Algorithms and Data Structures for Massive Datasets

Dzejla Medjedovic
Emin Tahirovic
Illustrated by Ines Dedovic

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Algorithms and Data Structures for Massive Datasets
Version 1

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP edition of *Algorithms and Data Structures for Massive Datasets*.

The unprecedented growth of data in recent years is putting the spotlight on the data structures and algorithms that can efficiently handle large datasets. In this book, we present you with a basic suite of data structures and algorithms designed to index, query, and analyze massive data.

What prompted us to write this book is that many of the novel data structures and algorithms that run underneath Google, Facebook, Dropbox and many others, are making their way into the mainstream algorithms curricula very slowly. Often the main resources on this subject are research papers filled with sophisticated and enlightening theory, but with little instruction on how to configure the data structures in a practical setting, or when to use them.

Our goal was to present these exciting and cutting-edge topics in one place, in a practical and friendly tone. Mathematical intuition is important for understanding the subject, and we try to cultivate it without including a single proof. Plentiful illustrations are used to illuminate some of the more challenging material.

Large datasets arise in a variety of disciplines, from bioinformatics and finance, to sensor data and social networks, and our use cases are designed to reflect that.

Every good story needs a conflict, and the main one in this book are the tradeoffs arising from the constraints imposed by large data --- a major theme is sacrificing the accuracy of a data structure to gain savings in space. Finding that sweet spot for a particular application will be our holy grail.

As a reader of this book, we assume you already have a fairly good command of the Big-Oh analysis, fundamental data structures, basic searching and sorting algorithms and basic probability concepts. At different points of the book, however, we offer quick knowledge refreshers, so don't be afraid to jump in.

Lastly, our humble expectation is that you will absolutely love the book and will talk about it at cocktail parties for years to come. Thank you for being our MEAP reader, we welcome and appreciate any feedback that you post in the liveBook Discussion forum and that might improve the book as we are still writing it.

—Dzejla Medjedovic, Emin Tahirovic, and Ines Dedovic

brief contents

1 Introduction

This chapter covers:

- What this book is about and its structure
- What makes this book different than other books on algorithms
- How massive datasets affect the algorithm and data structure design
- How this book can help you design practical algorithms at a workplace
- Fundamentals of computer and system architecture that make massive data challenging for today's systems

Having picked up this book, you might be wondering what the algorithms and data structures for *massive datasets* are, and what makes them different than "normal" algorithms you might have encountered thus far? Does the title of this book imply that the classical algorithms (e.g., binary search, merge sort, quicksort, fundamental graph algorithms) as well as canonical data structures (e.g., arrays, matrices, hash tables, binary search trees, heaps) were built exclusively for small datasets, and if so, why the hell no one told you that.

The answer to this question is not that short and simple (but if it had to be short and simple, it would be "Yes".) The notion of what constitutes a massive dataset is relative and it depends on many factors, but the fact of the matter is that most bread-and-butter algorithms and data structures that we know about and work with on a daily basis (such) have been developed with an implicit assumption that all data fits in the main memory, or *random-access memory* (RAM) of one's computer. So once you read all your data into RAM and stored it into the data structure, it is relatively fast and easy to access any element of it, at which point, the ultimate goal from the efficiency point of view becomes to crunch the most productivity into the fewest number of CPU cycles. This is what the Big-Oh Analysis ($O(.)$) teaches us about --- it commonly expresses the worst-case number of basic operations the algorithm has to perform in order to solve a problem. These unit operations can be

comparisons, arithmetic, bit operations, memory cell read/write/copy, or anything that directly translates into a small number of CPU cycles.

However, if you are a data scientist today, a developer or a back-end engineer working for a company that regularly collects data from its users, whether it be a retail website, a bank, a social network, or a smart-bed app collecting sensor data, storing all data into the main memory of your computer probably sounds like a beautiful dream. And you don't have to work for Facebook or Google to deal with gigabytes (GB), terabytes (TB) or even petabytes (PB) of data almost on a daily basis. According to some projections, from 2020 onward, the amount of data generated will be at least equal to every person on Earth generating close to 2 megabytes (MB) per second![1] Companies with a more sophisticated infrastructure and more resources can afford to delay thinking about scalability issues by spending more money on the infrastructure (e.g., by buying more RAM), but as we will see, even those companies, or should we say, especially those companies, choose to fill that extra RAM with clever and space-efficient data structures.

The first paper[2] to introduce external-memory algorithms --- a class of algorithms that today govern the design of large databases, and whose goal is to minimize the total number of memory transfers during the execution of the program --- appeared back in 1988, where, as the motivation, the authors cite the example of large banks having to sort 2 million checks daily, about 800MB worth of checks to be sorted overnight before the next business day. And for the working memories of that time being about the size of 2-4MB, this indeed was a massive dataset. Figuring out how to sort checks efficiently where we can at one time fit in the working memory (and thus sort) only about 4MB worth of checks, how to swap pieces of data in and out in a way to minimize the number of trips the data makes from disk and into main memory, was a relevant problem back then, and since then it has only become more relevant. In past decades, data has grown tremendously but perhaps more importantly, it has grown at a much faster rate than the average size of RAM memory.

One of the central consequences of the rapid growth of data, and the main idea motivating algorithms in this book, is that most applications today are *data-intensive*. Data-intensive (in contrast to CPU-intensive) means that the bottleneck of the application comes from transferring data back and forth and accessing data, rather than doing computation on that data (in Section 1.4 of this chapter, there are more details as to why data access is much slower than the computation.) But what this practically means is it will require more time to get data to the point where it is available for us to solve the problem on it, than to actually solve the problem; thus any improvement in managing the size of data or improving data access patterns are some of the most effective ways to speed up the application.

In addition, the infrastructure of modern-day systems has become very complex, with thousands of computers exchanging data over network, databases and caches are distributed, and many users simultaneously add and query large amounts of content. Data itself has become complex, multidimensional, and dynamic. The applications, in order to be effective, need to respond to changes very quickly. In streaming applications[3], data effectively flies by without ever being stored, and the application needs to the capture the

[1] Domo.com, "Data Never sleeps,"[Online]. Available: https://www.domo.com/solution/data-never-sleeps-6. [Accessed 19th January, 2020].
[2] A. Aggarwal and S. Vitter Jeffrey, "The input/output complexity of sorting and related problems," J Commun. ACM, vol. 31, no. 9, pp. 1116-1127, 1988.
[3] B. Ellis, Real-Time Analytics: Techniques to Analyze and Visualize Streaming Data, Wiley Publishing, 2014.

relevant features of the data with the degree of accuracy rendering it relevant and useful, without the ability to scan it again. This new context calls for a new generation of algorithms and data structures, a new application builder's toolbox that is optimized to address many challenges of massive-data systems. The intention of this book is to teach you exactly that ---- the fundamental algorithmic techniques and data structures for developing scalable applications.

1.1 An example

To illustrate the main themes of this book, consider the following example: you are working for a media company on a project related to news article comments. You are given a large repository of comments with the following associated basic metadata information:

```
{
    comment-id: 2833908
    article-id: 779284
    user-id: johngreen19
    text: this recipe needs more butter
    views: 14375
    likes:  43
}
```

We are looking at approximately 100 million news articles, and roughly 3 billion user comments. Assuming storing one comment takes 200 bytes, we need about 600GB to store the comment data. Your goal is serving your readers better and in order to do that, you would like to classify the articles according to keywords that recur in the comments below the articles. You are given a list of relevant keywords for each topic (e.g., 'sports', 'politics', etc.), and for initial analysis, the goal is only to count how often the given keywords occurs in comments related to a particular article, but before all that, you would like to eliminate the duplicate comments that occurred during multiple instances of crawling.

1.1.1 An example: how to solve it

One typical way to process the dataset to solve the tasks above is to create some type of a key-value dictionary, for example `set`, `map`, or `unordered_map` in C++, `HashMap` in Java, or `dict` in Python, etc. Key-value dictionaries are implemented either with a balanced binary tree (such as a red-black tree in C++ `map`), or, for faster insert and retrieval operations, a hash table (such as `dict` in Python or `unordered_map` in C++.) For simplicity, we will assume for the rest of this example that we are working with hash tables.

Using `comment-id` as the key, and the number of occurrences of that `comment-id` in the dataset as the value will help us eliminate duplicate comments. We will call this `(comment-id -> frequency)` hash table (see Figure 1.1). Then, for each keyword of interest, we build a separate hash table that counts the number of occurrences of the given keyword from all the comments grouped by `article-id`, so these hash tables have the format `(article-id -> keyword_frequency)`.

But to build the basic `(comment-id -> frequency)` hash table for 3 billion comments, if we use 8 bytes to store each `<comment-id, frequency>` pair (4 bytes for `comment-id` and 4

bytes for `frequency`), we might need up to 24GB for the hash table data. Also, our keyword hash tables can become very large, containing dozens of millions of entries: one such hash table of 10 million entries, where each entry takes 8 bytes, will need around 80MB for data, and maintaining such hash tables for say, top 1000 keywords, can cost up to 80GBs only for data. We emphasize the "only for data" part because a hash table, depending on how it is implemented, will need extra space, either for the empty slots, or for pointers.

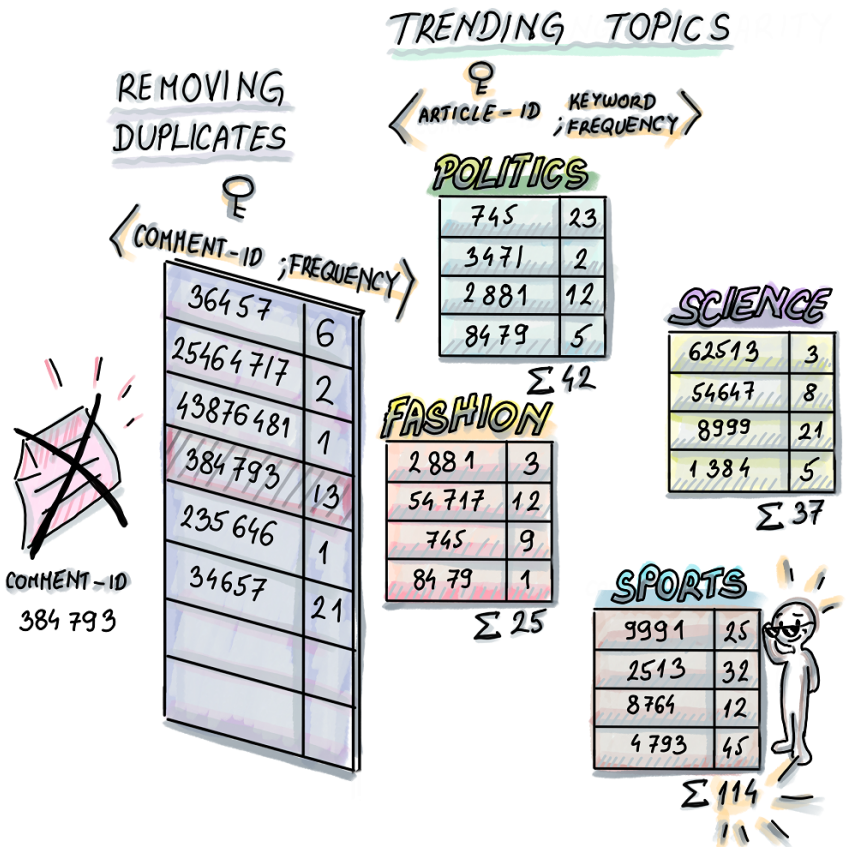

Figure 1.1: In this example, we build a (comment-id, frequency) hash table to help us eliminate duplicate comments. So for example, the comment identified by comment-id 36457 occurs 6 times in the dataset. We also build "keyword" hash tables, where, for each keyword of interest, we count how many times the keyword is mentioned in the comments of a particular article. So for example, the word 'science' is mentioned 21 times in the comments of the article identified by article-id 8999. For a large dataset of 3 billion comments, storing all these data structures can easily lead to needing dozens to a hundred of gigabytes of RAM memory.

We can build similar structures for analyzing the popularity of particular comments, users, etc. Therefore, we might need close to a hundred or hundreds of gigabytes just to build basic

structures that do not even include most of the metadata information that we often need for a more insightful analysis (see Figure 1.2).

Figure 1.2: Hash tables only use the amount of space linear in the size of data, asymptotically the minimum possible required to store data correctly, but with large dataset sizes, hash tables cannot fit into the main memory.

1.1.2 An example: how to solve it, take two

With the daunting dataset sizes, there is a number of choices we are faced with.

It turns out that, if we settle for a small margin of error, we can build a data structure similar to a hash table in functionality, only more compact. There is a family of *succinct data structures,* data structures that use less than the lower theoretical limit to store data that can answer common questions such as those relating to:

- **Membership** --- Does comment/user `X` exist?
- **Frequency** --- How many times did the user `X` comment? What is the most popular keyword?
- **Cardinality** --- How many truly distinct comments/users do we have?

These data structures use much less space to process a dataset of n items than the linear space $O(n)$) that a hash table or a red-black tree would need, think 1 byte per item or sometimes much less than that.

We can solve our news article comment examples with succinct data structures. A Bloom filter (Chapter 3) will use 8x less space than the `(comment-id -> frequency)` hash table and can help us answer membership queries with about 2% false positive rate. In this introductory chapter, we will try to avoid doing the math to explain how we arrived at these numbers, but for now it suffices to say that the reason why Bloom filter, and some other

data structures that we will see can get away with substantially less space than hash tables or red-black trees is that they do not actually store the items themselves. They compute certain codes (hashes) that end up representing the original items (but also some other items, hence the false positives), and original items are much larger than the codes. Another data structure, Count-Min sketch (Chapter 4) will use about 24x less space than `(comment-id -> frequency)` hash table to estimate the frequency of each `comment-id`, exhibiting a small overestimate in the frequency in over 99% of the cases. We can use the same data structure to replace the `(article-id -> keyword_frequency)` hash tables and use about 3MB per keyword table, costing about 20x less than the original scheme. Lastly, a data structure HyperLogLog (Chapter 5) can estimate the cardinality of the set with only 12KB, exhibiting the error less than 1%. If we further relax the requirements on accuracy for each of these data structures, we can get away with even less space. Because the original dataset still resides on disk, while the data structures are small enough to serve requests efficiently from RAM, there is also a way to control for an occasional error.

Another choice we have when dealing with large data is to proclaim the set unnecessarily large and only work with a random sample that can comfortably fit into RAM. So for example, you might want to calculate the average number of views of a comment, by computing the average of the `views` variable, based on the random sample you drew. If we migrate our example of calculating an average number of views to the streaming context, we could efficiently draw a random sample from the data stream of comments as it arrives using Bernoulli sampling algorithm (Chapter 6). To illustrate, if you have ever plucked flower petals in the love-fortune game "(s)he loves me, (s)he loves me not'' in a random manner, you could say that you probably ended up with "Bernoulli-sampled" petals in your hand --- this sampling scheme offers itself conveniently to the one-pass-over-data context.

Answering some more granular questions about the comments data, like, below which value of the attribute `views` is 90% of all of the comments according to their view count will also trade accuracy for space. We can maintain a type of a dynamic histogram (Chapter 7) of the complete viewed data within a limited, realistic fast-memory space. This sketch or a summary of the data can then be used to answer queries about any quantiles of your complete data with some error.

Last but definitely not the least, we often deal with large data by storing it into a database or as a file on disk or some other persistent storage. Storing data on a remote storage and processing it efficiently presents a whole new set of rules than traditional algorithms (Chapter 8), even when it comes to fundamental problems such as searching or sorting. The choice of a database, for example, becomes important and it will depend on the particular workload that we expect. Will we often be adding new comment data and rarely posing queries, or will we rarely add new data and mostly query the static dataset, or, as it often happens, will we need to do both at a very rapid rate --- these are all questions that are paramount to deciding on the type of database engine we might use to store the data.

Very few people actually implement their own storage engines, but to knowledgeably choose between different alternatives, we need to understand what data structures power them underneath. Many massive-data applications today struggle to provide high query and insertion speeds, and the tradeoffs are best understood by studying the data structures that run under the hood of MySQL, TokuDB, LevelDB and other storage engines, such as B-trees,

B^{ε}-trees, and LSM-trees, where each is optimized for a particular purpose (Chapters 9 and 10). Similarly, it is important to understand the basic algorithmic tricks when working with files on disk, and this is best done by learning how to solve fundamental algorithmic problems in this context like the first example in our chapter of sorting checks in external memory (Chapter 11).

1.2 The structure of this book

As the earlier section outlines, this book revolves around three main themes, divided into three parts.

Part I (Chapters 2-5) deals with hash-based sketching data structures. This part begins with the review of hash tables and specific hashing techniques developed for massive-data setting. Even though it is planned as a review chapter, we suggest you use it as a refresher of your knowledge of hash tables, and also use the opportunity to learn about modern hash techniques devised to deal with large datasets. The succinct data structures presented in the rest of Part I are also hash-based, so Chapter 2 also serves as a good preparation for Chapters 3-5. Hash-based succinct data structures that trade accuracy for reductions in space have found numerous applications in databases, networking, or any context where space is at premium. In this part, we will often consider the tradeoffs between accuracy and space consumption for data structures that can answer membership (Bloom filter, quotient filter, cuckoo filter), frequency (Count-Min Sketch), cardinality (HyperLogLog) and other essential operations.

Part II (Chapters 6-7) serves as a continuation of Part I in that it also attempts to reduce data size, but instead of considering all data and storing it imperfectly like in Part I, the techniques in Part II consider a subset of the dataset, but then processes the original items, not their hashes. In this part, we first explain how to decide which sampling techniques for probing your data are suited for a particular setting and how large data sizes affect efficiency of these sampling mechanisms. We will start with classical techniques like Bernoulli sampling and reservoir sampling and move on to more sophisticated methods like stratified sampling to counter the deficiencies of simpler strategies in special settings. We will then use the created samples to calculate estimates of the total sums or averages, etc. We will also introduce algorithms for calculating (ensemble of) ε-approximate quantiles and/or estimating the distribution of the data within some succinct representation format.

Part III (Chapters 8-11) shifts to the scenarios when data resides on SSD/disk, and introduces the external-memory model, the model that is well suited for the analysis of algorithms where the data transfer cost subsumes the CPU cost. We will cover some of the fundamental problems such as searching, sorting, and designing optimal data structures that power relational (and some of the NoSQL) databases (B-trees, LSM-trees, B^{ε}-trees). We will also discuss the tradeoffs between the write-optimized and the read-optimized databases, as well as the issues of sequential/random access and data layout on disk.

1.3 What makes this book different and whom it is for

There is a number of great books on classical algorithms and data structures, some of which include: *Algorithm Manual Design* by Steve S. Skiena[4], *Introduction to Algorithms* by Cormen, Leiserson, Rivest and Stein[5], *Algorithms* by Robert Sedgewick and Kevin Wayne[6], or for a more introductory and friendly take on the subject, *Grokking Algorithms* by Aditya Bhargava[7]. The algorithms and data structures for massive datasets are slowly but surely making their way into the mainstream textbooks, but the world is moving fast and our hope is that this book can provide a compendium of the state-of-the-art algorithms and data structures that can help a data scientist or a developer handling large datasets at work. The book is intended to offer a good balance of theoretical intuition, practical use cases and (pseudo)code snippets. This book assumes that a reader has some fundamental knowledge of algorithms and data structures, so if you have not studied the basic algorithms and data structures, you should first cover that material before embarking on this subject.

Many books on massive data focus on a particular technology, system or infrastructure. This book does not focus on the specific technology neither does it assume familiarity with any particular technology. Instead, it covers underlying algorithms and data structures that play a major role in making these systems scalable. Often the books that do cover algorithmic aspects of massive data focus on machine learning. However, an important aspect of handling large data that does not specifically deal with inferring meaning from data, but rather has to do with handling the size of the data and processing it efficiently, whatever the data is, has often been neglected in the literature. This book aims to fill that gap.

There are some books that address specialized aspects of massive datasets [8],[9],[10],[11]. With this book, we intend to present these different themes in one place, often citing the cutting-edge research and technical papers on relevant subjects. Lastly, our hope is that this book will teach a more advanced algorithmic material in a down-to-earth manner, providing mathematical intuition instead of technical proofs that characterize most resources on this subject. Illustrations play an important role in communicating some of the more advanced technical concepts and we hope you enjoy them.

Now that we got the introductory remarks out of the way, let's discuss the central issue that motivates topics from this book.

1.4 Why is massive data so challenging for today's systems?

There are many parameters in computers and distributed systems architecture that can shape the performance of a given application. Some of the main challenges that computers face in processing large amounts of data actually stem from hardware and general computer architecture. Now, this book is not about hardware, but in order to design efficient

[4] S. S. Skiena, The Algorithm Design Manual, Second Edition, Springer Publishing Company, Incorporated, 2008.
[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to algorithms, Third Edition, The MIT Press, 2009.
[6] R. Sedgewick and K. Wayne, Algorithms, Fourth Edition, Addison-Wesley Professional, 2011.
[7] A. Bhargava, Grokking Algorithms: An Illustrated Guide for Programmers and Other Curious People, Manning Publications Co., 2016.
[8] G. Andrii, Probabilistic Data Structures and Algorithms for Big Data Applications, Books on Demand, 2019.
[9] B. Ellis, Real-Time Analytics: Techniques to Analyze and Visualize Streaming Data, Wiley Publishing, 2014.
[10] C. G. Healey, Disk-Based Algorithms for Big Data, CRC Press, Inc., 2016.
[11] A. Rajaraman and J. D. Ullman, Mining of Massive Datasets, Cambridge University Press, 2011.

algorithms for massive data, it is important to understand some physical constraints that are making data transfer such a big challenge. Some of the main issues include: 1) the large asymmetry between the CPU and the memory speed, 2) different levels of memory and the tradeoffs between the speed and size for each level, and 3) the issue of latency vs. bandwidth. In this section, we will discuss these issues, as they are at the root of solving performance bottlenecks of data-intensive applications.

1.4.1 The CPU-memory performance gap

The first important asymmetry that we will discuss is between the speeds of CPU operations and memory access operations in a computer, also known as the CPU-memory performance gap[12]. Figure 1.3 shows, starting from 1980, the average gap between the speeds of processor memory access and main memory access (DRAM memory), expressed in the number of memory requests per second (the inverse of latency):

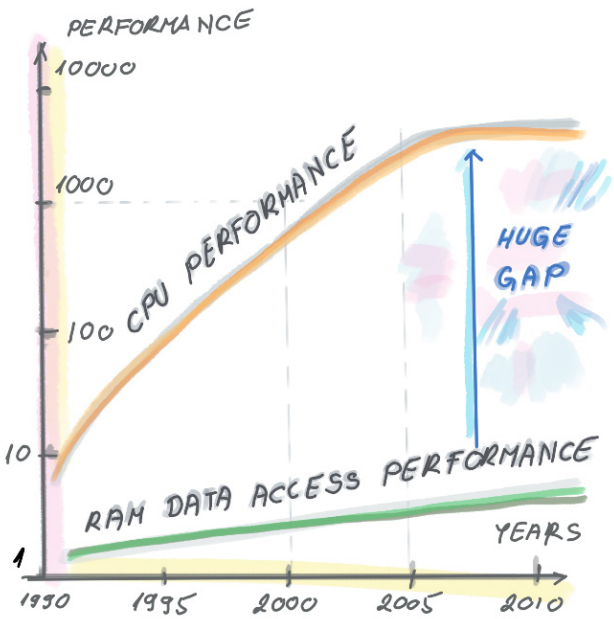

Figure 1.3: CPU-Memory Performance Gap graph, adopted from Hennessy & Patterson's well-known Computer Architecture textbook. The graph shows the widening gap between the speeds of memory accesses to CPU and RAM main memory (the average number of memory accesses per second over time.) The vertical axis is on the log scale. Processors show the improvement of about 1.5x per year up to year 2005, while the improvement of access to main memory has been only about 1.1x per year. Processor speed-up has somewhat flattened since 2005, but this is being alleviated by using multiple cores and parallelism.

