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 3

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 shape the design of algorithms and data structures
- 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, depth-first search, breadth-first search and many other fundamental 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 has 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 load all your data into RAM, 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 collects data from its users, storing all data into the working memory of your computer is often infeasible. Many applications today, ranging from banking, e-commerce, scientific applications and Internet of Things (IoT), routinely manipulate datasets of terabyte (TB) or petabyte (PB) sizes, i.e., you don't have to work for Facebook or Google to encounter massive data at work.

You might be asking yourself how large the dataset has to be for someone to benefit from the techniques shown in this book. We deliberately avoid putting a number on what constitutes a massive dataset or a "big-data company", as it depends on the problem being solved, computational resources available to the engineer, system requirements, etc. Some companies with enormous datasets also have copious resources and can afford to delay thinking creatively about scalability issues by investing in the infrastructure (e.g., by buying tons of RAM). A developer working with moderately large datasets, but with a limited budget for the infrastructure, and extremely high system performance requirements from their client can benefit from the techniques shown in this book as much as anyone else. And, as we will see, even the companies with virtually infinite resources choose to fill that extra RAM with clever space-efficient data structures.

The problem of massive data has been around for much longer than social networks and the internet. One of the first papers[1] to introduce *external-memory algorithms* (a class of algorithms that neglect the computational cost of the program in favor of optimizing far more time-consuming data-transfer cost) appeared back in 1988. As the practical motivation for the research, the authors use 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, using the working memories of that time (~2-4MB). Figuring out how to sort all the checks while being able to sort only 4MB worth of checks at one time, and figuring out how to do so with the smallest number of trips to disk, was a relevant problem back then, and since it has only grown in relevance. Namely, in past decades, data has grown tremendously, but 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.) Thus managing data size using succinct representations that preserve its key features, and modifying data access patterns to be hardware-friendly are among the crucial ways to speed up an application stuck on processing data.

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

[1] 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.

effective, need to respond to changes very quickly. In streaming applications[2], data effectively flies by without ever being stored, and the application needs to capture the 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 specific to 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: 2833908010
    article-id: 779284
    user-id: 9153647
    text: this recipe needs more butter
    views: 14375
    likes:  43
}
```

You are looking at approximately 3 billion user comments totaling 600GB in data size. Some of the questions you would like to answer about the dataset include determining the most popular comments and articles, classifying articles according to themes and common keywords occurring in the comments, and so on. But first we need to address the issue of duplicates that accrued over multiple instances of crawling, and ascertain the total number of distinct comments in the dataset.

1.1.1 An example: how to solve it

One common way to store unique elements is to create a key-value dictionary where each distinct element's ID is mapped to its frequency. Some of the libraries implementing key-value dictionaries include `map` in C++, `HashMap` in Java, `dict` in Python, etc. Key-value dictionaries are commonly implemented either with a balanced binary tree (e.g., a red-black tree in C++'s `map`), or a hash table (e.g., Python's `dict`.)

Efficiency note: the tree dictionary implementations, apart from lookup/insert/delete that runs in fast logarithmic time, offer equally fast predecessor/successor operations, that is, the ability to explore data using lexicographical ordering, whereas the hash table implementations don't; however the hash table implementations offer blazing fast constant-time performance on lookup/insert/delete. Let's assume in this example our priority is the speed of basic operations, so we will be working with the hash table implementation.

Using `comment-id` as the key, and the number of occurrences of that `comment-id` in the dataset as the value will help us store distinct comments and their frequencies, which we will use to effectively "eliminate" duplicates (the `(comment-id -> frequency)` dictionary from

[2] B. Ellis, Real-Time Analytics: Techniques to Analyze and Visualize Streaming Data, Wiley Publishing, 2014.

Figure 1.1). However, in order to store `<comment-id, frequency>` pairs for 3 billion comments, using 8 bytes per pair (4 bytes for `comment-id` and 4 bytes for `frequency`), we might need up to 24GB. Depending on the method we use to implement the underlying hash table, we need 1.5x or 2x the space taken for elements for the bookkeeping (empty slots, pointers, etc), bringing us close to 40GB.

If we are also to classify articles according to certain topics of interest, we can again employ dictionaries (other methods are possible as well) by building a separate `(article-id -> keyword_frequency)` dictionary for each topic (e.g., sports, politics, science, etc), as shown in Figure 1.1, that counts the number of occurrences of topic-related keywords in all the comments, grouped by `article-id`, and stores the total frequency in one entry --- for example, the article with the `article-id` 745 has 23 politics-related keywords in its associated comments. We pre-filter each `comment-id` using the large `(comment-id -> frequency)` dictionary to only account for distinct comments. A single table of this sort can contain dozens of millions of entries, totaling close to 1GB and maintaining such hash tables for say, 30 topics can cost up to 30GBs only for data, approximately 50GB in total.

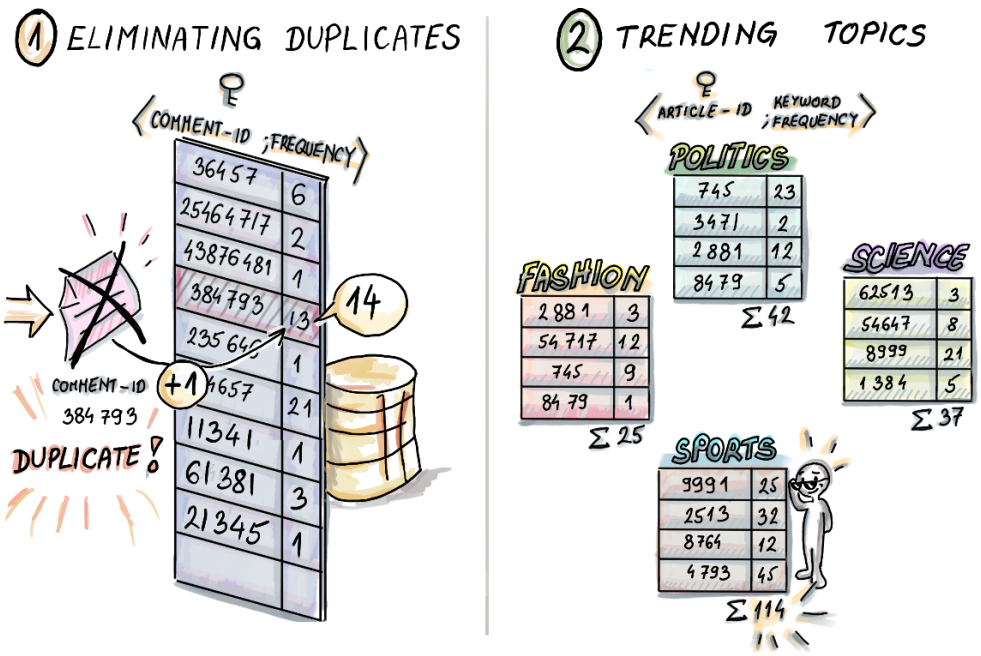

Figure 1.1: In this example, we build a (comment-id, frequency) hash table to help us store distinct comment-id's with their frequency count. An incoming comment-id 384793 is already contained in the table, and its frequency count is only incremented. We also build topic-related hash tables, where, for each article we count the number of times associated keywords appeared in its comments (e.g., in the sports theme, keywords might be: soccer, player, goal, etc). For a large dataset of 3 billion comments, these data structures may require dozens to a hundred of gigabytes of RAM memory.

When the number of items in a dataset becomes large, then every additional bit per item contributes to the system choking on data (Figure 1.2). Common data structures that are a bread-and-butter of every software developer can become too large to efficiently work with, and we need more succinct alternatives.

Figure 1.2: Like most common data structures, hash tables use the asymptotical minimum of space required to store the data correctly, but with large dataset sizes, hash tables cannot fit into the main memory.

1.1.2 How to solve it, take two/ A book walkthrough

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 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 a hash table, think 1 byte per item or less.

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 avoid getting into the gritty mathematical details of how we arrive at these numbers, but the difference between Bloom filters and hash tables worth emphasizing is that Bloom filters do not store the keys (such as `comment-id`) themselves. Bloom filters compute hashes of keys, and use them to modify the data structure. Thus the size of the Bloom filter mainly depends on the number of keys inserted, not their size.

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 topic hash 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 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, there is also a way to control for an occasional error.

COMMENT DATA AS A STREAM.

Alternatively, we could view the same problem of news comments and articles in the context of a fast-moving event stream. Assume that the event constitutes any modification of the comment data, such as viewing a comment, clicking 'Like' on a comment, inserting/deleting a comment or an article, and the events arrive real-time as streaming data to our system (Chapter 6). Note that in this setup, we can also encounter duplicates of `comment-id`, but for a different reason: every time someone clicks 'Like' on a particular comment, we receive the event with the same `comment-id`, but with amended count on the `likes` attribute. Given that events arrive rapidly and on a 24/7 basis and we can not afford to store all of them, for many problems of interest, we can only provide approximate solutions. Mainly, we are interested in computing basic statistics on data real-time (e.g., the average number of likes per comment in the past week), and without the ability to store the like count for each comment, we can resort to random sampling.

We could draw a random sample from the data stream as it arrives using *Bernoulli sampling algorithm* (Chapter 7). 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 `likes` is 90% of all of the comments according to their like count will also trade accuracy for space. We can maintain a type of a dynamic histogram (Chapter 8) 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 our complete data with some error.

COMMENT DATA IN A DATABASE.

Lastly, we might want to store all comment data in a persistent format (e.g., a database on disk/cloud), and build a system on top that would enable the fast insertion, retrieval, and modification of live data over time. In this kind of setup, we favor accuracy over speed, so we are comfortable storing tons of data on disk and retrieving it in a slower manner, as long as we can guarantee 100% accuracy of queries.

Storing data on a remote storage and organizing it so that it lends itself to efficient retrieval is a topic of the algorithmic paradigm called *external-memory algorithms* (Chapter 9). External-memory algorithms address some of the most relevant problems of modern applications, such as for example, the choice, or design and implementation of database engines. In our particular comments data example, we need to ask whether we are building a system with mostly static data, yet constantly queried by users (i.e., *read optimized*), or a system where users frequently add new data and modify it, but query it only occasionally (i.e., *write optimized*)? Or perhaps the combination, where both fast inserts and fast queries are equally important (i.e., *read-write optimized*).

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. The insert/lookup tradeoff is inherent in databases, and it is reflected in the design of data structures that run underneath MySQL, TokuDB, LevelDB and many other storage engines. Some of the most popular data structures to build databases include B-trees, B^{ε}-trees, and LSM-trees, and each sits on a different point of the insert/lookup tradeoff spectrum (Chapter 10). Also, we may be interested in solving other problems with data sitting on disk, such as ordering comments lexicographically or by a number of occurrences. To do that, we need an efficient sorting algorithm that will minimize the number of memory transfers (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 hash tables, and also use the opportunity to learn about modern hash techniques devised to deal with large datasets. Chapter 2 also serves as a good preparation for Chapters 3-5 considering the sketches are hash-based. Data structures we present in Chapters 3-5 such as Bloom filters, count-min sketch and hyperloglog, and their alternatives, have found numerous applications in databases, networking, etc.

Part II (Chapters 6-8) introduces data streams. From classical techniques like Bernoulli sampling and reservoir sampling to more sophisticated methods like stratified sampling, we introduce a number of sampling algorithms suitable for different streaming data models. The created samples are then used to calculate estimates of the total sums or averages, etc. We 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 9-11) covers algorithmic techniques for scenarios when data resides on SSD/disk. First we introduce the external-memory model and then present optimal algorithms for fundamental problems such as searching and sorting, illuminating key algorithmic tricks in this model. This part of the book also covers data structures that power modern databases such as B-trees, B^{ϵ}-trees and LSM-trees.

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[3], *Introduction to Algorithms* by Cormen, Leiserson, Rivest and Stein[4], *Algorithms* by Robert Sedgewick and Kevin Wayne[5], or for a more introductory and friendly take on the subject, *Grokking Algorithms* by Aditya Bhargava[6]. The algorithms and data structures for massive datasets are slowly making their way into the mainstream textbooks, but the world is moving fast and our hope is that our 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. We assume 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. Having said that, massive-data algorithms are a very broad subject and this book is meant to serve as a gentle introduction.

The majority of the 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 excellent books that address specialized aspects of massive datasets [7],[8],[9],[10]. 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.

---

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

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 stem from hardware and general computer architecture. Now, this book is not about hardware, but in order to design efficient 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 we discuss in this chapter 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.

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[11]. 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):

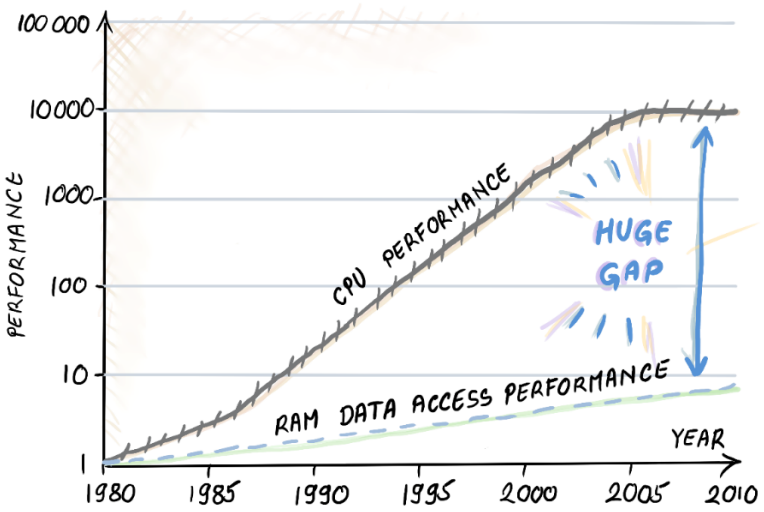

Figure 1.3: CPU-Memory Performance Gap graph, adopted from Hennessy & Patterson's Computer Architecture textbook. The graph shows the widening gap between the speeds of memory accesses to CPU

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

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.

What this gap points to intuitively is that doing computation is much faster than accessing data. So if we are stuck with the mindset that 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 exists a 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[12]. 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.

[12] 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].

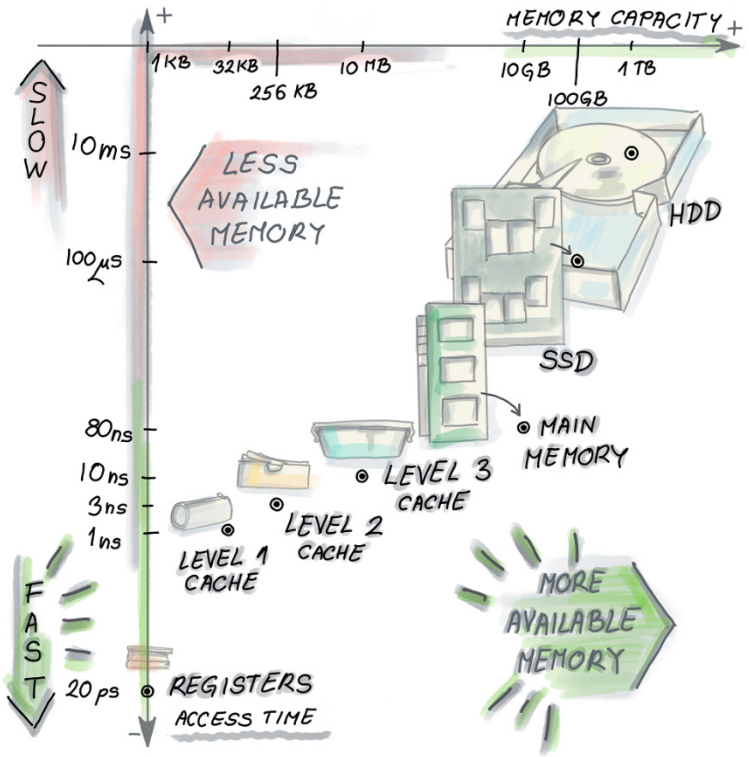

Figure 1.4: Different types of memories in the computer. Starting from registers in the bottom left corner, that are blindingly fast but also very small, we move up (getting slower) and right (getting larger) with Level 1 cache, Level 2 cache, Level 3 cache, main memory, all the way to SSD and/or HDD. 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 and the needle, being one of the few remaining mechanical parts of a computer work a lot like a record player. Placing the mechanical needle on the right track is the time-consuming part of accessing disk data. Once the needle is on the right track, the data transfer can be very fast, depending on how fast the disk spins.

1.4.3 Latency vs. bandwidth

Similar phenomenon, where **"latency lags bandwidth"**[13] holds for different types of memory. 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 is the important measurement in many scenarios

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

where the common user behavior involves many small random accesses as oppose to one large sequential one.

To offset the cost of the expensive initial call, data transfer between different levels of memory is done in chunks of multiple items. Those chunks are called cachelines, pages or blocks, depending on memory level we are working with, and their size is proportionate to the size of the corresponding level of memory, so for cache they are in the range 8-64 bytes, and for disk blocks they can be up to 1MB[14]. 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 data in sequential blocks effectively pre-fetches the items we will likely need in close future.

1.4.4 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.

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

1.5 Designing algorithms with hardware in mind

After looking at some crucial aspects of modern computer architecture, 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[15] 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 that can work around hardware limitations. Designing succinct data structures (or taking data samples) that can fit into small and fast levels of memory helps because this way we avoid expensive disk seeks. In other words, **reducing 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 does many random reads/writes into one that does sequential reads/writes. Throughout this book, we will see how classical algorithms can be transformed, and new ones can be designed having space-related concerns in mind.

However, ultimately it is also important to keep in mind 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.

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

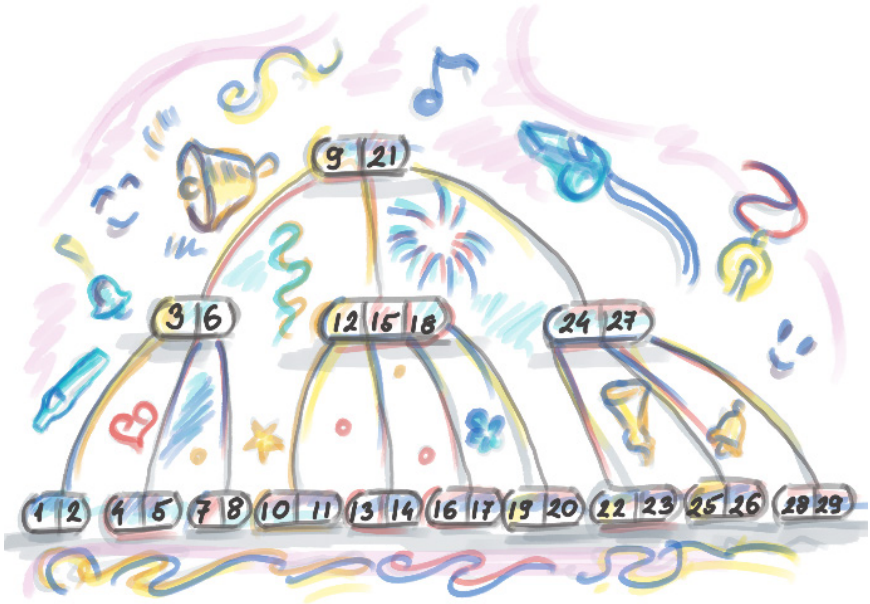

Figure 1.6: An efficient data structure with bells and whistles

1.6 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 number of reasons. First, classical hash tables have proved irreplaceable in modern systems, deeming it harder to find a system that does not use them than the one that does. Second, recently there has been a lot of innovative work addressing algorithmic issues that arise as hash tables grow to fit massive data, such as efficient resizing, compact representation and space-saving tricks, etc. In a similar vein, hashing has over time been adapted to serve in massive peer-to-peer systems where the hash table is split among servers; here, the key challenge is assignment of resources to servers and load-balancing of resources as servers dynamically join and leave the network. Lastly, we begin with hashing because it forms the backbone of all succinct data structures we present in Part I of the book.

Aside from the basics of how hash tables works, in this chapter we show examples of hashing in modern applications such as deduplication and plagiarism detection. We touch upon how Python implements dictionaries as a part of our discussion on hash table design tradeoffs. Section 2.8 discusses consistent hashing, the method used to implement distributed hash tables. This section features code samples in Python that you can try out

and play with to gain a better understanding of how hash tables are implemented in a distributed and dynamic multi-server environment. The last part of the section on consistent hashing contains coding exercises for a reader who likes to be challenged. If you feel comfortable with all things classical hashing, skip right to the Section 2.8, or, if you are familiar with consistent hashing, skip right ahead to Chapter 3.

2.1 Ubiquitous hashing

Hashing is one of those subjects that, no matter how much attention they got in your programming, data structures and algorithms courses, it was not enough. Hash tables are virtually everywhere --- to illustrate this, just consider the process of writing an email (see Figures 2.1-2.4). First, to log into your email account, the password is hashed and the hash is checked against the database to verify a match:

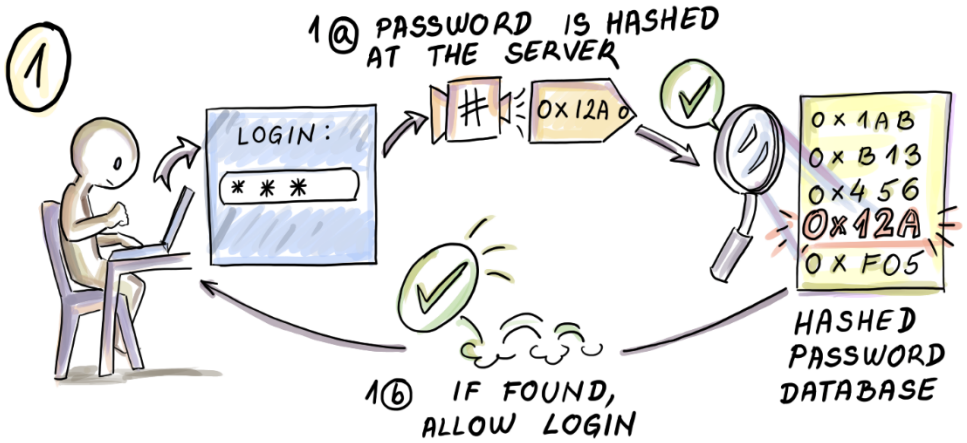

Figure 2.1: Logging into email and hashing.

While writing an email, the spellchecker uses hashing to check whether a given word exists in the dictionary:

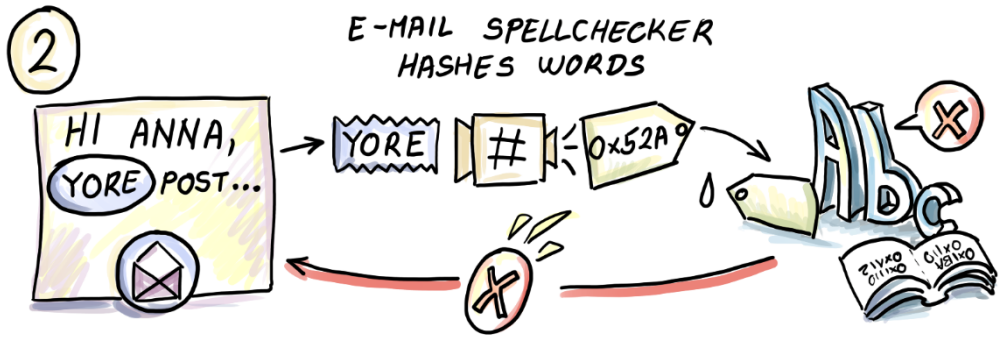

Figure 2.2: Spellchecking and hashing.

When the email is sent, it is separated into network packets, each of which contains a hashed destination IP address on it. If the hash does not match any of the hashes of IP addresses, the packet bounces.

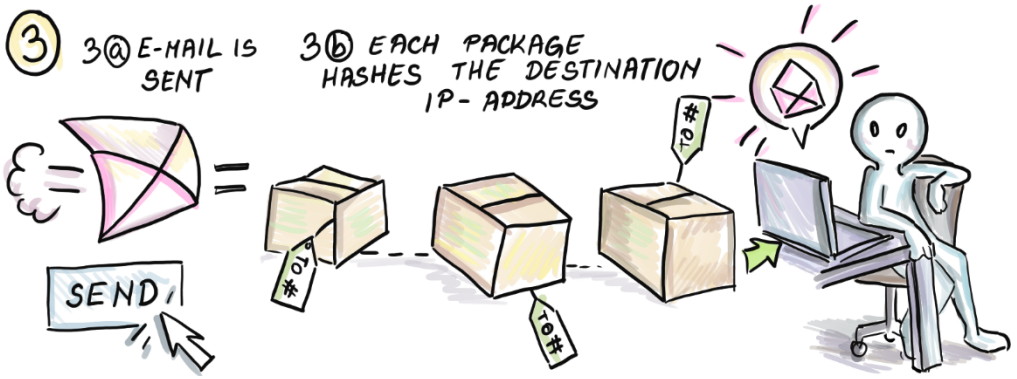

Figure 2.3: Network packets and hashing.

Lastly, when the email arrives at the destination, the spam filters sometimes hash email contents to find spam-like words and filter out likely spam:

Figure 2.4: Spam filters and hashing.

Why is a hash table the data structure of choice for so many tasks? To find out why, we need to look at how well different data structures implement what we call a *dictionary* --- an abstract data type that can do lookup, insert and delete operations.

2.2 A crash course on data structures

Many data structure can perform the role of a dictionary, but different data structures exhibit different performance tradeoffs and thus lend themselves to different usage scenarios. For example, consider a plain **unsorted array**: this rather simple data structure offers ideal constant-time performance on inserts (`O(1)`) as new elements are appended to a log, however, the lookup in the worst-case requires a full linear scan of data (`O(n)`). An unsorted array can serve well as a dictionary implementation for applications where we want extremely fast inserts and where lookups are extremely rare[1].

Sorted arrays allow fast logarithmic-time lookups using binary search(`O(logn)`), which, for many array sizes is effectively as good as constant time (logarithms of astronomically large numbers do not exceed 100). However, we pay the price in the maintenance of the sorted order when we insert or delete, having to move over a linear number of items in the worst-case (`O(n)`). Linear-time operations mean that we roughly need to visit every element during a single operation, a forbidding cost in most scenarios.

Linked lists can, unlike sorted arrays, insert/delete an element from anywhere in the list in constant time (`O(1)`) by rerouting a few pointers, provided we located the position where to insert/delete. The only way to find that position is to traverse the linked list by following pointers, even if the linked list were sorted, which brings us back to linear-time. Whichever way you look at it, with simple linear structures such as arrays and linked lists, there is at least one operation that costs (`O(n)`), and to avoid it, we need to break out from this linear structure.

[1] If we are guaranteed never to need a lookup, there is even a better way to "implement" inserts --- just do nothing.

Balanced binary search trees have all dictionary operations dependent on the depth of the tree, which is using different balancing mechanisms (AVL, red-black, etc) kept at `O(logn)`. So all insert, lookup and delete take logarithmic time in the worst case. In addition, balanced binary search trees maintain the sorted order of elements, which makes them an excellent choice for performing fast range, predecessor and successor queries. Logarithmic bound on all basic operations is pretty good, and in fact, if we allow the algorithm only to perform comparisons in order to locate an element, this is the best we can do. However, computers are capable of many other operations, including bit shifts, arithmetic operations, and other operations used by hash functions.

Using **hashing**, we can bring the dictionary operation costs down to `O(1)` on all operations. And, if you are thinking this is too good to be true, you are quite right: unlike the bounds mentioned so far, where the runtime is guaranteed (i.e., worst-case), the constant-time runtime in hash tables is expected. The worst case can still be as bad as linear-time `O(n)`, but with a clever hash table design, we can almost always avoid such instances. So even though the worst-case on a lookup for a hash table is the same as that on an unsorted array, in the former case, the `O(n)` will almost never happen, while in the latter case, it will quite consistently happen. Even when a pathological case occurs in hash tables, it is amortized against a huge number of blindingly fast common 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.

Hash tables are, on the other hand, poorly suited for all applications where having your data ordered is important. A good hash function scrambles the input and scatters items to different areas of the hash table --- the word hash comes from the French *'hache'*, often used to describe a type of dish where meat is chopped and minced into many little pieces (also related to *'hatchet'*). The natural consequence of this 'mincing' of data is that the order of items is not preserved. The issue comes in focus in databases where answering a range query requires navigating the sorted order of elements: for instance, listing all employees ages between 35 and 56, or finding all points on a coordinate x between 3 and 45 in a spatial database. Hash tables are most useful when 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.

2.3 Usage scenarios in modern systems

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

2.3.1 Deduplication in backup/storage solutions

Many companies such as Dropbox and Dell EMC Data Domain storage systems[2] deal with storing large amounts of user data by taking frequent snapshots and backups. Clients for these companies are often large corporations that hold enormous amounts of data, and if the

[2] 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].

snapshots are taken frequently enough (say, every 24 hours), the majority of data between the consecutive snapshots will remain unchanged. In this scenario, it's important to quickly find the parts that have changed and store only them, thereby saving time and space of storing a whole new copy of data. To do that, we need to be able to efficiently identify duplicate content.

Deduplication is the process of eliminating duplicates, and the majority of its modern implementations use hashing. For example, consider *ChunkStash*[3], a deduplication system specifically designed to provide 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 2.5).

Chunking the files helps to identify near-duplicates, where small edits have been made to a large file.

[3] 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.

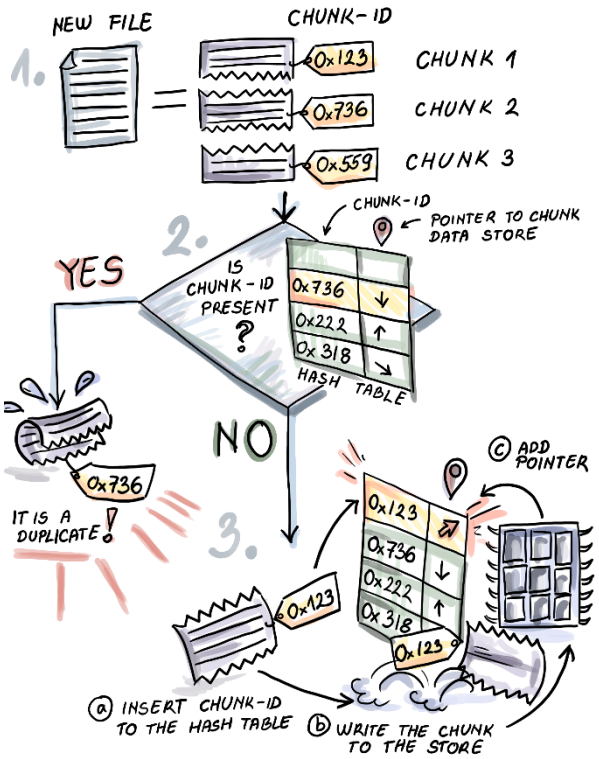

Figure 2.5: Deduplication process in backup/storage solutions. When a new file arrives, it is split into small chunks. In our example, the file is split into three chunks, and each chunk is hashed (e.g., chunk 1 has chunk-id 0x123, and chunk 2 has chunk-id 0x736.) Chunk-id 0x123 is not found in the hash table. A new entry is created for this particular chunk-id, and the chunk itself is stored. The chunk-id 0x736, having been found in the hash table, is deemed a duplicate and isn't stored.

There are more intricacies to this process than what we show. For example, when writing the new chunk to the flash 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 page, a particularly expensive operation in flash. But let's stay in the in-memory lane for now; buffering and writing efficiently to disk will be given more attention in the Part III of the book.

2.3.2 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[4]

[4] 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.

is a variant of Karp-Rabin string-matching algorithm[5] that relies on k-gram fingerprinting (k-gram is a contiguous substring of length k). Let's first review the algorithm.

Given a string t that represents a large text, and a string p that represents a smaller pattern, a string-matching problem asks whether there exists an occurrence of p in t. There is a rich literature on string-matching algorithms, most of which perform substring comparisons between p and t. Karp-Rabin algorithm instead performs comparisons of the hashes of substrings, and does so in a clever way. It works extremely well in practice, and the fast performance (which should not surprise you at this point) is partly due to hashing.

Namely, only when the hashes of substrings match, does the algorithm check whether the substrings actually match. In the worst case, we will get many false matches due to hash collisions, when two different substrings have the same hash yet substrings differ. In this case, the total runtime is $O(|t||p|)$, like that of a brute-force string matching algorithm. But in most situations when there are not many true matches, and with a good hash function, the algorithm zips through t. This is the randomized linear time, but clearly good enough to offer some real practical benefits. See Figure 2.6 for an example of how the algorithm works.

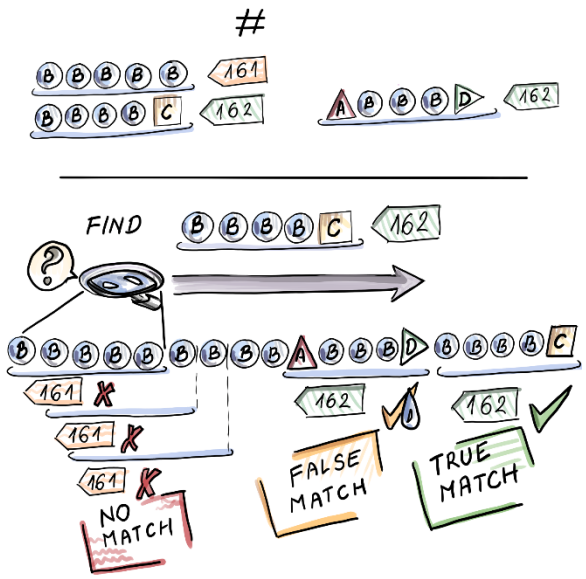

Figure 2.6: Example of a Karp-Rabin fingerprinting algorithm. We are looking for a pattern p=BBBBC in the larger string t=BBBBBBBBBABBBDBBBBC. The hash of BBBBC is equal to 162 and it is a mismatch for the hash 161 of BBBBB that occurs at the beginning of the long string. As we shift right, we repeatedly encounter hash mismatches until the substring ABBBD, with the hash of 162. Then we check the substrings and establish a false match. At the very end of the string, we again encounter the hash match at BBBBC and upon checking the substrings, we report a true match.

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

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 (see Figure 2.7). This can be done if the rolling hash function is such that it 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 of characters in the string.)

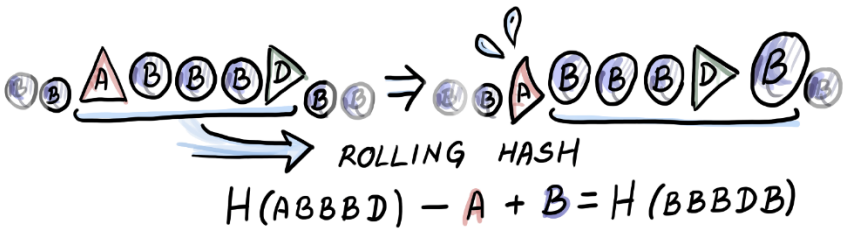

Figure 2.7: Rolling hash. Computing the hash for all but the first substring of t is a constant-time operation. For example, BBBDB, we needed to "subtract" A and "add" B to ABBBD.

Karp-Rabin algorithm could be used in a straightforward manner to compare two assignments for plagiarism by splitting files into smaller chunks and fingerprinting them. However, in MOSS, we are interested in a large group of submitted assignments, and all potential instances of plagiarism. This rings all-to-all comparisons and an impractical quadratic-time algorithm. 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.

2.4 $O(1)$ –– 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 does lookups, inserts and deletes all in `O(1)` in the worst case?

If we knew all the items to be inserted into the hash table beforehand, then we could conjure up a hash function customized to our data that distributes the items perfectly, one to each bucket, but what's the fun in that?! Part of the problem with not knowing data beforehand is that the hash functions needs to provide a mapping of any potential item to a corresponding hash table bucket. The set representing all potential items, whatever type of data we are dealing with, is likely extremely large in comparison to the number of hash table buckets. We will refer to this set of all potential items as the universe `U`, the size of our dataset as n, and the hash table size as `m`.

The values of `n` and `m` are roughly proportional, and both are considerably smaller than `U`. This is why the hash function mapping the elements of `U` to `m` buckets will inevitably end up with a fairly large subset of `U` mapping to the same bucket of the hash table. Even if the hash function perfectly evenly distributes the items from the universe, there is at least one bucket to which at least $|U|/m$ items get mapped. Because we do not know what items will be contained in our dataset, and if $|U|/m \geq n$, it is feasible that all items in our dataset hash to the same bucket. It is not very likely that we will get such a dataset, but it is possible.

For example, consider the universe of all potential phone numbers of the format *ddd-ddd-dddd*, where *d* is 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 (it would be pretty bad even if any constant fraction of the dataset ended up in one bucket). 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.

2.5 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, **chaining** associates with each bucket of the hash table an additional data structure (e.g., linked list, or a binary search tree), where the items hashed to the corresponding bucket get stored. New items get inserted right up front `(O(1)` but search and delete require advancing through the pointers of the appropriate list, the operation whose runtime is highly dependent on how evenly items are distributed across the buckets. To refresh your memory on how chaining works, see Figure 2.8:

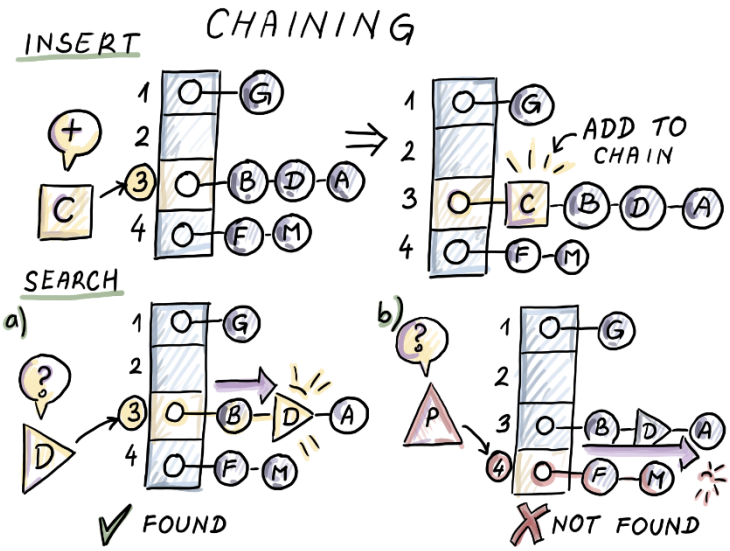

Figure 2.8: An example of insert and search with chaining.

Linear probing is a particular instance of *open addressing*, a hashing scheme where we store items inside the hash table slots. In linear probing, to insert an item, we hash it to a corresponding bucket, and if the slot determined by the bucket is empty, we store the item into it. If it is occupied, we look for the first available position scanning downward in the table, wrapping around the end of the table if needed. An alternative variant of open addressing, quadratic probing, advances in quadratic-sized steps when looking for the next position to insert.

The search in linear probing, just like the insert, begins from the position of the slot determined by the bucket we hashed to, scanning downward until we either find the element searched for, or encounter an empty slot. Deletion is a bit more involved, as it can not simply remove an item from its slot --- it might break a chain that would lead to an incorrect result of a future search. There are many ways to address this, for instance, one simple one being placing a tombstone flag at the position of the deleted element. See Figure 2.9 for an example of linear probing:

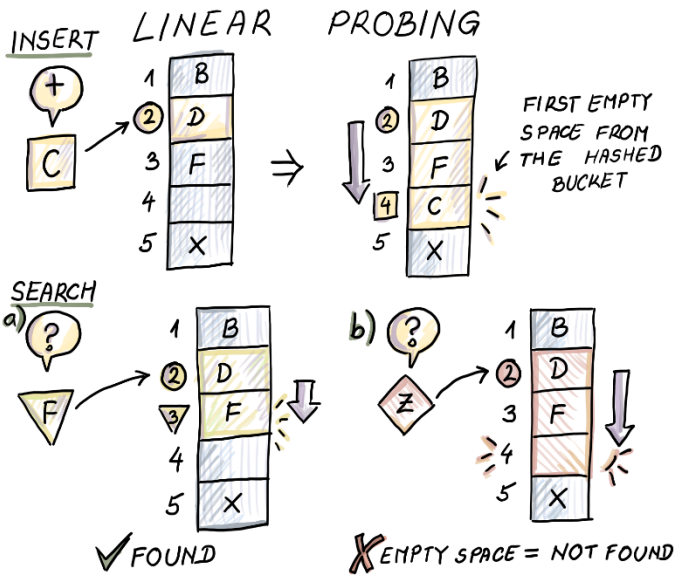

Figure 2.9: An example of insert and search with linear probing.

First let's see what 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. This then allows us to analyze the process of hashing using the analogy of throwing n balls into n bins uniformly randomly. With high probability, the fullest bin will have $O(\log n/\log \log n)$ balls[6], hence the longest chain in the chaining method is no longer than $O(\log n/\log \log n)$, giving an upper bound on the lookup and delete performance.

The high-probability bounds are stronger than the expectation bounds we have discussed earlier. The expression "with high probability" means that, if our input is of size `n`, then the high-probability event happens with the probability of at least `1-1/n^c` , where `c≥1` is some constant. The higher the constant and the input size, the tinier becomes the chance of the high-probability event not occurring. What this means practically, is that many other failures will happen before the high-probability event fails us.

The logarithmic lookup time 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 (assuming the number of items is proportional to the number of buckets in the chaining table).

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

Using pairwise independent hashing, one can show that the worst-case lookups in linear probing are close to $O(\log n)$.[7] Families of k-wise independent hash functions are the best we have gotten so far to mimicking 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 protects us from the adversary who can see our code: by choosing one among many hash functions randomly at runtime, we make it harder to produce a pathological dataset, and even if it happens, it will not be our fault (an important factor in the setting of our daily jobs).

It makes intuitive sense that the worst-case lookup cost in linear probing is slightly higher than that of chaining, as the elements hashing to different buckets can contribute to the length of the same linear probing run. But does the fancy theory translate into the real world performance differences?

Well, we are, in fact, missing an important detail. The linear probing runs are laid out sequentially in memory, and most runs are shorter than a single cacheline, which has to be fetched anyway, no matter the length of the run. The same can not be said about the elements of the chaining list, for which the memory is allocated in a non-sequential fashion. Hence, chaining might need more accesses to memory, which significantly reflects on the actual runtime. Similar case is with another clever collision-resolution technique called *cuckoo hashing*, that promises that an item contained in the table will be found in one of the two locations determined by two hash functions, deeming the lookup cost constant in the worst case. However the probes are often in very different areas of the table so we might need two memory accesses.

Considering the gap in the amount of time required to access memory vs CPU we discussed in Chapter 1, it makes sense why linear probing is often the collision-resolution method of the choice in many practical implementations. Next we explore an example of a modern programming language implementing its key-value dictionary with hash tables.

2.6 Usage scenario: How Python's dict does it

Key-value dictionaries are ubiquitous across different languages. For standard libraries of C++ and Java, for example, they are implemented as `map`, `unordered_map` (C++) and `HashMap` (Java); `map` is a red-black tree that keeps items ordered, and `unordered_map` and `HashMap` are unordered and are running hash tables underneath. Both use chaining for collision resolution. In Python, the key-value dictionary is `dict`. Here is a simple example of how to create, modify and access keys and values in `dict`:

