input and Bob the second.

Munro and Paterson also proposed a simple algorithm that achieves an upper bound which is about 4 times the lower bound. The algorithm works as follows: in the first pass, the m smallest elements are found, sorted and output. In each of the subsequent passes, one memory location is used to hold the largest element that has been output so that any elements ranked in previous passes will be ignored and the next $m-1$ largest elements are ranked. This algorithm requires $\lceil \frac{n-m}{m-1} \rceil + 1 = \lceil \frac{n-1}{m-1} \rceil$ passes for any input of n data elements.

Theorem 18. *(Munro and Paterson [1980]) Given m internal storage locations, it takes at least $\lfloor \frac{n}{4m} \rfloor$ but no more than $\lceil \frac{n-1}{m-1} \rceil$ passes to sort n elements.*

Next we will show that a simple argument can be made to close the gap between

the lower bound and the upper bound in this model.

Theorem 19. $\lceil \frac{n}{m} \rceil$ *passes are necessary in the worst case to sort* n *elements stored on a read-only tape using* m *internal storage locations.*

Proof. Assume without loss of generality that the elements are to be sorted and output in increasing order. Because the rank of an element is specified by the order in which it is output, the element will be output only after all the smaller elements have been output. Therefore, in the first pass, no elements will be output until the smallest one is found and output, which can only happen at the end of the first pass if the input is such that the data are stored on tape in descending order. At the end of the first pass, at most m elements are stored, sorted and output. Similarly, in each of the subsequent passes, at most m elements can be sorted and output, therefore at least $\lceil \frac{n}{m} \rceil$ passes are required to sort all n elements. □

6.3 A Relaxation

We have shown that no algorithm can do better than the naive algorithm if we are to output elements in sorted order. Next we study the sorting problem in a slightly relaxed model which we show to have more interesting time-space tradeoffs. In this model, we do not require elements to be output in sorted order, instead, the elements and their associated ranks can be output in any order. Although the lower bound no longer holds in this model, the upper bound achieved by the $\lceil \frac{n-1}{m-1} \rceil$-pass algorithm still holds. We present new lower bounds in the following section.

6.3.1 Deterministic Lower Bound

Our first lower bound is based on the observation that to determine an element's rank, at least two comparisons have to be made. The element must be compared to its two immediate rank neighbors, one larger and the other one smaller than the element. The only exceptions are the smallest and largest element, each of which has only one neighbor.

It is easy to see that at least $n - 1$ comparisons have to be made in order to determine the rankings of n elements. However, the argument used to prove Theorem

18 is no longer valid because it is now possible that more than m such comparisons can be made in one pass even given the worst case input. In essence, in the simple model, every comparison is considered to be of the same cost. That is, it takes exactly one memory location to store an element in order to make a comparison in the first half of a pass (forwarding), and this memory location can not be reused for another comparison until the rewinding, during which the same memory location may be used to make exactly one other comparison. In other words, a memory location is used to make at most two comparisons in one pass. This is no longer true in the new model, for example, given the same odd-ranked-then-even-ranked input, we can carry 3 forward to meet 2 and 4 which are located in the second half of the tape, then use the same memory to store 6 during rewinding and make another two comparisons with 5 and 7. Although we now need to make twice as many comparisons in the new model, it is not necessarily true that we will need to make twice as many passes as the memory may be used more efficiently.

On the other hand, the $\frac{n}{m}$ lower bound in Theorem 19 did not take into consideration any flexibilities that now exist in the new model. In order to improve the lower bound for the new model, we will not only look at the number of comparisons that have to be made, but also how the memory can be reused so as to make more comparisons in one pass. More specifically, we will look at the cost of each comparison, which is defined as the product of the amount of memory used to store an element (space) and for how long an element will have to be kept in memory (time) before it is compared to another element and eventually discarded from memory. Figure 6.1 shows an arrangement of n elements in which the $(3k+1)$-st smallest element is placed at the $(k+1)$-st location from the head, the $(3k+2)$-nd at the $(\frac{2n}{3}+k+1)$-st, and $(3k+3)$-rd at the $(\frac{n}{3}+k+1)$-st $(k=0,\ldots,\frac{n}{3}-1)$, it is easy to see that an element will have to be kept in memory for at least $\frac{n}{3}$ steps (ignoring the rewinding distance) before it can be compared to another element (either the one that is immediately smaller or the one that is immediately larger), or $\frac{2n}{3}$ steps to meet its two neighbors. The total *time* $\times$ *space* cost of these $n-1$ comparisons is therefore $\frac{(n-1)\times n}{3}$. On the other hand, at any moment there are at most m elements in internal memory and each of these m memory locations can carry an element for up to n steps in one pass.

In other words, at most $3m$ comparisons can be made in one pass and as a result, $\frac{n-1}{3m}$ passes are necessary to rank n elements.

1	4	7	...	N-2	3	6	9	...	N	2	5	8	...	N-1

Figure 6.1: An example showing *time* $\times$ *space* cost of $\frac{n}{3}$ of every comparison.

Theorem 20. $\frac{n-1}{3m}$ *passes are necessary for any sorting algorithm using* m *storage space to sort* n *data elements stored on tape and output the elements along with their ranks.*

6.3.2 Probabilistic Lower Bound

Next we consider the average number of passes made by any sorting algorithm with m internal memory locations. For the purpose of presentation, imagine the end of the tape is concatenated with the head of the tape to form a circle whose circumference is n. The reading head is always moving clockwise along the circle. Suppose we are to determine the relative rankings of three data elements $a < b < c$ which are the $(3r-2)$-nd, $(3r-1)$-st, and $3r$-th largest elements respectively.

Definition 2. *The directed distance* $\overrightarrow{ab}$ *between two points* a *and* b *on a circle is the length of the clockwise arc going from* a *to* b.

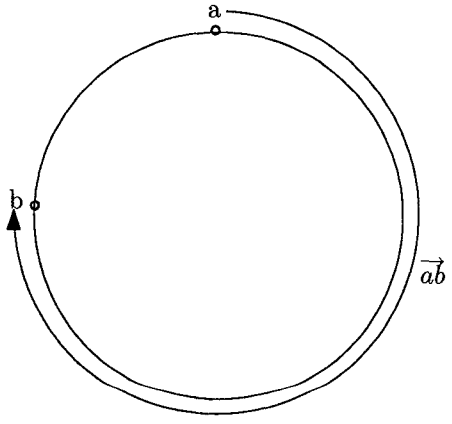

Figure 6.2: Directed distance between two points on a circle.

Definition 3. *The distance $\overline{ab}$ between two points a and b on a circle is the length of the shorter arc that connects these two points.*

$$\overline{ab} = \min\left\{\overrightarrow{ab}, \overrightarrow{ba}\right\}$$

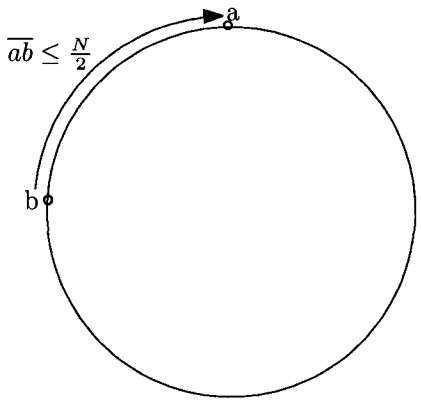

Figure 6.3: Distance between two points on a circle.