[12] J. L. Hennessy and D. A. Patterson, Computer Architecture, Fifth Edition: A Quantitative Approach, Morgan Kaufmann Publishers Inc., 2011.

What this gap points to intuitively is that doing computation is much faster than accessing data. So if we are still stuck with the traditional mindset of measuring the performance of algorithms using the number of computations (and assuming memory accesses take the same amount of time as the CPU computation), then our analyses will not jive well with reality.

1.4.2 Memory hierarchy

Aside from the CPU-memory gap, there is a whole hierarchy of different types of memory built into a computer that have different characteristics. The overarching tradeoff has been that the memory that is fast is also small (and expensive), and the memory that is large is also slow (but cheap). As shown in Figure 1.4, starting from the smallest and the fastest, the computer hierarchy usually contains the following levels: registers, L1 cache, L2 cache, L3 cache, main memory, solid state drive (SSD) and/or the hard disk (HDD). The last two are persistent (non-volatile) memories, meaning the data is saved if we turn off the computer, and as such are suitable for storage.

In Figure 1.4, we can see the access times and capacities for each level of the memory in a sample architecture[13]. The numbers vary across architectures, and are more useful when observed in terms of ratios between different access times rather than the specific values. So for example, pulling a piece of data from cache is roughly 1 million times faster than doing so from the disk.

[13] C. Terman, "MIT OpenCourseWare, Massachusetts Institute of Technology," Spring 2017. [Online]. Available: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-004-computation-structures-spring-2017/index.htm. [Accessed 20th January 2019].

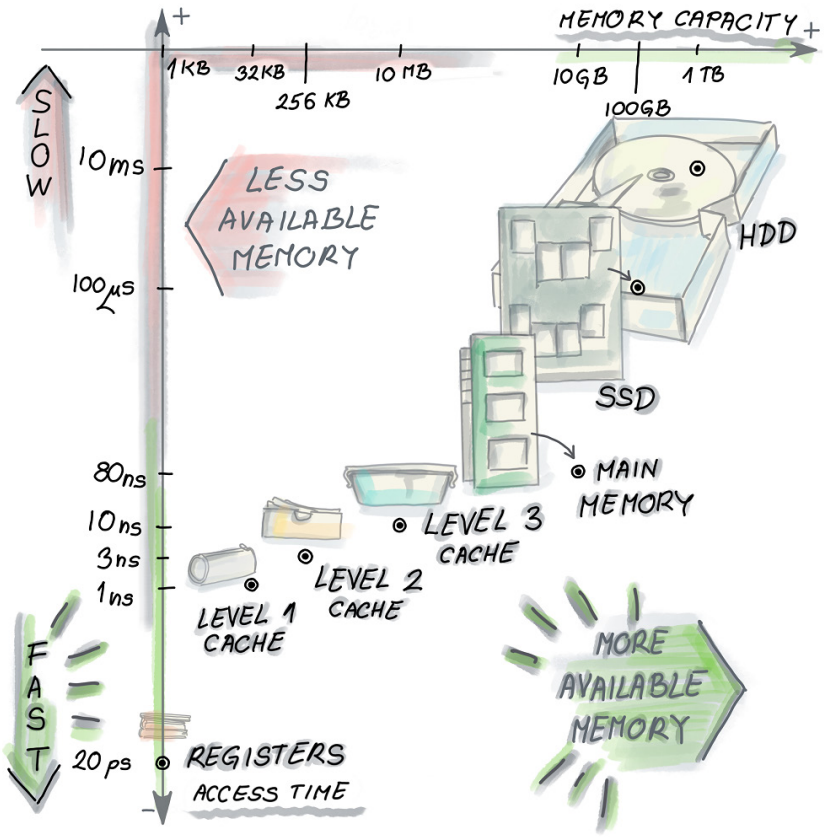

Figure 1.4: Different types of memories in the computer. Starting from registers in the bottom left corner, that are blindingly fast but also very small, we move up (getting slower) and right (getting larger) with Level 1 cache, Level 2 cache, Level 3 cache, main memory, all the way to SSD and/or HDD. Mixing up different memories in the same computer allows for the illusion of having both the speed and the storage capacity, by having each level serve as a cache for the next larger one.

The hard disk is the only remaining mechanical part of a computer, and it works a lot like a record player. Placing the mechanical needle on the right track is the expensive part of accessing data on disk. Once the needle is on the right track, the data transfer can be very fast, depending on how fast the disk spins.

Similar phenomenon, where **"latency lags bandwidth"** holds for other types of memory[14]. Generally, the bandwidth in various systems, ranging from microprocessors, main memory, hard disk, network, has tremendously improved over the past few decades, but latency hasn't as much, even though the latency might often be the more important

[14] D. A. Patterson, "Latency Lags Bandwith," Commun. ACM, vol. 47, no. 10, p. 71–75, 2004.

measurement for most scenarios --- a common pattern of user behavior is many small random accesses as oppose to one large sequential one.

Because of this expensive initial call, it is appropriate that the data transfer between different levels of memory is done in chunks of multiple items, to offset the cost of the call. The chunks are proportionate to the sizes of memory levels, so for cache that are between 8 bytes and 64 bytes, and disk blocks that can be up to 1MB[15]. Due to the concept known as *spatial locality*, where we expect the program to access memory locations that are in the vicinity of each other close in time, transferring blocks in a way pre-fetches the data we will likely need in the future.

1.4.3 What about distributed systems?

Most applications today run on multiple computers, and having data sent from one computer to another adds yet another level of delay. Data transfer between computers can be about hundreds of milliseconds, or even seconds long, depending on the system load (e.g., number of users accessing the same application), number of hops to destination and other details of the architecture, see Figure 1.5:

Figure 1.5: Cloud access times can be high due to the network load and complex infrastructure. Accessing the cloud can take hundreds of milliseconds or even seconds. We can observe this as another level of memory that is even larger and slower than the hard disk. Improving the performance in cloud applications can be additionally hard because times to access or write data on a cloud are unpredictable.

[15] J. L. Hennessy and D. A. Patterson, Computer Architecture, Fifth Edition: A Quantitative Approach, Morgan Kaufmann Publishers Inc., 2011.

How are all these facts relevant for the design of data-efficient algorithms and data structures, you might be asking yourself. The first important take-away is that, although technology improves constantly (for instance, SSDs are a relatively new development and they do not share many of the issues of hard disks), some of the issues, such as the tradeoff between the speed and the size of memories are not going away any time soon. Part of the reason for this is purely physical: to store a lot of data, we need a lot of space, and the speed of light sets the physical limit to how fast data can travel from one part of the computer to the other, or one part of the network to the other. To extend this to a network of computers, we will cite[16] an example that for two computers that are 300 meters away, the physical limit of data exchange is 1 microsecond.

Hence, we need to design algorithms with this awareness. Designing succinct data structures (or taking data samples) that can fit into fast levels of memory helps because we avoid expensive disk seeks. So one of the important algorithmic tricks with large data is that **saving space saves time**. Yet, in many applications we still need to work with data on disk. Here, designing algorithms with optimized patterns of disk access and caching mechanisms that enable the smallest number of memory transfers is important, and this is further linked to how we lay out and organize data on a disk (say in a relational database). Disk-based algorithms prefer smooth scanning over the disk over random hopping --- this way we get to make use of a good bandwidth and avoid poor latency, so one meaningful direction is transforming an algorithm that hops into one that glides over data. Throughout this book, we will see how classical algorithms can be transformed, and new ones can be designed having space-related concerns in mind.

Lastly, it is important to keep in mind that many aspects of making systems work in production have little to do with designing a clever algorithm. Modern systems have many performance metrics other than scalability, such as: security, availability, maintainability, etc. So, real production systems need an efficient data structure and an algorithm running under the hood, but with a lot of bells and whistles on top to make all the other stuff work for their customers (see Figure 1.6). However, with ever-increasing amounts of data, designing efficient data structures and algorithms has become more important than ever before, and we hope that in the coming pages you will learn how to do exactly that.

[16] D. A. Patterson, "Latency Lags Bandwith," Commun. ACM, vol. 47, no. 10, p. 71–75, 2004.

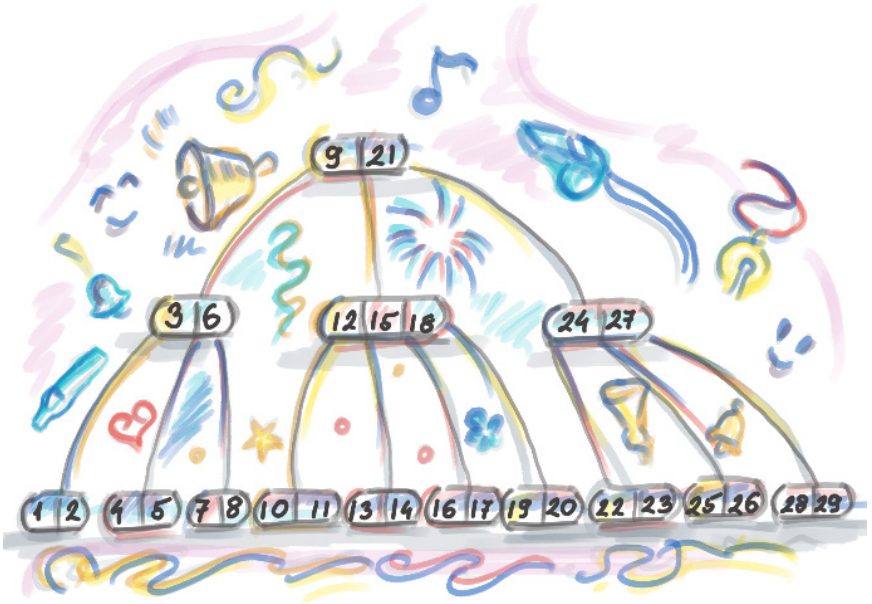

Figure 1.6: An efficient data structure with bells and whistles

1.5 Summary

- Applications today generate and process large amounts of data at a rapid rate. Traditional data structures, such as basic hash tables, and key-value dictionaries, can grow too big to fit in RAM memory, which can lead to an application choking due to the I/O bottleneck.
- To process large datasets efficiently, we can design space-efficient hash-based sketches, do real-time analytics with the help of random sampling and approximate statistics, or deal with data on disk and other remote storage more efficiently.
- This book serves as a natural continuation to the basic algorithms and data structures book/course, because it teaches how to transform the fundamental algorithms and data structures into algorithms and data structures that scale well to large datasets.
- The key reason why large data is a major issue for today's computers and systems is that CPU (and multiprocessor) speeds improve at a much faster rate than memory speeds, the tradeoff between the speed and size for different types of memory in the computer, as well as latency vs. bandwidth phenomenon. These trend are not likely to change significantly soon, so the algorithms and data structure that address the I/O cost and issues of space are only going to increase in importance over time.
- In data-intensive applications, optimizing for space means optimizing for time.

2

Review of Hash Tables and Modern Hashing

This chapter covers:

- Reviewing dictionaries and why hashing is ubiquitous in modern systems
- Refreshing some basic collision-resolution techniques: theory and real-life implementations
- Exploring cache-efficiency in hash tables
- Using hash tables for distributed systems and consistent hashing
- Learning how consistent hashing works in P2P networks: use case of Chord

We begin with the topic of hashing for a couple of reasons. First, hash tables have proved irreplaceable in modern systems and hashing is one of the most, if not the most applicable algorithm/data structure out there. Second, hash tables grow linearly with the size of the dataset they store, and there is a growing interest in techniques and tricks that address issues arising from having large hash tables, such as the choice of a collision-resolution technique and a hash function, compact representation, resizing, etc. Third, hashing has been over time adapted and repurposed to serve in massive peer-to-peer and distributed systems, and the new hashing methods, such as consistent hashing, have been designed for use in such settings. And fourth, hashing is the backbone for all the data structures we cover in Part I of the book that deals with sketching and approximate membership, frequency and cardinality data structures, so doing a brief review of hashing can't hurt as a preparation for future chapters.

This chapter will briefly review the fundamental ideas of hashing with a more practice-oriented bent, using examples of where hashing has been used in modern applications and how some modern programming languages implement hash tables and deal with different

design tradeoffs. Then we will discuss compact hash tables and hashing in the distributed context, i.e., consistent hashing.

If you feel comfortable with all things classical hashing, skip right to the Section 2.6, or if you already know consistent hashing, skip right ahead to Chapter 3.

2.1 The great idea of hashing

Hashing is one of those subjects that, no matter how much attention they got in your data structures and algorithms course, it was not enough. Nowadays, it seems much harder to find a system that does not use hashing than one that does. Just consider the process of writing an email (see Figure 2.1):

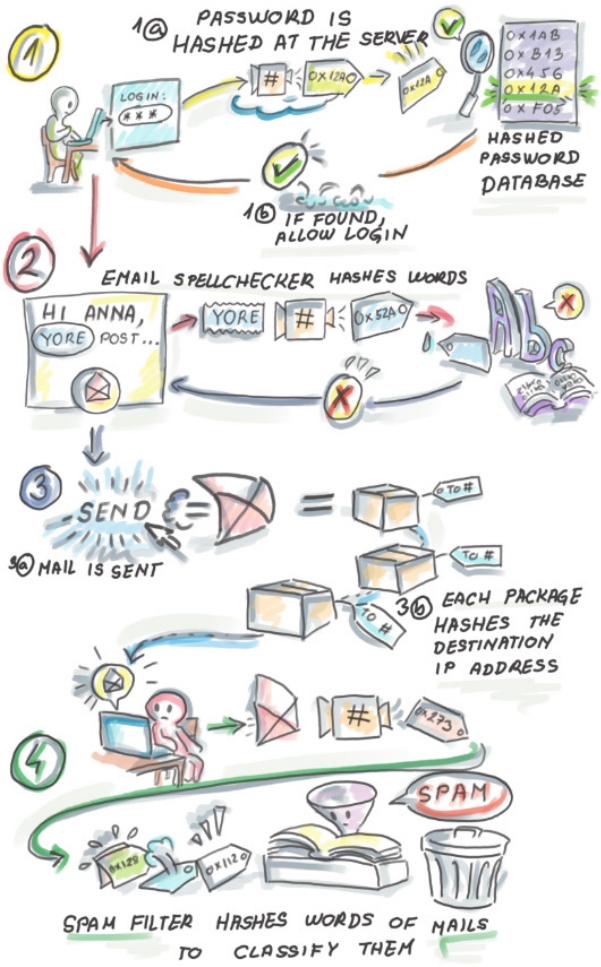

Figure 2.1: Where hashing pops up in e-mail communication: to log into your email account, the password you

type in is hashed to check against the database of stored hashes of existing passwords. While writing an email, the spellchecker hashes words to check against the database of hashes of dictionary words; when the email is sent, it is separated into packets, each of which has a hashed destination IP address on it. Lastly, some spam filters use hashing to classify the incoming emails.

Dictionary is a term for an abstract data type that can do lookup, insert and remove operations on data, so most applications implement the dictionary in some way. If you have taken the first course in data structures and/or algorithms, then you know that there are many different ways to implement dictionaries that involve different performance tradeoffs. For example, basic unsorted arrays offer ideal performance on inserts (by appending to the log in $O(1)$), but terrible performance on searches (by scanning the entire array in $O(n)$). Similarly contrasting performance on inserts versus searches is exhibited by linked lists and other linear data structures. When we break out of the linear data structures, things improve substantially, with balanced binary search trees achieving $O(\log n)$ on all operations, a major improvement over linear-time.

With asymptotic runtimes, it is always good to do quick math in your head to understand how good or how bad something is. For instance, logarithmic time on 1 billion items, is on the order of 30 items, so logarithmic is much closer to constant than to linear. Analogously, log-linear $(O(n \log n))$, the runtime of fast sorting algorithms, as a curve, is very close to the linear one $(O(n))$, and very far from quadratic $(O(n^2))$.

Hashing offers the illusion of the perfect dictionary where all operations run instantly (e.g., $O(1)$). So if you want to make your dictionary blazing fast, use a hash table! But wait, when wouldn't you want your dictionary blazing fast? Well, there are things you get with hash tables and things you don't.

Hash tables only promise $O(1)$ in the expected sense, meaning the worst case can be very bad ($O(n)$), but this is different than the $O(n)$) that we get with arrays: most real-life implementations are highly optimized and the bad case happens rarely; and when it does happen, it is amortized against a huge number of blindingly fast cases. There are also hashing schemes that perform $O(1)$ in the worst case, but it is hard to find their implementations in real systems, as they also tend to complicate the common case.

An objective disadvantage of hash tables is that they perform poorly on all operations that need an in-order (e.g., lexicographical) traversal over items, for example, predecessor, successor or range queries. Because a good hash function scrambles the input and scatters items to different areas of the hash table, a natural consequence is that hashing does not preserve the order of items. This issue comes in focus in databases where answering a range query (a very common kind of query) requires navigating the sorted order of elements: for instance, listing all employees ages between 35 and 56, or in a spatial database, finding all points on a coordinate x between 3 and 45. Hash tables are most useful when we are looking for an exact match in the database. However, it is possible to use hashing to answer queries about similarity (e.g., in plagiarism-detection), as we will see in the scenarios below.

2.2 Usage scenarios in modern systems

There are many applications of hashing wherever you look. Here are two that we particularly like:

2.2.1 Deduplication in backup/storage solutions

Many companies such as Dropbox and Dell EMC Data Domain storage systems[17] and various companies for data backup deal with storing large amounts of users' data and taking frequent snapshots and backups. Suppose you work at a company subscribed to a backup service that records a snapshot of all your company's data every 24 hours. For the storage solutions providing this service, the great majority of data will be the same between two consecutive snapshots. Generally speaking, if we consider data on the web, in our personal emails, local directories, we might find a lot of duplicated data, and especially if we consider teams of people working on joint projects. Even companies that process petabytes of data on a daily basis cannot afford to store every snapshot individually.

Deduplication is the process of eliminating duplicate content, and majority of its modern implementations use hashing. For example, consider *ChunkStash*[18], a deduplication system specifically designed to provide a fast throughput using Flash. In ChunkStash, files are split into small chunks that are fixed in size (say 8KB), and every chunk content is hashed to a 20-byte SHA-1 fingerprint; if the fingerprint is already present, we only point to the existing fingerprint. If the fingerprint is new, we can assume the chunk is also new, and we both store the chunk to the data store and store the fingerprint into the hash table, with the pointer to the location of the corresponding chunk in the data store (see Figure 2.2).

Instead of chunking, one can also use entire files and hash them, but that way we miss the subtle duplicates, i.e., near-duplicates, where small edits have been made to a large file.

[17] DELL EMC, https://www.dell.com. [Online]. Available: https://www.dell.com/downloads/global/products/pvaul/en/dell-emc-dd-series-brochure.pdf [Accessed 29 March 2020].

[18] B. Debnath, S. Sengupta and J. Li, "ChunkStash: Speeding up Inline Storage Deduplication Using Flash Memory," in Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, Boston, MA, 2010.

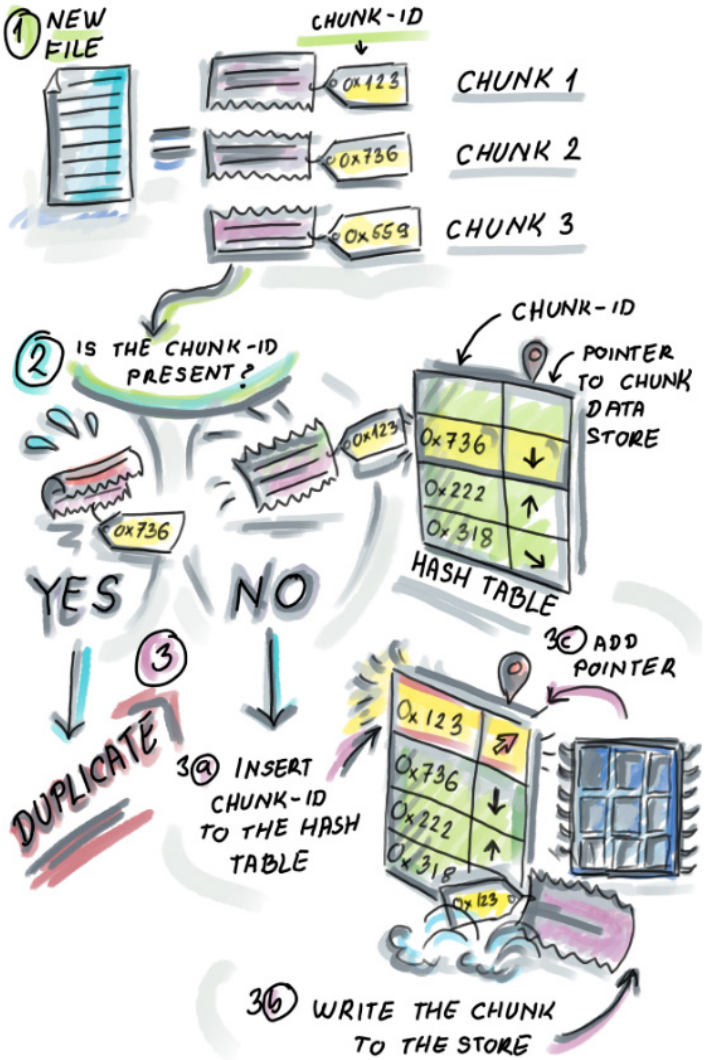

Figure 2.2: Deduplication process in backup/storage solutions. When a new file arrives, it is split into small chunks, each of which is later hashed. In our example, the file is split into three chunks, and chunk 1 becomes hash (or chunk-id) 0x123. Next, the chunk-id is compared to the database of chunk-ids, and because it is not found in the database, a new entry is created for this particular chunk-id, that is, the mapping between the chunk-id and the pointer to the location in the chunk store where we write chunk 1. For chunk 2, the chunk-id is 0x736, and because it is already found in the database, it is identified as a duplicate.

There are more details to this process than what is shown in Figure 2.2. For example, when writing the new chunk to the store, the chunks are first accumulated into an in-memory write buffer, and once full, the buffer is flushed to Flash in one fell swoop. This is done to avoid repeated small edits to the same Flash page when writing chunks one by one --- a fairly expensive operation. Buffering and writing efficiently to disk will be given more attention in the Part III of the book.

