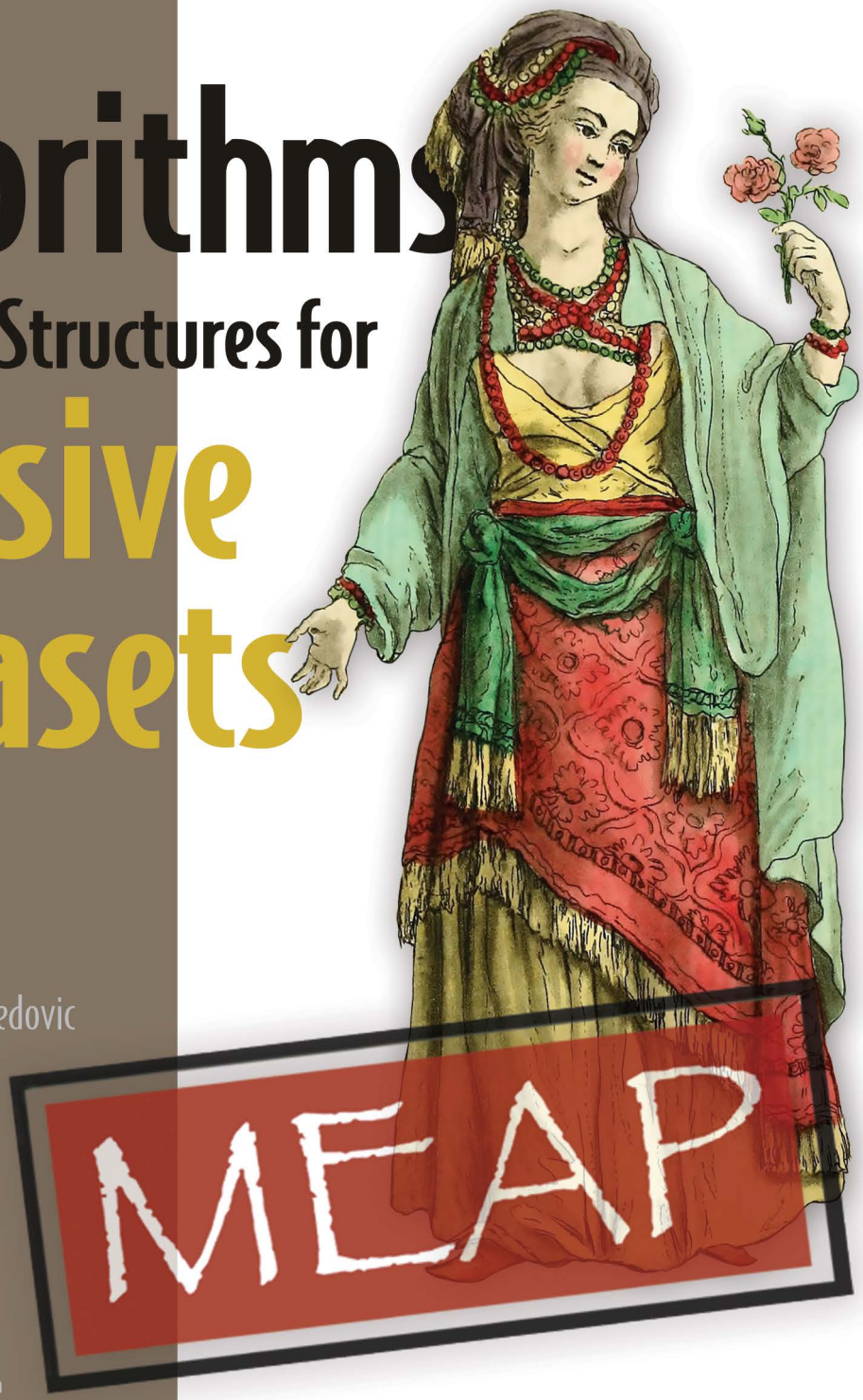
# Algorithms and Data Structures for Massive Datasets

Dzejla Medjedovic
Emin Tahirovic
Illustrated by Ines Dedovic




MANNING

**MEAP Edition**
**Manning Early Access Program**
# Algorithms and Data Structures for Massive Datasets
**Version 8**



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 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 have been developed with an implicit assumption that all data fits in the main memory, or *random-access memory* (RAM) of a computer. So, once you load all of 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 good old 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 what's 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. Yet, 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. In the following decades, data has grown tremendously, but more importantly, it has grown at a much faster rate than the average size of RAM memory.

The main consequence 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 once it's available. This fact is central to designing algorithms for large datasets, and it is from there that ideas of succinct data structures and external-memory oriented algorithms stem from. In Section 1.4, we will delve into more details as to why data access in a computer is much slower than the computation.

The picture only gets more complex as we zoom out of the view of a single computer. Most applications today are distributed and complex data pipelines, 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 formats have become

[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.

diverse, multidimensional and dynamic. The applications, in order to be 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 scraping, and ascertain the total number of distinct comments in the dataset.

### 1.1.1 An example: how to solve it

(…rolling up our sleeves) A common way to store unique elements in some data structure is to create a key-value dictionary where each distinct element's unique ID is mapped to its frequency. There are many libraries that implement key-value dictionaries, such as `map` in C++, `HashMap` in Java, `dict` in Python, etc. Key-value dictionaries are commonly implemented either as a balanced binary tree (e.g., a red-black tree in C++'s `map`), or alternatively, as hash tables (e.g., Python's `dict`.)

Efficiency note on red-black-tree vs. hash-table implementations: the tree dictionary implementations, apart from lookup/insert/delete that run in fast logarithmic time, offer equally fast predecessor/successor operations, that is, the ability to explore data back and forth efficiently using lexicographical ordering. Most hash table implementations lack the ability to efficiently traverse the items in the lexicographical order, however the hash table

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

implementations offer blazing fast constant-time performance on most common operations of lookup/insert/delete.

For simplicity of our example, let's assume we are working with Python's `dict`, a hash table. Using `comment-id` as the key, and the number of occurrences of that `comment-id` as the value will help us effectively eliminate duplicates (see the `(comment-id -> frequency)` dictionary from Figure 1.1 on the left).

However, we might need up to 24GB 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`). Depending on the method used to implement the underlying hash table, the data structure will 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 constructing a separate dictionary for each topic (e.g., sports, politics, science, etc), as shown in Figure 1.1 on the right. The role of the `(article-id -> keyword_frequency)` dictionaries here is to count the number of occurrences of topic-related keywords in all the comments --- 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.

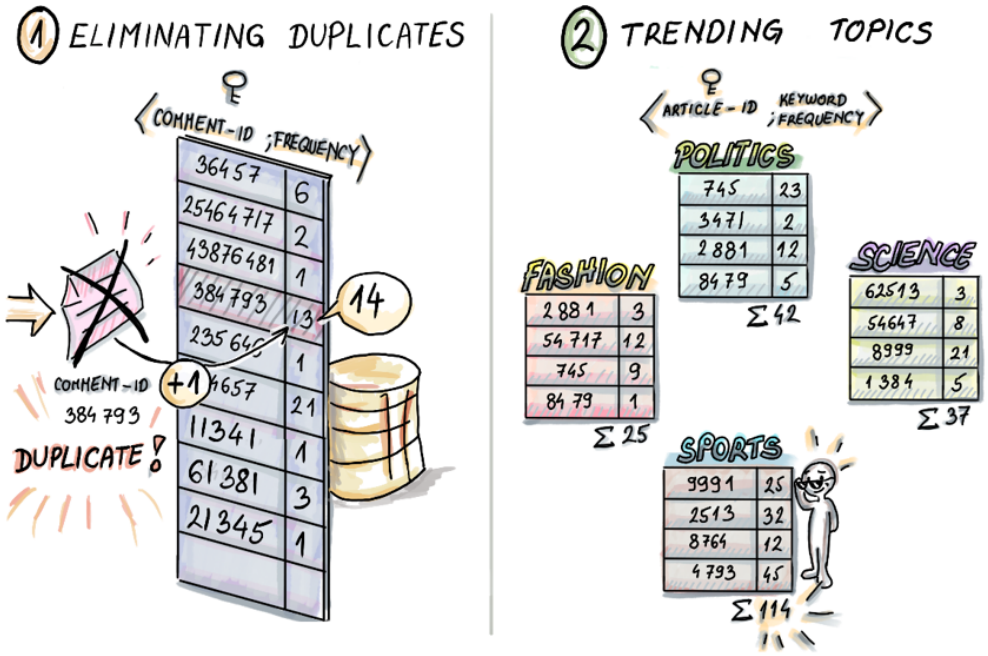



**Figure 1.1: In this example, we build a (comment-id, frequency) hash table to help us store distinct comment-id's with their frequency count. An incoming comment-id 384793 is already contained in the table, and its frequency count is only incremented. We also build topic-related hash tables, where, 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.**

We hope this small example demonstrates that we can very well start from a fairly common and naïve problem, and before we know it, catch ourselves with a number of clunky data structures we can't fit into memory.

You might think to yourself: Can't we multiply a couple of numbers beforehand, and easily predict how large the data structures are going to become? Well, in real life it often does not work like that. Rarely do people start designing their systems from scratch having massive data in mind. Companies often start out by trying to create a system that works, and can later become victims of their own success, where the user-base grows rapidly in a short amount of time, and the old system, designed by developers who have left, needs to grapple with this new demanding workload. Most often, parts of system get re-designed as the need arises.

When the number of items in a dataset becomes large, then every additional bit per item contributes to the burden on the system. 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 (see Figure 1.2).

Figure 1.2: Most common data structures, including hash tables, start to become difficult to store and manage with large amounts of data.

### 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* that comprise Part 1 of the book, the data structures that use a tiny amount of space to approximate the answers to these common questions:

- **Membership** --- Does a 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 vs. 8-16 bytes per item in a hash table).

A *Bloom filter* that we will discuss in Chapter 3, will use 8x less space than the `(comment-id -> frequency)` hash table and will 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 (or whether it's a string or a small or a large integer).

Another data structure that we will learn about in Chapter 4, *Count-Min sketch,* will use more than 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 also use Count-min sketch 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* from our Chapter 5 can estimate the cardinality of the set with only 12KB, exhibiting the error less than 1% of the true cardinality.

If we further relax the requirements on accuracy for each of these data structures, we can get away with even less space. Because the original dataset still resides on disk, there is also a way to control for an occasional error, so we are not stuck with the false positives, we just need a little extra effort to verify those.

#### COMMENT DATA AS A STREAM.

Quite likely, we might encounter the problem of news comments and articles in the context of a fast-moving *event stream* rather than as a static dataset. Assume that the event here constitutes any modification to the dataset, such as clicking 'Like' or inserting/deleting a comment or an article, and the events arrive real-time as streaming data to our system. We will learn more about this streaming data context in 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 cannot 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* that we cover in 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 (do not use this on a date). This sampling scheme offers itself conveniently to the one-pass-over-data context.

Answering some more granular questions about the comments data, like, how many likes a comment needs to be in the top 10% liked comments, 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* that we begin to explore in Chapter 9. External-memory algorithms address some of the most relevant problems of modern applications, such as for example, the design and implementation of database engines and their indices. 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 engineers actually implement their own storage engines, but almost all of them use them. 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 out there. Some of the most popular data structures to build databases include $B$-trees, $B^{\varepsilon}$-trees, and LSM-trees, and each serves a different sort of a workload. We will discuss these different sorts of performance and the tradeoffs in 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 a sorting algorithm that will efficiently sort data in a database or in a file on disk. You will learn how to do that in the last chapter of our book, 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 1 (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 the hashing chapter is planned as a review chapter, we suggest you use it as a refresher of hashing, 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 2 (Chapters 6-8) introduces data streams. From classical techniques like Bernoulli sampling and reservoir sampling to more sophisticated methods like sampling from a moving

window, 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) $\varepsilon$-approximate quantiles like Q-digest and T-digest.

Part 3 (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, $B^{\varepsilon}$-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 Design Manual* by Skiena[3], *Introduction to Algorithms* by Cormen, Leiserson, Rivest and Stein[4], *Algorithms* by Sedgewick and Wayne[5], and for a more introductory and friendly take on the subject, *Grokking Algorithms* by 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 code snippets in Python. 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],[11],[12]. With this book, we intend to present these different themes in one

[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.
[11] M. Kleppmann, Designing Data-Intensive Applications, O'Reilly, 2017.
[12] A. Petrov, Database Internals, O'Reilly, 2019.

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 (and learn from them).

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

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

There are many parameters in computers and distributed systems architecture that can shape the performance of a given application. Some of the main challenges that computers face in processing large amounts of data 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[13]. 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):

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

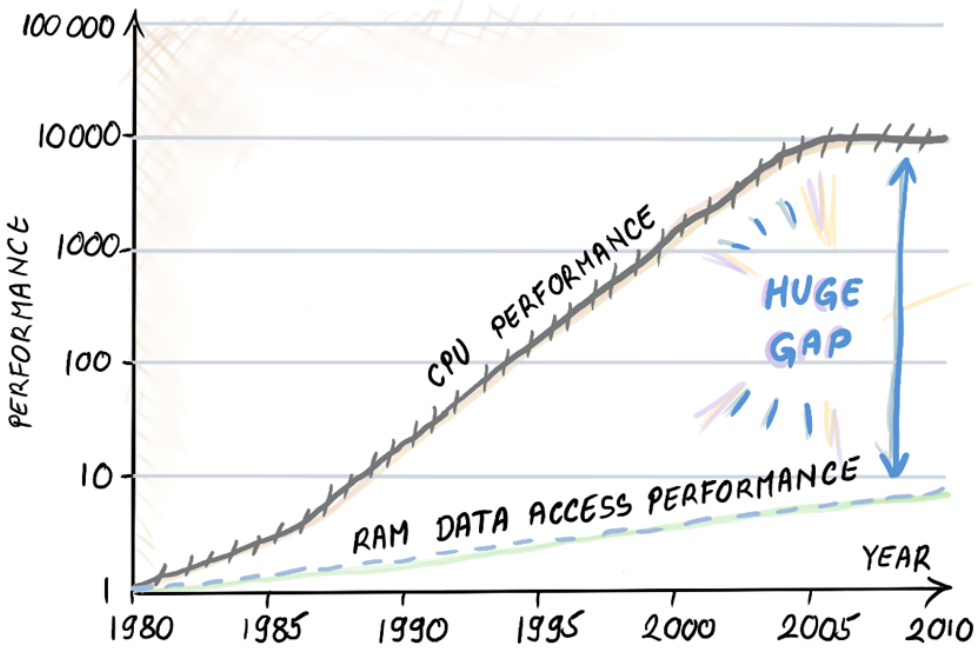



**Figure 1.3: CPU-Memory Performance Gap graph, adopted from Hennessy & Patterson's Computer Architecture. The graph shows the widening gap between the speeds of memory accesses to CPU and RAM main memory (the average number of memory accesses per second over time.) The vertical axis is on the log scale.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

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 only cares about optimizing CPU computation, then in many cases our analyses will not jive well with reality.

### 1.4.2 Memory hierarchy

Aside from the CPU-memory gap, there is 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[14]. 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.

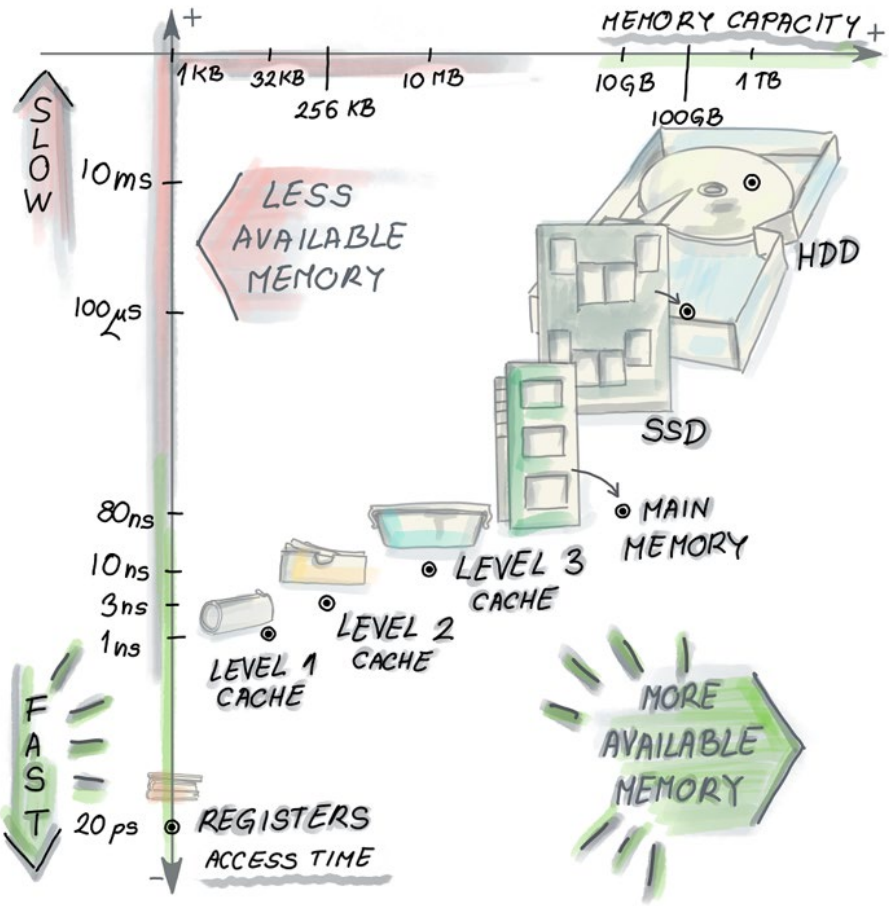



**Figure 1.4: Different types of memories in the computer. Starting from registers in the bottom left corner, that are blindingly fast but also very small, we move up (getting slower) and right (getting larger) with Level 1 cache, Level 2 cache, Level 3 cache, main memory, all the way to SSD and/or HDD. 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.**

[14] 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].

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"[15] 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 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[16]. 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 from hundreds of milliseconds to a couple of 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:

[15] D. A. Patterson, "Latency Lags Bandwith," Commun. ACM, vol. 47, no. 10, p. 71–75, 2004.
[16] J. L. Hennessy and D. A. Patterson, Computer Architecture, Fifth Edition: A Quantitative Approach, Morgan Kaufmann Publishers Inc., 2011.

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.

## 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[17] an example that for two computers that are 300 meters away, the lower 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**.

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

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 designed having space-related concerns in mind.

However, ultimately it is also important to keep in mind that modern systems have many performance metrics other than scalability, such as: security, availability, maintainability, etc. 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.

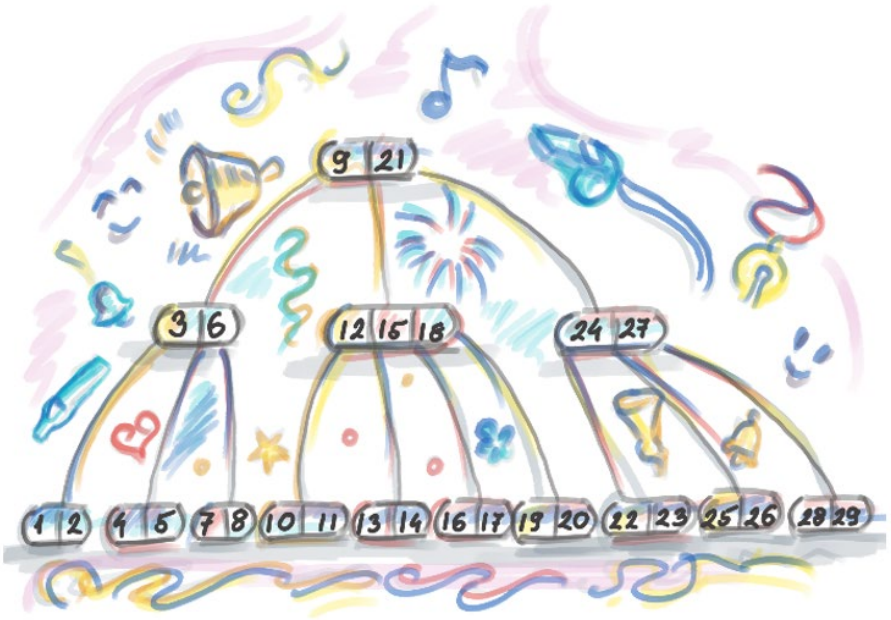



Figure 1.6: An efficient data structure with bells and whistles

## 1.6 Summary

- Applications today generate and process large amounts of data at a rapid rate. Traditional data structures, such as 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 you 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 trends are not likely to change 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 the 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 1 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 probably was not enough. Hash tables and hash functions 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 you typed in gets hashed and the hash is checked against the database to verify a match:

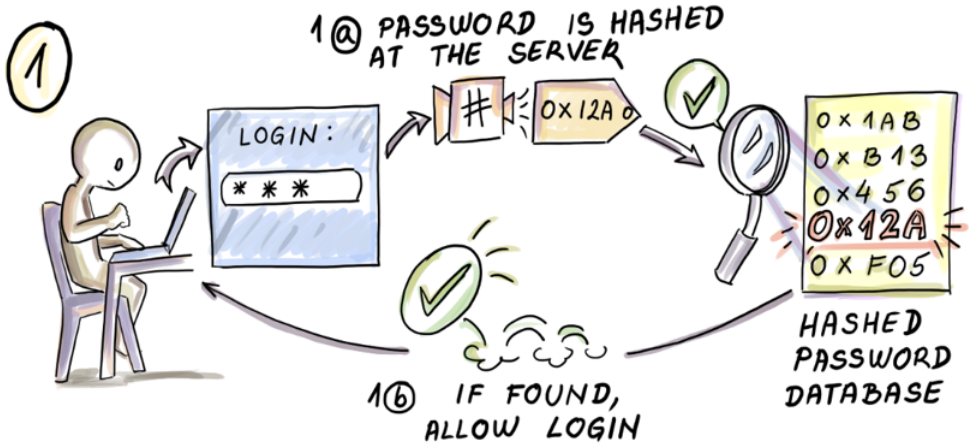



Figure 2.1: Logging into email and hashing.

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

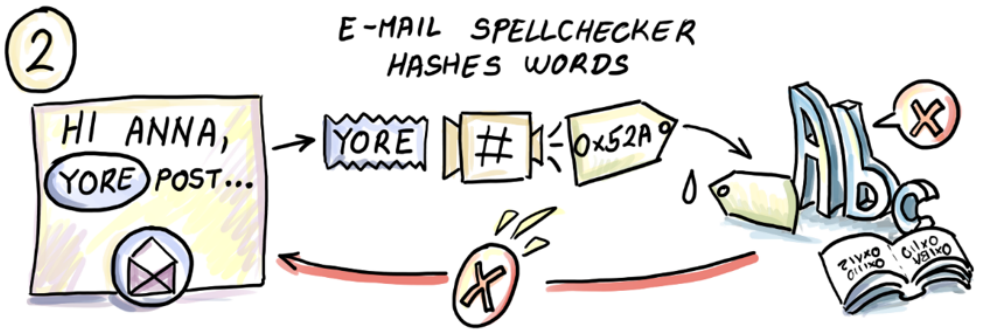



Figure 2.2: Spellchecking and hashing.

When the email is sent, often source-destination IP address pairs are hashed to determine to which intermediate server the packet should go in order to effectively load-balance the traffic.

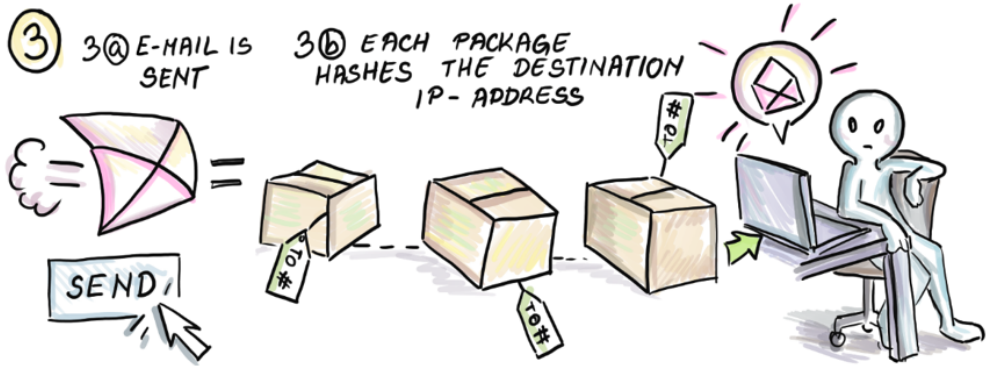



Figure 2.3: Network packets and hashing.

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.

This was one prominent example where hashing is used, but we bet you that in all the places where security is important, and in all the places where the lookup speed is important, you're bound to find stuff being hashed.

This chapter discusses both hashing and hash tables, and sometimes it switches unexpectedly back and forth. They are clearly not the same thing, but we will observe hashing less in the context of cryptography, for example, and more in the context of being utilized in a hash table --- or, in the future chapters, in some other data structures. Hash table are as ubiquitous as hashing, and programmers use them every day (e.g., when building key-value maps), often without knowing that there is a hash table sitting underneath.

To find out why hash tables are so widely used, we need to compare them to other data structures and see 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 thus lending 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 ($O(\log n)$), which, for many array sizes is effectively as good as constant time (logarithm with the base 2 of 1

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

billion is less than 30). 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 works in constant time by just inserting at the head of the list. Deletion is possible from anywhere in the list in constant time ($O(1)$) by re-linking a few pointers, provided we located the position where to insert/delete. More care is needed in the case of a singly-linked list, where in the case of a deletion, we would need to provide the pointer to the position before the element to be deleted. 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.

**Balanced binary search trees** have all dictionary operations dependent on the depth of the tree, and balanced binary trees use different balancing mechanisms (AVL, red-black, etc) that keep the tree depth at $O(\log n)$. Hence, all insert, lookup and delete operations take logarithmic time in the worst case. Just like with binary search, for many tree sizes, there is little difference in the performance between the constant time and logarithmic time. Logarithmic time is much closer to constant than to linear when it comes to speed, so we should be pretty happy to be able to do all dictionary operations in this amount of time guaranteed.

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. Balanced binary trees are provably your best choice for a dictionary if we compare all data structures that work based on element comparisons (<,>,=).

However, we are not limited to building data structures with comparisons only --- computers are capable of many other operations, including bit shifts, arithmetic operations, and other operations, and all of those are very cleverly used by hash functions to break out from the logarithmic bound.

The ultimate benefit of **hash tables** and **hashing** is that it cuts the dictionary operation costs down to $O(1)$ on all operations. If you are thinking this is too good to be true, to some extent you are 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 good 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 case of the hash table $O(n)$ will almost never happen, while in the case of an array, it will quite consistently happen.

The reason is the following: in a hash table, a good hash function will scramble the input item and based on that scrambled result, send the item to some bucket in a hash table where it can be found later. The word 'hash' comes from the French *'hachis'*, often used to

describe a type of dish where meat is chopped and minced into many little pieces (also related to *'hatchet'*). Because on average, different items will be minced into different results, then they are usually scattered to different buckets in a hash table. This enables the fast lookup, because no particular bucket will hold too many items. The lookup operation will mince the query element and directly look it up in the corresponding bucket. However, it is possible that the hash function minces very different input items into the same number and sends them all to the same bucket. In that case, mincing did not help our case, and we need to scan through all the items in the bucket to see whether our query item is present. This is an extremely rare case, and when it happens, we can decide to use a different hash function for that particular input.

Hash tables are, on the other hand, poorly suited for all applications where having your data ordered is important. The natural consequence of 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 in the next section. The table below summarizes the comparison between the most common data structures.

**Table 2.1: Summary of comparison of different data structure performance for dictionary operations. Unsorted arrays work well as data logs. Sorted arrays work well for the retrieval in a static dataset. Linked lists are good for fast deletions when the right position in the list is provided. Balanced binary search trees are both fast and versatile when it comes to various operations and guarantee fast worst-case performance. Predecessor/successor in balanced binary search trees runs in constant-time, provided the location of the element whose predecessor/successor we are looking for, otherwise it is logarithmic. Hash tables are the fastest in the expected sense. However, their ability to traverse the sorted order is not as good as that of balanced binary search trees.**

| | Lookup | Insert | Delete | Predecessor/ successor |
|---|---|---|---|---|
| Unsorted array | $O(n)$ | $O(1)$ | $O(n)$ | $O(n)$ |
| Sorted array | $O(\log n)$ | $O(n)$ | $O(n)$ | $O(1)$ |
| Linked list | $O(n)$ | $O(1)$ | $O(1)*$ | $O(n)$ |
| Balanced binary search tree | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | $O(1)$ |
| Hash table | $O(1)$ (expected) | $O(1)$ (expected) | $O(1)$ (expected) | $O(n)$ |

## 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 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. 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.

[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].

[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.

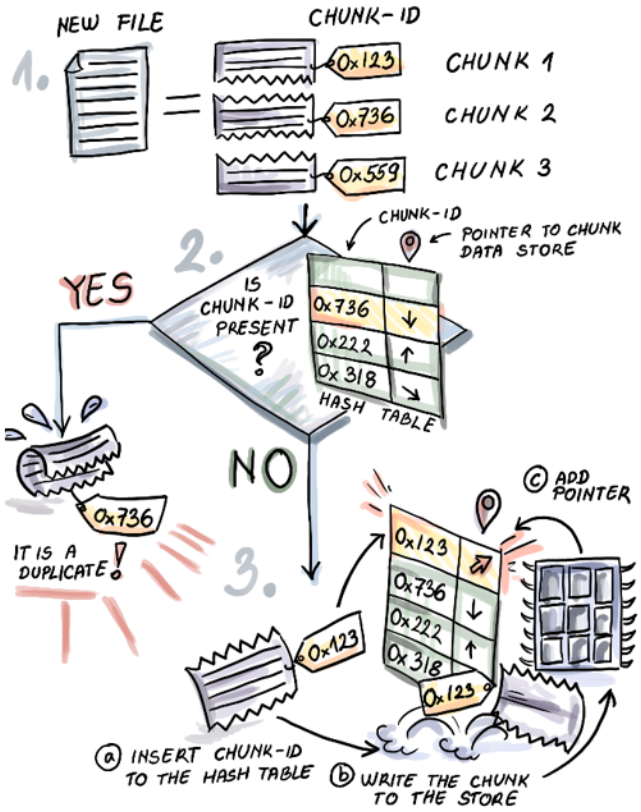



**Figure 2.5: Deduplication process in backup/storage solutions. When a new file arrives, it is split into small chunks. In our example, the file is split into three chunks, and each chunk is hashed (e.g., chunk 1 has chunk-id 0x123, and chunk 2 has chunk-id 0x736.) Chunk-id 0x123 is not found in the 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 3 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] 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 character by character. 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*, i.e., it works in linear time. False matches might contribute to the worst-case performance, but as discussed earlier, good hash function will make sure it does not happen as often. See Figure 2.6 for an example of how the algorithm works.

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

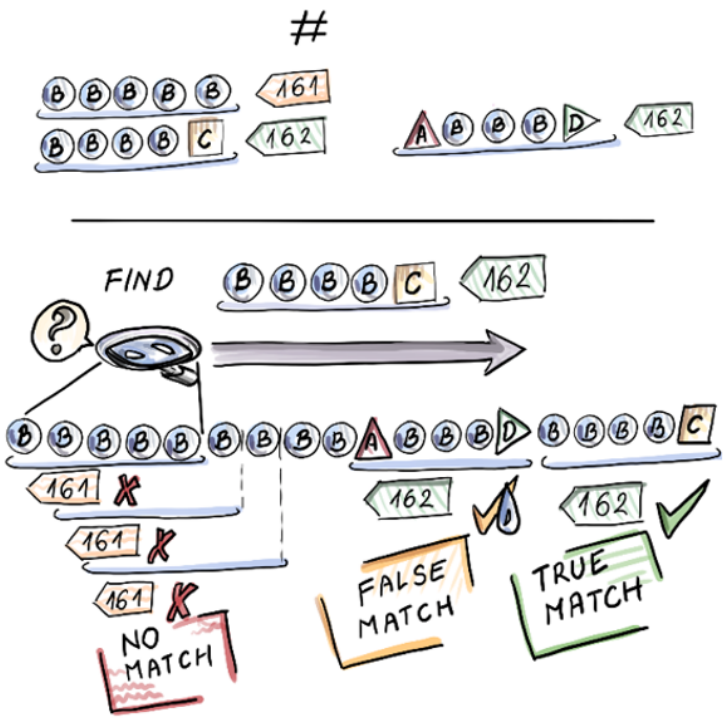



**Figure 2.6: Example of a Karp-Rabin fingerprinting algorithm. We are looking for a pattern p=BBBBC in the larger string t=BBBBBBBBBABBBDBBBBC. The hash of BBBBC is equal to 162 and it is a mismatch for the hash 161 of BBBBB that occurs at the beginning of the long string. As we shift right, we repeatedly 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.**

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.)

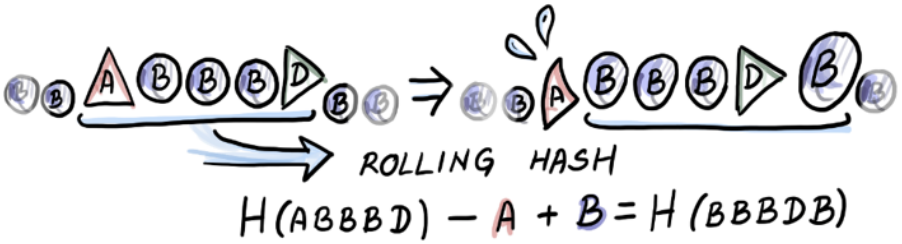



Figure 2.7: Rolling hash. Computing the hash for all but the first substring of t is a constant-time operation. For example, BBBDB, we needed to "subtract" A and "add" B to ABBBD.

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*, a way to map the fingerprint to its position in the documents where it occurs. From tis mapping, 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 use cases of hashing, let's do another peel of the onion of our understanding of dictionaries. Having read about all the tradeoffs between different aspects of performance in Section 2.2, you might be asking yourself why is it so hard to design a perfect data structure --- one that does lookups, inserts and deletes all in O(1) in the worst case. And more specifically, can we design a hash table that can guarantee constant-time operations? It's the "Why can't we just have it all?" question of the data structures. While in general this is not possible, there are special situations that would enable that.

For example, let's say you have a set of data; to make it simple, let's say you have a set of 100 numbers and equally-sized hash table. Because you have a static dataset, you could conjure up a custom hash function that would make sure that each item goes to a different bucket of the hash table, a hash function customized to this particular dataset. This would enable the perfect performance.

Another such scenario would be if you had a set of numbers that are positive integers and you know what the maximum is (call it $M$). If $M$ is not too large, we can design a hash table of size $M$, and have each number go to the bucket numbered by its value. Again, provided there are no duplicates, we get one element per bucket, resulting in constant-time performance on inserts, lookups and deletes.

But these are special situations, and generally speaking, knowing our data beforehand or having a very specific sort of input is more than we can expect most of the time.

The main challenge of hashing well is that hash functions need to provide a mapping of every 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, much larger than the size of our actual dataset, and consequently, the number of hash table buckets. We will refer to this set of all potential items as the universe $\mathrm{U}$, the size of our dataset as $n$, and the hash table size as $m$.

The values of $n$ and $m$ are roughly proportional. In other words, if you have 1 million elements to store, you would probably plan to have a hash table similar in size. Depending on what hash table design we want to use, we might use 0.5 million buckets, or 2 million buckets, or something else, but either way, a constant factor close to $n$. But both of those values 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-dd-ddd-dddd*, where *d* is a digit 0-9. Because each of the 12 digits can take on 10 different values, this means that $|U|=10^{12}$ and if $n=10^5$ (the dataset size), 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 --- consider the case of perfectly even distribution of the universe into buckets, then each bucket has $\frac{10^{12}}{10^6} = 10^6$ elements assigned to it. Because our dataset size is smaller than $10^6$, then it is possible to find such a dataset where all elements go to the same bucket. It would also be pretty bad if some constant fraction of our dataset, i.e., a half or a third went into the same 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 collision-resolution mechanisms: linear probing and chaining. There are many others, but we will cover these two as they are the most popular choices in the hash tables running underneath your code. 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:

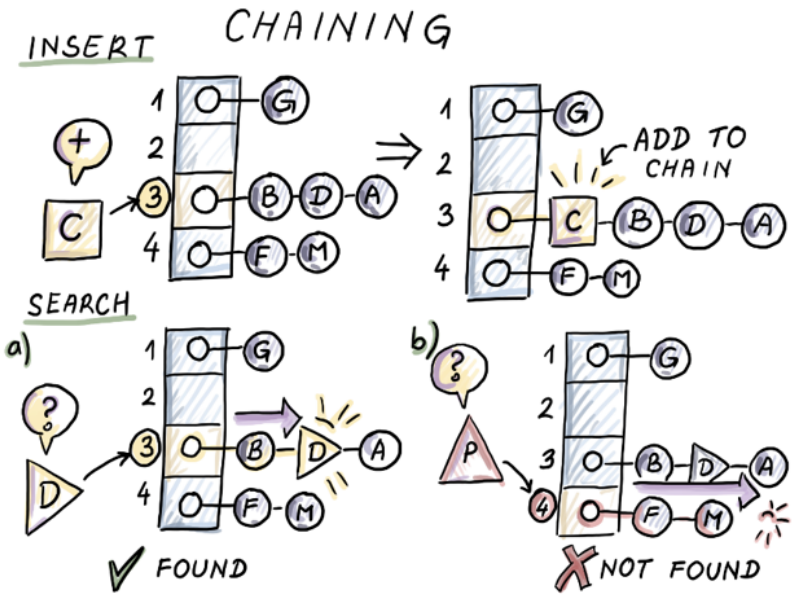



Figure 2.8: An example of insert and search with chaining.

**Linear probing** is a particular instance of *open addressing*, a hashing scheme where we store items inside the actual 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:

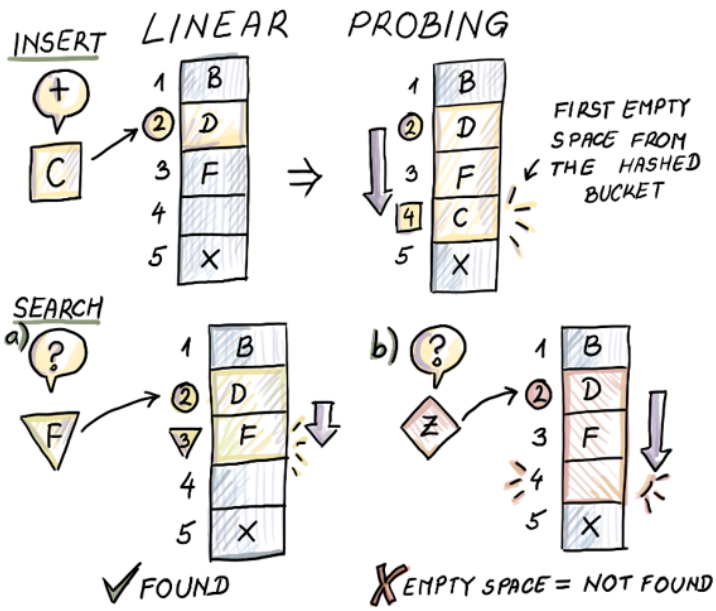



Figure 2.9: An example of insert and search with linear probing.

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 probability, and 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 $1-\frac{1}{n^c}$, where $c \geq 1$ is some constant. In our case, the high probability event will be a chain or a consecutive run in a hash table having an upper logarithmic limit on its size. In other words, we are upper-bounding the probability of the chain/run growing longer than logarithmic length. So, the higher the constant and the input size, the tinier becomes the chance of the high-probability

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

event not occurring, but already at $c = 1$, we're good. 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).

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. Decisions like this one can also affect the security of our application.

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?

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

[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.

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())
```

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 #A
```

**#A  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.

[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].

## 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 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
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/

[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].

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).


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.)

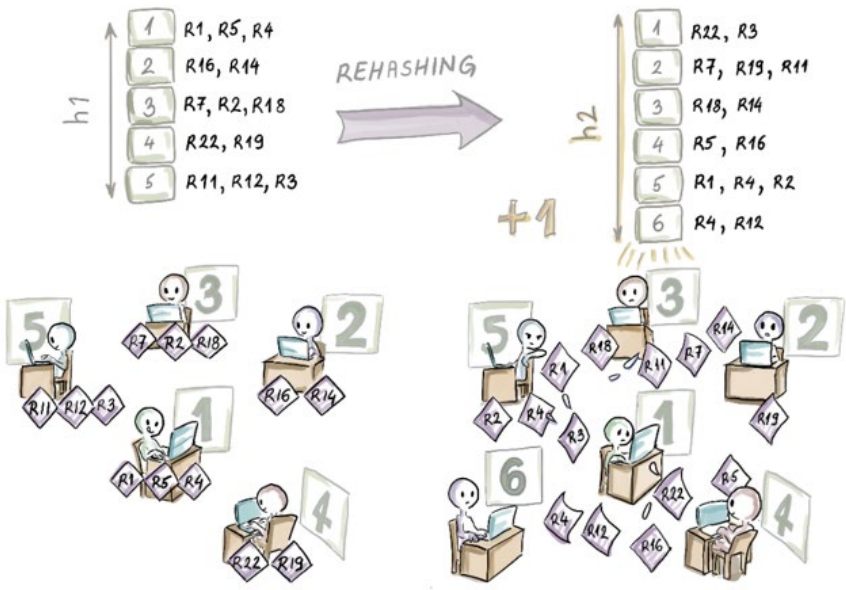



Figure 2.11: Rehashing is not feasible in a highly dynamic context, because one node join/failure triggers re-assignment of all resource-node allocations. In this example, changing the hash table size from 5 to 6 changed node allocations for most resources. The bottom right illustration shows the "in-between"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

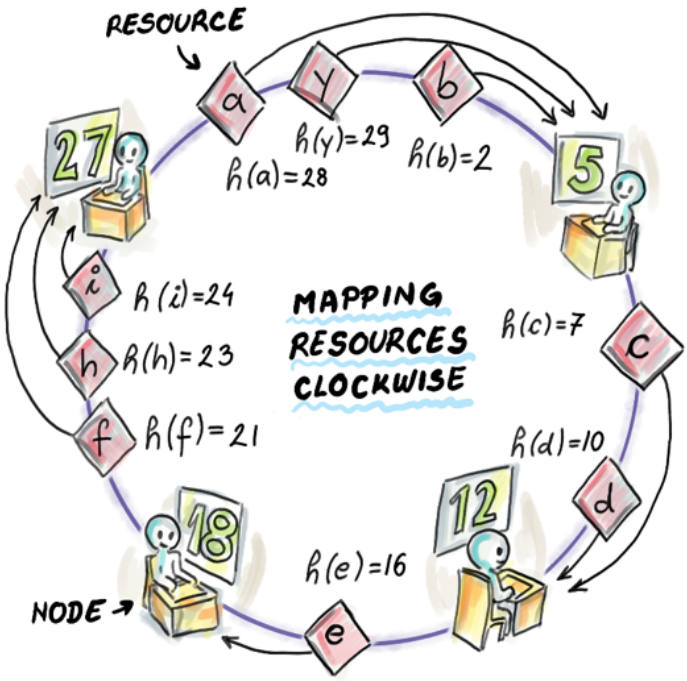



**Figure 2.12: Mapping resources to nodes in the hashring. The example shows the hashring $R = [0,31]$ and nodes whose whose hashes are 5, 12, 18 and 27. Resources a, y and b are assigned to node 5, c and d are assigned to node 12, e is assigned to node 18, and f,h and i are assigned to node 27.**

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):
        return self.min <= hashValue <= self.max
```

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)>
        [CA]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.

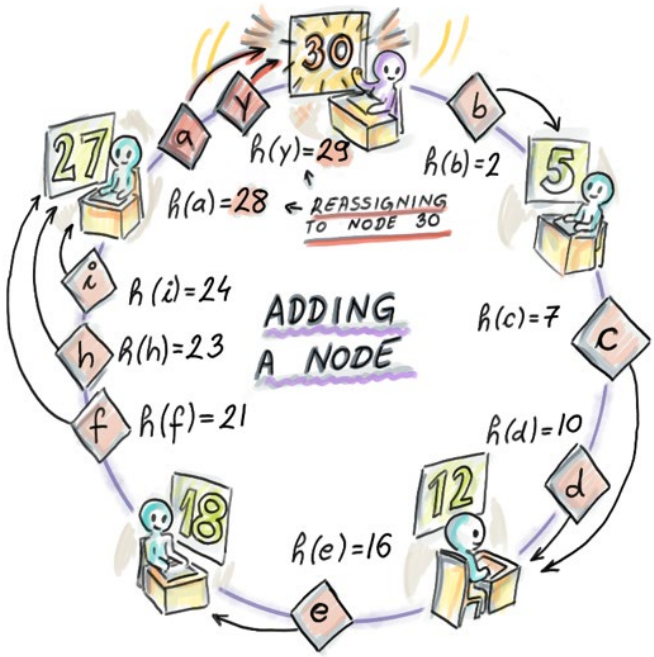



**Figure 2.13: New node arrival. The resources a and y, with respective hash values 28 and 29, are now being re-assigned to the newly inserted node with the hash value 30.**

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:

```
   def moveResources(self, dest, orig, deleteTrue):#A
       delete_list = []
for i, j in orig.resources.items():
    if(self.distance(i, dest.hashValue) < self.distance(i, orig.hashValue) or
       [CA]deleteTrue):
        dest.resources[i] = j
        delete_list.append(i)
        print("\tMoving a resource " + str(i) +  " from " + str(orig.hashValue) +
           [CA]" to " + str(dest.hashValue))

       for i in delete_list: #B
    del orig.resources[i]
```

**#A Move some resources from from orig to dest**
**#B delete the re-assigned resources from orig**

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):
 newNode = Node(hashValue)
       if self.head is None: #A
     newNode.next = newNode
     newNode.previous = newNode
     self.head = newNode
     print("Adding a head node " +  str(newNode.hashValue) + "...")
 else:
     temp = self.lookupNode(hashValue) #B
     newNode.next = temp
     newNode.previous = temp.previous
     newNode.previous.next = newNode
     newNode.next.previous = newNode
     print("Adding a node " +str(newNode.hashValue)+ ". Its prev is " +
        [CA]str(newNode.previous.hashValue) + ", and its next is " +
        [CA]str(newNode.next.hashValue) + ".")
     self.moveResources(newNode, newNode.next, False)
            if hashValue < self.head.hashValue: #C
                self.head = newNode
```

**#A Empty hashring**
**#B Successor**
**#C Changing the head pointer**

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.

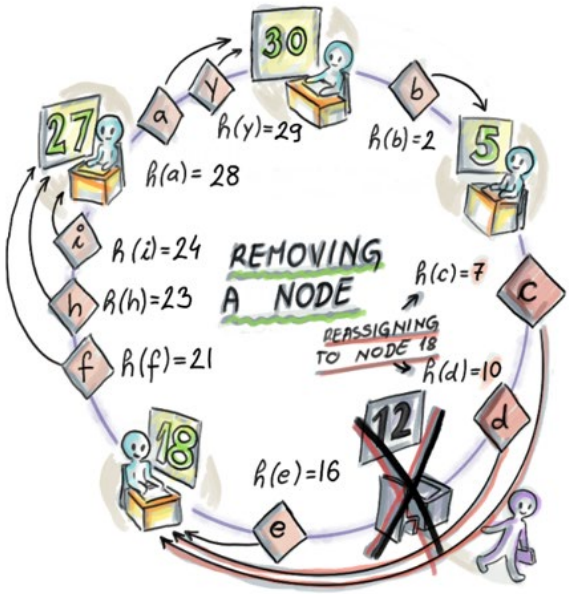



**Figure 2.14: Node removal. In this example, the node with the hash value 12 leaves the network, and its resources c and d, with hash values 7 and 10 respectively, are re-assigned to the node with the hash value 18, the previous successor of 12.**

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: #A
                self.head = temp.next
        if self.head == self.head.next: #B
                    self.head = None
    return temp.next
else:
    print("Nothing to remove.") #C
```

**#A Removing the head item**
**#B If removing from one-item hashring**
**#C 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 = " ")
        if not bool(temp.resources):
            print("Empty", end = "")
               else:
                   for i in temp.resources.keys():
                print(str(i), 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 $i \leq k$. The fingers' 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`:

[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.

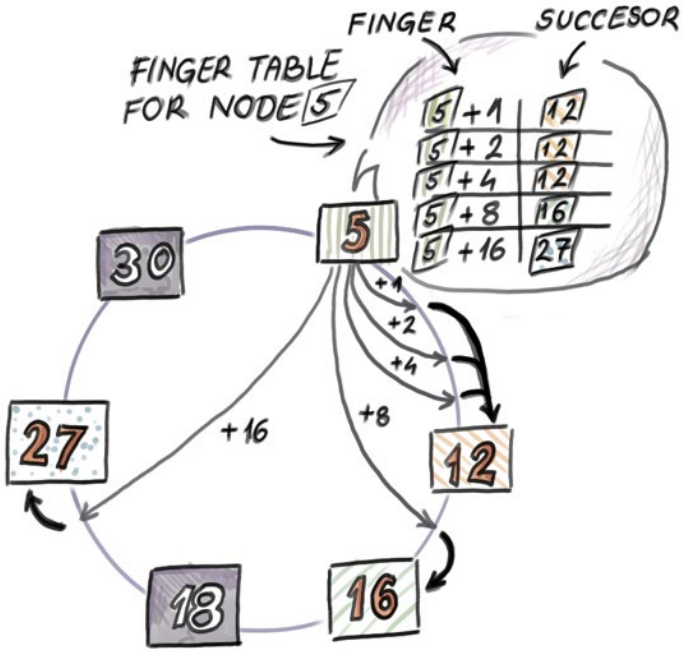



Figure 2.15: Example finger table for the node 5, on the hashring where R=[0,31]. Node 5 has 5 entries stored in its finger table, for the successors of the points 5+1=6, 5+2=7, 5+4=9, 5+8=13, and 5+16=21. The respective successors are 12, 12, 12, 16 and 27.

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:

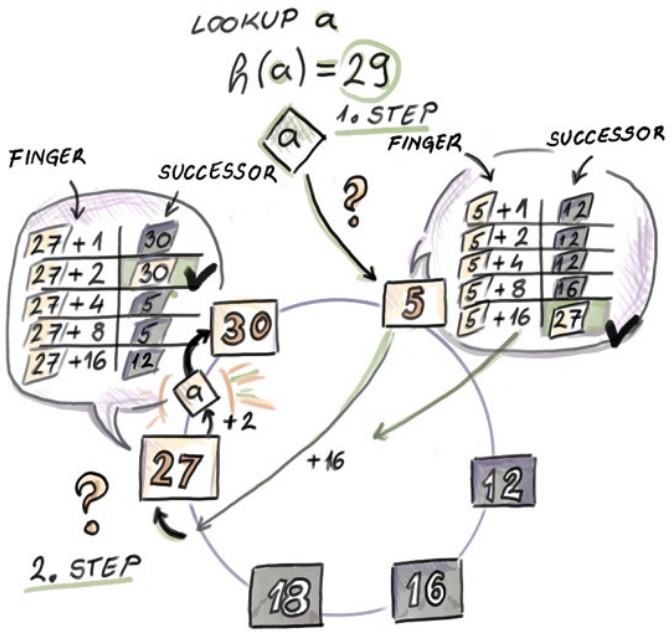



**Figure 2.16: Lookup procedure with a finger tables. To locate resource 29 starting from node 5, we first follow the finger (21=5+16) as it is the finger with smallest distance to 29. Its successor is 27, so the request is forwarded to 27. In the finger table of node 27, we take the finger 2, which 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 2.1**

**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 2.2**

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 2.3**

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. The bigger the hash table, the larger becomes the effect of cache efficiency than the effect of the number of probes on the performance.
- 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: Bloom filter and Quotient 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 quotienting works, as well as merging and resizing of a quotient filter
- Comparing the performance of Bloom filter and quotient filter

Bloom filters have become a standard in systems that process large datasets. Their widespread use, especially in networks and distributed databases, comes from the effectiveness they exhibit in situations where we need a hash table functionality, but do not have the luxury of space. They were invented in 1970s by Burton Bloom[1],[2] but they only really "bloomed" in the last few decades due to an increasing need to tame and compress big datasets. Bloom filters have also piqued the interest of the computer science research community who developed many variants on top of the basic data structure, in order to address some of its shortcomings and adapt it to different contexts.

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., 1 byte per item or less. This is a significant saving when keys take up 4-8 bytes. Bloom filters do not store the items themselves, and they use less space than the lower theoretical limit required to store the

[1] 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.
[2] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," in Internet Mathematics, 2002, pp. 636-646.

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. In situations 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, Bloom filters are used in Google's Webtable[3] and Apache Cassandra[4] that are among the most widely used distributed storage systems for handling massive 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 (a key might be a URL, and a value might be website attributes or contents, for example.) Webtable and Cassandra simultaneously handle adding new content into tables and answering queries, and when a query arrives, it is important to locate the SST containing the queried content without explicitly querying each table. To that end, dedicated Bloom filters in RAM are maintained, one per SST, to route the query to the correct table.

[3] 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.

[4] 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].

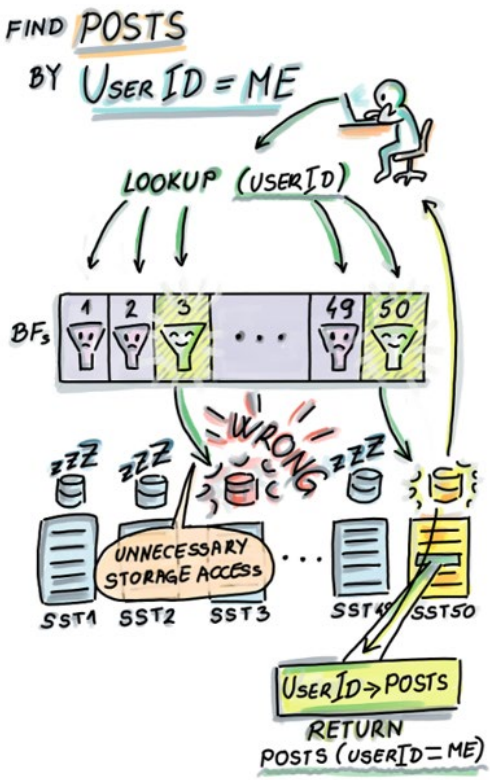



**Figure 3.1: Bloom filters in distributed storage systems. In this example, we have 50 Sorted String Tables (SSTs) on disk, and each table has a dedicated Bloom filter that can fit into RAM due to its much smaller size. When a user does a lookup, the lookup first checks the Bloom filters, thus avoiding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

In the example shown in Figure 3.1, we are given 50 SSTs on disk, and 50 associated Bloom filters in RAM. As soon as the Bloom filter reports `Not Present` on a lookup, the query is routed to the next Bloom filter. The first time a Bloom filter reports the item as `Present` (in this example, SST3's Bloom filter), we go to disk to check whether the item is present in the table. In case of a false alarm, we continue routing the query until a Bloom filter reports `Present`, and the data is also found on disk and returned to the user, as with SST50.

Bloom filters are most useful when placed strategically in high-ingestion systems. For example, having an application perform an SSD/disk read/write can easily bring down the throughput of an application from hundreds of thousands ops/sec to only a couple of thousands or even a couple of hundreds ops/sec. Instead, if we place a Bloom filter in RAM to serve the lookups, we can avoid this performance slump, and maintain consistently high throughput across different components of an application. More details on how Bloom filters are integrated in the bigger picture of streaming systems will be given in Chapter 6.

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 are more mathematically-inclined, we will spend some time understanding where the formulas relating important Bloom filter parameters come from and exploring whether one can do better than Bloom filter.

In Section 3.7, we will explore a very different data structure that is functionally similar to Bloom filter. **Quotient filter**[5] is a compact hash table that offers space-saving benefits at the cost of false positives, but also offers other advantages. If you are confident in your knowledge of Bloom filters, skip right ahead to Section 3.7.

## 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 $h_1$, $h_2$,…, $h_k$, 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
```

[5] 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.