Obviously, there are $\frac{n}{3}$ such non-overlapping triples ($r = 1, 2, \ldots, \frac{n}{3}$). To sort these three elements, at least two comparisons will have to be made: one between a and b and the other one between b and c.

First Probabilistic Lower Bound (considering pairs)

Let's consider two data elements with adjacent rank, say a and b. There are $\frac{n}{2}$ such pairs $([1st, 2nd], [3rd, 4th], \ldots, [(n-1)st, nth])$. Any sorting algorithm must compare a and b. To study what is the average *time* $\times$ *space* cost of such a comparison, assume a is located ahead of b by $d \leq n-1$ elements on the tape. To make the comparison between a and b, we have two choices in terms of how to move a and b on the circle in order to make them meet (and therefore make the comparison possible). To minimize the *time* $\times$ *space* cost, depending on whether $d \leq \frac{n}{2}$, we can either carry a forward for d steps or we can carry b forward for $n-d$ steps to meet a. Obviously, the cost is d and $n-d$, respectively. In either case, the minimum cost is at most $n/2$. Given a uniformly random input, meaning the distance between any two random elements on the circle is uniformly distributed on $[0, \frac{n}{2}]$, the average minimum cost of one

comparison is given below[1]:

$$\begin{aligned}\frac{1}{n}\left[\sum_{i=0}^{n-1}\min\{i,n-i\}\right] &= \frac{1}{n}\left[\sum_{i=0}^{\frac{n}{2}} i+\sum_{i=\frac{n}{2}+1}^{n-1}(n-i)\right] \\ &= \frac{1}{n}\left[\left(\frac{n^2}{8}+\frac{n}{4}\right)+\left(\frac{n^2}{8}-\frac{n}{4}\right)\right] \\ &= \frac{n}{4} \\ &= \int_{x=0}^{n/2}\frac{x}{n/2}dx.\end{aligned}$$

In other words, each of the m memory locations can be used to make 4 comparisons in one pass. Since at least $\frac{n}{2}$ comparisons have to be made, one for each pair of neighboring elements, an $\frac{n}{8m}$ lower bound on the average number of passes made by any algorithm follows immediately.

Theorem 21. *(First probabilistic lower bound) Any algorithm with m internal memory locations will make on average at least $\frac{n}{8m}$ passes to sort a random input of n elements.*

Second Probabilistic Lower Bound (considering triples)

Next we consider the average cost of determining the relative order of a triple: $a < b < c$, which takes at minimum two comparisons, one between a and b and the other one between b and c. As shown in Figure 6.4, let x denote the directed distance between a and b, and y the directed distance between a and c. Obviously, to make the comparison between a and b, we can either carry a in the internal memory for x steps to meet b ($a \rightarrow b$), or carry b for $n-x$ steps to meet a ($b \rightarrow a$). Similarly, there are two ways of carrying out the comparison between b and c ($b \rightarrow c$ or $c \rightarrow b$). Next we will study depending on the values of x and y, how to carry out these two comparisons so that the combined cost is minimized.

- Case 1: $\overrightarrow{ab} > \overrightarrow{ac}$ $(x > y)$

[1]Notice the equivalence between integral and summation. In general, Riemann integral theorem (Rudin [1976]) states that for a continuous function on a closed interval, such as the polynomial cost functions we will encounter in this chapter, integral is an approximation of summation for sufficiently large n. To simplify presentation, we will study the problem in continuous model in which integral instead of summation will be used in the rest of this chapter.

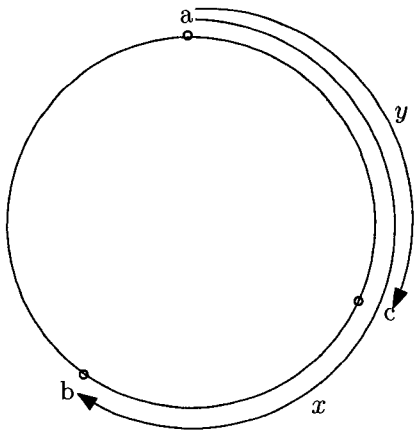

Figure 6.4: Case 1: $x > y$.

Listed below are the 4 possible ways of carrying out comparisons between a and b and between b and c and their respective costs:

(a) $a \rightarrow b, b \rightarrow c$. The combined cost is $x + n - (x - y) = n + y$.

(b) $a \rightarrow b, c \rightarrow b$. The combined cost is $2x - y$.

(c) $b \rightarrow a, b \rightarrow c$ $(b \rightarrow a \rightarrow c)$. The combined cost is $n - x + y$.

(d) $b \rightarrow a, c \rightarrow b$. The combined cost is $n - y$.

Depending on the exact positions of a, b and c, the algorithm chooses the one that has the lowest combined cost.

(a) will never be the most efficient way of determining the relative rankings of a, b and c because there is always an alternative that does exactly the same with lower combined cost: carry b for $n - x$ steps to meet a, then continue for another y steps to meet c $(b \rightarrow a \rightarrow c)$. Obviously, this is the same as (c), whose cost is $n - x + y$ and is less than $n + y$ no matter what values are x and y.

(b) is the most efficient if and only if

$$\begin{cases} 2x - y \leq n - x + y & \text{and} \\ 2x - y \leq n - y, \end{cases}$$

or equivalently,

$$\begin{cases} 0 \le x \le \frac{n}{3} \text{ and } 0 \le y < x & \text{or} \\ \frac{n}{3} \le x \le \frac{n}{2} \text{ and } \frac{3x-n}{2} \le y < x, \end{cases}$$

which defines an area as shown in Figure 6.5 over which the minimum cost is $2x - y$.

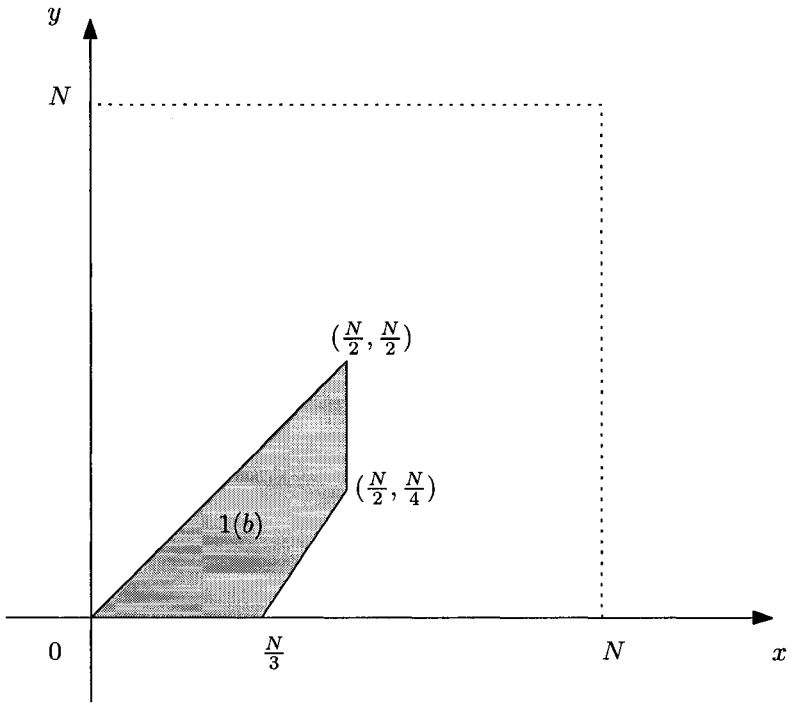

Figure 6.5: $a \to b, c \to b$ is the most efficient over $1(b)$.