```
d = {'turmeric': 7, 'cardamom': 5, 'oregano': 12}
print(d.keys())
print(d.values())
print(d.items())
d.update({'saffron': 11})
print(d.items())
```

[7] 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.

The output is as follows:

```
dict_keys(['turmeric', 'cardamom', 'oregano'])
dict_values([7, 5, 12])
dict_items([('turmeric', 7), ('cardamom', 5), ('oregano', 12)])
dict_items([('turmeric', 7), ('cardamom', 5), ('oregano', 12), ('saffron', 11)])
```

The authors of Python's default implementation, CPython, explain in its documentation[8] how `dict` is implemented (here we only focus on the case when keys are integers): for the table size $m=2^i$, the hash function is $h(x) = x \ mod \ 2^i$ (i.e., the bucket number is determined by the last i bits of the binary representation of x.) This works well in a number of common cases, such as the sequence of consecutive numbers, where it does not create collisions; it is also easy to find cases where it works extremely poorly, such as a set of all numbers with identical last `i` bits. Moreover, if used in combination with linear probing, this hash function would lead to clustering and long runs of consecutive items. To avoid long runs, Python employs the following probing mechanism:

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

where `j` is the index of a bucket where we will attempt to insert next. 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 and it makes sufficient skips to avoid clustering in common case. To make sure higher bits of the key are used in hashing, the variable `perturb` is used that is originally initialized to the $h(x)$ and a constant `PERTURB_SHIFT` set to 5:

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

If the insertions match our $(5 * j) + 1$ pattern, then we are in trouble, 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 bad case occurs.

2.7 MurmurHash

In this book, we will be interested in fast, good and simple hash functions. To that end, we make a brief mention of MurmurHash invented by Austin Appleby, a fast non-cryptographic hash function employed by many implementations of the data structures we introduce in our future chapters. The name Murmur comes from basic operations multiply and rotate used to

[8] 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].

mince the keys. One Python wrapper for MurmurHash is `mmh3`[9] which one can install in the console using

```
pip install mmh3
```

The package mmh3 gives a number of different ways to do hashing. Basic hash function gives a way to produce signed and unsigned 32-bit integers with different seeds:

```
import mmh3 as mmh3
print(mmh3.hash("Hello"))
print(mmh3.hash(key = "Hello", seed = 5, signed = True))
print(mmh3.hash(key = "Hello", seed = 20, signed = True))
print(mmh3.hash(key = "Hello", seed = 20, signed = False))
```

producing a different hash for different choices of `seed` and `signed` parameters:

```
316307400
-196410714
-1705059936
2589907360
```

To produce 64-bit and 128-bit hashes, we use `hash64` and `hash128` functions, where `hash64` uses the 128-bit hash function and produces a pair of 64-bit signed or unsigned hashes. Both 64-bit and 128-bit hash functions allow us to specify the architecture (x64 or x86) in order to optimize the function on the given architecture:

```
print(mmh3.hash64("Hello"))
print(mmh3.hash64(key = "Hello", seed = 0, x64arch= True, signed = True))
print(mmh3.hash64(key = "Hello", seed = 0, x64arch= False, signed = True))
print(mmh3.hash128("Hello"))
```

producing the following (pairs of) hashes:

```
(3871253994707141660, -6917270852172884668)
(3871253994707141660, -6917270852172884668)
(6801340086884544070, -5961160668294564876)
212681241822374483335035321234914329628
```

2.8 Hash Tables for Distributed Systems: Consistent Hashing

The first time the consistent hashing came to a spotlight was in the context of web caching.[10],[11] Caches are a fundamental idea in computer science that has improved systems

[9] "mmh3 3.00 Project Description", 26 February 2021. [Online]. Available: https://pypi.org/project/mmh3/

across many domains. On the web, for example, caches relieve the hotspots that occur when many clients request the same webpage from a server. Servers host webpages, clients request them via browsers, and caches sit in between and host copies of frequently accessed webpages. In most situations, caches are able to satisfy the request faster than the home servers, and distribute the load between themselves so that no cache is overwhelmed. Once a cache miss occurs, i.e., the webpage is not found in the cache, the cache fetches the website from the originating server. An important problem to solve in this setup is assigning web pages (in future text, **resources**) to caches (in future text, **nodes**), considering the following constraints:

1. A fast and easy mapping from a resource to a node --- client and the server should be able to quickly compute the node responsible for a given resource.
2. A fairly equal resource load among different nodes to relieve hotspots.
3. The mapping should be flexible in the face of frequent node arrivals and departures. As soon as the node leaves (i.e., a spontaneous failure occurs), its resources should be efficiently re-assigned to other node(s), and when a new node is added, it should receive an equal portion of the total network load. All this should happen seamlessly, without too many other nodes/resources being affected.

2.8.1 A typical hashing problem?

From the requirements (1) and (2), it looks like we have a hashing problem at our hands: nodes are the buckets to which resources get hashed, and a good hash function can ensure a fair load-balance. Holding a hash table can help us figure out which node holds which resource. So when a query occurs, we hash the resource and see what bucket (node) should contain it (Figure 2.10, left). This would be fine, if we were not in a highly dynamic distributed environment, where nodes join and leave (fail) all the time (Figure 2.10, right.) The challenge lies in satisfying requirement (3): how to re-assign node's resources when it leaves the network, or how to assign some resources to a newly arriving node, keeping in mind that load balance remains fairly equal, and without disturbing the network too much.

As we know, classical hash tables can be resized by rehashing using a new hash function with a different range and copying the items over to a new table. This is a very expensive operation, and it typically pays off because it is done once in a while and it is amortized against a large number of inexpensive operations. For our dynamic web caching scenario, where node arrivals and departures happen constantly, changing resource-to-node mappings every time a minor change to the network occurs is highly impractical (Figure 2.11).

[10] D. KTarger, 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.

[11] 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].

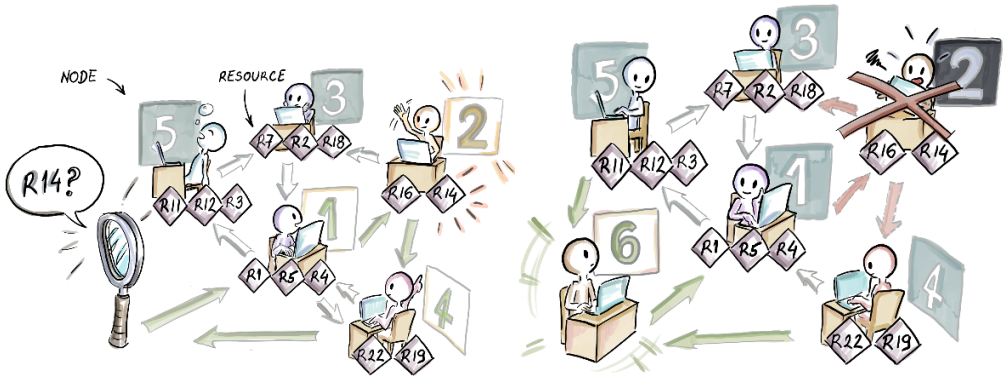

Figure 2.10: Using a hash table, we can map resources to nodes and help locate appropriate node for a queried resource (left). The problem arises when nodes join/leave the network (right.)

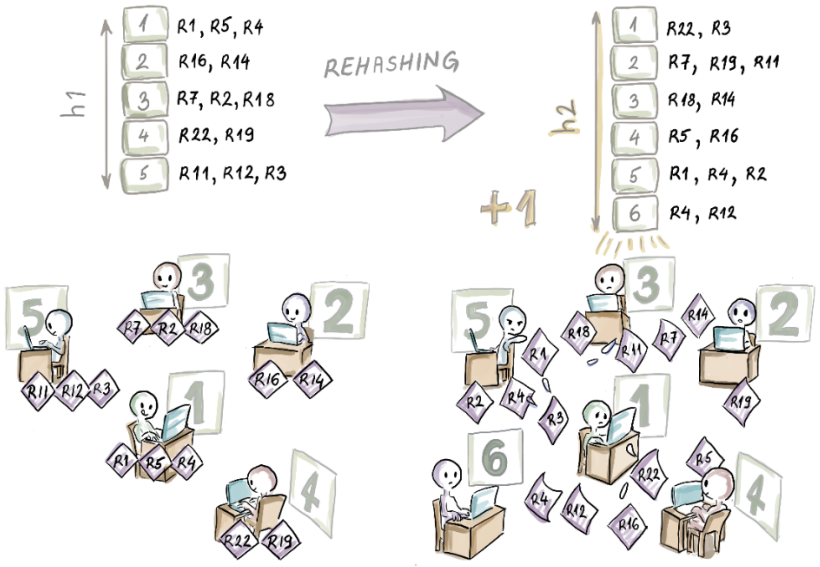

Figure 2.11: Rehashing is not feasible in a highly dynamic context, because one node join/failure triggers re-assignment of all resource-node allocations. In this example, changing the hash table size from 5 to 6 changed node allocations for most resources. The bottom right illustration shows the "in-between" moment when nodes hold some out-of-date and some new resources.

In the following sections, we will show how consistent hashing helps in satisfying all three requirements of our problem. We begin by introducing the concept of a hashring.

2.8.2 Hashring

The main idea of consistent hashing is to hash *both* resources and nodes to a fixed range $R = [0, 2^k - 1]$. It is helpful to visually imagine `R` spread out around a circle, with the northmost point being 0, and the rest of the range spread out clockwise in the increasing order uniformly around the circle. We denote this circle the *hashring*.

Each resource and each node have a position on the hashring defined by their hashes. Given this setup, each resource is assigned to the first node encountered clockwise on the hashring. A good hash function should ensure that each node receives a fairly equal load of resources. See an example in the Figure 2.12:

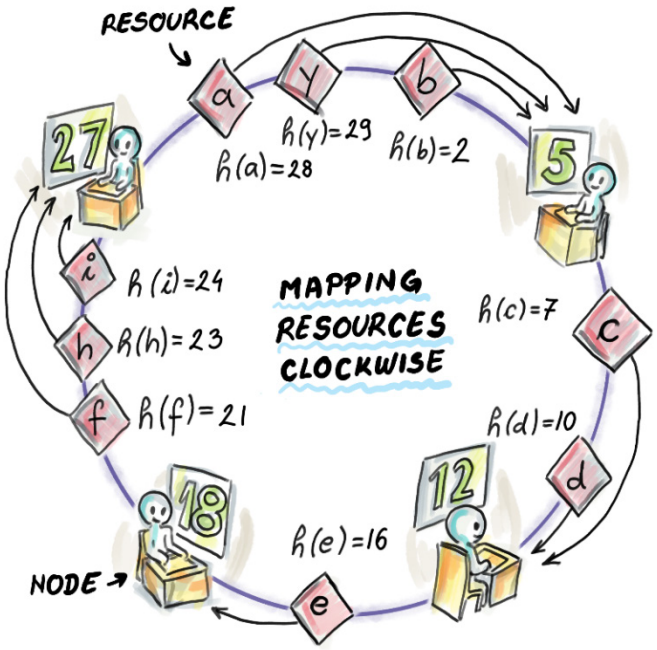

Figure 2.12: Mapping resources to nodes in the hashring. The example shows the hashring `R=[0,31]` and nodes whose whose hashes are 5, 12, 18 and 27. Resources a, y and b are assigned to node 5, c and d are assigned to node 12, e is assigned to node 18, and f,h and i are assigned to node 27.

To illustrate how consistent hashing works, along with node arrivals and departures, we show a simple Python implementation of the class `HashRing` step-by-step. Our implementation, shown in a sequence of small snippets, is only a simulation of the algorithm (the actual implementations of consistent hashing involve network calls between nodes, etc.) `HashRing` is implemented using a circular doubly-linked list of nodes where each node stores its resources in a local dictionary:

```
class Node:
    def __init__(self, hashValue):
         self.hashValue = hashValue
         self.resources = {}
         self.next = None
         self.previous = None

class HashRing:
    def __init__(self, k):
        self.head = None
        self.k = k
        self.min = 0
        self.max = 2**k - 1
```

The constructor of the `HashRing` class uses the parameter `k` which initializes the range to `[0,2^k-1]`. The `Node` class has an attribute `hashValue` that denotes its position on the ring, and a dictionary `resources` that holds its resources. The rest of the code is highly reminiscent of a typical circular doubly-linked list implementation.

The first basic method describes the legal range of resource and node hash values that we allow on the hashring:

```
    def legalRange(self, hashValue):
        if self.min <= hashValue <= self.max:
            return True
        return False
```

To assign the resources to their closest nodes, we define the notion of closest on the hashring using the following `distance` method:

```
    def distance(self, a, b):
        if a == b:
            return 0
        elif a < b:
            return b - a
        else:
            return (2 ** self.k) + (b - a)
```

For example, if we initialize an empty hashring with `k=5`:

```
hr = HashRing(5)
print(hr.distance(29,5))
print(hr.distance(29,12))
print(hr.distance(5,29))
```

we obtain the following output:

```
8
15
24
```

The ring distance from the resource 29 to the node 5 is 8, shorter than the distance from 29 to 12 (and in fact, shorter than to any other node from our example from figure 2.6, which makes node 5 the assigned node of resource 29). Keep in mind that the order of arguments in this function matters.

2.8.3 Lookup

The first functionality to implement with respect to `HashRing` is the lookup of the appropriate node given a hash value of the resource. We march along the hashring starting from the first node (with the smallest hash value), following the forward links as long as the current and the next node are 'on the same side' of the resource. The loop condition is broken when we are about to skip over the resource, that is, the current node precedes the resource and the next node comes immediately after the resource, and that is the node we need to return. If the resource is present, then that is the node containing the resource. This functionality is contained in the `lookupNode` method implemented below:

```
    def lookupNode(self, hashValue):
        if self.legalRange(hashValue):
            temp = self.head
            if temp is None:
                return None
            else:
                while(self.distance(temp.hashValue, hashValue) >
                      self.distance(temp.next.hashValue, hashValue)):
                    temp = temp.next
                if temp.hashValue == hashValue:
                    return temp
                return temp.next
```

In this implementation, we assume no hash collisions –-- no two distinct nodes (and no two distinct resources) will have the same hash value, however it can happen that the resource and a node land on the same position on the hashring, in which case the resource with hash value `i` is assigned to the node `i`.

2.8.4 Adding a new node/resource

When a new node `A` is added to the hashring, some of the resources previously belonging to what is now `A`'s successor might need to be re-assigned to `A`. These are the resources who now have a smaller distance to `A` than to their previously assigned node, i.e., `A` is on their clockwise path to their currently assigned node. See Figure 2.13 for an example of inserting a node with a hash value 30.

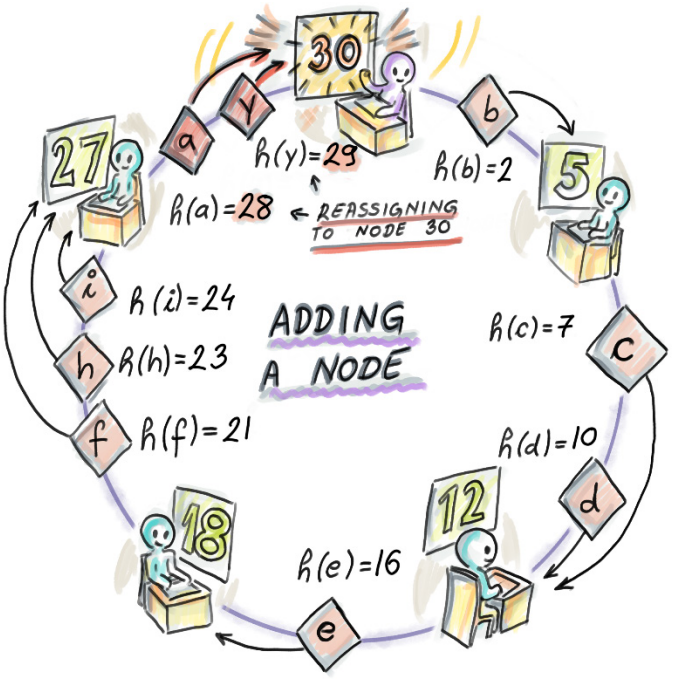

Figure 2.13: New node arrival. The resources a and y, with respective hash values 28 and 29, are now being re-assigned to the newly inserted node with the hash value 30.

Notice that this manner of adding a node is congruent with the constraint (3) from the beginning of the section: when a new node is added, only resources of one other node have potentially changed their mappings, and all other mappings remain untouched.

First let's see how the functionality of moving resources is implemented in a helper method `moveResources` that will also be used later for node deletions:

```
# move some resources to dest from orig
def moveResources(self, dest, orig, deleteTrue):
    delete_list = []
    for i, j in orig.resources.items():
        if (self.distance(i, dest.hashValue) < self.distance(i, orig.hashValue) or
            deleteTrue):
            dest.resources[i] = j
            delete_list.append(i)
            print("\tMoving a resource " + str(i) +  " from " + str(orig.hashValue) + "
                  to " + str(dest.hashValue))
    # delete the re-assigned resources from orig
    for i in delete_list:
        del orig.resources[i]
```

Special cases for node addition involve when the newly added node becomes the head node, or when the existing list is empty. For the common case, we use the lookup function from earlier to locate the correct place for a new node, and then do the needed rewiring of the hashring:

```
def addNode(self, hashValue):
    if self.legalRange(hashValue):
        ptr1 = Node(hashValue)
        temp = self.head

        # empty hashring
        if self.head is None:
            ptr1.next = ptr1
            ptr1.previous = ptr1
            self.head = ptr1
            print("Adding a head node " + str(ptr1.hashValue) + "...")
        else:
            temp = self.lookupNode(hashValue) #successor
            ptr1.next = temp
            ptr1.previous = temp.previous
            ptr1.previous.next = ptr1
            ptr1.next.previous = ptr1
            print("Adding a node " + str(ptr1.hashValue) + ". Its prev is " +
                  str(ptr1.previous.hashValue) + ", and its next is " +
                  str(ptr1.next.hashValue) + ".")
            self.moveResources(ptr1, ptr1.next, False)

            if hashValue < self.head.hashValue: #changing the head pointer
                self.head = ptr1
```

Now that we know how to add nodes, we can also add some resources. To add a new resource, we naturally employ the `lookupNode` method, and update the `resources` dictionary of the appropriate node with the new resource. To add a new resource, we require to have at least one node on the hashring:

```
def addResource(self, hashValueResource):
    if self.legalRange(hashValueResource):
        print("Adding a resource " + str(hashValueResource) + "...")
        targetNode = self.lookupNode(hashValueResource)
        if targetNode is not None:
            value = "Dummy resource value of " + str(hashValueResource)
            targetNode.resources[hashValueResource] = value
        else:
            print("Can't add a resource to an empty hashring")
```

2.8.5 Removing a node

Removal of a node in the hashring works in the following manner: when the node `B` leaves the hashring, which often corresponds to a spontaneous failure of a node, then the resources

previously belonging to B should be assigned to what was B's successor on the hashring (see Figure 2.14). Again, only a small fraction of resources is affected by this change.

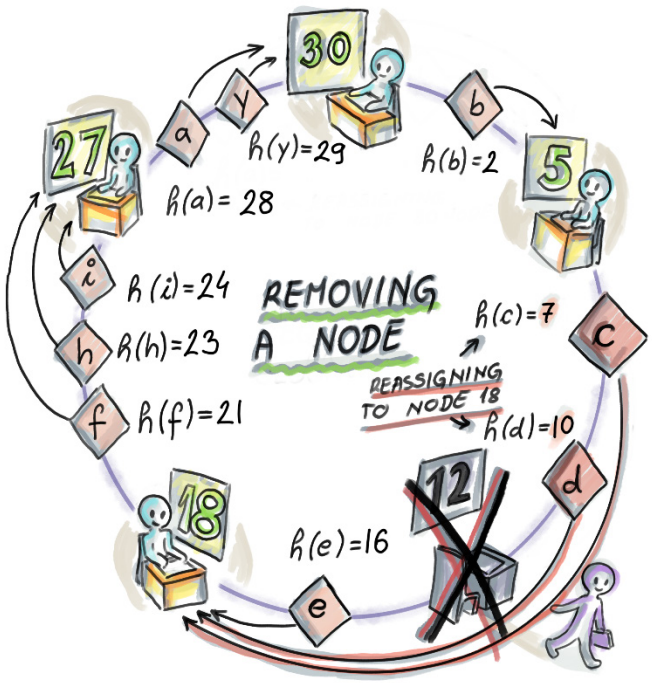

Figure 2.14: Node removal. In this example, the node with the hash value 12 leaves the network, and its resources c and d, with hash values 8 and 10 respectively, are re-assigned to the node with the hash value 18, the previous successor of 12.

The implementation needs to take into account the cases of empty and one-item hashring, attempting to remove a non-existent node, or removing the head item where the head pointer needs to be amended:

```
    def removeNode(self, hashValue):
        temp = self.lookupNode(hashValue)
        if temp.hashValue == hashValue:
            print("Removing the node " + str(hashValue) + ": ")
            self.moveResources(temp.next, temp, True)
            temp.previous.next = temp.next
            temp.next.previous = temp.previous
            if self.head.hashValue == hashValue: # removing the head item
                self.head = temp.next
                if self.head == self.head.next: # if removing from one-item hashring
                    self.head = None
            return temp.next
```

```
        else:
            print("Nothing to remove.") # no such node
```

Lastly, in order to be able to show the contents of hashring, we implement a simple print method that shows the current state of hashring with nodes printed out in the increasing (clockwise order) starting from the northmost point of the ring, along with each node's local resources stored in a local hash table:

```
    def printHashRing(self):
        print("*****")
        print("Printing the hashring in the clockwise order:")
        temp = self.head
        if self.head is None:
            print('Empty hashring')
        else:
            while (True):
                print("Node: " + str(temp.hashValue) + ",", end = " ")
                print("Resources: ", end = " ")
                for i in temp.resources.keys():
                    print(str(i), end = " ")
                    if not bool(temp.resources):
                        print("Empty", end = "")
                    temp = temp.next
                    print(" ")
                    if (temp == self.head):
                        break
        print("*****")
```

With all this functionality under our belt, we are now ready to show an example.

AN EXAMPLE

Let's start by running the process shown in Figures 2.12 and 2.13. First, we add a number of nodes and resources in the arbitrary order and watch how resource re-assignments take place as nodes 5, 27 and 30 get added. Note that any order of additions of nodes and resources (as long as the first object added is a node, not a resource) should result in the same hashring:

```
hr = HashRing(5)
hr.addNode(12)
hr.addNode(18)
hr.addResource(24)
hr.addResource(21)
hr.addResource(16)
hr.addResource(23)
hr.addResource(2)
hr.addResource(29)
hr.addResource(28)
hr.addResource(7)
hr.addResource(10)
```

```
hr.printHashRing()
```

which gives us the following output:

```
Adding a head node 12...
Adding a node 18. Its prev is 12, and its next is 12.
Adding a resource 24...
Adding a resource 21...
Adding a resource 16...
Adding a resource 23...
Adding a resource 2...
Adding a resource 29...
Adding a resource 28...
Adding a resource 7...
Adding a resource 10...
*****
Printing the hashring in the clockwise order:
Node: 12, Resources:  24 21 23 2 29 28 7 10
Node: 18, Resources:  16
*****
```

Now we add two remaining nodes from Figure 2.12 and see how resource re-assignments take place:

```
hr.addNode(5)
hr.addNode(27)
hr.addNode(30)
hr.printHashRing()
```

The output is as follows:

```
Adding a node 5. Its prev is 18, and its next is 12.
       Moving a resource 24 from 12 to 5
       Moving a resource 21 from 12 to 5
       Moving a resource 23 from 12 to 5
       Moving a resource 2 from 12 to 5
       Moving a resource 29 from 12 to 5
       Moving a resource 28 from 12 to 5
Adding a node 27. Its prev is 18, and its next is 5.
       Moving a resource 24 from 5 to 27
       Moving a resource 21 from 5 to 27
       Moving a resource 23 from 5 to 27
Adding a node 30. Its prev is 27, and its next is 5.
       Moving a resource 29 from 5 to 30
       Moving a resource 28 from 5 to 30
*****
Printing the hashring in the clockwise order:
Node: 5, Resources:  2
Node: 12, Resources:  7 10
Node: 18, Resources:  16
Node: 27, Resources:  24 21 23
Node: 30, Resources:  29 28
*****
```

The output above reflects the state of the hashring in Figure 2.13. Now let's remove a node:

```
hr.removeNode(12)
hr.printHashRing()
```

The final hashring, as shown in Figure 2.14, looks as follows:

```
Removing the node 12:
       Moving a resource 7 from 12 to 18
       Moving a resource 10 from 12 to 18
*****
Printing the hashring in the clockwise order:
Node: 5, Resources:  2
Node: 18, Resources:  16 7 10
Node: 27, Resources:  24 21 23
Node: 30, Resources:  29 28
*****
```

2.8.6 Consistent hashing scenario: Chord

Chord[12] is the distributed lookup protocol for peer-to-peer networks that uses consistent hashing. The scheme from Chord, aside from being used in a number of peer-to-peer 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[13].

The simplistic linked-list protocol we implemented leaves a lot to be desired in terms of efficiency for a real production system. To route a request from a resource, we expect to follow a linear number of forward pointers, and each such pointer translates into a network call between two machines. The time required to route the call will not scale in big systems. Also, to route the request, each machine needs to maintain a copy of the hashring, thus consuming a non-trivial amount of local memory.

Chord improves on the basic algorithm by having each node only store the information on other $O(log\ n)$ nodes. Each node `x` maintains a so-called *finger* table that stores the key-value mapping of points on the hashring at exponentially increasing distances from `x` (we call these keys fingers) to their successor nodes. This helps the lookup algorithm find the right node in a logarithmic number of steps.

Specifically, for the hashring with interval $R = [0, 2^k - 1]$, the finger table of a node `x` contains all fingers `f_i` such that `distance(x, f_i) = 2^(i-1)` for all `i≤k`. The fingers'

[12] 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.

[13] 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.

successors can be computed using the `lookupNode` method we earlier implemented. For an example, see Figure 2.15 and the finger table for node `x=5`:

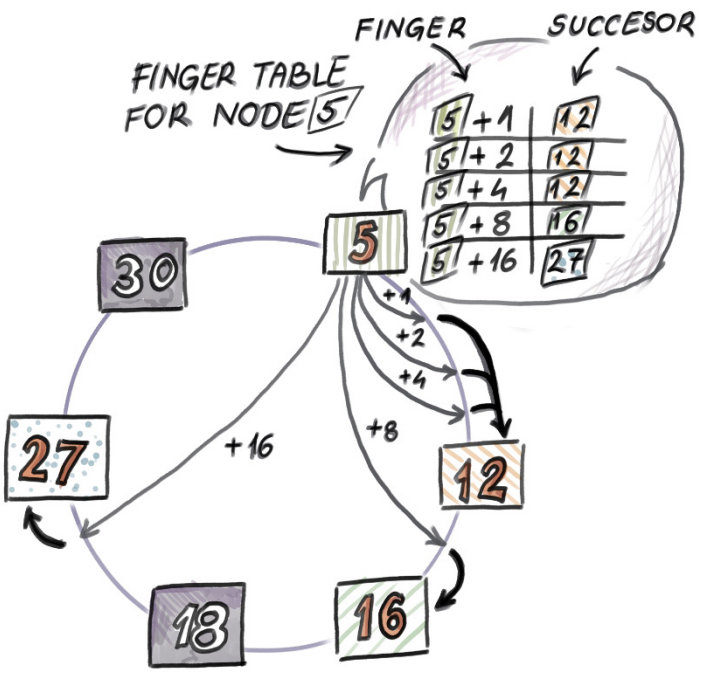

Figure 2.15: Example finger table for the node 5, on the hashring where R=[0,31]. Node 5 has 5 entries stored in its finger table, for the successors of the points 5+1=6, 5+2=7, 5+4=9, 5+8=13, and 5+16=21. The respective successors are 12, 12, 12, 16 and 27.

How can we use finger tables to speed up the lookup? 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 resource `y`, then the node forwards the request to the successor determined by the finger with the smallest distance to the resource. The example is shown in Figure 2.16 with the lookup of the resource with hash value 29 starting at node 5:

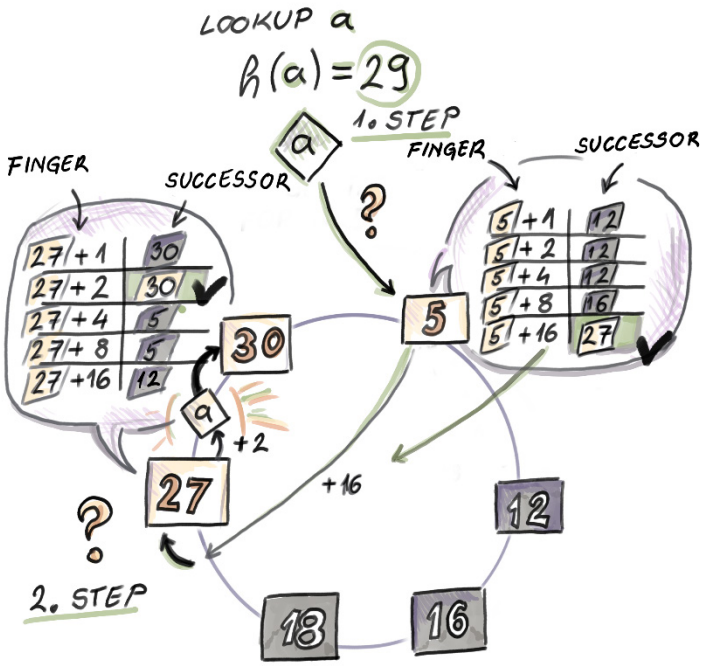

Figure 2.16: Lookup procedure with a finger tables. To locate resource 29 starting from node 5, we first follow the finger (21=5+16) as it is the finger with smallest distance to 29. Its successor is 27, so the request is forwarded to 27. In the finger table of node 27, we take the finger 2, which gives us exactly 29. Its successor is 30, where the request is finally routed, i.e., if the resource exists, it will be found at node 30.

Here are a couple of coding exercises to test your understanding of Chord and finger tables.

CONSISTENT HASHING: PROGRAMMING EXERCISES

Exercise 1:

Given the code for `HashRing` class, add a new attribute `fingerTable` of type `dict` to the `Node` class definition. Now implement a `buildFingerTables(self)` method in the `HashRing` class that builds a finger table for each node in the hashring using the methods we already implemented. Along with the (finger, successor) pair, your finger table should also store the direct pointer to the given node (to allow the direct access to the node from the finger table).

Exercise 2:

Now that each node contains its own finger table, implement a more efficient lookup in a `chordLookup(self,hashValue)` method. Then create a large hashring with dozens

of thousands of nodes and resources, and measure the average number of hops required by the new lookup method. Compare that to the naïve linear-time lookup we implemented.

Exercise 3:

With node additions and removal, finger tables can go out of date and need to be re-built. Modify the implementation of `HashRing` such that finger tables always remain up-to-date.

2.9 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 2.5). 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, such as Python's `dict` (Section 2.6) are about optimizing the common case and do not worry about solving rare pathological cases if they will complicate the common case.
- Murmurhash (Section 2.7) is an example of a widely-used fast and simple non-cryptographic hash function, often employed by hash-based data structures we will learn about in this book.
- Consistent hashing (Section 2.8) solves the problem of hash tables that are distributed among many machines, such is the case in peer-to-peer environments. 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.

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].

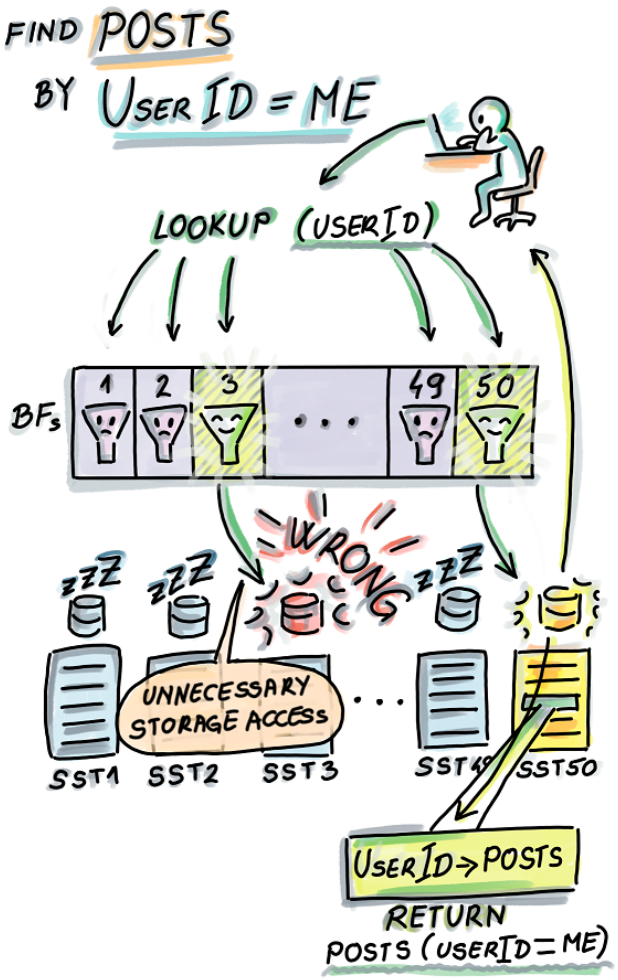

Figure 3.1: Bloom filters in distributed storage systems. In this example, we have 50 sorted string tables (SSTs) on disk, and each table has a dedicated Bloom filter that can fit into RAM due to its much smaller size. When a user does a lookup, the lookup first checks the Bloom filters. In this example, 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.

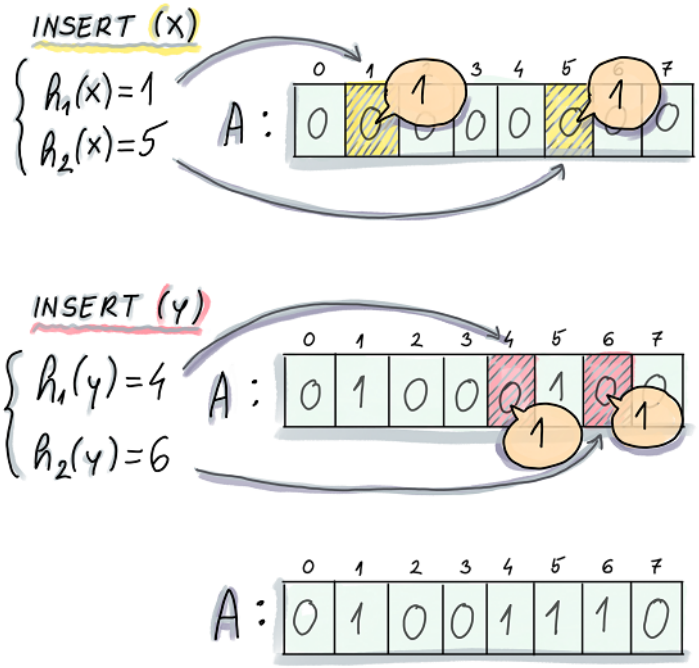

Figure 3.2: Example of insert into Bloom filter. In this example, an initially empty Bloom filter has m=8, and k=2 (two hash functions). To insert an element x, we first compute the two hashes on x, the first one of which generates 1 and the second one generates 5. We proceed to set $A[1]$ and $A[5]$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

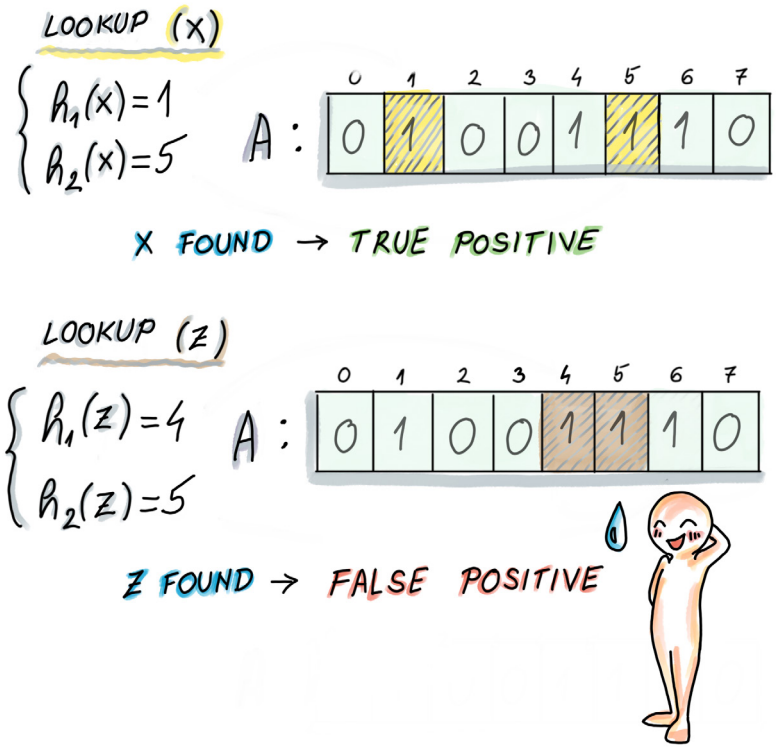

Figure 3.3: Example of a lookup on a Bloom filter. We take the resulting Bloom filter from Figure 3.2, where we inserted elements x and y. To do a lookup on x, we compute the hashes (which are the same as in the case of an insert), and we return Found/Present, as both bits in corresponding locations 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].

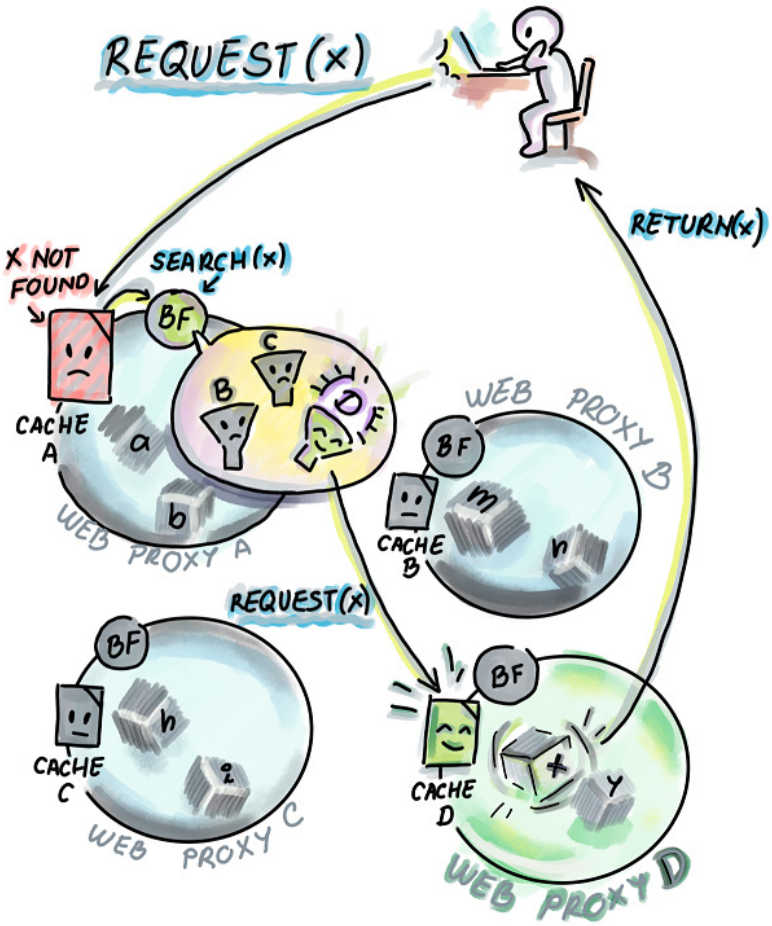

Figure 3.4: Usage of Bloom filter in Squid web proxy. Web proxies keep the copies of recently accessed web pages, but also keep the record of recently accessed web pages of their neighbors by having each proxy occasionally broadcast the Bloom filter of their own cache. In this example, a user requests 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):

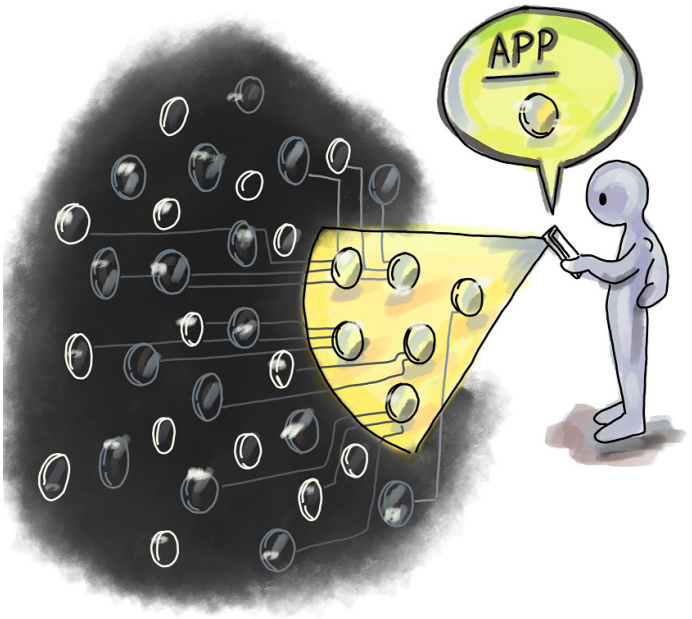

Figure 3.5: In Bitcoin, light clients can broadcast what transactions they are interested in, and thereby block the deluge of updates from the network.

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:

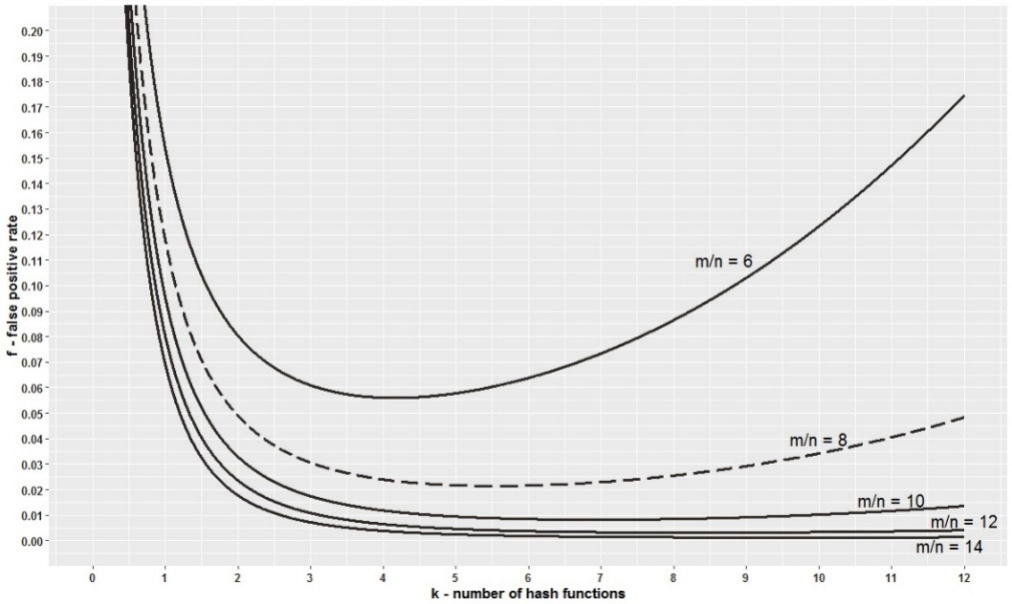

Figure 3.6: The plot relating the number of hash functions (k) and the false positive rate (f) in a Bloom filter. The graph shows the false positive rate for a fixed bits-per-element ratio (m/n), different curves corresponding to different ratios. Starting from the top to bottom, we have m/n=6, 8, 10, 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.

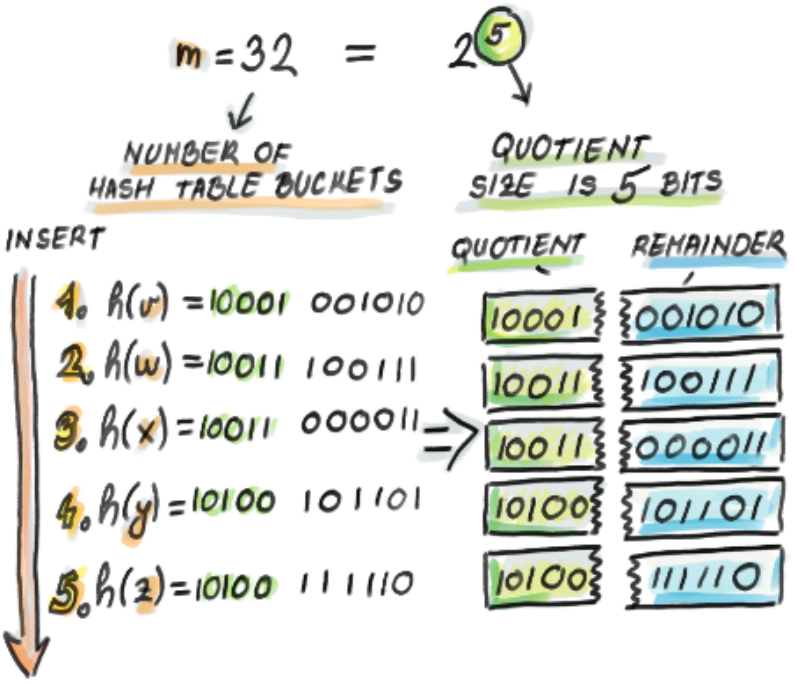

Figure 3.7: Quotienting in a hash table. In this example, the hash table has 32 slots and hashes are 11 bits long. We need 5 bits to distinguish between 32 slots, so the quotient in the hash will be the first 5 bits of a hash, and the remainder will take up the last 6 bits. Quotient determines the bucket 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.

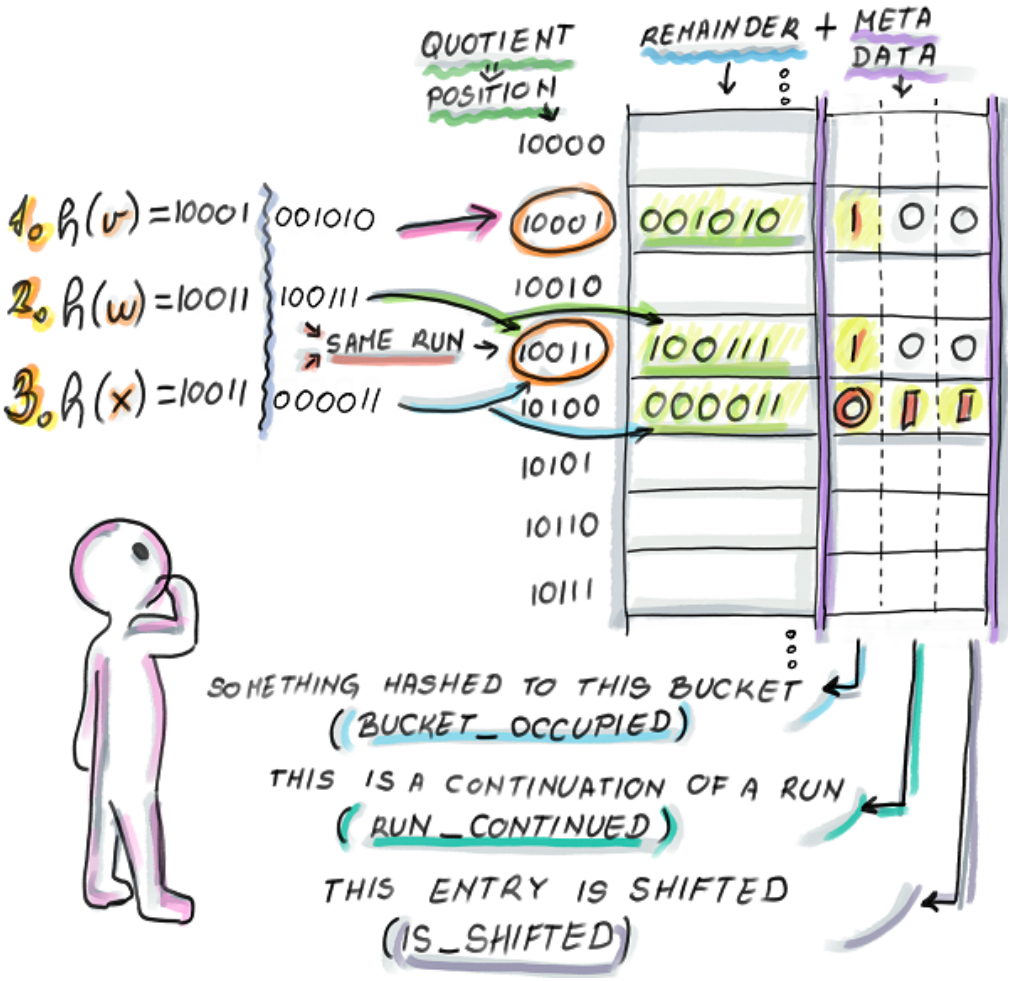

Figure 3.8: Metadata bits in a quotient filter. We inserted three elements into the filter in the following order: v, w and x. First we insert v, and set the bucket_occupied bit to 1, as an item has just hashed in that bucket. We leave the run_continued bit to be 0, because this item is not a continuation 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):

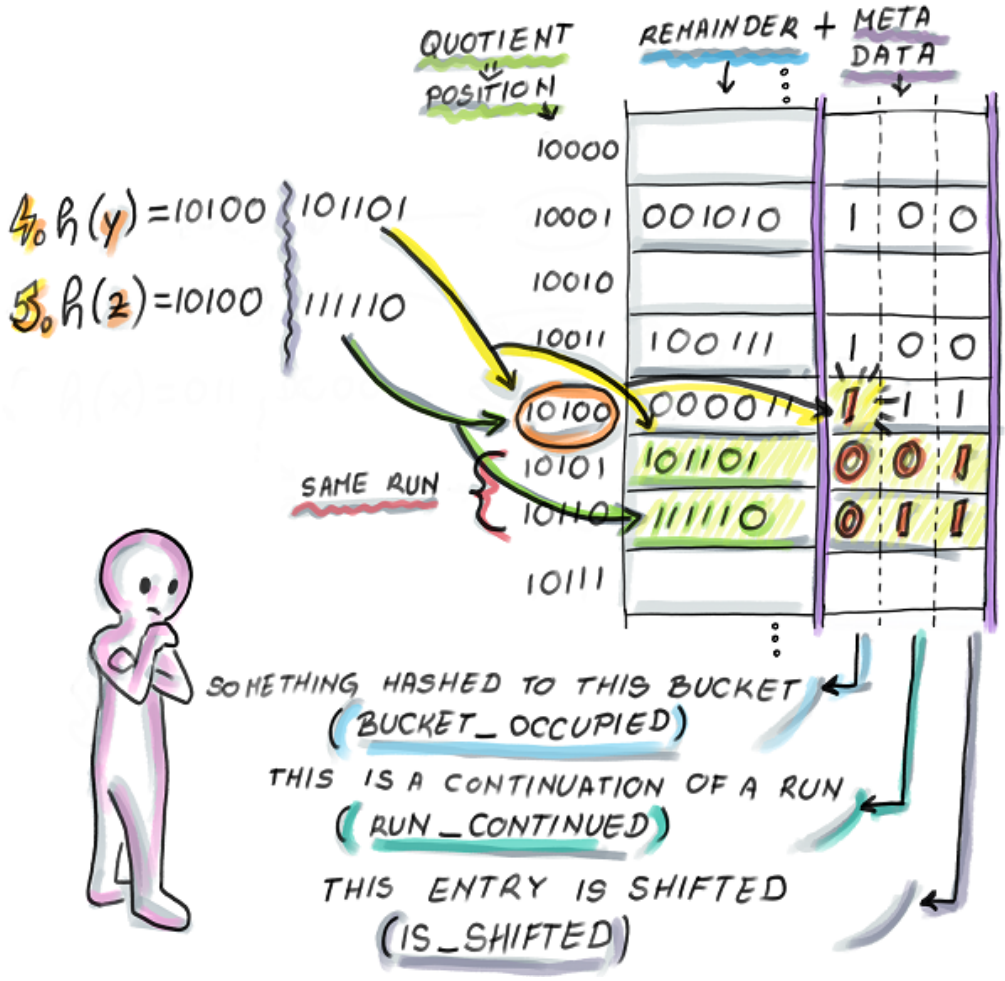

Figure 3.9: Decoding details in quotient filter. When we decode, we work on the level of a cluster. Cluster is a set of items that are not broken up by an empty slot. In this example, we have two clusters: one is just one-slot long, at position 10001, and the other one is 4 slots long, starting from 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).

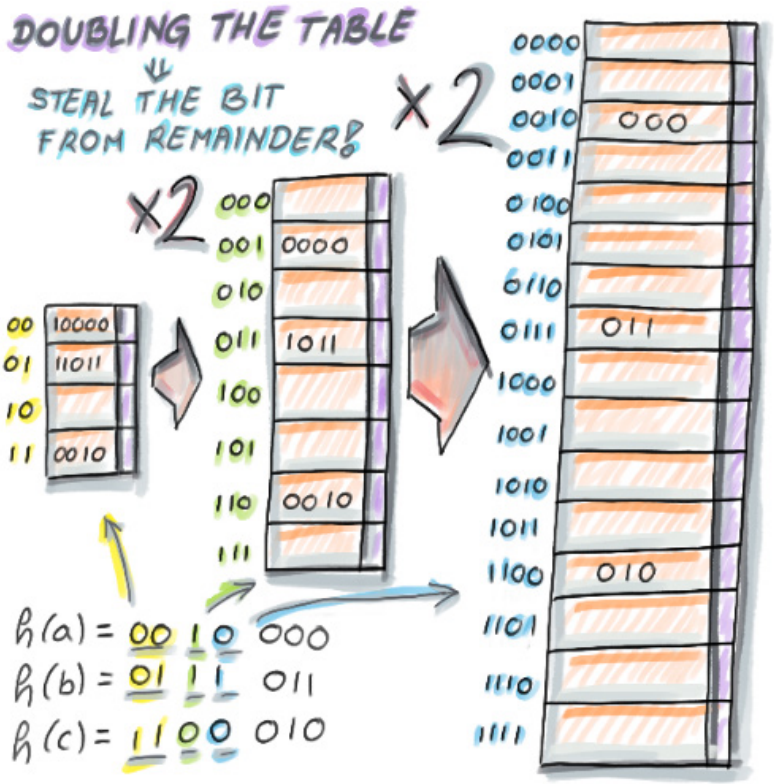

Figure 3.10: Resizing with quotienting. To double the existing table, we steal one bit from the remainder and give it to the quotient. In this example, the original table had 4 slots, and the quotient is 2 bits while the remainder is 5 bits (entire hash is 7 bits long). To double the size of the table, 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:

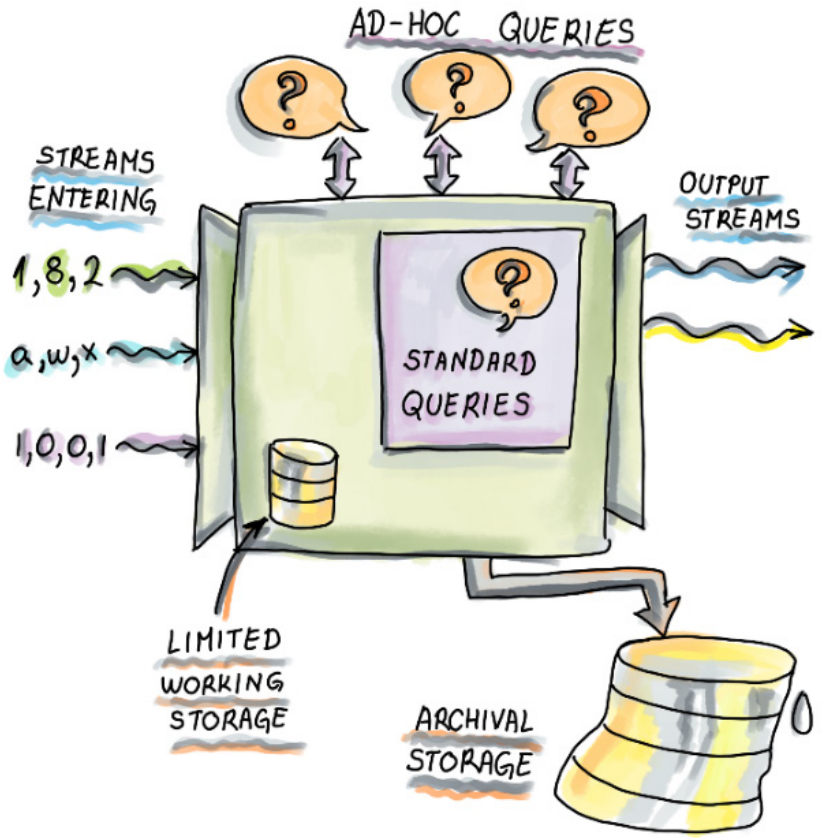

Figure 4.1: Streaming model. The streaming model differs from the traditional database management system in that data passes through the processor and a small amount of working storage, and it is either never stored, or it is stored into the archival storage that is usually too large and slow to be indexed 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:

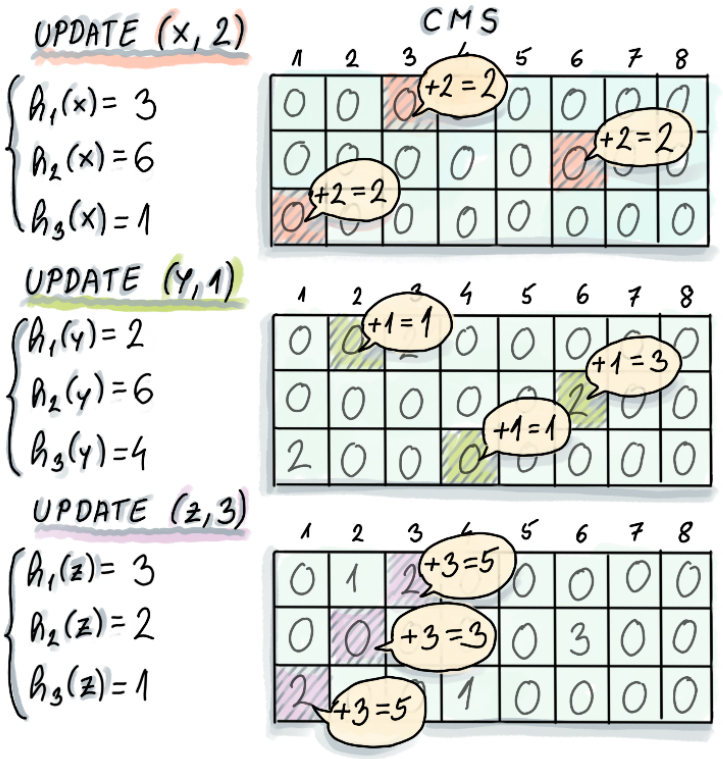

Figure 4.2: Three UPDATE operations of x, y and z performed on an initially empty CMS of dimensions 3x8. Accordingly, we have 3 hash functions that are computed on each element, and the resulting hash is the index into the cell number of the appropriate row. For example, $h_1(x) = 3$, so the location $\mathrm{CMS}[1][3]$ 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(a_t):
min = INT_MAX
for j ← 1 to d
       if(CMS[j][h_j(a_t)] < min)
       min = CMS[j][h_j(a_t)]
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.

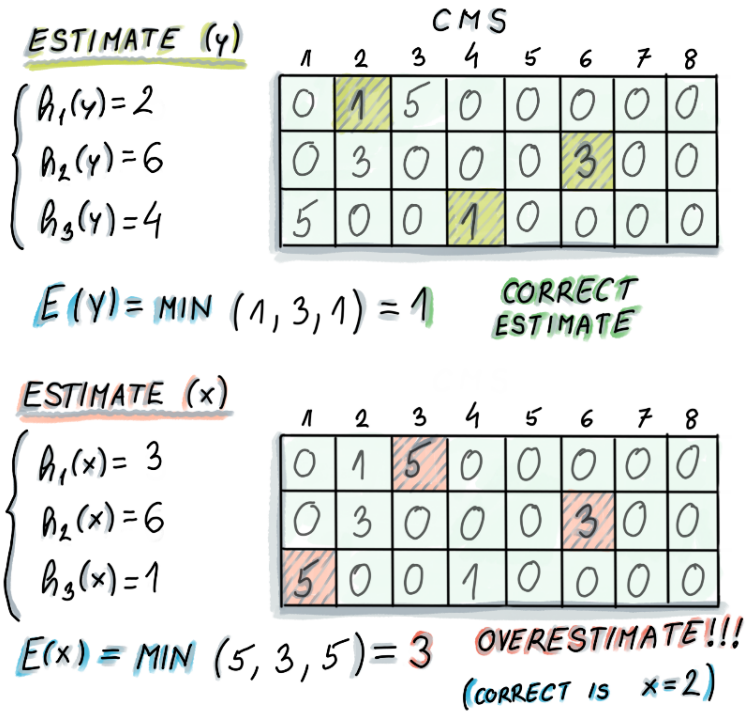

Figure 4.3: Example of estimate operations on the count-min sketch from Figure 4.2. In the case of element y, whose true frequency is 1, count-min sketch reports the correct answer of 1 (the minimum of 1, 3 and 1). However, in the case of the element x, whose true frequency is 2, count-min sketch reports 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/ε 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.

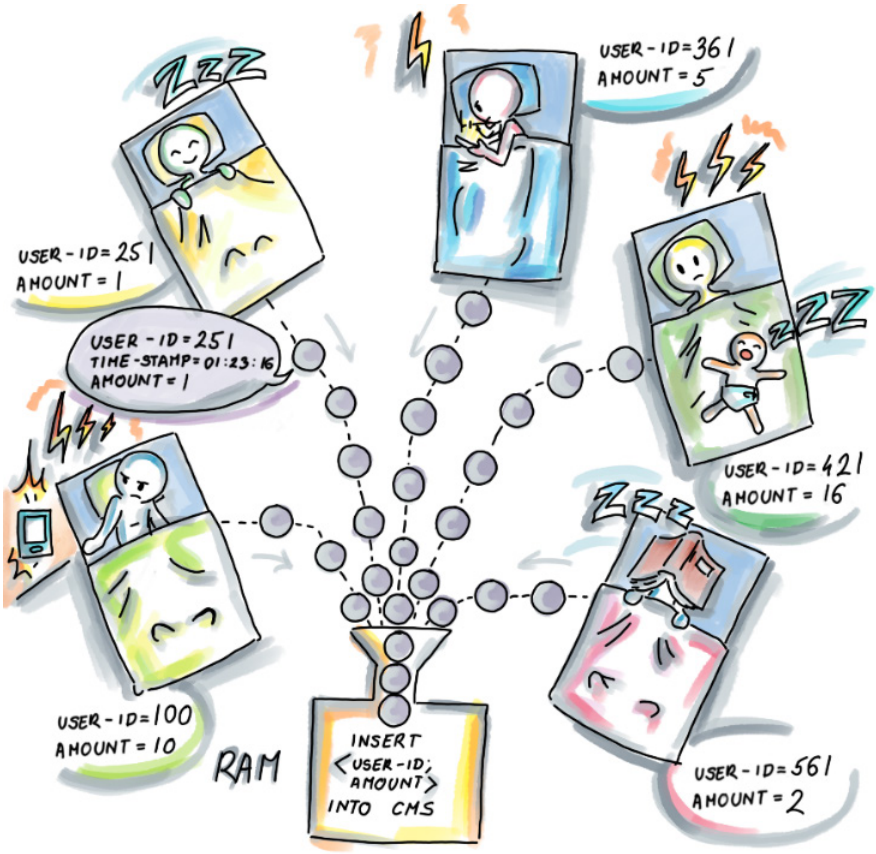

Figure 4.4: All sleep data is sent to a central archive, but before that, input into count-min sketch residing in RAM for later analysis. The (user-id, amount) pair is used as the input for the data structure, where the frequency of user-id increases by amount.

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:

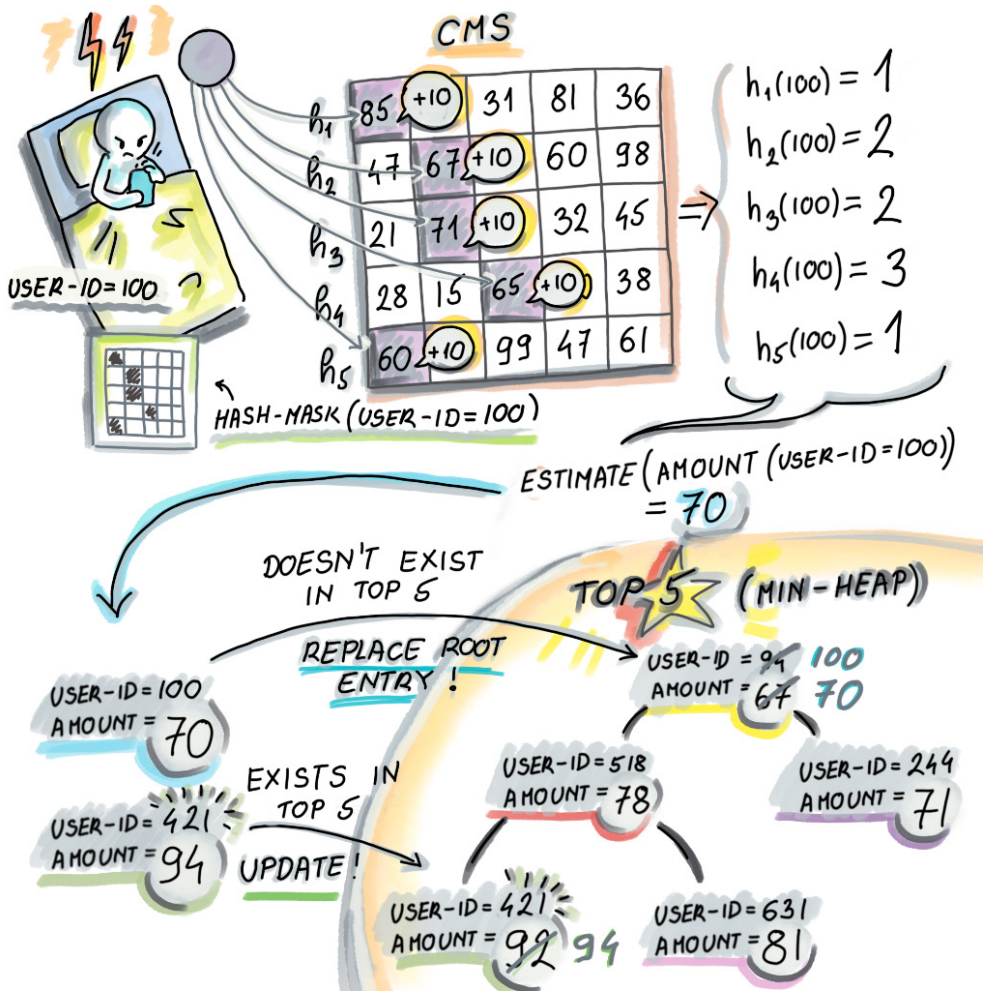

Figure 4.5: Every time the count-min sketch is updated with a (user-id, amount) pair, like with (100, 10) in this example, in order to maintain a correct list of top-k restless sleepers, we do an estimate on frequency of the recently updated user-id. In our case, the estimate for user-id 100 will be 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.

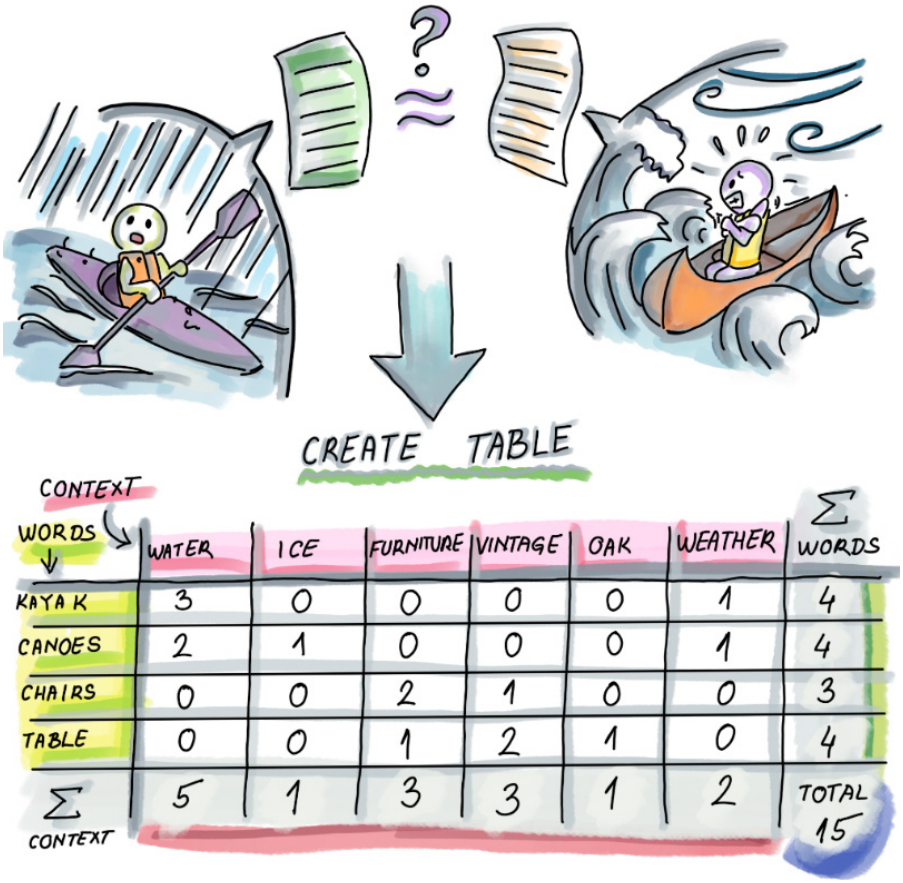

Figure 4.6: To preprocess the text corpus for computing PMI, one way is to create a matrix M where the entry M[A][B] contains the number of times the word A appears in the context B. So for example, 'kayak' appears 3 times in the context of 'water' and 0 times in the context of 'furniture'. We also produce 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:

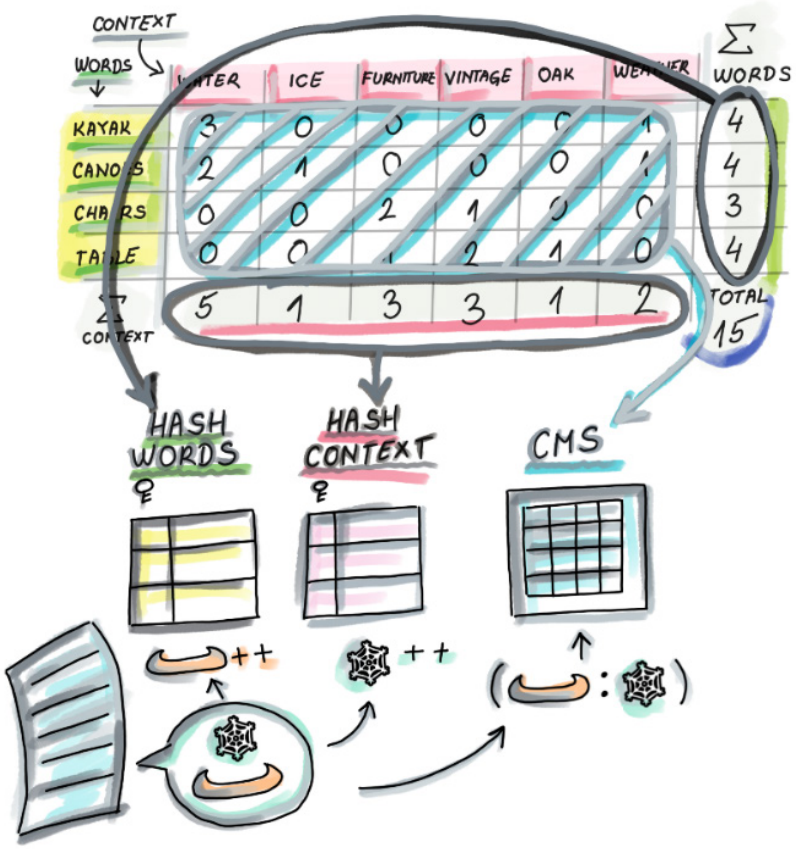

Figure 4.7: The transformation of the matrix from Figure 4.6 to save space: the word-context pairs stored in the main body of the matrix are replaced by a count-min sketch that stores frequencies of word-context pairs. Because the number of distinct words (and contexts) is not that large, we can store 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:

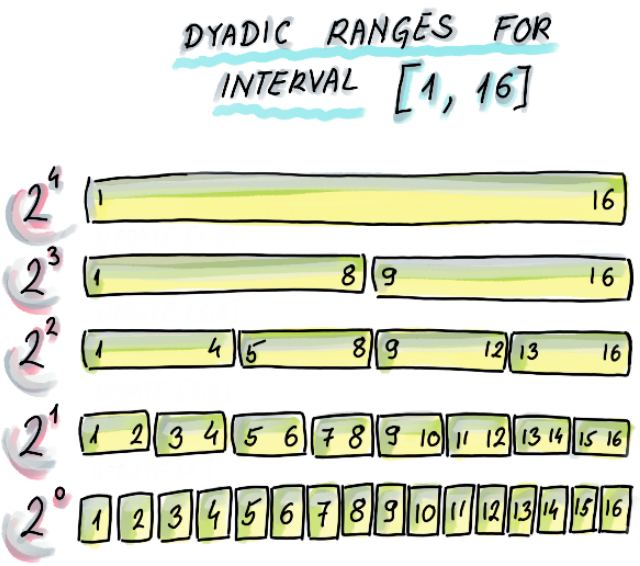

Figure 4.8 Dyadic ranges for the interval [1, 16]. Dyadic ranges of level 0 are the bottom-most with ranges of size 1, then the ranges of level 1 are the level above with the ranges of size 2, and in general dyadic ranges at level i are of size 2^i. Dyadic ranges across different levels are mutually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

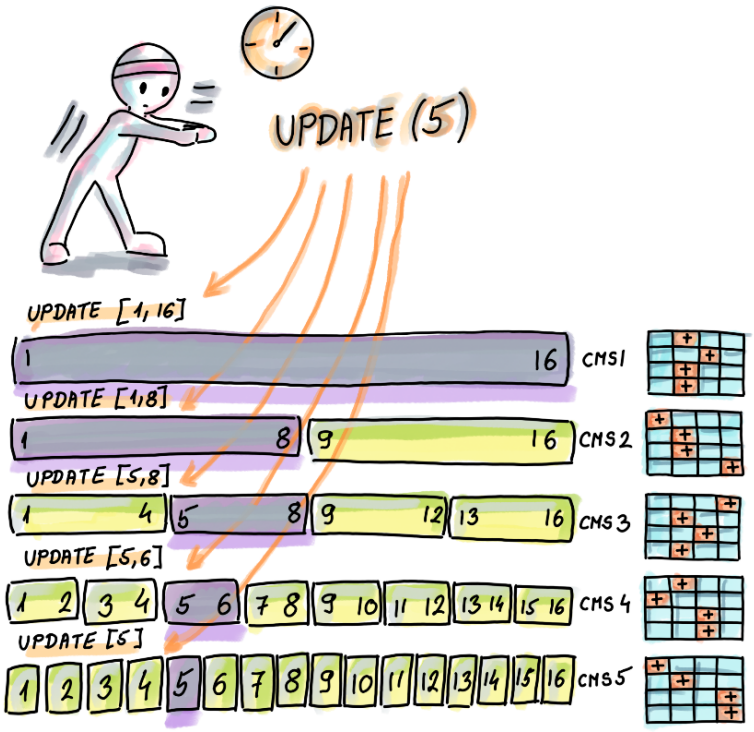

Figure 4.9: An update of one element is transformed into one update per level. For example, if we update 5, we effectively update [1,16] in CMS1, [1,8] in CMS2, [5,8] in CMS3, [5,6] in CMS4, and [5] in CMS5. Instead of updating an element, we are updating a corresponding range to which the element belongs,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

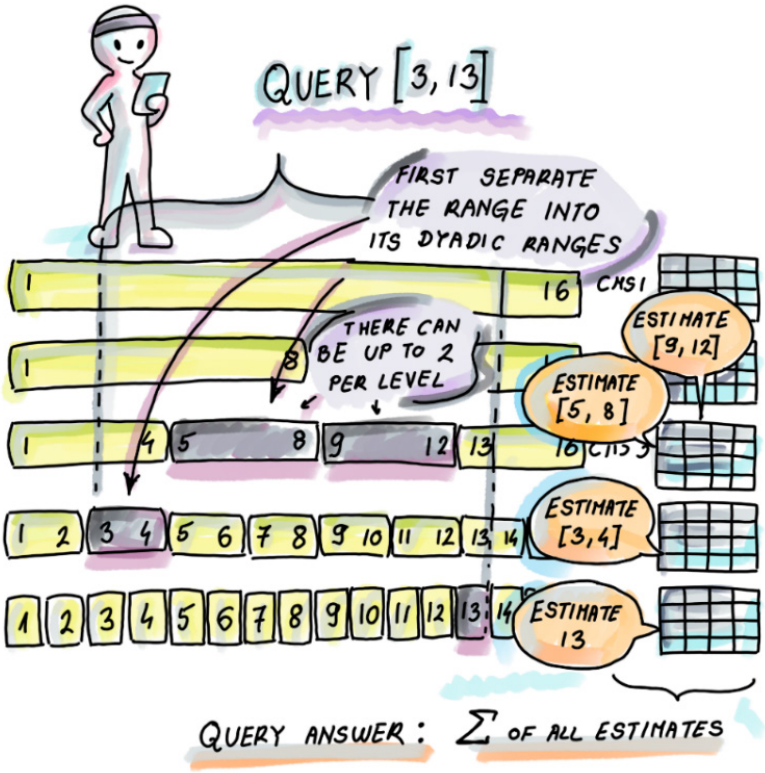

Figure 4.10: In this example, the query range $[3, 13]$ is separated into $[3,4]\cup[5,8]\cup[9,12]\cup[13]$, and we will obtain the frequency estimate for $[3,13]$ by obtaining the frequency estimates for the mentioned ranges and summing them up.

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:

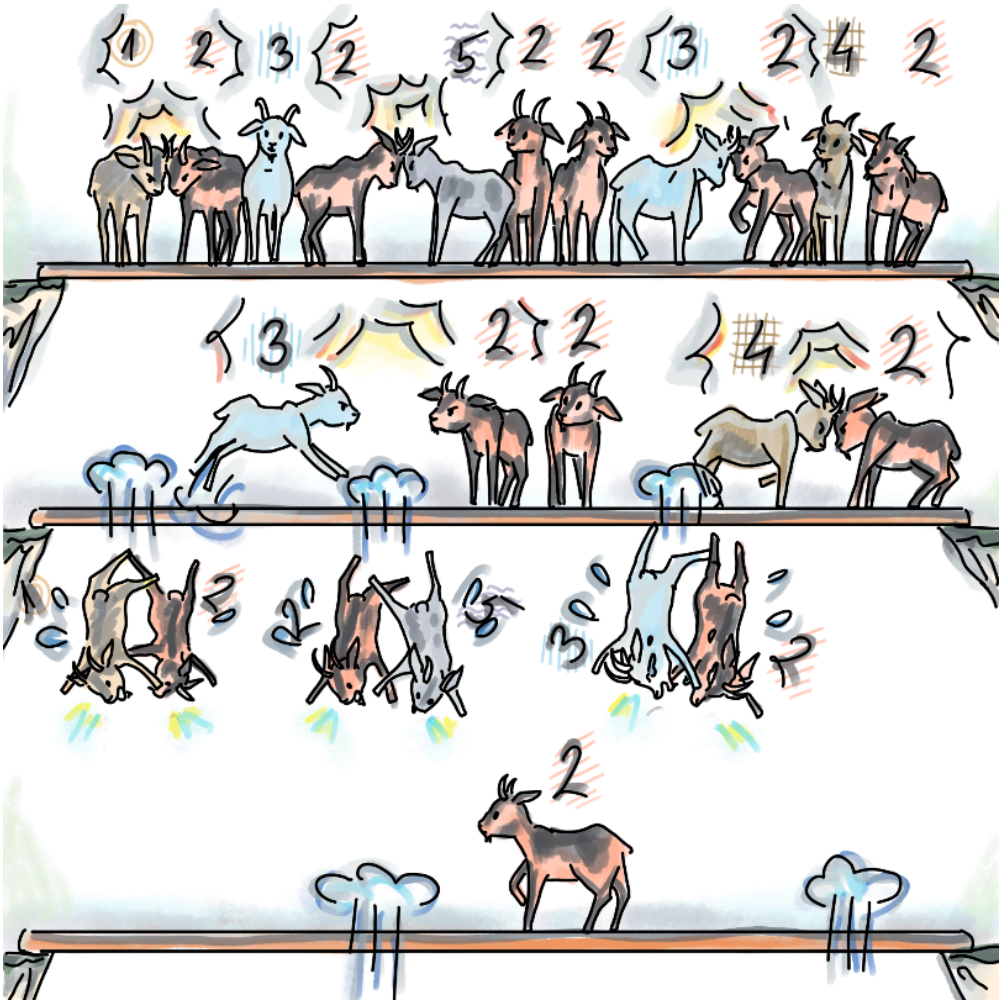

Figure 4.11: We find a majority element in an array by having different neighboring elements throw each other out. In this example, after we throw out (1,2), (2,5), (3,2), and then (3,2) and (4,2), we are left with the majority element 2.

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.

5

Cardinality Estimation and HyperLogLog

This chapter covers:

- Why cardinality estimation is important and the challenges that arise when measuring cardinality on large data
- Practical use cases where space-efficient cardinality estimation algorithms are used
- Teaching the incremental development of ideas leading up to and including HyperLogLog, such as probabilistic counting and LogLog
- How HyperLogLog works, its space and error requirements and where it is used
- Demonstrating how different cardinality estimates behave on large data using a simulation via an experiment
- Insights into practical implementations of HyperLogLog

Determining cardinality of a multiset (a set with duplicates) is a common problem cropping up in all areas of software development, and especially applications involving databases, network traffic, etc. However, since the expansion of internet services, where billions of clicks, searches and purchases are performed daily by a much smaller number of distinct users, there is a renewed interest in this fundamental problem. Specifically, there is a great interest in developing algorithms and data structures that can estimate cardinality of a multiset in one scan of data and in the amount of space substantially smaller than the number of distinct elements.

Cardinality estimation is nowadays used to determine how many distinct visitors are interested in a particular product, how many different users are using particular features of a Web app, or to detect sudden changes in the number of distinct source-destination IP addresses passing through the router (potentially indicating a denial-of-service attack). Because of the way in which information on the Web is replicated over and over, measuring cardinality also helps ascertain how many distinct pieces of content we are dealing with, for example, the number of distinct news articles, or copies of a particular website content.

One the other hand, in many common computation problems, knowledge of the true cardinality can be exploited to predict the runtime of an algorithm and help us choose the right algorithm for our setting. Problems like merging sorted lists with duplicates, or sorting, can be much faster in a set with many duplicates, and its runtime can be expressed as a function of the number of distinct items. Query optimizers in database engines also use information on the number of distinct rows in a table or a column to determine the most efficient query plan when performing complex joins, etc.

With large datasets of today, there is a burgeoning interest in designing algorithms that can accurately approximate set cardinality in the amount of space substantially smaller than the set itself. This chapter will examine one such algorithm called *HyperLogLog*, but first, let's dive into one classical application of measuring cardinality to see why classical solutions to measuring cardinality do not measure up.

5.1 Counting distinct items in databases

Perhaps one of the most familiar examples of measuring cardinality comes from databases and how SQL uses the keyword `DISTINCT`. Applied to a single column in table, `SELECT DISTINCT` returns all the distinct items in that column, while `SELECT COUNT DISTINCT` returns the number of distinct items in the given column.

Queries with `COUNT DISTINCT` are very common, especially in e-commerce when we want to obtain the usage statistics on the website. User visit data is often logged in the `DAILY_VISITS` table that tend to grow very large, with attributes such as: `session_id`, `timestamp`, `product_id`, `user_ip_address`, `visit_duration` and others. By issuing the following `SELECT` operation:

```
SELECT COUNT (DISTINCT user_ip_address) WHERE product_id = 9873947
FROM DAILY_VISITS
```

as a result, we will receive the number of distinct IP addresses (i.e., users) accessing the product with the ID 9873947 on a given day. On a busy website, a daily visit table can get few billion rows long, and this particular query might take a while.

The delay is mostly due to the sorting operation that the classical `COUNT DISTINCT` in most databases does (e.g., Azure SQL/SQL Server) unless the column was previously ordered --- after we sort the column, all duplicates have landed next to each other, and one sequential scan is sufficient to identify and count the distinct items. The sorting operation costs $O(n \log_2 n)$ on a table with n rows and doesn't scale well even on a few million, let alone few billion rows. To make matters worse, even simple queries do many `COUNT DISTINCTs` and `GROUP BYs` on different columns and sorting one column does not help reduce the complexity on sorting another one. We could instead use a hash table to make things faster, but a hash table still requires linear space in the number of distinct elements k. Because k can go up to n, we cannot afford to use hashing either.

An interesting (yet not so favorable) artifact of measuring cardinality is that, even when we only need to know how many distinct items the multiset has, without having to list the distinct items themselves, the complexity doesn't drop if we want to solve the problem exactly. To convince yourself of that, consider the element-distinctness problem, in which

given an array of n elements, we are asked to determine whether all elements in it are distinct; this problem has a lower bound of $\Omega(n \log_2 n)$.[48]

To address the scalability issues, the newer editions of database management systems and warehouses turn to cardinality estimates: SQL Server 2019 included a new `APPROX_COUNT_DISTINCT` operation[49] that uses a very small amount of space and works fast. Google BigQuery goes a step further and makes this approximate and probabilistic approach the default one in `COUNT_DISTINCT`, and reserves `EXACT_COUNT_DISTINCT` for the situations when we absolutely need the exact answer[50]. Running underneath these estimators is the algorithm called HyperLogLog originally invented by Flajolet et al.[51], that offers amazing space savings (think KBs) while processing trillion-sized datasets, and keeps the error fairly low --- on the order of $O(1/\sqrt{m})$ where n denotes the number of 5- or 6-bit-wide memory locations. One common choice for n is 2^{14}.

By this point in the book, saving space in exchange for giving up some accuracy is not a new idea, however HyperLogLog gives a whole new meaning to space-efficiency, almost always staying in the range of few kilobytes while hitting the true cardinality with a small error (e.g., $\pm 2\%$) on average.

What follows in the next section is the incremental development of ideas that lead to HyperLogLog. We will present the original algorithm, some examples, simulations and mathematical intuition around it, as well as mention some of the ways in which HyperLogLog has been implemented and optimized by companies such as Redis, Google, Facebook and others.

Is HyperLogLog a data structure or an algorithm (and does it matter)? Originally, HyperLogLog is referred to as an algorithm, and we will refer to it as an algorithm when we focus on the procedure that is performed on the input data. However, HyperLogLog also needs to store an array with values that are computed on the input data, and this structure is often stored for future use, as we will see in our aggregation example in Section 5.5. In that context, we might also talk about HyperLogLog as a data structure.

5.2 HyperLogLog incremental design

The essential idea of HyperLogLog (HLL) is to use probabilistic and statistical properties of uniform random bit strings to guess the cardinality of a multiset. To that end, elements are initially hashed into bit strings: the original implementation of HyperLogLog uses 32-bit hashes, and the more recent Google's reincarnation called HyperLogLog++[52] and the implementation by Redis[53] use 64-bit hashes to accommodate arbitrarily large cardinalities. Hashes are not random, and it is impossible to obtain random data from non-random data,

[48] Skiena, S. S. (2008). The Algorithm Design Manual, Second Edition. Springer.
[49] https://docs.microsoft.com/en-us/sql/t-sql/functions/approx-count-distinct-transact-sql?view=sql-server-ver15
[50] https://cloud.google.com/bigquery/docs/reference/legacy-sql#countdistinct
[51] Flajolet, P., Fusy, E., Gandouet, O., & Meunier, F. (2007). HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. AOFA '07: Proceedings of the 2007 International Conference on Analysis of Algorithms.
[52] Heule, S., Nunkesser, M., & Hall, A. (2013). HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm. Proceedings of the 16th International Conference on Extending Database Technology (pp. 683-692). Genoa, Italy: Association for Computing Machinery
[53] http://antirez.com/news/75

however, they mimick the randomness well enough for our purposes (i.e., they *look* random).

Given a multiset $M = \{a_1,a_2,\ldots,a_n\}$ with n elements and k distinct elements (we do not know k), using a hash function $h:U \rightarrow \{0,1\}^L$, we produce a hashed set $h(M)=\{h_1,h_2,\ldots,h_n\}$ where $h_i = h(a_i)$ with hash length $L = |h_i|$. For a large enough L (e.g., L=64), each item will map to a distinct hash with high probability, so that the number of distinct hashes will also be k or very close. Hashing by itself does not help us estimate cardinality just yet, but now we switched from estimating the number of distinct input elements to estimating the number of distinct hashes.

5.2.1 The first cut –- probabilistic counting

The roughest estimate called *probabilistic counting*[54] observes the bit patterns in the hash by computing ρ_i for each hash h_i such that:

$$\rho_i = (\text{The number of trailing zeros in } h_i) + 1$$

That is, ρ_i will denote the position of the first 1 encountered from the right (if the hash does not contain any 1s, then $\rho_i = L+1$.) Without loss of generality, we will use right instead of left in this and other places in chapter. So for example, for $h_1 = 1100$, $h_2 = 0111$ and $h_3 = 0000$, the respective values of ρ_i are $\rho_1 = 3$, $\rho_2 = 1$ and $\rho_3 = 5$. The cardinality estimate E will depend on $\rho_{max}=\max(\rho_1, \rho_2,\ldots, \rho_n)$, and it's equal to:

$$E = 2^{\rho\max}$$

Here is the idea of probabilistic counting expressed in pseudocode:

```
int p_max = 0
for a_i in M
     h_i = hash(a_i)
     p_i = num_trailing_zeros(h_i) + 1
     if(p_i > p_max)
          p_max = p_i
return power(2,p_max)
```

Example 1.

The example below shows probabilistic counting in action (Figure 5.1) where $n = 12$, and $k = 7$, and the final estimate of 2^5=32, with the lemon item significantly affecting the estimate:

[54] Flajolet, P., & Martin, G. N. (1985). Probabilistic Counting Algorithms for Data Base Applications. Journal of Computer and System Sciences, 182-209.

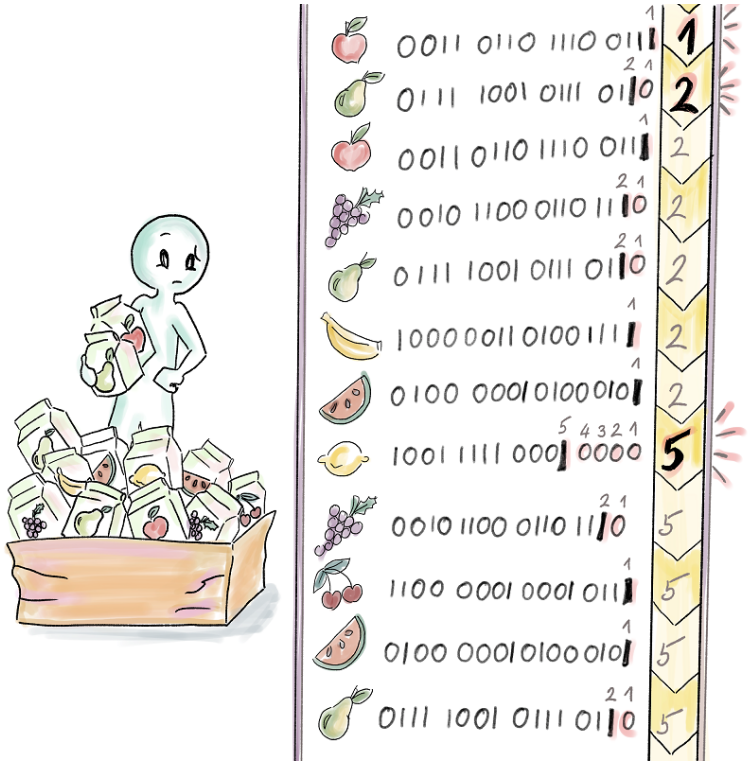

Figure 5.1: A dataset of 12 items is hashed into 16-bit hashes. As we scan the dataset, we keep the running maximum of the ρ_i. In this example, the lemon item, whose hash is $1001\ 1111\ 0001\ 0000$ holds the maximum $\rho_{max} = 5$, and our cardinality estimate is $E = 2^{\rho_{max}} = 32$, but the true distinct count is $k = 7$.

This is not as close to the truth as we would like, but we are just getting started. The rough intuition behind probabilistic counting is that if we managed to get an unusual hash (i.e., hash with many trailing zeros), then that is an indicator of the presence of many other hashes in the set.

(Putting on our probability hats…) In a uniformly randomly generated set of k bit strings, on average about $k/2$ bit strings have 0 as their last digit, and the other $k/2$ have 1. Out of the former $k/2$, half on average (i.e., $k/4$) have 00 as their two last digits and the other $k/4$ have 10, and so on. Ultimately, $k/2^i$ items on average have their last i digits all 0s and another $k/2^i$ have last i digits of the form 10^{i-1}.

Accordingly, the probability of generating a hash where $\rho_i = 1$ (hash ends with 1) is $1/2$, the probability of a hash where $\rho_i = 2$ (hash ends with 10) is $1/4$, and probability of a hash where $\rho_i = i$ (ends with 10^{i-1}) is $1/2^i$. For the event that occurs with probability $1/2^i$, on average we need 2^i repetitions for it to occur, so working backwards, having an element with ρ_i =

ρ_{max} on average implies the cardinality of $2^{\rho_{max}}$, corresponding to probabilistic counting estimate.

However, this is only the average behavior of random variables (i.e., expectation), and often, expectation is not what we expect (!). Deviations from this average will occur, and even a small deviation can significantly affect the estimate, considering that ρ_{max} is in the exponent. In general, we observe the estimate error of HyperLogLog as the **relative error** --- the fraction of the true cardinality by which the estimate is off in any direction $(\pm(E-k)/k)$; this fraction can get very large for small cardinalities.

5.2.2 Stochastic averaging or, when life gives you lemons...

There is a couple of problems with our first-cut solution --- even without outliers affecting the estimate, all estimates are powers of 2, which, for many cardinalities makes it hopeless to hit close to the right answer. To address the outlier issue, we will resort to a method called *stochastic averaging*, which divides the hash set uniformly randomly into $m = 2^b$ subsets of roughly the same size, by throwing hashes in the buckets determined by the first b bits of each hash. Once each hash is assigned to a bucket, we will perform probabilistic counting on each bucket individually: instead of 1 estimator ρ_{max}, we will have m estimators $\rho_{i,max}$, $1 \le i \le m$, where $\rho_{i,max}$ represents the ρ_{max} of hashes from the i^{th} bucket.

You can think of partitioning into subsets as a poor-man's hashing the whole set m times and obtainining m estimators that we can further combine. In reality, we cannot afford m hash functions and the computational cost of hashing each item m times.

Now that we have m estimators, first we will compute their arithmetic average:

$$A = \frac{\sum_{i=1}^{m} \rho_{i,max}}{m}$$

and use it to obtain the average bucket estimate

$$E_{bucket} = 2^A$$

the equivalent of a geometric mean of probabilistic counting estimates for individual buckets. To obtain the overall estimate E, we need to account for all m buckets:

$$E = m * E_{bucket} = m * 2^{\frac{\sum_{i=1}^{m} \rho_{i,max}}{m}}$$

Example1 (continued).

Let's see how this works on our example from above when $b = 2$, hence there are $m = 4$ buckets, and Figure 5.2 illustrates the contents and $\rho_{i,max}$ for each bucket. To compute the estimate, we first compute $A = (2+2+5+1) / 4 = 2.5$. From there, we have that $E_{bucket} = 2^A = 2^{2.5} \approx 5.66$, and $E = m * E_{bucket} = 4 * 5.66 = 22.64$, more accurate than our earlier estimate of 32.

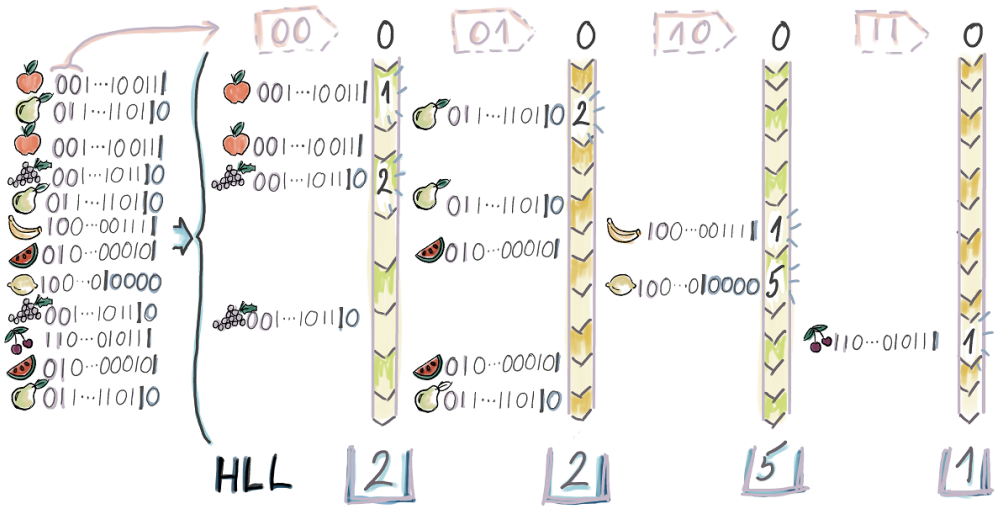

Figure 5.2: In this example, each hash is being mapped to a bucket based on its first two bits (e.g., the hash corresponding to the grape item is mapped to the bucket 00, while the hash corresponding to the pear item is mapped to the bucket 01.) When this process runs on large datasets, we would expect each bucket to receive the same number of distinct hashes. Each bucket computes its $\rho_{i,max}$, which in this case results in bucket values 2, 2, 5 and 1. Now the hash of the lemon item is only affecting the value stored in the bucket 10.

The pseudocode below shows how stochastic averaging works:

```
int b = [4..16]//number of bits that determine the bucket, usually between 4 and 16
int m = power(2,b) // number of buckets
int S[m]: initialize all fields to 0

for a_i in M // iterating over multiset M
     h_i = hash(a_i)
     p_i = num_trailing_zeros(h_i) + 1
     bucket = <h_i,1, h_i,2,…,h_i,b> // integer described by first b bits of h_i
     if(p_i > S[bucket])
          S[bucket] = p_i

double sum = 0
for S[i] in S
     sum = sum + S[i]

double arit_avg = sum / m
return m * power(2,avg)
```

5.2.3 LogLog

LogLog algorithm uses stochastic averaging in combination with a normalization constant $\tilde{\alpha}_m$ introduced to undo the systematic overestimate bias that occurs when we estimate cardinality with $\rho_{i,max}$ random variable (the maximum of geometric variables of parameter$1/2$). Hence, we modify the original estimate to the following formula:

$$E = \tilde{\alpha}_m * m * 2^{\frac{\sum_{i=1}^{m} \rho_{i,max}}{m}}$$

where the constant $\tilde{\alpha}_m$ is parameterized by m and it equals:

$$\tilde{\alpha}_m \sim 0.39701 - \frac{2\pi^2 + (\ln 2)^2}{48m}$$

For most practical purposes (specifically, when $m \geq 64$), one can use just $\tilde{\alpha}_m = 0.39701$. More details on how the expression for $\tilde{\alpha}_m$ is derived can be found in the original LogLog paper.[55]

Example 1 (continued).

To obtain the LogLog estimate for our running example (from Figure 5.2), we compute $\tilde{\alpha}_4$ to be approximately 0.292, so the LogLog estimate is 0.292 $*$ 22.6 ≈ 6.6, extremely close to the true cardinality!

ERROR AND SPACE CONSIDERATIONS IN LOGLOG

Using statistical analysis, it has been found that the relative error in LogLog can be closely approximated by $1.3/\sqrt{m}$. To put this in perspective for many modern implementations, the value of m is often set to 2^{14}, and we can expect the relative error to be $1.3/\sqrt{2^{14}} = 1.01\%$, regardless of the dataset size. If we take into account that 2^{14} 8-byte integer locations only take up about 130KBs, LogLog might seem like magic!

Still, it is important to recognize that we do not need 8 bytes for bucket counters. In fact, we need 5 or 6 bits, depending on how large the cardinalities are that we're estimating. If the upper cardinality limit of our dataset is k_{max}, then we need $O(\log_2 k_{max})$ to be the length of a hash to differentiate up to that cardinality, and then further we need $O(\log_2 \log_2 k_{max})$ bits to store the maximum value in the bucket (hence the log-log). A safe upper cardinality limit is $k_{max} = 2^{64}$, so one bucket needs 6 bits. The total storage requirement of LogLog is:

$$O(m \log_2 \log_2 k_{max})$$

Plugging in for the common value of $m = 2^{14}$, it turns out we need approximately 12KBs to store LogLog.

To be more exact, we would expect the maximum cardinality within one bucket to be closer to k_{max}/m (k_{max} is the worst case), which reduces the space requirement to:

$$O\left(m \log_2 \log_2 \left(\frac{k_{max}}{m}\right)\right)$$

In our example where $k_{max}/m = 2^{50}$, this, however, does not help as the logarithms are rounded up to their integer values (logarithm of 50 in this case will be rounded up to 6.)

[55] Durand, M., & Flajolet, P. (2003). Loglog Counting of Large Cardinalities. European Symposium on Algorithms (ESA) 2003 (pp. 605-617). Springer Berlin Heidelberg.

SUPERLOGLOG

One way to improve on the error of LogLog is to retain only a percentage θ of the lowest bucket values and base the estimate on those $m_\theta = \theta m$ buckets. This is called truncation rule. A similar approach called restriction rule uses only bucket values not larger than $\lceil \log_2 (k_{max} / \mathrm{m}) + 3 \rceil$, which removes outliers but also allows us to use buckets that are $\lceil \log_2 \lceil \log_2 (k_{max} / \mathrm{m}) + 3 \rceil \rceil$ bits wide. There is experimental evidence that the error drops to $1.05/\sqrt{\mathrm{m}}$ when employing truncation and the restriction rule.

Even though an improvement over the basic probabilistic counting approach, the arithmetic mean in the exponent can still draw the final estimate arbitrarily far from the mean because the arithmetic mean is very sensitive to outliers. It is similar in the 3D context, where the centroid (the 3D version of arithmetic mean) can end up arbitrarily far from the center of the mass due to one point being far away from all the others. Our final improvement, HyperLogLog, will use the harmonic mean of the bucket values to compute the estimate.

5.2.4 HyperLogLog -- Stochastic averaging with harmonic mean

The formula for harmonic mean applied to our buckets, that represents our new bucket average is as follows:

$$E_{bucket} = \frac{m}{\sum_{i=1}^{m} 2^{-\rho_{i,max}}}$$

For the final estimate, we will apply the appropriate bias-correcting factor α_m and account for all m buckets:

$$E = \alpha_m * m * E_{bucket} = \frac{\alpha_m m^2}{\sum_{i=1}^{m} 2^{-\rho_{i,max}}}$$

The bias-correcting factor is different than that of LogLog, and it can be approximated as follows:

$$\alpha_m = \frac{1}{2 \ln 2 \left(1 + \frac{1}{m}(3 \ln 2 - 1) + O(m^{-2})\right)}$$

For very large values of m, $\alpha_m = 1/2 \ln 2 = 0.72134$ is a good approximation, but it is also useful to build into our code some typical values of α_m:

$\alpha_{16} = 0.673$
$\alpha_{32} = 0.697$
$\alpha_{64} = 0.709$
$\alpha_m = 0.7213/(1 + 1.079/\mathrm{m})$ for $m \geq 128$

Example 1 (continued).

Applying the harmonic mean to our running example from Figure 5.2, we get:

$$E_{bucket} = \frac{4}{\left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^5 + \left(\frac{1}{2}\right)^1} = \frac{4}{\frac{33}{32}} \approx 3.88$$

We also get $\alpha_4 = 0.541$ from the formula for α_m, which further gives the following estimate:

$$E = 0.541 * 4 * 3.88 = 8.39$$

A little further in fact than our earlier LogLog estimate (6.6), but as datasets get bigger, as we will see in simulations in Section 5.4, HyperLogLog is a less biased estimator and with the smaller relative error. The statistical analysis shows that the relative error in the HyperLogLog algorithm is down to $1.04/\sqrt{m}$. For more details on how the error in HyperLogLog is derived, you can consult the original HyperLogLog paper.[56]

This is the end of our story about how we obtain the raw estimate in HyperLogLog, whose pseudocode is shown below. There are few minor tweaks after we obtained the raw estimate, specifically, when the cardinality we are computing is too small or too large, as shown in the final HyperLogLog pseudocode below:

```
/* RAW ESTIMATE */
define alpha_16 = 0.673, alpha_32 = 0.697, alpha_64 = 0.709,
       alpha_m = 0.7213/(1 + 1.079/m) for m>= 128

int b = [4..16]//number of bits that determine the bucket, usually between 4 and 16
int m = power(2,b) // number of buckets
int S[m]: initialize all fields to 0

for a_i in M // iterating over multiset M
     h_i = hash(a_i) /* 32-bit hash h1x2…x32 */
     p_i = num_trailing_zeros(h_i) + 1
     bucket = <h_i,1, h_i,2,…,h_i,b> // integer described by first b bits of h_i
     if(p_i > S[bucket])
          S[bucket] = p_i

double sum = 0
for S[i] in S
     sum = sum + power(2,-S[i])
double harmonic_avg = m /sum
double E = alpha_m * m * harmonic_avg      /*raw estimate*/

/* CORRECTED ESTIMATE */
double E_final
if E <= 5m/2 then  // small range correction
     Let V be the number of registers equal to 0
     if V ≠ 0 then
          E_final = mlog(m/V)
     else
          E_final = E
```

[56] Flajolet, P., Fusy, E., Gandouet, O., & Meunier, F. (2007). HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. AOFA '07: Proceedings of the 2007 International Conference on Analysis of Algorithms.