2.2.2 Plagiarism detection with MOSS and Rabin-Karp fingerprinting

MOSS (Measure of Software Similarity) is a plagiarism-detection service, mainly used to detect plagiarism in programming assignments. One of the main algorithmic ideas in MOSS[19] is a variant of Karp-Rabin string-matching algorithm[20] that relies on k-gram fingerprinting (k-gram is a contiguous substring of length k). Given a string t that represents a large text, and a string p that represents a smaller pattern that we are searching for, a string-matching problem asks whether there exists an occurrence of pattern p in string t. There is a rich literature on string-matching algorithms that compare substrings of the two strings in various ways; what differentiates Karp-Rabin from the majority of such algorithms is that it compares hashes of the substrings instead of the substrings themselves.

The algorithm proceeds in the way that only when the hashes match, we proceed to check whether the substrings actually match (see Figure 2.3). In the worst case, we will get many false matches, so the total runtime might be $O(|t||p|)$, but in most situations, and with a good hash function, we would expect linear time in size of the text $O(|t|)$. This is the randomized runtime, but clearly good enough to offer some real practical benefits.

Generally speaking, the time to compute the hash depends on the size of the substring (a good hash function should take all characters into account) so just by itself, hashing does not make the algorithm faster. However, Karp-Rabin uses *rolling hashes* where, given the hash of a k-gram $t[j,...,j+k-1]$, computing the hash for the k-gram shifted one position to the right, $t[j+1,..,j+k]$, only takes constant time (also see Figure 2.3). This can be done if the rolling hash function is such that is allows us to, in some way "subtract" the first character of the first k-gram, and "add" the last character of the second k-gram (a very simple example of such a rolling hash is a function that is a sum of ASCII values for each character of the string it is hashing.)

[19] S. Schleimer, D. S. Wilkerson and A. Aiken, "Winnowing: Local Algorithms for Document Fingerprinting," in Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, 2003.

[20] C. T. H., C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, Third Edition, The MIT Press, 2009.

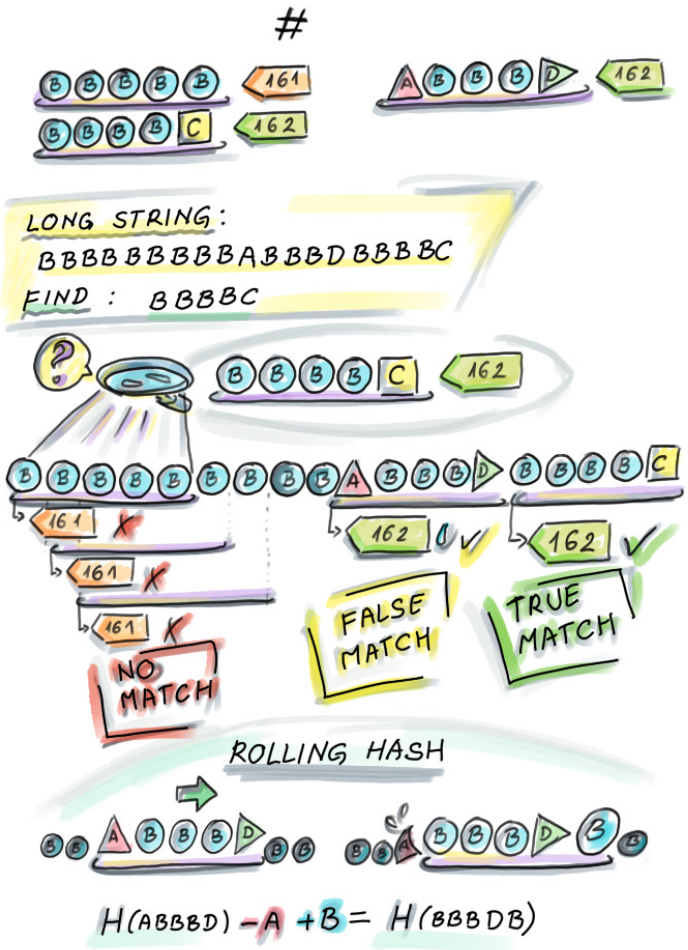

Figure 2.3: Karp-Rabin fingerprinting algorithm. We are looking for a pattern BBBBC in the larger string BBBBBBBBBABBBDBBBBC. The hash of BBBBC is equal to 162 and it is a mismatch for the hash 161 of BBBBB that occurs at the beginning of the long string. As we shift right in the long string, we repeatedly encounter hash mismatches until we encounter the substring ABBBD, whose hash equals 162, and is a match for the hash pattern. Then we actually check the substrings and determine that it was a false match. At the very end of the string, we again encounter the hash match; when we inspect the actual substrings, we report a true match. Computing the hash of the first substring is an operation whose time is proportional to the size of the substring; for all other substrings, it is a constant-time operation, as it is only a modification of the existing hash from the previous substring. For example, the hash of ABBBD equals 162, and to compute the hash for the substring one position to the right, BBBDB, we need to "subtract" A and "add" B.

In MOSS, however, we are comparing entire files, and if there is a large set of assignments to be mutually compared for plagiarism, that might lead to a quadratic number of pairwise

comparisons on the entire sets of fingerprints. To battle the quadratic time, MOSS selects a small number of fingerprints as representative of each file to be compared. The application builds an *inverted index* mapping the fingerprint to its position in the documents where it occurs. From the index, we can further compute a list of similar documents. Note that the list will only have documents that actually have matches, so we are avoiding the blind all-to-all comparison.

There are many different techniques on how to choose the set of representative fingerprints for a document. The one that MOSS employs is having each *window* of consecutive characters in a file (for instance, a window can be of length 50 characters) select a minimum hash of the k-grams belonging to that window. Having one fingerprint per window is helpful, among other things, because it helps avoid missing large consecutive matches.

2.3 $O(1)$ --- what's the big deal?

After seeing some applications of hashing, let's now turn to how to efficiently design hash tables. Namely, why is it so hard to design a simple data structure that just does everything in $O(1)$ in the worst case?

It could work if we knew all the items that will be inserted beforehand; then we could conjure up a customized hash function that distributes items perfectly, one to each bucket of the hash table, but what's the fun in that. Part of the problem with not knowing data beforehand is that the size of the universe U of potential items (read: all possible items you might get in a particular application) is much larger than n, the size of our dataset, whereas the table size m and n are values close to each other. Then, by the pigeonhole principle, it is inevitable that there is at least one bucket where at least $|U|/m$ items from the universe map to that bucket. And if $|U|/m \geq n$, then it is feasible that all items in our dataset hash to that very bucket.

Consider the example of all potential phone numbers of the format $DDD\text{-}DD\text{-}DDD\text{-}DDDD$, where D can be a digit 0-9.This means that $|U|=10^{12}$ and if $n=10^6$ (the number of items), and $m=10^6$ (size of the table), even if the hash function perfectly distributes items from the universe, we can still end up with all the items in one bucket. Now, the fact that this is possible should not discourage us. In most practical applications, even simple hash functions are good enough for this to very rarely happen, but collisions will happen in common case and we need to know how to deal with them.

2.4 Collision Resolution: theory vs. practice

We will devote this section to two common hashing mechanisms: linear probing and chaining. There are many others, but we will cover these two as they are the most popular choices in the production-grade hash tables. As you probably know, linear probing inserts items into the slots of the table, while chaining associates with each bucket of the hash table an additional data structure (e.g., linked list, or binary search trees), that will store all the items hashed to the corresponding bucket. To refresh your memory on how linear probing and chaining work, see Figure 2.4:

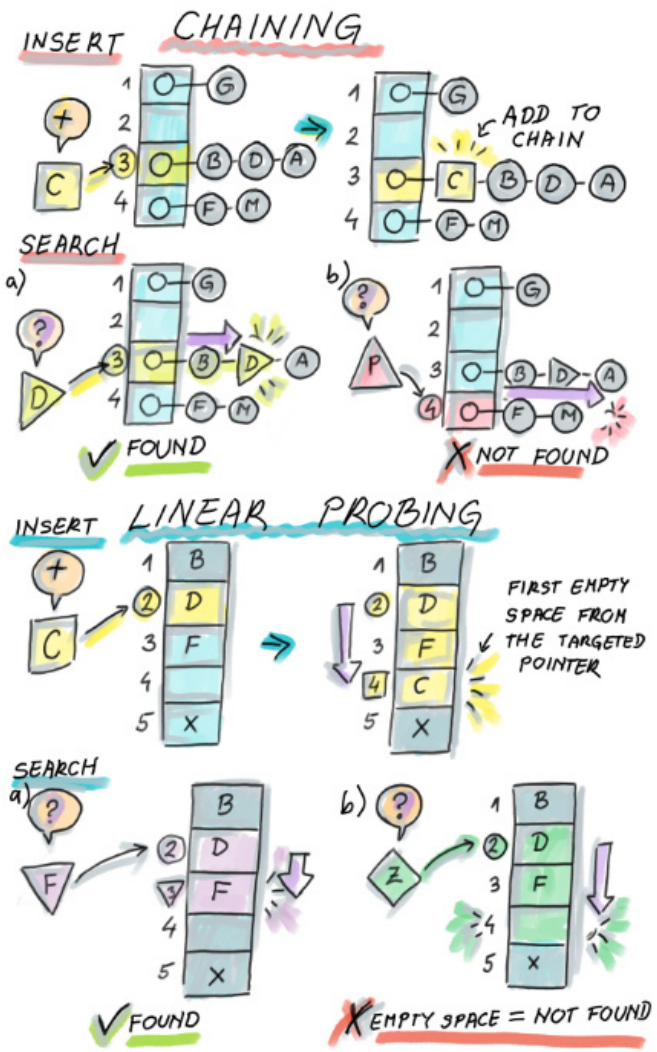

Figure 2.4: An example of insert and lookup operation with chaining and linear probing. In chaining, we insert an item at the front of the linked list defined by the bucket where the item hashed. For example, C hashed to bucket 3, where the associated list is B, D, A. After C's insert, the list will be C, B, D, A. Searching, analogously, needs to search inside the list associated with the bucket where the item hashed. In this example, D hashes to bucket 3, and it is found in the corresponding list B, D, A, whereas P is hashed to bucket 4, with the corresponding list F, M, and the search returns Not Found. With linear probing, insert looks for the first empty slot starting from the position where is hashed, wrapping around the end of the table if needed. In this example, C hashes to bucket 2, but is eventually stored at bucket 4, the first empty slot after 2. Search, similarly, starts at the bucket where the item hashes, and it ends at the first empty slot, also wrapping around the table if needed. Here, an item F hashes to bucket 2, and is found at bucket 3, whereas item Z hashes to bucket 2, and at bucket 4 it encounters the first empty slot without having found Z, and it returns Not Found.

First let's see what the theory tells us about pros and cons of these two collision-resolution techniques. Theoretically speaking, in studying hash functions and collision-resolution techniques, computer scientists will often use the assumption of hash functions being ideally random. So for example, one analogy for analyzing how long the longest chain is in the chaining method is to analyze what happens when we throw n balls uniformly randomly into n bins. One can prove that with high probability the longest chain in the chaining method is no longer than $O(\log n/\log \log n)$.[21]

This means the lookup on the chaining hash table is (at most) $O(\log n/\log \log n)$ almost always. This is not bad, but if all lookups were like this, then the hash table would not offer significant advantages over, say, a binary search tree. In most cases, though, we expect a lookup to only be a constant. (What we mean by high probability, leaving out a lot of math, is that one is more likely to get hit by a meteor while reading this than for the high-probability event not to happen. In more practical circumstances, it means that in a computer system, many other failures will happen before the high-probability event fails us.)

Because ideal random functions do not exist, computer scientists try to come up with closer-to-reality models of hash functions that are not ideally random but are close enough, that still guarantee properties like the above one. There are families of k-wise independent hash functions H that can mimic the random behavior pretty well. At runtime, one of the hash functions from the family is selected uniformly randomly to be used throughout the program. This is a suitable model when analyzing your algorithms against an adversary who can see your code: no matter how good of a hash function we designed, after some time, the adversary can come up with a pathological dataset. Choosing one among many hash functions randomly at runtime protects us from that --- the worst case can still happen, but it will be harder to produce and it will not be our fault.

For linear probing, the runs are naturally longer than those for chaining, because they are created not only by the items that hashed to the bucket. Using a pairwise independent family, one can show that lookups on long runs in linear probing are closer to $O(\log n)$.[22]

2.4.1 Cache-efficiency vs. number of probes

If we only observe the number of probes required to access the item on a lookup, theoretical analysis says chaining should be faster. But many practical benchmarks tell a different story: namely, if we take into account the modern computer hierarchy, then we should switch from thinking about the number of probes (number of CPU operations) to the number of cacheline fetches (number of memory accesses.) Here the picture changes, because in a linear probing table, items are laid out in the sequential memory order, and most runs are shorter than a single cacheline (cachelines are usually 4KB-64KB). On the other hand, items in a chaining table are organized in a linked data structure, where items are not necessarily in the same cacheline, and the pointers take up additional space that data could be occupying, so the lookup on one chain of the table can easily result in more memory requests than the lookup in linear probing. Similar is with some other collision-resolution techniques, such as *cuckoo*

[21] J. Erickson, "Algorithms lecture notes," [Online]. Available: http://jeffe.cs.illinois.edu/teaching/algorithms/notes/05-hashing.pdf. [Accessed 20 March 2020].

[22] A. Pagh, R. Pagh and M. Ruzic, "Linear Probing with Constant Independence," in Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing, San Diego, California, 2007.

hashing, that promises at most two locations in a table where an item can be found (so the lookup must be $O(2) = O(1)$ probes), but the probes can be in wildly different areas of the table, and sometimes requiring two accesses to memory. So for practical hash table implementations, cache-efficiency and spatial locality might be a better measure of efficiency than simply the number of probes.

But a collision-resolution technique should not be analyzed without considering the hash function used to index into the table. Let's see an example of how one modern programming language implements the hash tables that run underneath its key-value dictionary.

2.4.2 Usage scenario: How Python's dict does it

Most situations when we use hash tables while programming include key-value dictionaries that associate a piece of information with another piece of information (word with a count, for example,). For standard libraries of C++ and Java, these are respectively `unordered_map` and `HashMap`; both of these libraries use chaining for collision resolution. For Python, the classical key-value dictionary is `dict`. Here is one way to create an instance of `dict`:

```
Dict = {flour: 5, cauliflower: 2, milk: 19}
```

Python's default implementation, CPython, implemens dict as follows (here we only focus on the case when keys are integers): for the table size $m=2^i$, the hash function is $h(x) = x \bmod 2^i$ (that is, the last i bits of binary representation of x.) According to CPython, this works well in a number of common cases, such as a sequence of consecutive numbers (it does not create collisions). But if in the combination with this hash function, one used linear probing, this would lead to clustering, and having very long runs of consecutive items. So, in order to avoid long runs Python employs the following probe sequence:

```
j = ((5*j) + 1) mod 2^i,
```

where `j` is the index of a bucket where we will attempt to insert. If the slot is taken, we will repeat the process using the new `j`. This sequence makes sure that all m buckets in the hash table are visited over time. This is the first part of the probing method. The second part is to get the higher bits of x to also play a role in determining where x gets stored, using a variable `perturb`, that is originally initialized to the $h(x)$ and a constant `PERTURB_SHIFT` set to 5:[23]

```
perturb >>= PERTURB_SHIFT
j = (5*j) + 1 + perturb
//j%(2^i) is the next bucket we will attempt
```

This way the runs get some space between them. Now, if the insertions match our $(5 * j) + 1$ pattern, then we are in trouble again, but Python, and most practical implementations of hash tables focus on what seems to be a very important practical algorithm design principle: making the common case simple and fast, and not worry about an occasional glitch when a rare pathological case occurs.

[23] Python (CPython), "Python hash table implementation of a dictionary," 20 February 2020. [Online]. Available: https://github.com/python/cpython/blob/master/Objects/dictobject.c. [Accessed 30 March 2020].

2.5 Hash Tables for Distributed Systems: Consistent Hashing

The first time the consistent hashing came to a spotlight was in the context of web caching.[24],[25] Caches are one of those good things in life, and one of the fundamental ideas that improve systems across the domains in computer science. In the context of web, for example, caches carry a great significance by relieving the hot spots when many clients request the same web page from a server. The general setup is that servers host web pages, clients request them via browsers, and caches sit in between to host copies of frequently accessed web pages. Once a cache miss occurs, a cache fetches a website from the originating server; in most situations, caches are able to satisfy the request faster than the home servers, and distribute the load so that the hot spots do not occur.

In the early days of web caching, an important challenge was figuring out how to assign resources to caches. There is a number of considerations here to take into account:

1. It should be fast and easy both for the client and the server to compute which cache is responsible for a corresponding webpage,
2. There should be a fairly equal load among different caches, and
3. The mapping should be flexible in the face of frequent node arrivals and departures. That is, as soon as a cache leaves, its resources need to be re-assigned to another cache without too much fuss, and others should be affected as little as possible. Similarly, when a new cache is added, it should receive an equal portion of the total resource load.

Considering the need for a fast and simple mapping, it sounds like we have a hashing problem at our hands. A hash function with range $[0,m-1]$ could map a web page to a bucket in the table, where each bucket represents one cache. With a good hash function, we can expect a behavior close to uniform random, so that takes care of the equal load-balancing. The only real challenge lies in satisfying the third requirement, where the number of caches is constantly changing, so the question becomes how to design the hash table that unpredictably has buckets being added and taken away.

It is possible to resize the table and rehash, but rehashing, especially when we need to do it very frequently, can be an expensive operation. The copying over of all the items into another table, a linear-time operation, pays off only because we know that we need to do it after linear number of new inserts. This way, all inserts equally share the amortized cost of $O(1)$. Copying over into a new table every time a cache arrives or departs becomes very impractical. In addition, every time we rehash, we create a new mapping for all the webpages, then all previously stored resources at caches become obsolete and have to be downloaded again. Consistent hashing solves this problem.

The main idea of consistent hashing is to map **both** web pages *and* caches to a fixed range, say $R = [0, 2^{32} - 1]$. That is, our keys are web pages and caches, and our buckets are the integers in R. It is visually more helpful to think in terms of a *hashring*, where each webpage and a cache have a mapping somewhere on the ring depending on their hash value.

[24] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine and D. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web," in Proceedings of the twenty-ninth annual ACM symposium on Theory of computing, El Paso, Texas, 1997.
[25] G. Valiant and T. Roughgarden, "CS168 The Modern Algorithmic Toolbox," 01 April 2019. [Online]. Available: https://web.stanford.edu/class/cs168/l/l1.pdf. [Accessed 30 March 2020].

Think if throwing both caches and webpages as darts onto the hashring. The assignment of the webpages to caches goes in the clockwise direction: each web page is assigned the first following cache that comes in the clockwise order, i.e., each cache has a segment of the range that it is responsible for. See Figure 2.5 where the range for hashring is $[0,31]$:

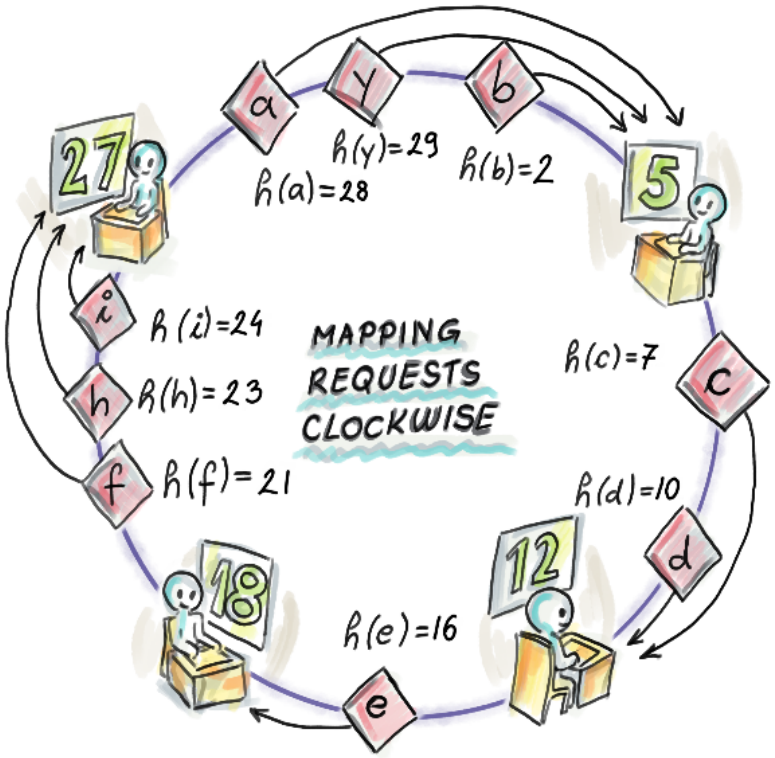

Figure 2.5: Mapping web page requests to caches. In this example, the interval for our hashring is [0,31], meaning that hashing a cache and a web page will produce an integer between 0 and 31. On this hashring, caches are denoted by their hash values and are placed in the corresponding positions on the hashring: 5, 12, 18, and 27. The requests are mapped in the clockwise order to the next cache: c and d respectively hash to 7 and 10, which maps them to cache 12. Request e hashes to 16, and is mapped to the cache 18, etc. With a good hash function, we can expect a roughly even distribution of caches around the ring, and a fairly equal number of requests assigned to each cache.

Because a good hash function distributes both items and caches evenly, we expect an approximately even load among caches. More importantly, let's observe what happens when a new cache A joins a hashring. In the case of a new cache, only the items whose new successor is the new cache A are affected, and those can only be the items in one segment of the ring. All other web pages keep their earlier assignment (see Figure 2.6):

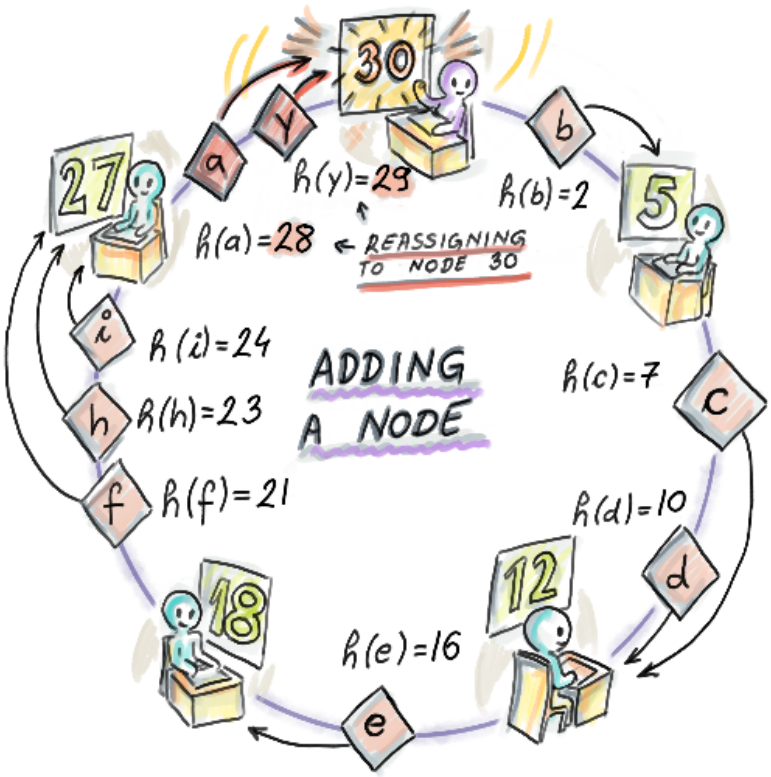

Figure 2.6: New cache arrival. When the new cache C arrives, the only candidate web page requests whose assignment is potentially changed are the ones who used to belong to C's current successor (allowing wrap-around) on the hashring. So in this example, the only requests whose assignment might change after the arrival of the cache 30 are the items who used to be assigned to 5. The requests actually assigned are a and y. In other words, when adding a new cache, we only meddle with the requests belonging to one cache, all else remains the same.