The weighted cost of $a \to b, c \to b$ over area $1(b)$ is given by:

$$\begin{aligned} & \int_{x=0}^{\frac{n}{3}} \int_{y=0}^{x} (2x - y)\, d\frac{y}{n} d\frac{x}{n} + \int_{x=\frac{n}{3}}^{\frac{n}{2}} \int_{y=\frac{3x-n}{2}}^{x} (2x - y)\, d\frac{y}{n} d\frac{x}{n} \\ = \; & \frac{1}{n^2} \left[\int_{x=0}^{\frac{n}{3}} \int_{y=0}^{x} d\left(2xy - \frac{y^2}{2}\right) dx + \int_{x=\frac{n}{3}}^{\frac{n}{2}} \int_{y=\frac{3x-n}{2}}^{x} d\left(2xy - \frac{y^2}{2}\right) dx \right] \end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{n^2}\left[\int_{x=0}^{\frac{n}{3}}\left(2x^2-\frac{x^2}{2}\right)dx\right. \\
&\quad \left.+\int_{x=\frac{n}{3}}^{\frac{n}{2}}\left(2x^2-\frac{x^2}{2}-\frac{2x(3x-n)}{2}+\frac{(\frac{3x-n}{2})^2}{2}\right)dx\right] \\
&= \frac{1}{n^2}\left[\int_{x=0}^{\frac{n}{3}}\frac{3x^2}{2}dx+\int_{x=\frac{n}{3}}^{\frac{n}{2}}\left(\frac{3x^2}{2}-3x^2+xn+\frac{9x^2}{8}+\frac{n^2}{8}-\frac{6xn}{8}\right)dx\right] \\
&= \frac{1}{n^2}\left[\int_{x=0}^{\frac{n}{3}}d\frac{x^3}{2}+\int_{x=\frac{n}{3}}^{\frac{n}{2}}\left(-\frac{3x^2}{2}+\frac{9x^2}{8}+\frac{n^2}{8}+\frac{xn}{4}\right)dx\right] \\
&= \frac{1}{n^2}\left[\frac{n^3}{54}+\int_{x=\frac{n}{3}}^{\frac{n}{2}}\left(-\frac{3x^2}{8}+\frac{n^2}{8}+\frac{xn}{4}\right)dx\right] \\
&= \frac{1}{n^2}\left[\frac{n^3}{54}+\int_{x=\frac{n}{3}}^{\frac{n}{2}}d\left(-\frac{x^3}{8}+\frac{n^2x}{8}+\frac{nx^2}{8}\right)\right] \\
&= \frac{1}{n^2}\left[\frac{n^3}{54}-\frac{n^3}{64}+\frac{n^3}{16}+\frac{n^3}{32}+\frac{n^3}{216}-\frac{n^3}{24}-\frac{n^3}{72}\right] \\
&= \frac{1}{n^2}\left[\frac{n^3}{54}+\frac{5n^3}{64}-\frac{2n^3}{216}-\frac{n^3}{24}\right] \\
&= \frac{1}{n^2}\left[\frac{n^3}{108}+\frac{5n^3}{64}-\frac{n^3}{24}\right] \\
&\approx 0.045717n
\end{aligned}
$$

(c) is the most efficient if and only if

$$
\begin{cases}
n-x+y \le 2x-y & \text{and} \\
n-x+y \le n-y, &
\end{cases}
$$

or equivalently,

$$
\begin{cases}
\frac{n}{3} \le x \le \frac{n}{2} \text{ and } 0 \le y \le \frac{3x-n}{2} & \text{or} \\
\frac{n}{2} \le x \le n \text{ and } 0 \le y \le \frac{x}{2} &
\end{cases}
$$

which defines an area as shown in Figure 6.6 over which the minimum cost is $n-x+y$.

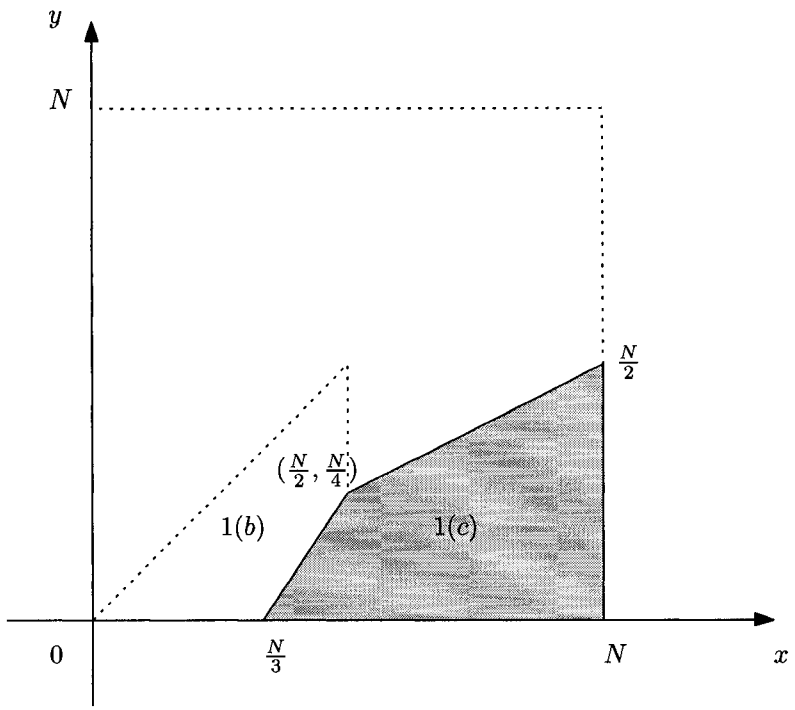

Figure 6.6: $b \longrightarrow a \longrightarrow c$ is the most efficient over $1(c)$.

The wighted cost of $b \longrightarrow a \longrightarrow c$ over the area is:

$$
\begin{aligned}
& \int_{x=\frac{n}{3}}^{\frac{n}{2}} \int_{y=0}^{\frac{3x-n}{2}} (n-x+y)\, d\frac{y}{n} d\frac{x}{n} + \int_{x=\frac{n}{2}}^{n} \int_{y=0}^{\frac{x}{2}} (n-x+y)\, d\frac{y}{n} d\frac{x}{n} \\
= \; & \frac{1}{n^2} \left[\int_{x=\frac{n}{3}}^{\frac{n}{2}} \int_{y=0}^{\frac{3x-n}{2}} d\left(ny - xy + \frac{y^2}{2}\right) dx \right. \\
& \left. + \int_{x=\frac{n}{2}}^{n} \int_{y=0}^{\frac{x}{2}} d\left(ny - xy + \frac{y^2}{2}\right) dx \right] \\
= \; & \frac{1}{n^2} \left[\int_{x=\frac{n}{3}}^{\frac{n}{2}} \left(\frac{n(3x-n)}{2} - \frac{x(3x-n)}{2} + \frac{(\frac{3x-n}{2})^2}{2} \right) dx \right. \\
& \left. + \int_{x=\frac{n}{2}}^{n} \left(\frac{xn}{2} - \frac{x^2}{2} + \frac{x^2}{8} \right) dx \right] \\
= \; & \frac{1}{n^2} \left[\int_{x=\frac{n}{3}}^{\frac{n}{2}} \left(\frac{5xn}{4} - \frac{3x^2}{8} - \frac{3n^2}{8} \right) dx + \int_{x=\frac{n}{2}}^{n} \left(\frac{xn}{2} - \frac{3x^2}{8} \right) dx \right]
\end{aligned}
$$

$$
\begin{aligned}
&= \frac{1}{n^2}\left[\int_{x=\frac{n}{3}}^{\frac{n}{2}} d\left(\frac{5nx^2}{8}-\frac{x^3}{8}-\frac{3n^2x}{8}\right)+\int_{x=\frac{n}{2}}^{n} d\left(\frac{nx^2}{4}-\frac{x^3}{8}\right)\right] \\
&= \frac{1}{n^2}\left[\frac{5n(\frac{n}{2})^2}{8}-\frac{(\frac{n}{2})^3}{8}-\frac{3n^2(\frac{n}{2})}{8}-\frac{5n(\frac{n}{3})^2}{8}+\frac{(\frac{n}{3})^3}{8}+\frac{3n^2(\frac{n}{3})}{8}\right. \\
&\quad \left.+\frac{n^3}{8}-\frac{n^3}{16}+\frac{n^3}{64}\right] \\
&\approx 0.091435n.
\end{aligned}
$$