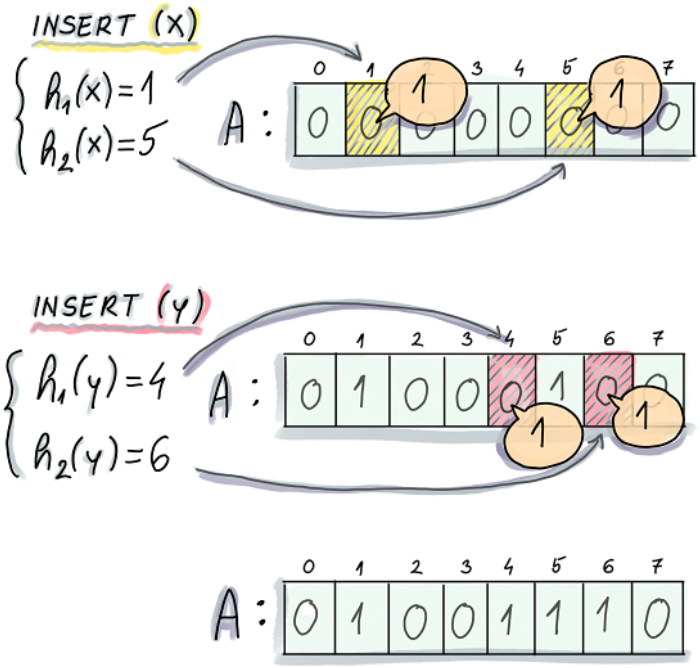



**Figure 3.2: Example of insert into Bloom filter. In this example, an initially empty Bloom filter has m=8, and k=2 (two hash functions). To insert an element x, we first compute the two hashes on x, the first one of which generates 1 and the second one generates 5. We proceed to set A[1] and A[5] to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

Figure 3.3 shows an example of a lookup on the resulting Bloom filter from Figure 3.2, and how it can generate true positives (on element $x$ that was actually inserted) and false positives (on element $z$ that was never inserted):

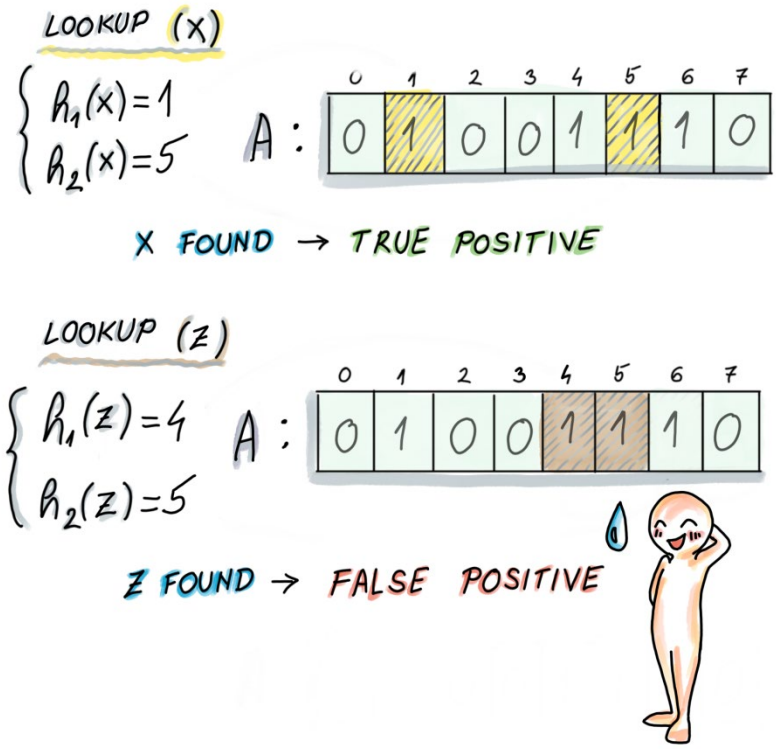



**Figure 3.3: Example of a lookup on a Bloom filter. To do a lookup on x, we compute the hashes (which are the same as in the case of an insert), and we return Found/Present, as both bits in corresponding locations equal 1. Then we do a lookup of an element z, which we never inserted, and its hashes are respectively 4 and 5, and bits at locations A[4] and A[5] equal 1, thus we again return Found/Present.**

As seen in the Figure 3.3, false positives can occur when some other elements together set the bits of some other element to 1 (in this example, two previous items $x$ and $y$ have set $z$'s bit locations to 1).

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; we will see that on average, an unsuccessful lookup in a well-configured Bloom filter takes about 1-2 probes before giving up. This makes for an incredible fast lookup operation.

## 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. Web proxies use caches to reduce web traffic by maintaining a local copy of recently accessed links from which they can serve clients' requests for webpages, files, etc. One of the protocols[6] suggests that each proxy keep a summary of the neighboring proxies' cache contents as well, and check the summaries before forwarding any queries to neighbor proxies. Squid implements this functionality using Bloom filters that they call Cache Digests[7] (see Figure 3.4).

Cache contents frequently go out of date, and Bloom filters are occasionally broadcasted between the neighbors. Because Bloom filters are not always up-to-date, false negatives can arise: when a Bloom filter claims the element is present in a proxy, but the proxy does not contain the resource anymore.

[6] 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.

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

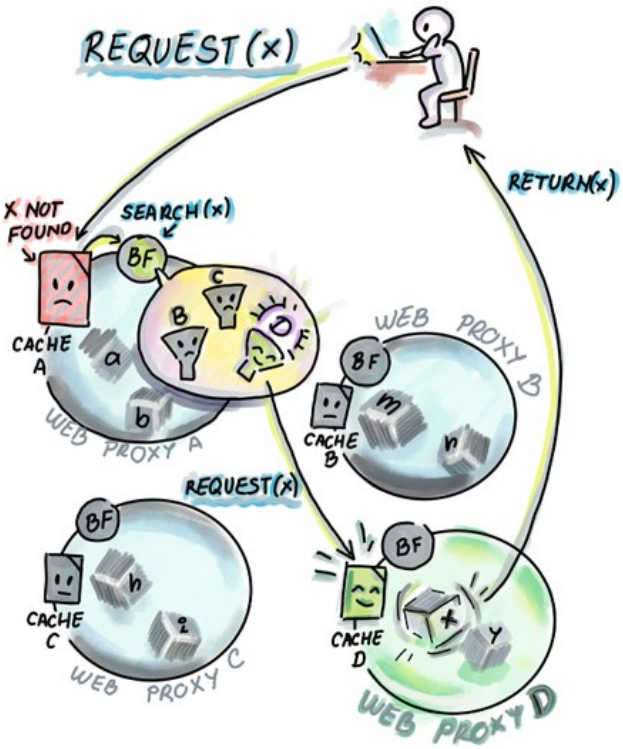



Figure 3.4: Usage of Bloom filter in Squid web proxy. A user requests a web page x, and a web proxy A can not find it in its own cache, so it locally queries the Bloom filters of B, C and D. The Bloom filter for D reports Present, so the request is forwarded to D. The resource is found at D and returned to the user.

### 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 ensured by having each node aware of 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):

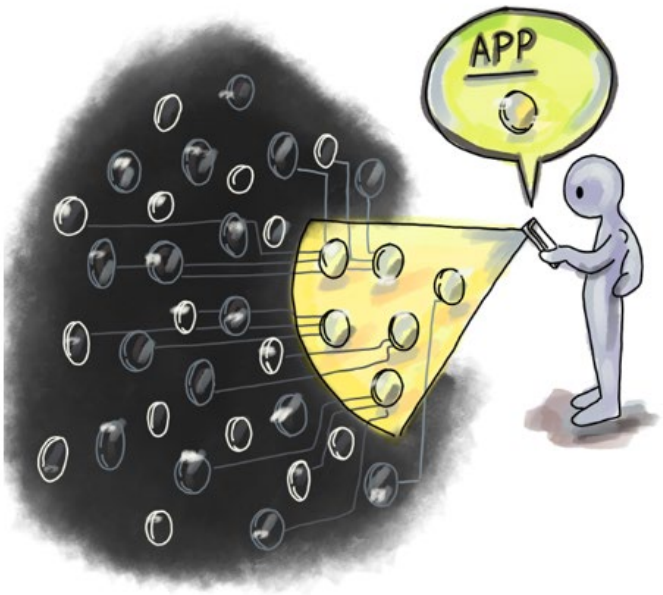



Figure 3.5: In Bitcoin, light clients can broadcast what transactions they are interested in, and thereby block the deluge of updates from the network.

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.[8]

More recently, Bitcoin has also offered other methods of transaction filtering with improved security and privacy properties.

## 3.3 A simple implementation

A basic Bloom filter is fairly straightforward to implement. Below we show a simple implementation that uses Python wrapper for MurmurHash `mmh3` that we discussed in Chapter 2. By setting `k` different seeds, we are able to obtain `k` different hash functions. The implementation also uses `bitarray`, a library that allows space-efficient encoding of the filter that you would need to install in order to run the code that's coming up.

[8] 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.

```
import math
import mmh3
from bitarray import bitarray

class BloomFilter:
    def __init__(self, n, f): #A
        self.n = n
 self.f = f
 self.m = self.calculateM()
 self.k = self.calculateK()
 self.bit_array = bitarray(self.m)
        self.bit_array.setall(0)
 self.printParameters()

    def calculateM(self):
        return int(-math.log(self.f)*self.n/(math.log(2)**2))

    def calculateK(self):
        return int(self.m*math.log(2)/self.n)

    def printParameters(self):
        print("Init parameters: ")
 print(f"n = {self.n}, f = {self.f}, m = {self.m}, k = {self.k}")

    def insert(self, item):
        for i in range(self.k):
     index = mmh3.hash(item, i) % self.m
     self.bit_array[index] = 1

    def lookup(self, item):
        for i in range(self.k):
     index = mmh3.hash(item, i) % self.m
     if self.bit_array[index] == 0:
         return False
 return True
```

**#A Provide the number of elements and the desired false positive rate**

You may try out the implementation by inserting a couple of items of type `string`, such as shown below.

```
    bf = BloomFilter(10, 0.01)
    bf.insert("1")
    bf.insert("2")
    bf.insert("42")
    print("1 {}".format(bf.lookup("1")))
    print("2 {}".format(bf.lookup("2")))
    print("3 {}".format(bf.lookup("3")))
    print("42 {}".format(bf.lookup("42")))
    print("43 {}".format(bf.lookup("43")))
```

The constructor of this sample implementation lets the user set the maximum number of elements ($n$) and the desired false positive rate ($f$), while the constructor does the job of setting two other parameters ($m$ and $k$). This is a common choice of a constructor as we often know how large of a dataset we're dealing with, and how high of a false positive rate

we are willing to admit. To understand how the remaining parameters are set in this implementation and how to configure a Bloom filter to get the most bang for your buck, read on.

## 3.4 Configuring a Bloom filter

First, we outline main formulas relating important parameters of the Bloom filter. We use the following notation for the four parameters of the Bloom filter:

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

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

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

If you would like to understand how this formula is derived, there are more details in Section 3.5. Right now, we are more interested in reasoning visually about this formula.

Figure 3.6 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. 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.

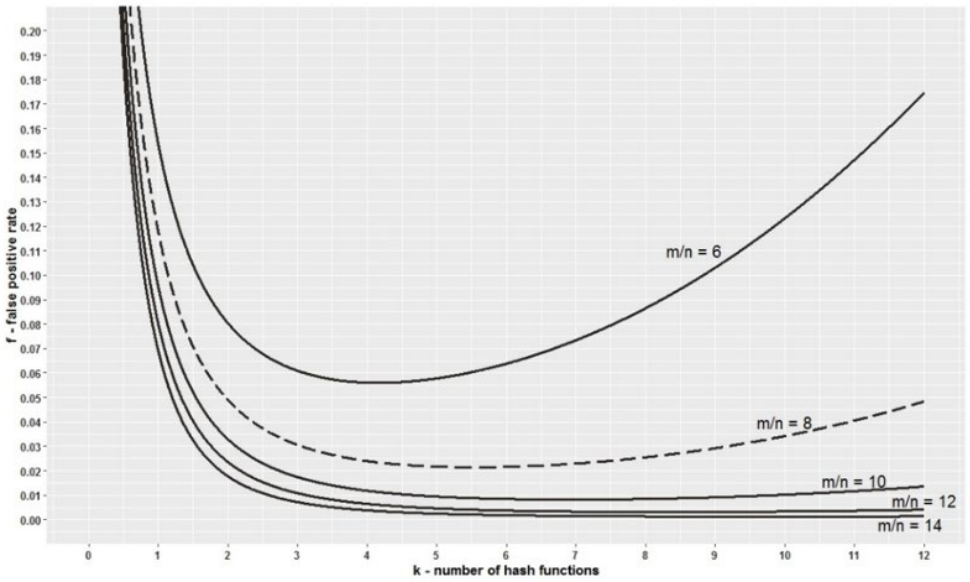



**Figure 3.6: The plot relating the number of hash functions (k) and the false positive rate (f) in a Bloom filter. The graph shows the false positive rate for a fixed bits-per-element ratio (m/n), different curves corresponding to different ratios.**

Starting from the top to the bottom curve, we have 6, 8, 10, 12 and 14 respectively as choices for bit-per-element ratio. As the bits-per-element ratio increases, the false positive rate drops, given the same number of hash functions. 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. This two-fold effect occurs because having more hash functions allows a lookup more chance to find a 0, but also on an insert, sets more bits to 1. Having said that, the shape of the curves indicates that it is better to err on the larger than smaller number than the optimal.

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.

The minimum false positive rate for each curve gives us the optimal *k* for a particular bits-per-element ratio (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*):

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

Formula 3 is obtained by plugging Formula 2 into Formula 1. The constructor in our implementation above takes in $n$ and $f$, and uses them to compute $m$ and $k$, using Formulas 2 and 3, while making sure that $k$ and $m$ have to be integers. If Formula 2 produces a non-integer, and we need to round up or down, then Formula 3 is also not an absolutely exact false positive rate anymore. The only correct formula to plug into, to obtain the exact false positive rate is Formula 1, but even with Formula 3, the difference produced by rounding up or down is minor. Often it is better to choose the smaller of the two possible values of $k$, because it reduces the amount of computation.

One might find the expression $(1/2)^k$ in Formula 3 interesting in connection with the fact that a false positive occurs when a lookup encounters $k$ 1s in a row. Indeed, an optimally filled Bloom filter has about 50% of probability that a random bit is 1. This is another way of saying that if your Bloom filter has too many 0s or too many 1s, the chances are it is not well configured.

There are different ways to write constructors for the Bloom filter, depending on what initial parameters we have been provided. Usually, $k$ is a synthetic parameter that is calculated from other, more organic requirements such as space, number of elements and the false positive rate. Either way, if you ever have to write different Bloom filter constructors, here are a couple of examples of how to compute the remaining parameters of Bloom filter.

**Example 3.1 Calculating *f* from *m*, *n*, and *k***

You are trying to analyze the false positive rate of an already existing Bloom filter that was initially built to store $10^6$ elements, but ended up storing ten times more. The Bloom filer has been giving very poor performance, and we are interested in its false positive rate. The filter capacity is 3MB, and it uses 2 hash functions.

**Answer for Example 3.1**

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 3.2 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 3.2**

From Formula 2, the ideal number of hash functions should be $k = \ln 2 * 8 * 10^6 / 10^6 = 5.544$. 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.

### 3.4.1 Playing with Bloom filters: Mini experiments

**Exercise 3.1**

Use the provided Python implementation to create a Bloom filter where $n = 10^7$ and $f = 0.02$. For elements, use uniform random integers (without repetition) from the range $= [0, 10^{12}]$ and convert them into strings (insert them as strings). Save the inserted elements into a separate file.

Perform $10^6$ lookups that are uniformly randomly selected elements from $U$. Keep track of the false positive rate and verify whether it is ~2%. Measure the time required to perform the lookups. Make sure not to include in your measurements the time required for the random number generation involving selecting the keys.

Now perform $10^6$ successful lookups by uniformly randomly (without repetition) choosing from the file of inserted elements, and measure the time required to perform the lookups. Make sure you do not include the time required to read from file, or generate random numbers. What takes more time, uniform random lookups or successful lookups?

**Exercise 3.2**

Using the provided implementation, create a Bloom filter such as the one in Example 2. Now create two other filters, one in which the dataset is 100 times larger than the original one, and another one in which the dataset is 100 times smaller, leaving the same false positive rate. What do you notice about the size of the filter as the data set size changes?

**Exercise 3.3**

In some literature, a variant of Bloom filter is described where different hash functions have the "jurisdiction" over different parts of the Bloom filter. In other words, $k$ hash functions split the Bloom filter into $k$ equal-sized consecutive chunks of $m/k$ bits, and during an insert, the $i$th hash function sets bits in the $i$th chunk. Implement this variant of the Bloom filter and check whether and how this change might affect the false positive rate in comparison to the original Bloom filter.

Next we give some intuition about how the formula for the false positive rate of Bloom filter is derived, and how lower bounds for the space-error tradeoff in data structures work. The coming section is theoretical and intended for the mathematically inclined readers. If you are more practically inclined, feel free to skip right ahead to Section 3.6.

## 3.5 A bit of theory

First let's see where the main formula for the Bloom filter false positive rate (Formula 1) comes from. 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 hash function maps keys uniformly randomly over the range [0… $m$ − 1].

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

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

because we need to get $k$ 1s in order to report `Present`. Obtaining $k$ 1s can also be a result of a successful lookup of an actually inserted element, however if we consider queries to be uniformly randomly selected from a universe much larger than the dataset, then the probability of a true positive is a negligible fraction of this quantity.

The value of $t$ is impossible to know before all inserts take place 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.:

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

The value of $p$ will in the probabilistic sense translate to the percentage of 0s in the filter($t$). 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 $p \approx e^{-nk/m}$.

It is tempting to just replace $t$ in the expression for the false positive rate with $p$, and this will give us Formula 1. After all, $p$ describes the expected value of a random variable denoting the percentage of 0s in the filter --- but what if the actual percentage of 0s can substantially vary from its expectation?

Using *Chernoff bounds* --- a theorem curtailing probability of a random variable deviating substantially from its mean --- we can show, in fact, that the fraction of 0s in the Bloom filter is highly concentrated around its mean. The general statement of Chernoff bounds holds for random variables $X$ that are a sum of mutually independent indicator random variables. We define a random variable $X$ that denotes the total number of 0s in Bloom filter $X = \sum_{i=1}^{m} X_i$, where $X_i = 0$ if the $i$th bit in Bloom filter equals 1, and $X_i = 1$ otherwise.

Using Chernoff bounds, we will show that the value $X$ does not significantly deviate from its mean. In our case, $X_i$s are not independent, however they are slightly negatively correlated (even better!) --- one bit being set to 1 slightly lowers the chance of other bits being set to 1.

The general statement of Chernoff upper bound (we can do something similar for the lower bound) where $\mu$ is the mean of the random variable $X$ is as follows:

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

Applied to our case, $\mu = E[X] = mp = me^{-nk/m}$ . If we choose $\delta = 1$, and plug into the Chernoff bound, we obtain the probability that $X$ will deviate from its mean by more than a factor of 2:

$$\Pr[X > 2pm] < \left(\frac{e}{4}\right)^{me^{-nk/m}}$$

We can safely assume that $nk = \theta(m)$, which further gives us that the probability that $X$ deviates from the mean by more than a factor of 2 is exponentially small

($< \left(\frac{3}{4}\right)^{\theta(m)}$). Hence $p$ is a good approximation for $t$, the percentage of 0s in the Bloom filter, which justifies replacing $t$ in the formula from the beginning of this section. This finalizes our derivation of Formula 1.

### 3.5.1 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? To answer this question, we need to derive a *lower bound* that relates the amount of space the data structure uses in bits (*m*) with the maximum false positive rate the data structure allows (*f*). Note that the lower bound is independent of any particular data structure design and tells us about theoretical limits of *any* data structure --- even one that has not been invented yet.

A data structure is an $m$-bit string and has a total of $2^m$ distinct encodings. Each individual encoding of a data structure, in addition to reporting `Present` for some $n$ elements, will also admits $f(U-n)$ false positives, a fraction $f$ of the rest of the universe. Of the total $n+f(U-n)$ elements for which the data structure reports `Present`, we do not know who are the true positives and who are the false positives, so one encoding of the data structure serves to represent every $n$-sized subset we grab. There are $\binom{n+f(U-n)}{n}$ such sets.

In total, the data structure neds to be able to "cover" all $\binom{U}{n}$ $n$-sized subsets in the universe $U$. Together, all these facts give us the inequality shown in the Figure 3.6 below that describes our lower bound:

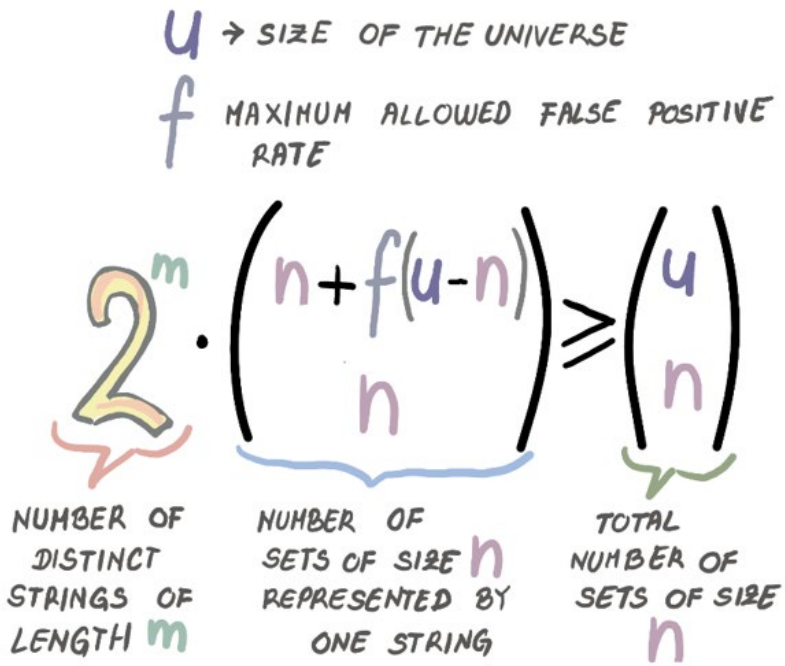



Figure 3.7: The inequality describing the space-error lower bound.

Taking the logarithm on both sides along with the fact that $U \gg n$, and some additional algebraic manipulation, the inequality from Figure 3.7 will give us that

$$m \geq n \log_2 \left(\frac{1}{f}\right).$$

Now how does that compare to the false positive rate of Bloom filter?

From Formulas 2 and 3, we can derive the relationship between $m$ and the false positive rate of Bloom filter (here we will refer to it as $f_{BF}$). We have that $f_{BF} = (1/2)^k \geq (1/2)^{\ln 2(m/n)}$. Again, taking the logarithm, and doing some algebraic simplifications, we get that

$$m \geq n \log_2 \left(\frac{1}{f_{BF}}\right) \log_2 e.$$

Comparing that to the lower bound, we obtain that Bloom filter space is $\log_2 e \approx 1.44$ factor away from optimal. There exist, in fact, data structures that are closer to the lower bound than Bloom filter, but some of them are very complex to understand and implement.

## 3.6 Bloom filter adaptations and alternatives

The basic Bloom filter data structure has been widely used in a number of systems, but also Bloom filter leaves a lot to be desired, and computer scientists have developed various modified versions of Bloom filters that address some of its drawbacks. For example, the standard Bloom filter does not handle deletions. There exists a version of the Bloom filter called *counting Bloom filter*[9] 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 (about 4x) 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 do with hash tables, is that in the Bloom filter, we do not store the items nor the fingerprints, so the original keys need to be brought back from the persistent storage in order to build a new Bloom filter.

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*[10] 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

[9] 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.

[10] J. Bruck, J. Gao and A. (. Jiang, "Weighted Bloom Filter," in IEEE International Symposium on Information Theory, 2006.

of Bloom filters that are *adaptive*, i.e., upon the discovery of a false positive, they attempt to correct it.[11]

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.

## 3.7 Quotient filter

Quotient filter[12], at its simplest, is a cleverly designed hash table that uses linear probing. The difference between a quotient filter and a common hash table is that instead of storing keys into slots as in a classic linear probing hash table, the quotient filter stores the hashes (the term 'fingerprint' will be used interchangeably). More precisely, the quotient filter stores a piece of each hash, but as we will see, it is able to reliably restore an entire hash.

On the "fidelity spectrum", quotient filter is somewhere between hash tables and Bloom filters. If two distinct keys hash to the same fingerprint, the quotient filter will not be able to differentiate them the way a hash table would. But if two keys hash to different fingerprints, quotient filter will be able to tell them apart; this is not the case with Bloom filters, where the query on a key with a unique set of $k$ hashes might generate a false positive.

Quotient filter has similar functionality to Bloom filter, but they have very different designs. Using longer fingerprints, quotient filters can reduce the false positive rate, but longer fingerprints might also blow up space.

In this section, we will see different tricks that the quotient filter employs to compactly store fingerprints. The ability to restore the full fingerprint comes in handy when we want to delete elements --- yes, quotient filters can efficiently delete.

In addition, quotient filter can resize itself, and merging two quotient filters into a larger quotient filter is a seamless fast operation. Efficient merging, resizing and deletions are all features that might make you want to consider using quotient filter instead of Bloom filter in some applications; these features come especially handy in dynamic distributed systems. Truth be told, quotient filters are a tad more complex to understand and implement than Bloom filters, but learning about them, in our opinion, is well worth your time.

In the coming sections, we will explore 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. The best way to view a quotient filter is as a game of saving a bit here or there and employing various tricks to that end. Quotient filter is not the only data structure of this sort, but some of the tricks that you learn here can be generally useful in understanding similar data structures based on compact hash tables.

---

[11] 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.

[12] 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.

### 3.7.1 Quotienting

Quotienting[13] divides a hash of each item into a *quotient* and a *remainder:* in the quotient filter, the quotient is used to index into the corresponding bucket of the hash table, and the remainder is what gets stored in the corresponding slot. For example, given the $h$-bit hash, and table size $m = 2^q$, the quotient is the value determined by the leftmost $q$ bits of the hash, and the remainder represents the remaining $r = h - q$ bits. Example in Figure 3.8 shows the fingerprint partition on a small example where $m = 32$ (so, $q = 5$) and $h = 11$:

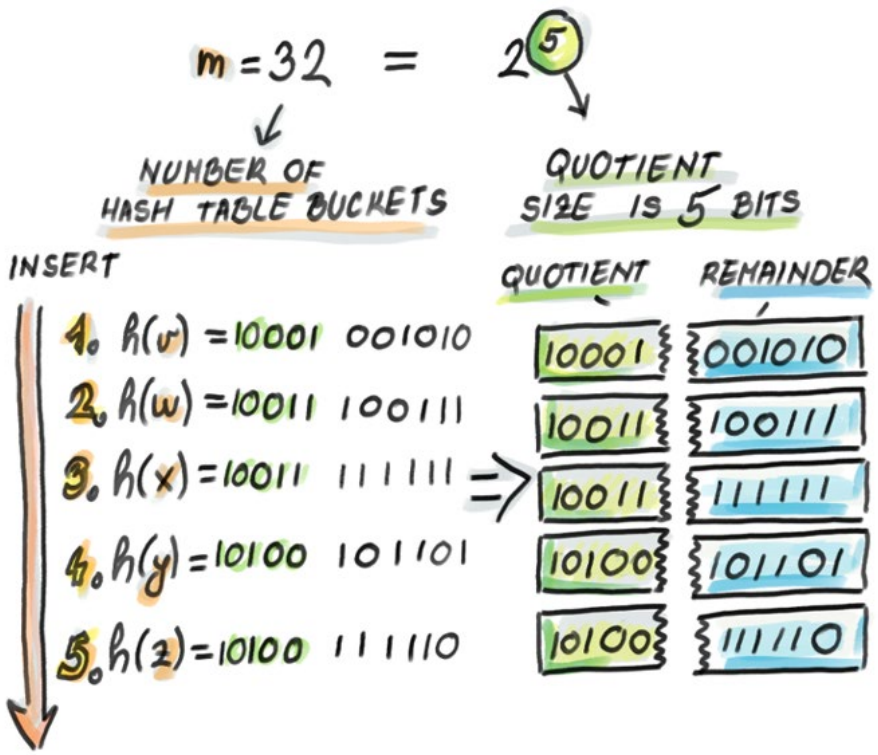



**Figure 3.8: Quotienting in a hash table. The item y has the hash 10100 101101, therefore the remainder 101101 (35) will be stored in the slot determined by quotient, at bucket 10100 (20).**

The snippet of code below shows how, once the key is hashed and the hash is stored into the variable `fingerprint`, how the insert into the quotient filter (`filter`) proceeds:

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

```
h = len(fingerprint) #A
q = log2(m) #B
r = h – q
quotient = math.floor(fingerprint / 2**r) #C
remainder = fingerprint % 2**r #D
filter[quotient] = remainder #E
```

**#A number of bits in the fingerprint**
**#B assume the size of a hash table is a power of 2**
**#C q leftmost bits of the fingerprint**
**#D r rightmost bits of the fingerprint**
**#E remainder gets stored in a bucket defined by quotient (quotient does not get stored)**

So far, so good. If no collisions occur, i.e., no two fingerprints ever have the same quotient, then every remainder occupies their own bucket $b$. It is easy to reconstruct a full fingerprint by concatenating the binary representation of the bucket number $b$ with the binary representation of the remainder stored at the bucket $b$. Even in this small example, we managed to save $q = 5$ bits per slot due to quotienting.

It is important to keep in mind that in a quotient filter, we can reconstruct the fingerprints, which helps with resizing and merging, but we cannot reconstruct the original elements. Again, we are somewhere between hash tables, that hold actual keys, and Bloom filters, that can not reconstruct which element had which hashes.

However, collisions in hash tables are quite common and quotient filter resolves collisions using a variant of linear probing. We smell trouble already, because as a consequence of linear probing, some remainders will be pushed down from their original bucket, thus losing the quotient-remainder association. To reconstruct the full fingerprint, quotient filter uses three extra metadata bits per slot. Three bits are a small price to pay for saving ~20-30 bits per slot on a quotient in larger hash tables. Below we explain how metadata bits facilitate operations in the quotient filter.

### 3.7.2 Understanding metadata bits

Before we introduce what metadata bits stand for, a bit of terminology is needed. If you get confused by terms in this section, do not worry too much, things should become clearer as we work on examples and see what runs and clusters are for, and what role each metadata bit plays when resolving collisions during an insert or a lookup.

A ***run*** is a consecutive sequence of quotient filter slots occupied with remainders, i.e., fingerprints with the same quotient (all fingerprints that collided on one particular bucket). All remainders with the same quotient are stored consecutively in the filter and in the sorted order of remainders. Due to collisions and pushing remainders down when collisions occur, a run can begin arbitrarily far from its corresponding bucket.

A ***cluster*** is a sequence of one or more runs. It is a consecutive sequence of quotient filter slots occupied by remainders, where the first remainder is stored in its originally hashed slot (we call this the *anchor* slot). The end of a cluster is denoted either by an empty slot, or by the beginning of a new cluster.

When performing a lookup on the quotient filter, we need to decode remainders along with relevant metadata bits to retrieve the full fingerprints. The decoding always begins at the

start of a containing cluster, i.e., at the anchor slot, and works downwards. The three metadata bits at each slot help decoding the cluster in the following manner:

- **bucket_occupied**: tells us whether any key so far has ever hashed to the given bucket --- it is 1 if some key hashed to the bucket, and 0 otherwise. This bit tells us what all the possible quotients are in the cluster.
- **run_continued**: tells us whether the remainder currently stored in this slot has the same quotient as the remainder right above it. In other words, this bit is 0 is the remainder is first in its run, and 1 otherwise. This bit tells us where each run in the cluster begins and ends.
- **is_shifted**: tells us whether the remainder currently stored in the slot is in its originally intended position or it has been shifted. This bit helps us locate the beginning of the cluster --- it is set to 0 only at an anchor slot, and set to 1 otherwise.

### 3.7.3 Inserting into a quotient filter: an example

Now let us work through the insertion of the elements $v$,$w$ and $x$, which you can follow along in the Figure 3.9. We begin with an initially empty quotient filter of $32 = 2^5$ slots, where each slot, as well as all three metadata bits are initially set to 0.

- Insert of $v$: $h(v) = 10001001010$. The bucket determined by the quotient $10001$ is previously unoccupied. We set the `bucket_occupied` bit to 1, and store the remainder into the slot determined by its quotient. Note that we do not need any additional action on other bits as this item is currently both a beginning of a run and of a cluster.
- Insert of $w$: $h(w) = 10011100111$. Again, we set the `bucket_occupied` bit to 1 at 10011, and store the remainder into the corresponding slot as it is available, and again do not modify any other bits.
- Insert of $x$: $h(x) = 10011111111$. The bucket 10011 is already occupied when we try to set the bucket to 1, so wherever we store the remainder, we will need to set `run_continued` of that slot to 1. The slot at the hashed bucket is taken, so wherever we store the remainder, we will need to set the `is_shifted` bit of that slot to 1 as well. Given that we are at the start of the cluster that has only one run (whose quotient is equal to the quotient of $x$), we scan downward to find the first available slot within the run, at bucket 10100. We store the remainder, and set the `run_continued` and `is_shifted` bits to 1.

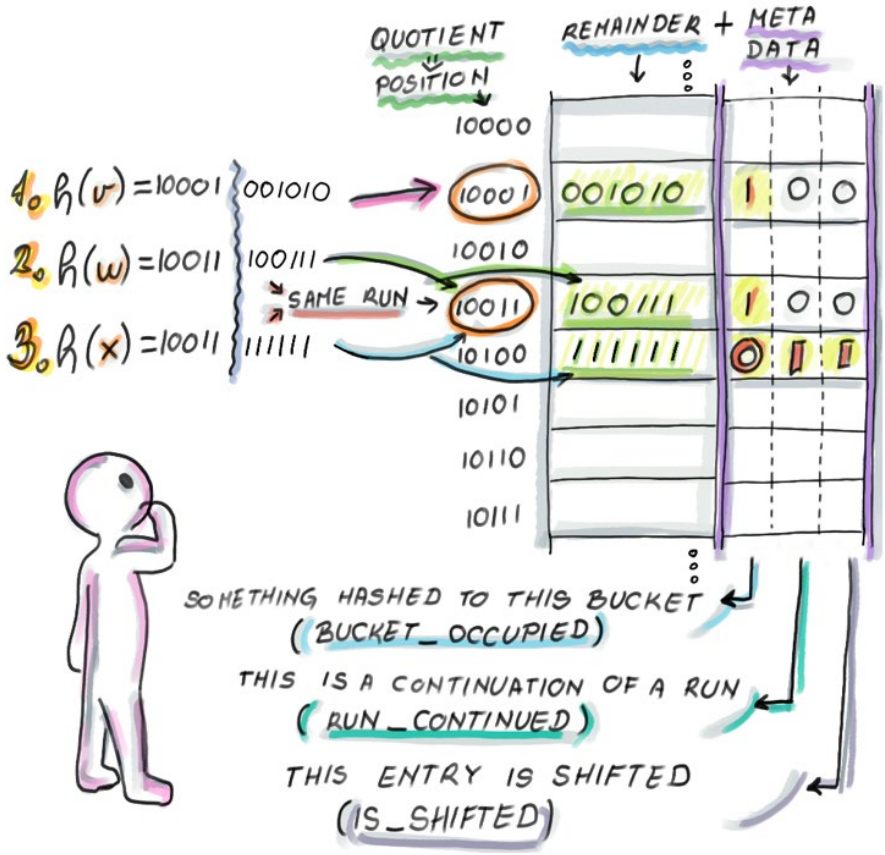



Figure 3.9: Insertion into quotient filter and metadata bits.

Currently, the quotient filter has two clusters, each of which has one run. Now we insert few more elements (follow along in the Figure 3.10):

- Insert of $y$: $h(y) = 10100101101$. The `bucket_occupied` bit at 10100 was 0 and we set it to 1 (`run_continued` at the final remainder's slot will be set to 0), and the slot at the hashed bucket is taken (`is_shifted` at the final remainder's slot will be set to 1). Starting from the beginning of the cluster, we look for the first place to store our new run. The first available slot is at bucket 10101, we store the remainder and set the metadata bits accordingly.
- Insert of $z$: $h(z) = 10100111110$. The `bucket_occupied` bit of `z`'s bucket is already set to 1 (`run_continued` at the final slot will be 1), and the original slot is taken (`is_shifted` at the final slot will be 1). Starting from the start of the cluster, we decode and find where our appropriate run is. We scan down the run to the bucket 10110 to store $z$'s remainder, and set bits accordingly.

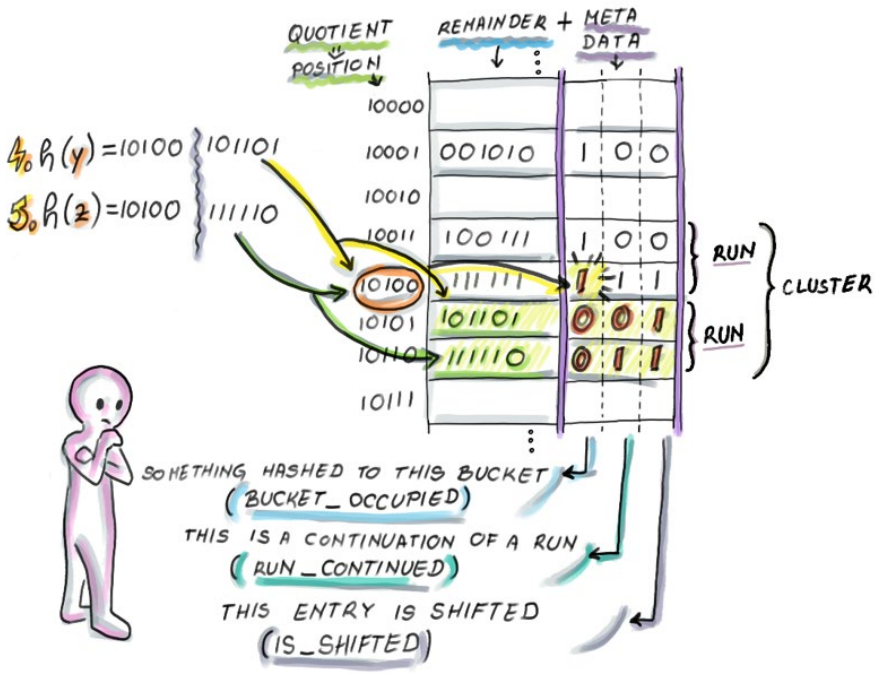



Figure 3.10: Insertion of y and z into the quotient filter.

Our insert sequence is rather simplified because the insertions arrive in the sorted order of fingerprint values. A sorted order of inserts into a quotient filter is a common scenario due to the way quotient filters are merged and resized --- similar to merging sorted lists in mergesort, but we also need to be able to handle scenarios when inserted fingerprints arrive in the arbitrary order.

When elements are inserted out of the sorted fingerprint order, then once the correct run is found for the remainder to be inserted into it, it might push down multiple items of that run and other elements in that cluster. Consider the example of inserting an element $a$, $h(a) = 10100000000$ into our resulting quotient filter from Figure 3.10. The element would belong to the second run of the second cluster that currently occupies slots determined by buckets 10101 and 10110. An element $a$ would move the entire run one slot below in order to get stored at slot 10101 because its remainder is the first in the ascending sorted order in that run, thus also triggering the changes in metadata bits.

Why do we need to have remainders sorted within a run (and runs among clusters)? The answer to that question will come when we talk about efficient resizing and merging.

Another important hash-table-related note here is that having to potentially move an entire cluster of items while inserting/deleting, and decoding a whole cluster while performing

lookup underlines the importance of small cluster sizes. The more empty space we leave, the smaller the probability of getting large clusters that insert and lookup operations need to scan through and decode. Just like with common hash tables with linear probing, quotient filters work faster when the load factor is kept at 75%-90% than when it is higher.

### 3.7.4 Python code for lookup

Now that we conceptually understand insert, let's show how lookup works using code in Python. For the purposes of understanding the underlying logic, we will for a moment set aside the mechanics of compactly storing the data structure, which involves a lot of bit-unpacking and bit-shifting. Our implementation below for the class `Slot` and the class `QuotientFilter` is "sparse" in that a metadata bit is an entire Boolean variable, so it takes up more than one bit of memory. Our Python code below is based on the pseudocode from the original paper.

```
import math

class Slot:
    def __init__(self):
        self.remainder = 0
 self.bucket_occupied = False
 self.run_continued = False
 self.is_shifted = False

class QuotientFilter:
    def __init__(self, q, r):
        self.q = q
 self.r = r
 self.size = 2**q #A
 self.filter = [Slot() for _ in range(self.size)]

    def lookup(self, fingerprint):
        quotient = math.floor(fingerprint / 2**self.r)
 remainder = fingerprint % 2**self.r
 if not self.filter[quotient].bucket_occupied:
     return False #B
        b = quotient
 while(self.filter[b].is_shifted): #C
     b = b - 1
 s = b

 while b != quotient: #D
     s = s + 1
     while self.filter[s].run_continued: #E
         s = s + 1
     b = b + 1
            while not self.filter[b].bucket_occupied: #F
         b = b + 1

 while self.filter[s].remainder != remainder: #G
     s = s + 1
     if not self.filter[s].run_continued:
         return False
 return True
```

**#A Filter size**
**#B If no element hashed to this bucket**
**#C Go up to the start of the cluster**
**#D b tracks occupied buckets and s tracks corresponding runs**
**#E go down the run and advance the bucket number**
**#F skip empty buckets**
**#G now s points to the start of the run where our element might be contained**

The lookup begins by searching for the beginning of the cluster where the item we are searching for might be contained. In other words, to find out whether an element is present or not, we need to decode a whole cluster.

After we find the beginning of a cluster that contains fingerprint's bucket, then for each occupied bucket, we locate its corresponding run, and skip through that run until encountering the bucket that equals the quotient of our fingerprint, and the start of its run (if that never happens, we return `False`.) Once the appropriate run is found, it linearly searches inside the run for the fingerprint's remainder.

Insertion code would also use a modified version of the lookup procedure that returns the position of the queried element, not a Boolean value. It would start by marking the appropriate bucket as occupied. Then it would use the lookup algorithm to find the appropriate location to insert the remainder, which might require shifting other remainders down until an empty slot is reached. Deletes work in a similar fashion, where they might need to move elements up to fill the hole created by the deletion, or sometimes deletes are implemented by placing a tombstone element in the said position, thereby eliminating the need for moving remainders around. If the deleted element was the only element in its run, then we also need to unmark the `bucket_occupied` bit. All operations in the quotient filter wrap around the table if the end of table is reached, just like in common hash tables with linear probing.

#### STORING A QUOTIENT FILTER

An important detail in the implementation of the quotient filter, as well as many other types of compact hash tables is how data is laid out in memory. Specifically, the slot size generally does not equal the unit sizes of addressable memory (e.g., bytes), so the byte boundaries and slot boundaries often do not align. For example, say we have a quotient filter with the remainder of length $r = 7$ bits, as well as 3 metadata bits (10 bits per slot in total.) Figure 3.11 shows the memory layout of the quotient filter slots.

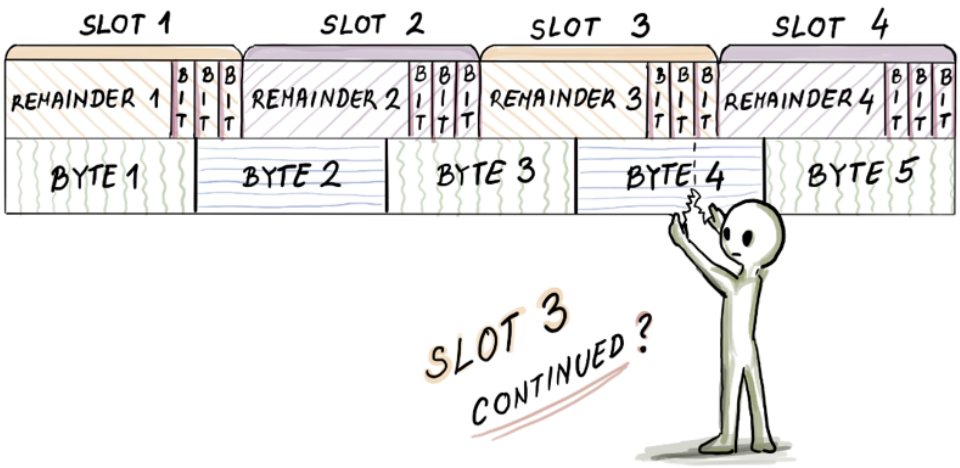



Figure 3:11: The memory layout of quotient filter slots.

For example, to read slot 3's `run_continued` bit, we need to access 5th bit of byte 4. To decode a cluster, we need to do a large number of bit-shifting and bit-unpacking operations, so the quotient filter implementations (most often written in C) pay the price of small space with extra CPU cost, which, as we will see in Section 3.8, reflects on the in-RAM insert and lookup performance for high load factors in a quotient filter. Unlike a Bloom filter insert that elegantly hops around setting bits to 1, quotient filter can be very CPU-intensive. This, however, can be a good kind of a problem, as quotient filter operations move sequentially through the table, while the Bloom filter pays the price in weak spatial locality.

### 3.7.5 Resizing and merging

If we want to double the size of the quotient filter, it is sufficient to retrieve the fingerprint, readjust the quotient and remainder size by stealing one bit from remainder and giving it to the quotient, and inserting the new fingerprint into twice-as-large quotient filter. Generally, resizing works by traversing the quotient filter in the sorted order, decoding fingerprints as we go, and inserting them in that sorted order into the new quotient filter. The fast append operation lets us zip through the quotient filter as inserting in the sorted order does not require a great deal of decoding and moving remainders around. A simple example of resizing a small quotient filter of size 4 is shown in Figure 3.12.

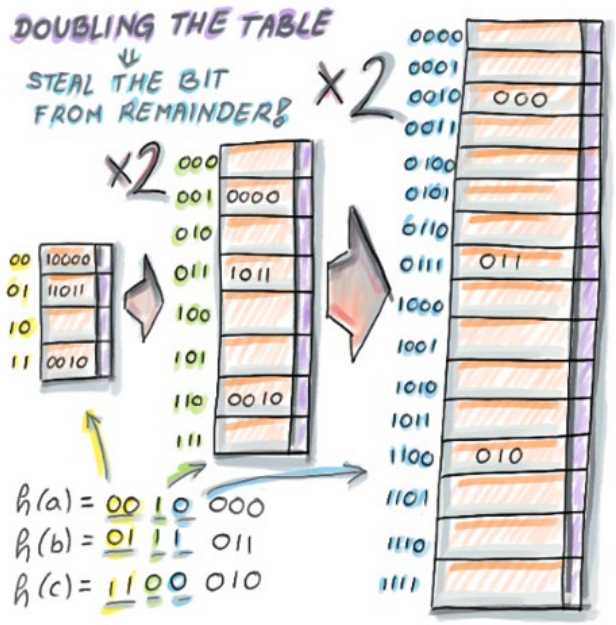



Figure 3.12: Resizing a quotient filter of size 4 that contains 3 elements. For the sake of simplicity, we assume that no collisions occurred between those 3 elements and ever remainder is stored in its original bucket. A fingerprint 110010 that hashed to bucket 11 (with remainder 0010) in the original quotient filter gets stored at 110 bucket of the second quotient filter (with remainder 0010), and in a 1100 bucket of the final quotient filter (with remainder 010.)

Similarly to how resizing is performed, we can merge two quotient filters in a fast linear-time fashion as we would do with two sorted lists in mergesort, again enabling a fast append into a larger quotient filter.

Recall that in a Bloom filter, we can not simply merge or resize as we do not preserve the original elements nor fingerprints. In order to resize a Bloom filter, we would need to save the original set of keys, and reload it into memory to build a new Bloom filter, which is infeasible in fast-moving streams and high-ingestion databases.

### 3.7.6 False positive rate and space considerations

In a quotient filter, false positives occur as a result of two distinct keys generating the same fingerprint. The analysis[14] shows that, given the table of $2^q$ slots and fingerprint length $p = q + r$, the probability of a false positive is comparable to $\frac{1}{2^r}$. The amount of space required by quotient filter hash table is $2^q(r+3)$ bits. The number of items inserted into a quotient filter

[14] 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.

is $n = \alpha 2^q$, where the load factor $\alpha$ has a significant effect on the performance of insert, lookup and delete operations.

There also exists a variant of quotient filter that only uses 2 metadata bits and uses $2^q(r+2)$ bits for storage without compromising the false positive rate, but this variant substantially complicates the decoding step, making common operations too CPU-intensive on longer clusters.

Practically speaking, due to extra space required for linear probing table, quotient filters tend to take up slightly more space than Bloom filters for common false positive rates. For extremely low false positive rates, quotient filters are more space-efficient than Bloom filters.

There are other succinct membership query data structures that are based on hash tables that we will not study in this chapter (e.g., cuckoo filter[15] based on cuckoo hashing). However, we hope that the ideas that you learn by studying Bloom filters and getting a taste of quotient filters, as well as their performance comparison in next section gives you the right lens to learn about similar data structures that are out there.

## 3.8 Comparison between Bloom filters and quotient filters

In this section, we will summarize performance differences between Bloom filters and quotient filters. Spoiler: differences in the performance are not dramatic in either direction. However, what is more interesting are the "behavioral" differences between the two data structures stemming from the way they have been designed, and that might help us understand the nature of these data structures better. Our analysis relies on the experiments previously performed in the original quotient filter paper, and our Figure 3.13 is a rough sketch of some of their findings.

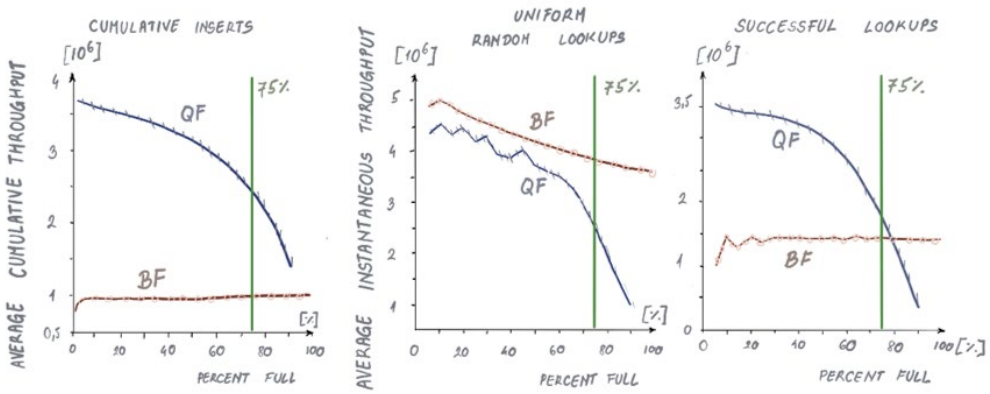



Figure 3.13: Performance comparison of Bloom filters and quotient filters on inserts, uniform random lookups, and successful lookups, respectively.

[15] B. Fan, D. G. Andersen, M. Kaminsky and M. D. Mitzenmacher, "Cuckoo Filter: Practically Better Than Bloom," in Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies, Sydney, Australia, 2014.

In all three graphs, the $x$ axis represents the fraction of the data structure fullness, and the $y$ axis represents the cumulative throughput as the data structure fills up. In this particular experiment, both data structures were given the same amount of space (2GB) and were filled up with as many elements as possible without deteriorating the false positive rate, which was set to $1/64$ for both data structures. In case of quotient filters, performance becomes very poor after 90% of occupancy, so quotient filters are only filled up until 90% full.