Similarly, when a cache B leaves the hashring, then the items that were mapped previously to it now have a new successor, and are assigned to the next cache in the clockwise order (see Figure 2.7). All operations on the hashring assume the usage of the `mod` operator, for the resources that cross the end of the range.

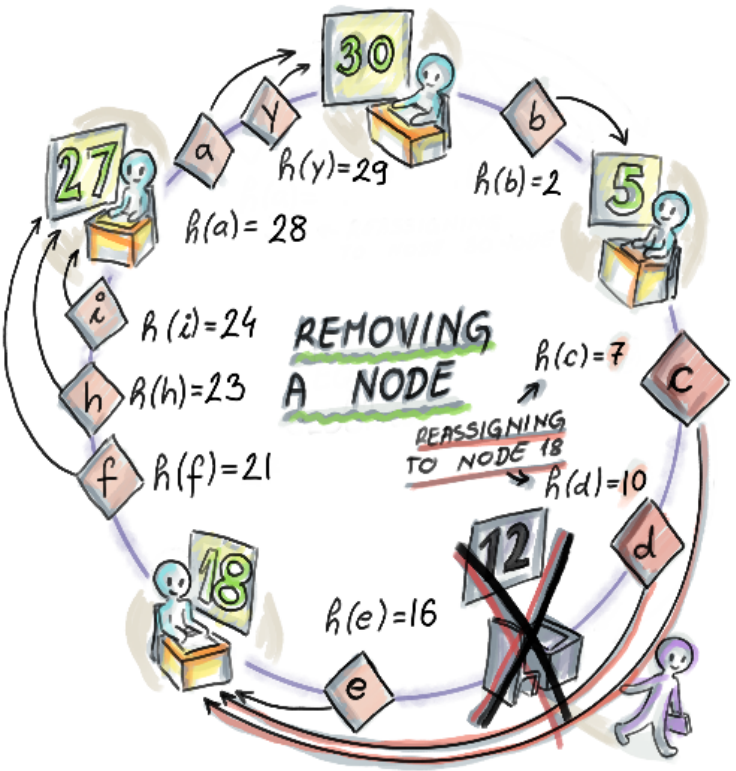

Figure 2.7: Cache departure. When a cache C leaves the hashring, its resources are re-assigned to the C's successor. Here, the cache 12 left the network, and its resources c and d got re-assigned to the cache 18.

Visually, this scheme seems to work, but how do we implement it? We need to store the caches (with their hash values as keys), into a data structure that efficiently performs the `successor` operation. A good choice of a data structure is a balanced binary search tree that performs `predecessor` and `successor` operations in $O(log\ n)$.. For instance, in C++ STL, `map` uses a red-black tree under the hood. To find the cache that a web page is assigned to, we use the hash of the webpage as the key to perform a successor operation on the tree. Node arrivals and departures are handled by inserting and deleting nodes from the tree. All these operations cost $O(log\ n)$.

2.5.1 Consistent hashing scenario: Chord

Notice that in our so far discussion of the consistent hashing, caches are distributed, but the lookup protocol isn't. Each node taking part in the mapping is informed about all the caches, i.e., whoever wants to be able to do the mapping needs to maintain their copy of the data structure storing caches and be informed about all changes to node arrivals and departures. Also, space requirement is that for k caches, each node needs to maintain $O(k)$ space for the

lookup to work. In a distributed context of peer-to-peer networks with a huge number of clients who are at the same time resource-holders, this is not feasible, because nodes only have limited space, and also limited knowledge of the rest of the network, so locating resources needs to be done without each node being informed about the whole network. For example, one simple way of maintaining a very small amount of information about other nodes is to maintain only a link to the successor node. Every request is repeatedly forwarded, such as in a linked-list lookup, to the successor node, until the correct node is found. This solution is $O(1)$ space, but $O(n)$ time.

Chord[26] is the distributed lookup protocol for P2P networks where each node requires the information only on other $O(\log n)$ nodes. Each node maintains a so-called *finger* table (Figure 2.8) that stores entries for the successors of items whose hash is of exponentially increasing distance from the cache's hash. For the ring of size m, the table of the node x is filled with successors $s = \text{successor}((x+2^{i-1}) \bmod m)$, where $1 \leq i \leq m$. For an example, see Figure 2.8:

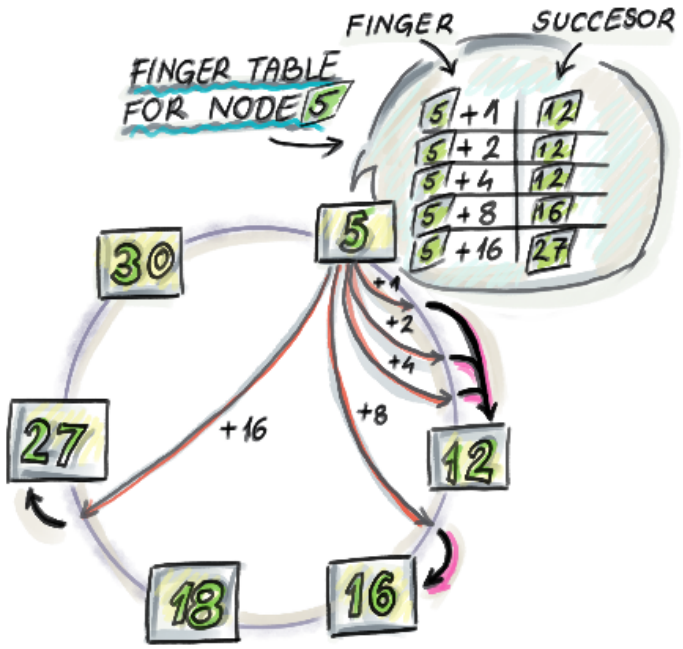

Figure 2.8: Example finger table for the node 5, on the hashring of the range [0,31]. Node 5 has 5 entries stored in its finger table, for the successors of the points on the hashring that are of exponentially increasing distance from 5. Specifically, 5 stores successors of 5+1=6, 5+2=7, 5+4=9, 5+8=13, and 5+16=21. The respective successor nodes of 6, 7, 9, 13, and 21 in this hashring are 12, 12, 12, 16 and 27.

[26] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications," IEEE/ACM Trans. Netw., vol. 11, no. 1, pp. 17-32, 2003.

The lookup operation in this scheme works in a way that, if the finger table of a node where the request originates does not contain the hash of the web page, then the node forwards the request to the successor determined by the finger whose hash is the largest one that is still smaller than the hash of our query. In this scheme, every node maintains a finger table that requires logarithmic space in the size of the range. The example is shown in Figure 2.9:

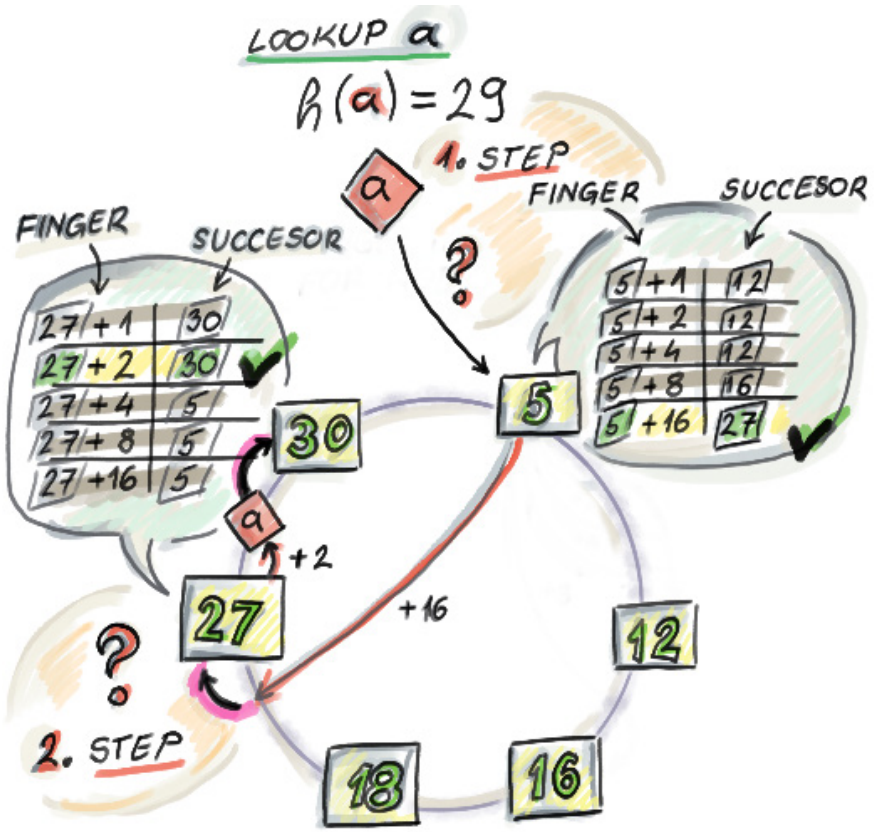

Figure 2.9: Lookup procedure with finger tables. In this example, the web page a is requested whose hash is 29, and the request originates at cache 5. In the finger table of node 5, not a single finger generates 29, so we take the finger with the largest value that is still smaller than 29. This happens to be the largest finger in this example, so we take the finger 21, whose successor is 27. In the finger table of the cache 27, we take the finger 2, which gives us exactly 29, and its successor is 30, where the request is finally routed.

The scheme from Chord, aside from being used in a number of P2P networks, has also been repurposed for Amazon's Dynamo, a highly scalable data store that stores various core

services of Amazon's e-commerce platform[27]. To do better load-balancing, Dynamo also uses replication of nodes via virtual nodes. Instead of having nodes map once to the hashring, we map nodes multiple times; those nodes are called virtual nodes. With replication, we ensure better load-balancing --- if one virtual node was "unlucky" to receive too many requests, then it is likely the other virtual node of the same originating node wasn't that unlucky. One can show that with sufficient replication (logarithmic number of virtual nodes per node), with high probability, each node will be responsible for $1/n$ fraction of the total range.

Also, if a particular node goes down, the burden of re-assigning its resources is divided amongst a couple of virtual nodes. Replication of nodes also happens in Dynamo when different nodes have different capacity, speed, etc, so for example, a node that has twice the capacity of all other nodes should be mapped to the hashring two times more than others.

2.6 Summary

- Hash tables are irreplaceable in modern systems, such as networks, databases, storage solutions, text-processing applications and so on. Depending on an application and the workload, hash tables can be designed to suit different needs, such as speed vs. space, simplicity vs optimizing the worst-case, etc.
- There is a large number of collision-resolution techniques, but the most frequently used ones are chaining and linear probing (Section 2.4). Linear probing has benefits when it comes to cache-efficiency. As hash tables grow, the cache-efficiency concern take over the number of probes required by a particular technique.
- Most production-quality hash tables care about optimizing the common case and do not worry about solving rare pathological cases if they will complicate the common case.
- Consistent hashing (Section 2.5) solves the problem of hash tables that adapt to dynamic peer-to-peer and web environments, where nodes (potential buckets) arrive and leave at a rapid rate. Consistent hashing has been implemented in many peer-to-peer products such as BitTorrent, and also in data store systems such as Amazon's Dynamo.

[27] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall and W. Vogels, "Dynamo: Amazon's highly available key-value store," SIGOPS Oper. Syst. Rev., vol. 41, no. 6, pp. 205-220, 2007.

3

Approximate Membership and Bloom Filter

This chapter covers:

- Learning what Bloom filters are, why and when they are useful
- Understanding how Bloom filters work
- Configuring a Bloom filter in a practical setting
- Exploring the interplay between Bloom filter parameters
- Learning about quotient filter as a Bloom filter replacement
- Understanding how quotient filter works, and its comparison to the Bloom filter

Bloom filters seem to be all the rage these days. Most self-respecting industry blogs have articles fleshing out how Bloom filters enhance the performance in their infrastructure, and there are dozens of Bloom filter implementations floating around in various programming languages, each touting its own benefits. Bloom filters are also interesting to computer science researchers, who have, in past decade, designed many modifications and alternatives to the basic data structure, enhancing its various aspects. A skeptic and a curmudgeon in you might ask himself: What's all the hype?

The large part of the reason behind Bloom filter popularity is that they have that combination of being a fairly simple data structure to design and implement, yet very useful in many contexts. They were invented in 1970s by Burton Bloom[28],[29] but they only really "bloomed" in the last few decades with the onslaught of large amount of data in various domains, and the need to tame and compress such huge datasets.

[28] B. H. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," Communications of the ACM, vol. 13, no. 7, pp. 422-426, 1970.
[29] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," in Internet Mathematics, 2002, pp. 636-646.

One simple way to think about Bloom filters is that they support insert and lookup in the same way the hash tables do, but using very little space, i.e., one byte per item or less. This is a significant saving when you have many items and each item takes up, say 8 bytes.

Bloom filters do not store the items themselves, and they use less space than the lower theoretical limit required to store the data correctly, and therefore, they exhibit an error rate. They have false positives but they do not have false negatives, and the one-sidedness of the error can be turned to our benefit. When the Bloom filter reports the item as `Found/Present`, there is a small chance it is not telling the truth, but when it reports the item as `Not Found/Not Present`, we know it's telling the truth. So, in the context where the query answer is expected to be `Not Present` most of the time, Bloom filters offer great accuracy plus space-saving benefits.

For instance, this is how Bloom filters are used in Google's Webtable[30] and Apache Cassandra[31] that are among the most widely used distributed storage systems designed to handle massive amounts of data. Namely, these systems organize their data into a number of tables called Sorted String Tables (SSTs) that reside on disk and are structured as key-value maps. In Webtable, keys might be website names, and values might be website attributes or contents. In Cassandra, the type of data depends on what system is using it, so for example, for Twitter, a key might be a User ID, and the value could be user's tweets.

When users query for data, the problem arises because we do not know which of the tables contains the desired result. To help locate the right table without checking explicitly on disk, we maintain a dedicated Bloom filter in RAM for each of the tables, and use them to route the query to the correct table, in the way described in Figure 3.1:

[30] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," ACM Trans. Comput. Syst., vol. 26, no. 2, pp. 4:1-4:26, 2008.

[31] S. Lebresne, "The Apache Cassandra Storage Engine," 2012. [Online]. Available: https://2012.nosql-matters.org/cgn/wp-content/uploads/2012/06/Sylvain_Lebresne-Cassandra_Storage_Engine.pdf. [Accessed 03 04 2016].

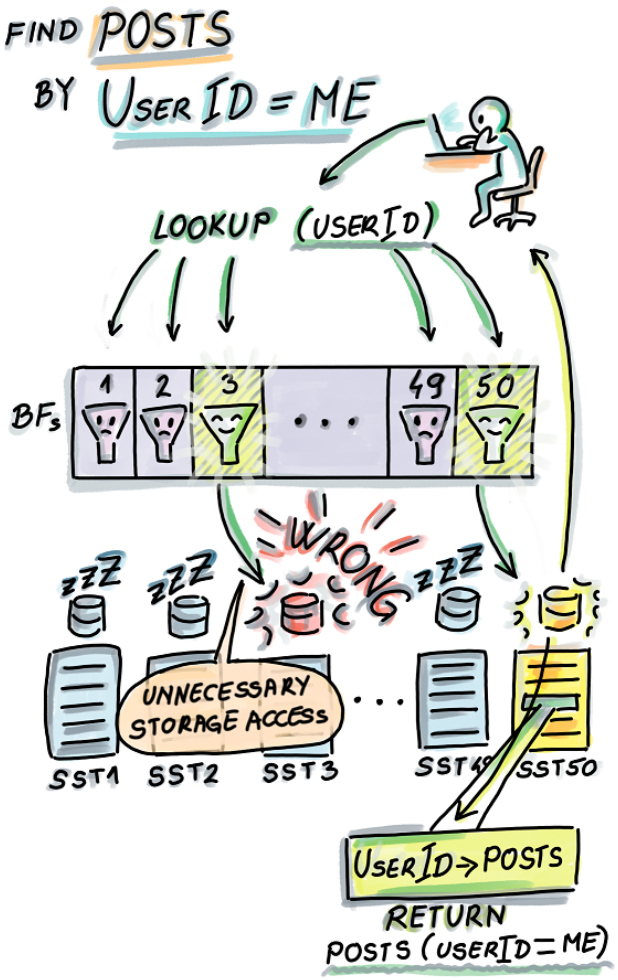

Figure 3.1: Bloom filters in distributed storage systems. In this example, we have 50 sorted string tables (SSTs) on disk, and each table has a dedicated Bloom filter that can fit into RAM due to its much smaller size. When a user does a lookup, the lookup first checks the Bloom filters. In this example, the first Bloom filter that reports the item as Present is Bloom filter No.3. Then we go ahead and check in the SST3 on disk whether the item is present. In this case, it was a false alarm. We continue checking until another Bloom filter reports Present. Bloom filter No.50 reports present, we go to the disk and actually locate and return the requested item.

Bloom filters are most useful when they are strategically placed in high-ingestion systems, in parts of the application where they can prevent expensive disk seeks. For example, having an application perform a lookup of an element in a large table on a disk can easily bring down the throughput of an application from hundreds of thousands ops/sec to only a couple

of thousands ops/sec. Instead, if we place a Bloom filter in RAM to serve the lookups, this will deem the disk seek unnecessary except when the Bloom filter reports the key as `Present`. This way the Bloom filter can remove disk bottlenecks and help the application maintain consistently high throughput across its different components.

In this chapter, you will learn how Bloom filters work and when to use them, with various practical scenarios. You will also learn how to configure the parameters of the Bloom filter for your particular application: there is an interesting interplay between the space (m), number of elements (n), number of hash functions (k), and the false positive rate (f). For readers who like a challenge, we will spend some time understanding where the formulas relating the important parameters of Bloom filter come from and exploring whether one can do better than Bloom filter.

In that light, we will spend a substantial amount of time exploring an interesting new type of a compact hash table called **quotient filter**[32] that is functionally similar to the Bloom filter, and also offers many other advantages. So if you already are well familiar with the Bloom filters, and are ready for another challenge, skip ahead to Section 3.6.

3.1 How It Works

Bloom filter has two main components:

- A bit array `A[0..m-1]` with all slots initially set to 0, and
- k independent hash functions $h_1, h_2, \ldots, h_k$, each mapping keys uniformly randomly onto a range $[0, m-1]$

3.1.1 Insert

To insert an item x into the Bloom filter, we first compute the k hash functions on x, and for each resulting hash, set the corresponding slot of `A` to 1 (see pseudocode and Figure 3.2 below):

```
Bloom_insert(x):
for i ← 1 to k
       A[hᵢ(x)] ←1
```

[32] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane and E. Zadok, "Don't Thrash: How to Cache Your Hash on Flash," in Proceedings of the VLDB Endowment (PVLDB), Vol. 5, No. 11, pp. 1627-1637 (2012), 2012.

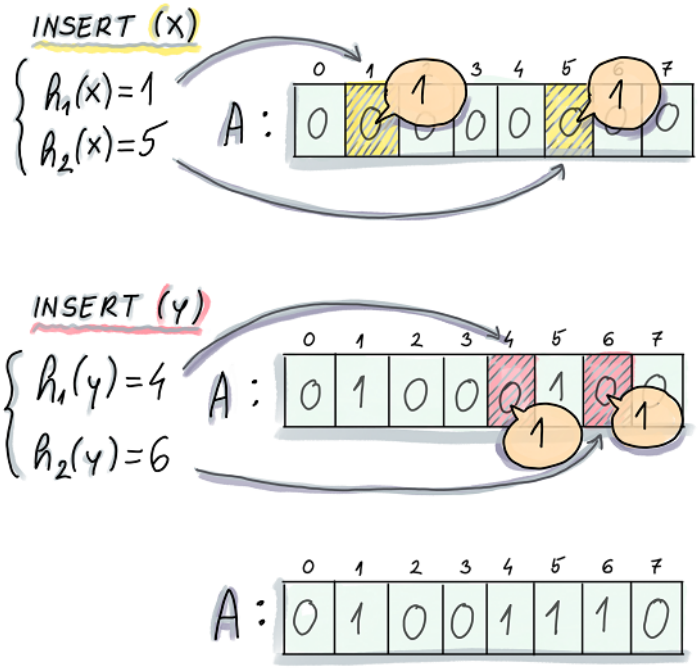

Figure 3.2: Example of insert into Bloom filter. In this example, an initially empty Bloom filter has m=8, and k=2 (two hash functions). To insert an element x, we first compute the two hashes on x, the first one of which generates 1 and the second one generates 5. We proceed to set $A[1]$ and $A[5]$ to 1. To insert y, we also compute the hashes and similarly, set positions $A[4]$ and $A[6]$ to 1.

3.1.2 Lookup

Similarly to insert, lookup computes k hash functions on x, and the first time one of the corresponding slots of A equal to 0, the lookup reports the item as Not Present, otherwise it reports the item as Present (pseudocode below):

```
Bloom_lookup(x):
for i ← 1 to k
      if(A[hi(x)] = 0)
      return NOT PRESENT
return PRESENT
```

Here is an example of a Bloom filter lookup (Figure 3.3):

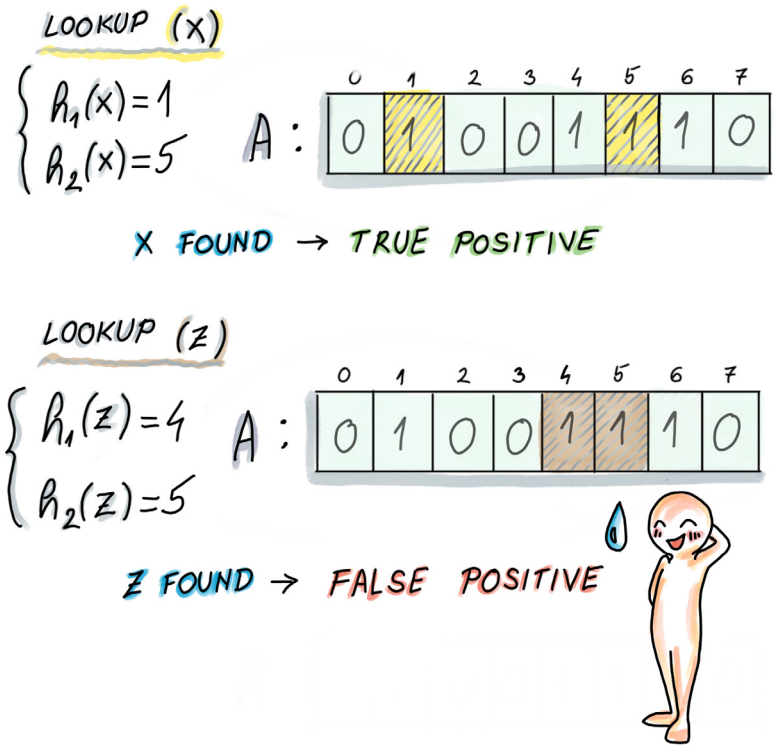

Figure 3.3: Example of a lookup on a Bloom filter. We take the resulting Bloom filter from Figure 3.2, where we inserted elements x and y. To do a lookup on x, we compute the hashes (which are the same as in the case of an insert), and we return Found/Present, as both bits in corresponding locations equal 1. Then we do a lookup of an element z, which we never inserted, and its hashes are respectively 4 and 5, and bits at locations A[4] and A[5] equal 1, thus we again return Found/Present. This is an example of a false positive, where two other items together set the bits of the third item to 1. An example of a negative (negative is always true), would be if we did a lookup on an element w, whose hashes are 2 and 5, (0 and 1), or 0 and 3 (0 and 0). If the Bloom filter reports an element as Not Found/Not Present, then we can be sure that this element was never inserted into a Bloom filter.