(d) is the most efficient if and only if

$$
\begin{cases} n-y \le 2x-y & \text{and} \\ n-y \le n-x+y, \end{cases}
$$

or equivalently,

$$
\frac{x}{2} \le y < x \text{ and } \frac{n}{2} \le x \le n,
$$

which defines an area as shown in Figure 6.7 over which the minimum cost is $n-y$.

The weighted cost of $b \to a, c \to b$ over this area is therefore

$$
\begin{aligned}
\int_{x=\frac{n}{2}}^{n}\int_{y=\frac{x}{2}}^{x}(n-y)d\frac{y}{n}d\frac{x}{n} &= \frac{1}{n^2}\left[\int_{x=\frac{n}{2}}^{n}\left[\int_{y=\frac{x}{2}}^{x} d\left(ny-\frac{y^2}{2}\right)\right]dx\right] \\
&= \frac{1}{n^2}\left[\int_{x=\frac{n}{2}}^{n}\left(\frac{nx}{2}-\frac{3x^2}{8}\right)dx\right] \\
&= \frac{5n}{64} = 0.078125n.
\end{aligned}
$$

- Case 2: $\overrightarrow{ab} < \overrightarrow{ac}$ $(x < y)$

  Similarly to Case 1, there are 4 ways of conducting comparisons between a and b and between b and c:

  (a) $a \to b, b \to c$ $(a \to b \to c)$. The combined cost is y.

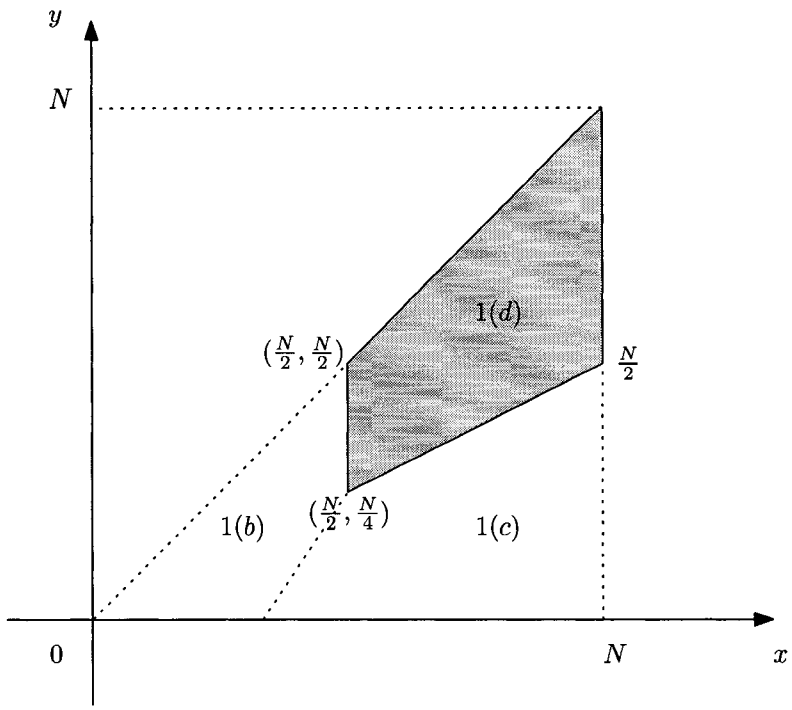

Figure 6.7: $b \rightarrow a, c \rightarrow b$ is the most efficient over $1(d)$.

(b) $a \rightarrow b, c \rightarrow b$. The combined cost is $x + (n - y + x) = n + 2x - y$.

(c) $b \rightarrow a, b \rightarrow c$ $(b \rightarrow c \rightarrow a)$. The combined cost is $n - x$.

(d) $b \rightarrow a, c \rightarrow b$. The combined cost is $(n-x)+(n-y+x) = 2n-y$. Obviously, this will never be the most efficient as $b \rightarrow c \rightarrow a$ is an alternative that does the same with lower cost.

(a) is the most efficient if and only if

$$\begin{cases} y \leq n + 2x - y & \text{and} \\ y \leq n - x, \end{cases}$$

or equivalently,

$$\begin{cases} 0 < x < \frac{n}{4} \text{ and } x < y < \frac{n}{2} + x & \text{or} \\ \frac{n}{4} < x < \frac{n}{2} \text{ and } x < y < n - x, \end{cases}$$

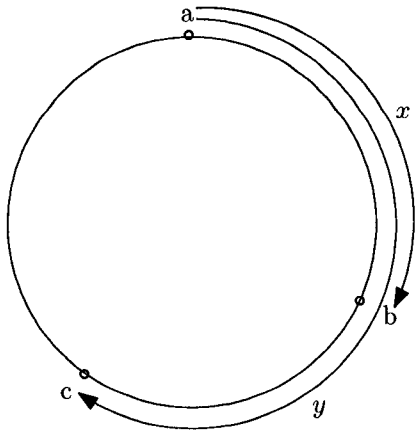

Figure 6.8: Case 2: $x < y$

which defines an area as shown in Figure 6.9 over which the minimum cost is y. The weighted cost of $a \rightarrow b \rightarrow c$ over the area is:

$$
\begin{aligned}
& \int_{x=0}^{\frac{n}{4}} \int_{y=x}^{\frac{n}{2}+x} y d\frac{y}{n} d\frac{x}{n} + \int_{x=\frac{n}{4}}^{\frac{n}{2}} \int_{y=x}^{n-x} y d\frac{y}{n} d\frac{x}{n} \\
= & \frac{1}{n^2}\left[\int_{x=0}^{\frac{n}{4}} \left[\int_{y=x}^{\frac{n+2x}{2}} d\frac{y^2}{2}\right] dx + \int_{x=\frac{n}{4}}^{\frac{n}{2}} \left[\int_{y=x}^{n-x} d\frac{y^2}{2}\right] dx\right] \\
= & \frac{1}{n^2}\left[\int_{x=0}^{\frac{n}{4}} \left(\frac{\left(\frac{n+2x}{2}\right)^2}{2} - \frac{x^2}{2}\right) dx + \int_{x=\frac{n}{4}}^{\frac{n}{2}} \left(\frac{(n-x)^2}{2} - \frac{x^2}{2}\right) dx\right] \\
= & \frac{1}{n^2}\left[\int_{x=0}^{\frac{n}{4}} \left(\frac{n^2}{8} + \frac{nx}{2}\right) dx + \int_{x=\frac{n}{4}}^{\frac{n}{2}} \left(\frac{n^2}{2} - nx\right) dx\right] \\
= & \frac{1}{n^2}\left[\int_{x=0}^{\frac{n}{4}} d\left(\frac{n^2x}{8} + \frac{nx^2}{4}\right) + \int_{x=\frac{n}{4}}^{\frac{n}{2}} d\left(\frac{n^2x}{2} - \frac{nx^2}{2}\right)\right] \\
= & \frac{5n}{64} = 0.078125n.
\end{aligned}
$$