#### UNIFORM RANDOM INSERTS

As seen in Figure 3.13 (left), inserts are significantly faster in quotient filters than in Bloom filters. While quotient filters at 75% occupancy have the cumulative throughput of ~3 million inserts per second, Bloom filters have the cumulative throughput of ~1.5 million inserts per second. Bloom filters need $k$ random writes for each insert, while quotient filters usually need only 1 random write, and that is the main cause of the difference in the performance. For higher false positive rates, Bloom filters might require more hash functions, which can further degrade the insert performance. For Bloom filters, the insert performance is a flat line as the data structure fills up, while on the other hand, inserts slow down for quotient filter as it fills up, as it has to decode larger clusters and drops significantly after $\alpha = 0.8$.

#### UNIFORM RANDOM LOOKUPS

Uniform random lookups are slightly faster in Bloom filters than in quotient filters (Figure 3.13, center); the difference becomes more striking after data structures reach 70% of occupancy, and quotient filter starts decoding larger clusters. Generally, uniform random lookups are faster than inserts in Bloom filters. Given a large enough universe, most of uniform random lookups are unsuccessful lookups, and when optimally filled, Bloom filters reject an unsuccessful lookup after 1-2 probes on average --- reading only 1-2 bits is hard to beat performance-wise.

Recall that in our Google's WebTable query-routing example at the beginning of the chapter, unsuccessful lookup is common, so having this operation run very fast is quite favorable for Bloom filters. The uniform random lookup performance only slightly drops as the Bloom filter gets fuller, because the proportion of 1 bits increases, and with it, the number of bits the lookup has to check before it gives up. Quotient filters need to do a bit more work than Bloom filters by decoding a containing cluster, but it still amounts to one random read plus additional bit-unpacking.

#### SUCCESSFUL LOOKUPS

Successful lookups exhibit similar performance trends to inserts (Figure 3.13, right) and quotient filters again outperform Bloom filters unless for very high load factors. Bloom filter has to check $k$ random bits on a successful lookup --- its performance is independent of how full the data structure is, while the quotient filter performance degrades with higher occupancy and larger clusters.

The experimental results do not point to a clear winner between Bloom filters and quotient filters, but their other features may point to better suitability in particular settings: Bloom filter is simpler to implement and creates less burden on the CPU. Quotient filter supports

deletions, and works very well in distributed settings where efficient merging and resizing is important. Quotient filter variants adapted for SSD/disk also outperform Bloom filter variants intended for SSD/disk, due to fast sequential merging and a small number of random reads/writes in quotient filters. To learn more about SSD/disk-adapted versions of Bloom filters and quotient filters, see a related review[16].

## 3.9 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 with linear probing and are functionally equivalent to Bloom filters, with the benefit of spatial locality, ability to delete, merge and resize efficiently.
- Quotient filters are based upon the space-saving method of quotienting and extra metadata bits that allow the full fingerprint reconstruction.
- Quotient filters offer better performance than Bloom filters on uniform random inserts and successful lookups, while Bloom filters win over on uniform random (unsuccessful) lookups. The performance of quotient filters is dependent on a load factor (lower load factor is better), and the performance of Bloom filters is dependent on the number of hash functions (fewer hash functions is better).

[16] 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.

# 4

# *Frequency Estimation and Count-Min Sketch*

**This chapter covers**

- Exploring practical use cases where frequency estimates arise and how count-min sketch can help
- Understanding the majority element and the heavy hitters problem
- Understanding why heavy hitters can not be solved exactly in a constrained setup of streaming data
- Learning how count-min sketch works
- Exploring use cases of sensor app and an NLP application
- Learning about the error vs. space tradeoff in count-min sketch
- Learning how to configure the data structure
- Understanding dyadic ranges and how range queries can be solved with count-min sketch

Popularity analysis, such as producing a bestseller list on an e-commerce site, computing top-$k$ trending queries on a search engine, and reporting frequent source-destination IP address pairs on a network, is a common problem in today's data-intensive applications. Anomaly detection, i.e., monitoring changes in systems that are awake 24/7, such as sensor networks or surveillance cameras, falls under the same algorithmic umbrella as measuring popularity. Anomaly detection is often observed through a sudden spike in the value of a certain parameter, such as the temperature or location change in a sensor, an object appearance in the frame, or the number of units by which a company's stock rose or fell in a given time interval.

At their most basic, both problems translate into measuring frequency: for each unique item key, build a data structure that can tell me how many times we encountered it. This is a problem where key-value dictionaries fit like a glove, and the amount of space taken by the dictionary would be proportional, not to the total sum of frequencies encountered ($N$) but to the total number of distinct items whose frequencies we would like to measure ($n$). That number, however, might be very large. This chapter deals with alternative solutions to the element-wise frequency problem when the number of distinct items is too large.

When do we encounter a large number of distinct elements? Let's say we want to count the number of times different products were sold on an e-commerce site. If we look at the distribution of sales across products, what often happens is that there is a small number of distinct products making up the majority of sales, and a large number of products being sold only a small number of times. This sort of distribution (also known as Zipf's Law) has been observed in many different domains. What's tricky about measuring frequency with this type of distribution is that there is both a legitimate need to solve this problem due to large variations in frequency, and a scalability issue lurking just around the corner due to many low-frequency items.

Another practical situation in which scalability issues can occur is when keys are pairs of elements, for instance, source-destination IP address pairs, or word pairs in a piece of text, where $n$ can grow quadratically with respect to the total number of distinct IP addresses (or words), which itself might be quite high.

Also, it is more than just limitations on space. In this chapter, we will be interested in measuring frequency in rapid-moving streams that pose a number of difficult constraints on our choice of a data structure and an algorithm. For instance, our algorithm will be able to see each element only once (or a small constant number of times), before we discard it and move onto the next one. Solving problems such as top-$k$ queries and heavy hitters exactly is often impossible in this highly constrained setup. Thus we need to resort to approximate solutions.

We will learn about how a probabilistic succinct data structure Count-Min sketch can help us approximately measure frequency and solve related problems while achieving enormous space savings. We begin with the problem of heavy hitters, and why linear space is essential for the correct solution to the problem if we are to solve it in one pass. Then we introduce count-min sketch, its design, and use case scenarios in the context of sensors and NLP. Towards the end of the chapter we also discuss the error vs. space tradeoff in count-min sketch, and how count-min sketch can be used to answer approximate range queries.

## 4.1 Majority element

Let's start with a simple problem: 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 that element.

**Exercise 4.1**

Before moving on, try to design and implement a linear-time algorithm with constant space (other than the storage for the array) for the majority problem.

This problem can be solved using a one-pass-over-the-array algorithm[1] (also known as Boyer-More Majority Vote Algorithm) that uses only two extra variables as shown in the following Python code:

```
def majority(A): #A
    index = 0
    count = 1
    for i in range(len(A)):
        if A[i] == A[index]:
     count+=1
 else:
     count-=1
 if count == 0:
     index = i
     count = 1
    return A[index]
```

**#A If there is a majority element in A, this function returns it**

The `majority` function tracks the current frontrunner for the title of majority element and resets the frontrunner once the number of occurrences of (one or more) other elements cancel it out. Assuming the provided list has a majority element, the algorithm shown above will output it; otherwise, it might output an arbitrary element. If we are uncertain about whether the array has a majority element, we can perform one more sweep over the array to make sure that the returned value is indeed majority. Below we show how the algorithm works on two examples --- a list with a majority element and a list with an element that is almost a majority:

```
A = [4, 5, 5, 4, 6, 4, 4]
print(majority(A))

C = [3, 3, 4, 4, 4, 5]
print(majority(C))
```

The output is:

```
4
5
```

There is also a more visual way of thinking about this problem: grab an arbitrary pair of adjacent numbers in the array that are not equal to each other, throw them out and contract the hole created by removing the two elements. Continue grabbing pairs of different elements anywhere in the array until you are left with only one distinct element, potentially its multiple copies; this element is majority. The figure below illustrates the algorithm by

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

showing goats vying for their position on the bridge, with type 2 goat winning as the "majority goat".

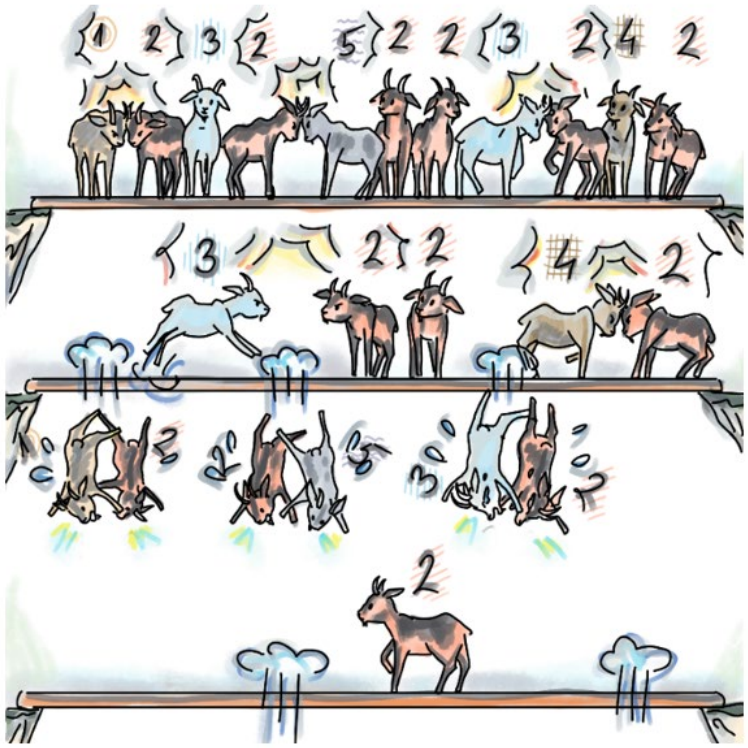


Figure 4.1: We find a majority element in an array by having different neighboring elements throw each other out. In this example, after we throw out (1,2), (2,5), (3,2), and then (3,2) and (4,2), we are left with the majority element 2.

Both algorithms illustrate that this problem can be solved simply in linear-time with a constant memory overhead. But does this approach extend to the case of general heavy hitters?

### 4.1.1 General heavy hitters

The general heavy hitters problem with a parameter $k$ asks to output all elements in the array of $N$ items that occur more than $N/k$ times (majority element is the simple instance where $k = 2$.) There can be at most $k - 1$ heavy hitters in an instance of a heavy-hitter problem, but there could also be fewer or none.

Considering that there could be many heavy hitters, the simple application of our frontrunner algorithm implies that we should maintain many heavy-hitter candidates concurrently. To

illustrate the difficulty arising from this setup, consider the following extreme case observed[2] when $k = N$: say we are witnessing a long data stream in which all elements discovered thus far have been distinct, and a single repetition of an element would term that element a heavy hitter. To identify a potential heavy hitter, we need to be saving each new incoming distinct element, as we do not know which one will have a repetition.

This toy example is a bit sneaky --- its purpose is not to illustrate the practically-occurring problem instance as much as to convince you that with larger $k$, the memory consumption and algorithmic complexity for solving heavy hitters grows if we want to solve the problem exactly, and that we need to turn to solving this problem approximately.

What will be approximated in heavy hitters? The $(\varepsilon, k)$-heavy hitters asks to report all elements that occur at least $N/k - \varepsilon N$ times, that is, all the heavy hitters, and in addition all the elements that are at most $\varepsilon N$ short from being heavy hitters for some previously set value of $\varepsilon$. In other words, the data structure that will store frequencies will overestimate the frequencies for some elements by the fraction $\varepsilon$ of the total sum of frequencies $N$.

The data structure recording approximate frequencies that we will focus in this chapter is Count-min sketch, devised by Cormode and Muthukrishnan in 2005[3]. Count-min sketch is like a young and 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. Next we explore how count-min sketch works.

## 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 false positives.)

Count-min sketch is represented as a matrix of integer counters with $d$ rows and $w$ columns `(CMS[1..d][1..w])` with all counters initially set to 0, and $d$ independent hash functions $h_1$, $h_2$,…, $h_d$. Each hash function has the range [1..$w$], and the $j^{th}$ hash function is dedicated to the $j$th row of the CMS matrix, $1 \le j \le d$.

[2] M. Charikar and N. Wein, "CS369G: Algorithmic Techniques for Big Data, Lecture 7: Heavy Hitters, Count-Min Sketch," Stanford (Lecture Notes), 2015-2016.

[3] 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.

### 4.2.1 Update

Update operation adds another instance (or $c_t$ instances) of an item to the dataset. Using $d$ hash functions, update operation computes $d$ hashes on $a_t$, and for each hash value

$h_j$ ($a_t$), $1 \leq j \leq d$, the respective 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)] += ct
```

An example of how update works is shown in the Figure 4.2 below, where we begin with an empty count-min sketch, and perform updates of elements $x$, $y$ and $z$, with quantities/frequencies 2, 1 and 3, respectively.

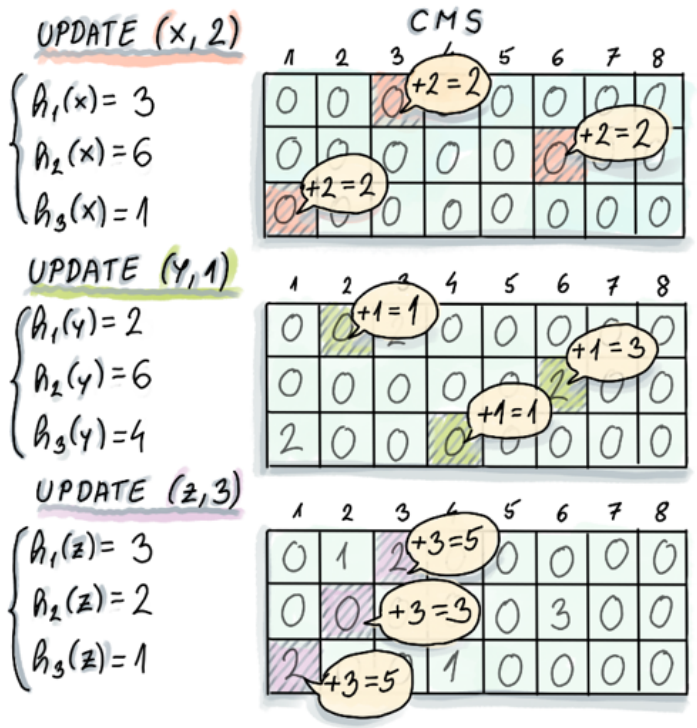



**Figure 4.2: Three update operations of $x$, $y$ and $z$ performed on an initially empty CMS of dimensions 3x8. Computed hashes indicate which columns in the respective rows of count-min sketch need to be updated.**

### 4.2.2 Estimate

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

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

```
CMS_ESTIMATE(aₜ):
    min = CMS[1][h₁(aₜ)]
    for j ← 2 to d
        if(CMS[j][hⱼ(aₜ)] < min)
            min = CMS[j][hⱼ(aₜ)]
    return min
```

An example of how estimate works is shown in Figure 4.3 below, where the estimate of element $y$ returns the correct answer, whereas the estimate of $x$ returns an overestimate. As we can see from the example, count-min sketch can overestimate the actual frequency due to hashes of different items colliding and contributing to counts of other elements, however, the overestimate only happens if there was a collision in each row.

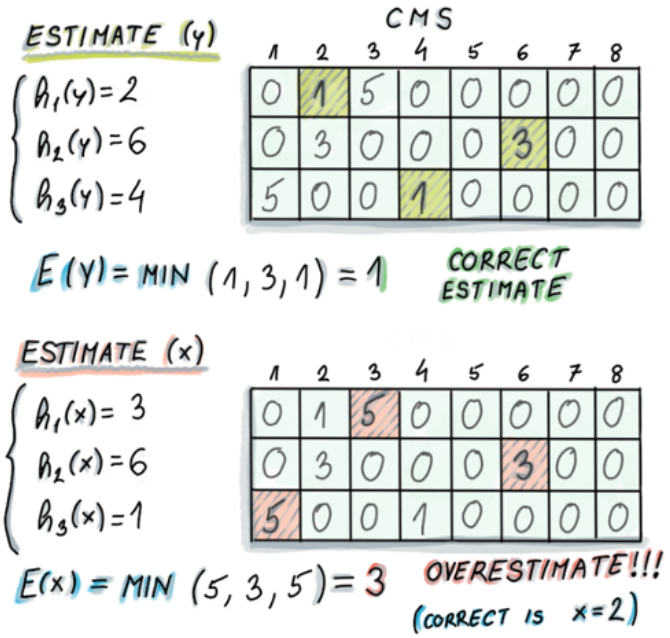



**Figure 4.3: Example of estimate operations on the count-min sketch from Figure 4.2. In the case of element y, whose true frequency is 1, count-min sketch reports the correct answer of 1 (the minimum of 1, 3 and 1). However, in the case of the element x, whose true frequency is 2, count-min sketch reports 3 (the minimum of 5, 3 and 5). Refer to the Figure 4.2 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.**

## 4.3 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

The quality of sleep has for a long time been linked to outcomes in an individual's mental and physical health. However, only recently, with the wide access to new technologies and ability to process enormous datasets, we have been able to capture very detailed sleep-related data for a large number of individuals. Smart beds, for example, that come equipped with hundreds of sensors capable of recording different parameters during sleep such as movement, pressure, temperature and so on, can help us gain new insights into people's sleep patterns. Based on sensor data, different bed components can adapt and modify real-time --- parts of the bed can be pulled up, warmed up, cooled down, etc.

Consider a smart-bed company that collects data from its users and stores it at one central database. There are millions of users and sensors that send out data every second, hence the amount of data is quickly becoming too large to process and analyze in a straightforward manner. Let's assume that one smart bed features 100 sensors, and that there are $10^8$ customers using this type of smart bed --- then our hypothetical company collects a total of $10^8$ (customers) $*$ 3,600 (seconds per hour) $*$ 24 (hours per day) $*$ 100 (sensors) = 8.6 $*$ $10^{14}$ tuples of data daily, resulting in terabytes of storage on a daily basis. Our specific example is hypothetical, but the size of the collected data and the related problem we study is not.

With every purchase of a smart bed comes the SleepQuality mobile app that allows the bed users to monitor their sleep over time. One of the new app features monitors restlessness in sleepers, and notifies the most restless sleepers that their sleeping patterns are out of whack in comparison to the rest of the smart sleepers. To implement this feature, the app takes into account different sensor readings, and collects its findings into one point on the quality-of-sleep scale. Due to the sheer amount of data coming in, the company engineers decided to try out count-min sketch to store the quantities received from users' sensors.

As shown in Figure 4.4, data arrives at a frequent rate, and at each timestep, the `(user-id, amount)` pair updates the count-min sketch by a given amount. In our simplified example, the keys in count-min sketch correspond to individual users; for a more refined analysis of how particular sensor readings change, the key should instead be a `(user-id,sensor-id)` pair.

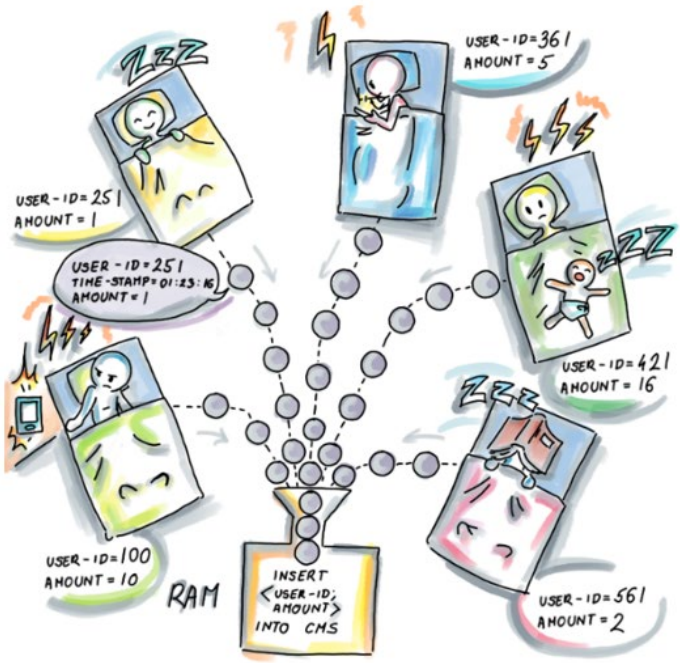



**Figure 4.4: All sleep data is sent to a central archive, but before that, input into count-min sketch residing in RAM for later analysis. The (user-id, amount) pair is used as the input for the data structure, where the frequency of user-id increases by amount. The key that is hashed is user-id.**

Having updated count-min sketch in such a manner, we will be able to produce the approximate frequency estimate for any queried user. But in order to maintain the list of top-$k$ restless sleepers, we will have to do a little bit more than just maintaining the count-min sketch. Remember that the count-min sketch is just a matrix of counters and it does not save any information on different user-ids or the ordering of relevant frequencies.

**Exercise 4.2**

**Before moving onto the solution, think what could be the right data structure to use alongside count-min sketch to store the top-$k$ restless sleepers at each point of time, using only $O(k)$ extra space.**

One solution uses a min-heap, as shown in the Figure 4.5 below. Min-heap maintains the current $k$ 'winners' of the restlessness contest, with possibilities of update each time new update to the count-min sketch is received. Namely, when a new element arrives to update count-min sketch, after the update, we perform the estimate operation on this particular item. If the reported frequency is higher than the minimum item in the min-heap (easily

accessible in $O(1)$), then the minimum of the heap is deleted and the new item is inserted. Also note that each time an update occurs for an element already in the heap, the updated frequency count needs to be reflected in the element's position in the heap.

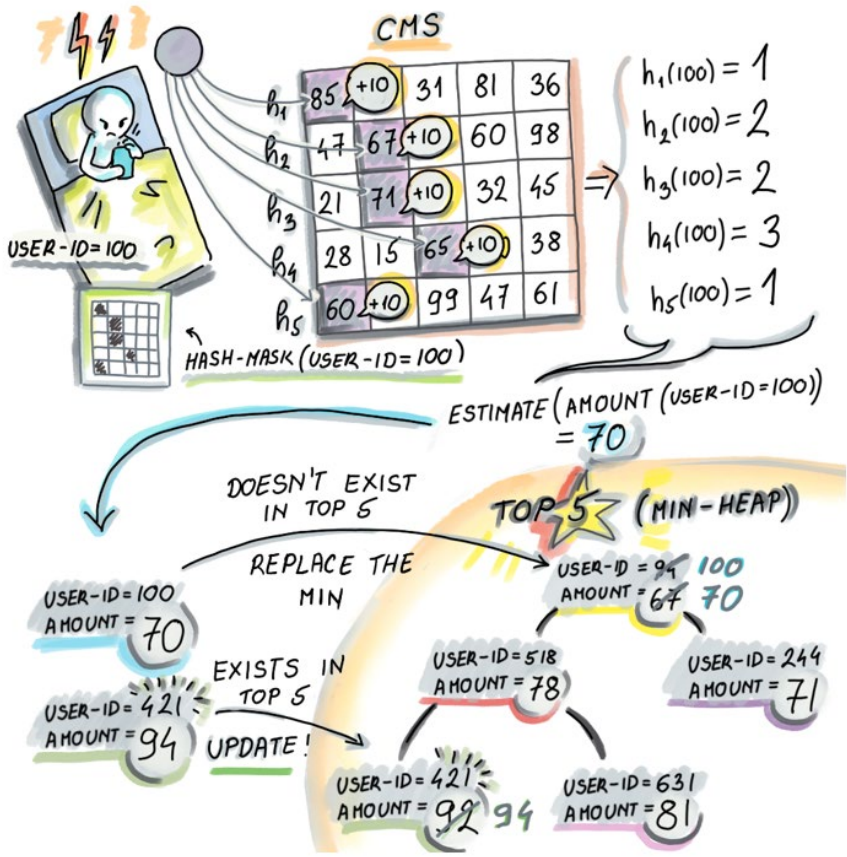



**Figure 4.5: Using min-heap and count-min sketch together to find top-k restless sleepers. Every time the count-min sketch is updated with a (user-id, amount) pair, like with (100, 10) in this example, in order to maintain a correct list of top-k restless sleepers, we do an estimate on frequency of 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.**

In this example, we showed that count-min sketch can be used instead of a typical key-value dictionary to preserve information on frequencies for the SleepQuality app, thereby saving a lot of space (we have not yet analyzed the space requirements, though). At the same time, we used a min-heap of size $O(k)$ to store the information on top-$k$ restless sleepers. Min-

heap maintains up-to-date estimates at each point in time, so whenever we wish to send the notification to such users, we have the information at our disposal.

### 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 the linguist John R. Firth has 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', 'river, etc. As a context for a given word, we choose the window of size $k$ (e.g., $k$ = 3) that includes $k$ words before and $k$ words after the given word in the text, or less, if we are crossing the boundary of a sentence.

One way to measure distributional similarity for a given word-context pair is using *pointwise mutual information (PMI)*[4]. 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. It's a fancy way of saying that PMI measures how likely $A$ and $B$ are to co-occur in our corpus in comparison to how often they would co-occur if they were independent. The higher the PMI, the more similar the words. 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 type of matrix shown in Figure 4.6 that contains all word-context pair frequencies:

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

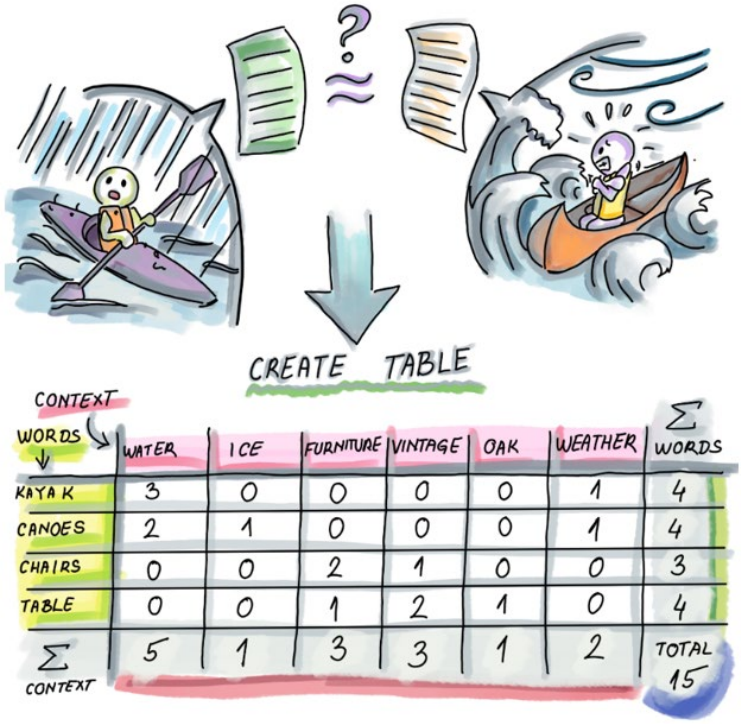



**Figure 4.6: To preprocess the text corpus for computing PMI, one way is to create a matrix M where the entry M[A][B] contains the number of times the word A appears in the context B. So for example, 'kayak' appears 3 times in the context of 'water' and 0 times in the context of 'furniture'. We also 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[5], 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-context pair tokens, and 215 million unique word-context pairs, and just storing those pairs with their counts takes 4.6GB. 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 with their counts in their own hash table (last column of the matrix), and the contexts with their counts in their

[5] 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.

own hash table (the last row of the matrix). The transformation can be seen in the Figure 4.7 below:

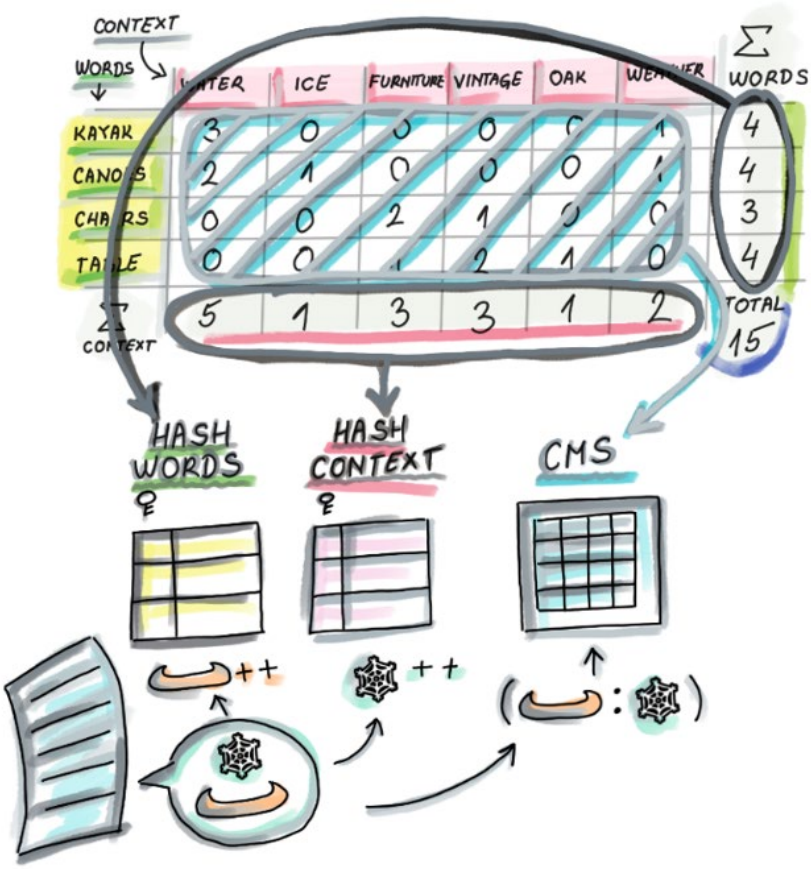



**Figure 4.7: The transformation of the matrix from Figure 4.6 to save space: the word-context pairs stored in the main body of the matrix are replaced by a count-min sketch that stores frequencies of word-context pairs. Because the number of distinct words (and contexts) is not that large, we can store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

Using count-min sketch in this particular example, the space savings achieved were over a factor of 100. 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 boon for the streaming datasets. We could produce top-$k$ PMIs with an additional sweep of the data.

You might be wondering how can we configure count-min sketch, ie, how do we set its dimensions? And what is the relationship between frequency overestimate and the size of count-min sketch? Count-min sketch has two error parameters $\varepsilon$ and $\delta$, and their values are used to determine the dimensions of the sketch. In the next section, we will delve into more detail about the relationship between the errors and space requirements.

## 4.4 Error vs. space in count-min sketch

Count-min sketch exhibits two types of errors: $\varepsilon$ (epsilon) that regulates the band of the overestimate, and $\delta$ (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 $\varepsilon$ 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 $f_{est}$:

$$fx \leq fest \leq fx + \varepsilon N$$

with probability at least $1 - \delta$. Usually $\delta$ 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. There is a small probability $\delta$ that the overestimate in CMS can be unbounded.

Just like with Bloom filter, we can tune CMS to be more accurate but that will cost us space. Whatever are the ($\varepsilon$, $\delta$) 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 be $w=e/\varepsilon$ and $d = \ln(1/\delta)$. Hence the space required by count-min sketch, expressed in the number of integer counters will be (Formula 1):

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

Note that CMS tends to be really small even when used on large datasets. Count-min sketch is often hailed for its space requirements --- that they do not depend on the dataset size --- but this is only true if you desire the error to be a fixed percentage of the dataset size. For example, keeping the allowed band of error fixed at 0.3% of $N$ will not require increasing the size of count-min sketch even if we double the value of $N$, but the actual absolute overestimate band will double. One could argue that with twice as large $N$, the application should be able to afford twice as large overestimate error.

However, what leaves us wanting when it comes to count-min sketch error properties is that the overestimate error is only sensitive to the total sum of frequencies $N$, not to the individual element frequency. Therefore, the band of error can wildly vary if we observe it with respect to the element's individual frequency: if the maximum overestimate is $\varepsilon N = 200$,

then we can equally expect that to be the overestimate for an element with frequency 10,000 and for an item whose frequency is 10. In the latter case, the estimate can overshoot the truth by 20-fold of the true frequency.

## 4.5 A simple implementation of count-min sketch

Now we are ready to see a bare implementation of count-min sketch. As before with Bloom filters, we use the `mmh3` MurmurHash wrapper for $d$ hash functions.

```
import numpy as np
import mmh3
from math import log, e, ceil

class CountMinSketch:
    def __init__(self, eps, delta):
        self.eps = eps
 self.delta = delta
 self.w = int(ceil(e/eps)) #A
 self.d = int(ceil(log(1. / delta))) #B
 self.sketch = np.zeros((self.d, self.w))

    def update(self, item, freq=1):
        for i in range(self.d):
     index = mmh3.hash(item, i) % self.w
     self.sketch[i][index] += freq

    def estimate(self, item):
        return min(self.sketch[i][mmh3.hash(item, i) % self.w] for i in range(self.d))
```

**#A Setting width**
**#B Setting depth**

Try the code below that shows the usage of the `CountMinSketch` class. Play with the updates and see how estimates change.

```
    cms = CountMinSketch(0.0001, 0.01)
    for i in range(100000):
        cms.update(f'{i}', 1)
    print(cms.estimate('0'))
```

In the next section (Section 4.5.1), we give few exercises to test the understanding of configuring count-min sketch. Section 4.5.2 discusses the intuition behind deriving error rates in count-min sketch and is more theoretical in nature. As such, it is primarily intended for readers with an interest in mathematical underpinnings of the data structure, and otherwise can be skipped.

### 4.5.1 Exercises

The following few exercises are intended to check your understanding of count-min sketch, how it is configured, and how its shape and size affect the error rate.

**Exercise 4.3**

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

**Exercise 4.4.**

Calculate the space requirements for the count-min sketch from Exercise 3.

**Exercise 4.5**

Consider what happens with the size (and more specifically, the shape) of count-min sketch if we desire a fixed absolute error ($\varepsilon N$) while $N$ increases. For example, say we want to keep the overestimate at 100 or less like in the Exercise 3, but for a twice as big $N$ as that of the Exercise 3.

**Exercise 4.6**

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)? What is the practical constraint limiting the depth of the count-min sketch to low values (<30)?

**Exercise 4.7**

How would you solve the problem of approximate $k$-heavy hitters mentioned in the beginning of the chapter with the help of count-min sketch? Specifically, how would you set $\varepsilon$ to facilitate solving this problem? Recall that in approximate $k$-heavy hitters, we would like to report all heavy hitters and also potentially those that are $\varepsilon N$ short of being heavy hitters.

### 4.5.2 Intuition behind the formula: math bit

As we have observed in our simple count-min sketch implementation, to achieve the overestimate of at most $\varepsilon N$ with probability at least $1-\delta$, the width of the count-min sketch $w$ should be set to $e/\varepsilon$, and the depth of the count-min sketch $d$ should be set to $\ln(1/\delta)$. Why is $w$ related to $\varepsilon$, and $d$ is related to $\delta$?

To understand why, let us consider the process of performing updates to count-min sketch, and specifically let us focus our attention on the first row. By the time we have performed all updates to count-min sketch, the sum of counters in the first row (and any single row) will be equal to $N$. Assuming that hash functions distribute updates uniformly randomly across cells, then the random variable $X$ describing the value stored at any one fixed cell $C$ in the

first row, after all updates have been performed, has average (or expected value) $E[X] = N/w$.

This also means that, when we perform an estimate for a particular item whose counter in the first row is to be found at cell $C$, other elements could contribute to its counter by no more than $N/w$ on average. To obtain the average overestimate in one row to be no more than $\varepsilon N$, we can set $w = 1/\varepsilon$. Clearly, the amount of overestimate is related to the width of the data structure.

Now, $E[X]$ tells us about the average behavior, but $X$ can significantly deviate from its expectation: in some cells, values can be much higher than in others. We can somewhat mildly bound an overestimate in one row using Markov's inequality, that tells us that if $X$ is a nonnegative random variable, and $c > 1$, then

$$\Pr(X \geq c * E[X]) \leq \frac{1}{c}$$

Applying Markov's inequality to our case, we get the following:

$$\Pr\left(X \geq \frac{eN}{w}\right) \leq \frac{1}{e}$$

In other words, the probability of a particular cell in the first row having the value of $eN/w$ or above is no larger than $1/e$. But this is not good enough: to bound the probability of overestimates higher than $\varepsilon N$, we consider all $d$ rows. Recall that to report an overestimate of $q$ in count-min sketch, the corresponding cells in *each* row all need to have the overestimate of at least $q$. If we apply the probability arising from Markov's inequality across all levels (note that outcomes of hash functions to different levels are mutually independent), we get that

$$\Pr\left(\text{the overestimate in each row is at least } \frac{eN}{w}\right) \leq \left(\frac{1}{e}\right)^d$$

By setting $w = e/\varepsilon$ and $d = \ln(1/\delta)$, we get that the probability that overestimate is more than $\varepsilon N$ is at most $\delta$.

## 4.6 Range queries with count-min sketch

As the final application of count-min sketch in this chapter, here we discuss how to report frequency estimates for ranges as oppose to single points. Range reporting has huge importance in databases, where the queries are often posed to reveal properties of groups and categories, rather than single data points. Queries such as: Give me all employees that have worked for the company between $a$ and $b$ years, or who have salaries between $x$ and $y$ naturally translate into range queries. Time series are another example of ranges, for example: How many books were sold on Amazon.com between December 20th and January 10th of this year?

Balanced binary search trees are 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 mere cost of outputting the range query results; this is in contrast to hash tables that scatter data all over the table and querying for a range might require 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 sketch.

True enough, the straightforward employment of count-min sketch to answer frequency estimates on ranges --- by turning the range query for the range $[x, y]$ into $y - x + 1$ point queries for each point along the query interval --- does not give desired results. Other than the query time growing linearly with the size of the range, the error also increases linearly with the size of the range, so instead of promising the overestimate of at most $\varepsilon N$ with probability at least $1 - \delta$, we can only promise at most $(y - x + 1)\varepsilon N$, which, for large ranges, will deem the data structure futile. For example, if we built a count-min sketch with maximum overestimate $\varepsilon N = 7$, a range query of interval size 10,000 could produce an overestimate of up to 70,000.

### 4.6.1 Dyadic intervals

To avoid the linearly growing error, we need to find a way to decompose an arbitrary range into a small number of sub-ranges. This way, we can obtain tighter frequency estimates, by summing up frequency estimates of the smaller ranges without accumulating substantial error[6,7].

The main idea is to divide a range into a small number of so-called *dyadic ranges*. Given a complete universe interval as $U = [1, n]$, we define a collection of dyadic ranges at $\log_2 n + 1$ different levels: dyadic ranges of level $i$, $0 \leq i \leq \log_2 n$, are of length $2^i$ and can be expressed as $[j2^i + 1, (j + 1)2^i]$, where $0 \leq j \leq n/2^i - 1$ (see Figure 4.8 for a set of dyadic ranges for universe interval $U = [1,16]$.)

The interesting property of dyadic ranges is that any arbitrary range can be decomposed into at most $2 \log U$ dyadic ranges. Later in this section, we will show the Python code that can decompose an arbitrary range into a set of dyadic ranges, but first as an example, consider our small universe from Figure 4.8, and see few examples of separating ranges into a smallest set of dyadic ranges:

- Range $[5,14]$ can be separated into 3 dyadic ranges: $[5,8], [9,12], [13,14]$
- Range $[2,16]$ can be separated into 4 dyadic ranges: $[2,2], [3,4], [5,8], [9,16]$
- Range $[9,13]$ can be separated into 2 dyadic ranges: $[9,12], [13,13]$

[6] 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.
[7] M. Charikar and N. Wein, "CS369G: Algorithmic Techniques for Big Data, Lecture 7: Heavy Hitters, Count-Min Sketch," Stanford (Lecture Notes), 2015-2016.

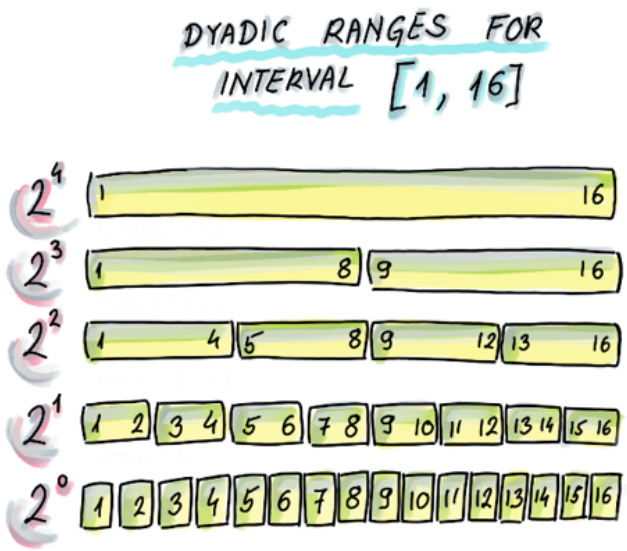



**Figure 4.8 Dyadic ranges for the interval [1, 16]. Dyadic ranges of level 0 are the bottom-most with ranges of size 1, then the ranges of level 1 are the level above with the ranges of size 2, and in general dyadic ranges at level $i$ are of size $2^i$. Dyadic ranges across different levels are mutually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

In order to report the range frequency as the sum of frequencies of dyadic ranges, as updates take place, we need to maintain the frequency information for each dyadic range. To that end, we can use one count-min sketch to serve all updates for dyadic ranges of one level (dyadic ranges of the same size) for a total of $O(\log n)$ count-min sketches. Below, we describe this scheme in more detail.

### 4.6.2 UPDATE phase

Considering that now our unit elements are dyadic ranges, we need to convert an update of a single element arriving to our system to an update of each dyadic range the element is contained in. So for example, when updating the frequency of element 5 in our example universe [1,16], we will update the frequency of the following dyadic ranges: [5], [5,6], [5,8], [1,8] and [1,16]. Ranges can be hashed just like regular elements, so there is no obstacle towards a dyadic range of the format $[l,r]$ being treated as one element.

We accomplish this using $O(\log n)$ count-min sketches, by building one count-min sketch for each level of dyadic range --- the elements to be updated/estimated in the count-min sketch on the level $i$ will be the dyadic ranges of that level. Given this scheme, Figure 4.9 shows how update of a new element takes place --- a new element arriving will be updated in each count-min sketch, by updating its containing range in the respective CMS.

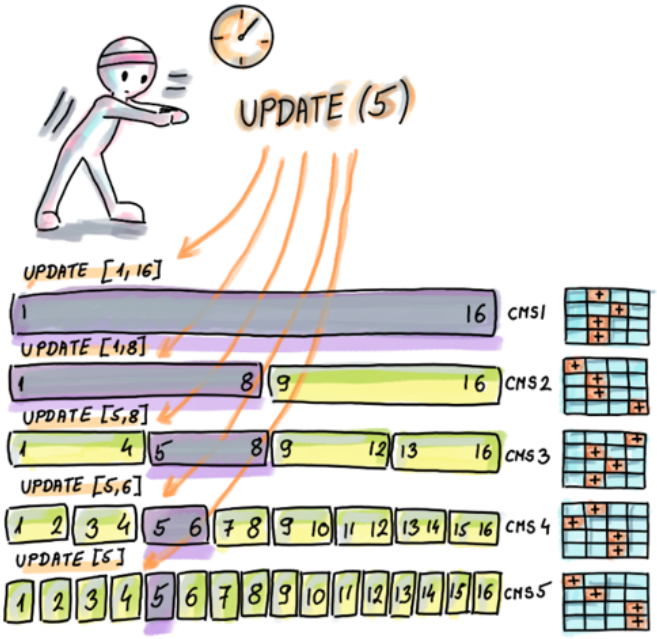



Figure 4.9: An update of one element is transformed into one update per level. For example, if we update 5, we effectively update [1,16] in CMS1, [1,8] in CMS2, [5,8] in CMS3, [5,6] in CMS4, and [5] in CMS5. Instead of updating an element, we are updating a corresponding range to which the element belongs,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

### 4.6.3 ESTIMATE phase

Now we are ready to perform estimate on a particular range using dyadic ranges. Namely, first we divide the query range into its own set of dyadic ranges. For each dyadic range, we perform estimate in the CMS that resides on its level (there can be at most two dyadic ranges on the same level per query.) The final result comes from summing up all the estimates. Figure 4.10 shows how we can do the range estimate for [3,13], whose frequency estimate we obtain by estimating the following dyadic ranges in the respective CMSs and summing them up: [3,4], [5,8],[9,12] and [13].

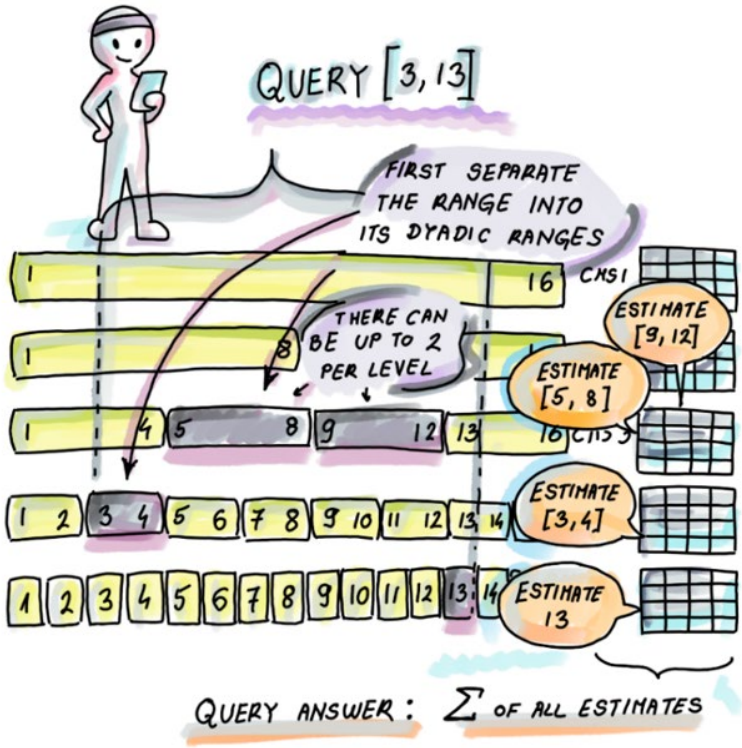



**Figure 4.10: In this example, the query range** [3, 13] **is separated into** [3,4]∪[5,8]∪[9,12]∪[13], **and we will obtain the frequency estimate for** [3,13] **by obtaining the frequency estimates for the mentioned ranges and summing them up.**

It helps to know that every range can be partitioned into at most 2log$n$ dyadic ranges (at most 2 per level). Both for the update and for the estimate, the runtime is logarithmic and also the error grows only logarithmically. 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 logarithms cancel out.

### 4.6.4 Computing dyadic intervals

The Python code shown below gives a decomposition of $I$ into dyadic intervals where we are given the large universe $U$, and a range $I \subseteq U$. First we build a full binary search tree based on the universe interval, similar to the depiction from Figure 4.8, whose each level corresponds to a level of dyadic ranges, and each node corresponds to a unique dyadic range. For instance, the root node represents the range $[1,n]$, its left child represents the range $[1,n/2]$, its right child represents the range $[n/2+1,n]$, etc. The leaves represent the ranges of size 1, and there are $n$ of them. First we construct such a tree from the universe interval:

```
from collections import deque

class Node: #A
    def __init__(self, lower, upper):
        self.data = (lower, upper)
 self.left = None
 self.right = None
 self.marked = False

def intervalToBST(left, right): #B
    if left == right:
        root = Node(left, right)
 return root
    if abs(right - left) >= 1:
        root = Node(left, right)
 mid = int((left + right) / 2)
 root.left = intervalToBST(left, mid)
 root.right = intervalToBST(mid + 1, right)
 return root
```

**#A Each node represents a dyadic range**
**#B Transforming the interval [left, right] into a binary search tree**

Given a particular range, now we compute a set of its dyadic ranges using the binary search tree we constructed. We also use `marked` attribute at each node. The nodes that end up having marked attribute `True` will be the nodes that represent the dyadic sub-ranges of the query range. The algorithm works first by marking each leaf that is a subrange of the interval $I$. Then it works in the level by level fashion going upward the tree, and if a node has both children marked, we mark that node and unmark the children. The algorithm stops after we have processed the root node.

Consider a simple interval $I = [1,5]$ in a universe interval $U = [1,16]$. On the bottom level of the tree, we mark the nodes representing following intervals: $[1,1]$, $[2,2]$, $[3,3]$,$[4,4]$ and $[5,5]$. Then we go one level up, and find that for node $[1,2]$, both of its children $[1,1]$ and $[2,2]$ are marked, so we mark $[1,2]$ (and unmark $[1,1]$ and $[2,2]$). Similarly, we mark $[3,4]$ because $[3,3]$ and $[4,4]$ are marked, and unmark $[3,3]$ and $[4,4]$. On the third level from the bottom, we mark $[1,4]$ because $[1,2]$ and $[3,4]$ are marked, and unmark $[1,2]$ and $[3,4]$. We also process nodes from all other levels all the way to the root, but we do not encounter any more nodes with both children being marked as `True`. Therefore, two nodes are left that we have marked, and those are nodes corresponding to the sub-ranges $[1,4]$ and $[5,5]$, and we report them as our dyadic ranges. This functionality is illustrated in the code below:

```
def markNodes(root, lower, upper):
    if root is None: #A
        return
    queue = [root]
    stack = deque()
    while(len(queue) > 0):
        stack.append(queue[0])
 node = queue.pop(0)
 if node.left is not None:
     queue.append(node.left)
 if node.right is not None:
     queue.append(node.right)

    while(len(stack) > 0): #B
        i = stack.pop()
 if i.data[0] >= lower and i.data[1] <= upper and
    [CA]i.left is None and i.right is None: #C
     i.marked = True

 if i.left is not None and i.right is not None:
     if i.left.marked and i.right.marked: #D
         i.left.marked = False
         i.right.marked = False
         i.marked = True

def inorderMarked(root): #E
    if root is None:
        return
    inorderMarked(root.left)
    if root.marked:
        print(root.data)
    inorderMarked(root.right)
```

**#A First traverse the nodes in the level-by-level order (BFS traversal)**
**#B The stack stores nodes in the level-by-level order starting from the leaves**
**#C Each leaf inside the interval is marked**
**#D Mark internal nodes whose two children were both marked, and unmark the children**
**#E Print dyadic ranges**

Here is how this implementation works on an example of universe interval $U = [1,16]$ and interval $I = [3,13]$:

```
k = 4
root = intervalToBST(1, 2**k)
markNodes(root, 3, 13)
inorderMarked(root)
```

The output dyadic intervals are:

```
(3, 4)
(5, 8)
(9, 12)
(13, 13)
```

The time to complete the algorithm in the worst case is the time asymptotically required by the breadth-first search algorithm on the universe tree, hence $O(n)$.

## 4.7 Summary

- Frequency estimation problems arise very commonly in the analysis of big data, especially in sets that have 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 are allowed only one pass over the data and sublinear space.
- Count-min sketch is well suited to solve the approximate heavy hitters problem and many other problems in sensor and NLP domains.
- Count-Min sketch is very space-efficient and it has two error parameters, $\varepsilon$ (controlling band of overestimate) and $\delta$ (controlling the failure probability) that are tunable and determine the dimensions of the count-min sketch. 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.
- It is possible to do fairly accurate frequency estimates for range queries using count-min sketch by decomposing a range into a set of dyadic ranges, and using $O(\log n)$ count-min sketches.

# 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 in applications involving databases, network traffic, etc. However, since the expansion of the 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.

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 DISTINCT`s and `GROUP BY`s 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.

Even when we only need to know how many distinct items the multiset has, without having to list the distinct items themselves, the complexity still remains. 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 $\Omega(n \log_2 n)$.[1]

To address the scalability issues, the newer editions of database management systems and warehouses turn to cardinality estimates: SQL Server 2019 has the `APPROX_COUNT_DISTINCT` operation[2] 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[3]. Running underneath these estimators is the algorithm called HyperLogLog originally invented by Flajolet et al.[4], that offers amazing space savings (think KBs) while processing trillion-sized datasets, and keeps the error fairly low --- on the order of $O\left(\frac{1}{\sqrt{m}}\right)$ where $m$ denotes the number of 5- or 6-bit-wide memory locations. One common choice for $m$ is $2^{14}$.

In this book, we have seen so far a number of examples of saving space in exchange for giving up some accuracy, 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

[1] Skiena, S. S. (2008). The Algorithm Design Manual, Second Edition. Springer.
[2] https://docs.microsoft.com/en-us/sql/t-sql/functions/approx-count-distinct-transact-sql?view=sql-server-ver15
[3] https://cloud.google.com/bigquery/docs/reference/legacy-sql#countdistinct
[4] 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.

hashes, and the more recent Google's reincarnation called HyperLogLog++[5] and the implementation by Redis[6] 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, however, they mimick the randomness well enough for our purposes (i.e., they *look* random).

Given a multiset $\mathrm{M} = \{a_1, a_2, \ldots, a_n\}$ with $n$ elements and $k$ distinct elements (we do not know $k$), using a hash function $h: U \to \{0,1\}^L$, we produce a hashed set $h(\mathrm{M}) = \{h_1, h_2, \ldots, h_n\}$ where $h_i = h(a_i)$ with hash length $L = |h_i|$. For a large enough $L$ (e.g., $L = 64$), each distinct 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.

We decided it would be best to demonstrate how HyperLogLog works by gradually building from the simplest algorithms, identifying their flaws and getting to more and more sophisticated algorithms. To aid in understanding, we obviate some of the technical details by showing Python-like pseudocode as oppose to the code itself.

In other words, we will try to put ourselves in the mindset of its inventors and start from something very simple and gradually improve it. This way, our hope is that you do not only learn about the final product but about the iterative process of algorithm design and little improvements at every stage. At some point, the subsections 5.2.1-5.2.4 might start feeling a bit dense from the mathematical point of view. But do not worry: Section 5.4 contains an experiment that tests the three versions of the algorithm that lead to HyperLogLog, and that should help you in understanding the ideas behind the algorithms and the need for respective improvements.

### 5.2.1 The first cut — probabilistic counting

The roughest estimate called probabilistic counting[7] observes the bit patterns in the hash by computing $\rho_i$ for each hash $h_i$ such that:

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

That is, $\rho_i$ 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 $\rho_i$ 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:

---

[5] 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

[6] http://antirez.com/news/75

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

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

Here is the idea of probabilistic counting expressed in Python-like pseudocode:

```
p_max = 0
for a in M #A
    h = hash(a)
    p = num_trailing_zeros(h) + 1
    if(p > p_max)
        p_max = p
return 2**p_max
```

**#A M is a multiset whose cardinality we wish to measure**

**Example 5.1**

**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:**

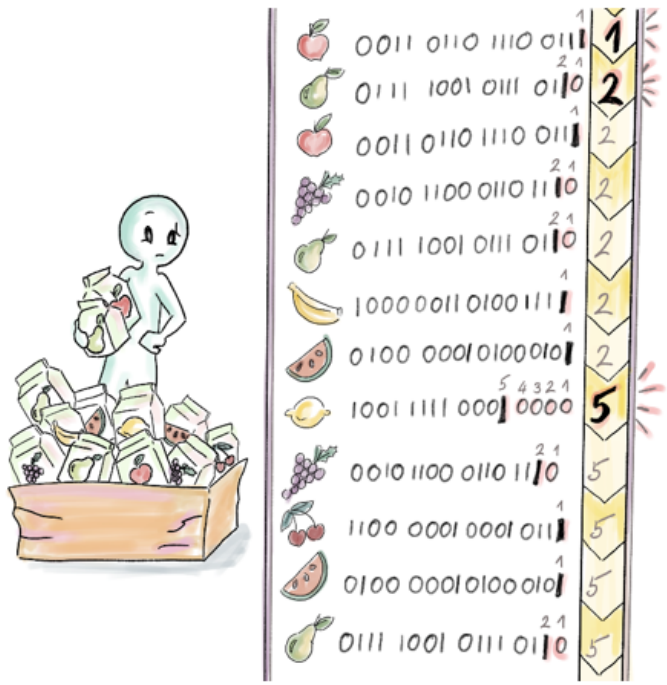


**Figure 5.1: A dataset of 12 items is hashed into 16-bit hashes. As we scan the dataset, we keep the running**

**maximum of the $\rho_i$. 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. Let's see why that is, but before we do, please keep in mind that we are still far off from the truth with this estimate, so do not get married to this method, and similarly, do not expect the mathematical explanation below to be the written in stone kind of truth, we are talking approximately.

(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 $\rho_i = \rho_{max}$ 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, the ground truth is far from average. Consider a dataset with two data points, 0 and 100, our average is 50, which tells little about the actual values in the set. Similar things happen with random variables, where deviations from this average will occur, and even a small deviation can significantly affect the estimate, considering that $\rho_{max}$ 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 ($\pm\frac{E-k}{k}$); 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 $\rho_{max}$, we will have $m$ estimators $\rho_{i,max}$, $1 \le i \le m$, where $\rho_{i,max}$ represents the $\rho_{max}$ 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:

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

**Example 5.1 (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 $A = \frac{2+2+5+1}{4} = 2.5$. 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. The ultimate value we are aiming at is 7.

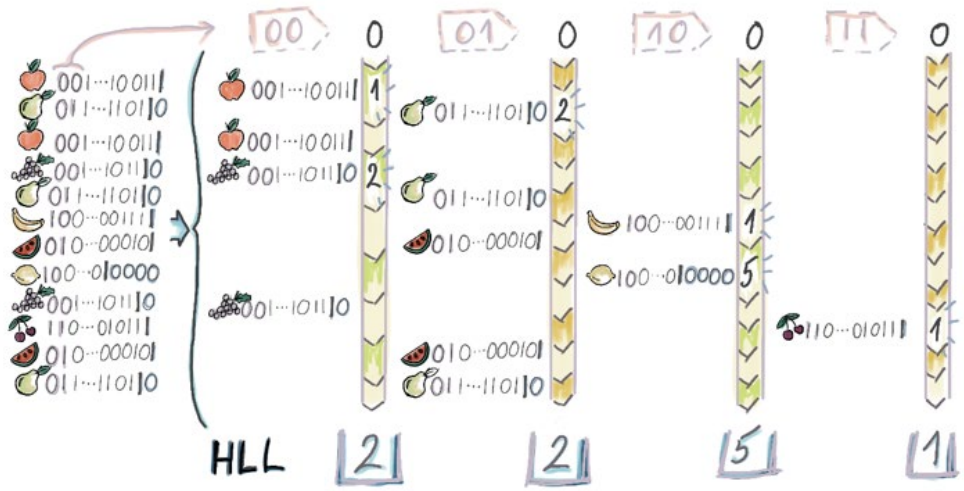



Figure 5.2: In this example, each hash is being mapped to a bucket based on its first two bits (e.g., the hash corresponding to the grape item is mapped to the bucket 00, while the hash corresponding to the pear item is mapped to the bucket 01.) When this process runs on large datasets, we would expect 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 Python-like pseudocode below shows how stochastic averaging works:

```
m = 2**b #A
S = 0 #B

for a in M #C
    h = hash(a)
    p = num_trailing_zeros(h) + 1
    bucket = first_bits(h, b) #D
    if(p > S[bucket])
        S[bucket] = p
sum = 0
for item in S
   sum += item

arit_avg = sum / m
return m * 2**avg
```

**#A m represents the number of buckets, b is number of bits used to index into a bucket**
**#B S is the list/array of m entries, storing max trailing zeroes per bucket**
**#C iterating over multiset M**
**#D integer decribed by first b bits of h**

### 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.[8]

**Example 5.1 (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 $0.292 * 22.6 \approx 6.6$, extremely close to the true cardinality of 7!**

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

#### ERROR AND SPACE CONSIDERATIONS IN LOGLOG

Using statistical analysis, it has been found that the relative error in LogLog can be closely approximated by $\frac{1.3}{\sqrt{m}}$. 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 $\frac{1.3}{\sqrt{2^{14}}} = 1.01\%$, 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 are the cardinalities 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 $\frac{k_{max}}{m}$ ($k_{max}$ 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 $\frac{k_{max}}{m} = 2^{50}$, 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.)

#### SUPERLOGLOG

One way to improve on the error of LogLog is to retain only a percentage $\theta$ 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 $\left\lceil \log_2 \left(\frac{k_{max}}{m}\right) + 3 \right\rceil$, which removes outliers but also allows us to use buckets that are $\left\lceil \log_2 \left\lceil \log_2 \left(\frac{k_{max}}{m}\right) + 3 \right\rceil \right\rceil$ bits wide. There is experimental evidence that the error drops to $\frac{1.05}{\sqrt{m}}$ 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 $\alpha_m$ 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:

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

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

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

**Example 5.1 (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 $\alpha_m$, 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 $\frac{1.04}{\sqrt{m}}$. For more details on how the error in HyperLogLog is derived, you can consult the original HyperLogLog paper.[9]

This is the end of our story about how we obtain the raw estimate in HyperLogLog, whose Python-like pseudocode is shown below (the first part of the pseudocode snippet excluding setting alpha parameters is identical to the earlier pseudocode snippet). 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:

```
alpha16 = 0.673 #A
alpha32 = 0.697
alpha64 = 0.709
alpha_m = 0.7213/(1 + 1.079/m) for m>= 128

m = 2**b
S = 0

for a in M
    h = hash(a)
    p = num_trailing_zeros(h) + 1
    bucket = first_bits(h, b)
    if(p > S[bucket])
        S[bucket] = p
sum = 0
for item in S
   sum += 2**(-1*item)

harmonic_avg = m / sum
E = alpha_m * m * harmonic_avg #B

#C
if E <= 5*m/2 #D
    V = num_registers_zero() #E
    if V != 0
        E_final = mlog(m/V)
    else
        E_final = E
if E <= 2**32 / 30 #F
    E_final = E
if E > 2**32 / 30  #G
    E_final = -2**32 * log(1 - E/2**32)
return E_final #H
```

#A Setting alpha for different values of m
#B Raw estimate
#C Computing corrected estimate
#D Small range correction
#E Let V be the number of registers equal to 0
#F Intermediate range, no correction
#G Large range correction
#H Corrected estimate with relative error of $\pm 1.04/\text{sqrt}(m)$

[9] 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.

In 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.[10]

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[11] and others.

#### ERROR AND SPACE CONSIDERATIONS IN HYPERLOGLOG

Statistical proofs show that HyperLogLog has the relative error around $\frac{1.04}{\sqrt{m}}$. 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.

[10] 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.

[11] 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. 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).[12]

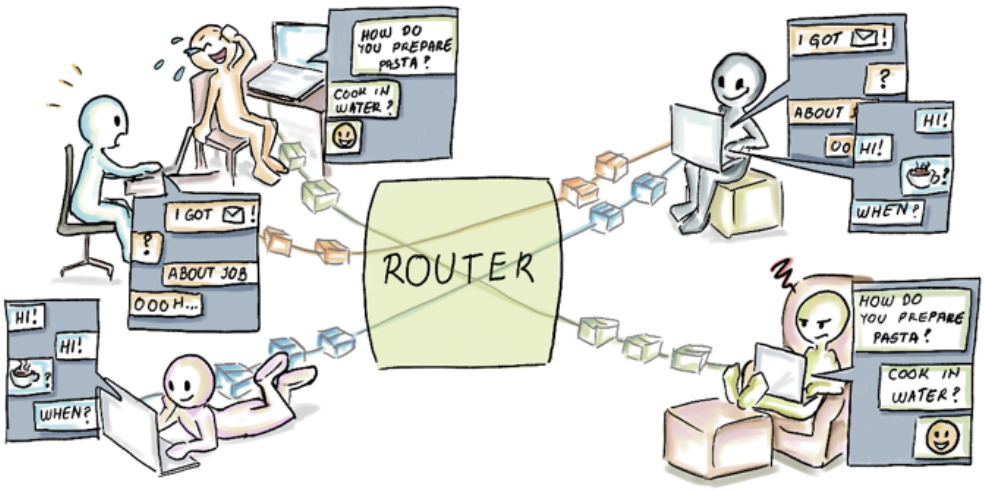



Figure 5.3: A healthy network flow. A fairly large number of packets but a small number of different flows.

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

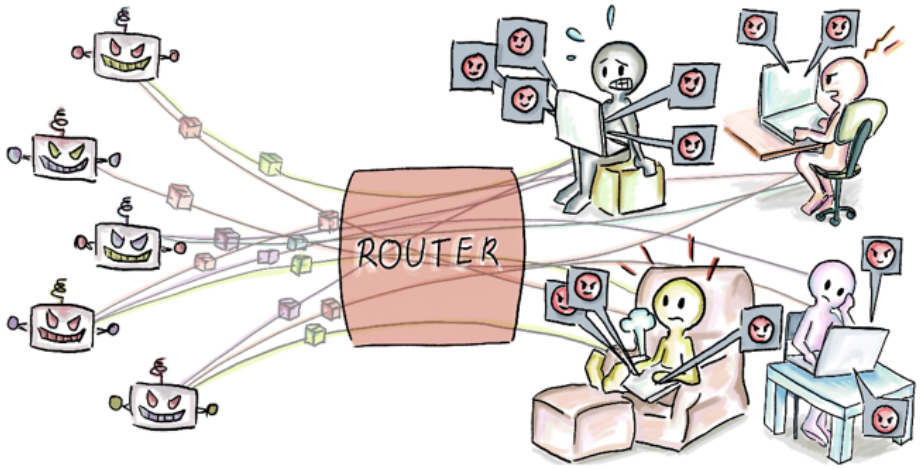



Figure 5.4: A suspicious flow — having many source-destination pairs, and one source opening a large number of different connections in a short amount of time.

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:

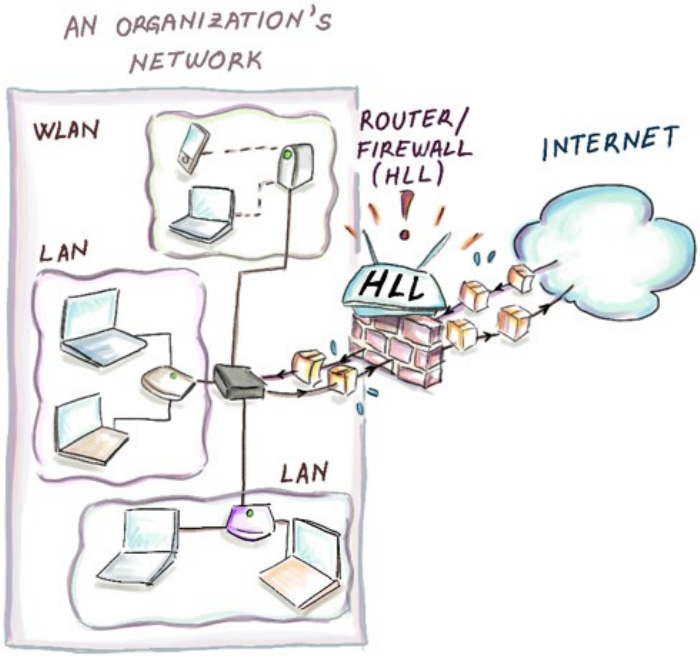



Figure 5.5: Placing a HyperLogLog at the entry point of the network within an organization can help us gather valuable statistics about the network traffic of that organization.

## 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 $\alpha_m$ (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.

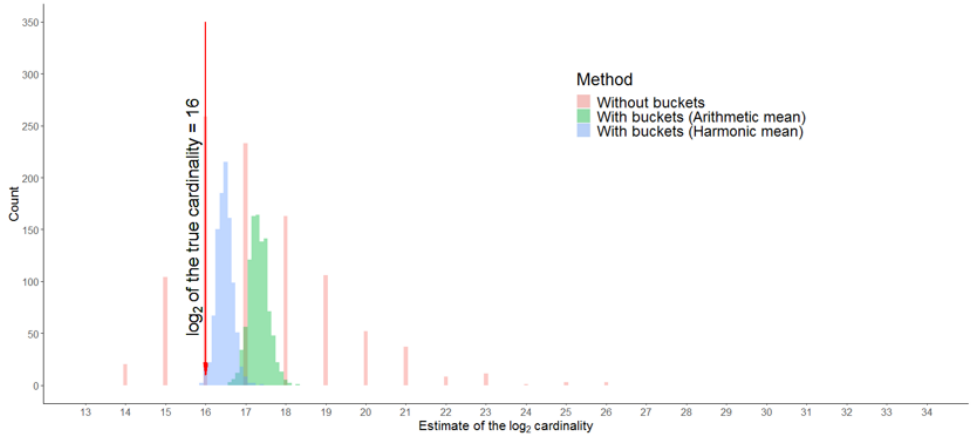



**Figure 5.6: Plot shows the comparison of the probabilistic counting (Without buckets), stochastic averaging with arithmetic mean (unnormalized) (With buckets (Arithmetic mean)) and stochastic averaging with harmonic mean (unnormalized) (With buckets (Harmonic mean)). All raw estimates show consistent 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 log2 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 $O\left(\frac{1}{\sqrt{m}}\right) = O\left(\frac{1}{8}\right)$, 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:

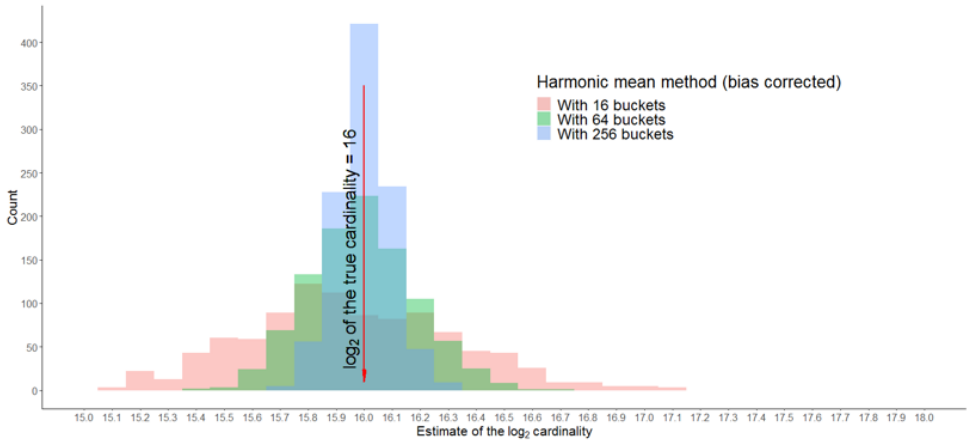



**Figure 5.7: Effect of different values of $m$ on the accuracy of log2 cardinality estimate in HyperLogLog. The larger the number of buckets, the smaller deviation from the true cardinality. In general, harmonic mean method once bias corrected, rarely over/underestimates by more than one unit of log2, 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):

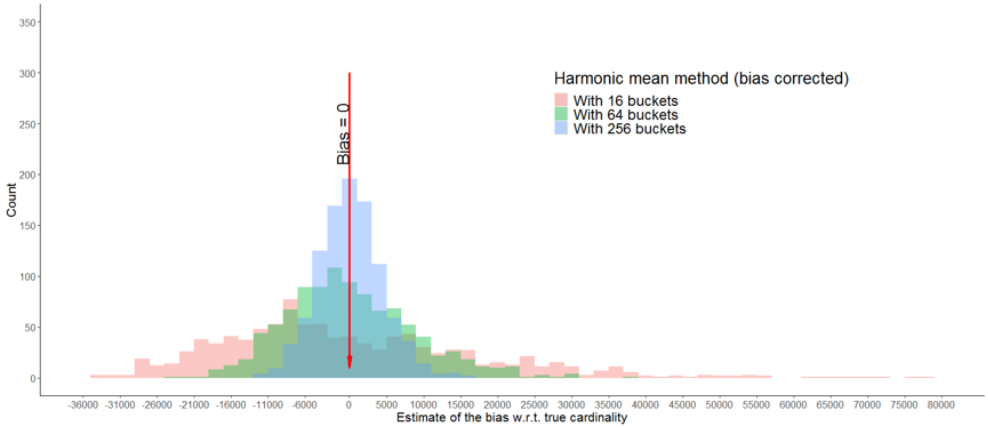



**Figure 5.8: The effect of buckets on cardinality estimate in HyperLogLog. Larger $m$ implies smaller bias and a more properly Gaussian-looking distribution.**

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 $\sigma = \frac{1.04}{\sqrt{m}}$, 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 $3\sigma$ fraction of true cardinality away from the true cardinality.***

To verify that in our simulations, we took the case of $m = 256$ buckets, hence $\sigma = \frac{1.04}{\sqrt{256}} = 0.065$. 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.[13]

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:

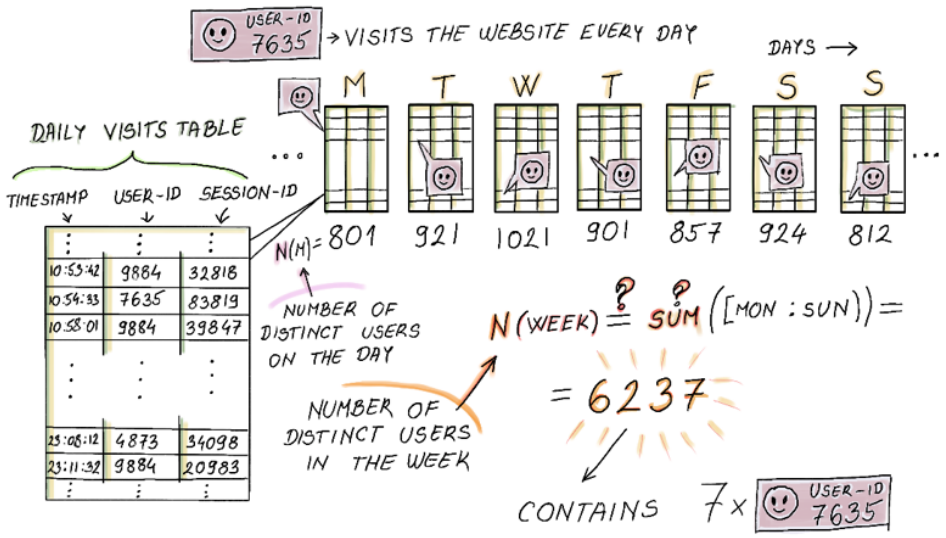



**Figure 5.9: In a daily visit table, each row indicates one visit by a user and each table maintains a separate distinct-count variable that tracks the number of different users. Considering that some users return to the website repeatedly, we can not simply sum up the individual counts to obtain the 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 bucket values are (1, 4, 2, 5) and (2, 2, 5, 3) would produce another HyperLogLog with bucket values (2, 4, 5, 5).

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

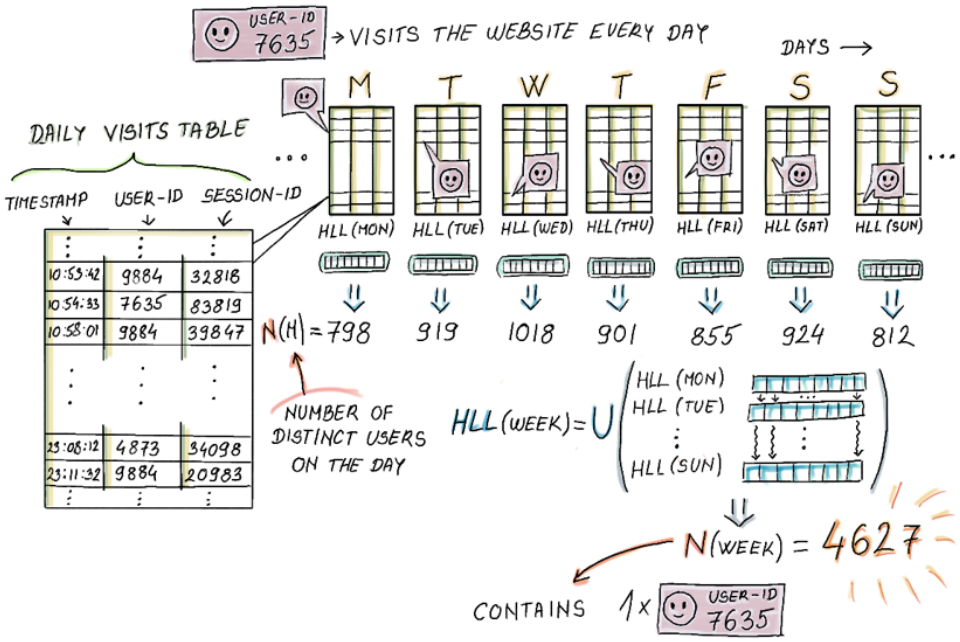



**Figure 5.10: Maintaining one HyperLogLog per daily table helps us later aggregate the HyperLogLogs over multiple days to obtain an estimate for more tables. In fact, HyperLogLog can be easily encoded so that we can maintain a table of HyperLogLog schemas , later to be decoded.**

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:

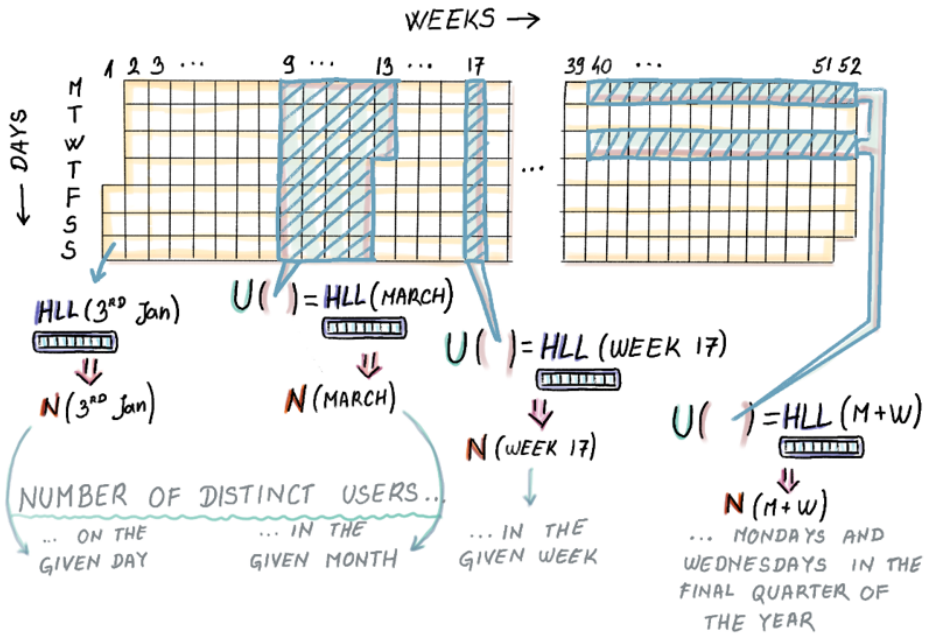



**Figure 5.11: Once we have stored daily HLLs, we can perform union over the arbitrary choice of interest to obtain aggregate cardinality estimate for a given time period.**

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.

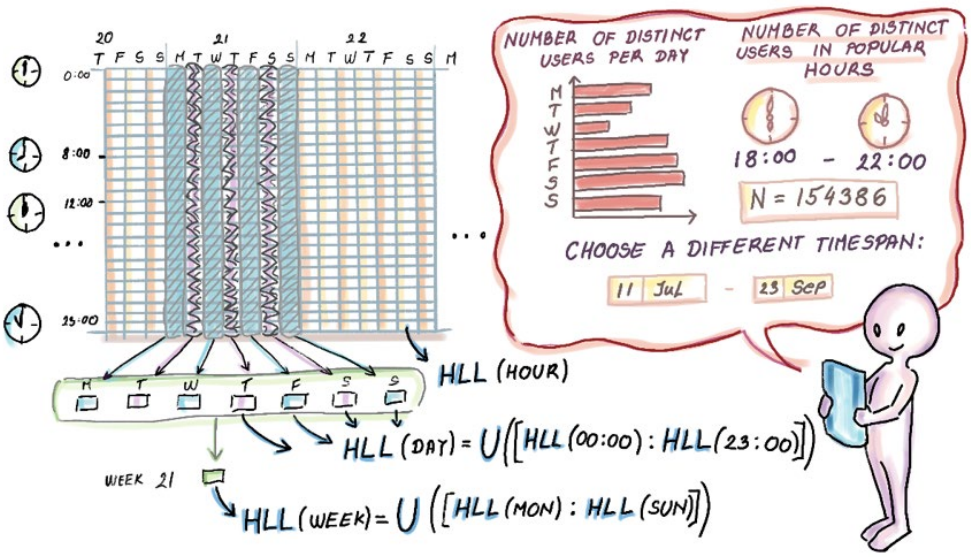



Figure 5.12: Aggregation happens over multiple levels in this case: hour, day, week, etc. Commonly in databases, when we group by different timespans, this requires a one scan of entire data for each level of aggregation.

## 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 $O(m \log_2 \log_2 k)$ and its relative error $\frac{1.04}{\sqrt{m}}$.
- 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**
- **Setting up basic computing constraints and concepts inherent to data streams**
- **Giving some probabilistic background for the next two chapters to follow**

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 the HyperLogLog 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 detailed view on massive data algorithms (needed to understand mechanics of the algorithm). Instead we will do some book-keeping and inspection of the wider context where the algorithms covered so far find their use. 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 system architecture. If this sounds too vague to you, a good place to concretize the concepts "streaming-data pipeline" and "their wider system architecture" is the Manning manuscript *Streaming Data* by Andrew G. Psaltis[1]. Now, you shouldn't think that you bought a book that tells you to buy another book, as we will introduce enough from there, of what you'll need here. Skimming sections 1.1, 1.2 and 1.3

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

in Psaltis though should clear up anything that made you curious beyond our scope here. We will show and use the model of the streaming data system / pipeline from Psaltis to stage and depict how and where Bloom Filters, Count-Min Sketches and Hyper-Log-Logs can be employed to save time/space in that particular architectural landscape.

In the past, streaming data was an exception reserved for systems controlling highly critical processes in nuclear facilities or airplanes where quick automatic reaction to anything unusual means saving human lives. With arrival of the internet, the myriad of requests issued by users to the server or a cloud of their choice is easy to conceptualize as streaming data. With the arrival of Internet-of-Things any device that is sophisticated enough to measure and then report its current state over some distance, becomes one of many producers feeding a centralized server or a cloud, with a constant stream of data. This happens at rapid rates, and in an unpredictable and volatile fashion.

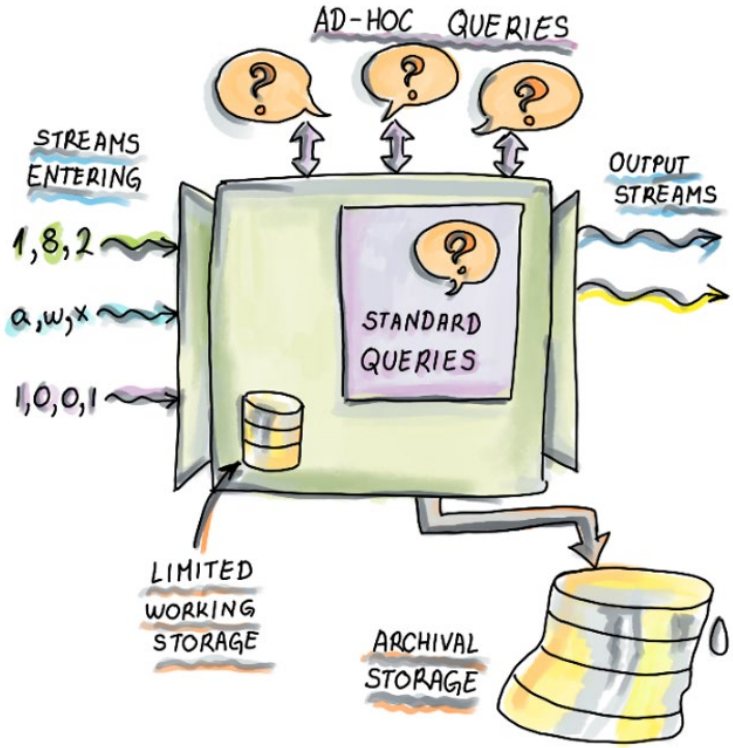



**Figure 6.1: Streaming data model. The streaming data model differs from the traditional database management system in that data passes through the processor and a small amount of working storage, and it is either never stored, or it is stored into the archival storage that is usually too large and 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 6.1[2] is a rough depiction of the streaming model that algorithm researchers use.

We are about to elucidate how components of this high-level view on streaming data (figure 6.1) correspond to specific components of a fully implemented and functional streaming data application.

Anyone trying to explain a streaming data algorithm would plant it in the *analysis tier* of the streaming-data pipeline model (Figure 6.2). There, in the heart of this system, is where its designers imagined its use. 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.

Say we need to sample some data tuples from a stream of requests issued to a cloud that hosts a popular web page or service (say Google or Amazon). We would probably design the algorithm to operate on some unique ID of each request. These requests would pass the *data-collection* and *message queuing* tier and actual random sampling of the stream of requests would happen in the *analysis tier* (Figure 6.2).

As it happens, sometimes users send the request, but fail to receive acknowledgement of receipt. Perhaps they walk out of the reach of their Wi-Fi signal. The logic in the device, smartphone or tablet, might re-send the identical request. We end up with two basically identical requests received. We have a problem now, because we don't want our sampling to be affected by such an extraneous process. If we leave the duplicates in, our sampling algorithm will pick those duplicated as many times more often, as there are identical copies, compared to the request that was received only once. Now remember, we started in the *analysis tier* but it seems that in order to use our out-of-the-box version of the sampling algorithm there, some preprocessing of the streaming data, up-stream of the *analysis tier* is necessary. Now it helps to take a step back and observe our sampling plan in a system. It just went from being a vanilla-out-of-the-box-just-plug-in-here algorithm confined to *analysis tier,* to a composite preprocessing + sampling algorithm that spreads wider over the streaming data architecture (Figure 6.2). The necessary de-duplication will happen in *message queuing* tier, hence our sampling algorithm has become possible only, if we can do de-duplication fast enough. All of a sudden we might think of Bloom Filter-enhanced de-duplication which we describe in a section to come.

Nevertheless, we made a point that although BloomFilter, HyperLogLog or stream sampling algorithms were designed to be used in confines of the safe, controlled *analysis tier* when applied, they organically create environment of interdependent problems that can be solved by applying them in a sequence or ensemble.

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

We started with the theoretical view in Figure 6.1 used to build these algorithms, but they became too simplistic on their own to solve the problem of sampling from the stream. We then zoomed out and saw that streaming data application is a natural habitat of such algorithms, with all its tiers. 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 but contributing to the same global task.

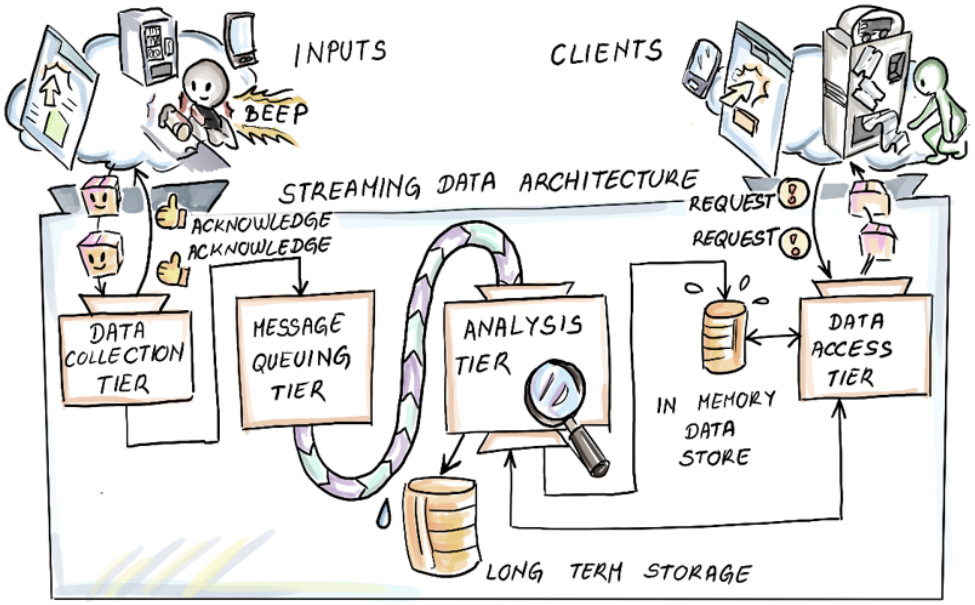



Figure 6.2: The general model of a streaming data pipeline is shown. Data producers are initiating connections by interacting with retailer's application.

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 maybe 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, compared to what producers have to do when passing data to collection tier. This 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. The stream of query results from the analysis tier are then 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. Phew! That part would probably benefit from a few more breathing pauses, so you should probably sprinkle an extra few, when you go through that again. Just make sure you don't "fall through the pipeline", as it became apparent to me once I read it, that this is what had happened to me. When you're done, good news is that the next on, are certainly "less loaded" chapter section summaries.

Section 6.1 exemplifies what an organic habitat a streaming data application makes for algorithms that we covered so far. They crop up in the streaming data application very naturally. Use-cases in this section 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 introduce concepts native to data streams that drive algorithm design and define inherent constraints under which such algorithms are developed. Figuratively we are zooming in on parts of the Figure 6.1. We should be able to recognize those constraints as an immutable feature of our data generating procedure in order to devise a solution operating within them. 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.

Section 6.3 revisits some probability theory behind sampling and estimation, as we prepare to introduce stream sampling algorithms in chapter 7. If you need to know how to modify the original algorithms and set parameters so that they are customized to your particular situation, you should probably "fight through" this part. This will happen if you find yourself in front of a huge data and even a small tweak can mean large savings in space / time. Otherwise you can skim over, so your eyes could get used to notation, because we will make use of it in chapter 7. Chapters 7 and 8 are somewhat more technical, than the previous, but the topics are inherently such.

## 6.1 Streaming Data System – a meta-example

Figure 6.2 shows the model of the streaming-data pipeline. 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

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

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.

This should not come as a surprise after our request sampling example. 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 operated on by our algorithms.

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 means that along the data-pipeline, payload (requests) sometimes becomes overhead of the data tuple, while metadata (unique request IDs) becomes relevant for the analysis depending on where in our streaming pipeline we currently are.

### 6.1.1 Bloom-join

Imagine a large retailer that sells its products online and in stores. Walmart or Wholefoods would fit the profile. The company may want a (close-to-) realtime analysis of the functional association between click patterns on its URLs and its sales transaction data. Maybe they would like to 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 or 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 brevity, we are abstracting away the necessary time-proximity that the join has to take into account. Namely, only clicks close-in-time to an online purchase from the same IP, are matched with that online purchase. Strictly, we would have to join on IP and some fixed concurrent time period when clicks and purchases happened (see figure 6.3).

Now that the OCD was soothed, by going into too much detail on a problem that is just a prerequisite, we can start with the main dish. We can assume that both databases are very large, but that HDFS-side one is larger, which is a plausible assumption. We concentrate on minimizing the size of the tables that we need to broadcast between these two systems in order to implement the desired 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) (not a bad time to glance at Figure 6.3). Here it is used as a type of a predicate (filter)

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

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.

**Exercise 6.1**

**We concretize our Bloom-Join now. Assume, for clarity, that a purchase is a tuple saving: time in milliseconds (4 bytes), IP address (4 bytes), list of items purchased (their codes, etc.) (64 Kb), and grand total bill (8 bytes). Clicks on the HDFS side are, again for clarity, Spartan tuples saving time in milliseconds (4 bytes), IP address (4 bytes) and URI (64 Kb). We can assume that purchases and clicks happen "intertwined" at constant rates that keep the ratio of clicks to purchases constant, in our case 45 clicks/purchase. All people click, but they don't all purchase something. Assume that the next join happens after 1 mil. distinct purchases were made. What would be the size of the communicated data that we save by employing a bloom filter here with a false positive rate of 0.1 % ?**

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 saves pairs of temporally proximate clicks and purchases from the same IP address (figure 6.3) or more accuratelly all their fields. This process can then serve as a type of a continuous producer for a data stream processing framework (i.e. Apache Kafka). The pairs (rows in the resulting table, figure 6.3) are then passed on to be queued, analyzed and used for perhaps (close-to-) realtime individualized ad campains.

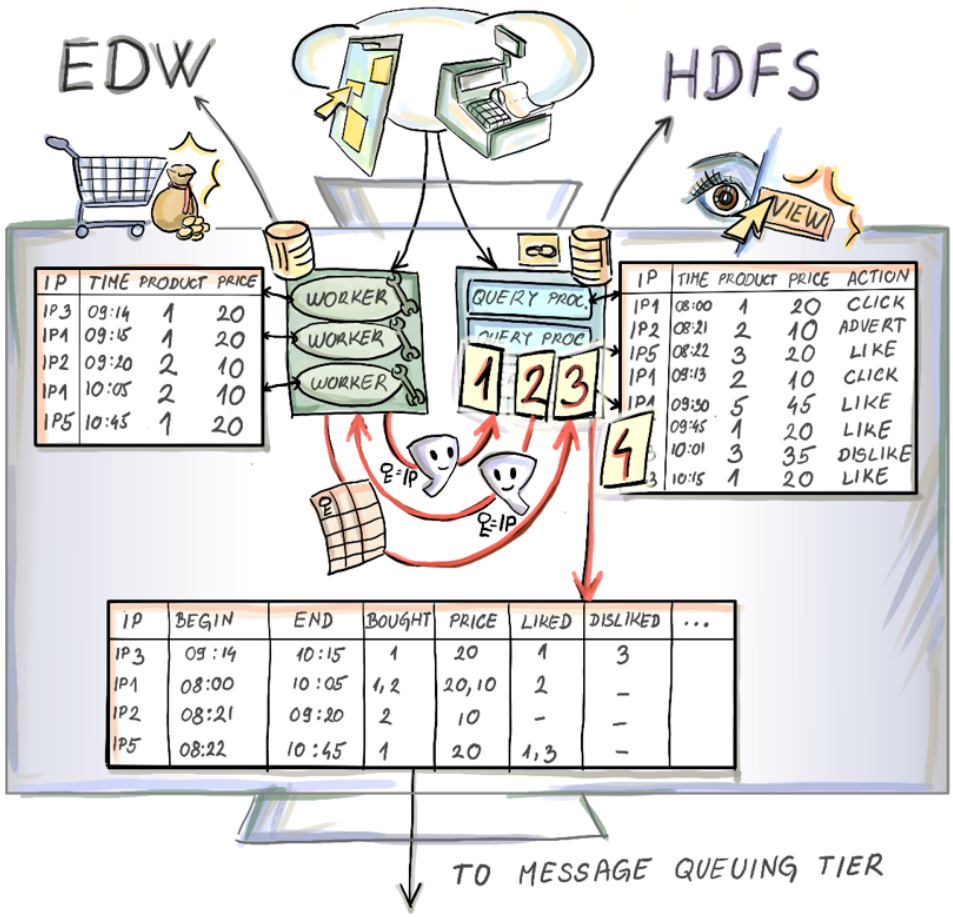



**Figure 6.3: Figure shows the pre-join communication between Enterprise Data Warehouse (EDW) implemented with fast parallel database (on the left) and a Hadoop Distributed File System (HDFS) (on the right). Before the data for financial transactions is sent from EDW side, exchange of Bloom Filters that 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 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 has passed through to the *analysis tier* and what should be next.

Nodes in *message queueing tier* are commonly called *brokers* and besides keeping message queues consistent, they do other preprocessing steps too. Imagine similalrly to our example with requests, 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 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. On the other hand, corporate systems 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 %), just by the vast majority who don't encounter any problems, 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 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. These worker nodes are connected to highspeed databases. 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

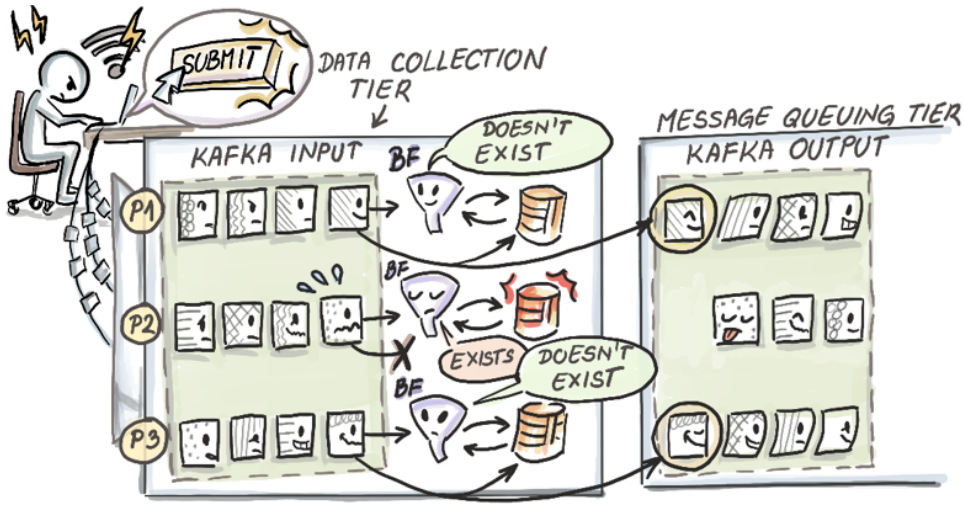



**Figure 6.4: Intermediate nodes connected to fast databases implementing de-duplication to remove the repeated messaging instances in a streaming data pipeline. Each node keeps a Bloom Filter of messages it saved and at arrival of the next message ID checks the message ID hash against its Bloom Filter. 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 outliers come in the shape of overloaded 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 relies on the monitoring of package headers in the network. The last the algorithm would need for this is the basic information abot each flow FL:=[source IP, source port, destination IP, destination port, protocol]). Monitoring the stream of requests of this form issued to a load balancer node in charge of the network of brokers allows us to identify a 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 a count-min-sketch are packet headers and the counter is incremented by the size of the current packet.

In the more specific case of a broker in the data streaming application, counter would be incremented by the number of requests queued for the same broker. This could happen due to some momentarily increasing number of producers (called *sudden-bursts* in this context). The traffic measuring or load balancing application would then first estimate min counts: How "heavy" is each flow or how "busy" is each broker and these would be periodically divided by the length of the measurement period. This gives us flow rates / queuing rates. A network administrator or a data pipeline engineer would then discriminate good flows from bad by applying some threshold informed from their use case. After identifying the culprits they can apply some curtailing strategy.

Due to the count-min-sketch algorithm the flows under the threshold are per definition 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 6.2**

**We revisit the de-duplication use-case for Bloom-Filter in the e-commerce system. Assume you were given an assignment by your superior to inspect and finish the solution started by the engineer that left the company recently. You understand the logic of the solution as it seems to save time and prevent charging the customer twice. Nevertheless there is a „price" to pay. What happens once the Bloom-filter delivers a *false* positive related to a particular request carrying information about a like, submitted comment or a clicked ad ? What if the client app submitted a payment ? Is there anything there we should see through ?**

A generic network traffic monitoring application that we gave as an example can be found in a cloud service on which your streaming data application is running, while the load balancing use-case would pertain to the inner workings of 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 of them becoming visible at different "image resolutions".

## 6.2 Practical constraints and concepts in data streams

Next, we will introduce some computing and streaming data concepts that the streaming data algorithm designers have to observe, and by which their algorithms are evaluated.

### 6.2.1 In Real 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 the sensory stimulation from the events we are immediate witnesses of. If this sounds like hair splitting to you, it would be hard for 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. If it produces results and helps in decision making, with latency that does not leave the bussines lag its competitors, we wouldn't be off my much to call it real-time. 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, 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 importance, we can continue with what we mean by *small time* and *small space*.

### 6.2.2 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.

### 6.2.3 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 an 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 area. Any solution would need to implement the logic for calculating rates (count divided by the length of the logging period) of occurrence of particular 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[6].

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

### 6.2.4 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 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 as the new ones.

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 a liability on average. When faced with concept shifts and drifts, queries in landmark streams are prone to inertia and can be too slow to "react" 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

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

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.

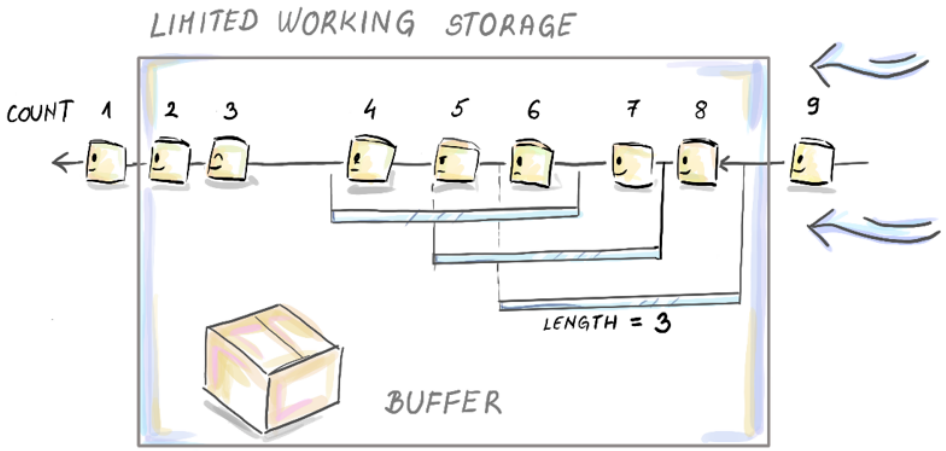



**Figure 6.6: Figure shows last three sliding movements by the count-based window. Notice that window length is not necessarily all that we can fit into our working storage, but that is definitely the maximum in history that we can cover. Hence in applications where we want "as much history as possible" 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.

Next section is intended as a review of sampling theory. This should make it easier for those more technically curious among you, to appreciate the nuances of chapter 7.

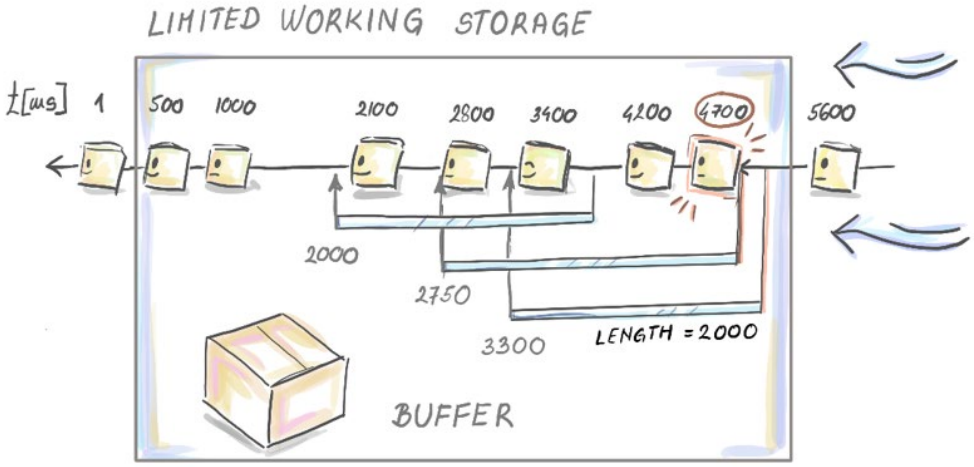



Figure 6.7: Here we can see the time-based sliding window of length W=2000 ms and its three last sliding (meaningful w.r.t. to the arrival times of the data tuples) movements. We started the data stream at time 0 ms and at time 1 ms the first data tuple arrived. Then two more arrived at 500ms and 1000 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).

## 6.3 Math Bit : Sampling and estimation

The idea of sampling came out of impossibility to answer questions about logistically intractable, large sets. For example, if you and someone you share an IP address with, surf the interet, your requests are received from the same IP, but using different browsers will leave a different HTTP fingerprint. We might be interested in the average number of HTTP fingerprints over all possible IP addresses. There is no chance in the world we would be able to get a correct answer, but even an estimate would be more than what we knew, when the idea popped into our head for the first time. To get an estimate we would *sample* from the IP space.

First recorded use of this idea came in 1786 as an attempt by Pierre Simon Laplace to estimate the population of France. Naturally, the estimate was not correct, but what Laplace did, and what made sampling a powerful tool, was to provide an upper bound on the chance, that his sampling based estimate is (too) far from the correct answer. Providing an estimate was not that novel, but adding to it a structured way to mesaure, and perhaps curb the uncertainty of the estimate compared to the truth, was new, and luckily, very generally applicable idea.

To define a sampling process in practice we need two components, a (finite) *population* that we are interested in (or some aspect of it) and some way to pick random members of that population, final, "materialized" artefact of which, is called

a *sample*. You will find that the word *sample* is often used for individual elements / observations that make up the *sample*. We find this to be confusing, hence for individual members of the sample we use elements / observations / members, while collection of these is the *sample*. Depending on the way by which we acquire these random members, sampling can be *representative* (unbiased) or *biased* with respect to the population.

If every subset of $\mathbf{k}$ elements from the population, has the same chance to become our final sample of size $\mathbf{k}$, then the sampling process is representative of the population. In our IP space, this would translate into making sure that each subset of $\mathbf{k}$ IP - addresses has equal probability to be picked into our sample. This also means that every individual member of the population has the same chance of being selected, independent of anything else about that member. If we can guarantee this, we got ourselves a *simple random sample* (SRS).

Now let's think about how this translates into our IP-space example. In reality, rates of requests over IP space are different. Crawlers (scripts visiting and cataloging web sites automatically) send requests perhaps more often than a human who is browsing, during a single session. If we were to pick IPs in the following manner "sample a random request and add its IP address to the sample", crawlers would have a higher chance to get into our sample. Once we decide that the sample is big enough we might have over-represented set of IPs related to crawlers, compared to the number of those used by humans. This would translate into having a wrong idea about the true average number of fingerprints per IP. If we make sure that each IP address has the same chance of making it into the sample, independent of anything else about that IP address, this can't happen. This is what one is usually after, because this allowed Laplace a) to believe he has a good estimate of the population of France and b) to calculate those bounds on uncertainty of his estimate. Our original sampling strategy to sample IP addresses directly via all requests recieved was a *biased sampling strategy.*

### 6.3.1 Biased sampling strategy

To describe what makes a sampling proces biased, we will use a hypothetical population of bills logged by a large retailer. For now, it is of no relevance how this data is available to us. It can reside in a database or it might be data received so far from a data stream. We denote the number of these individual purchases $\mathrm{N}$. We would like to know what is the proportion of purchases made via loyalty-card that the retailer offers. Had our marketing department made plans and informed us on time that they will be needing this informaton, saving and updating this information with every newly seen purchase is trivial, but if they all of a sudden decide they would like to now this (*ad-hoc query*), we can get a good estimate for them by keeping a sample, and returning the sample proportion of those loyalty-card purchases.

The purchases are partitioned into purchases with and without the use of loyalty-card. The true value $p_L$ of the proportion of purchases with a loyalty-card (L) is clear, it is the number of members with L divided by $\mathrm{N}$. In our sample of size $\mathbf{k}$ this translates into having any number $\mathbf{i}$, of elements with L, and any number $\mathbf{j}$, of members without L, with $\mathbf{i}+\mathbf{j}=\mathbf{k}$.

We will describe two sampling processes in this conext. First one is unbiased, while the second is a biased sampling process on the population of purchases that were logged. We will use the example with $\mathrm{N}=10$, $\mathrm{k}=5$ and $p_L=4/5$. Let the set beneath represent the population

$$\{ x_{1L}, x_{2\overline{L}}, x_{3L}, x_{4L}, x_{5\overline{L}}, x_{6L}, x_{7L}, x_{8L}, x_{9L}, x_{10L} \}$$

In the index for each element we can read off some form of ID and indicator about the presence/absence of the characteristic L (in this case elemnt with id 2 and 5 are bought without presenting a lolyalty card). For any 5-element subset of these 10 elements a simple combinatorial argument can be made about the probability to pick that specific subset. Introductory notes for either probability theory or discrete math course teaches you in the first few weeks that there are $\binom{10}{5} = 252$ different 5-element subsets and if each is supposed to be equaly probable, then probability to pick any specific one is $1/252$. Now imagine we have a $252$-sided die showing numbers from 1, 2, …, 252 and we (in any arbitrary way) enumerate all 252 5-element subsets. Then throwing this die and picking the subset indicated on the side of the die after it lands, is a representative sampling strategy.

It is helpful to know the probability for each $x_i, \mathrm{i} = 1,2, \ldots, 10$ to be selected into the sample. For this sampling process it is simply $\frac{5}{10} = 0.5$ for any of them, *independent of whether or not a loyalty-card was used*.

Any 5-element subset (our sample) that you can imagine here, belongs to one of the three types. We will call the sample type 5 when all 5 purchases turn out to be made via loyalty-card. Type 4 is made up of 1 purchase made without the loyalty card and 4 purchases made with it. And lastly, type 3, that has 2 purchases made without the loyalty-card and 3 purchases with it. Estimates of true $\mathrm{p_L} = 4/5$ from these three types of the sample would be 1, 4/5 and 3/5, in that order.

Now let's assume a different sampling strategy: we first roll a three-sided *biased* die (each side showing the number of purchases with the loyalty card, hence 3,4,5) to decide which type, 5, 4 or 3 of a sample we will draw. We then reach in "partition $\mathrm{L}$" and "partition $\overline{\mathrm{L}}$" separately and draw as many purchases from each, as decided by the first biased die. If we roll 4 for example, we know that we need to take 1 from "partition $\overline{\mathrm{L}}$" and 4 from "partition $\mathrm{L}$". For picking actual purchases from these two, we will use the representative strategy described above. This time we will need three pairs of dice for the second stage. We need a pair because we have to sample representatively from "partition $\overline{\mathrm{L}}$" and "partition $\mathrm{L}$". Three of them because, depending on the type of a sample we rolled using our first biased die, we will draw different number of purchases from "partition $\overline{\mathrm{L}}$" and "partition $\mathrm{L}$". This leads to $\binom{8}{5}$-sided and $\binom{2}{0}$-sided die sample type 5, $\binom{8}{4}$-sided and $\binom{2}{1}$-sided die for sample type 4 and $\binom{8}{3}$-sided and $\binom{2}{2}$-sided die for type 3. (those among you who remember counting techniques, will notice that first and third pair are actually same two dice, hence you actually need only 4 additional dice).

Now let's see what is the probability to pick a 5-element subset of a specific type 5, 4 or 3. Let's assume that the sides corresponding to sample type 5, 4 and 3 are seen with probabilities $\frac{2}{5}, \frac{2}{5}$ and $\frac{1}{5}$, respectively (this is where bias of the first die comes into play). For the sample of type 5, we have

$$\frac{2}{5} \times \frac{1}{\binom{8}{5}} \times \frac{1}{\binom{2}{0}} = \frac{1}{140},$$

for the sample of type 4 we have

$$\frac{2}{5} \times \frac{1}{\binom{8}{4}} \times \frac{1}{\binom{2}{1}} = \frac{1}{350},$$

and for the sample type 3 we have

$$\frac{1}{5} \times \frac{1}{\binom{8}{3}} \times \frac{1}{\binom{2}{2}} = \frac{1}{280}.$$

Hence, it is easy to see that this sample strategy is not representative, or it is biased, since not all 5-element subsets are equaly likely to be "materialized" into a sample. Same as the number of requests issued introduced bias into our request sampling strategy, the biased die allowed samples with more purchases made with a loyalty card, to be more probable than those with fewer loyalty-card purchases. So let us now see what does it do to the probability of selection of a single observation. Remember that for the representative sampling strategy, probability of selection into a sample for *any* single $x_i$ was $0.5$. *independent of the feature L*. For the biased sampling strategy, we don't expect this to be the same since a biased die "prefers" samples with higher number of purchases made with a loyalty card. This bias will "trickle down" to the level of a single observation and will "tilt" its chances to be a part of the final sample *depending on its L-feature*. In this case we need to calculate probabilities of selection separately for elements with L and elements without it. We leave the derivation of the exact probability for selection of a single $x_{iL}$ and $x_{i\bar{L}}$ as an exercise and just state here that for each $x_{iL}$ this probability is $0.525$, while for $x_{i\bar{L}}$ it amounts to $0.4$.

**Exercise 6.3**

**Calculating selection probabilities for $x_{iA}$, $A \in \{L,\bar{L}\}$ in the cae of our biased strategy can be done via iterative expectation method. We calculate for each $A \in \{L,\bar{L}\}$ separately a probability that $x_{iA}$ is selected into the sample conditioning on what the biased die shows. Can you get what we got for $x_{iA}$, $A \in \{L,\bar{L}\}$ ?**

**Exercise 6.4**

**What probabilities for the three sides of a biased die correspond to the representative sampling strategy with the line at 0.5 in Figure 7.1 ? In other words, we could use the second biased strategy with specifically chosen probability weights for each side, and emulate representative sampling strategy.**

This is where one can truly visualize the bias, becaue we see the "tilt" away from $0.5$ for each observation in the population *dependent on the feature L*. Figure 6.8 shows the bias in selection probabilities $p_i^{BS}$ for differently biased dice compared to $p_i^{SRS} = 0.5$ selection probabilities under a representative sampling strategy.

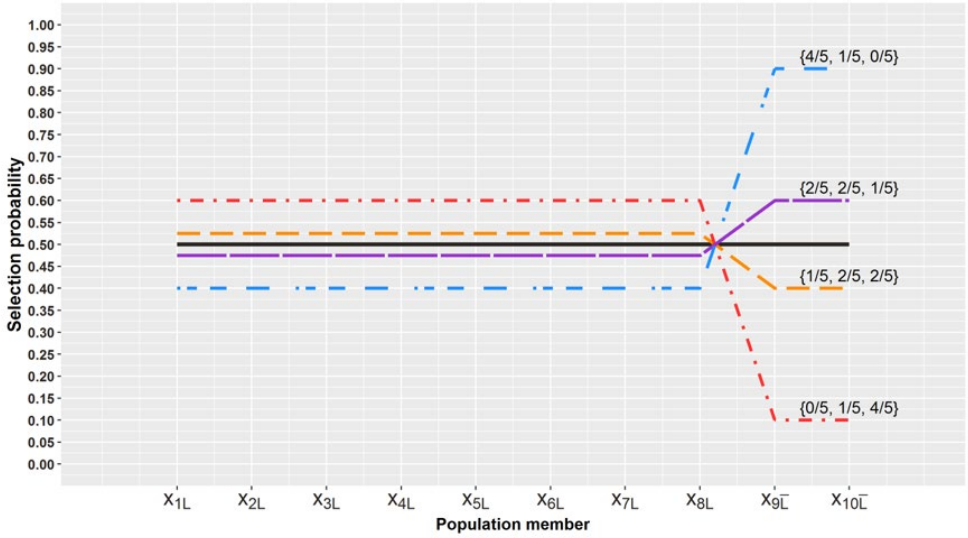



**Figure 6.8 Graph shows selection probabilities for differently biased die from our example. The probabilities to draw a sample of type 3,4 and 5 are shown on each line that shows probability of selection for all $x_{iA}, i = 1, \ldots, 10$ and $A \in \{L, \bar{L}\}$. Notice how for any biased sampling strategy, via the three-sided biased die, the selection probabilities "move" away for 0.5. The drection of the "movement" depends on loyalty card existence $A$ with respect to the particular purchase $x_{iA}$. Our original biased die corresponds to the shift denoted by $1/5,\ 2/5,\ 2/5$.**

## 6.3.2 Estimation from a representative sample

Of course, sampling without the estimation step that follows it, doesn't serve much purpose by itself, hence we will now see how the subsequent estimation step from a representative sample looks like. Assume we want to know exact number of purchases performed by presenting the loyalty card, correct value of this characteristic for our population is $\theta = 8$ and we get it, if we add 1 for every element in the population with the characteristic L and 0 for all those without it.

$$\theta = \sum_{i=1}^{10} c_{iL} = 8$$

We will estimate this population sum by utilizing its random sample counterpart, "sample sum" shown beneath.

$$\hat{\theta} = \sum_{j=1}^{5} I_{jL}$$

Sample $S$ is a random subset of size 5, so we have to take into account that instead of fixed values $c_{iL}$ , we are now dealing with random indicator variables $I_{jL}$ for $j = 1, 2, 3, 4, 5$. After some specific sample is materialized, we can speak of fixed values. $I_{jL}$ is 1 if the j-th element of the sample has characteristic L, and 0 otherwise. For any posible sample, value of this sum depends on our sample type from before (sample of type 3,4 or 5). All samples of type 3, 4 and 5 will have values 3, 4 and 5 for this sum, respectively.

Since the sum is random, let us check what is the expectation of the "sample sum" (its long-term behavior in an experiment where we draw a random sample of size 5 repetitively). The expectation of a single $I_{jL}$ for any $j = 1, 2, 3, 4, 5$ under our representative sampling strategy is $\frac{8}{10}$ (we leave this as an exercise to show), and 5 of those, make 4. With 4, we seem to be off from our aim by a factor of 2. But we can notice that this is exactly the factor by which our population size and sample size are off, too. Population has $10 = N$ members, while the sample S has $5 = k$. Hence, if we scale $\hat{\theta}$ up by $\frac{N}{k}$, we will be on target with the expectation. So our final form of the estimator is

$$\widehat{\theta_{su}} = \frac{10}{5}\sum_{j=1}^{5} I_{jL}$$

which makes it an unbiased and a consistent estimator of our population sum $\theta$ (as the sample size inceases, $\hat{\theta}_{su}$ converges to our true value $\theta = 8$). This general form of the scale-up estimator in the last equation is known as *Horvitz-Thompson type estimator*.

**Exercise 6.5**

**How would you go about calculating the expected value of $I_{jL}$? Can you show that it is, what we claim?**

The "scaling – up" step is a general strategy and we can use it to estimate other population sums too, like total amount spent on online purchases in the last month (what is the population of interest here?). Or, if you want to know how many purchases were higher than $100\,\$$ in the last month, you can use our scale up estimator with the characteristic L substituted by "purchase higher than $100\,\$$". Population sums like $\theta$ look specific, but they can be used to define any linear transformation of it (any average). Notice that if you divide $\theta$ by 10 you get $p_L = 4/5$ which is a type of average too, so you can use $\widehat{\theta_{su}}$ to estimate $p_L$ as well. Hence, representative sampling strategy and "scale – up" estimator are a powerfull general tool to get a good estimate of the corresponding population parameter.

Population of interest can be a classical one, it can be all the purchase tuples from last month residing in a database or it can be all purchases that arrived in the last 24 hours from our producers to our streaming data pipeline. The probabilistic argument described above is the same for all three, technical implementation of a sampling process within these three domains, on the other hand, is somewhat different.

## 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 applications 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 balancing are all real examples of such opportunities.
- 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 adjustments to detect such phenomena known as concept shifts and concept drifts.
- Sampling is a powerful and long established technique for answering questions about an intractable set by systematically forming its subset and answering the same question from the subset. Long and well established theory for statistical inference from the unbiased or biased sample can be of a lot of help to decide on the sample size and choose an estimator for our query answer to guarantee accuracy and precision necessary for practical purposes.

# 7

# *Sampling from data streams*

**This chapter covers**

- Sampling from an infinite landmark stream – Bernoulli Sampling, Reservoir Sampling and Biased Reservoir Sampling
- Incorporating recency by using sliding window and how to sample from it – Chain Sampling and Priority Sampling
- Showcasing the difference between representative and biased sampling strategy on a landmark stream with sudden shift - Reservoir Sampling vs. Biased Reservoir Sampling
- Exploring R and Python packages and libraries for writing and executing tasks on data streams

With section 6.3 under our belt, we are ready to fully appreciate sampling as a single task staged in the *Analysis Tier.* Although we showed that this division of the streaming data architecture is not so clear-cut, we will imagine the stream processor sampling the incoming stream in this tier. This will help to introduce the sampling algorithm without any additional complexity coming from de-duplication, merging or generally preprocessing of the data. In our fingerprint-rate example, the incoming requests would first go through IP de-duplication and then appear in front of the stream processor that would materialize a representative sample. The current state of the sample is then used to answer a continuous or an ad-hoc query approximately, but quickly. We will use our IP sampling use-case to illustrate each algorithm.

Theory for sampling from a stream developed naturally from database sampling. Database sampling comes with a long and rich research and publication record starting as early as 1986 with work by Olken and Roten[1]. One of research directions in database sampling, *online*

[1] F. Olken, D. Rotem, Simple random sampling from relational databases, in Proc. 12th VLDB (1986).

*aggregation*, served as an inception platform for our main topic in this chapter, *sampling from data streams*. We will introduce specific algorithms operating on different stream models introduced in section 6.2.

## 7.1 Sampling from a landmark stream

We will dip our fingers and try to "tap" into our first stream of data via sampling from a landmark stream model. This is a continuous stream of data that is not "windowed". Data items arrive continuously are operated on and disappear forever. Well, forever is too strong perhaps, since it usually moves to slow, massive, secondary memory storage. Sampling from such a stream "only" needs to somehow ensure, that at every moment in the stream evolution, we are keeping a representative sample of the data seen so far. This is an easier task compared to the one of sampling from a "windowed" (sequence based or timestamp based) data stream. Here we don't have to implement the logic for updating the sample, once an element of the sample has exited the window (ages out). This inevitably costs time and brings us to the criterion for evaluating the "goodness" of the sampling algorithms we will present. Algorithm that answers the query "well" should be able to create and / or update the sample in a single pass over the elements of the stream. It should also give an approximate answer to the (continuous or ad-hoc) query using the sample of size poly-logarithmic in N (N being the number of stream elements seen so far, for landmark streams). This notion of *approximate* has to be concretized, when algorithm designers want to be able to compare their solutions. The approximate answer then means, the answer should be within $\varepsilon$ absolute/relative error of the correct answer, except for some small failure probability $\delta$, when it is not. So we want to be $\varepsilon$ – accurate $100 \times (1 - \delta)$ percent of time. For windowed streams, the size of the window $\omega$ takes the role of the parameter N, when we talk about poly-logarithmic size. We assume that $\omega$ is too big for us to fit all tuples between $t$ and in $t - \omega$ in memory.

### 7.1.1 Bernoulli sampling

Bernoulli sampling is a classical sampling strategy (Daniel Bernoulli lived in 18th century when a sheet of paper with data on crop yield over time in different parts of a county would be the closest concept to a database). It is also representative sampling strategy that lived through its second spring once easily and quickly accessible data elements (rise of database sampling) were available. Easiest way to exemplify the strategy is to imagine playing the modified game of picking petals one at a time off a flower (with the underlying audio track wondering about the existence of affection towards you, from a person of a gender of your own choosing, of course). In the familiar manner, once you ruin a perfectly fine flower, you stop with the game. What you also did, is sampled the petals according to a degenerated version of *Bernoulli sampling*, namely you picked each petal with probability $p = 1$. Naturally, *Bernoulli sampling* of practical use will have $p \in (0,1)$ with $p$ representing the true sampling fraction for any number of elements so far encountered in the stream.

If we take our stream of de-duplicated IP addresses, we would pick every 1/p-th one that would pass. Here lies the beauty of the simplicity of this method, at any moment we can be sure that our sample is completely random sample of size $pN$, where $N$ is the number of

elements seen so far. We would then use those to calculate estimated number of fingerprints per IP in all the IP space.

For each arriving IP-address we would toss a coin showing heads with probability $p$ and tails with $1-p$. If we introduce the currently seen IP-address each time we see heads, each IP seen so far would have *p* chance to be in the sample. To exemplify, with a fair coin we would on average take every second IP seen. Notice here an inevitable feature, although $p$ remains constant, the size $k$ of the sample is a binomially distributed random variable, and its expected size $Np$ grows with $N$. We can now define *Bernoulli sampling* more formally, given a sequence of elements $e_1, e_2, e_3, \ldots, e_i, \ldots$ from a landmark stream, include each element $e_i$ of the stream with probability $p \in (0,1)$ independently from any other element already passed, or yet to come. Figure 7.1 illustrates this idea.

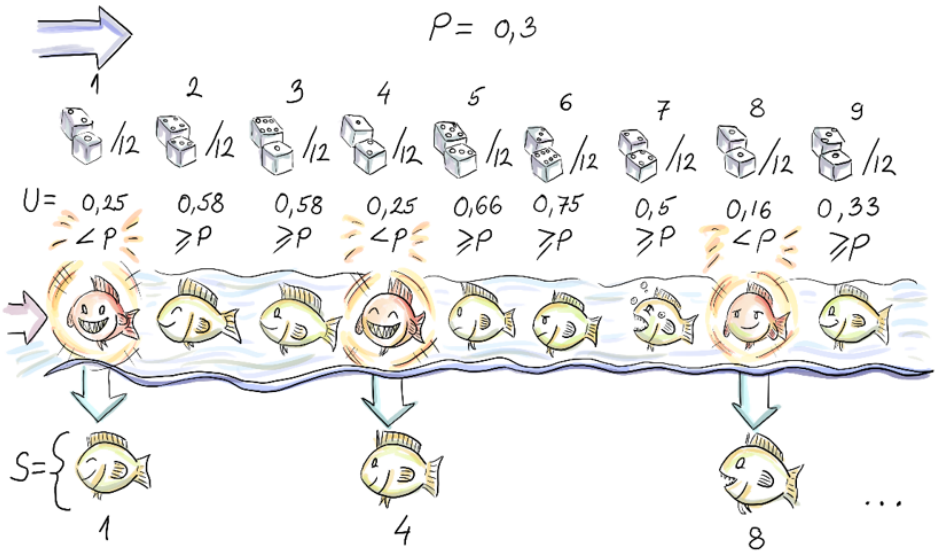



**Figure 7.1 You can see how first, fourth and eight element of the stream are included in the sample, since the corresponding pseudo-random value uniform on [0,1] turned out less than p=0.3.**

If you are not into knitty-gritty parts of the implementation, you now have all the necessary information about Bernoulli sampling to skip to the bottom of the next page. Naive implementation, invokes a pseudo-random number generator algorithm (PRNG) for each element seen, produces a uniformly distributed random number between 0 and 1 and includes the element into a sample, if the number is less than $p$.

Getting into the theory behind PRNG's would be a sudden and time consuming change of context at this point. Luckily all we need to know about PRNG's to appreciate what's to come, is that these are efficient deterministic algorithms producing sequence of pseudo-random numbers. These numbers, if the algorithm preserves some assumptions from number theory, become indistinguishable, for practical purposes, from a sequence of real random numbers.

Also important, is that they, although efficient, do cost some time and being in the streaming data context we don't want to call them more often than necessary.

It is perhaps cordial to notice, that PRNG are actually an anathema to randomness[2], but the number theory behind them is nothing short of fascinating, so if you haven't already, it's in no way waste of time to read up on them. Not the easiest, but probably the best place to do this is Ch. 3 of D. Knuth's "The Art of Computer Programming, Volume 2: Seminumerical Algorithms".

To save on calls to PRNG, our implementation will use the fact that the number of elements to skip after the last inclusion is geometrically distributed random variable. Instead of having to do something each time a new element in the stream is seen, we'll only operate when a new element is actually included in the sample. This is because each time we just generate a "skip" of indices that we will let by, leading us to the next element to include.

We make use of a very general theorem from probability theory called *inverse probability integral transform.* Sorry for the perhaps lofty words there! For our particular case, it says that if a U is a uniformly distributed random variable on the interval (0, 1), then $\Delta = \left\lfloor \frac{\log U}{\log(1-p)} \right\rfloor$ gives the number of elements to **skip, before** the next element that is to include ($\lfloor x \rfloor$ denotes the smallest integer less than or equal to x), if we are to include every p-th one. Phew! Beneath is the pseudo-code, but maybe have a breather.