Asymptotically, the insert operation on the Bloom filter costs $O(k)$. Considering that the number of hash functions rarely goes above 12, this is a constant-time operation. The lookup might also need $O(k)$, in case the operation has to check all the bits, but most unsuccessful lookups will give up way before; later we will see that on average, an unsuccessful lookup takes about 1-2 probes before giving up.

3.2 Use Cases

In the introduction, we saw the application of Bloom filters to distributed storage systems. In this section, we will see more applications of Bloom filters to distributed networks: Squid network proxy, and Bitcoin mobile app.

3.2.1 Bloom Filters in Networks: Squid

Squid is a web proxy cache --- a server that act as a proxy between the client and other servers when the client requests a webpage, file, etc. Web proxies use caches to reduce web traffic, which means they maintain a local copy of recently accessed links, in case they are requested again, and this usually enhances the performance significantly. One of the protocols[33] designed suggests that a web proxy locally keeps a Bloom filter for each of its neighboring servers' cache contents. This way when a proxy is looking for a webpage, it first checks its local cache. If the cache miss occurs locally, the proxy checks all its Bloom filters to see whether any of them contain the desired webpage, and if yes, it tries to fetch the webpage from the neighbor associated with that Bloom filter instead of directly fetching the page from the Web.

Squid implements this functionality and it calls Bloom filters Cache Digests[34] (see Figure 3.4.) Because data is highly dynamic in the network scenario, and Bloom filters are only occasionally broadcasted between proxies, false negatives can arise.

[33] L. Fan, P. Cao, J. Almeida and A. Z. Broder, "Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol," IEEE/ACM Trans. Netw., vol. 8, no. 3, pp. 281-293, 2000.

[34] Squid, "Squid Cache Wiki," [Online]. Available: http://wiki.squid-cache.org/SquidFaq/AboutSquid. [Accessed 19 03 2016].

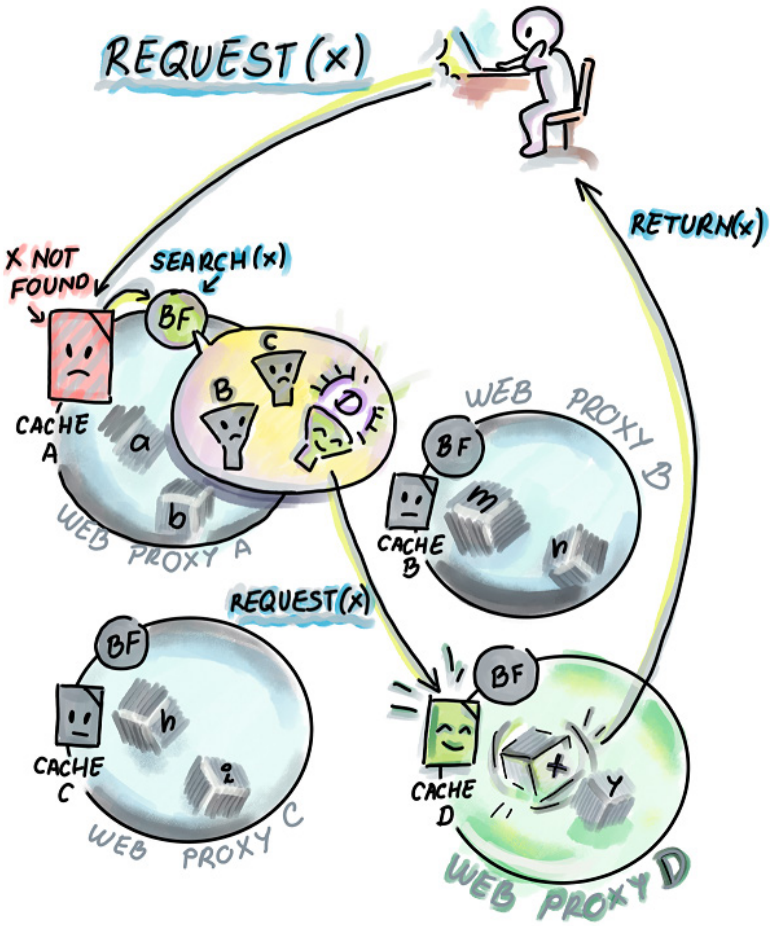

Figure 3.4: Usage of Bloom filter in Squid web proxy. Web proxies keep the copies of recently accessed web pages, but also keep the record of recently accessed web pages of their neighbors by having each proxy occasionally broadcast the Bloom filter of their own cache. In this example, a user requests a web page x, and a web proxy A can not find it in its own cache, so it queries the Bloom filters of B, C and D. The Bloom filter of D reports Found/Present for x, so the request is forwarded to D. Note that, because Bloom filters are not always up-to-date, and the network environment is highly dynamic, by the time we get to the right proxy, the cache might have deleted the resource that we are looking for. Also, false negatives may arise, due to the gap in the broadcasting times.

3.2.2 Bitcoin mobile app

Peer-to-peer networks use Bloom filters to communicate data, and a well-known example of that is Bitcoin. An important feature of Bitcoin is ensuring transparency between clients, i.e., each node should be able to see everyone's transactions. However, for nodes that are

operating from a smartphone or a similar device of limited memory and bandwidth, keeping the copy of all transactions is highly impractical. This is why Bitcoin offers the option of *simplified payment verification* (SPV), where a node can choose to be a *light node* by advertising a list of transactions it is interested in. This is in contrast to full nodes that contain all the data (Figure 3.5):

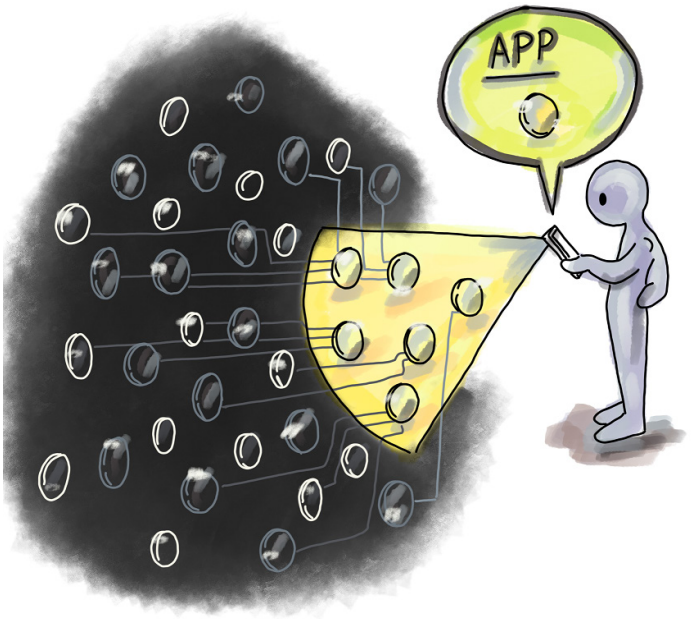

Figure 3.5: In Bitcoin, light clients can broadcast what transactions they are interested in, and thereby block the deluge of updates from the network.

Light nodes compute and transmit a Bloom filter of the list of transactions they are interested in to the full nodes. This way, before a full node sends information about a transaction to the light node, it first checks its Bloom filter to see whether a node is interested in it. If the false positive occurs, the light node can discard the information upon its arrival.[35]

3.3 Configuring a Bloom filter for your application

When using an existing implementation of a Bloom filter, the constructor will allow you to set a couple of parameters and do the rest on its own. For example,

```
bloom = BloomFilter(max_elements, fp_rate)
```

[35] A. Gervais, S. Capkun, G. O. Karame and D. Gruber, "On the privacy provisions of Bloom filters in lightweight bitcoin," in Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC 2014), New Orleans, LA, 2014.

allows the user to set the maximum number of elements and the desired false positive rate, and the constructor does the job of setting other parameters (size of the filter and the number of hash functions). Similarly, we can have:

```
bloom = BloomFilter(fp_rate, bits_per_element)
```

that allows a user to set the desired false positive and how many bits per element they are willing to spend, and the number of elements inserted and the number of hash functions are additionally set, or simply,

```
bloom = BloomFilter(),
```

where the implementation sets parameters to the default values. In the rest of the section, we outline the main formulas relating important parameters of the Bloom filter, which you will need if you decide to implement your own Bloom filter, or to understand how the existing implementations optimally configure the Bloom filter. We will use following notation for the four parameters of the Bloom filter:

- f = the false positive rate
- m = number of bits in a Bloom filter
- n = number of elements to insert
- k = number of hash functions

The formula that determines the false positive rate as a function of other three parameters is as follows (*Formula 1*):

$$f \approx \left(1 - e^{-\frac{nk}{m}}\right)^k$$

First let's reason visually about this formula. Figure 3.6 below shows the plot of f as a function of k for different choices of m/n (bits per element). In many real-life applications, fixing bits-per-element ratio is meaningful because we often have an idea of how many bits we can spend per element. Common values for the bits-per-element ratio are between 6 and 14, and such ratios allow us fairly low false positive rates as shown in the graph below:

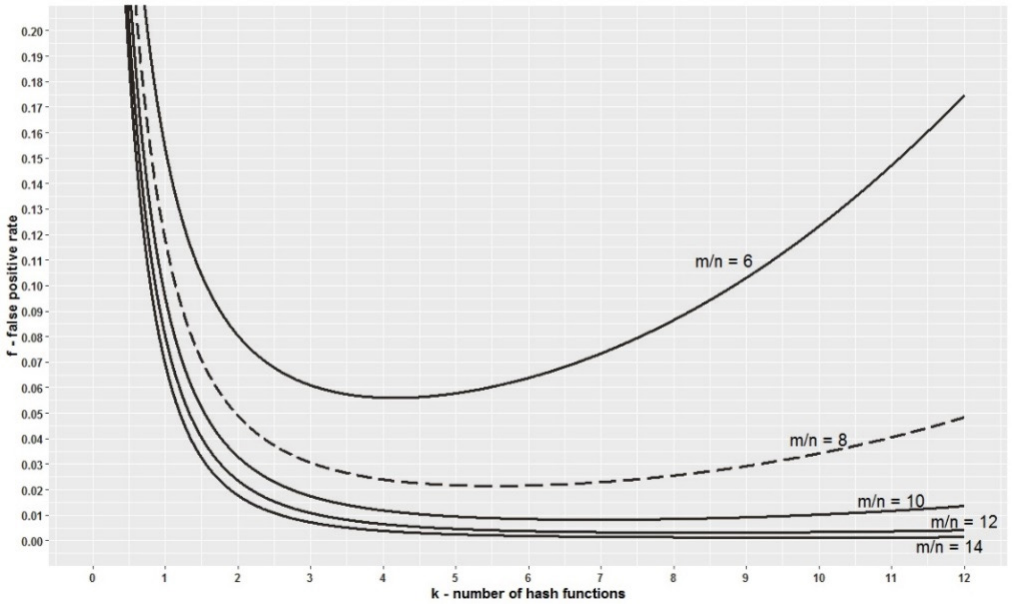

Figure 3.6: The plot relating the number of hash functions (k) and the false positive rate (f) in a Bloom filter. The graph shows the false positive rate for a fixed bits-per-element ratio (m/n), different curves corresponding to different ratios. Starting from the top to bottom, we have m/n=6, 8, 10, 12, 14. As the amount of allowed space per element increases (going from top to bottom), given the same number of hash functions, the false positive rate drops. Also, the curves show the trend that increasing k up until some point (going from left to right), for a fixed m/n, reduces the error, but after some point, increasing k increases the error rate. Note that the curves are fairly smooth, and for example, when m/n=8, i.e., we are willing to spend 1 byte per element, if we use anywhere between 4 and 8 hash functions, the false positive rate will not go above 3%, even though the optimal choice of k is between 5 and 6.

While increasing m or reducing n drops the false positive rate, i.e., more bits per element results in the overall lower false positive curve, the graph also shows the two-fold effect that k has on the false positive: up to some point, increasing k helps reduce the false positive, but from some point on, it worsens it: this is because having more hash functions allows a lookup more chance to find a zero, but also on an insert, sets more bits to 1. The minimum for each curve is that sweet spot that is the optimal k for a particular bits-per-element (which we get by doing a derivative on Formula 1 with respect to k), and it happens at (*Formula 2*):

$$k_{opt} = \frac{m}{n} \ln 2$$

For example, when $m/n = 8$, $k_{opt} = 5.545$. We can use this formula to optimally configure the Bloom filter, and an interesting consequence of choosing parameters this way is that in such a Bloom filter, the false positive rate turns out to be (*Formula 3*):

$$f_{opt} = \left(\frac{1}{2}\right)^k$$

This is particularly convenient, considering that a false positive occurs when a lookup encounters a cell whose value is 1 k times in a row, which means that in an optimally filled Bloom filter the probability of a bit being equal to 1 is ½.

Keep in mind that these calculations assume k is a real number, but our k has to be an integer. So if Formula 2 produces a non-integer, and we need to choose one of the two neighboring integers, then Formula 3 is also not an exact false positive rate anymore. The only correct formula to plug into is Formula 1, but even with Formula 3, we will not make too grave of a mistake. Often it is better to choose the smaller of the two possible values of k, because it reduces the amount of computation we need to do.

3.3.1 Examples

Here we show some examples of how to configure Bloom filters in different situations.

Example 1. Calculating f from m, n, and k

You are trying to analyze the false positive rate of an already existing Bloom filter that has been acting out. The filter capacity is 3MB, and over time it ended up storing 10^7 elements (~ 2.5 bits per element) and it uses 2 hash functions.

Answer for Example 1:

Using Formula 1, we obtain the following:

$$f = \left(1 - e^{-\frac{nk}{m}}\right)^k = \left(1 - e^{-\frac{2*10^7}{3*8*10^6}}\right)^2 = \left(1 - \left(\frac{1}{e}\right)^{\frac{5}{6}}\right)^2 \approx 32\%$$

Example 2. Calculating f and k from n and m

Consider you wish to build a Bloom filter for $n = 10^6$ elements, and you have about 1MB available for it ($m = 8 * 10^6$ bits). Find the optimal false positive rate and determine the number of hash functions.

Answer for Example 2:

From Formula 2, the ideal number of hash functions should be $k \approx 0.693 * 8 * 10^6 / 10^6 = 5.544$. Formula 3 tells us that the false positive rate is $f \approx (1/2)^{5.544} \approx 0.0214$, but we need a legal value of k. In this situation, we might choose $k = 5$ or $k = 6$. In both cases, we will still obtain 2% false positive rate.

Consider re-doing the Example 2 where the dataset becomes 100 times larger, and false positive rate is kept fixed: if we do the math, we will see that we will also require approximately 100 times larger Bloom filter. Therefore, Bloom filters grow linearly with the

size of the dataset and even though they are intended to be a small signature of the original data, they can also grow large enough to spill over to SSD/disk.

3.4 A bit of theory

First let's see where the main formula for the Bloom filter false positive rate (Formula 1) comes from, as Formulas 2 and 3 are the consequence of minimizing f in Formula 1 with respect to k. For this analysis, we assume that hash functions are independent (the results of one hash function do not in any way affect the results of any other hash function) and that each function maps keys uniformly randomly over the range $[0 \ldots m-1]$.

If t is the fraction of bits that are still 0 after all n insertions took place, and k is the number of hash functions, then the probability f of a false positive is:

$$f = (1-t)^k$$

considering that we need to get k 1s in order to report Present. It is impossible to know beforehand what t will be, because it depends on the outcome of hashing, but we can work with *probability* p of a bit being equal to 0 after all inserts took place, i.e.:

$$p = \text{Prob}(a \text{ fixed bit equals } 0 \text{ after } n \text{ inserts})$$

The value of p will in the probabilistic sense, translate to the percentage of 0s in the filter. Now we derive the value of p to be equal the following expression:

$$p = \left(1 - \frac{1}{m}\right)^{nk} \approx e^{-nk/m}$$

To understand why this is true, let's start from the empty Bloom filter. Right after the first hash function h_1 has set one bit to 1, the probability that a fixed bit in the Bloom filter equals 1 is $1/m$, and the probability that it equals 0 is accordingly $1 - 1/m$. After all the hashes of the first insert finished setting bits to 1, the probability that the fixed bit still equals zero is $(1-1/m)^k$, and after we finished inserting the entire dataset of size n, this probability is $(1-1/m)^{nk}$. The approximation $(1-1/x)^x \approx e$ then further gives $p \approx e^{-nk/m}$.

If we just replace t from the earlier expression $f = (1-t)^k$ with our new value of p, we will obtain Formula 1. But to make this replacement kosher, we first have to prove that p is a random variable that is very stably concentrated around its mean, and this can be proved using Chernoff bounds. This means that it is exponentially unlikely that p will differ substantially from t, so it is safe to replace one with the other, thus giving Formula 1.

3.4.1 Can we do better?

Bloom filter packs the space really well but are there, or can there be better data structures? In other words, for the same amount of space, can we achieve a better false positive rate than the Bloom filter? To answer this question, we need to derive a *lower bound* that relates the space in the Bloom filter (m) with the false positive rate (f). This lower bound (available in some more theoretical resources on the subject) tells us that the amount of space the

Bloom filter uses is $1.44x$ away from the minimum. There are, in fact, data structures that are closer to this lower bound than Bloom filter, but some of them are very complex to understand and implement.

3.5 Further reading: Bloom filter adaptations and alternatives

The basic Bloom filter data structure leaves a lot to be desired, and computer scientists have developed various modified versions of Bloom filters that address its various inefficiencies. For example, the standard Bloom filter does not handle deletions. There exists a version of the Bloom filter called *counting Bloom filter*[36] that uses counters instead of individual bits in the cells. The insert operation in the counting Bloom filter increments the respective counters, and the delete operation decrements the corresponding counters. Counting Bloom filters use more space and can also lead to false negatives, when, for example, we repeatedly delete the same element thereby bringing down some other elements' counters to zero.

Another issue with Bloom filters is inability to efficiently resize. One of the problems with resizing in the way we are used to with hash tables, by rehashing and re-inserting, is that in the Bloom filter, we do not store the items nor the fingerprints, so the original keys are effectively lost and rehashing is not an option.

Also, Bloom filters are vulnerable when the queries are not drawn uniformly randomly. Queries in real-life scenarios are rarely uniform random. Instead, many queries follow the Zipfian distribution, where a small number of elements is queried a large number of times, and a large number of elements is queried only once or twice. This pattern of queries can increase our effective false positive rate, if one of our "hot" elements, i.e., the elements queried often, results in the false positive. A modification to the Bloom filter called *weighted Bloom filter*[37] addresses this issue by devoting more hashes to the "hot" elements, thus reducing the chance of the false positive on those elements. There are also new adaptations of Bloom filters that are *adaptive*, i.e., upon the discovery of a false positive, they attempt to correct it.[38]

The other vein of research has been focused on designing data structures functionally similar to the Bloom filter, but their design has been based on particular types of compact hash tables. In the next part, we cover one such interesting data structure: *quotient filter*. Some of the methods employed in the next section will closely tie to the subjects of designing hash tables for massive datasets, the topic of our previous chapter, but we cover it here because the main applications of quotient filters are functionally equivalent to Bloom filters, and find uses in similar contexts.

[36] L. Fan, P. Cao, J. Almeida and A. Z. Broder, "Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol," IEEE/ACM Trans. Netw., vol. 8, no. 3, pp. 281-293, 2000.
[37] J. Bruck, J. Gao and A. (. Jiang, "Weighted Bloom Filter," in IEEE International Symposium on Information Theory, 2006.
[38] M. A. Bender, M. Farach-Colton, M. Goswami, R. Johnson, S. McCauley and S. Singh, "Bloom Filters, Adaptivity, and the Dictionary Problem," in IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS), 2018.

3.6 Quotient filter

Quotient filter[39] offers a number of advantages over classical Bloom filter, such as the ability to delete elements and to resize itself. Also, two quotient filters can be efficiently merged, in a similar fashion to the merge subroutine in merge sort (fingerprints in the quotient filter are sorted). The ability to sequentially scan elements and merge makes the quotient filter a particularly good choice as a building block in larger disk-based data structures, where the difference between the sequential merge of quotient filter and random read/write access pattern of the Bloom filter becomes particularly significant.

Even though only the part of the fingerprint is stored in the quotient filter, the full fingerprint can be recovered using metadata bits, and the false positives can only happen on the level of the same fingerprint/hash. That is, only if two distinct keys generate the same fingerprint, the quotient filter might mistake them one for another. This is in contrast with the Bloom filter, where an element can have a unique set of hashes, but still generate a false positive because some other two elements set its locations to 1. Quotient filter is, however, more complex to implement than a Bloom filter and insert and lookup operations can prove to be more time consuming due to all bit-packing, and especially as the filter becomes more full, just like in the classical linear probing hash table, which quotient filter effectively is.

Next we will describe the design of quotient filter, first by learning what quotienting is, then by describing how quotient filter uses metadata bits together with quotienting to save space. Quotient filter is not the only data structure of this sort, but some of the tricks that you learn here can be generally useful when designing similar space-saving data structures.

3.6.1 Quotienting

Quotienting[40] works differently from most hash tables, because instead of saving the key, it saves its hash, or more precisely, a part of the hash. In a quotienting table, we divide a hash of each item into two parts: *quotient* and a *remainder*. The hash table only stores the remainder. For example, if the hash is 64 bits long, and if the table size is a power of 2, $m=2^i$, then the quotient is i bits long, and the remainder is $64 - i$ bits long. The quotient is used to index into the corresponding bucket of the hash table, where the remainder gets stored. Example from Figure 3.7 shows the hash partition on an example:

[39] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane and E. Zadok, "Don't Thrash: How to Cache Your Hash on Flash," in Proceedings of the VLDB Endowment (PVLDB), Vol. 5, No. 11, pp. 1627-1637 (2012), 2012.

[40] D. E. Knuth, The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching, Addison Wesley Longman Publishing Co., Inc., 1998.