(b) is the most efficient if and only if

$$
\begin{cases} n + 2x - y \leq y & \text{and} \\ n + 2x - y \leq n - x, \end{cases}
$$

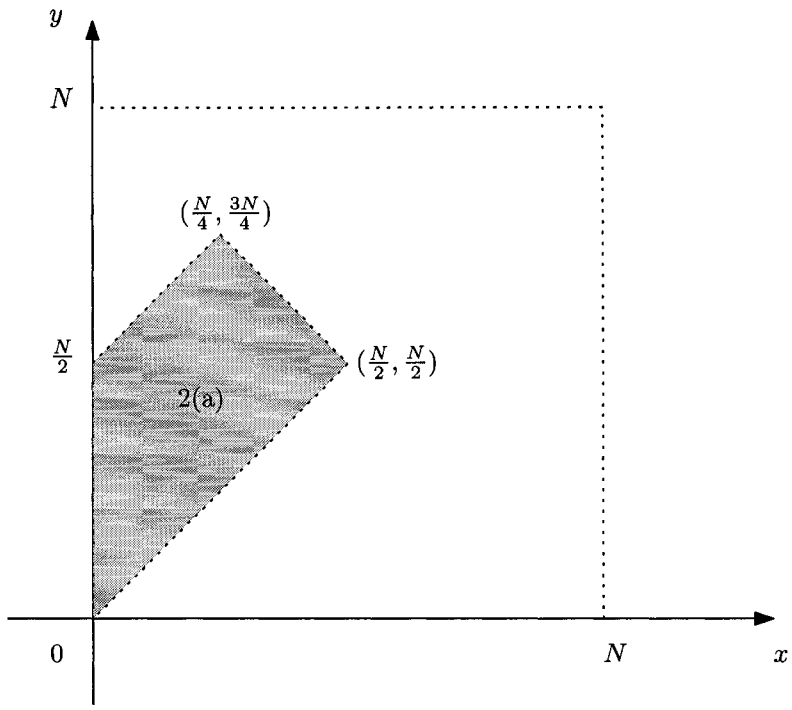

Figure 6.9: $a \rightarrow b \rightarrow c$ is the most efficient over $2(a)$.

or equivalently,

$$\begin{cases} 0 < x < \frac{n}{4} \text{ and } \frac{n}{2} + x < y < n & \text{or} \\ \frac{n}{4} < x < \frac{n}{3} \text{ and } 3x < y < n, \end{cases}$$

which defines an area as shown in Figure 6.10 over which the minimum cost is $n + 2x - y$.

The weighted cost of $a \rightarrow b, c \rightarrow b$ over the area is:

$$\begin{aligned} & \int_{x=0}^{\frac{n}{4}} \int_{y=\frac{n}{2}+x}^{n} (n + 2x - y) d\frac{y}{n} d\frac{x}{n} + \int_{x=\frac{n}{4}}^{\frac{n}{3}} \int_{y=3x}^{n} (n - 2x - y) d\frac{y}{n} d\frac{x}{n} \\ = \; & \frac{1}{n^2} \left[\int_{x=0}^{\frac{n}{4}} \int_{y=\frac{n+2x}{2}}^{n} d\left(ny + 2xy - \frac{y^2}{2} \right) dx \right. \\ & \left. + \int_{x=\frac{n}{4}}^{\frac{n}{3}} \int_{y=3x}^{n} d\left(ny + 2xy - \frac{y^2}{2} \right) dx \right] \end{aligned}$$

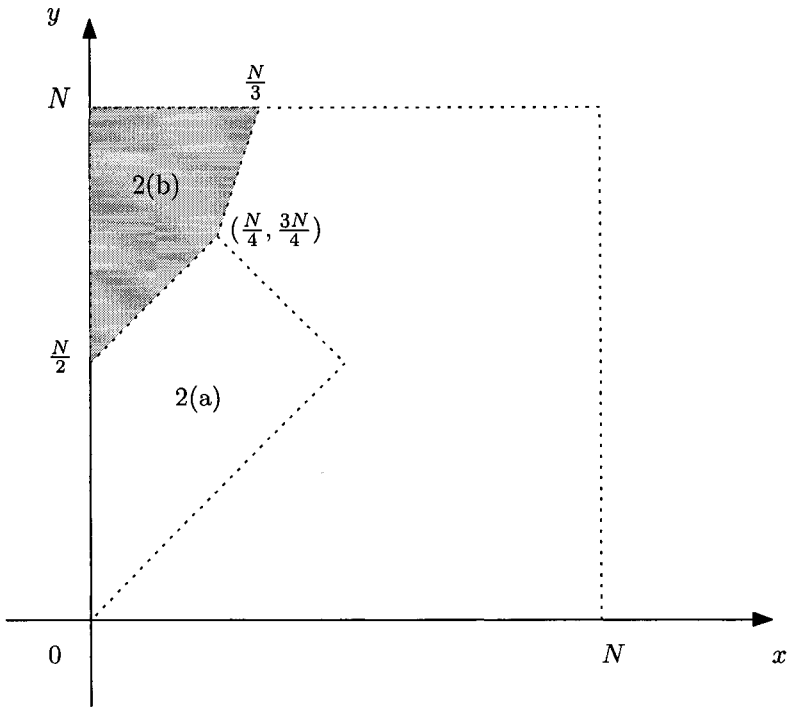

Figure 6.10: $a \to b, c \to b$ is the most efficient over $2(b)$.

$$
\begin{aligned}
&= \frac{1}{n^2}\left[\int_{x=0}^{\frac{n}{4}} \left(n^2 + 2xn - \frac{n^2}{2} - \frac{n(n+2x)}{2} - 2x\left(\frac{n}{2}+x\right) + \frac{(\frac{n+2x}{2})^2}{2}\right) dx \right. \\
&\quad \left. + \int_{x=\frac{n}{4}}^{\frac{n}{3}} \left(n^2 + 2xn - \frac{n^2}{2} - 3xn - 6x^2 + \frac{9x^2}{2}\right) dx\right] \\
&= \frac{1}{n^2}\left[\int_{x=0}^{\frac{n}{4}} d\left(-\frac{x^3}{2} + \frac{n^2x}{8} + \frac{nx^2}{4}\right) + \int_{x=\frac{n}{4}}^{\frac{n}{3}} d\left(\frac{n^2x}{2} - \frac{nx^2}{2} - \frac{x^3}{2}\right)\right] \\
&= \frac{1}{n^2}\left[-\frac{3n^3}{64} + \frac{5n^3}{54}\right] \approx 0.045717n.
\end{aligned}
$$

(c) is the most efficient if and only if $n - x \leq y$ and $n - x \leq n + 2x - y$.

$$
\begin{cases} n - x \leq y & \text{and} \\ n - x \leq n + 2x - y, \end{cases}
$$

or equivalently,

$$\begin{cases} \frac{n}{4} < x < \frac{n}{3} \text{ and } n - x < y < 3x & \text{or} \\ \frac{n}{3} < x < \frac{n}{2} \text{ and } n - x < y < n, & \text{or} \\ \frac{n}{2} < x < n \text{ and } x < y < n, \end{cases}$$

which defines an area as shown in Figure 6.11 over which the minimum cost is $n - x$.