```
S = []    #A
p = 0.01    #A
j = 0    #B
i = 1    #B

U = PRNG_unif(0,1)     #C

Δ = ⌊ log U / log(1−p) ⌋     #D

j = Δ + 1     #E

while (True):
    while (i!=j ):
        i += 1     #F

    S.append(e_i)    #G
    U = PRNG_unif(0,1)

    Δ = ⌊ log U / log(1−p) ⌋

    j = j + Δ + 1
```

**#A Initialize an empty buffer S to keep the sample in and set the sampling probability p.**
**#B Set j, index of the next element to include, to 0, and i index of the current element to 1.**
**#C Draw uniformly the first U**
**#D Produce the first skip.**
**#E Calculate the index of the first element to include.**
**#F Pass all indices between the last and the next inclusion.**
**#G Include the current element in the sample.**

[2] "Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin." – J. von Neumann

At any time *t* the sample S is a Bernoulli random sample from all tuples seen so far with inclusion probability p. As you can see from the pseudo-code we call the PRNG and log function only at times of inclusion of a new element into the sample S.

A generalized version of Bernoulli sampling uses for each item separate and unique inclusion probability $p_i$. This sampling scheme is called *Poisson sampling* and inclusion of $X_i$ is a Bernoulli trial with success probability $p_i$. If we have a fairly good idea about the multiplicities of $X_i$ (i.e. in the problem of estimating the total sum of dollars from purchases, some amounts $X_i$ show up more often than the others say $X_j$) we can let them be reflected in the probabilities for inclusion. $X_i$ with higher multiplicities will have higher $p_i$, while those amounts that appear only rarely, will have correspondingly smaller $p_i$. This type of *biased sampling* a) allows building a Horvitz-Thompson type of an estimator straightforwardly and b) reduces the variance of the estimate. Unfortunately, generating skips for Poisson sampling is not a trivial task.

In a distributed computing environment, in which any industrial streaming data application operates, sampling algorithm should offer itself simply to parallelization over a number of stream (sampling) operators. This advantage of Bernoulli sampling should be obvious, sampling $r$ sub-streams via Bernoulli sampling with inclusion probability $p$ will result in a representative sample from the whole stream after the $r$ samples are pooled at a designated master node.

Main drawback of Bernoulli as well as Poisson sampling is the random sample size that in the case of a landmark stream in theory can grow infinitely. Some attempts were made to combine Bernoulli sampling and a sample-size-curtailing strategy for deleting some elements from the Bernoulli sample or using reservoir sampling, once the sample size exceeds a certain threshold. Such strategies introduce some bias into the original sampling algorithm. In the case of the Bernoulli sampling, the sampling strategy becomes biased, with different bias $p_i^{BS} - p$ for each element $i$. There is often no closed functional form of this bias which that we can use to recover $p_i^{BS}$ by adding that bias to $p$, `hence it becomes difficult to correctly use a` Horvitz-Thompson type of an estimator.

### 7.1.2 Reservoir sampling

*Reservoir sampling* solves the problem of variable sample size. The algorithm was popularized among computer scientists by the 1985 paper by Vitter[3]. For any number o elements read from the stream, a sample, selected using *Reservoir sampling,* will be uniformly distributed among all samples of size $k$. Proof for this claim is widely available, so we won't show it here. Consequently, with *Reservoir sampling* we get a SRS strategy of a fixed size $k$ from an infinite landmark stream. Magic!

Reservoir sampling algorithm operates on a data stream as follows: First $k$ elements of a stream are included in the reservoir deterministically (simple append for the first $k$ elements). For every additional incoming element with index $i$ probability of inclusion is $\frac{k}{i}$ for any $i > k$. If a new element with an index $i > k$ is to include, an element currently in the residing in the reservoir is removed uniformly on random, to make place for the newly arrived element. If we add a similar short-cut we used before, to traverse elements by

[3] J.S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software*, vol. 11, no. 1, pp. 37-57, 1985

generating skips between those to include, instead inspecting each, you have all that is necessary to test your understanding of the algorithm on the example given in Figure 7.1. If you are not going to implement the algorithm, but merely use it you can nevertheless apreciate the figure. It depcits reservoir sampling for first 7 elements of the stream, using a reservoir of size $k = 3$.

Before we discuss the runtime of the reservoir sampling algorithm we will describe in detail one possible efficient implementation of this sampling strategy. Those of you who are not intrested in this level of detail, can safely skip the next twoish pages.

Vitter gives an efficient implementation of the algorithm using the same idea of generating the number $\Delta_i$ of skipped elements after the element with index $i$ has just been included in the sample. Notice that here, the number of skipped elements is equipped with an index, hence skips have different distribution depending on how much of a data stream has been seen so far. They change as the stream evolves. Generating such skips is more involved than for Bernoulli sampling, because of the unequal (decreasing) probability of inclusion as the stream evolves. Theory behind it, is not too hard, and can be surveyed either in the original paper or in the section 3.2 by Haas[4] of the manuscript by Garofalakis, Gehrke and Rastogi (we will refer to this work as GGR from now on).

The method makes use of our familiar *inverse probability integral transform* to generate skips $\Delta_i$ for "early" $i$'s . For "later" $i$'s, *acceptance-rejection*[5] method in combination with *squeezing* argument is used.

For the latter we have to know exact functional form of $f_{\Delta_i}$ which is not always the case. Luckily for reservoir sampling it is and Vitter derived it for us, nevertheless it costs to sample from it. More accuratelly, the cost increases as we see more an more elements. Hence we don't want to sample from it. Instead we sample using a different, easy-to-evaluate function and use a probabilistic argument to proclaim that some of the elements sampled actualy come from $f_{\Delta_i}$ indirectly. That's the high level idea.

More specifically, we find an integrable "hat" function $h_i$ , over the range of $f_{\Delta_i}$ which is the probability mass function of $\Delta_i$ . To serve as a "hat" for $f_{\Delta_i}$ we have to have $f_{\Delta_i}(x) \leq h_i(x)$ , meaning that the probability of $\Delta_i$ being $x$ is always smaller than $h_i(x)$. We then normalize $h_i$ with the finite value $\alpha_i$ which is its integral over the range of $f_{\Delta_i}$ to get a valid probability density function $g_i(x) = \frac{h_i(x)}{\alpha_i}$. So it makes sense to sample from the range of $f_{\Delta_i}$ using $g_i(x)$ . We draw a random value $X$ from $g_i(x)$ and a uniformly distributed $U$ from $(0,1)$. If $U \leq \frac{f_{\Delta_i}(x)}{\alpha_i g_i(x)}$ we take $x$ , the current realization of $X$, to be a random deviate from $f_{\Delta_i}$, otherwise we generate the next pair $(X, U)$ until the condition is fullfilled. Abusing the notation and theory heavily, one would say that $g_i(x)$ conditioned on $U \leq \frac{f_{\Delta_i}(x)}{\alpha_i g_i(x)}$ is the same as $f_{\Delta_i}$.

Ok, if you stuck through this, maybe get a cup of your favorite warm beverage as we have one more detail to cover. Remember how we want to eschew evaluating $f_{\Delta_i}$ due to

[4] Peter J. Haas, "Data Stream Management," in *Data Stream Management - Processing High-Speed Data Streams*, M Garofalakis, Gehrke J., and Rastogi R., Eds.: Springer-Verlag, 2016, ch. 3.2.

[5] Von Neumann J., "Various techniques used in connection with random digits. Monte Carlo methods," in *Monte Carlo Method*, A. S. Householder, G.E. Forsythe, and H.H. Germond, Eds. Washington, DC: US Government Printing Office, 1951, vol. 12, ch. 13, pp. 36-38.

its cost ? *Squeezing* introduces "reversed" hat or perhaps a bowl (bathtub?) function that "props" $f_{\Delta_i}$ "from beneath".

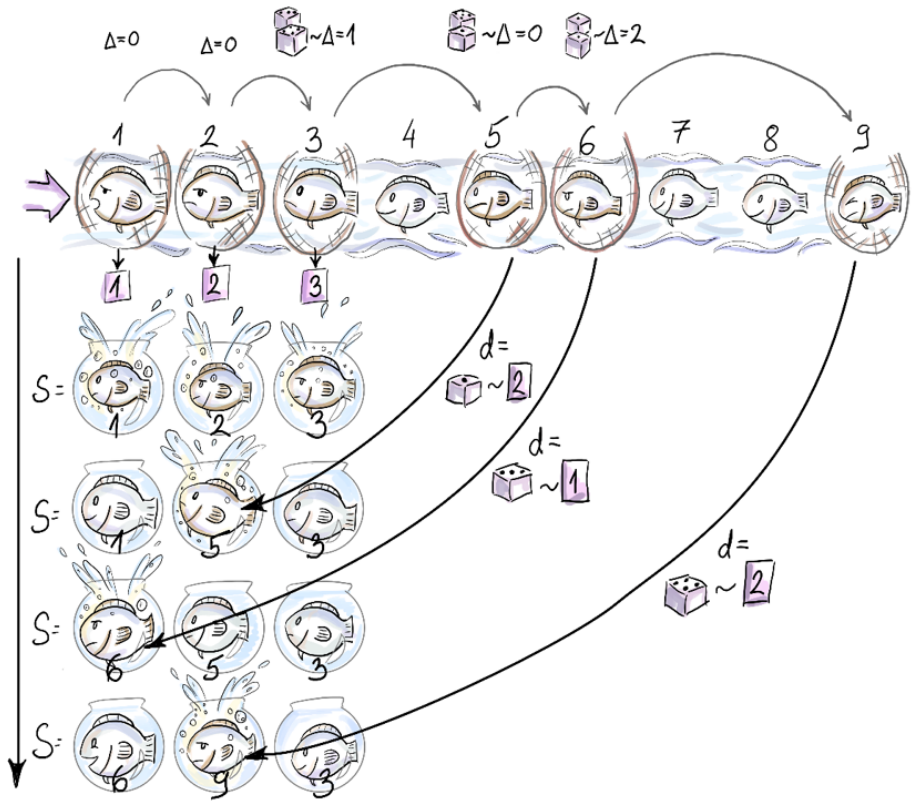



**Figure 7.2 : The figure shows the content of the reservoir after each of the first seven arrivals. First three elements are included in the sample deterministically. Subsequently, $e_4$ is skipped (this corresponds to $\Delta_3$ the number of elements to skip, after $e_3$ is included being 1). $e_5$ is then randomly replaces $e_2$ in the reservoir $(d = 2)$.. Next element, $e_6$ is included at the random position 1 in the reservoir, and replaces $e_1$ (notice that this means that $\Delta_5$ was 0). $\Delta_6$ is definitely bigger than 0, since at least one element, $e_7$, is skipped.**

More accurately, *squeezing* then, is finding a function $r_i$ that is inexpensive to evaluate, such that $r_i(x) \leq f_{\Delta_i}(x)$ for all $x$ in the range. Then asking $U \leq \frac{f_{\Delta_i}(x)}{\alpha_i g_i(x)}$ can be confirmed by the affirmative answer to $U \leq \frac{r_i(x)}{\alpha_i g_i(x)}$ , and only in the case of a negative answer, more expensive $U \leq \frac{f_{\Delta_i}(x)}{\alpha g_i(x)}$ has to be evaluated. We will use $f_{\Delta_i}, g_i, \alpha_i, r_i$ and the cumulative distribution function $G_i$ as Vitter derived them for us. In particular,

$$f_{\Delta_i}(m) = \mathrm{P}(\Delta_i = m) = \frac{k}{i-k} \frac{\prod_{j=0}^{m}(i-k+j)}{\prod_{j=0}^{m}(i+1+j)}$$

$$F_{\Delta_i}(m) = \mathrm{P}(\Delta_i \leq m) = 1 - \frac{\prod_{j=0}^{m}(i+1-k+j)}{\prod_{j=0}^{m}(i+1+j)}$$

$$\alpha_i = \frac{i+1}{i-k+1}$$

$$g_i(x) = \frac{k}{i+x}\left(\frac{i}{i+x}\right)^k$$

$$G_i(x) = 1 - \left(\frac{i}{i+x}\right)^k$$

$$r_i(m) = \frac{k}{i+1}\left(\frac{i-k+1}{i+m-k+1}\right)^{k+1}$$

```
S = [None] * k    #A
j = 0
i = 0
Δ = 0    #A

while (i<k)   #B
    i+=1    #B
    S[i]=e_i   #B

i = i + 1     #C
U = PRNG_Unif(0,1)    #D

while  [1 − ∏_{j=0}^{Δ}(i+1−k+j) / ∏_{j=0}^{Δ}(i+1+j)] > U  :  Δ = Δ + 1    #E

j = k + Δ + 1    #F

while (True):
    if i<=C AND i==j    #G
        U = PRNG_Unif(0,1)
        d = 1 + floor(k*U) #H
        S[d] = e_i
        U = PRNG_Unif(0,1)
        while [1 − ∏_{j=0}^{Δ}(i+1−k+j) / ∏_{j=0}^{Δ}(i+1+j)] > U
            Δ = Δ + 1
            j = i + Δ + 1     #I

    else if i>C AND i==j #J
        U = PRNG_Unif(0,1)
        d = 1 + floor(k*U)
        S[d] = e_i

        U=1
        while U >= f_{Δ_i}(X) / (α_i g_i(X))   #K
            V = PRNG_Unif(0,1)
            X = I * (V**(-1/k)-1) #L
            U = PRNG_Unif(0,1)
```

```
            if U ≤ r_i(X) / (α_i g_i(X))  #M
                break
            #N

        Δ = X      #O
        j = j + Δ + 1      #P
    i = i + 1      #Q
```

#A Initialize an empty buffer (reservoir) S of size k. Set j, index of the next element to include, i index of the current element and the first skip Δ to 0
#B Include first k elements of the stream, e_i is the load
#C Move the index after repeat-block.
#D Draw U for the first skip.
#E Draw first Δ from as $F_{\Delta_1}^{-1}(U)$
#F Calculate the index j of the first element to include.
#G Branch for small indices i.
#H Draw d, the index of the current buffer element to overwrite.
#I Set index for the new element to include.
#J Branch for bigger indices i. ( i > C)
#K Do acceptance rejection with squeezing.
#L Draw X as $G_i^{-1}(1 - V)$.
#M Use $r_i$ instead of $f_{\Delta_i}$ to break.
#N Otherwise check the more expensive $f_{\Delta_i}$
#O Drawn X is the new skip Δ
#P Update j to point to the next element to include.
#Q Move to the next element in the stream.

Pseudo-code given above shows efficient implementation of reservoir sampling. Sample size k should be set to some value. You can go through the code using the example from Figure 7.2 to connect the code to your understanding of the algorithm.

Techniques like *inverse probability integral transform*, *acceptance – rejection method,* and *squeezing* trick are general methods for efficient sampling from any probability distribution, so although one might need a bit of perseverence to understand how they are used, once understood you can tackle a wide domain of sampling tasks efficiently. Notice that the algorithm works for any number of elements coming from the stream, and can be stopped at any time, resulting in a simple random sample of size $k$ of all elements seen up to that point.

Runtime of the reservoir sampling algorithm is $O\left(k + \log\frac{n}{k}\right)$, so time and space requirements definitely fit our "small space – small time constraint", at least in principle since $n$ in a landmark stream is infinite. Better might be, to think about a continuous query which uses the sample, after some specific number of elements has been seen. For the analysis of the difference between the runtime of the naive implementation and the one we presented with geometric jumps, see book by Luc Devroye [6] "Non-Uniform Random Variate Generation" (Ch. 12, pp 639-640).

To put some intuition behind the way reservoir sampling algorithm delivers a SRS we will visualize a very fine balance between two sequences of probabilities. First one is $\frac{k}{i} := p_i$

[6] http://www.nrbook.com/devroye/

probability that $i$-th element is included (we referred to it so far as *inclusion probability)*, Second one is the probability that the $i$-th element is not removed from the reservoir, once it is included, if we see $N - i$ elements after it. The second probability pertains to the event when each $e_j$ (for $i < j \leq N$) that shows up at the "door" after $e_i$ fails to "remove" element $e_i$ that sits in the reservoir.

For one specific $e_j$, this second probability is the sum of the probability that $e_j$ was not selected for inclusion in the first place (remember skips, well $e_j$ is skipped in this case) and the probability that, if it was selected for inclusion, it failed to remove the $e_i$ in particular. This sum is written as

$$q_{ij} = 1 - \frac{k}{j} + \frac{k}{j}\left(\frac{k-1}{k}\right) = 1 - \frac{1}{j}$$

Hence, the probability that $e_i$ is part of the reservoir after N elements are seen is

$$P_N(e_i \in S) = \min\left(1, \frac{k}{i}\right) \quad \times \prod_{j=\max(i,k)+1}^{N} \frac{j-1}{j}$$

We will call this the *probability of residing at N*. First part of the product above (left from $\times$) we denote it as $p_i$, is *inclusion probability* for $e_i$. The second part (right from $\times$) is the cumulative product (we will denote it $\pi_{iN}$) capturing the probability that none of the later elements remove $e_i$ (assuming we see N elements in total). We used $\min$ there to generalize the expression, as to accommodate the first $k$ elements (that are included deterministically), as well. Figure 7.3 shows these "opposing two forces", and the resulting $P_N(e_i \in S)$ for every $e_i$ , and $N = 100$ and $k = 10$.

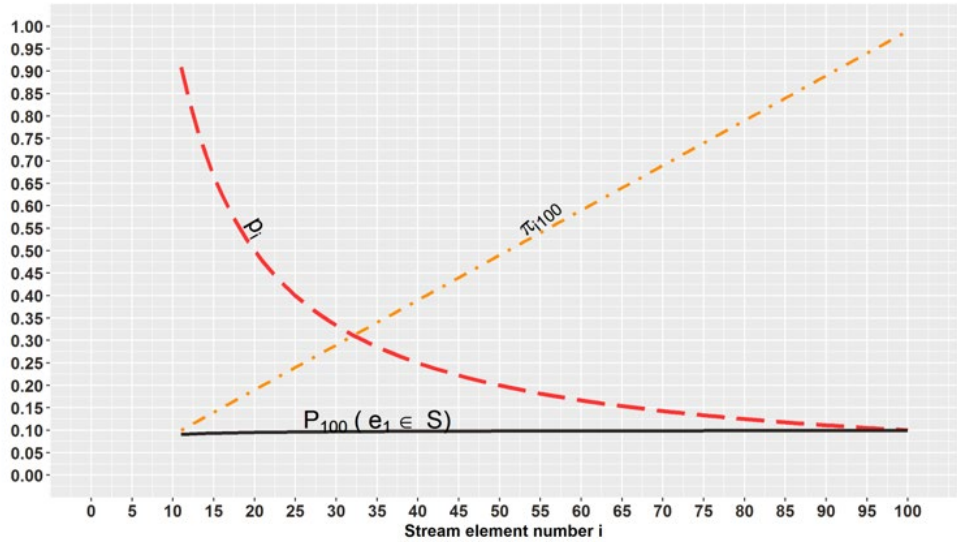



**Figure 7.3 : We can see the balance between probabilities of inclusion (red, longdash line,) and probabilities of "removal up to the time point N=100" (orange, dotdash line) reflected in $P_{100}(e_i \in S)$ (black line) being approximately constant for all elements seen so far (when the reservoir is of size 10). This means that each element, no matter when it has been seen, has the same chance of being in the sample.**

You might remember that there aren't many realistic data streams where we want the distant past to influence our current query to the same degree as the more recent past. This different weighting of elements based on their arrival time obviously cannot be accomplished with *reservoir sampling*. For that purpose researchers try to "tilt" the balance shown in Figure 7.1 in the direction where more recent elements are more probable to be a part of the final sample, compared to those that arrived earlier. This way the black line would go up as it approaches the current moment. This leads us to our next sampling algorithm, *biased reservoir sampling*.

### 7.1.3 Biased reservoir sampling

To bias the sample we will focus on *probability of residing at N,* $P_N(e_i \in S)$, that determines if the element $e_i$ at time *N* resides in the reservoir, having seen $N - i$ elements after it. We would like to be able to tilt the $P_N(e_i \in S)$ from the representative equilibrium where all $e_i$ 's have equal $P_N(e_i \in S)$ (see Figure 7.3). One way to do it, is to assume that $P_N(e_i \in S)$ decreases every time a new element arrives from the stream. We are aging out the elements non-deterministically. This way, when $e_i$ becomes ever more distant past and we would rather not have it in our current reservoir, probability of that happening becomes very small. In our IP example you might be interested in the average number of fingerprints per IP over only the last day of traffic. In this case you would like a mechanism to govern the residing probabilities that will allow a day old elements, but not older than that, in the reservoir.

To achieve this we model the *probabilities of residing* using some memoryless bias function $f(i, N)$. Even though this function has two parameters, $i$ ($i$-th element from the stream), and $N$ (number of elements seen up to now), it evaluates equally for all pairs $(i, N)$ that have the same "distance" $(N - i)$ between them. Therefore, $P_N(e_i \in S) = P_{N+k}(e_{i+k} \in S)$, meaning that $e_i$ after we have seen $N$ elements, has the same probability of residing in the reservoir as $e_{i+k}$ after we have seen $N + k$ elements. We are inquiring about elements that are at the same distance in the past from their two respective querying moments, $N$ and $N + k$. The fact that we saw $k$ new elements in the meantime leaves the two *residing probabilities*, $P_N(e_i \in S)$ and $P_{N+k}(e_{i+k} \in S)$, untouched. This is what is meant when we say *memoryless*. Function does not "remember" what absolute moment in time it is, it just needs how far in the past we are inquiring about.

You can imagine tilting $P_N(e_i \in S)$ (check the graph in Figure 7.3 where fixed $N = 100$ is set). Notice that there $P_N(e_i \in S)$ stays constant for increasing $i$. This is what we expect from a classical *reservoir sampling* algorithm. *Biased reservoir sampling* would have $P_N(e_i \in S)$ larger as we approach the current moment and smaller towards the start of the stream ("beginning of time").

In the original paper for biased reservoir sampling by Aggarwal[7] author makes use of *memoryless exponential bias* function $(i, N) = e^{-\lambda (N-i)}$. Do you notice $(N - i)$ in the negative exponent there? If we observe the expression we notice that, the wider the gap, the bigger the exponent (hence, the smaller the negative exponent). This makes the whole expression smaller, which in fact are our residing probabilities $P_N(e_i \in S)$. For financial data streams, or any stream for which aging elements out is beneficial, this is what we would like.

The parameter $\lambda$ serves as the *speed of aging* factor. Figure 7.4 shows $P_N(e_i \in S) = f(i, N)$ for several different values of $\lambda$. We will equip you with some intuition about $\lambda$. You can convince yourself as a part of an exercise that $\frac{P_N(e_i \in S)}{P_N(e_{i+1} \in S)} = e^{-\lambda}$. In other words, after a single new element arrives, the residing probability of the current element decreases by the factor of $e^{-\lambda}$. Edge case $\lambda = 0$ means "never forget" and this is, for our purpose, a useless value of $\lambda$. It helps to observe what happens near it though. If we part from $\lambda = 0$ to the right in small increasing steps (i.e. $\lambda = 0$ , 0.001, 0.01, 0.1 …) $e^{-\lambda}$ takes up values 1, 0.999, 0.99, 0.9 in that order. Here we see how $\lambda$ governs the *speed of aging*, single new elements arrival makes the residing probability 99.9%, 99% or 90% of what it was before the arrival. We can now deduce from this progression governed by $\lambda$ , how many elements have to arrive for $e_i$ to *age out* completely. Extrapolating the reasoning, we have that $e^{-\lambda (N-i)}$ is the inverse of the number of elements that need to arrive to decrease $P_N(e_i \in S)$ by the factor of $e^{-1}$ (which is multiplication by approx. 0.36). Phew! A mouthful! See Figure 7.4

[7] Charu C Aggarwal, "On Biased Reservoir Sampling in the Presence of Stream Evolution," in *Proceedings of the 32nd International Conference on Very Large Data Bases*, Seoul, Korea, 2006, pp. 607–618.

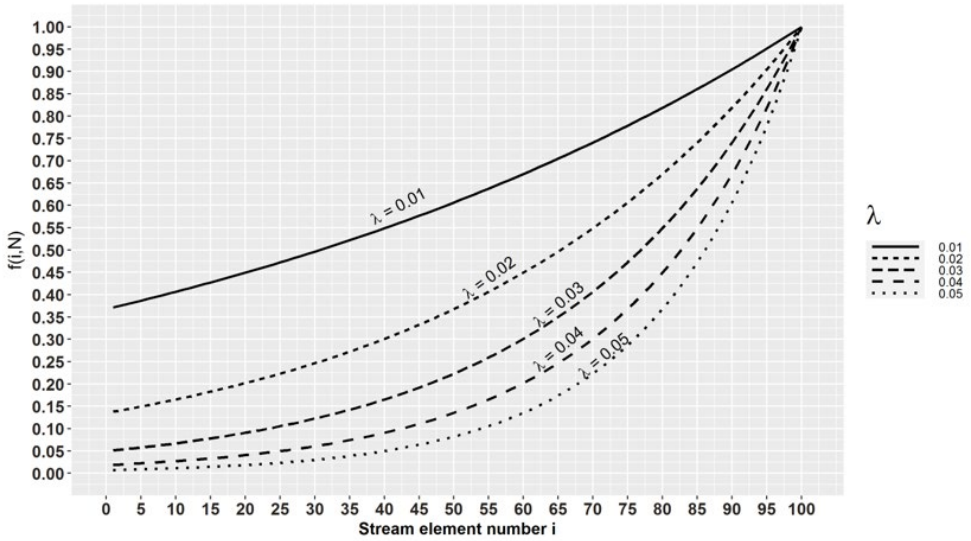



**Figure 7.4 : Notice that for $\lambda = 0.01$ we need 100 elements to reduce f(i,N) by the factor of 0.36, while for $\lambda = 0.02$ this number is 50. So the higher the $\lambda$, easier it is to forget old elements.**

**Example 7.1**

**For our average number of fingerprints per IP – use case, we might have a constant arrival rate of 12 elements per second. We would like to know the average number of fingertips per IPs that appeared in the last 24 hours. That is 86400 seconds in which we see 12 x 86400 = 1036800 IP addresses with their FP. Our $\lambda$ needs to decrease $P_1(e_1 \in S)$ from 1, so that $P_{1036801}(e_1 \in S)$ is effectively 0. This means that after we see 1036801 elements the residing probability of the first one has to fall to 0, effectively. For our particular application, what would be the value of $\lambda$ that would make this happen for us? This is equivalent to the question for which $\lambda$ is $e^{-\lambda x\, 1036800} = 0$ effectively. By trial and error you can check that for $\lambda = 10^{-6}$ $e^{-\lambda x\, 1036800}$ is 0.35 while for $\lambda = 10^{-5}$ this becomes $3 \times 10^{-5}$. So between these two values is where our $\lambda$ would be if we want to "cling-out" elements gradually over a day.**

We will now see how one such particular $\lambda$ governs the sample size too. This biased sampling scheme does not come for free, and to maintain a sample over a landmark stream, we have to meet some minimal space requirements; we have to know how the sample size grows with elements seen. Authors of the original paper denote the sample $S(n)$ to indicate its dependency on the number of seen elements. Conveniently, they also prove that for large $N$ (conforming with realistic streams) the size of the sample is bounded above by $\frac{1}{1-e^{-\lambda}}$. From that, we bound the maximal sample size that is needed to achieve rates $\lambda$ at which $P_N(e_i \in S)$ reduces appropriately slow (fast). That first bound can be replaced by $\frac{1}{\lambda}$ (using a basic calculus theorem). So "all" we have to facilitate is enough space for our specific, application driven $\lambda$ to govern our bias. If the $\lambda$ that we calculated in the example above, fits

this constraint, we can use the exponential bias function with that $\lambda$. In our case maximum sample size would be somewhere between $10^4$ and $10^5$. For the case where we can hold the entire maximal size of the sample within the (efficient) space constraints of the stream application, we can use the following, simple algorithm for maintaining a biased sample over any number of elements from the stream.

Assume that $j$-th element in the stream just arrived, denote the "occupied" proportion of the reservoir by $F(j) \in (0,1)$. The new element $e_j$ is added to the reservoir deterministically. This can happen in two ways though: with probability $F(j)$, $e_j$ substitutes a randomly chosen element from the reservoir, while, with the complementary probability, $e_j$ is just appended to the reservoir without resulting in any removals. Pseudo-code for this version of biased reservoir sampling is shown beneath.

Figure 7.5 shows biased reservoir sampling for a reservoir of size $k = 3$ $\left(\lambda = \frac{1}{3}\right)$ for first 7 elements of the stream.

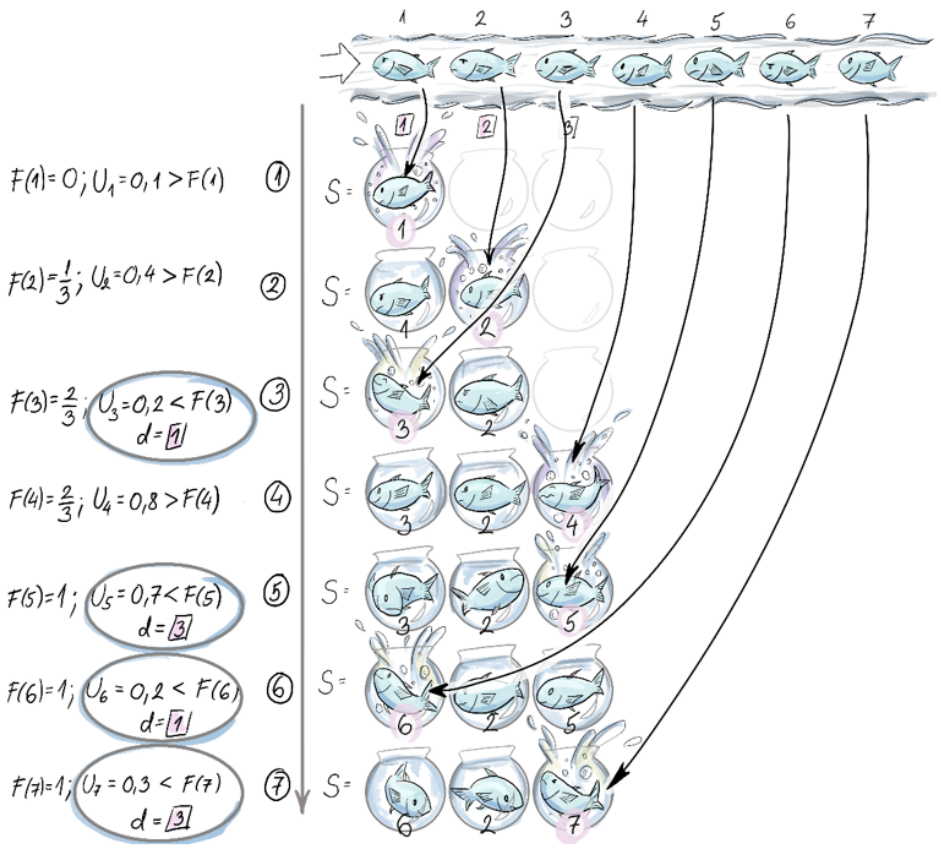



**Figure 7.5: The figure shows the content of the reservoir for first 7 arrivals under the the *biased reservoir sampling* strategy.**

$e_1$ is included deterministically, $e_2$ is inserted without removing any existent elements from the reservoir, due to the values $F(2)$ and $U_2$. Since the portion of the reservoir that is occupied grows, the probability of including the next element "at the expense" of one existent is higher ($U_3 < F(3)$), hence $e_3$ is saved at the first spot (d=1). Notice that there were two possible positions for this, 1 and 2. $e_4$ is included at the third position without removing any elements ($U_4 > F(4)$). Since at this point the whole reservoir is occupied, elements $e_5$ , $e_6$ and $e_7$ are included all at the expense of elements saved in positions 3,1 and 3 in that order.