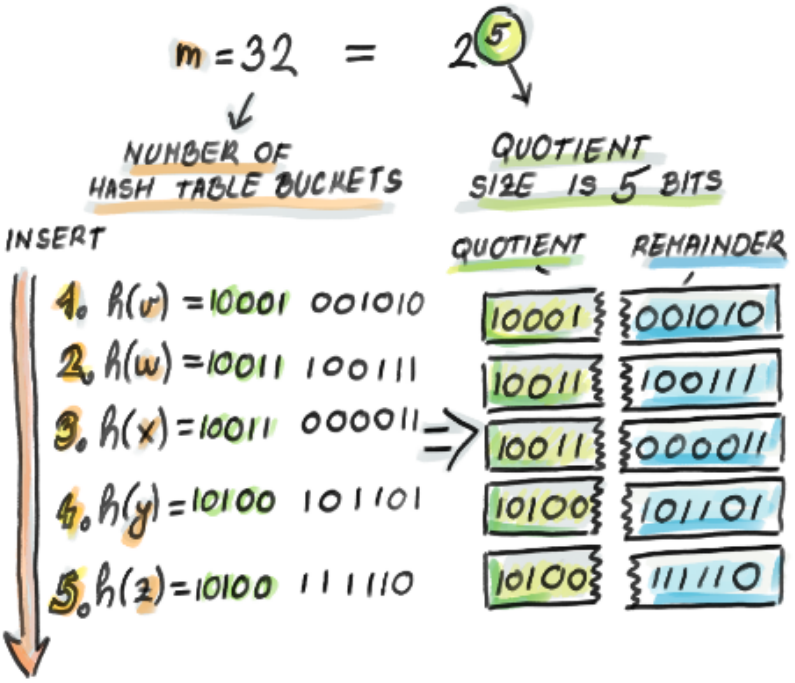

Figure 3.7: Quotienting in a hash table. In this example, the hash table has 32 slots and hashes are 11 bits long. We need 5 bits to distinguish between 32 slots, so the quotient in the hash will be the first 5 bits of a hash, and the remainder will take up the last 6 bits. Quotient determines the bucket at which we will store the remainder. So for example, the item y has the hash 10100 101101, so it will be stored in the bucket 10100 (bucket 20), and the value stored will be 101101 (value 35). This way, instead of storing all 11 bits, we only store 6 bits.

Notice that if we use quotienting in combination with linear probing (or any collision-resolution method where elements can move around the table), we run into trouble. As remainders move down as a result of collisions on the level of quotient, we are losing the association between quotients and remainders, and thus are not able to recover the full hash. (Note that in this sort of hash table, if two items collide on the whole hash, then we consider them the same item, and collisions occur when two hashes have the same quotient. This is applicable for contexts where the hash is sufficiently large that full-hash collisions are highly unlikely, or where some degree of false positive is allowable.)

One way to link quotients to remainders is to use extra metadata bits in each hash table slot that enables the recovery of the original hashes. In the examples shown in Figures 3.8 and 3.9 below, we show how to insert elements and later decode them in the scheme with 3 metadata bits.

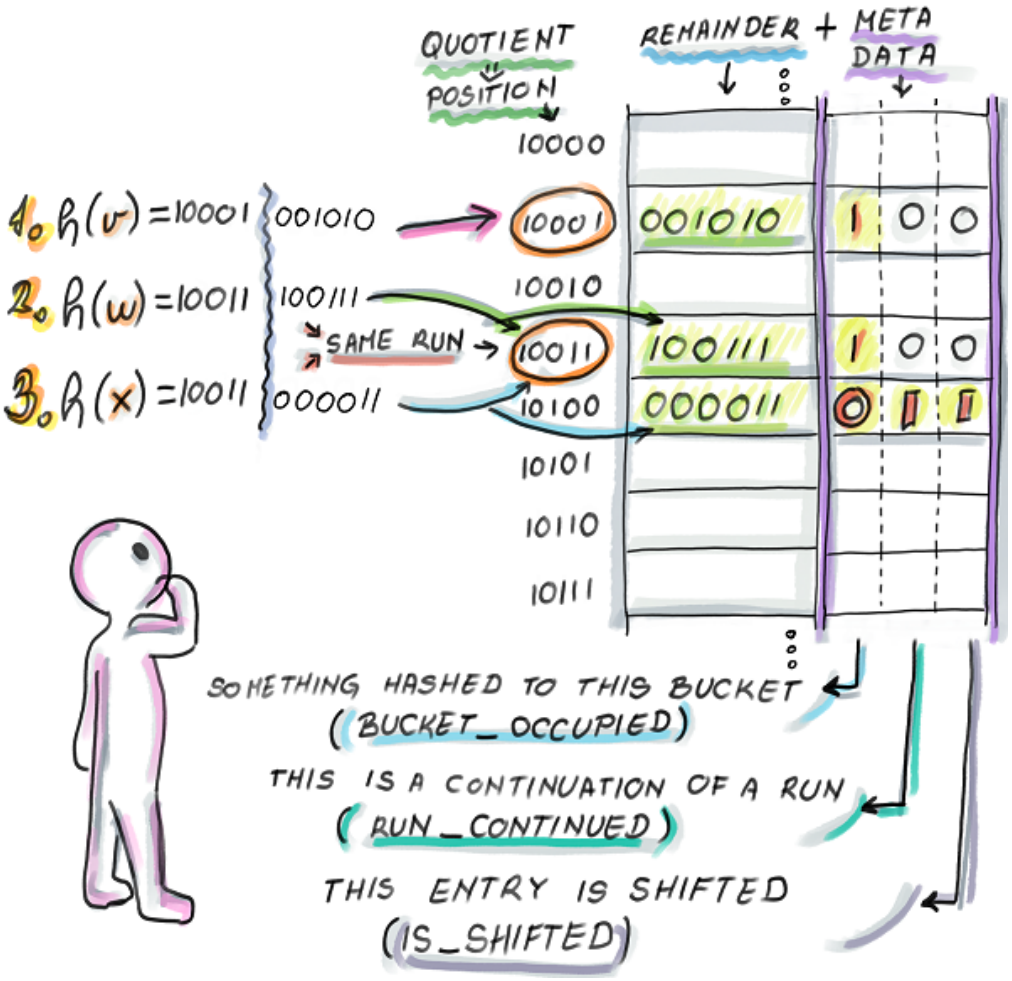

Figure 3.8: Metadata bits in a quotient filter. We inserted three elements into the filter in the following order: v, w and x. First we insert v, and set the bucket_occupied bit to 1, as an item has just hashed in that bucket. We leave the run_continued bit to be 0, because this item is not a continuation of a sequence of items that hashed to the same location. Similarly, we leave the is_shifted bit to be 0 because the item v is currently in its original position – it is not shifted. Then we insert w, and similarly set all identical bits as in the case of v. Then we insert w: because its original slot is taken (and the bucket_occupied bit for its bucket is already set to 1, we continue down the table until the first available slot, which is the slot 10100. We store the remainder in that position, while leaving the bucket_occupied bit of that position set to 0, as nothing hashed there. We also set the run_continued bit to 1, because the item right above x hashed to the same bucket as x, and we set is_shifted to 1, because the item x is not stored in its originally intended position.

The main role of metadata bits is to enable decoding the actual hashes when we do a lookup, for example. The decoding is done on the level of a cluster (a consecutive set of items that are not interrupted by an empty slot):

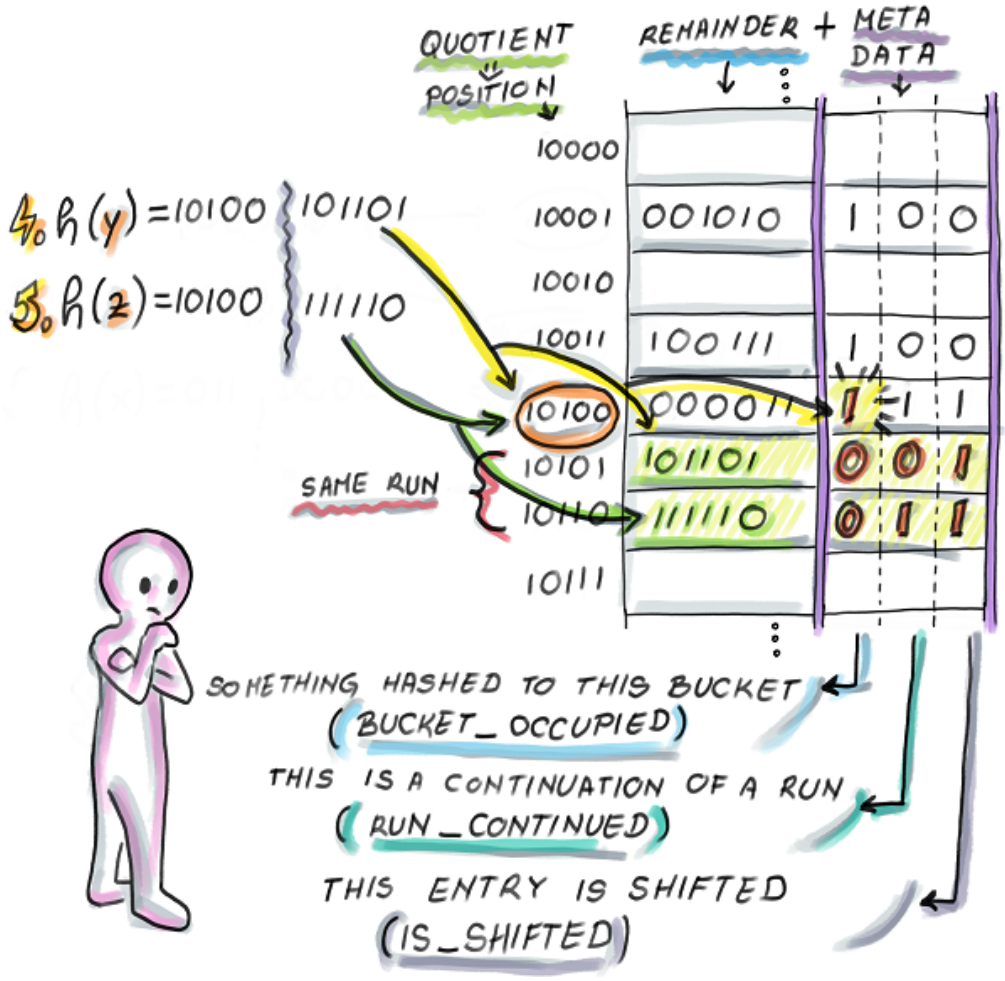

Figure 3.9: Decoding details in quotient filter. When we decode, we work on the level of a cluster. Cluster is a set of items that are not broken up by an empty slot. In this example, we have two clusters: one is just one-slot long, at position 10001, and the other one is 4 slots long, starting from 10011 and ending at 10110. Inside that cluster, we have multiple items that hashed to different positions originally. A consecutive set of items that hashed to the same location originally is called a run. For example items stored at positions 10101 and 10110 belong to the same run because the first one has the run_continued bit as zero, and the second one has it as 1. In this example cluster, items hashed to two distinct buckets, 10011 and 10100 so there are two runs in this cluster. Item at the location 10011 is an anchor — it is a beginning of a cluster, so this item is in its

own original position (we can reconstruct this item as 10011 100111), but the item right below it is a continuation of the same run and is shifted (thus we can reconstruct it as 10011 000011). Similarly for two items below, the item at the slot 10101 is, even though shifted from where it originally hashed, not a continuation of the run — it is the beginning of its own run (we can reconstruct it as 10100 101101 – our y), but the second is the continuation (so we can reconstruct it as 10100 111110 – our z).

3.6.2 Resizing

One particular advantage of quotienting is that it allows us to do resize operation on the hash table without having to do an expensive rehash operation on all the items. If we want to double the table, we steal one bit from remainder and give it to the quotient, and vice versa (example in Figure 3.10).

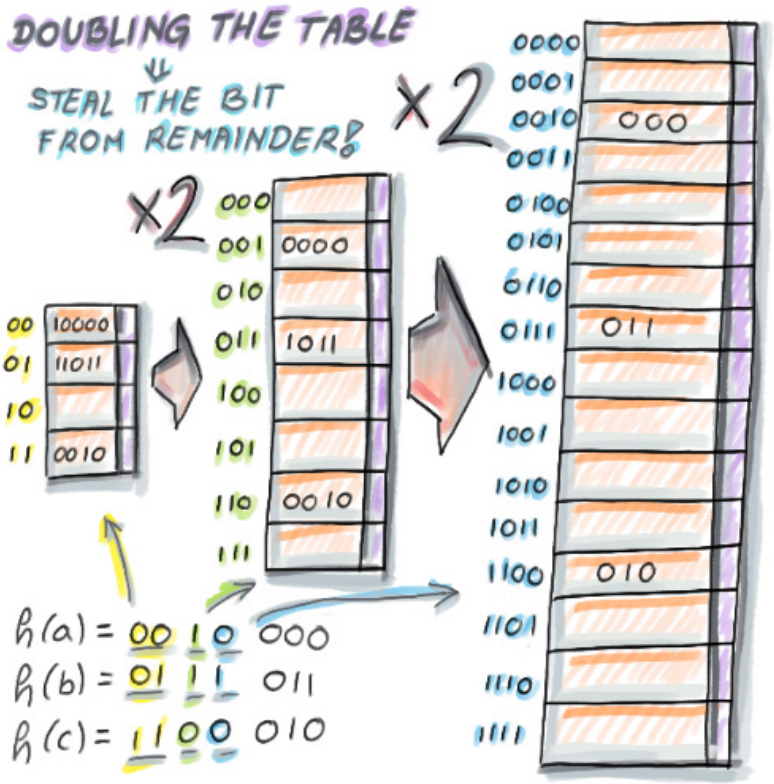

Figure 3.10: Resizing with quotienting. To double the existing table, we steal one bit from the remainder and give it to the quotient. In this example, the original table had 4 slots, and the quotient is 2 bits while the remainder is 5 bits (entire hash is 7 bits long). To double the size of the table, we steal one bit from the remainder and give it to the quotient, so an item that was before stored in the bucket 01 with the remainder 11011, is in the second table stored in the bucket 011 with the remainder 1011, and when the table is again doubled, it is stored in the bucket 0111 with the remainder 011.

There is a number of different tricks and methods one can use to make the quotient-filter-type data structure more space-efficient. The scheme presented above can also be implemented with only two metadata bits, but that substantially complicates the decoding step, making common operations too CPU-intensive on longer clusters. Another similar data structure based on a different collision-resolution technique, cuckoo hashing, is called a *cuckoo filter*. One of the key advantage of cuckoo filter in the comparison to the Bloom filter is the fast lookup: even on a successful lookup, the data structure needs at most 2 random reads while the Bloom filter might need up to k.

3.7 Summary

- Bloom filters have been widely applied in the context of distributed databases, networks, bioinformatics, and other domains where regular hash tables are too space-consuming.
- Bloom filters trade accuracy for the savings in space, and there is a relationship between the space, false positive rate, the number of elements and the number of hash functions in the Bloom filter.
- Bloom filters do not meet the space vs. accuracy lower bound, but they are simpler to implement than more space-efficient alternatives, and have been adapted over time to deal with deletes, different query distributions, etc.
- Quotient filters are based on compact hash tables and are functionally equivalent to Bloom filters, with the benefit of the cache-efficient operations, and ability to delete, merge and resize.
- Cuckoo filters are based on cuckoo hash tables, and promise the lookup of $O(1)$. Just like quotient filters, they store fingerprints instead of the actual keys.

4

Frequency Estimation and Count-Min Sketch

This chapter covers:

- Understanding the streaming model and its constraints
- Exploring practical use cases where frequency estimates arise and how count-min sketch can help
- Learning how count-min sketch works
- Exploring the error in count-min sketch and configuring the data structure
- Understanding how range queries can be solved with count-min sketch
- Exploring heavy hitters and approximate heavy hitters with count-min sketch

Measuring frequency is one of the most common operations in today's data-intensive applications. Any kind of popularity analysis on a massive-data application, such as producing the bestseller list on Amazon.com, computing top-k trending queries on Google, or monitoring the most frequent source-destination IP address pairs on the network are all frequency estimation problems. Estimating frequency also shows up when monitoring changes in systems that are awake 24/7, such as sensor networks or surveillance cameras. Here we can observe changes in parameters such as the temperature or location change of a sensor in the ocean, new object appearance in the frame, or the number of units by which a stock on the stock market rose or fell in a given time interval. For this purpose, in the next section, we introduce the streaming model of data that emphasizes challenges related to this particular setup.

The amount of space required to exactly measure frequency is related to the number of distinct items in the dataset (n), not the entire quantity of the dataset (N). So for example, if on Amazon.com, we only have a couple of slam-dunk bestsellers that make up all the sales,

storing all distinct bestsellers and their sales numbers is not particularly challenging from the space point of view (it can even be done in $O(1)$. On the other hand, selling a small number of copies of each book requires a lot of space to store ($O(N)$, as in this case $n = O(N)$), but in many datasets with duplicates, we will find neither of the two scenarios to be the case. Real datasets exhibit certain commonalities regardless of the domain in which they appear, and they tend to contain a small number of items with very high frequencies, *and* also a large number of items with small frequencies, which in our Amazon.com example corresponds to having few slam-dunk bestsellers, and many books that get sold only a couple of times. This combination of quantity and diversity of items is challenging from the storage point of view.

In this chapter, we will learn how to solve popularity problems of interest such as top-k queries, heavy hitters and frequency range queries under the constraint of limited space and time. We will see that with the limitations of the streaming model, many problems that had rather trivial solutions before can now only be solved approximately, yet with count-min sketch, we can achieve enormous space savings and not lose a lot of accuracy.

To that end, we will learn about Count-Min sketch. Count-min sketch has been devised by Cormode and Muthukrishnan in 2005[41] and can be thought of as a young, up-and-coming cousin of Bloom filter. Similarly to how Bloom filter answers membership queries approximately with less space than hash tables, the count-min sketch estimates *frequencies* of items in less space than a hash table or any linear-space key-value dictionary. Another important similarity is that the count-min sketch is hashing-based, so we continue in the vein of using hashing to create compact and approximate sketches of data. But as we will see in this chapter, count-min sketch is a different animal than Bloom filter, mainly due to the contexts in which its main task of estimating frequency arises.

The rest of the chapter outline is as follows: first we introduce some basic details of the streaming model. Then we introduce the count-min sketch data structure, show how it works, and we follow-up with a number of practical scenarios involving sensors and natural language processing applications. Lastly, we show how count-min sketch can be used to solve problems involving range queries and approximate heavy hitters.

4.1 Streaming data

Streaming data is definitely big, but not all big data is streaming. More and more applications nowadays produce and process data at rapid rates, and in an unpredictable and volatile fashion. We may visualize streams as never-ending sequences of data and huge datasets made up of many tiny pieces; most of the time, we are not particularly interested in the tiny pieces per se: "What was the exact temperature recorded by the sensor ID 1092 at 11:34pm on May 15, 2003?" sounds like a question someone might only ask in court. And for such purposes, data is stored in the archival storage. But what we care about on a daily basis is the imperfect big picture that is reported real-time for the users. This setup stands in contrast from how we are used to thinking of traditional databases that take great pride in

[41] G. Cormode and S. Muthukrishnan, "An Improved Data Stream Summary: The Count-Min Sketch and Its Applications," Journal of Algorithms, vol. 55, no. 1, p. 58–75, 2005.

providing perfect accuracy but on their own clock. The figure below[42] is a rough depiction of the streaming model:

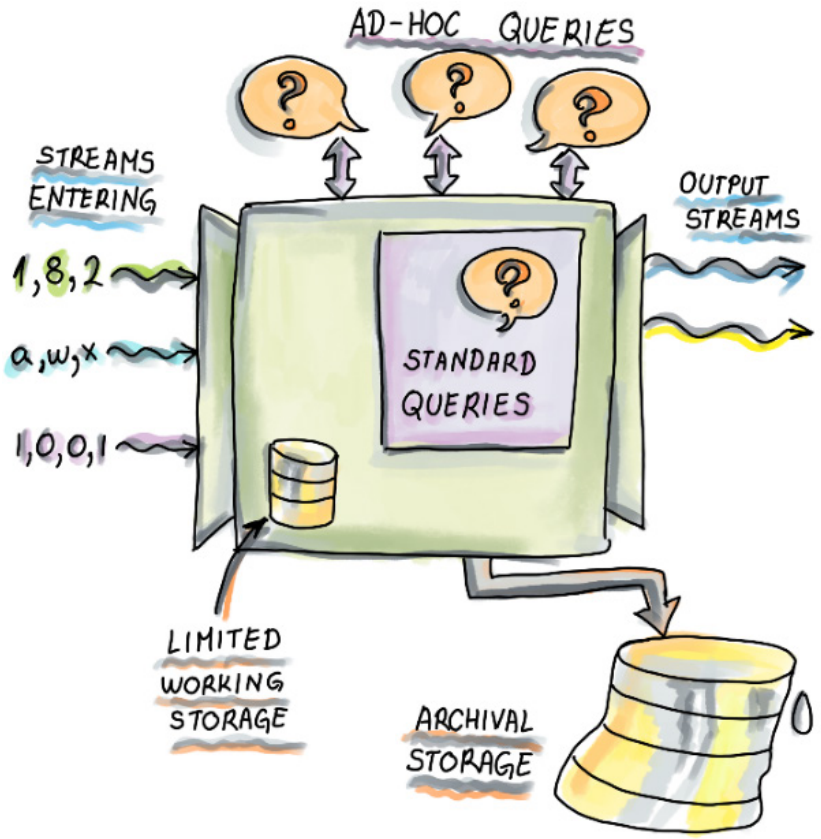

Figure 4.1: Streaming model. The streaming model differs from the traditional database management system in that data passes through the processor and a small amount of working storage, and it is either never stored, or it is stored into the archival storage that is usually too large and slow to be indexed and searched. Items can be found there but we should not count on doing it often and quickly. All the real-time analysis is done on-the-fly. There are standard (or standing) queries, ones that need to be computed all the time, and ad-hoc queries, that show up at unexpected times and their content is externally controlled.

Generally speaking, once a piece of data has passed through our processor and working memory, we should not expect to see it ever again. This effectively makes most algorithms in this model one-pass, and most data structures sublinear because the working storage is insufficient to store all data.

[42] Partly adopted from A. Rajaraman and J. D. Ullman, Mining of Massive Datasets, Cambridge University Press, 2011.

The rise of streaming applications is what has led to the development of the large number of sketches, sampling methods and one-pass algorithms that can make sense of the galloping data. What makes streaming context specific is that we are limited both by time and space, and in the next sections, we will see how count-min sketch can help solve many of the problems we care about when analyzing massive datasets in such a challenging context.

4.2 Count-min sketch: how it works

Count-min sketch (CMS) supports two main operations: update, the equivalent of insert, and estimate, the equivalent of lookup. For the input pair (a_t,c_t) at timeslot t, update increases the frequency of an item a_t by the quantity c_t (if in a particular application c_t =1, that is, the counts do not make particular sense, we can override update to just use a_t as an argument). Estimate operation returns the frequency estimate of a_t. The returned estimate can be an overestimate of the actual frequency, but never an underestimate (and that is not an accidental similarity with the Bloom filter feature of false positives but no false negatives.)

Count-min sketch is represented a matrix of integer counters with d rows and w columns (`CMS[1..d][1..w]`), and d independent hash functions h_1, h_2,..., h_d, each with range $[1..w]$, where the j^{th} hash function is dedicated to the jth row of the CMS matrix, $1 \leq j \leq d$. In the count-min sketch, all counters are originally initialized to 0.