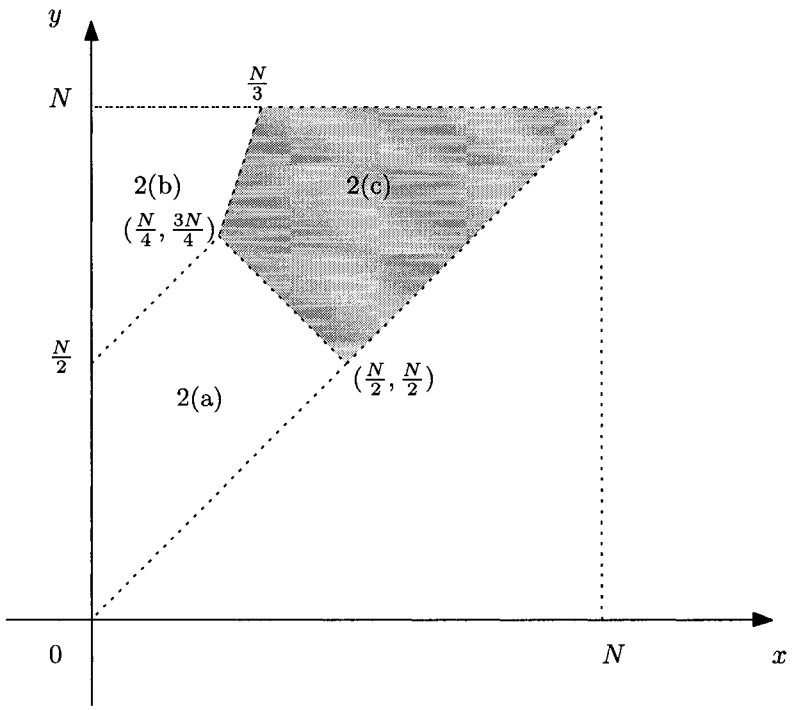

Figure 6.11: $b \rightarrow c \rightarrow a$ is the most efficient over $2(c)$.

The weighted cost of $b \rightarrow c \rightarrow a$ is:

$$
\begin{aligned}
&\int_{x=\frac{n}{4}}^{\frac{n}{3}}\int_{y=n-x}^{3x}(n-x)d\frac{y}{n}d\frac{x}{n}+\int_{x=\frac{n}{3}}^{\frac{n}{2}}\int_{y=n-x}^{n}(n-x)d\frac{y}{n}d\frac{x}{n}\\
&+\int_{x=\frac{n}{2}}^{n}\int_{y=x}^{n}(n-x)d\frac{y}{n}d\frac{x}{n}\\
=\;&\frac{1}{n^2}\left[\int_{x=\frac{n}{4}}^{\frac{n}{3}}\int_{y=n-x}^{3x}d\left(ny-xy\right)dx+\int_{x=\frac{n}{3}}^{\frac{n}{2}}\int_{y=n-x}^{n}d\left(ny-xy\right)dx\right.\\
&\left.+\int_{x=\frac{n}{2}}^{n}\int_{y=x}^{n}d\left(ny-xy\right)dx\right]\\
=\;&\frac{1}{n^2}\left[\int_{x=\frac{n}{4}}^{\frac{n}{3}}\left[3nx-3x^2-n(n-x)+x(n-x)\right]dx\right.\\
&+\int_{x=\frac{n}{3}}^{\frac{n}{2}}\left[n^2-xn-n(n-x)+x(n-x)\right]dx\\
&\left.+\int_{x=\frac{n}{2}}^{n}\left(n^2-xn-nx+x^2\right)dx\right]\\
=\;&\frac{1}{n^2}\left[\int_{x=\frac{n}{4}}^{\frac{n}{3}}d\left(-n^2x+\frac{5nx^2}{2}-\frac{4x^3}{3}\right)+\int_{x=\frac{n}{3}}^{\frac{n}{2}}d\left(\frac{nx^2}{2}-\frac{x^3}{3}\right)\right.\\
&\left.+\int_{x=\frac{n}{2}}^{n}d\left(n^2x-nx^2+\frac{x^3}{3}\right)\right]\\
=\;&\frac{1}{n^2}\left[\frac{5n^3}{27}-\frac{3n^3}{32}\right]\approx 0.091435n.
\end{aligned}
$$

As already mentioned, (d) will never be the most efficient because of the alternative $b \longrightarrow c \longrightarrow a$.

Therefore, the overall average combined *times* $\times$ *space* cost of determining the relative order within one triple is approximately

$$
\begin{aligned}
&0.045717n+0.091435n+0.078125n+0.078125n+0.045717n+0.091435n\\
=\;&0.430554n.
\end{aligned}
$$

As there are $\frac{n}{3}$ triples, we immediately have the following theorem in the continuous model.

Theorem 22. *(For sufficiently large n) The expected number of passes needed to sort n elements stored on a tape using an internal memory of m locations is at least*

$$\frac{\frac{n}{3} \times 0.430554n}{nm} \approx \frac{0.143518n}{m} \approx \frac{n}{6.967767m}.$$

6.4 Summary

Sorting is one of the most studied problem in computer science. Yet the problem of sorting data stored on tape in the data stream model received little attention since the publication of Munro and Paterson's paper in 1980. There is an obvious time space trade-off associated with the problem, *i.e.*, between the number of passes over the data and the size of the internal memory. We extend the results in the original study of the problem by Munro and Paterson [1980] and show that there is a matching lower bound on the number of passes any algorithm would have to make given a fixed amount of internal memory. We also give a probabilistic analysis of the problem which implies that even the optimal algorithm will have to make on average at least a constant times ($> \frac{1}{7}$) the number of passes the naive algorithm would make.

Chapter 7

Conclusion

Although the prevalence of data stream algorithms has been a quite recent phenomenon, the idea of making one or a few passes over the data for performing computations can be traced back to the early days of Automata theory (Knuth [1998]). In a more recent development, the algorithms that utilize only one or few passes for selection and sorting caught a few researchers' attention in the early 80's (Munro and Paterson [1980], Borodin and Cook [1982]), but the area seemed to have largely been in hibernation ever since.

All has changed in the past few years as an increasing number of applications have emerged that deal with massive data sets, such as Internet traffic analysis and data mining of large databases. This has led to a surge of interest in new computational models that reflect the real challenges posed by large data sets. It is important because only within these new computational models can an algorithm's actual performance be analyzed. This can then lead to the design of efficient algorithms or, on the other side of the coin, hardness results showing that a problem cannot be solved efficiently within the limitations of the existing computing architecture.

One such model is the *Data Stream Model* in which a machine with small memory is allowed to make only a few linear scans (*passes*) over the input data. This reflects the two most important challenges imposed by massive data sets: First, in the applications such as networking, the data are generated continuously at such high speed that it is impossible to store the complete data set, and the only way of accessing the data is by essentially a single pass over the data. Other applications where the data stream model fits well include large database mining. In these applications, although the complete data set can be stored and accessed as many times as needed, the amount of data is so huge that they cannot fit in main memory but only on secondary storage devices such as hard drives or tapes. As a result, the assumption of the (fast)

constant time cost of random access to any data is no longer valid. Instead, the most efficient way of accessing data in large databases is through sequential access, that is, making a few passes over the data set.