```
if E <= power(2,32)/30 // intermediate range, no correction
     set E_final = E
if E > power(2,32)/30 then // large range correction
     E_final = -power(2,32)log(1-E/power(2,32))
return E_final // estimate with relative error of ±1.04/sqrt(m)
```

In the case of very small cardinalities (in relation to the number of buckets), many buckets will remain empty, and in that case, we will resort to the probabilistic method called *linear counting* to establish the true cardinality. Namely, this approach follows the logic of balls-and-bins setup where, if we throw n balls into m bins uniformly randomly, based on how many buckets remained empty, we can estimate the total number of balls. More details can be found in the paper on linear counting.[57]

An interesting artifact of using linear counting is that right at the cross-over point, when the cardinality becomes large enough to switch to HyperLogLog estimate, there is a large spike in bias. The authors of HyperLogLog++ tried to alleviate this issue by experimentally ascertaining average amounts of bias for each cardinality around that point, and then returning the estimate by that bias amount. Redis implementation instead uses polynomial regression that approximates the curve of the bias and then returning the estimates by that predicted amount.

Considering that our pseudocode reflects the original paper's implementation of HyperLogLog, one issue that might arise when using 32-bit hashes, as they are in the original paper, is that for very large cardinalities, hashes start colliding, so we start losing accuracy even on the hashing level, and a correction to the estimate is also needed in that case. This is however not a problem if we use a 64-bit hash, as it is used in all modern implementations by Google, Redis, Facebook[58] and others.

ERROR AND SPACE CONSIDERATIONS IN HYPERLOGLOG

Statistical proofs show that HyperLogLog has the relative error around $1.04/\sqrt{m}$. Space consumption is exactly the same as in LogLog:

$$O(m \log_2 \log_2 k_{max})$$

And just like in LogLog, we can use 6-bit fields for buckets. Are we being stingy by insisting on custom 6-bit fields as oppose to standard 8-bit fields for an algorithm that already has a very small memory footprint, and are we sacrificing the valuable CPU time by unpacking those bits? Answers to these questions are largely dependent on a particular application. For example, when embedding algorithms in hardware, or when aggregating a large number of HyperLogLogs into one, such differences add up and every space optimization trick is very much worth it.

Before experimentally testing the features of data structures/algorithms introduced in this section, we will break up the technical discussion with an example of a context where HLL can be used.

---

[57] Whang, K.-Y., Vander-Zanden, B. T., & Taylor, H. M. (1990). A Linear-Time Probabilistic Counting Algorithm for Database Applications. ACM Trans. Database Syst., 208-229.

[58] https://engineering.fb.com/data-ifnfrastructure/hyperloglog/

5.3 Use case: catching worms with HLL

Applications and intrusion detection systems that monitor network traffic keep track of changes in various network parameters that might reveal impending security breaches, in an organization's network for example. One indicator of network health is related to the source-destination IP address pairs available on packet headers passing through a router.

Stable network traffic is marked by a (potentially large) number of packets exchanged between a much smaller number of pairs of computers. Depending on a type of security threat, having one source open a large number of connections to (sometimes random) destinations in a short time interval, or simply a significant rise in the number of distinct source-destinations IP address pairs might indicate a virus (see Figures 5.3 and 5.4).[59]

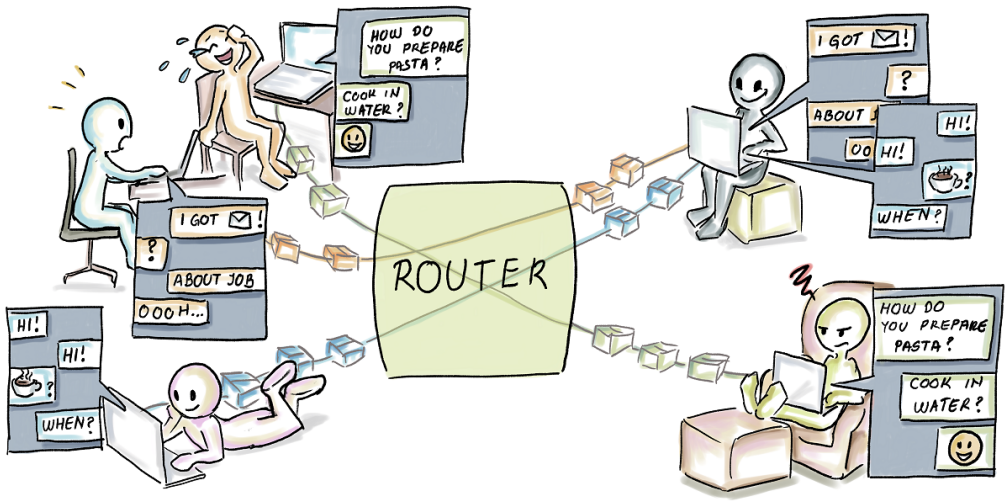

Figure 5.3: A healthy network flow. A fairly large number of packets but a small number of different flows.

[59] Estan, C., Varghese, G., & Fisk, M. (2006). Bitmap Algorithms for Counting Active Flows on High-Speed Links. IEEE/ACM Transactions on Networking, 925-937.

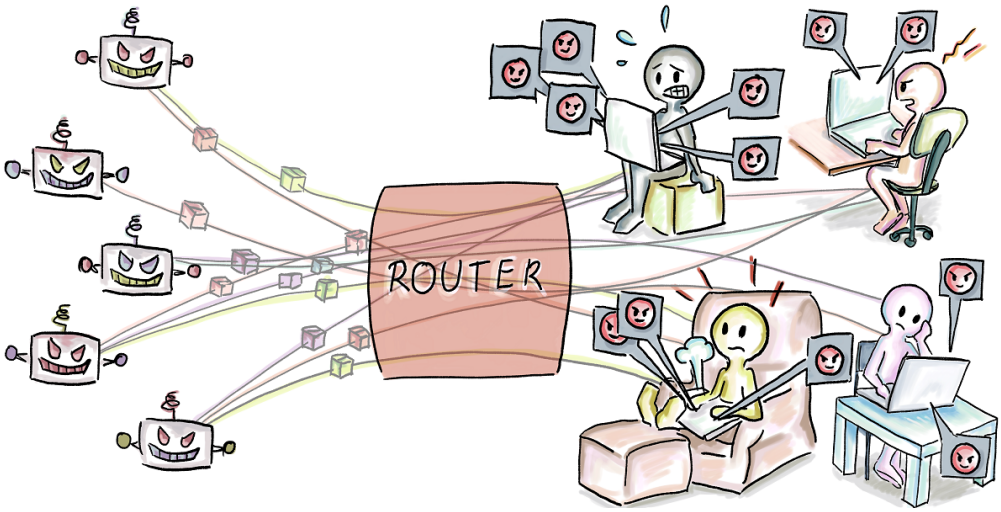

Figure 5.4: A suspicious flow — having many source-destination pairs, and one source opening a large number of different connections in a short amount of time.

Thus, embedding a HyperLogLog in the software that is wired into a router can be very beneficial, especially due to the need for fast computations times and a small memory footprint. Another good place to strategically place a HyperLogLog and other data structures/algorithms that help analyze the busy network traffic with small space and time requirements is entry point in an organization's network, as shown in Figure 5.5:

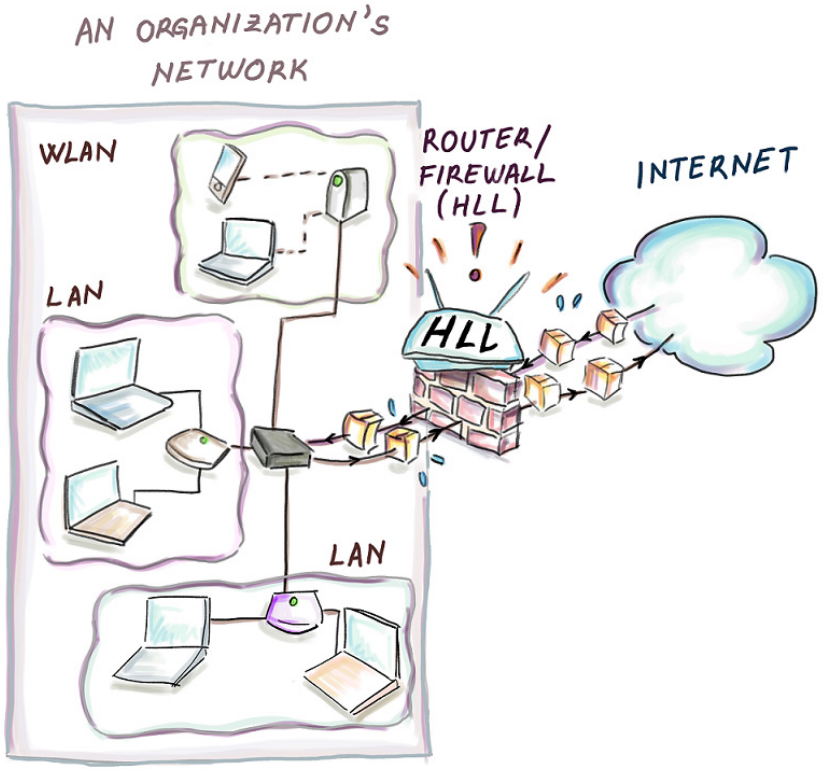

Figure 5.5: Placing a HyperLogLog at the entry point of the network within an organization can help us gather valuable statistics about the network traffic of that organization.

5.4 But how does it actually work? A mini experiment

In this section, we run simulations to gather some intuition on how various estimates --- probabilistic counting, LogLog and HyperLogLog compare with respect to bias and accuracy when run on a reasonably-sized dataset. We design an experiment to see how well the error bounds derived from probabilistic analysis correspond with numbers from the practical context. We are also interested in how much the normalization factors $\tilde{\alpha}_m$ (for LogLog) and α_m (for HyperLogLog) improve the accuracy, as well as the effect of the number of buckets in HyperLogLog on the accuracy and the width of the distribution.

Data for all plots in this section is derived from running the following experiment 1000 times: we generate $N = 2^{16} = 65{,}536$ 32-bit strings where each bit is chosen uniformly randomly. We are starting from uniform random strings that act like hashes (and we will refer to them in future text as hashes) because we are interested in producing 1000 hashsets of the same (or almost the same) cardinality. Considering that there can be 2^{32} hashes and our hashset size is 2^{16}, in most experiments, we will not encounter hash collisions, and the

total number of distinct hashes/items is in most experiments will be equal to the dataset size $N = k = 65{,}536$; there is an occasional hash collision, but the distinct count k never goes below 65,531, marking a negligible difference in cardinality between different experiments. We designed the experiment without duplicates because they do not influence the estimates of our methods, so this experiment could also serve to demonstrate how even much larger hashsets than 2^{16} but with 2^{16} distinct items behave.

In our first plot, shown in Figure 5.6, we compare the following methods:

- probabilistic counting
- stochastic averaging with arithmetic mean unnormalized ($m = 64$) and
- stochastic averaging with harmonic mean, unnormalized ($m = 64$).

The x axis shows the logarithm base 2 of cardinality; we indicate on the plot the position of true cardinality (at 16). The y axis shows the count of the number of experiments.

The plot shows probabilistic counting to have the largest deviation of the three methods, having some instances of the experiment by as much as 12 units of log2 cardinality apart, and 384 instances of the experiment (over a third) with log2 cardinality of 18 and over. Probabilistic counting is followed by stochastic averaging with arithmetic mean unnormalized, spreading about 1.5 unit of log2 cardinality, and the stochastic averaging with harmonic mean unnormalized is the narrowest of the three.

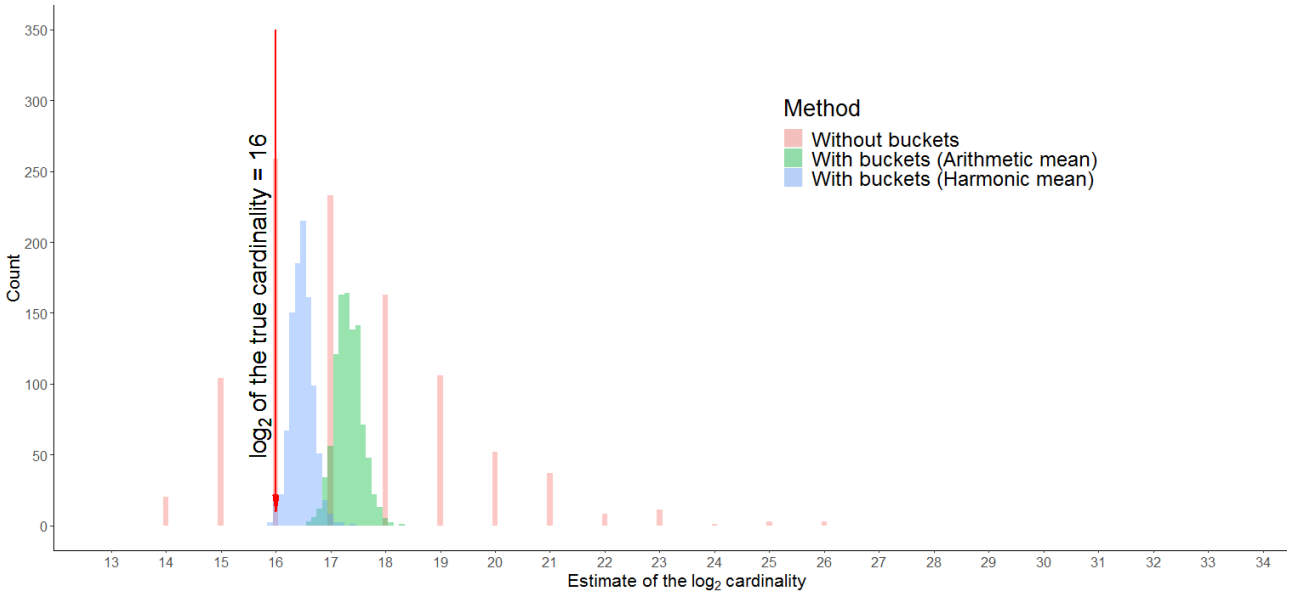

Figure 5.6: Plot shows the comparison of the probabilistic counting (Without buckets), stochastic averaging with arithmetic mean (unnormalized) (With buckets (Arithmetic mean)) and stochastic averaging with harmonic mean (unnormalized) (With buckets (Harmonic mean)). All raw estimates show consistent overestimate bias, however the least bias on average is shown by harmonic mean method, followed by the

arithmetic mean method and probabilistic counting. The largest deviation in the estimates (having different experiment vary in estimates by a factor of 2^{12} is exhibited by probabilistic counting, that only has estimates that are powers of 2, followed by the arithmetic mean method, and then followed by the harmonic mean method.

Closest to the true estimate on average is the *harmonic mean method*. The average log2 cardinalities in this experiment are: 17.31 (probabilistic counting), 17.32 (stochastic averaging with arithmetic mean unnormalized), and 16.47 (stochastic averaging with harmonic mean unnormalized).

After we normalize the arithmetic and harmonic mean estimates with respective constants $\tilde{\alpha}_{64} = 0.3907$ and $\alpha_{64} = 0.709$, average $\log 2$ cardinalities drop down to 15.97 (LogLog) and 15.93 (HyperLogLog) respectively, with an average bias from the true cardinality in both cases around 13%. It is pleasant that we obtain this result, considering that the estimated error in both cases is $O(1/\sqrt{m}) = O(1/8)$, approximately 12.5%.

5.4.1 The effect of the number of buckets (m)

Here we show the experiment with the same hashsets from above, but this time we measure the effect of using three different choices for number of buckets in HyperLogLog: $m = 16$, $m = 64$ and $m = 256$. As expected, the plot below shows that, the more buckets we have, the less variance we encounter in the obtained estimates:

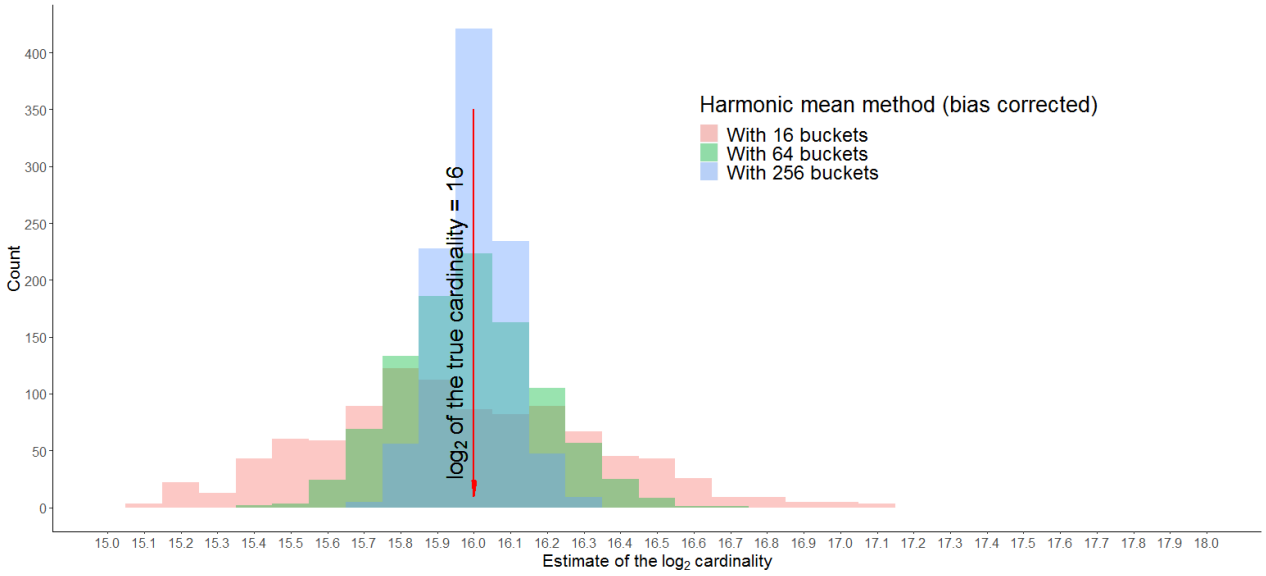

Figure 5.7: Effect of different values of m on the accuracy of $\log 2$ cardinality estimate in HyperLogLog. The larger the number of buckets, the smaller deviation from the true cardinality. In general, harmonic mean method once bias corrected, rarely over/underestimates by more than one unit of $\log 2$, in all three cases.

Because bias-corrected harmonic from HyperLogLog gets very close to the ground truth, here we also show the same graph, only plotting the bias from actual cardinality in each experiment (now x axis is the true cardinality not the logarithm):

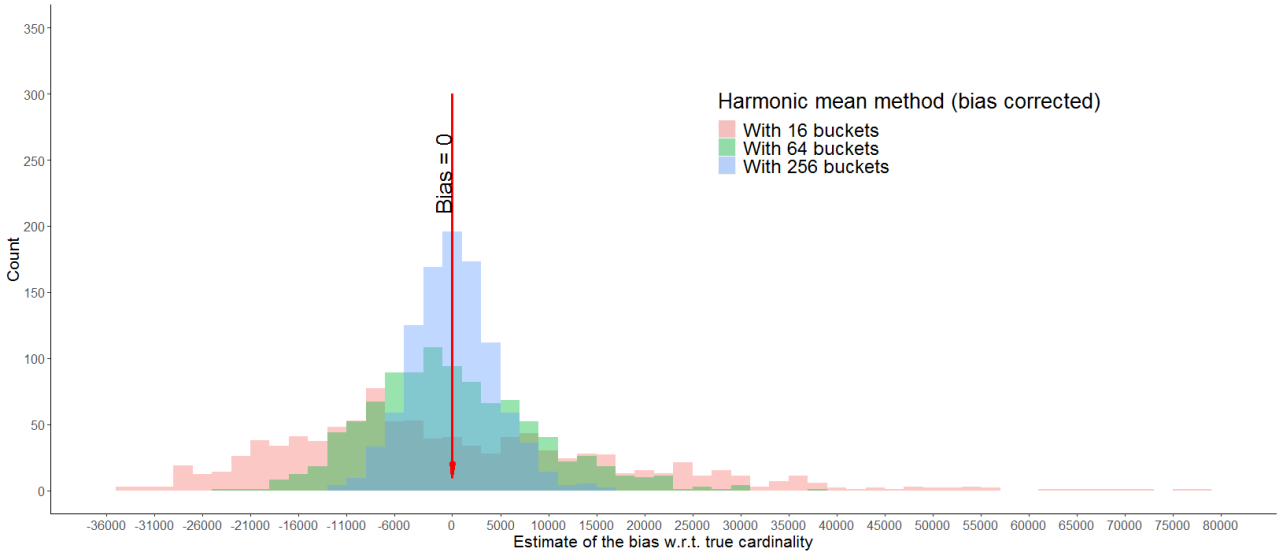

Figure 5.8: The effect of buckets on cardinality estimate in HyperLogLog. Larger m implies smaller bias and a more properly Gaussian-looking distribution.

As observed in the original paper, the distribution of cardinalities appears Gaussian, with shorter tails when m is larger. The distribution being roughly Gaussian can help us draw the following practical conclusion:

Given the standard error (or relative error) of HyperLogLog as $\sigma = 1.04/\sqrt{m}$***, then respectively about*** $65\%, 95\%$ ***and*** 99% ***of values (a value here being the cardinality estimate for one dataset) will fall within*** $\sigma, 2\sigma$ **and** 3σ **fraction of true cardinality away from the true cardinality.**

To verify that in our simulations, we took the case of $m = 256$ buckets, hence $\sigma = 1.04/\sqrt{256} = 0.065$. Therefore, 6.5%, 13% and 19.5% are respectively one, two and three standard errors away from the truth. It turns out that in our experiment, respectively 71%, 94.8% and 99.2% fall within the boundaries of the mentioned errors, indicating roughly Gaussian behavior (even a bit more tight). Thus, when we implement HyperLogLog, we can expect the estimates to behave in a fairly predictable manner and most often very close to the mean (true cardinality).

5.5 Use case: Aggregation using HyperLogLog

Let's revisit the example from introduction with tables of daily customer visits on a popular website. As we have seen, computing the distinct count on a column (e.g., finding a total number of users) in a large table is a challenge, but the real issue crops up when we need to aggregate those insights over a timespan of days, weeks, months, and so on. Individual daily are very costly to maintain for a long period of time, yet it is crucial for many businesses to be able to go back and pull out relevant statistics from an arbitrary moment in past. Unsplash, a photography website hosting a large number of images and receiving millions of visits per day, uses HyperLogLog to solve this exact problem.[60]

One issue with calculating the distinct count on one or more columns in a table is that, even if we were magically given the distinct counts, that in no way helps compute the aggregated count, as shown in the Figure 5.9:

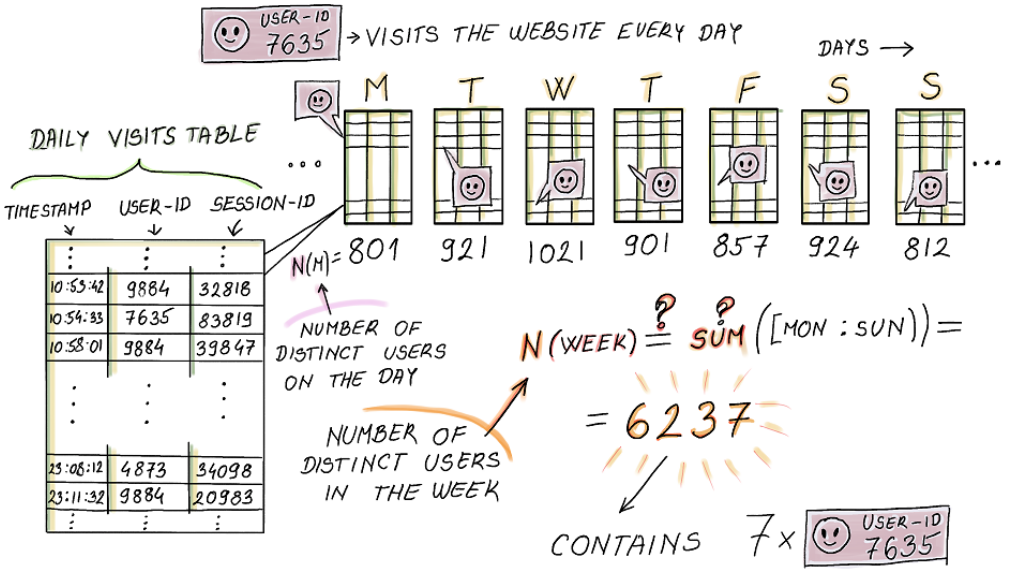

Figure 5.9: In a daily visit table, each row indicates one visit by a user and each table maintains a separate distinct-count variable that tracks the number of different users. Considering that some users return to the website repeatedly, we can't simply sum up the individual counts to obtain the week's distinct user count.

However, if instead of the distinct count, one could maintain one HyperLogLog per daily table, then aggregate the results over multiple days by performing a union operation between two (or more) HyperLogLogs of the same size and the same hash function.

The union operation of two HyperLogLogs HLL1[1..m] and HLL2[1..m] works by creating a new HyperLogLog HLL_UNION[1..m], and assigning max(HLL1[i], HLL2[i]) to HLL_UNION[i], for each i, $1 \leq i \leq m$. For example, the union of two HyperLogLogs whose

[60] https://medium.com/unsplash/hyperloglog-in-google-bigquery-7145821ac81b

bucket values are (1, 4, 2, 5) and (2, 2, 5, 3) would produce another HyperLogLog with bucket values (2, 4, 5, 5).

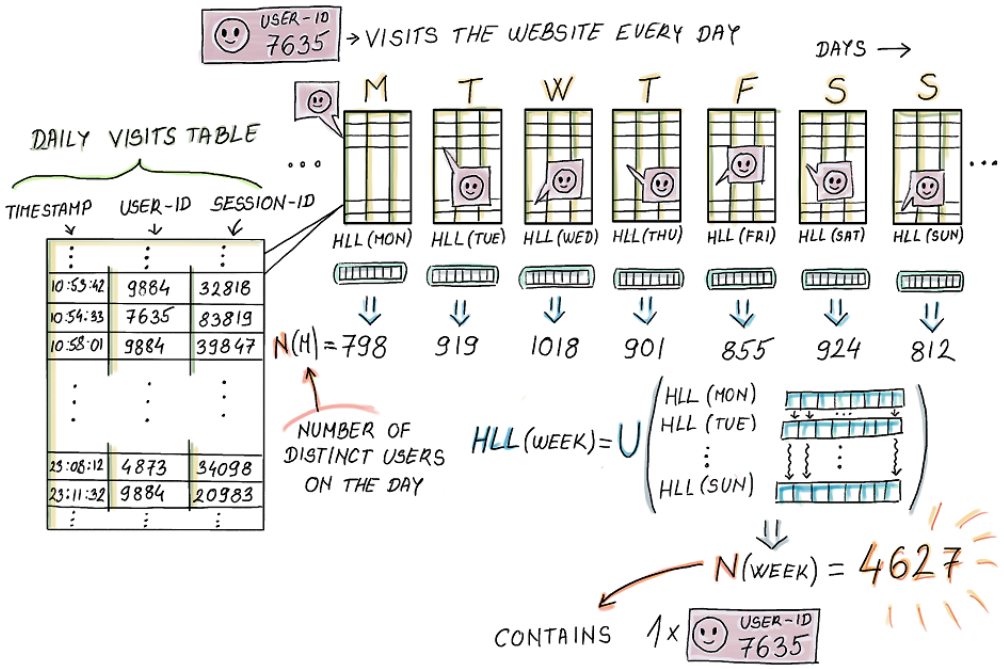

Figure 5.10: Maintaining one HyperLogLog per daily table helps us later aggregate the HyperLogLogs over multiple days to obtain an estimate for more tables. In fact, HyperLogLog can be easily encoded so that we can maintain a table of HyperLogLog schemas, later to be decoded.

What happens to the error when we aggregate a large number of HyperLogLogs? The relative error, being dependent on the number of buckets m, stays the same after aggregation, as the number of buckets remains unchanged. But as much as we might be tempted to think that in HyperLogLog the error does not depend on the size of the dataset, as it is often advertised, it is important to keep in mind (just like with count-min sketch) that the relative error *is* the percentage of true cardinality, which generally tends to increase with dataset size. So even though the error rate stays the same after union, the constant by which the error increases actually grows proportionally with the number of distinct elements.

HyperLogLog has a simple encoding which makes it conducive to storing as a record in a table of HyperLogLogs, whose space requirements are dramatically smaller than maintaining the equivalent daily tables. This enables us to aggregate HLLs over arbitrary time intervals or at certain specific dates, as shown in Figure 5.11:

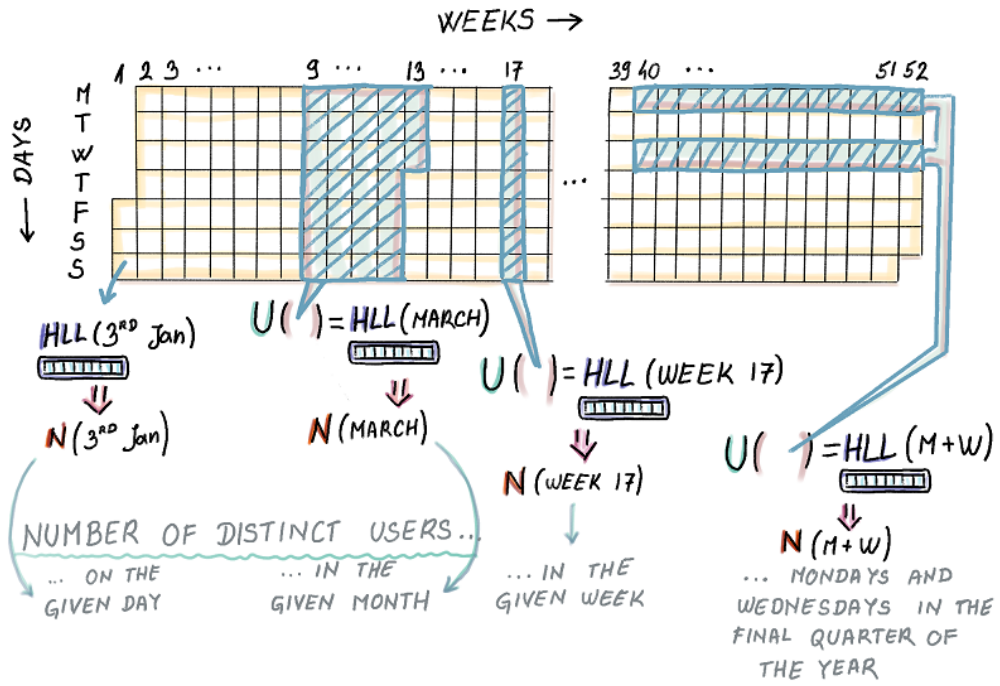

Figure 5.11: Once we have stored daily HLLs, we can perform union over the arbitrary choice of interest to obtain aggregate cardinality estimate for a given time period.

Moreover, the estimates can be performed on many levels, where we can aggregate hourly HLLs into daily HLLs, then use daily HLLs to compute weekly HLLs, etc (shown in Figure 5.12). In the world of traditional databases, doing a number of groupings on different levels usually means having to scan whole data once for each grouping we want to do. With HyperLogLog, we only need to scan all data only once to produce HyperLogLogs, and after that, we only read and combine HyperLogLogs.

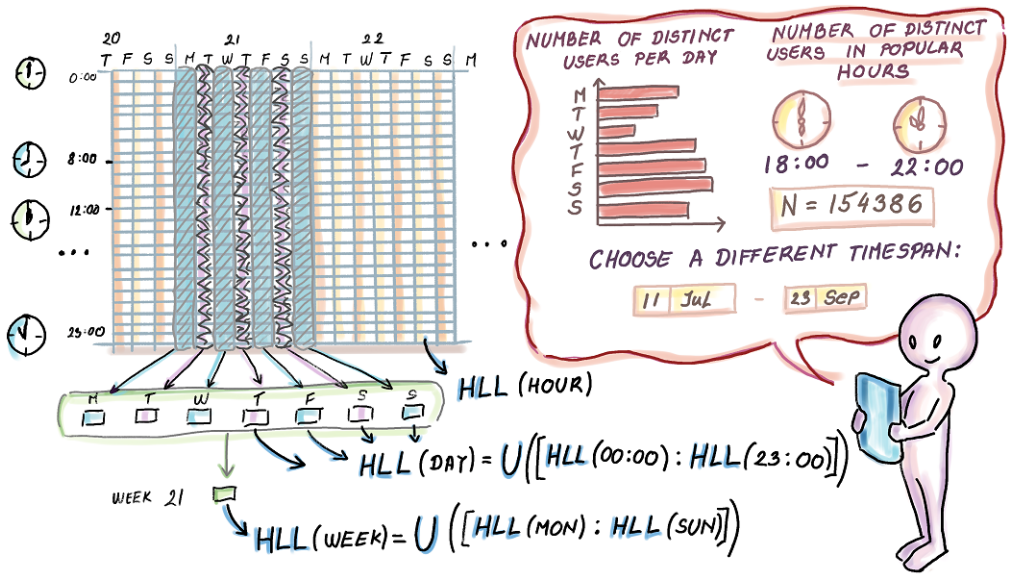

Figure 5.12: Aggregation happens over multiple levels in this case: hour, day, week, etc. Commonly in databases, when we group by different timespans, this requires a one scan of entire data for each level of aggregation.

5.6 Summary

- Cardinality estimation arises in many areas of software development, primarily databases, network traffic and e-commerce. Due to the volume of data, classical database functions for exact cardinality computation are being replaced with probabilistic methods that offer great space savings in exchange for a small error in accuracy.
- HyperLogLog is the algorithm/data structure that uses hashing and probabilistic properties of random bit strings to gauge the set cardinality. Its space consumption is $O(m \log_2 \log_2 k_{max})$ and its relative error $1.04/\sqrt{m}$.
- Many companies that run large systems have implemented HyperLogLog for their use, and improved and modernized various aspects of it (e.g., implementations by Google, Redis, Facebook and others.)
- The estimates provided by HyperLogLog have roughly Gaussian shape. In our simulations on a hashset of 2^{16}, we ascertained that HyperLogLog obeys the rules of Gaussian distribution, by letting approximately 70% of data fall within one, 95% within two, and 99% within three standard errors.
- The true power of HyperLogLog is visible when doing aggregations of a huge number of large individual tables that represent data over time. Instead of keeping the large tables, we can instead store a table of HyperLogLogs, and choose to aggregate and merge HyperLogLogs for the periods of interest (e.g., week, month, quarter, etc.)

6

Streaming Data: Bringing Everything Together

This chapter covers

- Learning about the streaming data pipeline model and its distributed framework
- Determining where streaming data applications and the data stream model meet
- Identifying where algorithms and data structures fit in data streams
- Recognizing the current duality of the data processing paradigm
- Setting up basic computing constraints and concepts inherent to data streams

Previous chapters introduced a number of algorithms / data structures for sketching (an important characteristic of) huge amounts of data residing in a database or, as you saw in the application of HLL in network traffic surveillance, arriving and expiring at a lightning rate. Within the pages to follow, we will round up these at the corner where they usually meet anyway.

The first part of the current chapter will serve the purpose of zooming out of the very granular algorithmic view on massive data algorithms and try to do some book-keeping and landscape inspection of the wider context where the algorithms covered so far, or in some sense their application areas "live". Conveniently for us, who at this point need to start dealing with data streams, one of their natural habitats are streaming-data pipeline applications and their wider architectural context. This wider context is extensively discussed in the Manning manuscript *Streaming Data* by Andrew G. Psaltis[1], hence we will use the model of the streaming data system / pipeline from that book to stage and depict how and where Bloom Filters, Count-Min Sketches and Hyper-Log-Logs can be found in that particular architectural landscape.

[1] Andrew G. Psaltis, Streaming Data. NY: Manning Publications Co., 2017.

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.

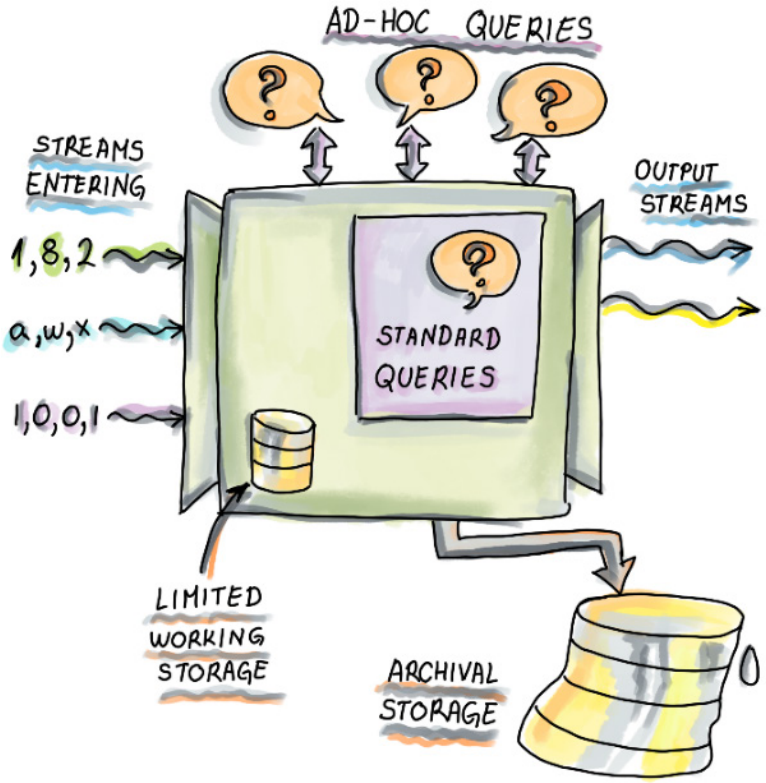

Figure 6.1: Streaming model. The streaming model differs from the traditional database management system in that data passes through the processor and a small amount of working storage, and it is either never stored, or it is stored into the archival storage that is usually too large and slow to be indexed 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.

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

providing perfect accuracy but on their own clock. The figure 5.1[2] is a rough depiction of the streaming model that algorithm researchers use.

We are about to elucidate how this high-level view on streaming data (figure 6.1) fits into a fully implemented and functional streaming data cloud application. Namely, the streaming data model illustrated in figure 6.1 embodies the maximum complexity that should be permitted to be left of a realistic problem, when one wants to develop a functional algorithm to solve it. Such simplification facilitates algorithm designers (very smart people, most often theoretical computer scientists, applied mathematicians etc.) not to dissipate their time and focus, while trying to flesh out and communicate their ideas and eventually solve the problem. On the other hand, figure 6.1 is also the minimum complexity algorithm designers would be wise to allow, if they want their algorithm to be recognized as relevant and applicable in the practical setting. This sort of balance between abstraction and reality is somewhat analogous to Occam's razor principle.

Therefore, the streaming data model specified above fully serves its purpose, when one is to design a streaming-data processing algorithm, or the actual "heart" of a high-throughput data-intensive system.

In the *Streaming Data* book, the author pedagogically plants this model in the analysis tier of the streaming-data pipeline model (see figure 6.2), because there, in the heart of this system, is where its designers imagined its use. This organic setting makes it easier for authors to teach it and for readers to internalize its main algorithmic ideas. For purposes of understanding an algorithm, such "zooming-in" is helpful, but it blurs our vision when we want to develop an intuition about when to use i.e. bloom filter as our solution. In this case, it helps to take a step back and observe the algorithm in a system, where it is used to solve several apparently different problems. This is the only way a novice in the area can recognize commonalities in different areas of application, with these commonalities being the key to successfully developing practice-relevant skills on the topic. We already saw illustrative use-cases in previous chapters for each algorithm / data structure, but with a streaming data pipeline (figure 6.1), we have a chance to see them in close juxtaposition used for different purposes.

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

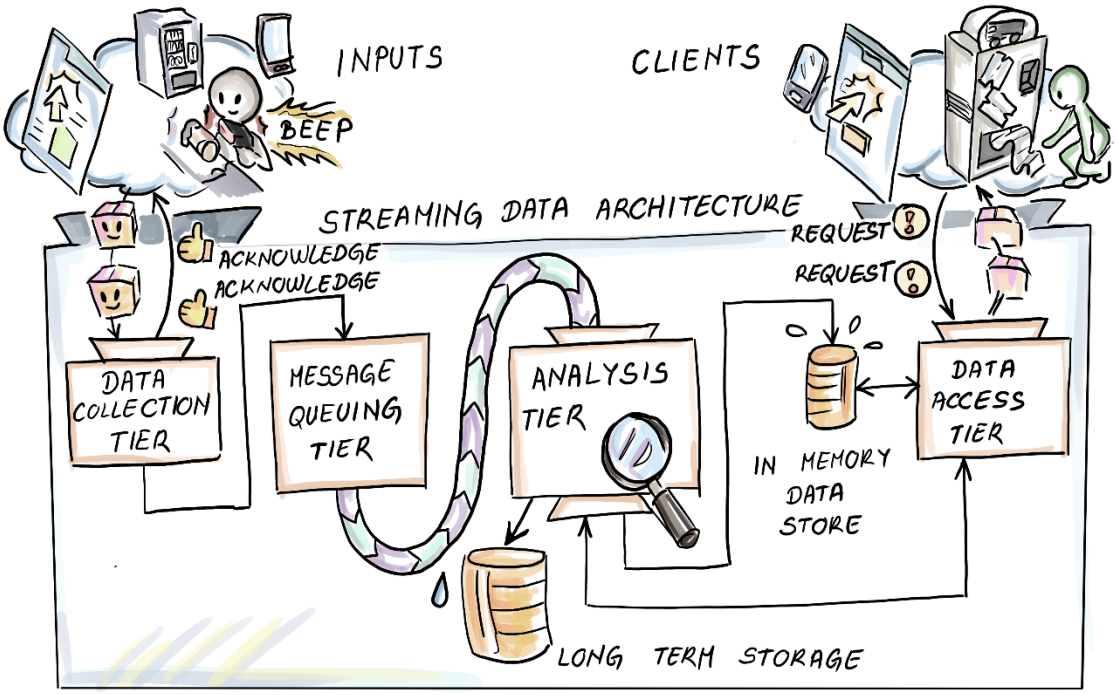

Figure 6.2: The general model of a streaming data pipeline is shown. Data producers are initiating connections by interacting with retailer's web site / mobile application / online shop / offline self-check-out service or their data is being pulled periodically.

We will use figure 6.2 to describe the common evolution of data through streaming data pipeline. Data is consolidated in a centralized data center that integrates data coming from different geographical areas. Here some transformation, data augmentation and pre-processing might happen. Data is then sent to message queuing servers (in-house or commodity hardware gathered around a cloud service) that check and, if possible, re-establish structural, temporal and perhaps causal consistency of the data. Through message queuing paradigm this layer establishes and maintains the balance between ingestion rate of data from the collectors and consumption rate from the side of the analysis tier. This may happen because analysis tier is more computation intensive, than producers passing data to collection tier, which can easily lead to data congestion, *load shedding* (deliberate dropping of unprocessed tuples) hence loss of data. Finally, data reaches the analysis tier where different synopses of the data (sub)-streams are calculated and kept, streams are sampled, continuous and ad-hoc queries are answered. Subsequently, after collapsing the query answers output streams from the analysis tier are forwarded to supply different data-consumers on the "edge" of the streaming data system, like data dashboards, real-time add bidding application or some automated industrial production control application.

Section 6.1 describes a meta-example of sorts, that should contribute to your ability to recognize which problems, in an inherently massive-data context such as (distributed) streaming-data pipeline, are well suited to solve by applying our previous acquaintances (Bloom Filter, Count-Min Sketch …) and our future ones that we will get to know in chapters to follow. We hope that this will help you develop a skill of honing in on parts of the system that represent solutions for a narrow, localized problem and a skill of zooming out to a birds-view on the data intensive distributed applications from their "source" to their "sink". For those of you still inexperienced in streaming data applications, this will be a chance to have a safe first look from the "belly of the beast".

In section 6.2 we will report on future trends and predictions, when it comes to the duality of paradigms: traditional, database batch processing vs. continuous data stream processing.

Section 6.3 introduces concepts native to data streams that drive algorithm design and define inherent constraints under which such algorithms are developed, in order to recognize those constraints in our own implementation or when making a decision about which one to use. To achieve this goal, our sincere recommendation is to read the book by Psaltis[3], which in combination with the one you are holding in front of you makes up a well-rounded and powerful streaming data toolbox.

6.1 Streaming Data System – a meta-example

Figure 6.2 shows the model of the streaming-data pipeline (adapted from Psaltis). Keep in mind that the depicted tiers are not so clearly discernable from one another in practice, as they are in figure 6.2. As we will see, these tiers often overlap, in that some parts of the system integrate tasks that cannot be clearly attributed to a single tier only.

Through a realistic example we will zoom in on the tiers shown there and identify where an application of an algorithm or a strategy (say hashing) for handling massive data is present in this wider, architectural streaming-data context. The first and most plausible place to look for them is the "Analysis tier" in figure 6.2. This is where all our previously introduced algorithms will find their immediate purpose and here is where the majority of all our use cases so far are anchored (restless sleeper, top-k trending queries on Google etc.). Figure 6.1 describes the components and make-up of the "Analysis tier" with Bloom filter, Count-Min Sketch and others offering themselves more naturally for showcasing in this tier.

But, if one looks a little closer at what happens in collection tier or message queing tier, we are going to recognize problems where algorithms and data sctructures for massive data crop up quite organically as possible solutions. This should not come as a surprise, since streaming data pipelines have vast numbers of data tuples fly by along their "whole length". The only difference, as you will see, is, which components of the data tuples that are being sent / emmited, queued, transported, received and analyzed are relevant for each of these operations.

We know that the most general model of sending data along some network entails at least two components, metadata and payload. We will see that, depending on where we are in the streaming data pipeline, the metadata and payload can change their conotation. This

[3] Andrew G. Psaltis, Streaming Data. NY: Manning Publications Co., 2017

means that along the data-pipeline, payload (defined as such w.r.t. bussiness problem that is to solve) sometimes becomes overhead of the data tuple, while metadata becomes relevant for the analysis. Now if this "switch" got you a little dizzy, it is high time to exemplify what we mean.

6.1.1 Bloom-join

We will use an example of a large retailer that sells its products online and in stores. Walmart or Wholefoods would fit the profile. The company might be interested to have a (close-to-) realtime analysis of the correlation between click pattern on their URLs and their sales transaction data, to potentially optimize their strategy for bidding in real-time ad campaigns[4]. These days it is still not uncommon to have these two types of data in two different database systems to make a so called hybrid warehouse. The sales data is more valuable for the company, hence it often resides in a parallel database on high-end servers called enterprise data warehouse (EDW), while for the click-stream data a comodity server network like Hadoop Distributed File System (HDFS), often might suffice.