4.2.1 Update

Update operation adds another instance (or a couple of instances) of an item to the dataset. Using d hash functions, update computes d hashes on a_t, and for each hash value h_j (a_t), $1 \leq j \leq d$, that position in the j^{th} row is incremented by c_t (pseudocode shown below:)

```
CMS_UPDATE(at,ct):
for j ← 1 to d
       CMS[j][hj(at)] = CMS[j][hj(at)] + ct
```

An example of how update works is shown in the figure below:

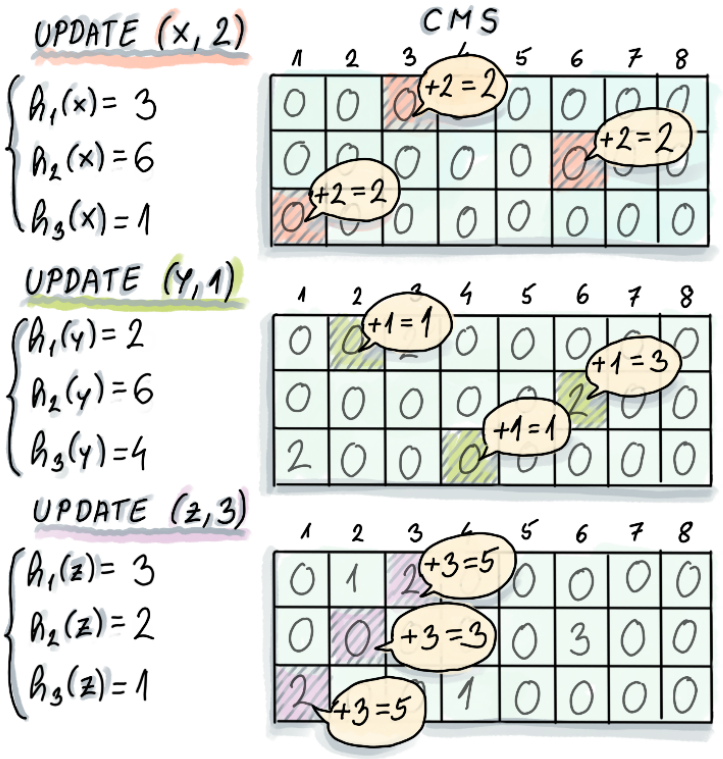

Figure 4.2: Three UPDATE operations of x, y and z performed on an initially empty CMS of dimensions 3x8. Accordingly, we have 3 hash functions that are computed on each element, and the resulting hash is the index into the cell number of the appropriate row. For example, $h_1(x) = 3$, so the location $\mathrm{CMS}[1][3]$ is being incremented by 2 during the update of x.

4.2.2 Estimate

Estimate operation reports the approximate frequency of the queried item. Just like update, estimate also computes d hashes, and it returns the minimum among d counters in d different rows, where the counter location in the j^{th} row is specified by hash

$h_j(a_t)$, $1 \le j \le d$ (pseudocode below):

```
CMS_ESTIMATE(aₜ):
min = INT_MAX
for j ← 1 to d
       if(CMS[j][hⱼ(aₜ)] < min)
       min = CMS[j][hⱼ(aₜ)]
return min
```

An example of how estimate works is shown in Figure 4.3 below. As we can see, count-min sketch can overestimate the actual frequency of an item when, during updates for different items, hashes collide, but the overestimate only happens if there was a collision in each row.

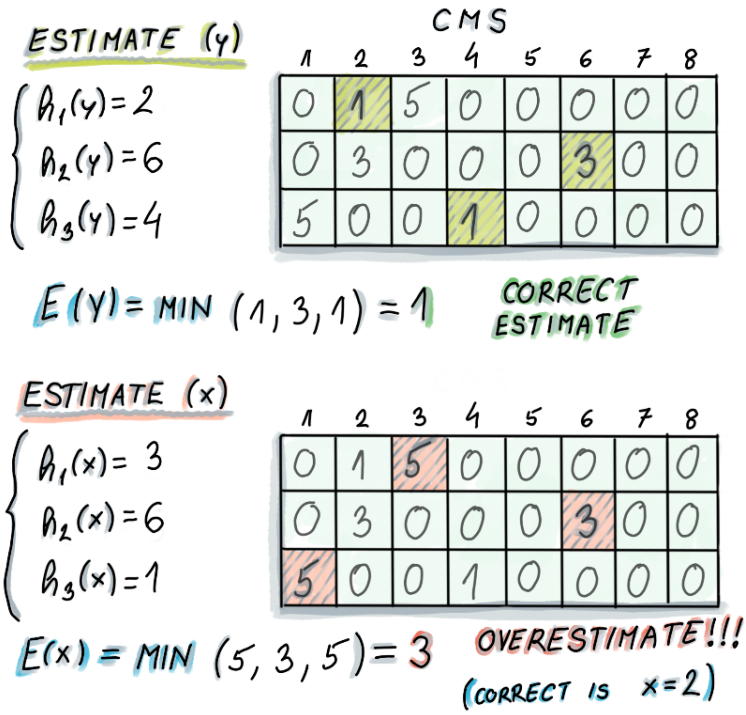

Figure 4.3: Example of estimate operations on the count-min sketch from Figure 4.2. In the case of element y, whose true frequency is 1, count-min sketch reports the correct answer of 1 (the minimum of 1, 3 and 1). However, in the case of the element x, whose true frequency is 2, count-min sketch reports 3 (the minimum of 5, 3 and 5). Refer to the Figure 3.2 to convince yourself that during earlier update operations, y and z together incremented all the counters that are used by x, thus resulting in an overestimate for x.

4.2.3 Space and error in count-min sketch

Count-min sketch exhibits two types of errors: ε (epsilon) that regulates the band of overestimate error, and δ (delta), the failure probability. For a stream S that has come up to the timeslot t, $S=(a_1, c_1), (a_2, c_2), \ldots, (a_t, c_t)$, if we define N as the total sum of frequencies observed in the stream $N = \sum_{i=1}^{t} c_t$, then the overestimate error ε can be expressed as the percentage of N by which we can overshoot the actual frequency of any item. In other words, for an element x and its true frequency f_x, count-min sketch estimates the frequency as f_{est}:

$$f_x \leq f_{est} \leq f_x + \varepsilon N$$

with probability at least $1-\delta$. Usually δ is set to be small (e.g., 0.01) so that we can count on the overestimate error to stay in the promised band with high probability. In other words, there is a small probability δ that the overestimate in CMS can be unbounded.

Example 1.

Given N=10^8, $\varepsilon = 10^{-6}$ and δ=0.1, determine the error properties of the count-min sketch.

Solution for Example 1.

CMS estimates of items' frequencies will overshoot no more than 100 above the actual frequency in at least 90% of the cases.

Just like with Bloom filter, we can tune CMS to be more accurate but that will cost us space. Whatever are the (ε,δ) values that we desire for our application, in order to achieve the bounds stated above, we need to configure the dimensions of count-min sketch to $w=e/\varepsilon$ and $d=\log(1/\delta)$. This way we can achieve the error bounds from above, and the space required by count-min sketch, expressed in the number of counters will then be (Formula 1):

$$O\left(\frac{e\log\left(\frac{1}{\delta}\right)}{\varepsilon}\right)$$

Example 2.

Calculate the space requirements for the count-min sketch from Example 1.

Solution for Example 2.

Applying Formula 1 to our count-min sketch from the example above, we find that it will need ~2.7 million counters, and with 4-byte counters, we need only about 11MB. We could also get away with 3-byte counters and an 8MB count-min sketch, as the maximum number of bits required to store the value N is 24.

Note that CMS tends to be really small even when used on large datasets. In many resources, you will find that CMS miraculously does not depend on the size of the dataset that it is used for and in some sense that is true: after all, our expression for the required space does not have N in it. This is the case if you think of the error band as a fixed percentage of the dataset size: for example if you want to keep your allowed band of error fixed at 2% of N whatever your N is, then increasing N does not require a larger CMS to guarantee the same bounds. This way of looking at the error makes sense in many applications, where with larger data, we are willing to tolerate larger error.

Something to think about - 1.

As an experiment, consider what happens with the size (and the shape) of count-min sketch if we desire a fixed constant error (εN). For example, say we want to keep the overestimate at 100 or less like in the example above, but for a twice as big N.

Something to think about - 2.

Can you design two count-min sketches that consume the same amount of space but have very different performance characteristics (with respect to their errors). Can you think of particular types of applications each of the two designs would be good for?

By now, you can conclude that the width in the count-min sketch seems to be related to the band of the error ε, and the depth is related to the failure probability δ, but what is the intuition behind that? Without doing the actual proof, it is hard to see exactly what happens, but the main idea is that stretching the CMS width will reduce how much different elements' hashes collide within any one row on average, but collisions will still be likely to happen a lot. The depth allows us to reduce that probability because for the overestimate to happen, we require each row to have an overestimate in a corresponding cell. So for instance, if the chance of going outside the allowed band of error in a given cell in any one row is at most $1/2$, then with d rows, the chance of overestimate for an element will be at most $(1/2)^d$, substantially smaller.

4.3 Use cases

Now we move onto practical applications of count-min sketch in two different domains: a sensor smart-bed application, and a natural-language-processing (NLP) application.

4.3.1 Top-k restless sleepers

Science of sleep is a big thing these days (one might say that people are losing sleep over the quality of their sleep.) The invention of smart beds that come equipped with dozens of sensors capable of recording different parameters such as movement, pressure, temperature and so on offers new opportunities to analyze people's sleep patterns and cater to individual sleeper's needs. Based on the data, the bed components can be pulled up (e.g., to help with snoring), warmed up, cooled down, etc. Consider a smart-bed company that collects data at one central database and now that there are millions of users and sensors send out data every second, the amount of data is quickly becoming too large to process and analyze in a straightforward manner. Over the course of one day only, our hypothetical company collects a total of $10^8 \text{ (customers)} * 3{,}600 \text{ (seconds per hour)} * 24 \text{ (hours per day)} * 100 \text{ (sensors)} = 8.6 * 10^{14}$ tuples of data, resulting in terabytes of storage on a daily basis (the specific example is hypothetical, but the size of the collected data and the related problem we study is not.)

One of the new features in the focus of our company's *SleepQuality* app is analyzing restlessness in sleepers, where we can envision each sleeper being mapped somewhere on the quality-of-sleep scale. To notify the customers with the most erratic sleeping patterns, the app also maintains a top-list of most restless sleepers.

We can create a hash table that holds a separate entry for each user along with an integer that keeps track of their quality of sleep data, but that will result in an enormous hash table that needs many gigabytes. Also, if we wished to perform a more in-depth analysis by separately storing the movement information coming from different sensors of

the same bed, that would result in 10 billion distinct `(user-id, sensor)` pairs. Instead of building an enormous hash table, we will build a count-min sketch.

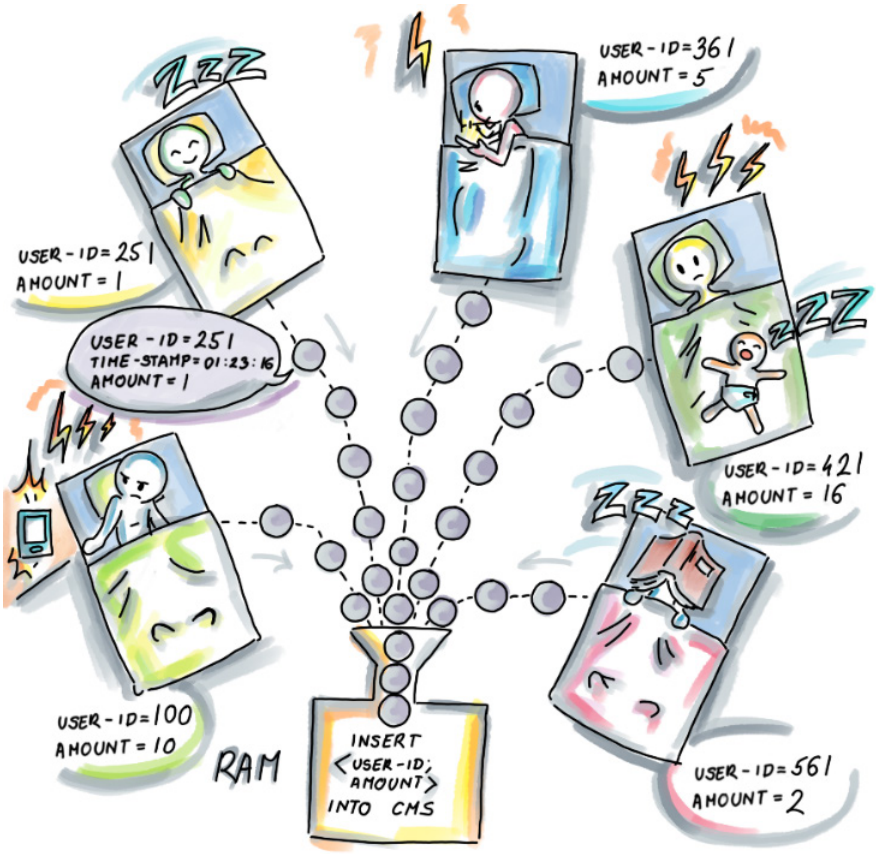

Figure 4.4: All sleep data is sent to a central archive, but before that, input into count-min sketch residing in RAM for later analysis. The (user-id, amount) pair is used as the input for the data structure, where the frequency of user-id increases by amount.

As shown in the Figure 4.4, data arrives at a frequent rate from each sleeper, and at every timestep, the `(user-id, amount)` pair updates the count-min sketch. The frequency for the given user is updated by a given amount. The count-min sketch will at all times then be available to produce approximate estimates for any user who requests it. But in order to maintain the list of top-k restless sleepers, we will have to do a little bit more than just updating the count-min sketch. Remember that the count-min sketch does not maintain any information on different user-ids, it is just a matrix of counters. So we can query it, but we need to know the user-id, or we have to store the important ones somewhere.

Something to think about – 3.

Before moving onto the solution, think what could be the right data structure to help with storing the top-k restless sleepers in a space-efficient manner.

One solution is to use a min-heap, as shown in the Figure 4.5 below:

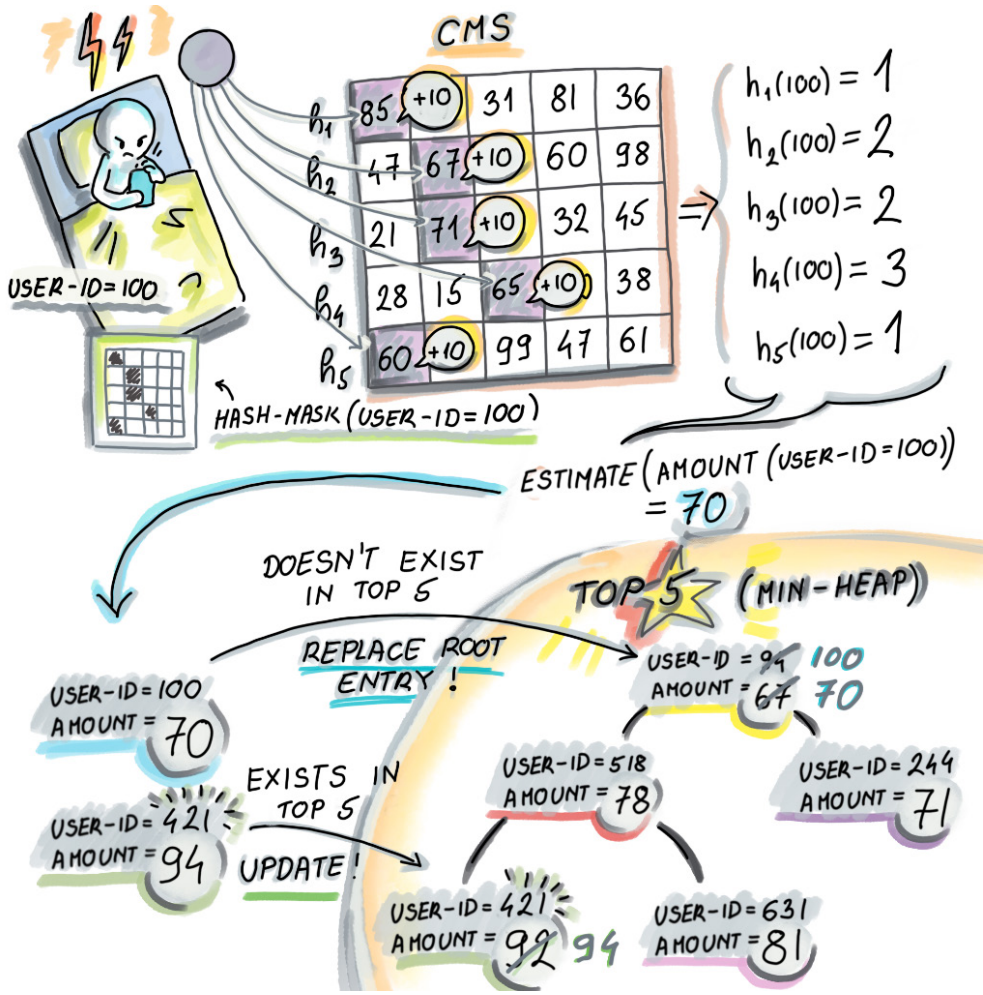

Figure 4.5: Every time the count-min sketch is updated with a (user-id, amount) pair, like with (100, 10) in this example, in order to maintain a correct list of top-k restless sleepers, we do an estimate on frequency of the recently updated user-id. In our case, the estimate for user-id 100 will be 70. Then, if the user-id is not present in the min-heap and it has a higher value than the min (as it does in our example), we will extract the min and insert the new (user-id, amount) pair into the min-heap. If the pair was already present, its amount needs to be updated by deleting and re-inserting the pair with the new updated (higher) amount.

Sensor sleep data is an example of a dataset where the big picture is more important than individual data points. In addition to producing the list of the most restless sleepers, we could do a more refined analysis of the most active sensors for a particular sleeper by having a mini min-heap per sleeper of interest, where the unique identifier would be a combination of user–id and the sensor-id.

Next we will see how count-min sketch is used in NLP.

4.3.2 Scaling distributional similarity of words

The *distributional similarity* problem asks that, given a large text corpus, we find pairs of words that might be similar in meaning based on the contexts in which they appear, or as a well-known linguist John R. Firth put it: "You will know a word by a company it keeps". So for example, the words 'kayak' and 'canoe' will appear surrounded by similar words like 'water', 'sport', 'weather', 'equipment', etc. As a context for a given word, usually the window of size k (e.g., $k = 3$) is chosen, including k words before and k words after the given word, or less if we are on the boundary of a sentence.

One way to measure distributional similarity for a given word-context pair is *pointwise mutual information (PMI)*[43]. The formula for PMI for words A and B is as follows:

$$PMI(A,B) = \log_2 \frac{Prob(A \cap B)}{Prob(A)Prob(B)}$$

Where $Prob(A)$ denotes the probability of occurrence of A, that is, the number of occurrences of A in corpus divided by the total number of words in the corpus. The intuition behind this formula is that it measures how likely A and B are to occur close to each other in our corpus (enumerator) in comparison to how often they would co-occur if they were independent (denominator). The higher the PMI, the more similar the words are. Typically, to compute the PMIs for all word-context pairs or the particular word-context pairs of interest, we would preprocess the corpus to produce the following type of matrix:

[43] D. Jurafsky and J. H. Martin, Speech and language processing (2nd edition), Upper Saddle River, N.J.: Pearson Prentice Hall, 2009.

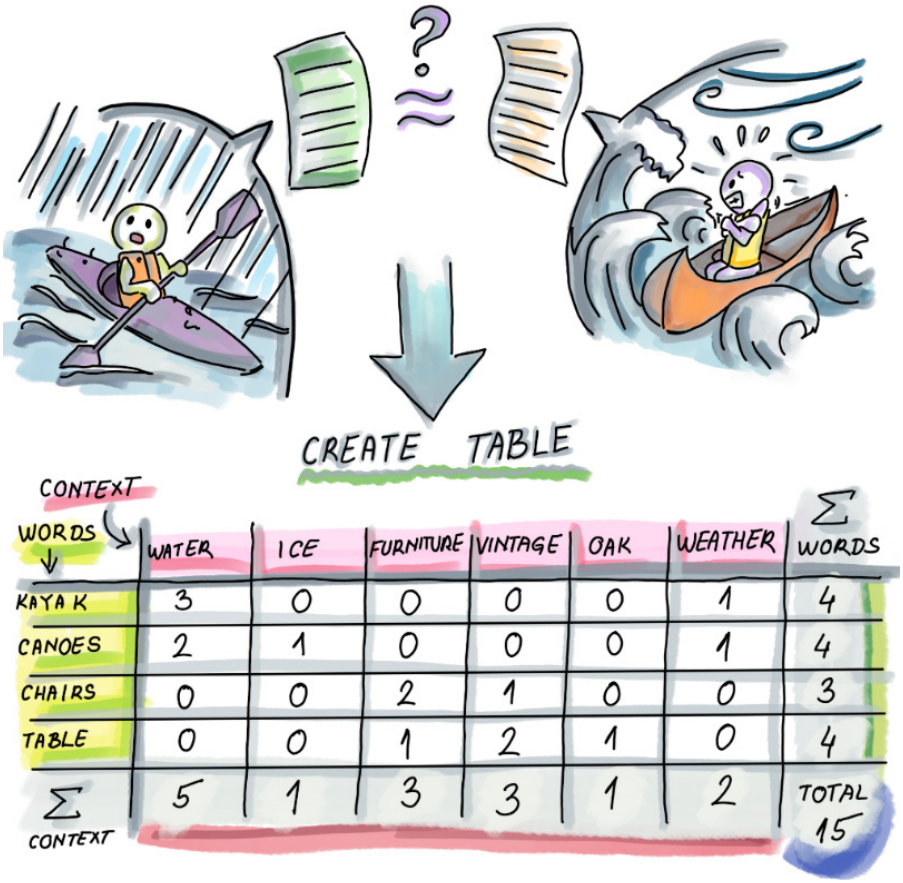

Figure 4.6: To preprocess the text corpus for computing PMI, one way is to create a matrix M where the entry M[A][B] contains the number of times the word A appears in the context B. So for example, 'kayak' appears 3 times in the context of 'water' and 0 times in the context of 'furniture'. We also produce the additional count for each word (the last column of the matrix), and count for each context (the last row in the matrix), as well as the total number of words (lower right corner).

For better association scores between the words, the more text we use, the better, but with the larger corpus, even if the number of distinct words is fairly reasonable in size, the number of word-context pairs quickly gets out of hand.

For example, authors of a paper that analyzes sketch techniques in NLP, and who analyze distributional similarity[44] use the Gigaword dataset obtained from English text news sources with 9.8GB of text, and about 56 million sentences. This results in having 3.35 billion word-

[44] A. Goyal, H. Daume III and G. Cormode, "Sketch Algorithms for Estimating Point Queries in NLP," in Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, Jeju Island, Korea, 2012.

context pair tokens, and 215 million unique word-context pairs, and just storing those pairs with their counts takes 4.6GB.