Due to their constrained computational power, most data stream algorithms can only compute efficiently approximate answers, but for a large number of practical problems, exact answers are not required and approximate answers are quite acceptable as long as the benefits they enable (efficiency in computation or computability) outweigh the cost associated with the inaccurate answers. Therefore, not surprisingly, much of the research in data streams has been focused on different trade-offs between (deterministic or probabilistic) accuracy, computing time and storage space. In this thesis, we studied several problems in the data stream model and various trade-offs in their solutions.

7.1 Summary of Results

In Chapter 3, we showed a near optimal trade-off between storage space and accuracy in frequency estimation. We also showed that no randomized algorithm can do much better than deterministic algorithm in terms of accuracy given the same amount of memory. In fact, we proved that any randomized algorithm can at best be about 6 times as accurate as the deterministic algorithm. Then we analyzed the trade-offs between all three elements for the problems of approximate range mode and range median queries, namely, space, time, and accuracy, and we proposed the first non-trivial approximate algorithms for computing range mode and range median queries on lists in Chapter 4 and higher dimensional matrices in Chapter 5. In Chapter 6, we studied the problem of sorting large data sets with limited internal memory, we closed the gap in one of the pioneering works in data stream algorithms by Munro and Paterson [1980], in which there was a gap of a multiplicative factor of 4 between the lower bound and the upper bound on the number of passes made by a data stream algorithm. We also derived the first probabilistic lower bound for the problem.

7.2 Open Problems

Although the deterministic frequency estimation problem has been (almost) solved by a near optimal algorithm, which achieves close to optimum accuracy given a fixed amount of memory. There is still a gap between that and the randomized lower bound ($\frac{(1-\alpha)n}{m}$ versus $\frac{0.17157(1-\alpha)n}{m}$). The question remains whether there is a randomized algorithm that closes this gap.

For the approximate range mode and range median queries on lists, our algorithms are almost optimal in terms of space usage and query time ($O(\frac{n}{1-\alpha})$ space and $O(\log_2 \log n)$ time for range mode queries, $O(\frac{n}{1-\alpha})$ space and $O(1)$ time for range median queries). However, significant gaps still exist between our solutions and the lower bounds in higher dimensional space. In the case of approximate range mode queries, we have managed to achieve $O(n^3 \log_{\frac{1}{\alpha}} n)$ storage space and $O(\log_2 \log_{\frac{1}{\alpha}} n)$ query time on an $n \times n$ matrix in two dimensional space. In the case of approximate range median queries, the gaps are narrower, our data structure uses $O\left(\frac{8^d n^d}{(1-\alpha^{1/d})^d}\right)$ storage space versus the lower bound of $O(n^d)$, and the query time is almost optimal at $O(d)$ on a matrix in d-dimensional space.

The same ideas are expected to apply to three or higher dimensional spaces, in which *hyper-surfaces* instead of points (as in arrays) or curves (as in two dimensional space) can be used to specify range modes of the same frequency. However, it remains to see how to actually achieve better space usage for approximate range mode queries over the naive solution of precomputing and storing a mode for each range, that is to store just a small number (compared to the number of all possible query ranges) of hyper-surfaces each consisting of all the modes of the same frequency that is significantly (a factor of $1/\alpha$ to be exact) higher than those on the preceding hyper-surface. It is all possible because of the monotonic nature of these surfaces as well as the increasing "distance" between adjacent surfaces as they are further away from the origin. The crux of the problem is the succinct (approximate) representations of these hyper-surfaces.

Although we have closed the gap in the classic sorting-on-tape model between lower bound and upper bound for deterministic sorting algorithms with limited internal memory, there is still a gap between the trivial upper bound and our lower bound

($\frac{n}{m}$ versus $\frac{n}{3m}$). It remains an open problem to find a non-trivial probabilistic upper bound that is close to our lower bound of $\frac{n}{6.967767m}$.

Bibliography

P. K. Agarwal, S. Krishnan, N. H. Mustafa, and S. Venkatasubramanian. Streaming geometric optimization using graphics hardware. In *Proceedings of the 11th European Symposium on Algorithms (ESA 2003)*, pages 544–555, 2003.

R. Agarwal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, 1993.

R. Agarwal and A. Swami. A one-pass space-efficient algorithm for finding quantiles. In *Proceedings of the 7th International Conference on Management of Data (COMAD-95)*, pages 1–12, 1995.

N. Alon, Y. Matias, , and M. Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the 28th ACM Symposium on the Theory of Computing (STOCS 1996)*, pages 20–29, 1996.

N. Alon and B. Schieber. Optimal preprocessing for answering on-line product queries. Technical Report 71/87, Tel-Aviv University, 1987.

K. Alsabti, S. Ranka, , and V. Singh. A one-pass algorithm for accurately estimating quantiles for disk-resident data. In *Proceedings of the 23rd International Conference on Very Large Databases (VLDB 1997)*, pages 346–355, 1997.

A. Arasu and G. S. Manku. Approximate counts and quantiles over sliding windows. In *Proceedings of the 23rd ACM Symposium on Principles of Database Systems (PODS 2004)*, pages 286–296, 2004.

Z. Bar-Yossef, T. S. Jayram, R. Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. In *Proceedings of 43rd IEEE Symposium on Foundations of Computer Science (FOCS 2002)*, pages 209–218, 2002.

S. Battiato, D. Cantone, D. Catalano, G. Cincotti, , and M. Hofri. An efficient algorithm for the approximate median selection problem. In *Proceedings of the 4th Italian Conference on Algorithms and Complexity,*, pages 226–238, 2000.

R. Bayer. Binary b-trees for virtual memory. In *Proceedings of the 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control*, pages 219–235, 1971.

M. Ben-Or. Lower bounds for algebraic computation trees. In *Proceedings of the 15th annual ACM Symposium on the Theory of Computing (STOC 1983)*, pages 80–86, 1983.

L. Bhuvanagiri, S. Ganguly, D. Kesh, and C. Saha. Simpler algorithm for estimating frequency moments of data streams. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithm (SODA 2006)*, pages 708–713, 2006.

M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, , and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

A. Borodin and S. Cook. A time-space tradeoff for sorting on a general sequential model of computation. *SIAM Journal on Computing*, 11(2):287–297, 1982.

P. Bose, E. Kranakis, P. Morin, and Y. Tang. Bounds for frequency estimation of packet streams. In *Proceedings of the 10th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2003)*, pages 33–42, 2003.

P. Bose, E. Kranakis, P. Morin, and Y. Tang. Approximate range mode and range median queries. In *Proceedings of the 22nd International Symposium on Theoretical Aspects of Computer Science (STACS 2005)*, pages 377–388, 2005.

N. Brownlee, C. Mills, and G. Ruth. Traffic flow measurement: Architecture. RFC2722, October 1999.

A. Chakrabarti, S. Khot, and X. Sun. Near-optimal lower bounds on the multi-party communication complexity of set disjointness. In *Proceedings of the 18th Annual IEEE Conference on Computational Complexity*, pages 101–117, 2003.

T. M. Chan and E. Y. Chen. Multi-pass geometric algorithms. In *Proceedings of the 21st Annual Symposium on Computational Geometry*, pages 181–189, 2005.

M. Charikar, K. Chan, and M. Farach-Colton. Finding frequent items in data streams. In *Proceedings of the 29th International Colloquium on Automata, Language, and Programming (ICALP 2002)*, pages 693–703, 2002.

D. Coppersmith and R. Kumar. An improved data stream algorithm for frequency moments. In *Proceedings of the 15th ACM Symposium on Discrete Algorithms (SODA 2004)*, pages 151–156, 2004.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* The MIT Press, 2001. ISBN 0262531968.