For now we will assume that the click stream data tuples arrive and are stored in HDFS, and we want to join the click stream data and the data on sales made online. We will do this by using the *IP address* column as the join key (here, for the sake of clarity, we are abstracting away the time-proximity that the join has to take into account, so that only clicks close-in-time to a on online purchase, are matched with that purchase, see figure 6.3). We can assume that both relations are large, but that HDFS-side one is larger, which is a plausible assumption. Without getting into all intricacies that such distributed computation between these storage systems carries with itself, we concentrate on minimizing the size of the tables that we need to broadcast between these two systems in order to implement such join operation. This saves bandwidth and time, particularly when the local predicates and / or projections applied to the tables are not highly selective (we end up with tables that are not much smaller than all the data that the storage systems hold). The common strategy in such case is for each side to first make a bloom filter (BF) of the join key. Let us assume that the final join happens on the HDFS side, then a global BF_{EDW} on the EDW side calculated for the *IP address* column is sent to each HDFS query processor (HQP). Here it is used as a type of a predicate to identify the resulting smaller table that will actually participate in the final join. In case data needs to be shuffled among HQP processors, only data with a join key in BF^{EDW} needs to be moved (up to the false positive rate of the BF_{EDW}). Then the HDFS side makes its global BF^{HDFS} and sends it to the EDW side, which uses it to further reduce the number of rows that need to be sent. After all this is done, EDW side sends the resulting table after applying the predicates, projections and BF^{HDFS} on its original table. Through this two-way use of bloom filters, only those records that participate in the join will be sent over the network and only the neccesary shuffles of data between HQPs on the Hadoop side, need to be executed.

[4] (https://digitalmarketinginstitute.com/blog/the-beginners-guide-to-programmatic-advertising), last accessed on 10/20/2020

Exercise 1

Assume that a single IP-address is saved using 4 bytes. The join between EDW and HDFS systems happens at a regular rate that keeps the number of communicated IP-addresses from which purchases have been made constant (this means, that for business purposes we can assume both purchases and clicks are uniformly distributed over time). Hence, the join described, happens at regular intervals. Assume that the next join happens after 1 mil. distinct purchases were made. What is the size of the communicated data that we save by employing a bloom filter here with a false positive rate of 0.1 %, 1 % and 5 % respectively?

Now let's look at where, in the streaming data pipeline, all this just happened. Such *Bloom-join* can be anchored somewhere in the collection tier of our schema from figure 6.2. It is a type of a data augmentation / preprocessing step to generate data apt for answering the question of interest from the bussiness domain of the company. The resulting table created in the HDFS-side join with tuples being combinations of temporally proximate clicks and purchases from the same IP address (figure 6.3), can then serve as a type of a producer for a data stream processing framework (i.e. Apache Kafka). The data tuples (rows in the resulting table, figure 6.2) are then passed on to be queued, analyzed and used for perhaps (close-to-) realtime individualized ad campains.

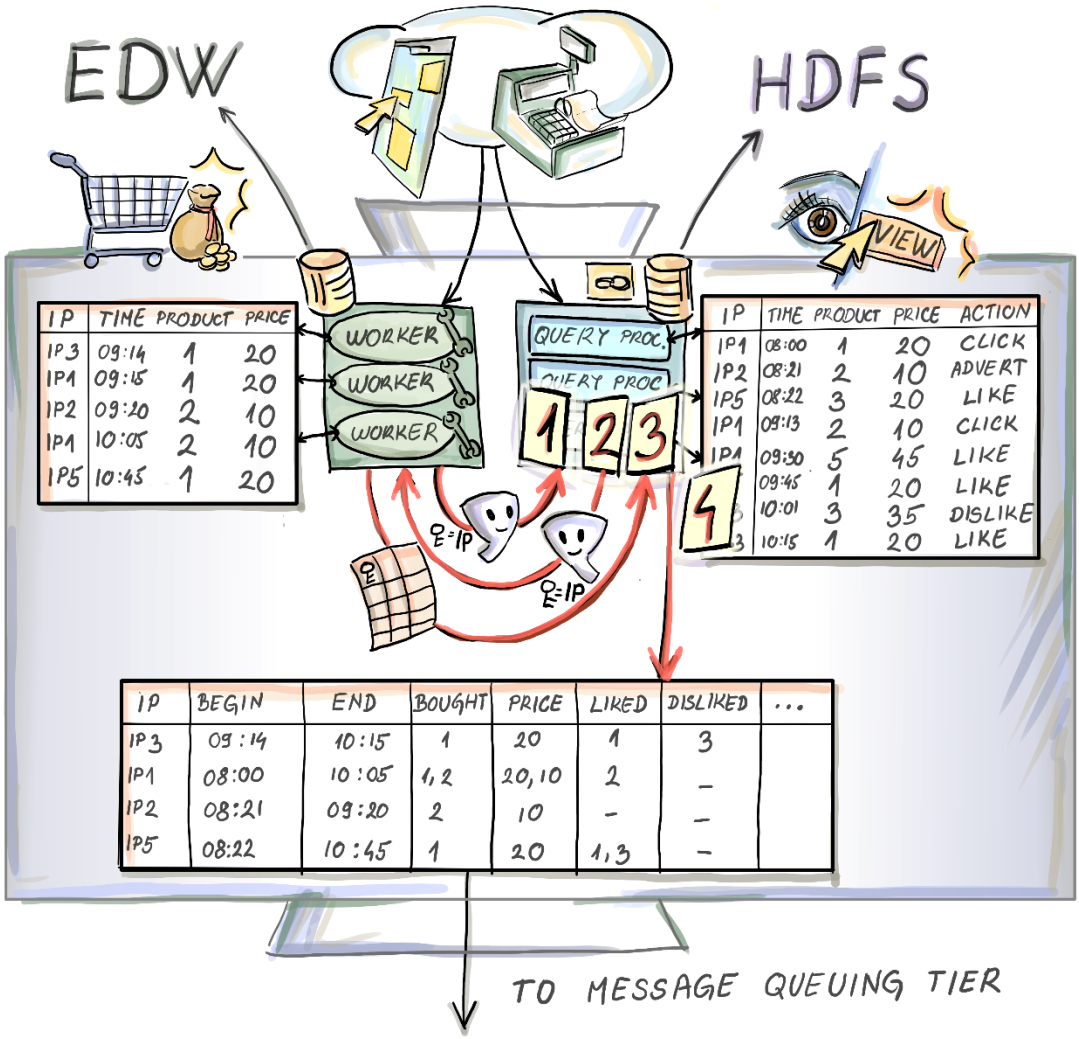

Figure 6.3: Figure shows the pre-join communication between Enterprise Data Warehouse (EDW) implemented with fast parallel database (on the left) and a Hadoop Distributed File System (HDFS) (on the right). Before the data for financial transactions is sent from EDW side, exchange of Bloom Filters that both storage systems make for the mutual join key (IP address) is made. Bloom filters are used by each side as a criterion to identify tuples that will participate in the final-join. HDFS can then shuffle only necessary data among its nodes and move the minimum of data to the node that will execute the join to come. EDW will identify which IP addresses didn't appear in HDFS and send only those that will participate in the final join. This way, purchase data is augmented with the click-stream component from that IP address and can further be used as a data source in a hybrid streaming data pipeline.

6.1.2 De-duplication

Due to the high ingestion frequency from data producers (see figure 6.2), and consequentially large flow of (perhaps preprocessed) data through the pipeline, each of the tiers shown in figure 6.2 is made up of a large number of nodes (machines) connected through a network. These computation nodes are implementing the task of their tier in parallel, as fast as possible. The message queueing tier is there to prevent congestion, loss of data (due to, say different rates at which data is produced and consumed), implement deduplication if necessary, etc. These safety mechanisms inherently entail some resolution steps between the nodes, that keep track of what data was passed through to the analysis tier and what should be next.

Nodes in this tier are commonly called *brokers* and besides keeping message queues consistent, they do other preprocessing steps too. Imagine that a user, while interacting with a retailers website, loses wirelles reception, or enters an elevator and hence does not receive acknowledgment from the server side of her like, submitted comment, ad-click, submitted payment etc. The user or the mobile app will try to send the same request to the server again, leading to duplicates being received. Not to speak about the issue that a user might have if he/she is charged twice for a one-time service, but corporate systems also don't like seeing duplicate payments either. Some systems, especially e-commerce ones, have a deduplication mechanisms to keep such redundancies out. The percentage of duplicates in realistic scenarios is not too large (maybe up to 1 %), nevertheless, in a system that logs billions of events these can lead to inefficiencies reflected as a significant loss of profit.

Now, we had a de-duplication problem solved once already in the previous chapter. Can you remember the example with large file storage and back-up services? This reality where a small portion of messages (our data tuples from before describing click-patterns with message-id as a key) is duplicated, offers itself nicely to another bloom filter application. One solution allocates "intercept" nodes into the streaming application, built perhaps on Apache Kafka Streams (we won't get into specific architectures here, as this would go beyond the scope of the book, but "Streaming Data" book does). These worker nodes are connected to highspeed databases and aside from permanently saving (all or just a "window" of) messages to facilitate a possible roll-back in case data is lost, they keep a bloom filter of all message ID's of the messages they save. Each message that arrives is checked against the filter, and if it's reported present (subject to BF false positive rate), worker nodes discard them. Deduplicated message streams proceed to queue in possibly Kafka output topics[5] where they can now be forwarded by a load balancing node to several brokers preceding the analysis tier. (figure 6.4)

[5] For more details on this de-duplication architecture see https://segment.com/blog/exactly-once-delivery, last accessed 10/22/2020

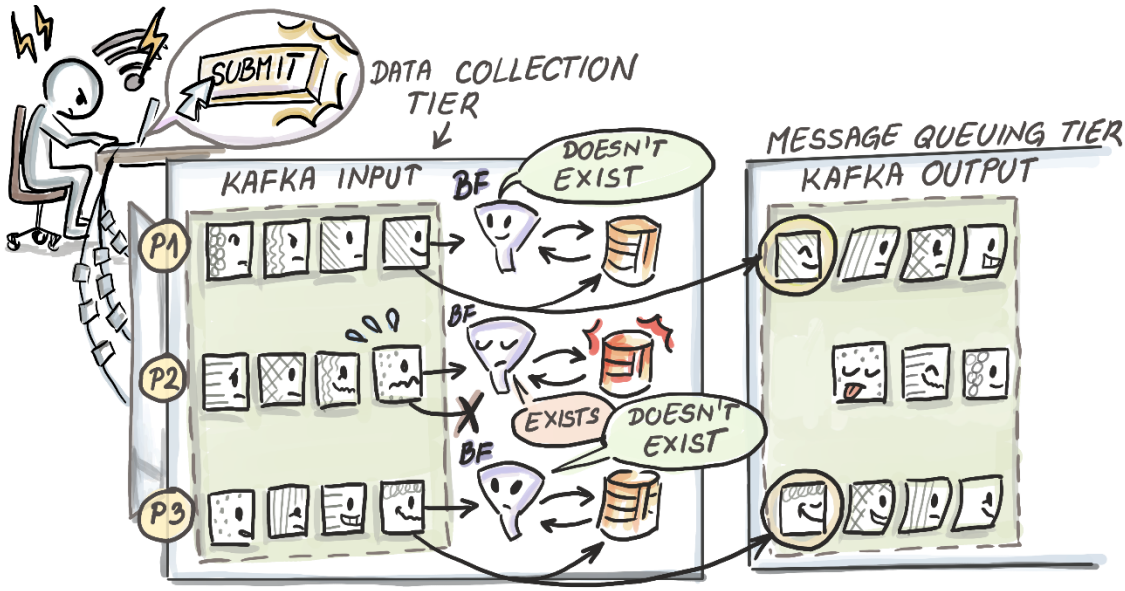

Figure 6.4: Intermediate nodes connected to fast databases implementing de-duplication to remove the repeated messaging instances in a streaming data pipeline. Each node keeps a Bloom Filter of messages it saved and at arrival of the next message ID checks the message ID hash against its Bloom Filter. In case the Bloom Filter reports that the message already exists, this message with its data is discarded, otherwise the message is saved and propagated on to the message queuing tier.

6.1.3 Load balancing and tracking the network traffic

As in any distributed computing system, in streaming data applications too, load balancing among the brokers is of paramount importance. Unbalanced load issued to brokers can cause one of them to receive disproportionately more connections, requests etc. Considering that the service is as fast as the slowest broker this can cause high end-to-end latencies and make real-time application not as real-time as to be useful. Near-real-time detecting of overused resources in a network is a clasical network traffic / distributed queueing problem and it boils down to detecting outliers / anomalies instantly. Such outiers come in the shape of overloading packet flow patterns and are typical for denial-of-service attacks on servers. Modern defense strategies rely on statistical methods to detect them in real time.

One such class of algorithmic solutions for this issue in network traffic domain relies on monitoring package headers in the network (most basic data tuples facilitting this taks could be FL:=[source IP, source port, destination IP, destination port, protocol]). In load balancing problem domain, monitoring is exceuted for the stream of requests to a load balancer node in charge of the network of brokers operating in the message queuing tier of our data pipeline. This is done to potentially identify small number of flows that constitute most of the

network traffic, the so called *heavy-hitters*. In practice, we want to detect a number of them whose rate (nr of packets / requests (bytes) in a unit of time) is above some threshold.

We already had a chance to see in chapter 5 a Hyper-log-log based solution for a worm detection problem in a generic network. Another solution might be to employ a count-min sketch that counts the aggregate size of the flow by adding the size of each packet sent through that flow (flow here is a pair of source/destination determinants). Keys hashed into the sketch are packet headers and the counter is incremented by the size of the packet or, in the case of a broker in the data streaming application, the number of requests to the same broker from momentarily increasing number of clients (called *sudden-bursts* in this context). The traffic measuring or load balancing application would then first estimate min counts. These would be periodically divided by the length of the measurement period to calculate flow rates / load rates. This allows us to identify rates above certain threshold and apply control strategy for the identified culprits. The flows under the threshold are definitely not malicious, while out of those identified, some may be false positives due to the overestimate inherent to count min sketch. We could then check quickly the exact size/rate of the small number of flows breaching the threshold and remove false positives leaving only the true culprits in the set.

Exercise 2

Consider using count-min sketch in the application of package flow threshold breach. In particular, recall two error parameters of count-min sketch, and the manner in which the overestimate in count-min sketch is expressed as a percentage of the total quantity (total flow, in this example, not individual flows.) Given a total flow of N=10 billion, and a threshold T of 1000, how would you design and use count-min sketch for two different scenarios:

a) Count-min sketch reports only the true breachers in 99% of the cases, disregarding that there might be more breachers that it is not reporting, and

b) reports potential breachers such that their true flow is at least 70% of the reported flow, also in 99% of the cases?

The network traffic monitoring application fits organically in the cloud service on which your streaming data application is running and is implemented as part of the cloud service, while the load balancing use-case pertains to the streaming application itself (See figure 6.5).

Figure 6.5: Figure shows a cloud computing architecture servicing different client data pipelines. Network traffic monitoring application installed to monitor communication of data producers (or any communication attempts originating outside of the cloud) is implemented with the help of a Count-Min Sketch. CMS identifies network flows that exhibit flow rates above a certain pre-determined threshold and acts accordingly. Similar problem of load balancing between the brokers of the message queuing tier is solved analogously by applying another CMS at the load balancer node in message queuing tier. Notice that both of these CMS are operating on the package / message headers and that the payload of the packets / messages, namely data on clicks issued by the user are not yet analyzed until they reach the analysis tier. There with the business problem at hand, other Bloom filters, Hyper-log-logs, sampling procedures or other synopsis are calculated.

The purpose of this short and surface-scratching excursion into realistic streaming data architecture was to give you some insight into omnipresence of the algorithms and data structures, covered so far, in the state-of-the-art streaming data applications. Aside from that, we hoped to help you to a glance at all accompanying problems that need to be resolved in such an inherently distributed computing landscape. We hope to have weaved a *rug to tie the room together* for you so far.

Our second goal was to relate the level of abstraction needed to develop streaming data algorithms (figure 6.1) and the level necessary for building a realistic streaming data application / pipeline (Figure 6.2). We could see that the former crops up in several places of the latter and streaming data glues them naturally together, each becoming visible at different "image resolutions" of the system shown in figure 6.2. You noticed that our meta-example had HDFS as a part of a collection tier of some streaming data pipeline. This might sound to some of you as mixing two paradigms that perhaps shouldn't be as interdependent as they appear here. In the next section we will talk about some trends in the data processing world that houses both the batch-data procesing paradigm as well as streaming data approach.

6.2 The Future is coming: in discrete batches or as a continuous stream ?

If you still haven't, chances are you soon will, come across a discussion on your favorite data base / data analytics / Big Data community web site that debates the question of batch vs. stream processing of data. In our opinion the general resolution strategy that you can take off the shelf and apply to your use-case to make a decision, is so far not existent. The answer almost always depends on knowing external maket forces that steer one's bussines model. In case you are planning to switch from your current batched data processing software to stream processing framework, your deliberations should concentrate on if and how (near-)real-time knowledge can offer you an edge over the competition. In other words, you should know when, why and how will you use real-time analytics after you implement such a paradigm changing migration.

Recent study titled *"Digital Transformation: Are We Finally Past the Unmet Expectations?"* and comminisoned by Couchbase[6] showed that out of 450 surveyed technology executives from the U.S., UK, France and Germany, 52% believe that the pressure for digital transformation could lead to wasting a big chunk of their planned investment for this purpose on ill-planned, rushed projects. The chances of this happening when switching to the technology that is still in its infancy like streaming data, are even higher. Taking into consideration the multi-disciplinary nature of the skill-set necessary to deploy a functional, robust and gainfully employed streaming data architecture, this endevor sounds even riskier.

Another sobering finding from the study reveals that 35 % of the participants believe, that the main engine in enterprise digital transformation are advances made by competitors. This means that the arguments for or against real-time analytics depend more on dynamics between bussinesses and less on proactive, original ideas that should propel one's own bussines forward. Reducing this to the lemonade-stand-level argument, would equate the situation where you, while you sell lemonade on your neigborhood corner, notice that the neighbor, with the same stand on the other side of the street, introduced one of those retractable belt barriers to help form a swirling queue, like those set-up in front of the airport security check. Although you know both of you see around 10 customers per afternoon, you get one too, in case someone decides to throw a block party and all your customers who can't fit on the sidewalk, migrate to your neighbor's finely arranged stream of customers harborred in the safety of the sidewalk next to its stand.

Judging by the voices of authorities in both data-processing paradigms, the consensus is that the change will come gradually and the necessity for batch computation will probably stay around in one form or another for a long time. What we can currently observe are systems that integrate some streaming data solutions into their legacy batch systems. Our streaming application with the Bloom join step in the data collection tier, that then proceeds to be a data source in the data pipepline, would fit this profile. Such trends are in accord with the words of Doug Cutting, creator of Hadoop who expressed his skepsis for the prospect of streaming data completely replacing batch-data systems: "I don't think there will be any

[6] "Digital Transformation: Are We Finally Past the Unmet Expectations", https://resources.couchbase.com/cio_survey2019/2019-cio-survey-digital-transformation

giant shift toward streaming. Rather, streaming now joins the suite of processing options that folks have at their disposal."

On the other hand, if we look at the way data is produced these days, we can all agree that each user, application, website or an enterprise sub-system generates data more or less as a continuous stream, while consumption of a possibly transformed stream happens continuously as well, by the same culprits. The question might then be, why make everything in the middle anything else but a stream as well.

As you probably noticed, the answer to the question, whether the future is "coming" in the form of a stream or will it be showing up in batch steps is multifaceted and subject to many ifs and buts. What all recent studies on trends in ICT can agree on though, is that the biggest hurdle for wide adoption of streaming data technology will be the lag in know-how, in particular, developers attaining knowledge to write robust and performant applications. This includes ability and experience to recognize when, where and how to employ and parametrize algorithms and data-structures that operate in small time and small space. This, I would feel, speaks in favor of the professional future of anybody who decides to pick up this book.

6.3 Practical constraints and concepts in data streams

The part that follows introduces some computing and data partitioning concepts which we will need in order to differentiate constraints under which streaming data algorithms need to be designed.

6.3.1 Time

Designing a streaming data application is a task that takes our conception of time from a phylosophical passtime, to a very practical time-keeping excercise. First question that came to my mind and, judging by some posts on data analystics forums, I am not alone in this dillema, is, can real-time analytics ever exist? Any semi-respectable streaming data reference will tell you that data in a data stream is continuosly received (from possibly numerous producers) with such pace that both saving it and making more than *one-pass* over each data tuple is infeasible. In some applied domains like analysis of financial data streams to make trading decisions, having this unrealistic option to save and query the whole history is deemed useless, since decisions will depend only on the data from the most recent week, perhaps even from the last minute. Hence, it is reasonable that typical requirements for streaming data algorithms are for them to operate in *one-pass* in *small time* and in *small space.*

Now let's revisit our question of the existence of real-time analytics. Even if our algorithms are built under these requirements, computation (not to mention security, communication, scheduling, load balancing all part of a typical cloud based streaming data aplication) costs time. Strictly speaking, the only data that *really* is real-time comes as sensory stimulation from the events we are immediate witnesses of. If this sounds like hair splitting to you, it is hard form me to come up with a convincing argument of the contrary, but please, bear with me. If you are like myself, and are just irked by this subtlety, I have good news, we will resolve this. Let us agree that *real-time* and notion of (near) real-time

analytics is decided by the state-of-the art solution for a particular streaming data (bussines) problem that produces results and helps in decision making, with latency that does not leave the bussines lag its competitors. In other words, users / clients have a final saying in what is (near) real-time for them, even when they over- or underestimate their needs.

If we think about it, even when witnessing live events in our lives, we are all experiencing identical latency when it comes to our sensory and congnitive appreciation of the events happening in front of us. This is why we agree among us so easily about the concept of real-time, we are all equally "late". Now that we settled this dilemma, of perhaps disputedly pressing, critical importance, we can continue with what we mean with *small time* and *small space*.

6.3.2 Small time and small space

For our considerations small space will be defined with respect to the available working memory depicted in figure 6.1 as *limited working storage*. In it we have to keep any data the stream-query processing engine needs to answer the ad-hoc or continuous queries in time. Here is where all the different data synopses: bloom filters, hyper log-logs, results from sampling algorithms (buffer) on data stream, histograms of the stream etc. need to fit.

Small time refers to processing time of the algorithm per each new arrival, as well as time needed to issue an aswer to a particular query (*query processing time*). Small time usually means sublinear, typically poly-logarithmic in N, where N is the length of the sub-stream that we can fit in the limited working memory.

6.3.3 Concept shifts and concept drifts

As much as data stream is continuous in its time component, the data generating mechanism can and will exhibit discontinuities.

Let's take a real-time streaming data application at Facebook that has a task of warning users of immediate local threat due to armed conflict, natural disaster or similar immanent danger that affects larger geographical area. Assume further that the application keeps counts of word occurences in sub-streams of user announcements on the platform. Sub-streams may be defined using some geographical criteria that makes warnings about such events relevant for people in the affected area. Any solution would need to implement the logic for calculating rates (count divided by the measurement time) of occurrence of some reserved words. An immanent local threat to human lives in the area would translate in the sudden increase in rates of word occurences related to such a disaster. Streaming algorithm should be able to detect such abrupt changes in the data stream, in literature known as *concept shifts*.

Behavior of the data stream similar to concept shift that is exhibited over a longer period of time and is characterized less by abrupt and more by gradual changes is called, *concept drift*. Detecting these is a less trivial problem compared to *concept shifts* and it has been a long standing research topic. For a good review of available methods see a review article by Sebastiao and Gamma[7].

[7] Raquel Sebastiao and Joao Gama, "A Study on Change Detection Methods," in 14th Portugiese Conference on Artificial Inteligence, EPIA 2009, 2009.

Both concepts are intimately related to the notion of windowed data stream, one of the mechanisms for accounting for recency in a data stream.

6.3.4 Sliding window model

Theoretically, a data stream is infinite. The stream processing is asummed to begin at some well defined time t_0 and that at any time t the queries are answered while *taking into consideration* all observed tuples seen between $t_{0 \text{ and } t.}$ This model of a data stream is reffered to as *landmark stream.*

Hopefully by now you may feel that this is just impossible, since we meanwhile know, that we cannot keep the stream in working memory and cannot make multiple passes on its data. At least not in time that would deem the answer relevant for practice. Luckily, the phrase *taking into consideration* means that the synopses that we make of the "galloping" data or samples that we may take from the stream data while it flies by, persist to be a function of *all* tuples seen so far. Hence, even the older data tuples that appeared long ago and, due to a our limited working memory, are long ago discarded or retired in *archival storage* (figure 6.1) contribute with the same weight the new ones.

Remember concept shifts and drifts? For some applications, like financial data streams, having old data (definition of old being bussines model-specific) govern current answers to queries is useless at best, and liability in avergae case. In combination with concept shifts and drifts, queries in landmark streams are prone to inertia and can be too slow in reacting to changes in concept. For this purpose different time-decay mechanisms have been introduced that relate age of the data tuple and the weight with which it influences the answers to queries.

Most prominent of them is *sliding window model* that considers only a certain number (a window) of most recently arrived data tuples. Data tuples outside of the window are automatically removed from the analysis or given the weight zero. Beware that they can still theoretically be in the limited working memory, if the sliding window is designed smaller than what we can fit in the space available to us for *one-pass* computing.

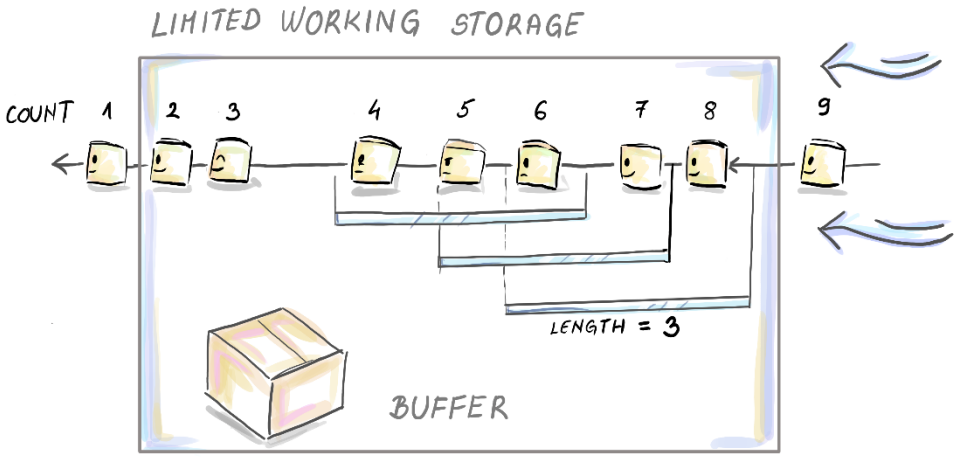

Figure 6.6: Figure shows last three sliding movements by the count-based window. Notice that window length is not necessarily all that we can fit into our working storage, but that is definitely the maximum in history that we can cover. Hence in applications where we want "as much history as possible" to influence our data stream analysis one would extend the length of the window to "everything we can fit". Keep in mind that aside from a sub-stream that we need to operate in one-pass mode we need to preserve space for our synopses and computation needed to build and update them. This is indicated by the buffer space show.

Sliding movement of the window can be either *time-based* or *count-based*. In time-based windows any data tuples that arrived in the last W time units are in the window, while for count-based windows sliding movement is governed by maintaining a constant number of W items in the window (it will be clear what we mean by W for each future mention). Figures 6.6 and 6.7 show both of the models on a generic data stream example for 3 most recent sliding window movements.

Our list of constraints related to data streams is by all means not exhaustive, but with the ones explained above, we can fare well through next couple of chapters without having to leave loose ends for any of the algorithms we will learn.

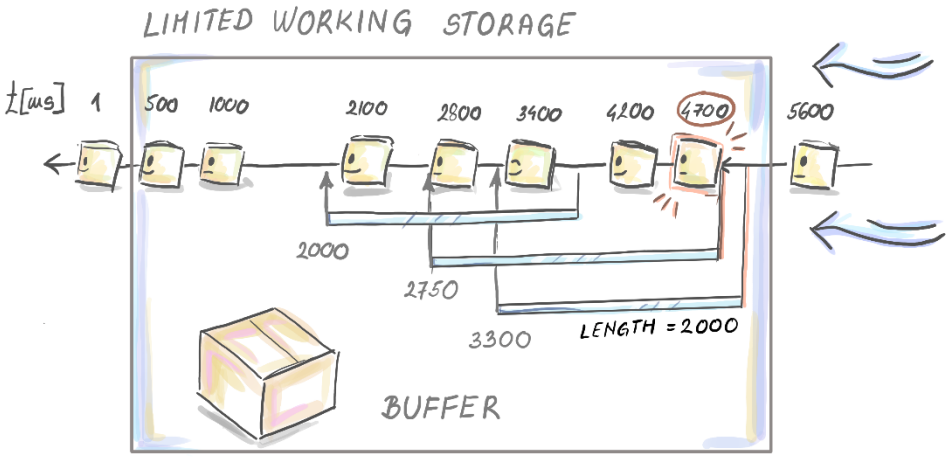

Figure 6.7: Here we can see the time-based sliding window of length W=2000 ms and its three last sliding (meaningful w.r.t. to the arrival times of the data tuples) movements. We started the data stream at time 0 ms and at time 1 ms the first data tuple arrived. Then two more arrived at 500ms and 1000 ms. We are currently observing the stream after 5500ms. The three last movements of the window are indicated, that changed the content of the window: from 1400 to 1401 ms (data tuple arrived at 3400 ms enters the window), from 2700 ms to 2701 (notice that this movement resulted in 4 data tuples in the window) and from 2800 ms to 2801 ms (data tuple at 2800 ms is discarded).

6.4 Summary

- Streaming data pipeline is a natural environment for showcasing the algorithms and data structures we learned so far and those we have yet to put under our belt.
- The combination of distributed computing and imperative of real-time delivery of results in streaming data aplications create numerous opportunities for shortening the end-to-end latencies by smart use of hashing, bloom filters, count-min sketches and hyper log-logs.
- Tasks like joining large tables saved across a heterogeneous storage systems, de-duplication in the stream, monitoring network traffic and load balance are all real examples of such opportunities.
- We need to be able to recognize and know how to use these chances where thinking about payload and overhead of a packet is important in such process of tuning of your data-pipeline
- Future of data processing architecture is an elegant dance between two paradigms: batch-processing and data streams. The transition from former to latter might happen, though it currently looks more as if hybrid systems might be first to take the stage for some time.
- Real-time analytics is possible if stakeholders can agree on level of tolerance for latency in such systems. Data generating mechanism is prone to periodic or incidental

changes and our streaming data algorithms should be able to accommodate and detect those in time. There are data stream models like *count-based* or *time-based sliding windows* to allow for recency adustments to detect such phenomena known as *concept shifts* and *concept drifts*.