Using count-min sketch in this particular example of Gigaword, the space savings achieved were over a factor of 100. The solution they employ is to transform the matrix such that the word-context pair frequencies are stored in the count-min sketch, and because the number of distinct words is not too large, we can afford to store words and their counts in their own hash table (last column of the matrix), and the contexts and their counts in their own hash table (the last row of the matrix). The transformation can be seen in the Figure 4.7 below:

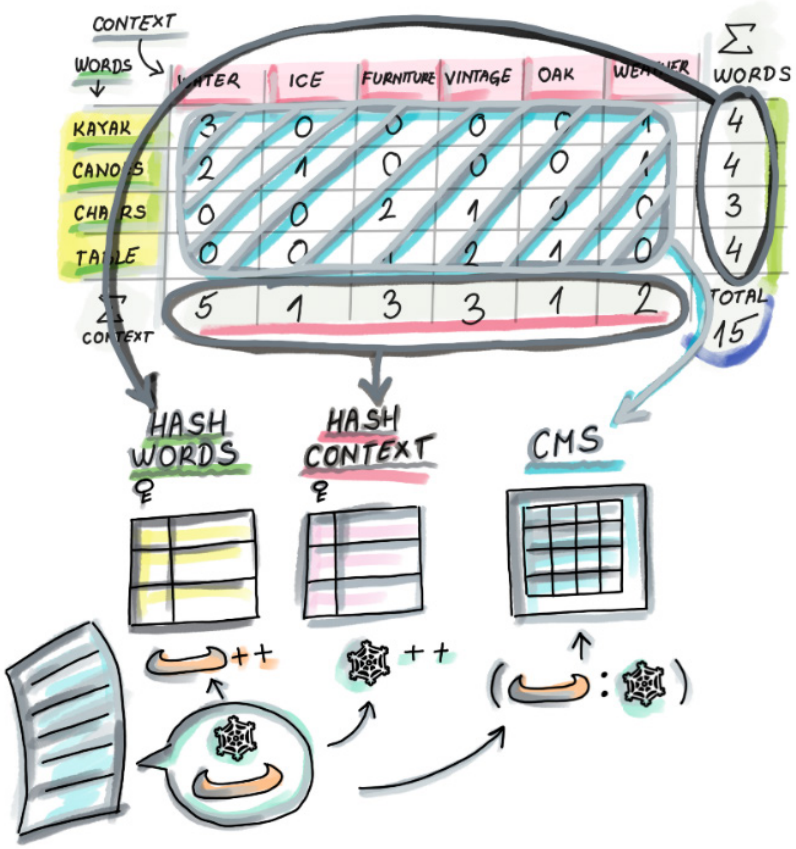

Figure 4.7: The transformation of the matrix from Figure 4.6 to save space: the word-context pairs stored in the main body of the matrix are replaced by a count-min sketch that stores frequencies of word-context pairs. Because the number of distinct words (and contexts) is not that large, we can store each in their own hash table with the appropriate counts. In other words, when we encounter a new pair (word, context), we increment the count of the pair in the CMS, and also increment respective counts in the word hash table and the context hash table. To calculate the PMI for a word-context pair, we do the estimate query on the count-min sketch, and find the appropriate counts of the word and the context in the respective hash tables.

The authors of this research report that a 40MB sketch gives results comparable to other methods that compute distributional similarity using much more space. Producing this count-min sketch and the two hash table takes only one pass over preprocessed and cleaned data, which is a big plus for the streaming datasets. We could produce top-k PMIs with an additional sweep of the data.

Keep in mind that we can not in the same straightforward manner apply the heap solution from the sensor example, because with the changing word and context counts as we go through the text, the PMIs change all the time and not only for the words appearing in the (word, context) pair that was just updated. This means that we would need to do a lot of work on the heap after every update operation if we wanted to avoid doing another sweep of data.

4.4 Range queries with count-min sketch

In this section, we will learn how to answer frequency estimates for ranges as oppose to single points. Range reporting is frequent in databases, as people often think in terms of categories, groups and classifications, which naturally translates into questions about ranges, such as: give me all employees that have worked for the company between a and b years, or who have salaries between x and y. Time series are another example of ranges, for example: How many books were sold on Amazon.com between December 20th and January 10th?

Balanced binary search trees are an example of a good data structure for navigating ranges, as the items are ordered in the lexicographical order so the cost of the range query, after the initial point search, is proportional to the cost of reporting the points in the range --- the minimum possible; this is in contrast to hash tables that scatter data all over the place and querying for a range requires a full scan of the table, even if zero items are reported. As you might imagine, that does not paint a promising picture for exploring ranges using our hash-based sketches.

By now, you can conclude that the width in the count-min sketch seems to be related to the band of the error ε, and the depth is related to the failure probability δ, but what is the intuition behind that? Without doing the actual proof, it is hard to see exactly what happens, but the main idea is that stretching the CMS width will reduce how much different elements' hashes collide within any one row on average, but collisions will still be likely to happen a lot. The depth allows us to reduce that probability because for the overestimate to happen, we require each row to have an overestimate in a corresponding cell. So for instance, if the chance of going outside the allowed band of error in a given cell in any one row is at most $1/2$, then with d rows, the chance of overestimate for an element will be at most $(1/2)^d$, substantially smaller.

One straightforward way to employ the count-min sketch to answer frequency estimates on ranges is to turn the range $[x,y]$ into $y - x + 1$ point queries, assuming only integers are valid data points. Other than the query time growing linearly with the size of the range, the error would also increase linearly with the size of the range, so instead of promising the overestimate of at most εN with probability at least $1 - \delta$, we can only promise at most

$(y - x + 1)\varepsilon N$, so if we expect an overestimate of at most 7 per point query, a range of size 10,000 could produce an overestimate of up to 70,000, which for big ranges deems the data structure useless.

It turns out that we can get tighter frequency estimates by using count-min sketch, that is, a couple of them together, in a creative way[45,46]. The main idea is that instead of dividing the range into unit ranges, we will divide the range into so-called *dyadic ranges*. Dyadic ranges are always the size of a power of two, and if we have a complete universe interval as $I = [1, n]$, we define a collection of dyadic ranges at different levels: dyadic ranges of level i, $0 \leq I \leq \log_2 n$, are of length 2^i, starting at the beginning at the whole interval and can be expressed as $[j * 2^i + 1, (j + 1)2^i]$, where $0 \leq j \leq n/2^i - 1$. Specifically, let's say we are analyzing sales over a 16-day period. Then we can divide the universe interval $I = [1, 16]$ into dyadic ranges in the following way:

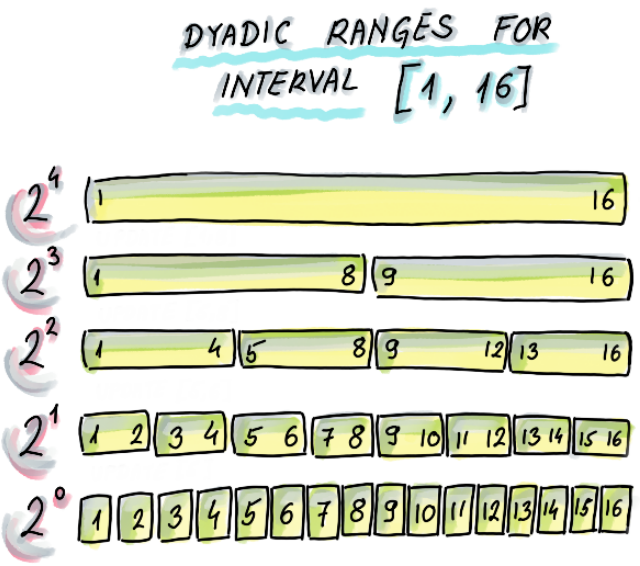

Figure 4.8 Dyadic ranges for the interval [1, 16]. Dyadic ranges of level 0 are the bottom-most with ranges of size 1, then the ranges of level 1 are the level above with the ranges of size 2, and in general dyadic ranges at level i are of size 2^i. Dyadic ranges across different levels are mutually aligned.

We will attach one count-min sketch to each level and the elements in the count-min sketch on the level i will be the dyadic ranges of that level (ranges can be hashed just like regular elements.) Given this scheme, Figures 4.9 and 4.10 respectively show how update of a new element as well as estimate for a range takes place. Every new element arriving will be

[45] G. Cormode and S. Muthukrishnan, "An Improved Data Stream Summary: The Count-Min Sketch and Its Applications," Journal of Algorithms, vol. 55, no. 1, p. 58–75, 2005.
[46] M. Charikar and N. Wein, "CS369G: Algorithmic Techniques for Big Data, Lecture 7: Heavy Hitters, Count-Min Sketch," Stanford (Lecture Notes), 2015-2016.

updated in each count-min sketch, by updating its containing range in the respective CMS, as shown in Figure 4.9. Thus one update operation updates each count-min sketch ($O(\log_2 n)$ of them):

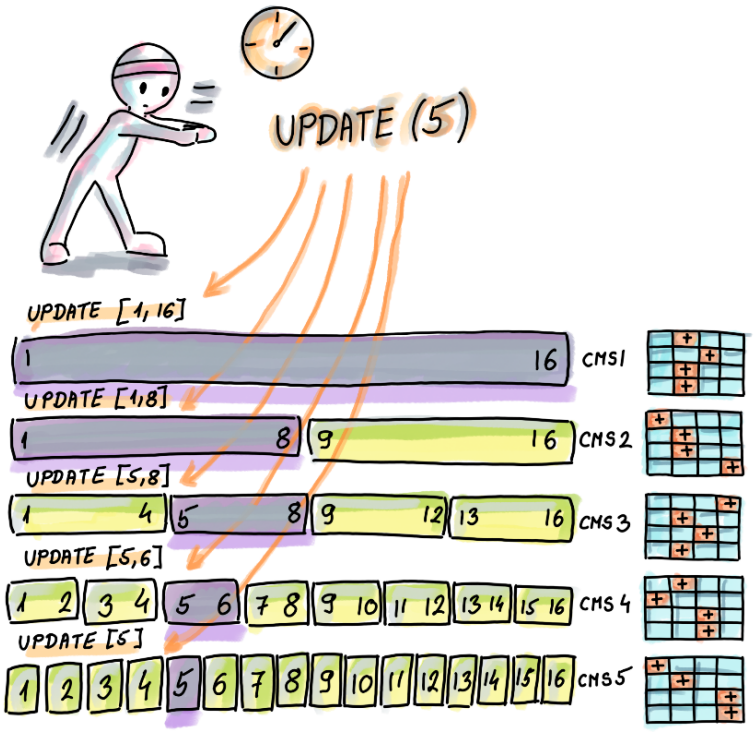

Figure 4.9: An update of one element is transformed into one update per level. For example, if we update 5, we effectively update [1,16] in CMS1, [1,8] in CMS2, [5,8] in CMS3, [5,6] in CMS4, and [5] in CMS5. Instead of updating an element, we are updating a corresponding range to which the element belongs, in the relevant CMS.

Having performed update in such a manner, we are now ready to perform estimate on a particular range. Namely, we will divide the query range into its own dyadic ranges. **For each dyadic interval, we perform estimate in the CMS that resides on its level. The final result is summing up all the estimates. Figure 4.10 shows how we can do the range estimate for [3,13]:**

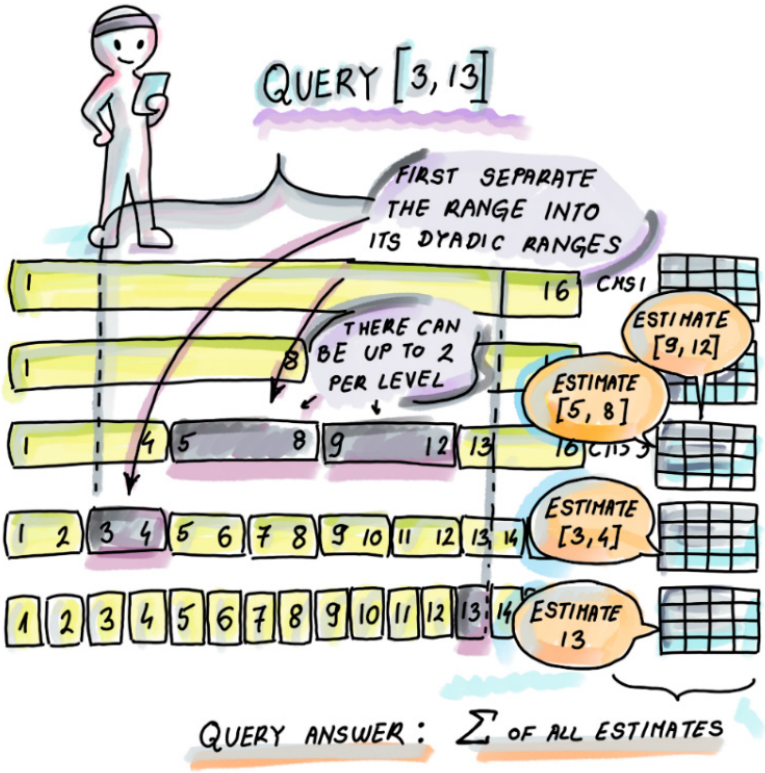

Figure 4.10: In this example, the query range $[3, 13]$ is separated into $[3,4]\cup[5,8]\cup[9,12]\cup[13]$, and we will obtain the frequency estimate for $[3,13]$ by obtaining the frequency estimates for the mentioned ranges and summing them up.

It helps to know that every range can be partitioned into at most $2\log n$ dyadic ranges (at most 2 per level). Whichever range we are given (unless we are given a range that exactly corresponds to one of the dyadic ranges, in which case we can trivially partition), we can find a level on which the interval crosses only one boundary between dyadic ranges. In the example of $[3,13]$, it will be the level 3, where the range crosses $[1,8]$ and $[9,16]$. Now that we have found this boundary at 8, we need to handle the left side of the interval $[3,8]$ and the right side of the interval $[9,13]$ (in the case of a range that touched one of the boundaries, we have only one side). Because the boundaries of dyadic ranges are aligned, we can divide each side into at most $\log_2 n$ smaller ranges on each side (in the same fashion in which we can represent each non-negative integer with $\log_2 n$ binary digits).

Both for the update and for the estimate, the runtime is logarithmic and also, the error grows only logarithmically, not linearly. (We can make the error the same as in the original single count-min sketch by making the individual CMSs in this scheme by a logarithmic factor wider, so that the logs cancel out.)

4.5 Approximate heavy hitters

Approximate heavy hitters is the last (but not the least) application of count-min sketch that we will discuss in this chapter. In our practical scenario with restless sleepers, we were interested in answering a top-k query. The problem of *heavy hitters* is similar to top- k query in that heavy hitters are also the most frequent elements in the dataset, more precisely, $HH(N,k)$is an instance of a heavy hitter problem where in the stream with the total sum of frequencies N (or, if frequencies are all 1, then N corresponds to the number of elements encountered thus far in the stream), we are interested in outputting all items that occur at least N/k times. By pigeonhole principle, there can be at most k heavy hitters, so whenever we have a heavy hitter, we also have an element that is in top- k, but the other direction does not hold: a top- k element need not be a heavy hitter.

4.5.1 Majority element

The simplest version of heavy hitters when $k = 2$ is similar to a popular problem called the *majority element*. Given an array of N elements, and provided that the array contains an element that occurs at least $\lfloor N/2 \rfloor + 1$ times (i.e., majority element), the task is to output the majority element.

> **Something to think about – 4.**
>
> **Before moving on, try to design the best algorithm you can (both from the time and space perspective) for the majority problem.**

This problem can be solved using a one-pass-over-the-array algorithm[47] that uses only two extra variables. The algorithm works by storing the current frontrunner in the battle for the majority element, along with the counter that records by how much the current frontrunner leads. As we sweep the array from left to right, at the current element of the array `A[i]`, if the counter is 0 (no one leads), `A[i]` is made the frontrunner and the counter is set to 1. Otherwise, if the counter was not 0 (the existing frontrunner leads), then the counter is either incremented --- when `A[i]` equals the frontrunner, or it is decremented --- when `A[i]` is different from the frontrunner. So for example, in the array:

```
A = [4, 5, 5, 4, 6, 4, 4]
```

the sequence of `(frontrunner, counter)` pairs goes like this:

```
(4,1),(4,0),(5,1),(5,0),(6,1),(6,0),(4,1)
```

The last frontrunner 4 indeed is the majority element. Another, more visual way to think about this problem is to grab an arbitrary pair of adjacent numbers in the array that are not equal to each other and throw them out. Then contract the hole created by throwing out that pair, and continue this process until there are no more distinct pairs to throw out (there is only one distinct element in the array). The element we end up with --- potentially multiple

[47] T. Roughgarden and G. Valiant, "The Modern Algorithmic Toolbox Lecture #2: Approximate Heavy Hitters and Count-Min Sketch," Stanford (Lecture notes), 2020.

occurrences of it --- is guaranteed to be the majority. The example below shows how this algorithm works:

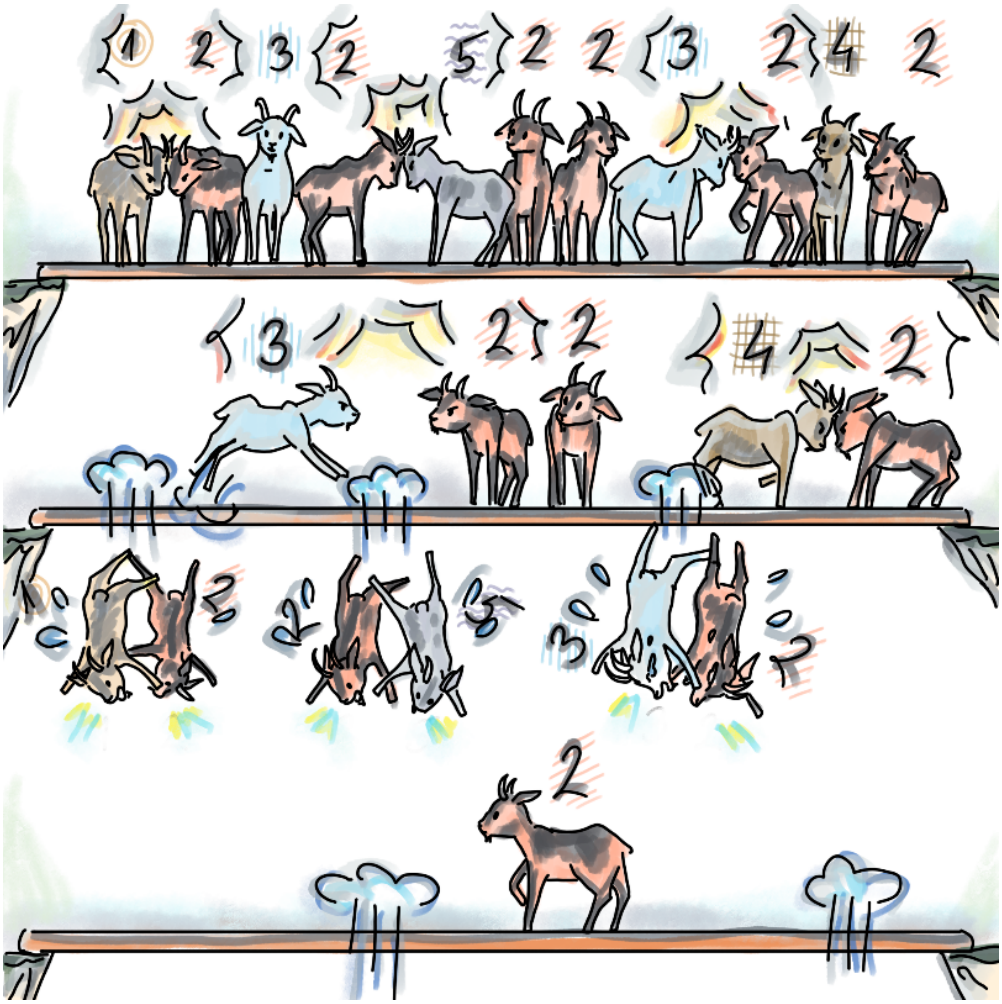

Figure 4.11: We find a majority element in an array by having different neighboring elements throw each other out. In this example, after we throw out (1,2), (2,5), (3,2), and then (3,2) and (4,2), we are left with the majority element 2.

The beauty of this algorithm is that it works no matter in which order we get rid of the pairs, and we can get rid of multiple pairs at the same time --- something that might be useful in a parallelized setting such as MapReduce, where we can split the large array into multiple sub-arrays, have every sub-array individually perform this algorithm, and then finally merge the results and finish up at one node.

4.5.2 General heavy hitters

It would be nice to extend the neat solution for the majority element to the general heavy hitters problem. The issue now is that there are many potential heavy hitters, and therefore many different counters to be maintained. In the extreme case where $n = k/2$, we are looking for the elements that occur twice or more.

If you consider a long data stream where all the elements we discovered thus far have been distinct, then the next item we encounter might either be a repetition of an existing element, in which case we have a heavy hitter, or a new element. This setup is really stretching us out in terms of memory consumption because in order to know whether we have a repeat or a new element, we have to keep all the elements encountered thus far. This sneaky toy example can be generalized to other values of k, and its existence should convince you that we cannot always solve the general heavy hitter problem in one-pass and with sublinear space, and that we need to turn to solving this problem approximately.

Solving *approximate* heavy hitters means that we will report all elements that occur at least $N/k - \varepsilon N$ times, that is, all the heavy hitters plus all the elements that are at most εN short from being heavy hitters. We can effectively identify approximate heavy hitters by constructing a count-min sketch with $\varepsilon = 1/2k$. Again, just like in the restless sleepers scenario, we can use a min-heap to keep the top scorers and as we scan the stream, insert into min-heap only the elements for whom the count-min sketch reports the frequency of N/k and above. For example, when $N=10^9$ and $k=10^6$, the min-heap will contain all the elements for whom the count-min sketch reports the frequency of 1,000 and above, while in reality those elements might have the frequency of 500 and above.

Another way to obtain heavy hitters using count-min sketch is to use the construction of dyadic ranges from our range queries exploration (refer to Figure 4.10 for orientation) by doing queries on the top-level range first (size of the entire universe interval), and if the reported frequency is higher than N/k, we continue to the next level of dyadic ranges (size of the half of the universe interval), and so on. We proceed down the levels by only querying the dyadic ranges whose parent dyadic range, the one right above it, has reported the frequency of N/k or above. The last level brings us to unit intervals that correspond to individual elements that have the frequency of N/k and above.

4.6 Summary

- Frequency estimation problems arise very commonly in the analysis of big data, especially in sets that have both many occurrences of very few items and a small number of occurrences of many items. Even though in the standard RAM setting, frequency estimation can be simply solved in linear-space, solving this problem becomes very challenging in the context of streaming data where we both have only one pass over the data and we need sublinear space.
- Streaming data is a specific type of big data because it often collects large amount of data from multiple sources and at a rapid rate. Keeping every little piece of data is less important than doing real-time analysis of the "big picture", and one of such problems is finding the most popular elements, i.e., frequency estimation.
- Count-Min sketch can efficiently solve the problems of frequency estimation, top-k

query, range queries, heavy hitters, and others in a very small amount of space. If the allowed band of overestimate error is kept as a fixed percentage of the total quantity of data N, then the amount of space in count-min sketch is independent of the dataset size.

- Range queries are usually not that well solved with hash-based sketches, but with count-min sketch, it is possible to do a construction that does frequency estimates for ranges with a fairly small error, using some more space (i.e., using a couple of count-min sketch.)
- Count-min sketch is well suited to solve the approximate heavy hitters problem, that in the streaming context, can only be solved approximately. The basic version of this problem is majority element, that has a one-pass and constant-memory algorithm, but the same can not be said for the general version of heavy hitters.