```
S = [None] * 1/λ    #A
COP = 0 #A
i = 1    #A

while (True)
    U = PRNG_Unif(0,1)
    if U < COP
        U = PRNG_Unif(0,1)
        D = 1 + floor(k * COP * U) #B
        S[d] = e_i #C
    else
        d = 1 + floor(k * COP) #D
        S[d] = e_i
        COP += 1/k #D
```

**#A Initialize an empty buffer (reservoir) S of size k = 1 / λ. Set i, index of the current element to 1 and currently occupied proportion (COP) of the reservoir to 0.**
**#B Draw index d between 1 and the maximal occupied index k x COP of the reservoir.**
**#C Save the element e_i at that random index in the occupied part of the reservoir.**
**#D Append the current element to the reservoir. In this case we have to update the COP too.**

You can go through pseudo-code using the example from Figure 7.4. to check your reasoning. Notice that once the reservoir is full, COP is 1 and the IF-branch is always executed after that.

**Exercise 7.1**

**Implement biased reservoir sampling using the above pseudo-code and the R package `stream` introduced beneath in section 7.3 or Python 3.0.**

In the case when $\frac{1}{\lambda}$ cannot fit in our available working buffer memory, the algorithm is modified to "slow down" the insertions by introducing $p_{ins} = k\lambda$ probability of insertion instead of $p_{ins} = 1$. This allows us to implement same bias in the sampling, but with a lower sample size $\frac{p_{ins}}{\lambda}$.

This modification introduces the issue of too slow initial filling of the reservoir. This can lead to long waiting times to answer queries with the sample size that guarantees accuracy and precision standards acceptable for our practical purpose. Author gives a strategy how to solve this problem so one should turn to the original paper, if necessary, to implement this.

We will now move on to sampling from a *sliding window*.

## 7.2 Sampling from a sliding window

We will first discuss how to sample from a sequence-based window. Here the recency is measured in ordinal sense as a number of arrived elements. In our IP addresses stream, the IP addresses could come at different times, but the window would be 1000 of them long, no matter what that 1000 spans on the time-scale. We will be dealing with a sequence of

windows, (hence, sliding) $W_j$, $j \geq 1$ where each indexed window, entails $n$ elements $e_j, e_{j+1}, e_{j+2}, \ldots, e_{j+n-1}$. This $n$ does not change with stream evolution as $N$ does. The gaps between two arrivals can generally be different in absolute time units, but sequence-based window deems these irrelevant and logs the elements in consecutive integer positions. We will be maintaining a sample of size $k$ from the current window. Notice that we now have to devise a strategy for updating the sample not only when we decide to insert a new element in the sample, but also every time when the "oldest" current member of the sample exits the current window (is "aged out"). We don't assume that the size of the window $n$ can fit our working memory, therefore it makes sense to sample from it.

### 7.2.1 Chain Sampling

First we explain how to select a random sample of size 1 from the current window and update it as the window moves. The algorithm for a random sample of size $k$ is then just simultaneous (parallel) execution of $k$ instances (chains) of the strategy for keeping just one random element.

The initial phase of the algorithm that lasts $n$ discrete time steps (length of the window) is the regular unbiased reservoir sampling with some additional operations. Each arriving element $e_i$ will be selected as the sample $S = \{e_i\}$ with the probability $\frac{1}{i}$. This is the reservoir sampling part. The addition that handles sliding window part is to pick the element that will substitute $e_i$ `when this one is aged out`. We did not do this in the reservoir sampling algorithm. Hence, each time, we pick a random future index $K \in \{i+1, i+2, \ldots, i+n\}$ and add $(K, .)$ to the chain. We know that $K$-th element will be saved there, once it enters the window. This becomes the second element of the chain. First element $(i, e_i)$ saves the current sample of size 1. After the whole window $W_1$ has been seen ($n$ elements pass), we are in possession of a simple random sample from $W_1$ of size 1, because reservoir sampling guarantees that. In addition we have the latest $K$, the index of the tuple that will replace it, once the sample of size 1 expires. For each arriving element now $i = n+1, n+2, \ldots$ we have two options:

1. With probability $1/n$ we discard the current sample $S = \{e_j\}$ and its associated chain saving the index $K$ and element $e_K$ that was supposed to "inherit" it once $e_j$ expires.

   We replace it with $S = \{e_i\}$. Now $e_i$ has to have a successor to "take over" after $e_i$ expires, hence we sample a random future index $K \in \{i+1, i+2, \ldots, i+n\}$, and add it as the second element in the newly created chain.

2. With probability $(1 - 1/n)$ we check if $i$ is the the next replacement element to be saved in the chain $(K = i?)$. If so, we save the $i$-th tuple into the (last) chain element. We sample a random future index $K \in \{i+1, i+2, \ldots, i+n\}$ of the element that will replace $e_i$ once it expires, and add it at the end of the chain. This is how the chain grows. In case $i = j + n$, meaning $e_j$ is leaving the window, the second element in the chain moves up, while the expired sample $S = \{e_j\}$ is removed from the top of the chain.

The above delivers, at every discrete window-update "moment" $i$, a simple random sample of size 1 from the window $W_{i-n+1}$. Figure 7.5 shows *chain sampling* for first 7 elements and the size of the window $n = 3$.

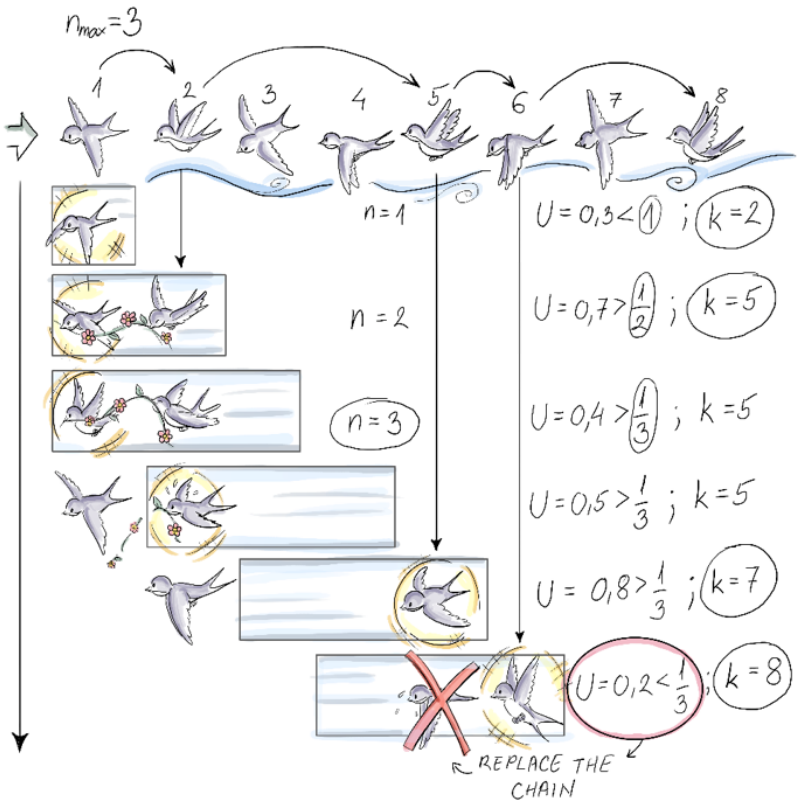



**Figure 7.6 : Figure shows the content of the chain (list L) for the first 7 elements from a sequence-based windowed stream of size n=3.**

You should try to follow the Figure 7.6 as you read. First element $e_1$ is included deterministically. We then pick a future index $K$ that will replace $e_1$ when $e_K$ arrives. Well $K$ seems to be 2. After we finish with the first element, the chain entails $(1, e_1)$ and $(2, .)$. At that moment $e_1$ is the random sample of size 1. But to continue our example, the element 2 arrives and we notice that it is supposed to be saved as successor of $e_1$. This is immediately done, and the chain now saves $(1, e_1)$ and $(2, e_2)$. These successor bookkeeping operations are done even before we decided, if we will sample $e_2$ with probability ½ (reservoir sampling) and discard $e_1$ and its chain altogether. We throw a die and $U > \frac{1}{2}$ (in our case $U = 0.7$), hence $e_2$ does not cause us to discard the existent chain. To finish dealing with $e_2$ its successor is

drawn and it turns out to be $K = 5$. The current chain is hence $(1, e_1)$, $(2, e_2)$ and $(5, .)$. $e_3$ will not cause the deletion of the existent chain either since $U_3 > \frac{1}{3}$. Since $e_3$ is nobody's successor, it won't be included in the chain and we move on. At the 4th arrival, since the length of the window is $n = 3$, $e_1$ expires. We first need to update the current sample element with its successor in the chain. The role is taken over by $e_2$. $e_4$ does not cause us to discard the existent chain ($U = 0.4 > 1/3$) and it was not chosen as anyone's successor, so we move on. Notice that $e_4$ is the first element in the second non-reservoir sampling phase. At the 5th arrival, two things happen. First, $e_5$ is added to its designated position in the chain, and its successor index is also included, $= 7$, so currently the chain is $(2, e_2)$, $(5, e_5)$ and $(7, .)$ and it is at the "peak" of its length. Second, since $e_2$ expires, next element in the chain, the newly added $e_5$ replaces it. Once we finish with $e_5$ we have $(5, e_5)$ and $(7, .)$ as the current chain and $e_5$ as the current sample of size 1. When $e_6$ arrives, we randomly choose to discard the current sample and start a new one ($U = 0.2 < 1/3$). The current chain together with the sample is discarded, $e_6$ is added to the new chain and becomes the new sample. Index of its successor $K$ is drawn, which is 8 in our example above. $e_7$ does not cause the chain to be discarded, and since it is not anyone's successor (it was $e_5$'s, but that chain was disbanded) we move past it. At each moment of the stream evolution you can read off two important points from the figure 7.5, a) what is the sample of size 1 (*the shiny sparrow*) and which sliding window does it represent (the window it is in). The sample is always the top element of the list.

Extensively commented pseudo-code for selecting a sample of size 1 using *chain sampling* is shown beneath. You can follow the code through Figure 7.5 and see where the chain gets longer, and when its elements are discarded to start a new chain.

```
L = []    #A
i = 1    #A
K = 0    #A

while i<=n #B
    U = PRNG_UNIF(0,1)
    if U < 1/I #C
        L.clear() #D
        L.append([e_i, i]) #E
        U = PRNG_Unif(0,1)
        K = i + floor(n*U) + 1 #E
    else
        if i == K #F
            L.append([e_i, i])
            U = PRNG_Unif(0,1)
            K = i + floor(n*U) + 1
    i+=1

while True #G
    if (i==j+n)
        L = L.pop(1) #H
    U = PRNG_Unif(0,1)
    if U < 1/n
        L.clear()
        L.append
        U = PRNG_Unif(0,1)
```

```
        K = i + floor(n*U) + 1
    else if i==K
        L.append([e_i, 1])
        U = PRNG_Unif(0,1)
        K = I + floor(n*U) + 1
    i+=1
```

#A L is empty at the start. Set i index of the current element to 1. Set K, the index of the future replacement element to 0.
#B First phase with n elements.
#C **Reservoir sampling decides to keep** $e_i$
#D Remove the current sample and its chain.
#E Add the current element e_i to the chain and determine its successor K.
#F Reservoir sampling decides to skip e_i, so we check if it is anyone's successor.
#G Second phase for i=n+1, n+2, ...
#H Remove the top element of the list because it presently expires from the window.

**Exercise 7.2**

Implement chain-sampling for a window of length 100, sample size 1 and any N > 100.

Analysis of the space complexity for keeping $k$ independent chains can be found in the original technical report by Babcock, Datar and Motwani[8] or in GGR[9] . Expected memory consumption for $k$ chains is $O(k)$, meaning that all of them have at most length bounded by a constant. The space complexity of the algorithm does not exceed $O(k \log n)$ with probability $1 - O(n^{-c})$, hence by our criteria it is *efficient*.

Notice that each chain in the *chain sampling* algorithm delivers at each time-point a *simple random sample* ***without*** *replacement* of size 1 from the current window. Nevertheless, when we maintain $k$ parallel chains at a time, the algorithm will deliver a *simple random sample* ***with*** *replacement* of length $k$. This is not a limiting factor though.

We will now present a similar algorithm by the same authors for keeping a sample of size $k$ from a timestamp-based sliding window.

### 7.2.2 Priority Sampling

When we are dealing with timestamp-based windows we don't know the exact number of elements $n$ in the window, so it is not possible to anchor our algorithm on that parameter. To keep a simple random sample (SRS) of size 1 over a timestamp-based window, we generate a priority $p_t$ for each arriving element $e_t$ as a uniform draw from the interval (0,1). The element with the highest priority in the window ($now - \omega < t < now$) is our SRS of size 1. As in chain-sampling we will be keeping successors to inherit the sample, once the current one exits the time-based window.

[8] Brian Babcock, Mayur Datar, and and Motwani Rajeev, "Sampling From a Moving Window Over Streaming Data," in *2002 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, 2002, pp. 633-634.
[9] Peter J. Haas, "Data Stream Management," in *Data Stream Management - Processing High-Speed Data Streams*, M Garofalakis, Gehrke J., and Rastogi R., Eds.: Springer-Verlag, 2016, pages 30-31

First element $e_{t_1}$ becomes our sample deterministically, since there is no priority to beat $(p_0 = 0)$. When the second element $e_{t_2}$ arrives at time $t_2$ , we check if $p_{t_2} > p_{t_1}$ , if true, $e_{t_2}$ replaces $e_{t_1}$ ($e_{t_1}$is removed from memory). Otherwise, $(e_{t_2}, p_{t_2})$ is saved in a linked list as the first element of the list (sample is saved separately and is not an element of the list). After $e_{t_3}$ arrives there are three different scenarios for ordering the priorities in memory $p_{t_1}$ , $p_{t_2}$ and $p_{t_3}$ of the newly arrived element $e_{t_3}$ :

1. $p_{t_1} > p_{t_2} > p_{t_3}$ : $e_{t_3}$ is added to the tail of the list that keeps replacements in case the element $e_{t_1}$, that is the current samples, expires. The list is ordered by descending priority and, and per creation, ascending time.
2. $p_{t_1} > p_{t_2} < p_{t_3}$ : $e_{t_3}$ is added behind $e_{t_1}$, while all the elements (currently this only includes $e_{t_2}$ ) with lower priority and (inevitably) lower timestamp are removed from the list / memory. The list remains ordered by descending priority and, and per creation, ascending time.
3. $p_{t_3} > p_{t_1} > p_{t_2}$ : $e_{t_3}$ is added at the beginning while all other elements are removed from the list / memory. The list remains ordered by descending priority and, and per creation, ascending time.

The first case adds the new element at the end of the (by priority) sorted list and keeps the whole list. Third case discards the previous list and adds the new element at the top of the new one. The second case keeps a part of the list that has a higher priority than the new element. The rest are discarded and the new element is saved last.

The algorithm continues to update the list with each arriving element in the manner described by one of the three possible cases. At each moment $l$, on top of the list is the element $e_t$ with the second highest priority among elements from the window $W_{l-\omega}$, where $\omega$ is the duration of the window $(l - \omega < t < l)$. Current sample is the element with highest priority within the time $t$ $(l - \omega < t < l)$. Element from the top of the list substitutes the current sample, and becomes the new SRS of size 1, once the current sample exits the timestamp-based window. Figure 7.6 shows priority sampling for 6 elements arriving at indicated times $t_i$'s for a window size of 600 ms.

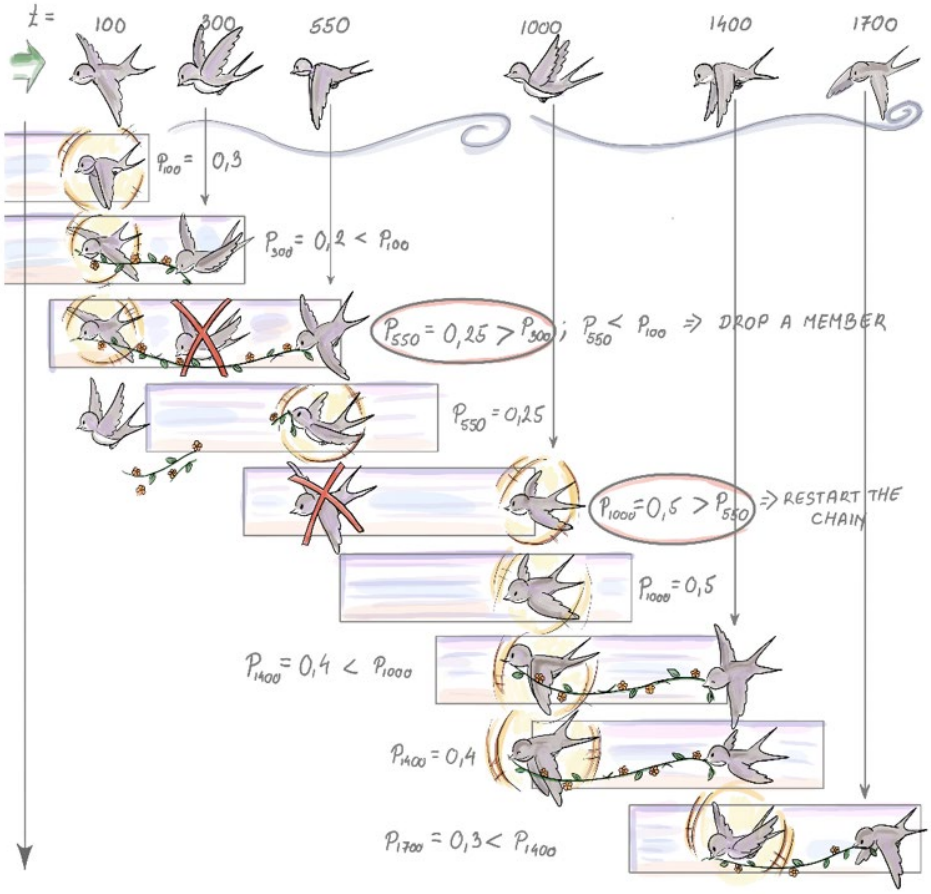



**Figure 7.7 : Figure shows content of the list of successors and the current sample for first 6 arrival times in ms for a timestamp-based window of length 600 ms.**

We will explain what happens in Figure 7.7 as time passes by, so maybe it would be good to keep it in front of you while you read. First element arrives at 100ms and $p_{100}$ is drawn to be 0.3. $e_{100}$ is set to be the current sample, while the list with the successors stays empty. At 300ms, when $e_{300}$ arrives, its priority $p_{300}$ is set to 0.2. Since it is smaller than the priority of the current sample, it is added in the list as the first element, together with its priority and its time of arrival. Upon arrival, $e_{550}$ will "break" the priority list there, where elements that have lower priority ($p < p_{550} = 0.25$) start. This causes $e_2$ to be discarded from the list. The new content of the list is $e_{550}$ only, together with its priority and its time of arrival. This probabilistic type of trimming of the list makes sure that the priority lists doesn't grow too large.

But to continue, since $p_{550} < p_{100}$, the new element does not substitute the current sample, it just becomes its new and only successor for now. At time 700 ms no element arrives, but we have to replace the current sample with its successor since $e_{100}$ expires. $e_{550}$ becomes the current sample and the list is empty. At 1000ms the new element arrives. Its priority is 0.5, which is higher than the priority of the current sample. This causes $e_{550}$ to be removed and replaced by $e_{1000}$ together with its priority and its time of arrival. List with successors remains empty. $e_{1400}$ does not have a higher priority than the current sample, hence it is added into the list as the only element so far. At 1600ms current sample $e_{1000}$ expires and is inherited by $e_{1400}$. The list is now empty again. When $e_{1700}$ arrives, its priority is set to be 0.3, so it is lower than $p_{1400} = 0.4$. . Therefore, $e_{1700}$ is added to the list as the first successor of $e_{1400}$ once it expires.

The pseudo-code and detailed comments for *priority sampling* of sample of size 1 are given beneath. This solution will work if the inter-arrival time between any two elements of the stream is always less than ω, the window length.

```
L = []    #A
i = 1    #A
p = 0    #A

while True:
    if len(L) == 0    #B
        p = PRNG_unif(0,1)
        t = t_i
        L.append([e_ti, p, t_i ])
    else if ( t_i − t ≥ ω )
        L = L.pop(1)    #C
        p = PRNG_unif(0,1)
        if p ≥ L[1][2]    #D
            L.clear()     #E
            t = t_i    #E
            L.append([e_ti, p, t_i ])     #E
        else    #F
            j = 0
            while p ≤ L[j][2]
                j+=1     #G
                L = L[0:j]    #H
                L.append([e_ti, p , t_i])    #H
    i = i + 1    #I
```

**#A L is empty at the start. Set i index of the current time point to 1. Set p the priority before any elements are seen to 0.**
**#B Handle the first ever element.**
**#C If the current sample element expired at time $t_i$ remove the top first element of the list**
**#D If the element that just arrived has a higher priority than the top first element of the list.**
**#E Empty the list and append the new element as the only member of the list. Update the time t of the current sample.**
**#F We have to break the priority list somewhere "beneath" the top first element.**
**#G Find the position where to break the list and discard the "tail".**
**#H Discard the "tail" and append the current element in its place.**
**#I Move to the next timestamp (arrival time).**

The expected number of elements stored in memory for this strategy at any given time is $O(\ln n)$. To maintain a sample of size $k$ we can keep $k$ lists, assign $k$ priorities $p_{t_{i1}}, p_{t_{i2}}, p_{t_{i3}}, \ldots, p_{t_{ik}}$ to each arriving element $e_{t_i}$ and repeat the algorithm as many times with $e_{t_i}$ as there are lists. For this algorithm the expected memory cost is $O(k \log n)$, while, with high probability, the cost does not exceed $O(k \log n)$ (the space complexity analysis can be found in the same references as for *chain sampling*).

Notice that the algorithm with k lists delivering sample of size k, generates a *simple random sample* ***with replacement*** from a timestamp-based window.

To try out sampling from the stream in practice, before we get to actual implementation of the sampling algorithm, we first have to have a framework to handle data streams as some objects. Setting up such an environment yourself using low level OS functions or even using special R or Python libraries to communicate with a streaming framework like Apache Kafka can be quite an overhead. Especially if you are someone who is just trying to quickly check if her streaming algorithm works as it is supposed to. In the next section we show how to **use** these algorithms in R programming language within a simple data stream framework. We will spare ourselves some ground work with the help of the R-package `stream`. We give some reference for similar framework in Python.

## 7.3 Sampling algorithms comparison

Now that we have gotten to know few algorithms for sampling from a stream, we will demonstrate how one would go about using some of these algorithms in R programming language and in particular R-package `stream`[10].This puts at your disposal a data streams framework with *data stream data objects* (DSD). These can be wrappers for a real data stream, for data stored in memory or on disk, or a generator which simulates a data stream with known properties for controlled experiments. Once we define what kind of data we will receive from the *data stream data object*, we implement *the task*. In our case, this will be to maintain a random sample over the stream and to use it to estimate the average value. For this we will be using the class *data stream task* (DST). For more detailed introduction to `stream` previous reference should get you far.

### 7.3.1 Simulation set up, algorithms and data

We will compare how well biased and unbiased sampling strategies adapt to sudden and gradual changes in the data stream. Algorithms in comparison are *reservoir sampling* and *biased reservoir sampling*. We will generate a stream with a sudden change in concept to check the robustness of the sampling algorithms with respect to this characteristic of a stream. The two algorithms operate on landmark streams, so we can talk about a random sample of size $k$, from what we've seen so far. Biased reservoir sampling lays more weight on more recently seen elements, and it depends on the bias function and the parameter λ how fast the "older" elements are *aged out*.

To simulate sudden change in concept we create a stream with the help of the function `DSD_Gaussians()`. This generator of normally distributed data creates $10^6$ Gaussian

[10] Michael Hahsler, Matthew Bonalos, and John Forrest, "Introduction to "stream" : An Extensible Framework for Data Stream Clustering Research with R," *Journal of Statistical Software*, vol. 76, no. 14, pp. 1-50, 2017.

deviates. The observations from the data stream change their distribution from $N(1,1)$ to $N(3,1)$ in single step. This means that the stream source simulates sudden shift at one point. We have split the stream in half for this purpose. We receive 500K random values from $N(1,1)$ followed by 500K random values from $N(3,1)$. We will try out two sample (reservoir) sizes $\in \{10^4, 10^5\}$.

We first create the stream, then save it as a `csv` file permanently. This is done so that we can sample the same data with our two algorithms. The code for generating the stream with sudden shift and saving the data from it is given beneath.

```
rm(list=ls()) #A
if (!'stream' %in% installed.packages()) install.packages('stream')    #A
library(stream)    #A

setwd(" ")    #B
set.seed(1000)    #C
stream_FirstHalf <- DSD_Gaussians(k = 1,    #D
                                  d = 1,
                                  mu=1,
                                  sigma=c(1),
                                  space_limit = c(0, 1)
                                  )

write_stream(stream_FirstHalf, "DStream.csv", n = 500000, sep = ",")    #E

stream_SecondHalf <- DSD_Gaussians(k = 1,    #F
                                   d = 1,
                                   mu=3,
                                   sigma=c(1),
                                   space_limit = c(0, 1)
                                   )

write_stream(stream_SecondHalf, "DStream.csv", n = 500000, sep = ",", append=TRUE)    #G
```

**#A Remove possibly leftover objects in the workspace. If package 'stream' is not installed already, install it. Then bind the package to the workspace.**
**#B Choose the path where you would like DStream.csv with the data to be saved.**
**#C Set the starting position of a PRNG so that the same random data is created every time.**
**#D Make a Data Stream Data object for the first half of the stream.**
**#E Write 500K elements from the Data Stream Data object to the file DStream.csv.**
**#F Make a Data Stream Data object for the second half of the stream.**
**#G Append 500K elements from the Data Stream Data Object to the file DStream.csv.**

Implementation of the biased and unbiased reservoir sampling algorithms presented in this chapter are available in the package `stream` in the form of the function `DSC_Sample()`. In our simulations we will use two different sample sizes, 10K and 100K elements. We load the stream from the file using the `DSD_ReadCSV` class. The stream is processed in batches of 100K elements, hence the whole stream is processed in 10 steps. In each step we call the function `update(CurrentSample, stream_file, n=100000)`. It expects a Data Stream Mining Task - object, Data Stream Data - object and the number of new elements to read from the stream. The paradigm behind `update()` is that there is a task that we are executing

on the stream. In our case it is sampling from the stream. Once we read 100K new elements from the stream, we have to adjust the current sample accordingly. Hence, calling `update()` makes sure our sample object has integrated the new 100K elements into its current state. Since we call `update()` ten times, we have ten snapshots of the sample, each after additional 100K elements have been seen. At these ten stops we will calculate the average of the current sample and save it. We will use this later to evaluate how well the samples from biased and unbiased reservoir sampling adjust their average to the sudden shift in the average of the stream data. We repeat this scenario for biased and unbiased reservoir sampling with sample sizes 10K and 100K for both. Remember that for biased reservoir sampling, $\lambda$, speed of aging factor is reciprocal of the sample size. The code for unbiased sampling and two sample sizes is given beneath.

```
rm(list=ls())
if (!'stream' %in% installed.packages()) install.packages('stream')

stream_file <- DSD_ReadCSV("DStream.csv")    #A
CurrentSample <- DSC_Sample(k=10000, biased=FALSE)    #B


MeanResults_Size10K <- NULL     #C

for(i in seq(1,10)){
        update(CurrentSample, stream_file, 100000)    #D

        names(CurrentSample$RObj$data) <- "sample_so_far"    #E

        current_sample_avg <- mean(as.numeric(CurrentSample$RObj$data$sample_so_far))  #F