G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. In *Proceedings of the 6th Latin American Theoretical Informatics (LATIN 2004)*, pages 29–38, 2004.

M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing*, 31(6):1794–1813, 2002.

J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

E. D. Demaine, A. L'opez-Ortiz, and J. I. Munro. Frequency estimation of internet packet streams with limited space. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA 2002)*, pages 348–360, 2002.

D. Dor and U. Zwick. Selecting the median. In *Proceedings of the 6th ACM-SIAM Symposium on Discrete Algorithms (SODA 1995)*, pages 28–37, 1995.

D. Dor and U. Zwick. Median selection requires $(2+\epsilon)n$ comparisons. *SIAM Journal on Discrete Mathematics*, 14(3):312–325, 2001.

P. Drineas and R. Kannan. Pass efficient algorithms for approximating large matrices. In *Proceedings of the 14th ACM-SIAM Syposium on Discrete Algorithms (SODA 2003)*, pages 223–232, 2003.

J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *Proceedings of the 2002 ACM SIGCOMM conference*, pages 323–336, 2002.

M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *Proceedings of the 24th International Conference on Very Large Databases (VLDB 1998)*, pages 299–310, 1998.

W. Fang and L. Peterson. Inter-as traffic patterns and their implications. In *Proceedings of the 1999 IEEE Global Telecommunications Conference (GLOBECOM 1999)*, pages 1859–1868, 1999.

J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004)*, pages 207–216, 2004.

J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. Graph distances in the streaming model: the value of space. In *Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms (SODA 2005)*, pages 745–754, 2005.

A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True. Deriving traffic demands for operational ip networks: Methodology and experience. *IEEE/ACM Transactions on Networking*, 9(3):265–279, 2001.

W. Feller. *Introduction to Probability Theory and its Applications, 3rd edition.* John Wiley and Sons, 1968.

M. J. Fischer and S. L. Salzberg. Finding a majority among n votes: Solution to problem 81-5 (Journal of Algorithms, june 1981). *Journal of Algorithms*, 3(4): 362–380, 1982.

P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.

S. Ganguly. Estimating frequency moments of data streams using random linear combinations. In *Proceedings of APPROX-RANDOM*, pages 369–380, 2004.

P. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 331–342, 1998.

P. Gibbons and Y. Matias. Synopsis data structures for massive data sets. In *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms (SODA 1999)*, pages 909–910, 1999.

N. K. Govindaraju, N. Raghuvanshi, and D. Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 611–622, 2005.

P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *Proceedings of the 21st International Conference on Very Large Databases (VLDB 1995)*, pages 311–322, 1995.

J. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, Fourth Edition.* Morgan Kaufmann Publishers, 2007.

M. Henzinger. Algorithmic challenges in web search engines. *Internet Math*, 1(1): 115–123, 2003.

M. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. *DIMACS series in Discrete Mathematics and Theoretical Computer Science*, 50:107–118, 1999.

G. Iannaccone, C. Diot, I. Graham, and N. McKeown. Monitoring very high speed links. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 267–271, 2001.

P. Indyk and D. Woodruff. Optimal approximations of the frequency moments of data streams. In *Proceedings of the 37th annual ACM Symposium on Theory of Computing (STOC 2005)*, pages 202–208, 2005.

R. Jain and I. Chlamtac. The p^2 algorithm for dynamic calculation of quantiles and histograms without storing obervations. *Communications of the ACM*, 28(10): 1076–1085, 1985.

B. Kalyanasundaram and G. Schnitger. The probabilistic communication complexity of set intersection. In *Proceedings of the 2nd Structure in Complexity Theory Conference*, pages 41–49, 1987.

R. Karp, C. Papadimitriou, and S. Shenker. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems*, 28(1): 51–55, 2003.

D. G. Kirkpatrick. Optimum search in planar subdivisions. *SIAM J. Comput.*, 12(1): 28–35, 1983.

D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*, volume 3. Addison Wesley, 1998.

D. Krizanc, P. Morin, and M. Smid. Range mode and range median queries on lists and trees. In *Proceedings of the 14th Annual International Symposium on Algorithms and Computation (ISAAC 2003)*, pages 517–526, 2003.

E. Kushilevitz and N. Nisan. *Communication Complexity.* Cambridge University Press, 1997.

X. Lin, H. Liu, J. Xu, and J. X. Yu. Continuously maintaining quantile summaries of the most recent n elements over a data stream. In *Proceedings of the 20th International Conference on Data Engineering*, pages 362–373, 2004.

J. K. Mackie-Mason and H. R. Varian. Pricing the internet. In *Public Access to the Internet, JFK School of Government, May 26–27, 1993*, pages 269–314, 1993.

G. S. Manku and R. Motwani. Aproximate frequency counts over data stream. In *Proceedings of the 13rd International Conference on Very Large Data Bases (VLDB 2002)*, pages 346–357, 2002.

G. S. Manku, S. Rajagopalan, and B. Lindsay. Approximate median and other quantiles in one pass and with limited memory. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 426–435, 1998.

R. Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21:840–842, 1978.

R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. *Theoretical computer science*, 12:315–323, 1980.

A. M. Odlyzko. Internet pricing and the history of communications. *Computer Networks*, 36(5–6):493–517, 2001.

J. Pagter and T. Rauhe. Optimal time-space trade-offs for sorting. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science (FOCS 1998)*, pages 264–268, 1998.

J. S. Park, M. S. Chen, and P. S. Yu. An effective hash based lagorithm for mining association rules. In *Proceedings of 1995 ACM SIGMOD International Conference on Management of Data*, pages 175–186, 1995.

M. Paterson. Progress in selection. In *Proceedings of the 1997 Scandinavian Workshop on Algorithm Theory*, pages 368–379, 1997.

I. Pohl. A minimum storage algorithm for computing the median. Technical Report RC 2701 (# 12713), IBM T J Watson Center, 1969.

W. Rudin. *Principles of Mathematical Analysis, 3rd Edition.* McGraw-Hill Publishing Co., 1976. ISBN 0070856133.

J. Sun and P. Varaiya. Pricing network services. In *Proceedings of the 2003 IEEE INFOCOMM*, pages 1221–1230, 2003.

S. Suri, C. D. T'oth, and Y. Zhou. Range counting over multidimensional data streams. In *Proceedings of the 20th Annual Symposium on Computational Geometry*, pages 160–169, 2004.

S. Venkataraman, D. Song, P. Gibbons, and A. Blum. New streaming algorithms for fast detection of superspreader. In *Proceedings of Network and Distributed Systems Security Symposium*, pages 149–166, 2005.

J. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):208–271, 2001.

A. C. Yao. Probabilistic computations: Towards a unified measure of complexity. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS 1977)*, pages 222–227, 1977.

A. C. Yao. Some complexity questions related to distributed computing. In *Proceedings of the 11th annual ACM Symposium on the Theory of Computing (STOC 1979)*, pages 209–213, 1979.

A. C. Yao. Space-time tradeoff for answering range queries. In *Proceedings of the 14th annual ACM Symposium on the Theory of Computing (STOC 1982)*, pages 128–136, 1982.

F. F. Yao. On lower bounds for selection problems. Technical Report TR-121, Massachusetts Institute of Technology, 1974.

Y. Zhang, L. Breslau, V. Paxson, and S. Shenker. On the characteristics and origins of internet flow rates. In *Proceedings of the 2002 ACM SIGCOMM*, pages 309–322, 2002.