        MeanResults_Size10K <- c(MeanResults_Size10K, current_sample_avg)    #G
}

reset_stream(stream_file, pos=1)    #H

CurrentSample<-DSC_Sample(k=100000, biased=FALSE)
MeanResults_Size100K<-NULL
for(i in seq(1,10)){
  update(CurrentSample, stream_file, 100000)
  names(CurrentSample$RObj$data) <- "sample_so_far"
  current_sample_avg <- mean(as.numeric(CurrentSample$RObj$data$sample_so_far))
  MeanResults_Size100K<-c(MeanResults_Size100K, current_sample_avg)
}
close_stream(stream_file)    #I
```

**#A Create a Data Stream Data - object stream_file from our DStream.csv.**
**#B Make a Data Mining Task – object implementing reservoir sampling with the option "biased" set to FALSE and sample size 10K**
**#C Empty vector to save 10 averages from the 10 consecutive snapshots of the sample.**
**#D Update the sample with 100K new elements.**
**#E Rename the variable where the sampling object saves the sample to something more informative.**
**#F Calculate the sample average.**
**#G Save the sample average in the vector.**
**#H Reset the stream for unbiased reservoir sampling with the sample size 100K.**
**#I Close the Data Stream Data object.**

`CurrentSample` is an object of a `DSC_Sample` class which is a sub-class of the *data stream task* (DST) class. One can therefore use `DSC_Sample` class to implement any sampling strategy as a task on a data stream. Code for the biased version of the reservoir sampling is identical, with the parameter `biased` set to `TRUE`. The $\lambda$ parameter governing the bias towards new arrivals is $1/k$ in that case.

Figure 7.8 shows sample averages during the evolution of the stream for these two sampling strategies. We can see how sample average changes for *reservoir sampling* and *biased reservoir sampling* and the reservoir sizes $k = 10^4, 10^5$ on a landmark stream with the sudden shift at $i = 500K$.

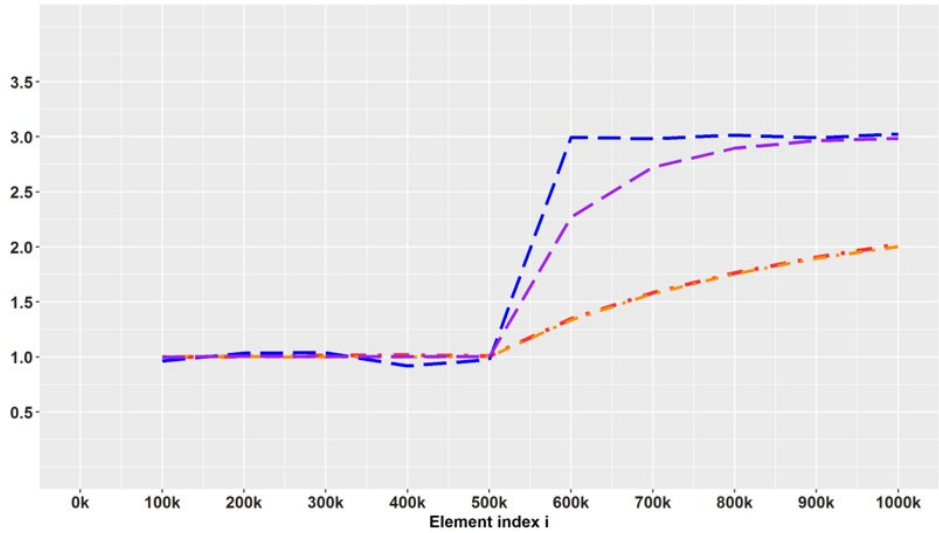



**Figure 7.8 Dashed lines (purple and blue) connect sample averages of samples selected using biased reservor sampling. Blue (upper) with $\lambda = 10^{-4}$ and $\lambda = 10^{-5}$ for the purple (lower) dashed line. Dot-dashed lines (red and range) track sample averages at every 100k new elements for two unbiased reservoir sample sizes.**

We see how biased reservoir sampling due to uneven weighting of the recent and distant past adapts to the sudden shift and estimates the mean unbiaseadly very quickly after the shift, while the unbiased *reservoir sampling* fails to do this. How quickly the biased strategy can "detect" the shift depends on the parameter $\lambda$ . For $\lambda = 10^{-4}$ the probability of remaining in the sample decreases by $e^{-1}$ every 10000 elements, so this $\lambda = 10^{-4}$ "forgets" faster or reaches shorter in the past. Therefore, the biased sample with $\lambda = 10^{-5}$ is slower in moving towards the current true mean. This simulation hopefully gave you some sense about realistic conditions under which you would be sampling from a stream, and decisions you need to make given the data at hand.

Aside from R package `stream`, for those of you who would like to try out sampling from the stream using Python, there are two options familiar to this author. First one is more lightweight, and allows you to deploy a simple Python-based Kafka producer that reads from

a .csv file of time-stamped data. The repo can be found on GitHub under MIT license (github.com/mtpatter/time-series-kafka-demo). Second Python resource is Faust - a library for building streaming applications in Python (faust.readthedocs.io/en/latest). This is a very well documented and rich library better suited for production level processing of data streams. This might be too much, if you just want to convert a csv file of time-stamped data into a real-time stream useful for testing your sampling algorithm.

## 7.4 Summary

- We have gotten to know five algorithms for sampling from a data stream (three for landmark streams and two for windowed streams). Bernoulli sampling is a very simple and representative sampling algorithm, but if you want to use it, you have to think about some sample size curbing strategy, without introducing much bias.
- Reservoir sampling solved our problem of variable sample size and delivered a SRS from elements seen so far at any moment. If we would want to accentuate more recent elements in our sample, biased reservoir sampling would be one way to go. Here, we need to think about our desired speed of aging factor and how it relates to our available space. This depends on the realistic parameters that you face in your own application.
- The other option to accentuate recently arrived elements in the stream is via *sliding window*. If we decide that we need to sample from that window due to its size, we learned how to implement chain sampling for sequence-based windows and priority sampling for time-based windows.
- We saw how biased reservoir sampling in our simulation managed to adjust to sudden shift in concept, while reservoir sampling was unable to react to this change, and led to biased answer of the query about the current / recent true average of the stream data. You do need to adjust the speed of aging parameter though, so that it suits notion of *sufficient recency* for your particular use-case.

# 8

# *Approximate quantiles on data streams*

**This chapter covers**

- **Reviewing the concept of exact *quantiles* and understanding constraints imposed by streaming data context**
- **Understanding different types of errors for *approximate quantiles***
- **Applying T-digest and Q-digest algorithm to a data stream**
- **Comparing T-digest and Q-digest on a realistic web-site-time-spent data**

Different algorithms presented in the previous chapter allow us to select an (un)biased sample of from all data-tuples that arrived up to the current moment. In a way, a sample is a very flexible data-sketch: you form it once and you can then use it to claim that its mean, or any other feature is a good estimate of that same feature of the whole data that came from the stream so far. Now, remember why we moved from Bernoulli sampling to sampling procedures that keep a sample of fixed size? Yes, good job, it is because Bernoulli sample grows with the number of elements seen and in combination with streaming data, this is just not practical.

What you get with Bernoulli sampling though, and don't get with others is a constant sampling rate for any value of N. You take on average every 1/p-th element and this won't change no matter what N is. You "pay" for this nice property by having an average size of the sample pN. Why is it good to have the sample size grow? The Central Limit Theorem says that any estimator from our sample has a standard error (precision) that decreases with the square root of the sample size. So statistically, it is good to have a higher sample size. The problem is, as we see more and more elements from the stream, *density* of the sample does not change for Bernoulli sampling, but it does for others. For algorithms that keep fixed

sample size k, the density of the sample inevitably diminishes, since we always have k / N density, while N grows.

This *density,* loosely said, measures how well the points we took in our sample, are spread among all points in the stream. If you are interested in a more formal treatment of this concept of density of sample, you can check the introduction part of the paper[1] by Cormode, Karnin and Liberty.

Algorithms in this chapter aim to "marry" the finite size constraint and a specific notion of *constant density* (or precision) as N grows, to answer queries about *approximate quantiles*. Although it might sound magical, under assumptions that don't affect algorithms practical importance, this can be achieved.

We'll see what that means on a realistic example of *web-site-time-spent data* described beneath.

## 8.1 Exact quantiles

Uninterrupted online presence and continuous contact and service for customers are of paramount importance for companies today. This is why a business or an organization is highly invested in making the availability of content and access to their web sites continuous and fast. One of the features of interest for an operating web-site, is the time that a user spends browsing it. From this data we can conclude what is the average time a user spends on the web-site. From some stable average profile, we could then identify pathologic examples of web-site-time-spent data.

We simulated some data that tracks the distribution of the real data described and shown on *Apache DataSketches*[2] site. The data there shows real data extracted from one of their back-end servers. It represents one hour of web-site time-spent data measured in milliseconds. Data on the original scale has a very long right tail, hence showing it on a single milliseconds scale as a regular histogram is not something any ophthalmologist would stand behind. Such data is better shown as in Figure 8.1. The actual bin widths are increasing as we go in the range from left to right. Nevertheless, we draw them as equal width bars. This way, we can enjoy the visual representation better. Note that this is then technically a bar chart, not a histogram.

[1] Relative Error Streaming Quantiles, https://arxiv.org/pdf/2004.01668v1.pdf
[2] https://datasketches.apache.org/docs/QuantilesStudies/QuantilesStreamAStudy.html

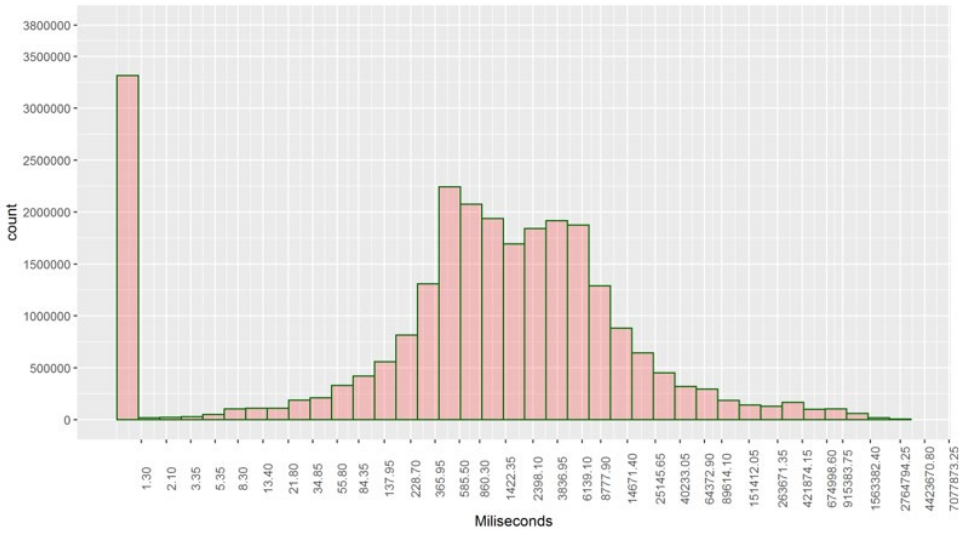



**Figure 8.1 : Bar chart (manipulated histogram) of the distribution of around 26 mil. web-time-spent data points measured in milliseconds. Visitors would visit the site within a single hour. Each length of visit is logged and this is how users time – spent looks for a single hour.**

For each user-visit to a website, back-end server(s) hosting it, logs the beginning and the end of each visit. The data shows the length of stay in milliseconds of around 26 million visits. There is a large proportion, around 14 %, of visits whose time spent is logged as 0.

For now we don't care how this data arrived, from a stream or from a database. In any case, the answers we might be interested in are:

- What is the median length of stay for visits hosted by a single back end server?
- What is the 95th percentile of the length of stay for visits hosted by a single back end server?
- What is the 95th percentile of the length of stay for visits over all back end servers that host this site?

The purpose of such queries is clear, I would like to know when my website's *gripping power* starts attenuating or when it becomes a liability in the form of prolonged server latencies (which of course we can query in a similar fashion directly). This can be revealed by the movement of the median over time (or any other quantile for that matter, that you decide is of specific importance to track, to optimize some business process).

So what is this thing, a *quantile*, that can lead our decision-ship through the angry sea of data. Unsurprisingly, the concept of a *quantile* was domesticated into human collective experience a lot before there was any big data around. Consequently, it carries around a scent of chalk-dusted sleeves. In other words, you will have to put up with some greek

letters and theory first. Theoretical quantile $\phi$ (*phi*) of some (continuous) probability distribution (density $f(x)$) is an inverse of the cumulative distribution function $F(x)$

$$\phi_x = F(x) = P(X < x) \iff F^{-1}(\phi_x) = x$$

For those of you for whom this version was too "zipped" let's illustrate using our time spent data. It helps if you insert the values we will talk about into the expression above. So, if we take $\phi_x = 0.5$, we then want to know what is the length of a visit $x$, "below" which half $(0.5)$ of the lengths of all visits are. This means that half of all visits are shorter than this length $x$. This is prominently *the median* of our data. We can calculate any $\phi-$quantile of our web-site length of stay data by sorting it and then picking out $\phi N$ –th element in the sorted sequence. For example, with 25961440 visits, the median length of stay is 12980721-st record in the sorted sequence. This record has the value of 1150.592 milliseconds (R(1150.592) = 12980721). The last expression in the parenthesis is read as rank of 1150.592 ms in the data is 12980721st.

We would similarly go about to calculate any other *exact quantile* (i.e. 95th percentile has 95% of data "below" itself). The problem of calculating quantiles is known in computer science as *sorting and selection* problem; hence by the name itself you can assume that it comes with a long history of intellectual efforts and research.

Finding minimum or maximum (respectively, ranks 1 and N) requires only linear-time and constant extra space. Similar goes for ranks that are near the edges of the data; i.e., finding any rank that is a constant away from a minimum or a maximum is similarly simple (e.g., rank $c$ or $N-c$ for some constant $c$). To find other, less trivial ranks, we can easily employ sorting, where after having sorted the array $A[0,..N-1]$ in $O(N \log N)$, finding the rank $r$ only requires that we access the element $A[r-1]$ of the array, i.e., constant time. The high price for sorting will pay off if the data is largely static, and we expect to perform many rank queries. However, if we need to locate only a few specific ranks and/or the data is frequently modified, sorting becomes an expensive hobby.

Indeed, it is sufficient to spend $O(N)$ to find any apriori-fixed rank on a given unsorted dataset. The deterministic worst-case $O(N)$-time *median of medians algorithm*, devised by Blum, Floyd, Pratt, Rivest and Tarjan[3] (BFPRT) works recursively by splitting data into groups of size 5, selecting a median of each of the $\lfloor n/5 \rfloor$ groups (linear-time operation!) and recursing on the medians until a single element remains. This element is then used as a high-quality pivot and the input to the Quick-select algorithm (highly reminiscent of quicksort), that rearranges data around the pivot and recurses on the side of rank $r$. Provided high-quality pivots from the BFPRT recursive scheme that splits data into two constant fractions, Quick-select runs in $O(N)$.

So at this point you might say, well there is our algorithm, why not just use that. Advising against such an idea is a result as classical as 1980 itself, derived by Munro and Paterson[4]. They show that any algorithm that calculates the median exactly using at least p passes over the data requires at least $O(N^{1/p})$ working memory. It's easy to identify p we have to deal with, namely, in streaming data setting, we only get one pass. This means we

[3] Blum, M., Floyd R.W., Pratt V., Rivest R.L., Tarjan R.E., "Time bounds for selection", Journal of Computer and System Sciences 7, 448-461, 1973.
[4] J.I. Munro and Paterson M.S., "Selection and sorting with limited storage," Theoretical Computer Science, pp. 315-323, 1980.

need memory linear in the size of the input. This sobering result should make it easier for us to accept some error $\varepsilon$ when estimating $\phi_x$ for streaming data.

## 8.2 Approximate Quantiles

Now that we know it is not possible to get exact quantiles under streaming data constraints, we can, in a glass-half-full-way, talk about the error. Algorithms developed for this setting always have to give some guaranteed error bounds. Well, all algorithms calculating approximate answers should come with those, for that matter. There are three types of errors you will encounter, if you sift through the (un)published research in this area:

- Additive error of the approximation of the rank
- Relative (multiplicative) error of the approximation of the rank
- Relative error in the actual domain of your data

### 8.2.1 Additive error

Most of the algorithms developed for this problem operate so as to guarantee fixed additive error of $\varepsilon N$ in rank approximation *for any* $\phi \in [0,1]$. N here is the number of elements seen so far. This leads us to *ε-approximate* $\phi$*-quantile*. This thing implies that if we ask for a quantile $\phi_x \in [0,1]$, we will always get an element $z$ with a rank $R(z) \in [\phi N - \varepsilon N, \phi N + \varepsilon N]$. $\phi_x$ implies in notation that $\phi$*-quantile* of the data is actually $x$ and not $z$, hence the bound on error around $\phi N$. A closer inspection of the error bound reveals that $z$ "promises", in the name of its rank $R(z)$, not to be further than $\varepsilon N$ away from the true rank $\phi N$, of the element $x$ we were actually interested in, but didn't get to see. So with this guarantee on $R(z)$ for any returned $z$, we can at least rely on

$$|\phi N - R(z)| \leq \varepsilon N.$$

If we allow some randomness in our algorithm then this error-bound still needs to hold, except for some small failure probability $\delta$ (delta). Designers of non-deterministic algorithms deliver a probabilistic proof that under some loose assumptions, they won't "fool" you too often. That's where $\delta$ hails from.

Notice two important consequences of such definition of an approximation error:

- the error is measured by the units of rank, not the units of your underlying data domain
- the error is constant for fixed N, but since for streaming data our N will increase with each new arrival, the *allowed actual error* for *ε-approximate* $\phi$*-quantile* will increase in absolute sense as well.

This is something to keep in mind.

To illustrate the concept of additive error we will use a set of lengths in miliseconds of ten visits to the web site

55.3, 43.1, 70.4, 64.6, 52.3, 72.4, 89.2 , 82.6, 67.7, 95.6

We first sort this set

43.1, 52.3, 55.3, 64.6, 67.7, 70.4, 72.4, 82.6, 89.2 , 95.6

For $\varepsilon = 0.1$ , $x = 50$ and $R(x) = 2$. Legal ranks are then all from the interval $[(R(x) - 0.1 \times 10, R(x) + 0.1 \times 10] = [1,3]$. Returning 1, 2 or 3 as the rank of 50 miliseconds respects the additive error bound. The *ε-approximate 0.1-quantile* can then be 43.1 or 52.3. Notice that the absolute errors on the scale of our actual data measured in miliseconds would be $|43.1 - 50| = 6.9$ and $|52.3 - 50| = 2.3$ miliseconds.

**Exercise 8.1**

**We continue the example on additive error. If we receive 90 new elements (in addition to 10 we have used to illustrate the concept) via data stream this will increase our set size to 100 (for simplicity and reproducibility assume that all are larger than 95.6). What would now be approximate ranks and ε-approximate 0.1-quantiles that respect the additive error bound?**

### 8.2.2 Relative error

Under the name *relative error* in this area, you will find the following definition of the error $z$ is carrying around,

$$|R(x) - R(z)| \leq \varepsilon R(x)$$

The word *relative* here comes from the fact that the error is proportional to the actual rank that you want to estimate, and not to the number of elements you have seen.

The only rank whose precision does not change between the relative and additive error is obviously the maximum $x_{max}$ ($R(x_{max}) = N$). *ε-approximate quantile for the* minimum is allowed to be only $\varepsilon$ away from the true rank 1. Let's look at our sorted example data again

43.1, 52.3, 55.3, 64.6, 67.7, 70.4, 72.4, 82.6, 89.2 , 95.6

If we are to ask for the *0.1-approximate* minimum of this data ($R(x) = 1$ and $x = 43.1$), the best approximation would be 52.3 with its rank 2. This does not preserve the definition of 0.1 - approximate quantile in the relative error sense. Therefore 2 is the best we can do and is not good enough if we want to keep relative error bound. Actually, for $\varepsilon = 0.1$ the only order statistic we would be able to hold the relative error bound for, is the maximum. You can check that.

Hence, to guarantee relative error, in general, is harder than to hold the additive one. For the same $\varepsilon$ , algorithm that holds the relative (multiplicative) error bound, will trivially hold the additive error bound, but not the other way around.

Relative error is important for estimating accurately quantiles in the tails of the distribution. Most of data produced online are long tailed (as our web-site visits data is too). When $R(x) \ll N$ or $N - R(x) \ll N$ (corresponding to the left and right tail quantiles, respectively) we would like the accuracy of estimates to be high, since percentiles like 99th or 99.5th or 99.975th can exhibit large absolute differences. Remember how we had to extend bar widths as we went further into the right tail of web-site data, it is because of this increasing difference. In another us-case, network latencies monitoring, the few very bad response times can cause a lot of problems for a portion of users, that then might take their frustration to their favorite social network page. Even though latencies experienced by the majority of users are not long, this majority is silent (similarity with Ex-president Nixon's election strategy is coincidental, of course). The long tailed data might have a large difference between 99.5th and 99.975th percentile, in absolute terms, say more than 30 seconds. This is why one would like to have an option of higher fidelity w.r.t. quantile estimation in the tails of the distribution. Multiplicative relative error bounds are better than additive error notion at gauging this unusual tail-behavior.

### 8.2.3 Relative error in data domain

The third type of an error you might encounter is the error relative with respect to values of your actual data items. For a quantile $\phi$, with $R(x) = \phi N$ we would like to see an element $z$ such that

$$|x - z| \leq \varepsilon x$$

holds.

This sort of an error concept is fixed to the actual scale of your data and it is applicable only to numerical data. Both of the previous error definitions, since defined with respect to the ranks, can be used to bound errors for any data that can be ordered. Due to this lack of generality, not many researchers decide to develop algorithms whose quality is judged by this error type.

Now, that we not only established that we're always wrong, but also how wrong we are, we can ask how do we mechanistically implement sketches or digests that will serve this purpose.

## 8.3 T-digest - how it works

All algorithms for approximate quantiles that you will encounter are a form of self-organizing, data-distribution-reactive data structures. They will go by the names: summary, digest or sketch. On a very high level, they save some small portion of observed data together with some meta-data for every saved data item. They then use this to answer a query about an approximate rank of an item, or conversely returning an (approximate) data item for a particular quantile query.

It often occurs that an efficient method is proposed before one can provide a disclaimer of sorts in the form of error guarantees. For example, Random Forest algorithm was used extensively (for around 13 years) before the asymptotic behavior (consistency and standard

error) of this non-parametric estimator was proven. First algorithm we will present comes from this heuristic-flavored part of the theoretical-computer-science town. Similarly like with any other well behaved and efficient heuristic, it has been widely adopted by the community since it was presented in 2013. According to their official documentation pages, T-digest is used in many prominent database and streaming data / distributed computing frameworks and libraries like Apache Kylin, Apache Druid, Apache DataSketches, PostgreSQL, Elastic Search etc. T-digest, by its construction as we'll see, offers *empirical* relative errors in the quantile space that appear to be more than acceptable for wide variety of applications. On the formal proof for the error bound though, the jury is still out. Let us first define what is *a digest*.

### 8.3.1 Digest

If you know how the famous magazine *Reader's Digest* started off, then you can draw a loftier analogy here compared to a simple one involving actual digestion. This thing has to a) consume some data and b) decide what to actually save and what to integrate through meta-data before discarding. This metadata is the actual data structure itself. You can imagine it as series of equidistant blobs along the miliseconds axis, like clusters that span the range of the data. In the case of our web-site data, the query about the median would involve data represented by clusters below the middle of the distribution. The number of these clusters is usually set ahead of any data arriving and it factors in subsequently into the space requirements of the algorithm. We will now give an example with some concrete numbers what a digest is *in one particular moment*.

Figure 8 shows how a digest with 5 clusters would look like for N=10 elements that arrived in that order. We have no reason to believe these will come in any particular order. We first partition the data items into sets $\pi_i$ according to their consecutive, non-overlapping *arrival index* intervals. This corresponds to the ant-level of the Figure 8.2. $\pi_1$ is {52.3, 72.4, 83.2} it spans arrival indices 1 to 3. These partitions are referred to as clusters and we calculate for each the mean length of visit, called *centroid*, and the number of data points contributing to this mean, called *weight*. You can observe this in the first leaves-level in Figure 8.2. We then sort the clusters according to their mean. Notice that the second leaves-level is sorted on size. In addition for each cluster we note the sum of weights left and right of it. We now have a digest which has less pieces of information compared to our original received data.

We could use this resulting structure to answer a query about the $R(72.4)$. We would find the first mean equal or larger than $72.4$. Adding all the weights left from this cluster we would have some estimate of an approximate rank. In the example from Figure 8.2, we would return 8 and be off by 1, from the true rank 7.

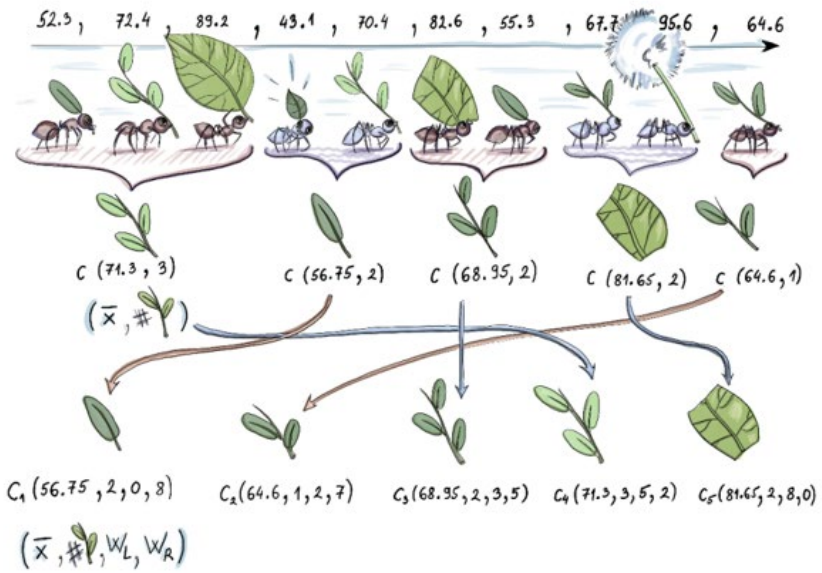



**Figure 8.2: Figure above shows a digest for 10 elements. We order the clusters according to their mean. Clusters 1, 2, 3, 4 and 5 consist of 3, 2, 2, 2, and 1 data point respectively. Each, aside from count, saves the mean value as well. $W_{left}$ and $W_{right}$ are keeping the number of data items left and right from each cluster. Notice that this digest is not strongly ordered.**

We call a digest strongly ordered if

$$i < j \implies x \leq y \text{ for } x \in \pi_i \text{ and } y \in \pi_j.$$

A digest is weakly ordered if

$$i + \Delta < j \implies x \leq y \text{ for } x \in \pi_i \text{ and } y \in \pi_j.$$

For some positive integer $\Delta \geq 1$. The example in Figure 2 shows the resulting weakly ordered digest.

**Exercise 8.2**

**What would be the smallest parameter Δ in the example from Figure 8.2? Is there one?**

Trivially, restricting the cluster size to 1 (by choosing singletons as a partition) will make the resulting digest strongly ordered. Smart, dynamic cluster size choices are the heart of

the T-digest algorithm. We will now see how cluster sizes are governed indirectly by mapping them with forethought to the widths of intervals partitioning the quantile range [0,1].

### 8.3.2 Scale functions

Ingenious part of the T-digest is how sizes (*related to weights, but not being the weights themselves*) of clusters are dynamically updated as new data is coming in. To explain this we will assume to *feed* the data into T-digest in a sorted, ascending order. This can be done for a small, finite (w.r.t. available working memory) number of observations by having a buffer of working memory at our disposal and sort them before inserting them into the T-digest. It will become clear that this is not a restricting assumption at all.

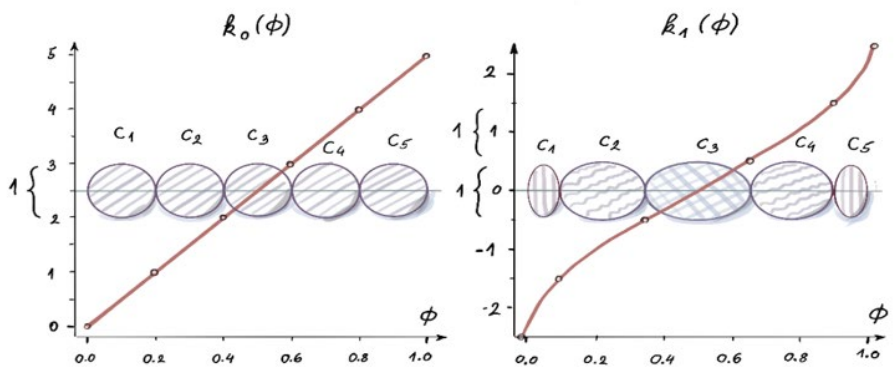



**Figure 8.3: You can see here how $\mathcal{K}^{k_1}$ and $\mathcal{K}^{k_0}$ differ between $k_1$ and $k_0$. The k-sizes for both functions allocate 5 clusters (both T-digests are shown as an ordered set of clusters that are fully merged, no two consecutive clusters can be combined without violating the weight bound). Clusters k-sizes in both cases are all 1, nevertheless, subintervals the clusters inform are different and for $k_1$ they are variable with smaller clusters around the "edges" and larger in the middle. For $k_0$ they inform same width subintervals of [0,1]**

Key lever here are functions that determine which cluster needs to merge with their neighbor and when. At each particular moment the cluster distribution along the axis is determined by the scale-functions. Figure 8.3 shows how clusters partition the quantile range [0,1] for two different scale functions. In both cases the quantile range [0,1] is covered, but the widths of the intervals pertaining to same clusters are different between them. We'll see why one is better than the other for our purposes of estimating an approximate quantile.

Clusters in figure 8.3 came to their widths by repetitive integration with neighboring clusters, until this new merge-result cluster reaches a maximal width. This process of growth stops when the resulting cluster oversteps the size bound that is determined by the scale function. Good scale functions don't hold the size bound the same for every cluster. They make it depend on their position in the subinterval of the quantile range [0,1]. In other words, the size bound above which one merges no further is different depending on where the cluster integration is ought to happen. It is different if it is close to the middle of the

range compared to when it happens at the edges of the interval. Therefore the exact part of the quantile range [0,1] that a cluster at each moment "informs", is determined by the scale function.

There are several functions suggested to be used for these purposes. Those given in the original paper for T-digest[5] are:

$$k_0(\phi) = \frac{\delta}{2}\,\phi$$

$$k_1(\phi) = \frac{\delta}{2\pi}\,\sin^{-1}(2\phi - 1)$$

$$k_2(\phi) = \frac{\delta}{4\,\frac{\log n}{\delta} + 24}\,\log\,\frac{\phi}{1-\phi}$$

$$k_3(\phi) = \begin{cases} \dfrac{\delta}{4\,\frac{\log\, n}{\delta} + 21}\,\log 2\phi\,, & \text{if } \phi \le 1/2 \\ \dfrac{\delta}{4\,\frac{\log n}{\delta} + 21}\,(-\log 2(1-\phi))\,, & \text{otherwise.} \end{cases}$$

Here $n$ is the current number of data items received. These do look cryptic and all you need to do is understand $k_0$ for a fixed $\delta$ (it is a simple line with a slope $\frac{\delta}{2}$). The others you can glance over if curious. What they all share is the same shape of the curve illustrated in Figure 8.3. All you need is to be able to think about is the shape, for what's to come.

Remember those sizes we talked about, we will now explicate their dependence on the scale function $k$ and refer to the size of the cluster $C_i$ as its $k$-size $\mathcal{K}_i$. All these scale functions are monotonically increasing as you can see, and they are defined for any quantile $\phi \in [0,1]$. What we will use them for is to calculate differences $k(\phi_{right}) - k(\phi_{left})$ for (sub)ranges $[\phi_{left}, \phi_{right}]$ of $[0,1]$. First cluster for scale function $k_0$ in Figure 8.3 spans [0, 0.2] for example. Its k-size is $k(0.2) - k(0) = 1 - 0 = 1$. Two different k functions, don't share a domain as you can see.

$k$-size $\mathcal{K}_i$ of a cluster $C_i$ is related to the width $\phi^i{}_{right} - \phi^i{}_{left}$ of the subinterval that $C_i$ currently informs. The boundaries of the subinterval $\phi^i{}_{left}$ and $\phi^i{}_{right}$ are

$$\phi^i{}_{left} = \frac{\mathcal{W}_{left}(C_i)}{n}$$

$$\phi^i{}_{right} = \phi^i{}_{left} + \frac{|C_i|}{n},$$

Where $\mathcal{W}_{left}(C_i) = \sum_{j<i} |C_j|$ is the sum of weights of all clusters to the left of the i-th one in the ordered set. Bound on the k-size for each cluster $C_i$ is then

[5] Ted Dunning and Otmar Ertl, "Computing Extremely Accurate Quantiles Using t-Digests". https://arxiv.org/abs/1902.04023

$$\mathcal{K}_i = k\left(\phi^i{}_{right}\right) - k\left(\phi^i{}_{left}\right) \leq 1$$

You can see that the k – size is actually the difference between k-values of the boundaries. As soon as a single cluster reaches $k$- size 1, it cannot continue admitting any more neighboring singletons or neighboring clusters. This is how scale functions govern the cluster sizes. Beware, on the y-axis in Figure 8.3 are $k$-values not $k$-sizes. K-sizes are calculated as differences between the $k$-values (1's hiding behind a left-hand curly braces).

The asymmetric concept of *admitting* clusters into oneself is implemented in T-digest in the form of merging of the clusters (integration of neighboring clusters into one). This is the way they grow in absolute weight $(|C_i|)$ and $k$-size. In a *fully merged* T-digest, no two (neighboring) clusters can be merged, since that *blows* their $k$-size bound of 1.

$$\mathcal{K}_{i,i+1} = \mathcal{K}_i + \mathcal{K}_{i+1} > 1$$

For scale function $k_1$, say you pick the parameter $\delta = 10$, $k_1(0) = -\frac{10}{4}$ and $k_1(1) = 10/4$, as in the Figure 8.3. That means that $k_1$ spans 5 units on the $k$-size scale, while its argument moves a single step from 0 to 1.

You might have figured out that this is a story of a derivative of $k_1$. On the periphery of the [0,1] interval, $k_1$ changes faster, and then the rate of change attenuates to a minimum somewhere around the middle. There it becomes linear with a constant slope (notice that $k_0$ has this constant slope behavior on the whole [0,1] interval). After that, it starts picking up again and ends with the same large rate of change as when it began. This way the width of the subinterval of [0,1] that the cluster is informing, is inversely associated with the rate of change of the scale function on that subinterval. The steeper the function on that subinterval, smaller the actual sizes of the clusters there. There, the fast changing scale function reaches the maximal k-size of 1 faster in that part of the interval. There the clusters remain actually small.

Consequently, as you noticed from Figure 8.3, we have smaller cluster sizes on the edges of the interval, and larger in the middle. Notice also that $k_0$ keeps all cluster sizes equal, no matter which subinterval of [0,1] you're in. This reveals bounds on the minimum number of clusters we keep at any point. Figure 8.3 shows the minimum number of clusters you can have for $\delta = 10$. It is 5, hence the minimum is the range that the scale function spans, since within each unit at most one cluster is allowed due to the $k$- size bound of 1. These 5 clusters must be at their maximum size.

Hence, $k_1$ offers an opportunity to "zoom in" on those tails close to the minimum and maximum of the data, where unusual things happen (anomaly detection). For the quantiles $\phi_{left}$ and $\phi_{right}$ very close to 0 and 1, $k_1(\phi_{right}) - k_1(\phi_{left})$ is rounded up to 1 any time $k_1(\phi_{right}) - k_1(\phi_{left})$ turns out less. It can namely happen that k-size reaches 1, without one whole data item allowed in the cluster and you can't put half of data item in a cluster. This is just a numerical artifact of the scale function and the number of data items seen so far, hence we don't want to have (and physically cannot have) less than 1 data item per cluster.

**Exercise 8.3**

What is the maximal number of clusters we can have for δ=10? How do k-sizes look when we have a maximum number of clusters? (knowing, that each time two neighboring clusters ascertain that their integration would stay within the k-size bound, they will merge )

### 8.3.3 Merging T-Digests

Now that we understood what scale function does, the actual algorithm is incredibly simple. We describe here the merge-version of the algorithm. The algorithm is the same for updating a single T-Digest with a newly arrived set of data items and for merging two T-digests. There are two phases that happen consecutively: sorting and merging.

We assume, aside for space allocated for T-digest $S_n$, that we have an extra buffer to receive a finite number $l$ of incoming data items $X_L = [x_1, x_2, x_3, \ldots, x_l]$. As they arrive we concatenate them with $S_n$ . We put them side by side. You can imagine it as an unordered union of T-digest and $X_L$ that represents $l$ singletons with means identical to the data $x_i$ and weight 1.

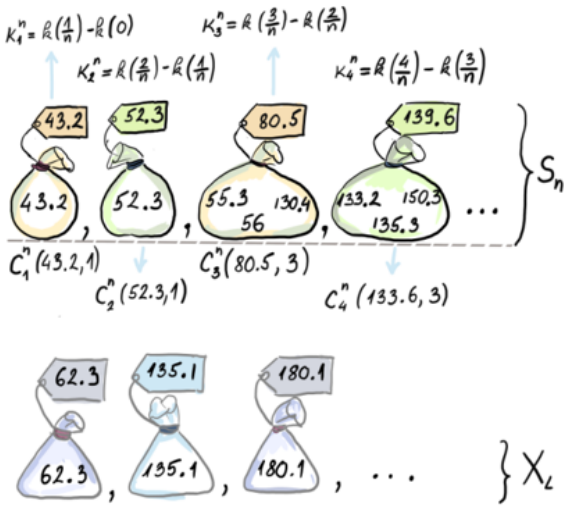



**Figure 8.4 : Strongly ordered T-digest $S_n$ (superscripts on K and C denote the number of elements from the stream seen so far). Notice that the argument of the k-function is appears in steps of 1/n (above). $X_L$ represents $l$ new elements that we are adding to T-digest from the stream (we are showing only the left tail of $X_p$ and $S_n$) .**

We then sort all $|S_n| + |X_L|$ clusters by their mean and do a following check from left to right. Is there a neighbor to the right of the cluster with which it could merge, while staying within its k-size bound of 1. If two neighboring clusters can merge like that, they proceed to

do so, from left to right. If a cluster can merge with its neighbor, then the new resulting cluster will have the mean equal to the weighted mean of the centroids of the two merging clusters. The weight of the resulting cluster will be the sum of weights of individual ones.

After sorting according to the centroid value, we move from left to right and check if adding the cluster to the right will increase the k-size over the bound of 1. This time the argument of the k-function is incremented by $1/(n+l)$. The $\mathcal{K}_i^n$ in Figure 8.4 were calculated with advancing by $1/n$ for every element represented by the cluster. This is because we now have $l$ new elements to account for.

Figure 8.5 shows the starting point after the sort phase. This becomes the input on which merge starts working.

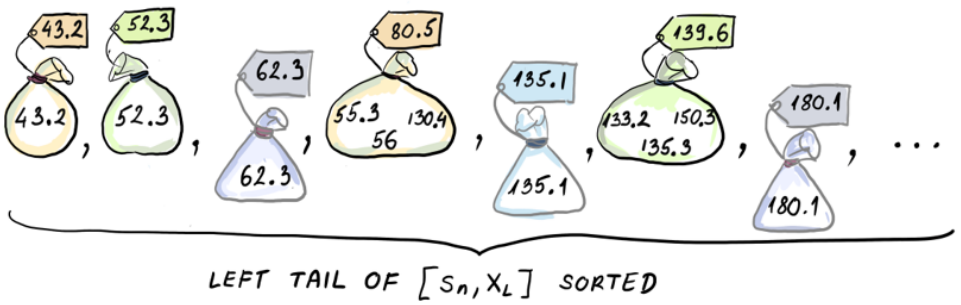



**Figure 8.5: The figure is showing sorted lower end of $[S_n, X_L]$ before the MERGE phase begins**

Figure 8.6 shows the merge process in incremental steps for the left tail of $S_n$ and $X_L$. In other words we just show the lower ends of the whole $S_n$ and $X_L$, but since the merging phase starts from left and moves to the right, this should be enough for getting a grasp of how it's done. Between this and the pseudo-code in the original paper, one should have good basis if you ever want to implement it yourself.

Figure 8.6 shows that the attempt of the first cluster to *assimilate* its neighboring singleton fails, because the resulting k-size is too big. This means that we stop, and form the first cluster from the left as a singleton (with the same value as before and weight as before the merge). The next attempt at merging is performed by the second singleton towards its right neighbor (see Figure 8.6, time point 3). This goes well as judged by the k-size of the potential new future cluster. The same process continues and now the wish to merge is widened towards the next cluster to the right. Next cluster with centroid 80.5 has weight 3. When added to the weights of singletons 2 and 3, it will cause the k-size to be bigger than 1. This means we ought to stop and merge the two singletons that remain within the legal k-size bound (Figure 8.6, time point 5). Simultaneously, the cluster with centroid 80.5 starts *looking* to the right and inquires if the sum of its weight and the weight of the singleton right from it, causes a sustainable increase in k-size. It seems that it doesn't and at time point 6 (see Figure 8.6) we create a new *old* cluster with centroid 80.5 and weight 3. The process than continues analogously as shown in the Figure 8.6. Every time a new cluster

is created, if it is a result of merging one or more clusters. Its centroid is newly calculated and its weight updated.

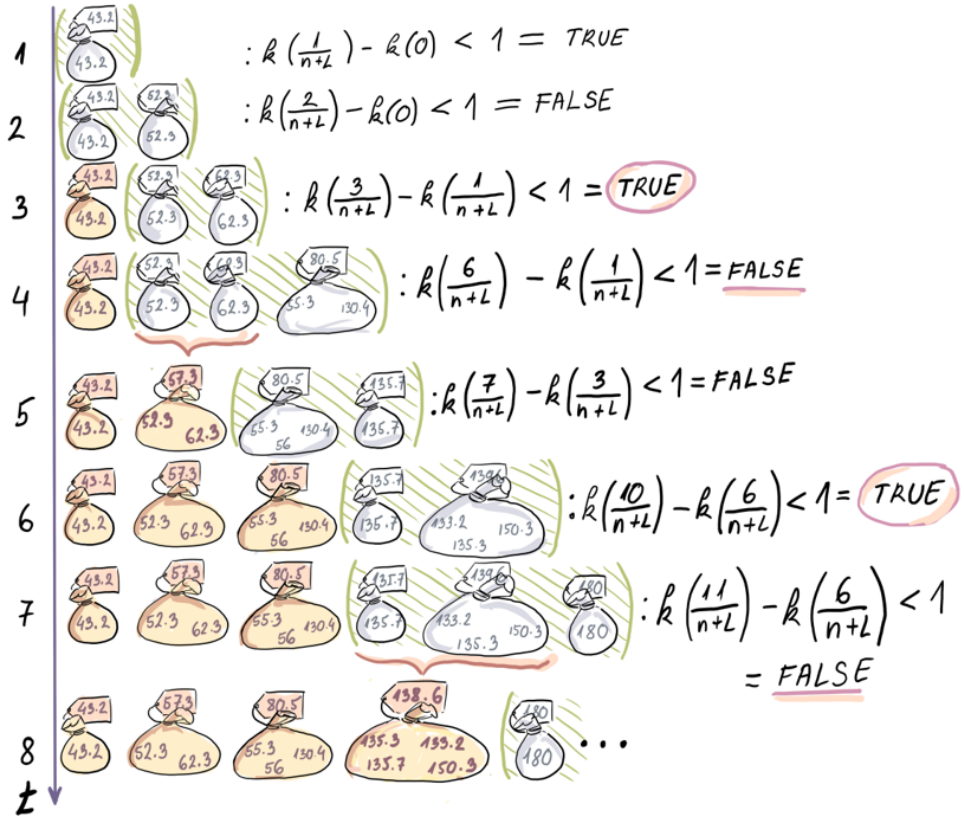



**Figure 8.6 : The figure shows the merge phase of the T-digest algorithm. Each time the assimilation of the cluster to the right is not allowed (indicated by the negative answer to the k-size query) we form a new cluster (indicated by the closed yellowish bag). The current attempts to merge with the neighbor(s) to the right are indicated by the green parenthesis at each time point. The clusters at the time point 8 are the first 4 clusters of the new $S_{n+1}$**

Numerical values on the labels of the bags from Figure 8.6 are what we actually save, while actual observations from the stream (shown in the bag) are discarded. The raw values in the bag are not kept by the algorithm, but we show them here for clarity. In addition we save for every new cluster its weight as well.

Notice that when we do the merge, strongly ordered T-digest can become a weakly ordered one. 62.3 miliseconds is larger than 55.3 miliseconds, nevertheless 62.3 miliseconds is part of the $C_2$ (in $S_{n+1}$ ) , while 55.3 is represented by $C_3$ (in $S_{n+1}$ ), after the MERGE phase

is over. This can be even more drastic when we use this algorithm to merge two T-digests, with singletons here replaced by actual clusters of the second T-digest. Large $\Delta$ values (remember $\Delta$ that decides how well ordered is the digest) in this case can increase the error, but it seems that this doesn't happen often in practice.

The case of merging two T-digests proceeds identically as in Figure 8.6, we now are not dealing with $l$ singletons from the stream, but with some $m_2$ number of clusters of the second T-digest. Idea is identical though.

The fact that we can get a T-digest for "two worlds" $D_1 \cup D_2$ by merging T-digests built on $D_1$ and $D_2$ separately, is very powerful, since it allows the use of MapReduce computing architecture. Say our data about length of a visit to a website is partitioned into substreams based on geographical location where the visit is coming from. If the data is huge, and it might be with popular sites like social networks or search engines, you will probably have to parallelize the task of approximating tail quantiles. You can send the substreams according to a geographical split to different nodes in a streaming application. They would make their T-digests built on disjoint sets of data and send them to their master node. There the T-digests are aggregated by merging them together to approximate quantiles of the whole data. Such characteristic of a sketch or a summary is very sought for and good algorithms keep this feature called *mergeability*. This ability of a summary to merge with other summaries, but prevent the error of the resulting summary to grow, is formally defined in a paper by Agarwal et al.[6] . This paper is quite technical, but if you can follow the ideas in it, it makes a very exciting read, hence, a warm recommendation.

### 8.3.4 Space bounds for T-digest

If we look at the space bounds of this algorithm, for a single T-digest we just have to keep $m$ clusters and each cluster saves a constant amount of information, mean and weight and perhaps some housekeeping metadata, if necessary. So space needed for a T-digest is bounded by the maximum number of clusters we keep at any time. The maximum number of clusters can be derived for function $k_1$ as follows: since each time clusters are allowed to merge, they will merge, the maximum number of clusters occurs when clusters are *almost* allowed to merge, but not quite.

This means that merging neighboring cluster takes the k-size just above one (like 1.01). If this is true then the average k-size of a cluster is not less than 0.5, otherwise at least 1 pair could merge. If $n > \delta$, the maximum number of centroids is $\delta$ the parameter of the scale function (remember the minimum was $\frac{\delta}{2}$). More detailed analysis of the number of clusters and the maximal weight of each cluster can be found in a short paper[7] by Ted Dunning one of the creators of T-digest.

Be it as it may, we seem to have a constant space bound of $O(\delta)$ for any number $n$ of data items that arrived. $\delta$ parameter of a T-digest is called *compression parameter* for obvious reason. This is actually pretty close to magical, if you ask me (leaving out the fact that the error needs to be empirically ascertained for each application anew and no universal

[6] Pankaj K. Agarwal et al., "Mergeable Summaries," in Proceedings of the 31st Symposium on Principles of Database Sys-, 2012. https://users.cs.duke.edu/~pankaj/publications/papers/merge-summ.pdf

[7] https://arxiv.org/abs/1903.09921

guarantee exists). Nevertheless, the magic seems to be untainted by this for majority of applications T-digest is used in.

## 8.4 Q-Digest

In this section, we will present different quantile digest (or Q-digest) introduced by Shrivastava et al[8], a precursor to T-digest heuristic that actually offers worst-case guarantees on error and space. Apart from soothing our algorithmic souls, Q-digest serves as a good example of a data structure that most accurately answers queries for the elements with highest frequencies. This is a desirable feature when working with frequency data, yet it is not shared by many other data structures --- recall from Chapter 4 that count-min sketch gave the identical overestimate range both for the bestsellers and the books that never got sold.

Q-digest can be used when elements have a pre-specified range of legal values $U = [1, \sigma]$. Basically you need to know the possible maximum that your data has to use it. This is a realistic assumption for any data that is generated by some logging or smart-devices. The goal of Q-digest is to summarize the dataset $S$ that comes in the form of key-value pairs $S = \{a_1 : c_1,\ a_2 : c_2, \ldots, a_\sigma : c_\sigma\}$ where $a_i$ is an element from $U$, and $c_i$ is the weight/frequency of the element $a_i$. Also, $\sum_{i=0}^{\sigma} c_i = n$ (total sum of observations).

Here is a rough idea behind how Q-digest blurs accuracy to save space: if an element is deemed to have too low of a count to be separately stored as a key-value pair, its count information will be merged with that of a similarly low count neighboring element. In that way we preserve information about the number of items in the relevant range, but lose information on counts of specific elements.

For instance, depending on how the parameters are set in q-digest, two key-value pairs $\{3:1, 4:1\}$ might get merged into one pair: $\{[3,4]:2\}$. In other words, we go from knowing that there is 1 copy of 3 and 1 copy of 4 to knowing that there are two copies in interval $[3,4]$. This idea is further applied to intervals with small counts. Intervals themselves can then be merged to save space, if the elements in those intervals do not have high enough frequencies. The decision on what constitutes a high or a low count and the subsequent "blurring" step is determined by the compression parameter $k$. Remember in T-digest we had scale functions, this is an analogous notion of something that will control number of intervals. Next we show how to construct (and store) Q-digest.

### 8.4.1 Constructing a Q-digest from scratch

For the purposes of understanding how Q-digest works, we will envision an implicit tree $T$ (tree never gets actually stored.) The tree is a full binary tree with $\sigma$ leaves (so as big as your universe $U$), where the $i$th leaf from the left denotes the $i$th element from the universe $U$. In our simple example shown in Figure 8.7, our universe is $U = [1,8]$, and our dataset is $S = \{1:1, 3:5, 4:9, 5:2, 7:2, 8:1\}$. This frequency information for each element will be stored at

[8] Shrivastava, N., Buragohain, C., Agrawal, D., Suri, S., "Medians and Beyond: New Aggregation Techniques for Sensor Networks", SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems

each corresponding leaf of $T$. Each node $v$ of $T$ has an associated *count(v)*, and in the beginning, only leaves have their *count(v)* filled up.

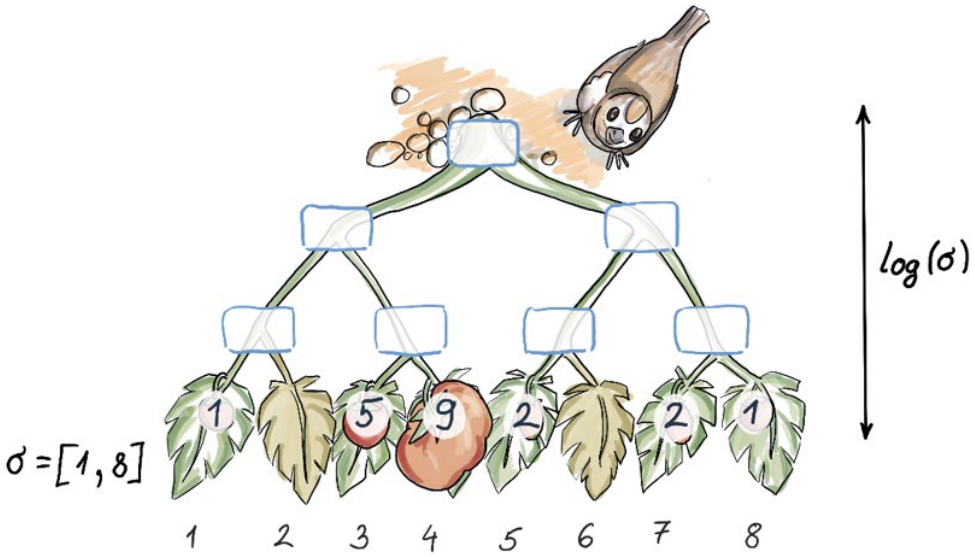



**Figure 8.7 Original data and its frequencies represented through the leaves of an implicit tree.**

To produce a Q-digest from this implicit tree, we follow 2 rules:

1. For every internal node $v$ in $T$ that is not a root, $\text{count}(v) \leq n/k$ (leaf and root nodes are allowed to break this rule).
2. For every node $v$ in $T$ that is not a root, $\text{count}(v) + \text{count}(v_s) + \text{count}(v_p) > n/k$, where $v_s$ denotes the sibling of $v$, and $v_p$ denotes the parent of $v$. (The root node is allowed to break this rule).

Below depicted is the process of turning our implicit tree from Figure 8.7 into a Q-digest according to Rules (1) and (2), in a couple of steps. In this example, we have that $n = 20$, and $k = 5$, so the maximum value allowed at the node is 4, and every "triangle sum" from Rule (2) needs to be at least 5. Note that in the beginning of the process, Rule (1) is not broken as it does not refer to leaf nodes, and they are the only ones containing any values in the beginning.

The process begins at the leaf level, where, going left to right (remember in T-digest too, going from left to right), we identify all "triangles" on the bottom level that break the Rule (2). Each time that happens, the values of $v$ and $v_s$ are summed up and added to the value in $v_p$, and then deleted from $v$ and $v_s$. For example, in the rightmost triangle on the bottom of the first tree in Figure 8.8, we sum up 2 and 1, and place 3 in their parent, and then delete 2 and 1. We continue in the same fashion with the next level, going again from left to

right and finding problematic triangles. The process ends at the root, and the root is allowed to have indefinitely low or high values.

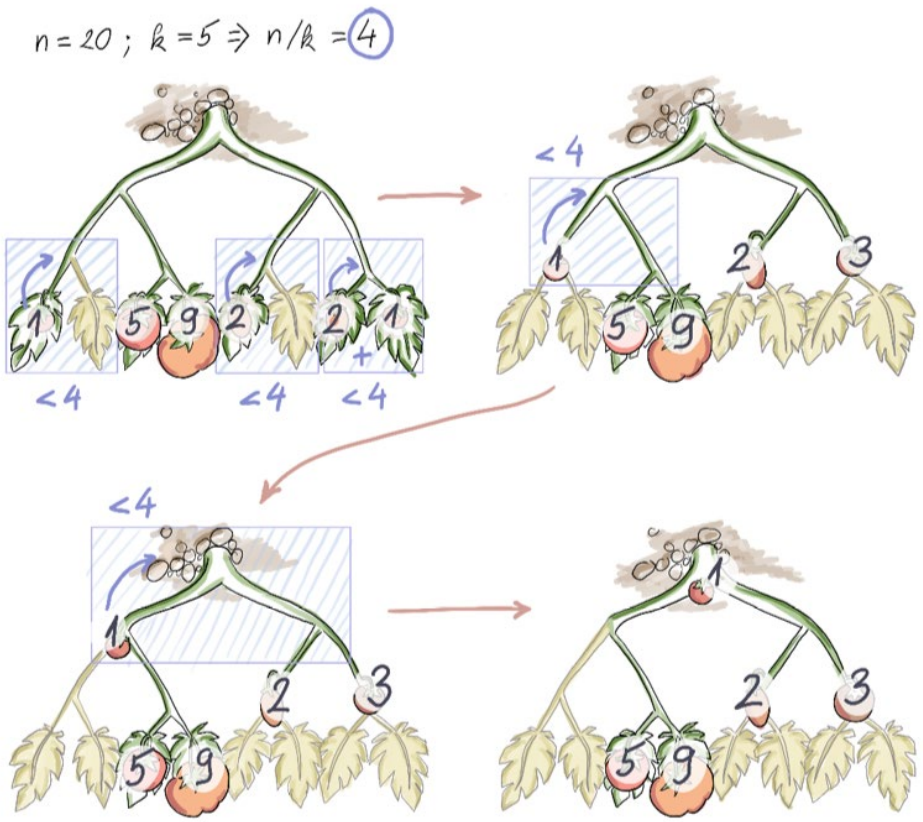



Figure 8.8 Building a Q-digest from scratch

Using this process, we arrive at the final Q-digest, represented by the last implicit tree in Figure 8.8. The tree is never explicitly stored, only the nodes that have values in them. If we assign a level-by-level, left-to-right enumeration to nodes of the tree, then the resulting Q-digest in Figure 8.8 saves the following information:

$Q = \{[1,8]:1, [5,6]:2, [7,8]:3, [3]:5, [4]:9\}$. If we enumerate each node and corresponding interval in a level-by-level left-from-right fashion, we get the following description: $Q = \{1:1, 6:2, 7:3, 10:5, 11:9\}$. Okay, so we have gone from storing 6 key-value back when we had original data to storing 5 of them. Not that great. But with a larger universe and many low counts starting at the leaf nodes (typical of Zipfian distribution exhibited by web-site-time-spent data), space savings become quite substantial.

### 8.4.2 Merging Q-digests

Q-digests have been originally devised for the sensor-network and distributed setting, where Q-digests can be locally computed and then later merged with Q-digests at other nodes. The process of merging Q-digests is fairly simple, if both Q-digests refer to the same universe. Given two trees $T_1$ and $T_2$, the Q-digests can be merged by creating a tree $T$ over the identical universe, and summing up the respective nodes from $T_1$ and $T_2$ into $T$. See the process in Figure 8.9 below where $n_1 = n_2 = 30$, $k_1 = k_2 = 6$. So in the original $T_1$ and $T_2$, the maximum value at each node is 5.

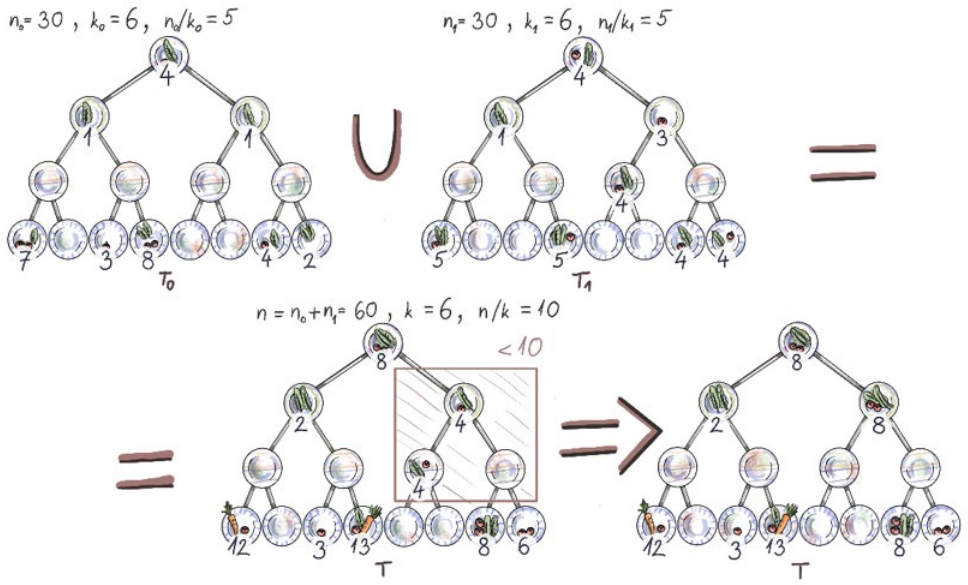



Figure 8.9 Merging two Q-digests where $n_1=n_2=30$, $k_1=k_2=6$. Because the digests are of the same sizes, the resulting Q-digest is of the same size, where the values at nodes are the sums at respective nodes. For example, the resulting Q-digest has the value 8 at the root, because the Q-digests participating in the merging operation both have the value of 4. However, this might lead to a Q-digest that is not fully merged. In this particular example, the resulting Q-digest has $n=n_1+n_2=60$, and $k=k_1=k_2=6$, so we are looking for triangles whose sum is 10 or less, and we propagate those values up to the parent, as before.

Once $T$ is initially created from $T_1$ and $T_2$, it also needs to undergo the process of constructing a legal Q-digest that respects both rules we mentioned earlier using the following parameters: $n = n_1 + n_2$, $k = k_1 = k_2$. If two Q-digests being merged have roughly comparable or equal sum of observations, then the resulting Q-digest's threshold ($n/k$) will be twice the size of the threshold for the earlier Q-digests. In our example, maximum node value for the resulting Q-digest is 10. The two original Q-digests being merged stored in total 16 key-value pairs, but the resulting Q-digest uses only 8 key-value pairs, twice as less.

### 8.4.3 Error and space considerations in Q-digest

The maximum space used by a Q-digest depends on the compression parameter $k$, and it equals $3k$. Denote the size of $Q$ as $|Q|$ (measured in the number of key-value pairs). Then from Rule (2), we have that:

$$\sum_{v \text{ in } T} \text{count}(v) + \text{count}(v_s) + \text{count}(v_p) > |Q| * n/k$$

Also, it holds that

$$3 * \sum_{v \text{ in } T} \text{count}(v) \geq \sum_{v \text{ in } T} \text{count}(v) + \text{count}(v_s) + \text{count}(v_p),$$

since in the right expression, the count of each node is being at most triple counted (each node appears once as a parent, once as a sibling and once as itself - think about what's counted for the leaves and the root here).

The left expression equals $3n$, hence we get: $|Q| * n/k < 3n$, which gives us $|Q| < 3k$.

As for the error rate, before we calculate the error when posing quantile queries, we need to observe how far a value within one node of the implicit tree $T$ can be off. All nodes on the path between the root and node $v$ can hold values potentially belonging to the interval specified by $v$. With that fact and the depth of the tree being $\log \sigma$, the total error within one node is $\log \sigma * n / k$. If we observe this error in the relative term (as a percentage of $n$), we get that the error within one node is at most $\log \sigma / k$.

### 8.4.4 Quantile queries with Q-digest

In order to perform quantile queries with Q-digest, it is useful to sort the nodes of the implicit tree in a post-order traversal fashion. In other words, in the sorted sequence, we place the interval $i = [x,y]$ only after all of its descendant subintervals have already been placed. Once we have done that, and provided with a quantile query $x$, we sweep the array of key-value pairs, accumulating the values until $x$ has been reached or overshot. The quantile reported is the right end of the interval where $x$ was overshot. The example of a quantile query is given below, for the resulting Q-digest from Figure 8.9.

The post-order sequence of nodes in this tree with nodes enumerated in a level by level fashion is $\{8:12, 10:3, 11:13, 2:2, 14:8, 15:6, 3:8, 1:8\}$. This corresponds to right-sorted ranges $\{[1]:12, [3]:3, [4]:13, [1,4]:2, [7]:8, [8]:6, [5,8]:8, [1,8]:8\}$. Let's say we are given a query $\phi = 0.5$, hence $x = n/2 = 30$ i.e., we are looking for a median. Sweeping left to right and accumulating values in the list, we will accrue exactly the sum of 30 at 12+3+13+2=30, and we will report the right end of our last node as the median. For the node containing 2, the right end of its range is 4. The reported median is 4. Similarly, say we are looking for $3n/4 = 45$. We again start from left accruing the sum 12+3+13+2+8+6+8=52. With the last value 8, we overshoot the value we are looking for but our error will be commensurate with the max node values, so we report the right end of the interval corresponding to the key-value pair $[5,8]:8$, which is 8. Our returned result is 8.

Q-digest can be also used to answer many other types of queries, most notably range queries, inverse quantiles and consensus queries. Provided memory of $m$ locations to build a Q-digest, the error in quantile query is at most $\varepsilon \leq \frac{3 \log \sigma}{m}$. We obtain this by setting the compression factor $k$ to be $m/3$.

## 8.5 Simulation code and results

To witness T-digest and Q-digest in action we devised simulation scenario to showcase their empirical error behavior and compare their estimates of percentiles far in the right tail.

We drew 10 samples without replacement each with $10^5$ elements from our 2Gb of web-site data shown in Figure 8.1. Since Q-digest works on integers only, we rounded the web-site data to the nearest millisecond. This way we are able to use both algorithms on the same samples. The 10 samples with 100K observations each, that the code is using, as well as the total web-site data are available in the code repository of the book.

To calculate our results we used the `tdigest` library in `Python` and the version of Q-digest implemented in Python from https://papercruncher.wordpress.com/2011/07/31/q-digest/ after validating the code on several small stream examples. The code to calculate and save results from T-digest and Q-digest respectively is shown beneath. The code reads in 10 samples and after each of the 10 is consumed by their own T-digest or Q-Digest object, a query is executed to get 95th and 99th percentile of the data. We end up with 10 estimates of 95th and 99th percentile.

```
import pandas
from pandas import DataFrame
from tdigest import TDigest
import numpy as np
import os

df = pandas.read_csv('./test.csv')    #A

resNinetyFive = np.array([])    #B
resNinetyNine = np.array([])    #B

columns = list(df)
for j in columns:
    tDigest = TDigest(delta=1 / 200)    #C

    tDigest.batch_update(df[j], w=1)    #D

    resNinetyFive = np.append(resNinetyFive, tDigest.percentile(95))    #E
    resNinetyNine = np.append(resNinetyNine, tDigest.percentile(99))    #E

res = DataFrame({'NinetyFive': resNinetyFive, 'NinetyNine': resNinetyNine})    #F

os.chdir("./")     #G
res.to_csv("Results_TD_WebsiteSample.csv", index=False)    #G
```

**#A Read in 10 samples of length 100K each as a data frame. The csv file can be found in the book's code repository.**
**#B Make empty arrays to save the results for 95th and 99th percentile.**
**#C For each of the ten samples make a T-digest with the reciprocal of $\delta$ = 200 (delta is parameterized differently in the implementation and the paper)**

#D Let the digest consume the j-th sample. w here means we are adding singletons with weight 1. For two different two digests these would be actual weights of the clusters.
#E Add the estimate of the 95th and 99th percentile to the results array.
#F After all ten samples are consumed make a data frame for the results.
#G Specify the directory and save the results.

The efficiency of the implementation seems to be much better for the T-digest solution, much so to make the efficiency comparison trivial.

```
import numpy as np
df = pandas.read_csv('/path/to/test/data')    #A

resNinetyFive = np.array([])    #B
resNinetyNine = np.array([])    #B

columns = list(df)

for j in columns:
    universeSize = max(df[j])+1    #C
    qDigest = QDigest(universeSize, 20)    #C

    length = len(df['sample1'])    #D
    for i in range(length):
        qDigest.insert    #D

    qDigest.compress()    #E

    resNinetyFive = np.append(resNinetyFive, qDigest.quantile_query(([0.95])))    #F
    resNinetyNine = np.append(resNinetyNine, qDigest.quantile_query(([0.99])))    #F


res = DataFrame({'NinetyFive': resNinetyFive, 'NinetyNine': resNinetyNine})    #G
res.to_csv("/path/to/test/output", index=False)    #G
```

#A Read in 10 samples of length 100K each as a data frame. The csv file can be found in the book's code repository.
#B Make empty arrays to save the results for 95th and 99th percentile.
#C For each of the ten samples make a Q-digest with the size of the universe as the parameter. +1 is for the 0's
#D QDigest class takes one element at a time. Digest consumes the j-th sample one – by – one.
#E QDigest gets reorganized after the whole sampel has been consumed to comply with the two triangle rules.
#F Add the estimate of the 95th and 99th percentile to the results array.
#G After all ten samples are consumed make a data frame for the results.
#G Specify the directory and save the results.

When it comes to the parameters chosen for creation of the two digests, we chose them to make them of approximately equal size. For Q-digest we chose the compression parameter $= 20.$ , while for the T-digest we chose $\delta = 200.$ , which yields around 1kB size of each digest. For each sample we found the maximum length of a visit since it is required for creation of a Q-digest. For all 10 samples maximums were between 2742437 milliseconds and 2763605 milliseconds, hence the universe sizes do not vary enough to prevent meaningful cross-sample evaluation of the results.

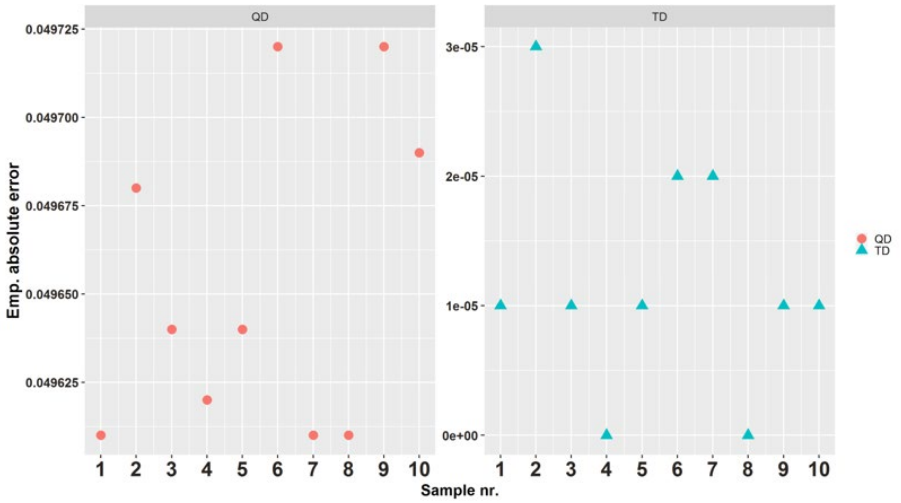



**Figure 8.10 : The graphs show empirical absolute error that Q-digest (left) and T-digest (right) exhibit when estimating 95th percentile. The error in estimating $q \in [0,1]$ is calculated as follows: if the true rank of the answer (value) that a digest provides is $r$, then the absolute error is $| r / n - q |$ where n is the number of elements *digested* so far.**

The error that we are showing is calculated as follows: for each of the 10 samples we can get the exact 95th and 99th percentile, because we can sort the data and find the actual values. These are the $x$'s we hope to get. From the digests we get $z$' s and we can check what is the difference between R(x) and R(z). In the case of the 95th percentile R(x) should be 95000. If the digest returned 94 990th element then the absolute error we are showing is |0.94990-0.95000| = 0.00010. You can check in the right graph, this is how close and closer the T-digest actually comes. Figure 8.10 shows resulting absolute empirical errors for estimation of the 95th percentile for each of the 10 samples. First thing to notice is that we are not showing the absolute empirical errors on the same y-axis. The average absolute empirical errors of Q-digest are around 5000 times higher, than the average absolute empirical errors of the T-digest. We wouldn't be able to appreciate their within (red and blue) group variability visually, had we showed them both on the same axis. Average absolute error of T-digest (blue triangles) is $1.2 \times 10^{-5}$ , while the one for Q-digest shows average of $4965.4 \times 10^{-5}$. It seems that T-Digest outperforms the Q-digest, as judged by the absolute empirical error on this data, by the order of magnitude of $10^3$.

To fully appreciate this difference we need to first understand what it means to be off by $1.2 \times 10^{-5}$ when estimating 95th percentile. Since we created the data, we know exact quantile for any $q \in [0,1]$ (as long as the sample offers that fidelity, i.e. it is hard to get an exact 99th percentile of a sample with 10 elements). Hence, if the true rank of $z$ that we get back from the digest when we ask for 95th percentile is $r$, then the absolute error becomes $\left|\frac{r}{n} - 0.95\right|$ where $n$ is the number of elements in the sample (or seen from the stream so far).

So if one is of by $1.2 \times 10^{-5}$ from $0.95$ this means that the T-digest delivers 95001st or 95002nd web-site-time-spent, instead of 95000th. According to the same reasoning, Q-digest returns 90035th or 99966th web-site-time-spent in their ordered sequence, instead of 95000th. Another thing to notice is that we are not talking about miliseconds here, but the ranks that the lengths of visits occupy.

You can notice the same pattern in Figure 8.11, where we show analogous results for 99th percentile. Although the difference between average absolute empirical error is 7 times smaller than for the 95th percentile, nevertheless T-digest still errs by a single element, while Q-digest errs by close to a thousand. Nevertheless, the error bound claimed in the original paper on Q-digest is upheld. With our maximal universe size of 2763605 and $k = 20$ we can calculate that the error should be below 0.74. This is definitely true for our case here. Nevertheless, an additive error bound for estimating 95th percentile of 0.74, means that even giving 22nd percentile as the answer would manage to keep the theoretical upper bound on the error.

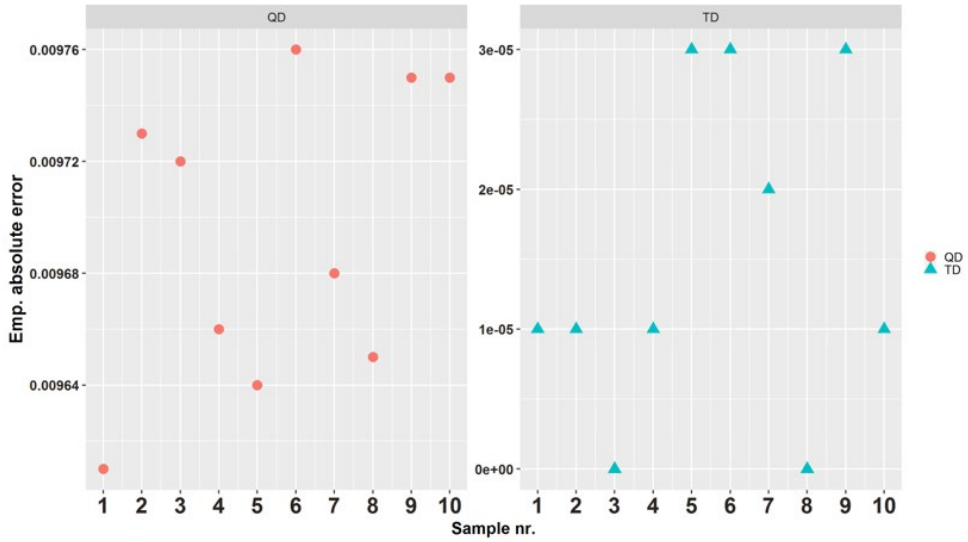



**Figure 8.11 : The graphs show empirical absolute error that Q-digest (left) and T-digest (right) exhibit for estimating 99th percentile. The error in estimating $q \in [0,1]$ is calculated as follows: if the true rank of the answer (value) that a digest provides is $r$, then the absolute error is $| r / n - q |$ where $n$ is the number of elements *digested* so far.**

According to our results T-digest clearly outperformed Q-digest when estimating *high-end* quantiles like 95th and 99th percentile on this data. Since T-digest does not come with any error upper bound, we cannot claim that it stays beneath it, but judging by our empirical results, it certainly stays under Q-digest one for this data.

## 8.6 Summary

- Having ad-hoc approximate quantiles any time in a streaming data application, especially one that is used to detect anomalies, is very important. Having efficient online algorithms that save small summaries of the whole data and deliver answers about quantiles with some error guarantee, means being able to create a small sketch of the data distribution in the form of a histogram. This way you can keep an eye on multiple quantiles at once and set sensitive thresholds.
- Error with which algorithms approximating quantiles work is either additive or relative (multiplicative). Additive error $\varepsilon n$ is the same no matter which quantile we estimate. Relative error $\varepsilon R(x)$ is relative to the specific quantile we are interested in, so it is smaller for small quantiles and largest $\varepsilon n$ for the maximal value. Error in the data space instead of the ranks space, is sometimes called relative error as well, but it has nothing to do with the relative (multiplicative) error above.
- We have gotten to know T-digest, very popular heuristic algorithm for calculating approximate quantiles. We saw the mechanism of keeping small samples on the edges of quantile interval and larger in the middle and how k-size regulation via scale functions makes sure of this. This results in better accuracy when estimating tail quantiles like 95-th-percentile or 99-th percentile, or even finer ones close to 1. We saw how T-digest delivers very accurate estimates in those ranges, on 10 samples of realistic web-time-spent data.
- We learned what Q-digest algorithm is and how to build one from scratch, as well as how to merge two or more. Q-digest works for integers only and one needs to be familiar with the universe the data is coming from in order to use it. For streaming data this can and might not be true. In case we don't really know the range of data we are about to see, Q-digest is of limited use. T-digest does not share that constraint and seems to perform better than Q-digest, with absolute empirical errors smaller by 2 to 3 orders of magnitude for estimating same *high-end* quantiles.

# 9

# *Introducing the External-Memory Model*

**This chapter covers**

- Introducing computer limitations that affect the design of data-intensive applications
- Introducing and describing the external-memory model (DAM model)
- Understanding why DAM model is useful for analyzing algorithms for massive data
- Building simple scanning, searching and merging algorithms in external memory
- Reviewing use cases where data scientists and programmers work with huge files
- Using Big-Oh notation to measure I/O-efficiency of the algorithms
- Understanding differences in the design of in-RAM and on-disk algorithms
- Discussing drawbacks of the external-memory model

This chapter introduces fundamental ideas that form Part 3 of the book. We begin by introducing external-memory algorithms and the external-memory model[1]. This model will teach us how to view the efficiency of algorithms and data structures in the context of working with large datasets stored on disk.

Most applications maintain data on some type of local or remote storage, files and databases being prominent examples. Storage offers the flexibility of capturing large amounts of data persistently and very cheaply. Even when the system benefits from data summaries that quickly satisfy queries from RAM, one still wants to preserve the original data on some slower and larger storage. As we have seen in case of Bloom filters and Google's BigTable, when the query returns `Present`, we make a trip to disk to fetch the `(key,value)` pair and metadata, or to establish that we have a false positive.

[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.

Data structures that power relational (and other types of) databases take lessons from storage and memory design to offer optimal on-disk performance, and they are different from data structures that perform the same tasks optimally in RAM. As we will see, the worlds of in-RAM data structures and those on disk significantly differ. Through building analogies between two types of data structure and algorithm designs, our aim is to demonstrate common themes and algorithmic tools you may find helpful for solving any on-disk problem at your hands.

Before moving on, let's clarify what we mean when we say 'disk'. Depending on the context, data can be stored in different types of local or remote storage, such as local solid state drive (SSD), a local magnetic disk, a cloud, or some combination of the three. In this chapter, we will refer to all these storage devices simply as 'disk', as the central issues we plan to address are shared by all of the types of storage, the main one being the slow speed of access. There are many differences in access times between different storage technologies; for instance, SSDs have superior speed performance characteristics than magnetic disks in many respects, however, the performance is not superior enough to make the problems we will delve into throughout this chapter entirely go away.

The specifics of computer design, such as CPU-performance gap, memory hierarchy, latency and bandwidth, form an important foundation to understanding how to design efficient data structures for external memory. For a refresher on those topics, you might want to peek back at Section 1.4 of the book. Below are the key peculiarities that make the issues surrounding the design of external-memory algorithms new and unique:

1. Main memory (RAM) is significantly smaller than a large dataset residing on disk, and can only hold a small portion of the entire dataset at one time. Therefore, to solve a problem (say, sort a file), data has to be brought in and out of main memory piece-wise.
2. Data from disk is fetched in consecutive chunks (i.e., blocks/pages). Bringing a block of data into main memory is an expensive operation which is offset by having the chunk carry many elements. Whether we make use of just one or dozens of thousands of elements in the chunk, we pay the same cost of one data transfer.
3. Performing a single input/output of a block, that is, an I/O transfer from disk to main memory is very slow, and it can be up to a factor of 100-1000 slower than a typical computational operation in RAM.
4. Sequential order of accessing data on disk is faster than the random order. The sequential access makes use of the high bandwidth, whereas the random access faces the penalty of high latency.
5. Disk reads tend to be faster than disk writes.

The main takeaway from this chapter will be learning to zoom out of RAM, and see the bigger picture of how data travels back and forth between the disk and main memory. Understanding that the data-transfer aspect is the bottleneck for many applications out there, while everything that happens in RAM (e.g., which internal-memory algorithm or a data structure we use) is often a second-order concern, is one of the key lessons of this chapter. Switching to this manner of thinking is not always easy, considering that we are

used to counting comparisons, arithmetic operations and other stuff that happens in RAM, as well as being used to having entire dataset available at our hands. We will continue to use the Big-Oh notation to characterize the runtime of algorithms, but from now on, the expressions inside the Big-Oh will reflect the number of data transfers, not CPU cycles. To anchor us in this new worldview, next we introduce the external-memory model, and show a couple of basic algorithmic examples to simulate it.

## 9.1 External-memory model: The preliminaries

The external-memory model, or the Disk-Access Model (DAM) was first suggested in 1988, back when many large organizations started encountering their first massive-data issues. Since then, it proved to be an incredibly helpful tool to analyze algorithms if you are working with data-intensive applications. The Figure 9.1 depicts this model.

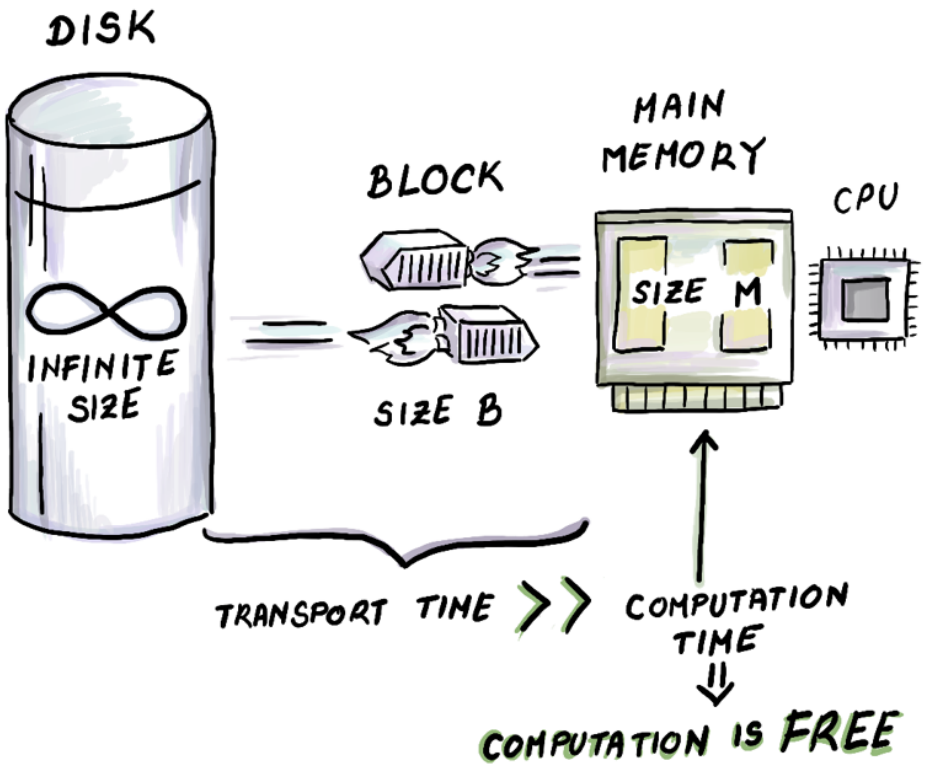



**Figure 9.1: The external-memory model. This model is suitable for analyzing massive-data applications, where the cost of computation in RAM is subsumed by a much larger cost of transferring data from disk to main memory and back. Computation is not exactly free — but it is so much cheaper than that cost of transferring data that in many applications, it is effectively free.**

In the external-memory model, the computer consists of external storage (disk) of infinite size, and the main memory of limited size $M$. Data originally sits on disk, and is transferred between disk and main memory in blocks of size $B$. Once data arrives into main memory, all computation on it, and whatever is done with it, is given out for free. Once you brought a block of data into memory, you can consider it sorted if you need it to be sorted, for example. No sorting cost will be charged. You can consider any meaningful computation (that does not require overshooting the size of remaining memory!) already done. The only cost charged will be that of 1 I/O data transfer to RAM.

You may think of parameters $M$ and $B$ as values plugged into the algorithm. Different computers have different values for $M$ and $B$, depending on the particular hardware setup. These parameters will also appear in the Big-Oh analysis of external-memory algorithms as they will be used to analyze the efficiency. The place where each parameter appears in the bound will help us better understand the role that each plays. Our input size all throughout Part 3 of the book will be $N$, and it will denote the total volume of data (i.e., $N$ unit-sized elements.)

Keep in mind that even though in the Big-Oh analysis, constants do not matter, and the values of B and M are constant for a given computer, we will view them as parameters that can grow and change and the runtime of algorithms should be parameterized by them. So, for example, we will not simplify $O(N/B)$ to $O(N)$ even though $B$ does not grow the same way we expect $N$, our input, to grow.

Now, does $B$ stand for $B$ bits, $B$ integer variables, $B$ 64-bit words or something entirely different? If we pick the same unit consistently for $N$, $M$ and $B$, then the choice is not that important, as the relevant ratios remain the same, for instance, $N/B$ (number of blocks the dataset takes up on disk), or $M/B$ (the number of blocks that fit into main memory).

We will assume here that the unit is the memory footprint of one particular data item in the dataset that we are working with, be it a string, an integer, float, or some larger, more complicated object. For the sake of simplicity, we will also assume that in one dataset, all items are the same type and they take up the same amount of space, even though the reality will defy us on this one. The experience and research show that records in databases, for example, can be of very different sizes, which can significantly impact the runtime of algorithms that blindly assume all records being the same size. Check out the research on B-trees for different-sized atomic keys (Bender, Hu, Kuszmaul), as well as sorting with size-priced information (Bender, Goswami, Medjedovic, Montes, Tsichlas) if you want to learn more.

But to get anything done in life, we need to make some simplifying assumptions. Either way, we proceed with the following parameters in mind:

- $N$ = number of records in the dataset sitting on disk
- $M$ = number of records that can fit into main memory
- $B$ = number of records that can fit into one block

The size of $B$ is usually the size of a block transferred between disk and memory, and it is usually between 4KB and 64KB. On some SSDs, a block is on the order of a couple of MBs.

Either way, it is important to understand that one block carries thousands of elements consecutively placed on disk. Memory sizes also vary, and currently an average computer would have somewhere between 8GB and 32GB. However, not all that space can be used for computation, as the RAM holds all working programs, the operating system, etc. Size of $N$ also varies, but if we are talking about data sitting on disk, then we should be ready for some very big datasets. As an example, Teradata Relational Database Management system claims that they can host databases of up to 50 petabytes (PB) in size. In summary, it is correct to assume that in most situations, $N$ is much much larger than $M$, and $M$ is still significantly larger than $B$ even though the latter gap is much smaller. Now let's put this model into action with few examples.

## 9.2 Example 1: Finding a minimum

Quite commonly, we need to find a minimum value in a set of values. From the perspective of traditional algorithms, this requires a linear scan of elements in the list where the items are stored. Now consider an analogous external-memory use case.

### 9.2.1 Use case: Minimum median income

Say you are working for a startup that is modeling demography and census data and visualizing it for their users (e.g., check out Social Explorer at www.socialexplorer.com.) There is a large number of tables that carry aggregated data, and for the income table data, the individual records are aggregated up to a demographic block (as in 'A new kid on the block') level. We will differentiate the demographic blocks and disk blocks by calling them 'demographic blocks' and 'blocks', respectively. The US territory is divided into over 10 million demographic blocks, and for each demographic block, the table carries substantial amount of information all organized sequentially per block. One of the variables includes median income and we are interested in finding the demographic block with the minimum median income in the entire US.

Effectively, what we are given is an unsorted array of $N$ integer records on disk, and we need to find the minimum value. In the world of "regular" algorithms, a simple `for` loop requiring $O(N)$ comparisons suffices. If we apply the analogous approach when data resides on disk, then we instead pick up blocks of data, starting from where the data begins, to the last block containing any of our items. Consider below the two patterns of how we read data in RAM and on disk in pseudocode:

How we read data in RAM:

```
min = INT_MAX
for i in range(N)
    if (A[i] < min)
        min = A[i]
```

How we read data from disk:

```
BLOCK_SIZE = 1024
min = INT_MAX
for i in range(ceil(N/BLOCK_SIZE))
    BLOCK = read_block(filename, file_start + i*BLOCK_SIZE, BLOCK_SIZE)
    for i in range(BLOCK_SIZE):
        if (BLOCK[i] < min):
            min = BLOCK[i]
```

The second block of pseudocode reads a block of data by specifying the filename (this tells us the starting position of the file), the offset, that is, the position within the file, and size of a block to read starting from that position. This pseudocode makes a couple of simplifying assumptions, such as assuming that `file_start` occurs right at the block boundary, which might not be the case. Because disk is partitioned into blocks of memory, and blocks tend to be fairly large, there is no guarantee that our file will begin at the beginning of the block. Our `read_block` function assumes that if the position we seek is in the middle of the block, then it should pick up the block containing the item we desire.

A few notes about programming in external memory: when working with large files in Python, for example, we can use a number of libraries that allow us to perform system calls such as `open` (to open a file), `seek` (to look up a particular position in the file) or `write` (to write to a specific position in the file.) Roughly speaking, `seek` corresponds to an expensive I/O read that we have been referring to, however, it is difficult to track how exactly blocks are being shuffled to and fro by the operating system. The operating system has a large number of built-in optimizations that work under the hood to address I/O issues. For instance, if the operating system observes that we have accessed a number of blocks in sequential fashion, it might fetch a number of blocks that follow it even before we requested them, assuming this is what we might want to do next. Also, when we read in a block and modify it in main memory, the operating system might not want to immediately write it back. Instead it might prefer to buffer it in RAM, and write it back together with a number of other blocks in sequence later.

This is a large and important topic, however what we aim for in our teaching of external-memory algorithms is understanding the concepts from the abstract point of view and understanding the algorithmic tricks involved. To that end, we show examples in Python-like pseudocode where we physically pick up particular blocks to show the workings of the algorithm, even though this is not how this type of code would regularly get written. We believe this simplified view aids the sort of understanding we're after.

Now, back to our example. As we sequentially scan blocks on disk, we input blocks one by one into main memory. Once the block is actually read, we do not need it anymore, so at all times, we just need one block in memory simultaneously and a `min` variable that we update accordingly. The Figure 9.2 below shows the process on a toy example where $N = 11$, $M = 6$ and $B = 3$. Because there are at most $N/B + 1$ blocks that our data occupies, the algorithm requires $O(N/B)$ memory transfers (or I/Os).

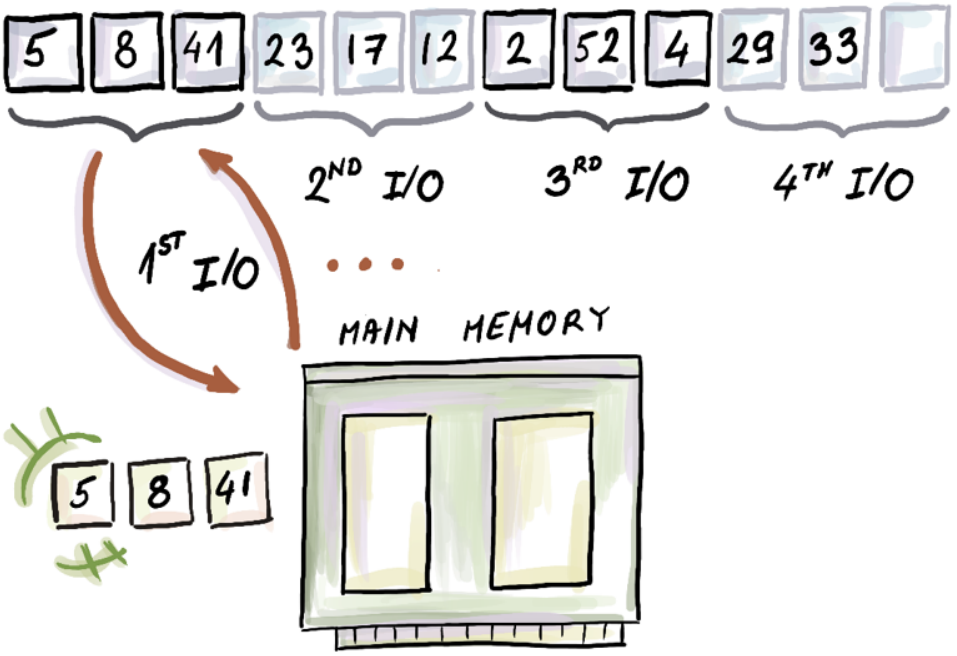



**Figure 9.2: Finding a minimum in external memory. There is a total of 11 elements that occupy 4 blocks of size 3. The last block is not full. In this example, the beginning of the file is block-aligned.**

Here we come to the first difference in runtimes between RAM bounds and external-memory bounds. In the external-memory world, "linear-time" naturally becomes $O(N/B)$. This is the cost of a linear sweep over consecutively ordered data. It is good if we can achieve the "$/B$" part, as in the worst case, if data is not sequentially ordered, or if we are accessing it randomly, reading $N$ elements might require up to $O(N)$ I/Os.

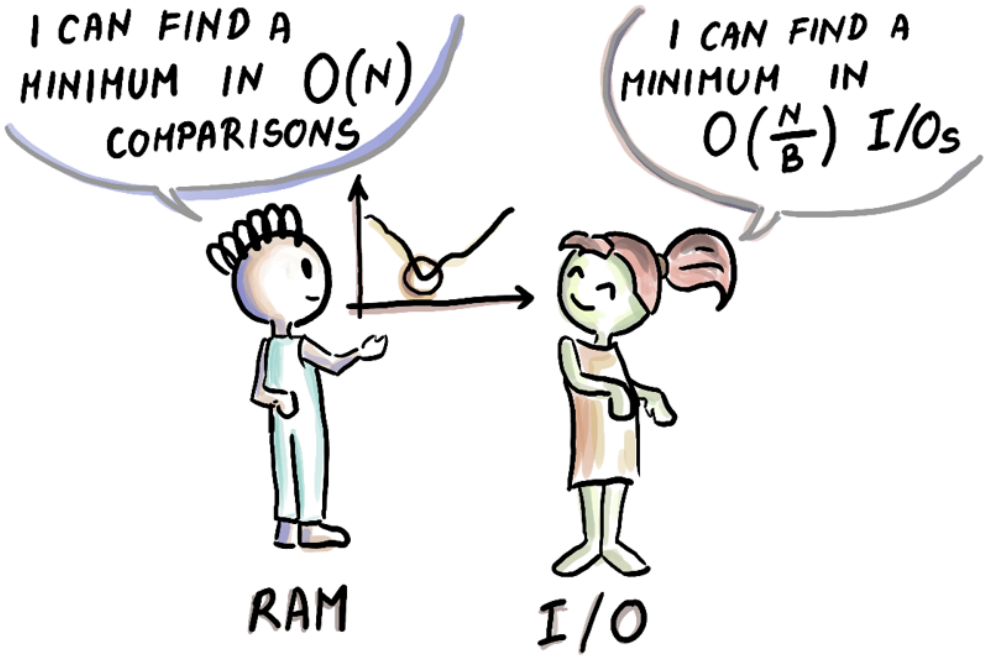



Figure 9.3: The difference in the way bounds look for in-RAM algorithm (left), and an equivalent on-disk algorithm (right.) It is important not to directly compare them as they represent different units. The comparison in the way the bounds look is there to understand how algorithms change as the consequence of different design.

**Exercise 9.1**

Using Python's `open`, `seek`, `read`, `write`, `close`, etc, create a file with 1 billion integers (one per line) and store it on disk.

Then use the same calls to complete the following tasks:

a) Compute the sum of the first 1 million integers.
b) Compute the sum of the randomly chosen 1 million integers.

For both (a) and (b), separately time the task of data transfer (such as `seek` call), and the computational task (summing up data). Compare how much time is needed for each. Also, compare the amount of time required to read sequentially and randomly. If you're feeling like it, in (b) part, time whether `seek` call takes the same amount of time at every point of the experiment, and if not, think about why not.

## 9.3 Example 2: Binary search

Now let's see how we would adapt our good ol' binary search to disk. There is a number of important use cases for doing this --- whenever we need to search in an ordered file for a particular value, binary search comes to mind. Because binary search jumps all over the file, it will be interesting to see how many different block transfers we incur. But first, consider the following use case.

### 9.3.1 Bioinformatics use case

You are working as a computer scientist for a bioinformatics startup, and you are working on the problem of DNA sequencing. In the particular problem you are given, you have been given a large number of $K$-mers ($K$-length substrings of a number of given DNA sequences). Each $K$-mer has its own string value, as well as a small number of key other properties important for further study. Data is laid out in the form of one `(K-mer, property1, property2, …)` tuple per line. The file has grown to be over 1TB. $K$-mers are sorted, deduplicated and what you are interested in is locating particular $K$-mers in the file. Data is static, so you are not interested in modifying the file, or re-organizing data in the file, you just want to be able to query the file in the fastest possible way, so you resort to binary search.

How would binary search work in RAM vs out-of-RAM? Let's see the pseudocode and the related figure, Figure 9.4:

Binary search in RAM:

```
binarySearch(arr, left, right, x)
    while left <= right:
        mid = left + (right - left) // 2
 if arr[mid] == x
     return mid
 elif arr[mid] < x
     left = mid + 1
 else
     right = mid – 1
    return -1
```

Binary search on disk:

```
BLOCK_SIZE = 1024 #A
def binarySearchExtMem(filename, file_start, left, right, x):
    while left + BLOCK_SIZE <= right:
        mid = left + (right - left) // 2
 BLOCK = read_block(filename, file_start + mid, BLOCK_SIZE):
        if BLOCK[BLOCK_SIZE - 1] < x:
     left = mid + 1
 elif BLOCK[0] > x:
     right = mid – 1
        else:
            return binarySearch(BLOCK, 0, BLOCK_SIZE - 1, x)
    BLOCK1 = read_block(filename, file_start + left, BLOCK_SIZE)
return binarySearch(BLOCK1, 0, BLOCK_SIZE - 1, x)
```

```
#A BLOCK_SIZE is the same as B in the text and runtime analysis
```

The in-RAM binary search performs $O(\log_2 N)$ comparisons and also that many cell reads.

The external-memory version of the same algorithm is just slightly modified. In essence, we still wish to perform the same sequence of comparisons, and we pick up blocks that contain items that we wish to compare. (Again, here, we obviate many important details when we perform `read_block(filename, file_start + mid, BLOCK_SIZE)`, because it will most certainly not be the case that `file_start + mid` is always at the block boundary. Once the block is in main memory, in case `x` is smaller than the smallest element in the block, we proceed with binary search on the left side of the array, and in case `x` is larger than the largest element in the block, we proceed on the right side. In the remaining case, we call the in-RAM binary search function to figure out whether the element is in the block or not.

However, the process proceeds only until the file size on which we perform binary search is larger than a block. Once the array size drops under the block size, we input the remaining data that fits into one block and all remaining comparisons are performed in RAM using the original algorithm.

In the example from Figure 9.4, we have that $N = 128$, $M = 64$ and $B = 16$. If we were to only count the number of comparisons in this identical binary search algorithm, we would need roughly 7 comparisons to find the element we are looking for. Because we are mostly interested in counting block transfers, we need approximately 3 I/Os for the search in the worst case.

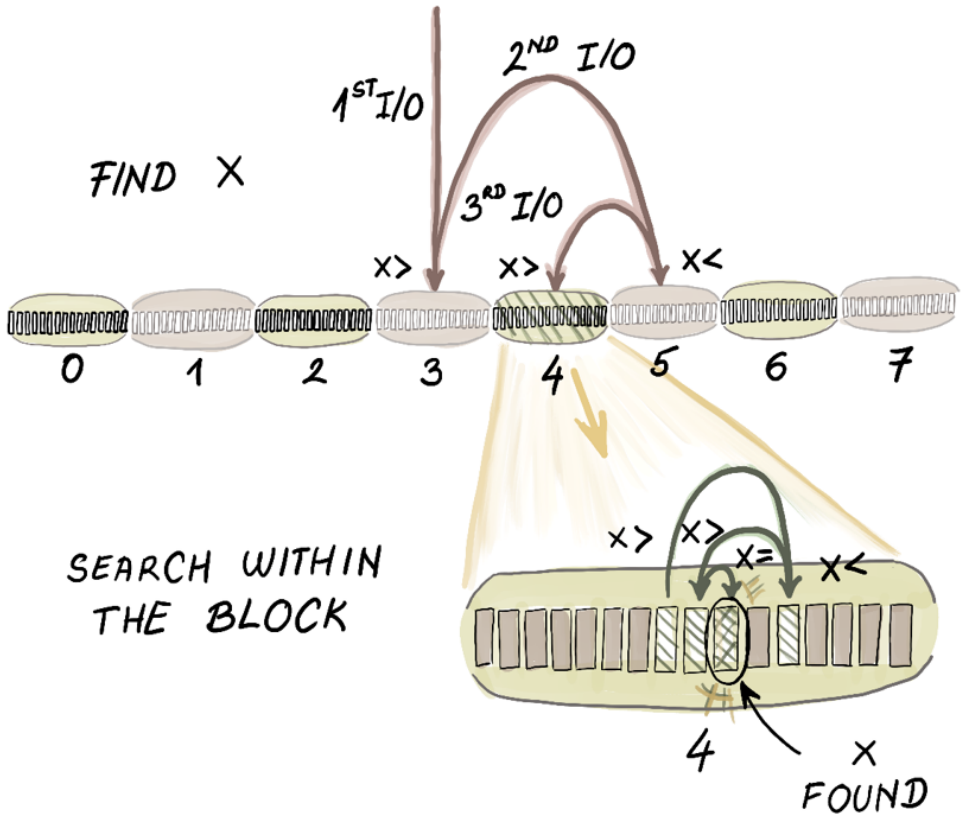



Figure 9.4: Binary search in external memory. The algorithm accesses a separate block for each pivot accessed, except from the last few pivots that all reside in the same block.

### 9.3.2 Runtime analysis

For most of the algorithm execution, in order to perform a comparison(s) that will direct us to the left or the right side of the array, we spend 1 I/O, which amounts to $O(\log_2 N)$ I/Os. However, when the array becomes the size of a block, from that moment on, we perform only one more block input, thus having the last $\log_2 B$ comparisons all use 1 I/O. In total, we get $O(\log_2 N - \log_2 B + 1) = O\left(\log_2 \left(\frac{N}{B}\right)\right)$ I/Os.

Also it is important to analyze what happens when the block enters the main memory. It does not help us, in the asymptotic sense, that we compare $x$ against the two borderline elements of the block, instead of just one element as in the original algorithm. Consider the first block input. Once the algorithm decides, based on two comparisons, to go to the left or the right side of the array, we are still left with $\frac{N-B}{2} \approx \frac{N}{2}$ elements. Still, examining the

borderline elements in the block, or even sequentially scanning the whole block is a smart thing to do considering we already have the block in memory. In practical terms, this tweak would likely make a difference in performance, even though it is not reflected in the asymptotic runtime.

Now that we did things the somewhat clumsy way, here is a more natural way to view the binary search algorithm in external memory. We can consider the algorithm block-based, i.e., instead of looking for the exact position of $x$ in the array, we might think of locating the correct block where $x$ is located (or where it should be, if it is not present.) Therefore the algorithm is performing binary search among the blocks, not among the elements. Once the element is within outside boundaries of some input block, the algorithm execution ends. This also simplifies analysis. We have $O\left(\frac{N}{B}\right)$ blocks, and each step of binary search costs 1 I/O. This gives us $O\left(\log_2\left(\frac{N}{B}\right)\right)$ I/Os, just like before.

In RAM, logarithmic runtime represents the optimal bound for searching, whether it be using binary search on a sorted array, or traversing down a balanced binary search tree of logarithmic depth. The question now, however, presenting itself, is whether the analogous binary search algorithm on disk that runs in $O\left(\log_2\left(\frac{N}{B}\right)\right)$ time is optimal searching mechanism for files and databases on disk?

If data is simply laid out in a sorted array, and we are not allowed to re-organize it or re-group it in any way, then sure, binary search is the best way to go indeed. But if we are allowed to preprocess data in some way (e.g., re-arrange elements) to promote better query time, then we can do a lot better than binary search. Read on to find out how, but before moving on, try this small exercise.

**Exercise 9.2**

**Create a file on disk with 1 billion ordered integers, one per line. Write a Python program to open the file, and to run binary search on it. Use the `readline()`, `seek()` and `tell()` system calls to complete this task. Run it on a few examples. Time different parts of the program and determine what the most time-consuming parts are.**

## 9.4 Optimal searching

To understand why binary search is not our best bet for an optimal external-memory searching algorithm, just consider any block input that takes place during binary search. Even though we are performing a really expensive I/O operation that brings thousands of elements into main memory, we are only effectively using one or only a couple of elements from it. It's like hiring a bus to take you (and only you) to work. We can do better than that.

To see how, see 9.5(a) and $N$ sorted elements in it. Given a block size $B = 3$, what would be the best choice of elements to pack into the block we intend to bring into memory first? Well, the three elements that divide the array into 4 equal parts, and those are, in this specific example, elements 15, 31 and 40. If we created such a block, then after bringing in the

block, we would be left with, not $N/2$-sized array to search, but $N/(B+1)$. We can recursively continue in the same fashion with each remaining sub-array by selecting $B$ equally spaced pivots. To understand how to build these blocks, consider building an implicit binary search tree on top of the sorted array (Figure 9.5(b)), and then, starting from the top of the tree, clumping in top $B$ nodes in the tree to form one node (Figure 9.5(c)). This way, we create a searching data structure such as one in Figure 9.4.d, where a node allows us to branch into $B+1$ (in this example 4, but in the real world, 1000s) different directions, based on just one block input. The higher the branching, the fewer levels in the tree. Each level represents one I/O we need to perform, so a higher branching factor brings down the number of memory transfers.

To understand the difference on this small example, the number of comparisons we need to make to perform binary search is equivalent to the depth of the binary search tree in 9.4b, which is 4 comparisons. Because the last two levels of the tree will be in the same block, then we need 4-2+1=3 block inputs if using common binary search. But if we use the new structure shown in 9.4d, we only need 2 block inputs as this structure only has 2 levels.

The difference seems trivial because our dataset is small, and more importantly, because our $B$ is small. The difference, in fact, is enormous: while the number of I/Os on binary search is $O\left(\log_2\left(\frac{N}{B}\right)\right)$ I/Os, the number of I/Os we need using the structure from 9.4d is $O(\log_B N)$ I/Os.

Usually the base of the logarithm is irrelevant asymptotically if both bases are constants. However, here, the base of logarithm being $B$ makes a tremendous difference.

Consider a dataset of size 1 billion ($\approx 2^{30}$) and a block that can fit 1000 ($\approx 2^{10}$) elements. For example, block size is 64KB, and each element takes 8 bytes. This means that with binary search, we need $\approx 20$ block inputs, whereas with our new structure, we need only 3.

The structure from 9.5d is the cartoon version of what is known as a $B$-tree. B-tree forms the backbone of database indices for the majority of large relational databases. Despite massive data size, $B$-trees rarely exceed the depth of 5 or 6, thereby limiting the number of I/Os we need to do to perform a query.

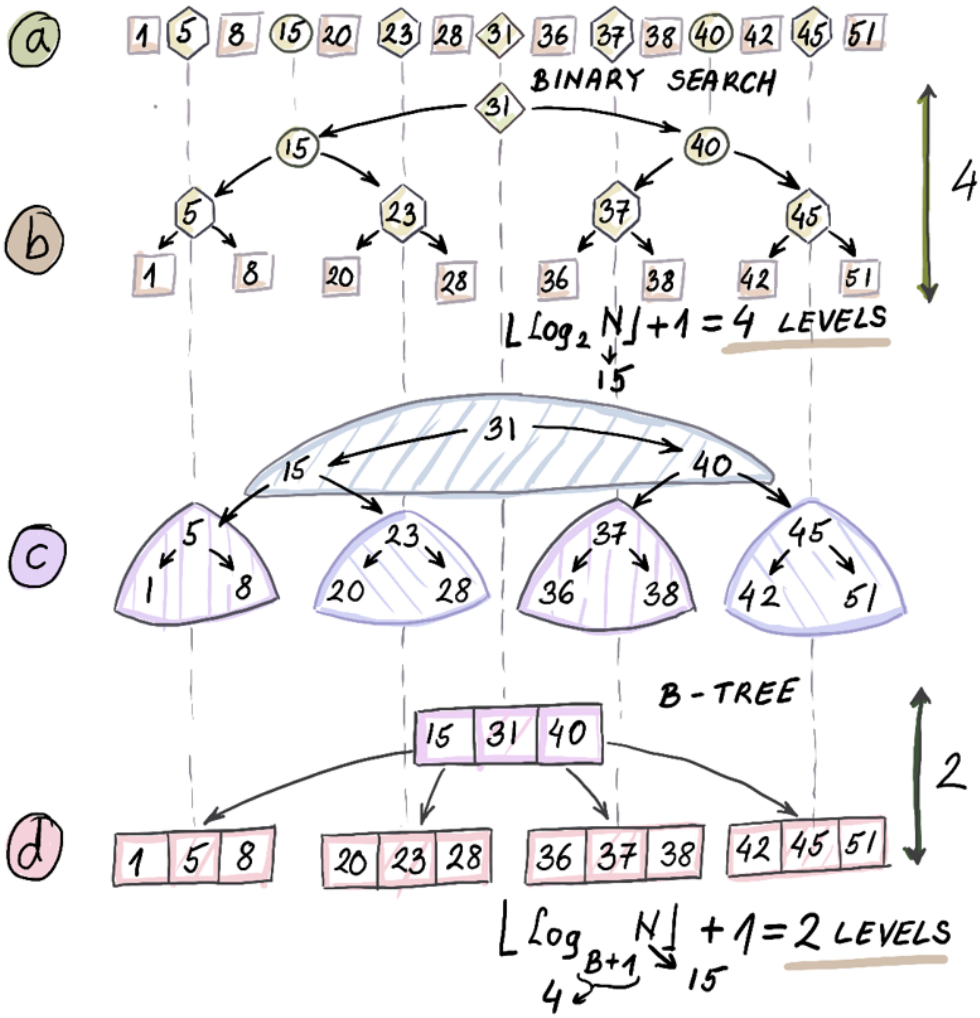



**Figure 9.5: How to transform a sorted array into a structure that enables optimal searching in external memory (a B-tree.) An optimal searching data structure in RAM look something like 9.5(d), where each node has one pivot based on which it is decided how to continue down the tree. The optimal searching data structure in external memory has large nodes (size of a block) that have many elements, because we pull data into memory block by block.**

In the next chapter, we will see many more details on B-tree, its different reincarnations, and other data structures that power relational databases.

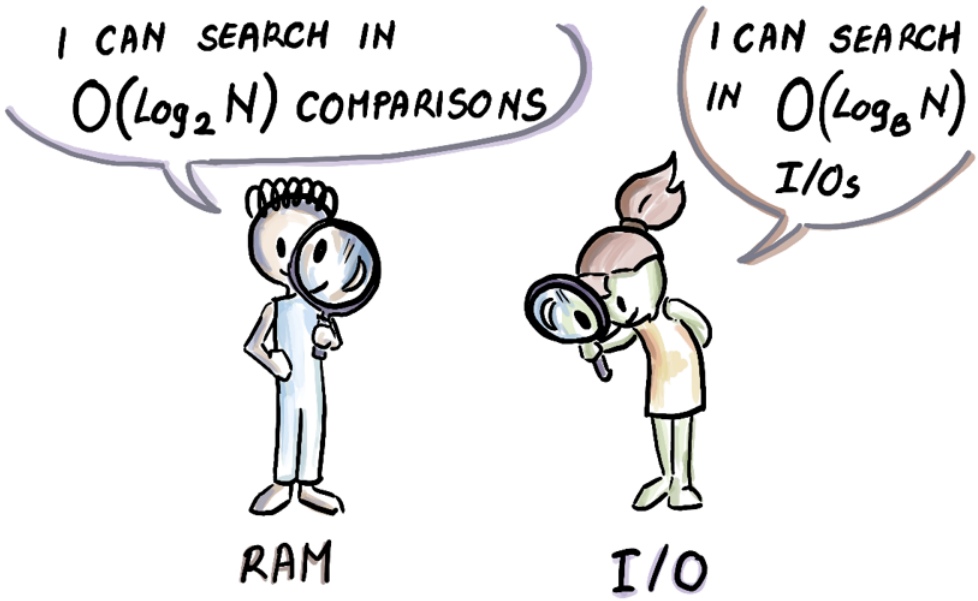



Figure 9.6: Another difference in the way bounds can look in the RAM model and the external-memory model. Often, the runtime with a logarithm base 2 can turn into logarithm base B, as we can examine B different elements simultaneously with the help of block. Base 2 does not always turn into base B, as we will see later.

Notice that so far, the memory size was not extremely important, as long as it could hold at least one block and some extra space to keep a few variables. Both while scanning, binary searching or $B$-searching, we needed to input many blocks, but we needed only one at one time. There are other problems, however, where memory size is important, and where the external-memory algorithm can efficiently make use of it.

## 9.5 Example 3: Merging K sorted lists

Let's consider how we merge data in external memory. A popular problem when merging data from different sources, for example, comes down to merging multiple lists. In main memory, this is often solved with the help of a heap that repeatedly extracts the minima from particular lists. Now we will see how the problem of merging $K$ sorted files can be solved in external memory, and what this teaches us about the nature of merging many lists simultaneously in RAM vs external memory. This will prove important later when we start adapting mergesort to external memory.

### 9.5.1 Merging time/date logs

You are working for a company whose product is a load balancer that supports high-traffic multi-server applications. The application also has a significant security component, and it

collects traffic data at many different sites. You are interested in whether there is any connection between the times and dates when certain attacks occur, so you are analyzing a large number of event logs collected at different sites, each sorted by time/date. The key step is merging the files in the ascending order of the time/date into one gigantic file. Your local computer has the memory of 16GB, and the total size of files is 1TB, shared between 16,000 different files.

The problem stated above translates into merging $K$ **sorted** lists. Let us assume that all lists together have $N$ items. Before starting on the version where files to be merged sit on disk, let's recall how to solve this problem in RAM.

#### RAM VERSION

To most effectively merge $K$ sorted lists in RAM, we can employ a min-heap that holds one representative from each sorted list (so in total $K$ items), and extracts minimum elements one by one. Once an element leaves the heap as its minimum, an element from its same list is supplied back into heap. Individual operations on the heap of size $K$ cost $O(\log_2 K)$, and because each element has to be both inserted into a heap of size at most $K$ at some point, and also removed from it, in total we need $O(N \log_2 K)$ comparisons to solve this problem.

#### EXTERNAL-MEMORY VERSION

Now, let's consider what happens if $N$ is too huge to fit into RAM. In this example, we will assume that even though the total size of every file, and even each individual file might be too large to fit into RAM, the number of files is small enough to accommodate one block of data from each file in RAM, and still leave half of the memory available. In other words, we assume that $K \leq M/2B$. This is not a very strict assumption, as it tolerates up to a million lists in some common hardware setups.

We are going to make use of this fact by reserving one block of memory for each file. In the beginning, we will read in the first block of each file.

Now we can employ the in-RAM solution on the blocks we just read in. We start by having each block insert its minimum to the min-heap. Then we begin extracting minima from the heap. Every time we extract a minimum, we supply the next element to the heap from the same block from which the minimum came from. Once we run out of one block, we supply the next block from that very same file, until we reach the end of the file. Figure 9.7 shows the process, and the pseudocode listing below shows more details.

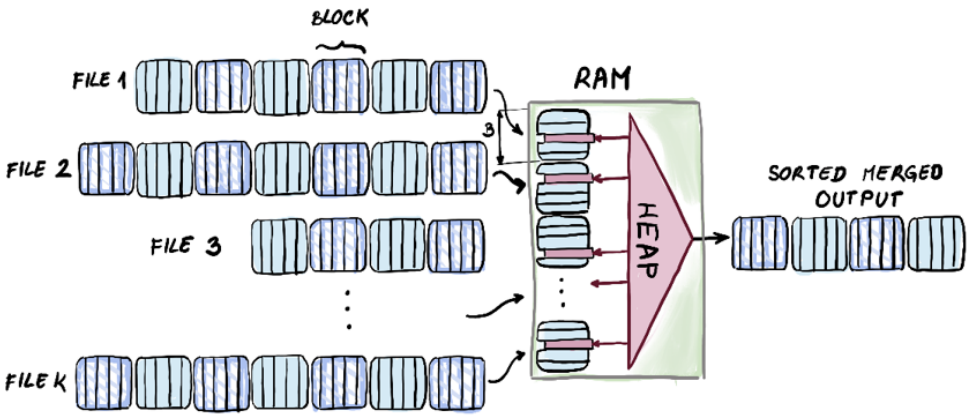



**Figure 9.7: K-way merging of sorted files. Each file has one block of data allotted in main memory, and through that block, each file sends its remaining minimum elements into the heap. Minima is repeatedly extracted from the heap and sent to the output. This scheme works independently on the total accumulated file size, as long as $K = O(M/B)$.**

In the pseudocode below, a list `file_names` contains the names of files and allows us to access the starting positions of each file, and `files_loc` is a list containing the current position we are at within each file. The list `buffer_in` stores individual blocks into main memory ($K$ of them), and each block can be indexed as a list of `BLOCK_SIZE_ELEMENTS` number of elements, so one way to think of `buffer_in` is a 2-dimensional list. The list `buffer_out` stores already merged elements ready for output, and as soon as it contains `BLOCK_SIZE_ELEMENTS` number of elements, we write that block out to `file_dest` location of `outfile_name` that holds the output file name.

The list `file_processed` indicates for each file we are merging, whether we already used up all of its elements. The list `merge_pos` denotes the element we currently point to while merging $K$ block-sized mini lists in RAM. Once the counter reaches the maximum elements in the block, we trigger for the new block to be read from the corresponding file, unless we were already at the end of the file.

```
BLOCK_SIZE = 1024
ELEMENT_SIZE = 64
BLOCK_SIZE_ELEMENTS = BLOCK_SIZE / ELEMENT_SIZE

buffer_in = []
buffer_out = []
file_processed = []
merge_pos = []
file_dest = 0
for i in range(K) #A
    file_processed[i] = False
    files_loc[i] = 0
      buffer_in[i] = readBlock(files_names[i], files_loc[i], BLOCK_SIZE) #B
    files_loc[i]+=BLOCK_SIZE #C

for i in range(K)
    H.insert(tuple(buffer_in[i][0], i)) #D
    merge_pos[i]=1

while(!H.empty())
    element, i = H.extractMin()
    buffer_out.append(element)
    if buffer_out.size == BLOCK_SIZE_ELEMENTS: #E
        flushBlock(outfile_name, file_dest, buffer_out, BLOCK_SIZE)
        file_dest+=BLOCK_SIZE
 buffer_out.clear()
    if(!file_processed[i]): #F
        H.insert(buffer_in[i][merge_pos[i]])
 merge_pos[i]+=1
 if merge_pos[i] == BLOCK_SIZE_ELEMENTS && files_loc[i]!=EOF: #G
           readBlock(files_names[i], files_loc[i], BLOCK_SIZE)
     merge_pos[i] = 0
     files_loc[i]+=BLOCK_SIZE
        elif file_loc[i] == EOF
     file_processed[i] = True
```

**#A K is a number of sorted lists we are merging**
**#B Read the first block of each list into main memory**
**#C Advance the position within file**
**#D Heap H stores (element, list index) pairs**
**#E If there is a full block of merged elements, flush the block**
**#F If we did not use up the file**
**#G If we reached the end of a read-in block, but not the end of file**

Note that in this problem, we make use of a large main memory to simultaneously merge a large number of files. The runtime is fairly simple to analyze, as we perform only 1 I/O for each block. This gives us $O\left(\frac{N}{B}\right)$ I/Os, this cost including all the blocks written out.

This is an interesting artifact of external memory, because in internal memory, we could never merge more than a constant number of sorted lists in linear time. In external memory, we can merge a large (up to $M/B$) number of sorted lists in one linear pass over data.

Now, what happens when we can not allot one block per file in the main memory? We will leave this case for Chapter 11, where we will revisit merging many lists as the backdrop for the optimal external-memory sorting algorithm.

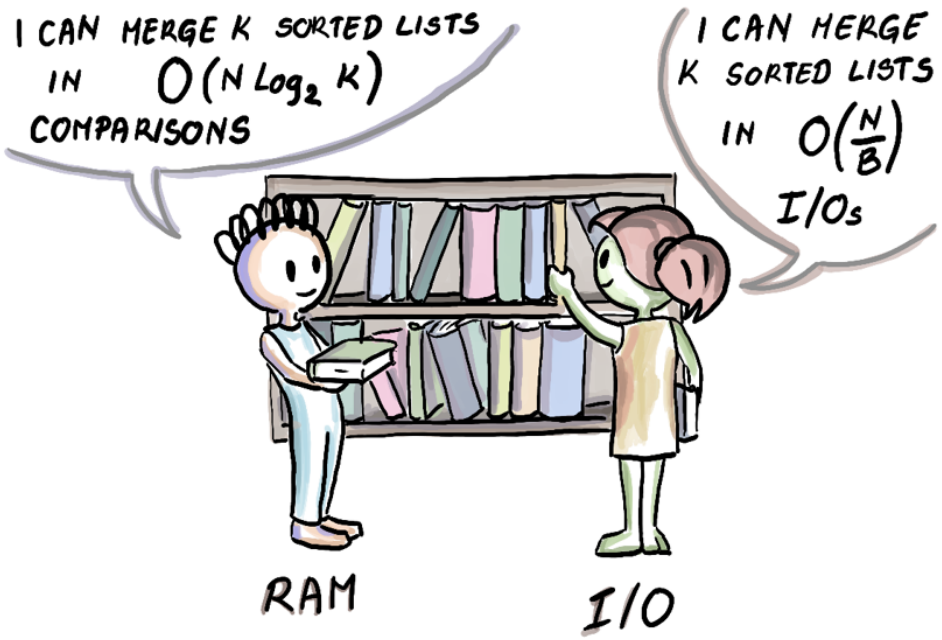



Figure 9.8: Difference in the bounds between merging many sorted lists in main memory and in external memory. In external memory, we can merge many sorted lists with just one sweep over the input data. In internal memory, merging more than a constant number of lists results in more than linear cost in number of comparisons. This same internal-memory cost remains in our external-memory $K$-way merging algorithm, it is just not the most important cost.

We hope that after having seen a couple of examples of how things change when we move from RAM to external memory helps develop an intuition about the aspects of performance captured by the external-memory model. However, this model fails to show some important aspects of I/O-related issues. In next section, we discuss where the differences are and how much they matter for the correct prediction of the efficiency of a real online application.

### 9.5.2 External-memory model: simple or simplistic?

One of the primary things visually springing to mind when looking at the depiction external-memory model (Figure 9.1), is that it only contains two levels of memory: RAM and disk. As we know, the computer hierarchy is much more complex, containing many levels of memory. This should not dishearten us, however. Whatever our dataset size, we can always find the smallest level where data can fit, and term that "the disk", while all other smaller levels

together can form the "RAM". Block size $B$ and memory size $M$ are the parameters that will be affected by where our dataset size fits inside memory hierarchy. For instance, if our data fits in main memory, but cannot fit in cache, then data between those two levels is transferred in cachelines, smaller than disk pages/blocks.

The original external-memory model also accommodates for the possibility of a database being shared among many disks, and in that sense, allowing parallel data transfer. In most algorithms, if there are $P$ disks, we can divide the entire performance cost by $P$.

Some of the simplifying assumptions of the external-memory model, that disk space is infinite, and that computation is entirely free in RAM, does not reflect reality. However, the disk space is extremely cheap, and neglecting the CPU performance will not nearly affect us as much as doing unnecessary disk seeks.

One important drawback of the external-memory model is disregarding the sequential vs. random performance. Whether we read $x$ consecutive blocks, or $x$ blocks in very different locations of disk, the cost remains $x$ I/Os. This is far from truth for most storage technologies. Part of the reason for this is the way things work in hardware, but also, various operating system optimizations. For instance, often times, if we are doing a sequential read of a number of blocks, the operating system will notice this happening and try to pre-fetch the next block. Despite its imperfections, the external-memory model remains the most popular model to date for analyzing the performance of algorithms in data-intensive contexts.

## 9.6 What's next

In this chapter, we started answering the question of how to optimally query on disk, and started introducing the general idea of $B$-trees. However, their different implementations and variants are left for the next chapter. Specifically, we plan to answer the questions such as:

- What variants of B-trees exist and are implemented in real-life systems?
- How can we add, delete, and modify items in a $B$-tree, and what is the mechanics of doing that?
- How can we know that $B$-tree search is optimal in external memory?
- Is $B$-tree also an optimal data structure for inserts and modifications as well as lookups?

This last question will also motivate the introduction of two other data structures we will learn about: $B^{\varepsilon}$-trees and LSM-trees, two data structures oriented towards write-optimized databases.

## 9.7 Summary

- Many data-intensive applications maintain large amounts of data on disk. Large files and large databases residing on disk need a different set of data structural and algorithmic tricks to function efficiently
- The external-memory model is a useful tool in analyzing algorithms for large datasets that can not fit into main memory. This model assumes that all data initially resides on disk of infinite size, and in order to perform computation, chunks of data are brought to and from the main memory of a limited size.
- The external-memory model foregoes the computation cost of an algorithm in order to emphasize the cost of data transfers that tend to be up to 1000x more expensive than computation operations in RAM.
- When scanning sequential data, external-memory algorithm will tend to have an "$/B$" part, suggesting that we packed items into consecutive blocks
- Unlike in internal memory, binary search is not the optimal searching algorithm in external memory, as it does not make a very good use of block inputs and outputs. A better runtime can be achieved by repackaging blocks and building a data structure called $B$-tree.
- One major advantage of having a large main memory is that we can simultaneously merge a large number of sorted list/files, in just one linear sweep, independently of how large the total file sizes are. This process is also a base for the optimal external-memory sorting algorithm we study later.

# 10

# *Data Structures for Databases: B-trees, $B^{\varepsilon}$-trees, LSM-trees*

**This chapter covers**

- **Learning how database indexes work under the hood**
- **Exploring data structures that live underneath MySQL, LevelDB, RocksDB, TokuDB, etc.**
- **Learning what B-trees are and how lookups, inserts and deletes in B-trees work**
- **Understanding how $B^{\varepsilon}$-trees work, and how buffering helps writes**
- **Learning how log-structured merge-trees (LSM-trees) work, and their performance benefits**

Choosing the right database for one's application requires some understanding of the way in which different database engines are built. Specifically, most databases implement indexes to speed up the search of their large and frequently queried tables. Commonly, an index is placed on one of the columns to speed up the searches on that column. To understand all the performance ramifications of creating an index, we need to understand how different databases build and maintain their indexes.

In most basic terms, an index is a data structure, usually separate from the database table itself, that helps to efficiently route the query to the correct row back into the table. Without an index, the search operation boils down to a linear scan of keys in a given column. When dealing with systems that record 10 billion new rows daily or more (or less!), it is safe to say that the linear search will not cut it.

In this chapter, we will learn about three most common data structures used to build *indexes* in modern storage engines. Each data structure is optimized for a different kind of workload in terms of a ratio between lookups and inserts/deletes, and other important operations. As usual, the cost of these operations will be at odds with each other. External-memory data structures lie at the heart of efficient databases, and in our opinion, they are a wonderful

example of all the algorithmic tricks and tradeoffs involved when working with data on disk. But before diving into deeper, more technical levels of database design, let's first see how the basics of indexing work.

## 10.1 How indexing works

An index is most useful when built on a column on which we frequently query. The query needs we pose might need to return the entire row where the key matches the query key, but to locate the record, we only need the key --- a value from the designated index column.

Consider the following simple example of building indices, shown in Figure 10.1. We are given a table that stores the information about employees in a particular department store. In order to speed up the search, we can build an index on the 'Name' column, and to do that, values in this column need to be unique. In other words, if we search for 'John', the index should give us one place where the row with name equal to 'John' can be found in the table (which is why 'Name' solely is a poor choice of a column to build an index on.)

When no single column has unique values, one can use a combination (i.e., concatenation) of columns and build an index on that. We can also build multiple indexes, to speed up searches on two or more independent columns, as shown in Figure 10.1.

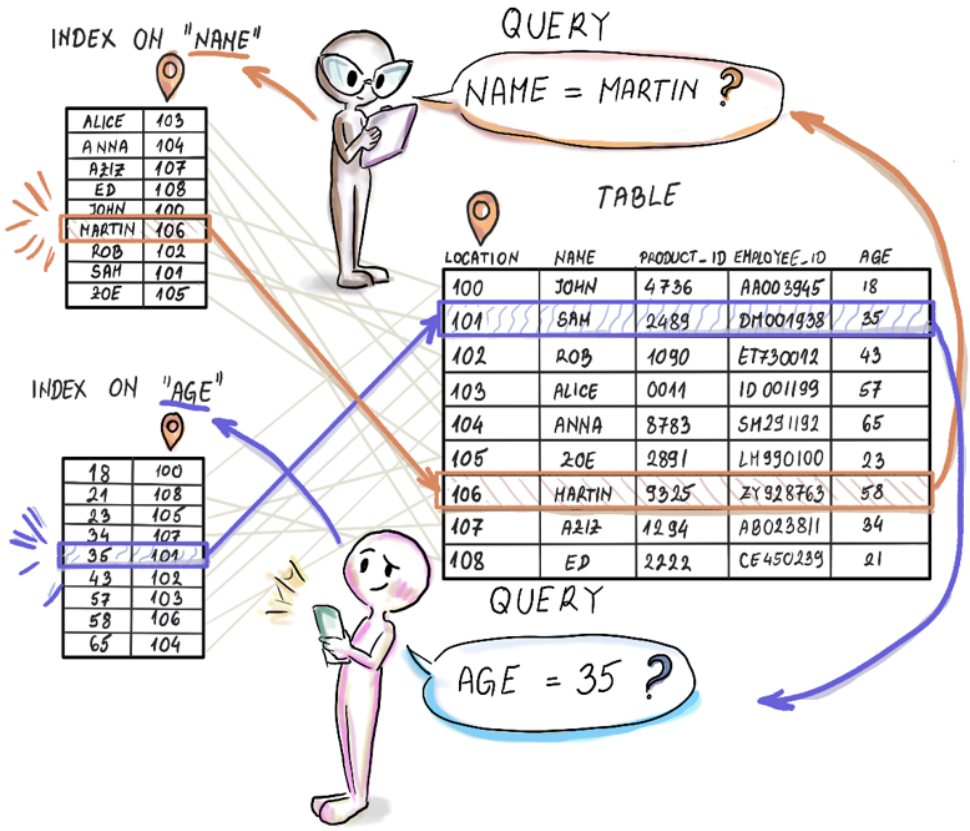



Figure 10.1: Building two indices on top of two different columns in a table.

One of the ways to implement an index is to build a data structure separate from the table itself where keys are lexicographically sorted so the lookup is fast (e.g., a search tree). The key in the data structure would be the column we are building the index on, and the value would be the location of the row in the table containing the given key, as shown in Figure 10.1. The query then works by first quickly locating the key within the index, and then uses the location provided by the value to instantly fetch the corresponding row from the table.

What we just described is sometimes called *unclustered index*, where the actual table data is not being in any way re-arranged when building an index. When there are multiple indices, like in our example, then they must be unclustered. A *clustered index*, on the other hand, orders data inside the table when an index is being built, so there can be only one clustered index per table.

As you might imagine, having one index helps us speed up the search on one column, but it is completely useless when we search on some other criteria. In our example from Figure

10.1, if we want to query by the employee age, we need a whole new index. Technically speaking, we could build an index on each table column just to be safe, however having many indices quickly reaches the point of diminishing returns. Namely, each time the table is updated, i.e., a row is inserted, deleted, or a value in a key column is modified, the index needs to be updated as well. Indices speed up the searches, but they slow down all other operations that modify the contents of the table. Having many indices per table is only a safe bet when we know that data will not be often modified, when search speed is much more important than the update speed, or when there isn't enough data to worry about speed.

The need to update index along with the table teaches us the first important lesson of databases: that the cost of lookups is closely intertwined with the cost of inserts and deletes. This should come as no surprise, as we have seen similar tradeoffs take place with in-RAM indices/dictionaries.

However, in databases, this relationship will be much more complex than what we have seen thus far. As explained in some of the more recent literature[1], insert operation in many systems contains an embedded lookup. For instance, when performing deduplication, an insert operation first asks whether there is a record with a particular key present, and inserts only if the key was not found. In this situation, the worst-case total cost of insert is the cost of a lookup in addition to the modification cost of insert. So, if we tuned the system to have blazingly fast inserts at the cost of incredibly slow lookups, we might get very disappointed in our ultimate insert performance when we see that it includes the lookup cost. Finding the optimal performance in many real-life cases such as this one becomes a delicate balancing act.

## 10.2 Data structures in this chapter

In this chapter, we give most attention to B-trees[2],[3] as they form the backbone of the largest number and most popular database engines out there such as PostgreSQL[4], MySQL[5], etc. B-trees are a lot like binary search trees, but with huge nodes whose node size corresponds to that of a page/block on disk. Because such nodes can fit a large number of keys, B-trees have a large branching factor (branching factor is a number of children a node has, sometimes also referred to as fan-out), guaranteeing a small depth of the tree and thus excellent search performance. Aside from learning the mechanics of B-tree operations, in this chapter we will mathematically show that B-trees exhibit optimal performance for searching in external memory. It is no wonder then that, since the time they were devised in 70s, B-trees have remained the most popular choice when it comes to designing database engines.

B-trees are universally hailed for their fast searches, however, it turns out it is possible to have a data structure with only slightly slower searches and substantially faster insert and

[1] M. A. Bender, M. Farach-Colton, W. Jannen, R. Johnson, B. C. Kuszmaul, D. E. Porter, J. Yuan and Y. Zhan, "An Introduction to Be-trees and Write-Optimization," ;login: , vol. 40, no. 5, 2015.
[2] D. Comer, "The Ubiquitous B-Tree," ACM Computing Surveys, vol. 11, no. 2, pp. 121-137, 1979.
[3] R. Bayer and E. M. McCreight, "Organization and Maintenance of Large Ordered Indices," in Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control, Houston, Texas, 1970.
[4] P. G. D. Group, "PostgreSQL 13.3 Documentation," [Online]. Available: https://www.postgresql.org/docs/13/index.html. [Accessed 08 August 2021].
[5] O. Corporation, "The Physical Structure of an InnoDB Index," Oracle, 2021. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/innodb-physical-structure.html. [Accessed 08 August 2021].

delete performance than a B-tree. $B^\varepsilon$-tree[6] is an alternative data structure behind storage engines such as TokuDB that have more recently gained a lot of popularity due to their superior insert/delete performance. With data becoming more dynamic, many applications today need to maintain a much faster insert/delete throughput rate than what B-tree-based databases can offer, while maintaining fast lookups. For these types of workloads, $B^\varepsilon$-trees are an ideal data structure. $B^\varepsilon$-trees manage to keep the same asymptotic cost of lookups while improving (asymptotically) on the cost of inserts/deletes. In other words, the slowdown in their lookups is somewhat felt, but the speedup in inserts and deletes is felt much more.

The secret sauce in the $B^\varepsilon$-tree performance is that the inserts and deletes are not executed in an immediate fashion, like in B-trees, where a sole insert/delete/modify, immediately travels down to the leaf of the tree and modifies it (B-trees just take life way too seriously.) In $B^\varepsilon$-trees, insert and deletes act as messages that are buffered and delayed on their way to the leaves. The idea behind delaying operations is collecting enough insert/delete messages at one node that are headed in the same direction, and then sending them together in one memory transfer, an idea not dissimilar to carpooling. By carpooling inserts and deletes, a $B^\varepsilon$-tree can make a great use of its I/Os to process as many inserts and deletes as possible. This is in contrast to B-trees, where a single element descending down the tree triggers many expensive I/O operations just for its own benefit.

Lastly, we discuss LSM-trees[7]. LSM-trees are data structures behind LevelDB, RocksDB, Cassandra[8],[9], and some other engines that solely care about high-performance inserts that run faster even than those in $B^\varepsilon$-trees. The benefit of LSM-trees is that they make use of a very fast sequential-scan feature of disks. While B-trees and $B^\varepsilon$-trees access random blocks while descending down the tree, LSM-trees organize their data in sequential runs that are occasionally merged as in mergesort. Merging of two runs can be done at the speed of scanning ($N/B$ for all elements, or $1/B$ per element) which is optimal, and occasional merging between runs makes sure that we do not end up with too many runs to have to query when the query time comes. Regardless, the lookups in this data structure do take a hit, but this can be somewhat improved --- by Bloom filters.

## 10.3 B-trees

B-trees are a natural extension of binary trees to external memory: where binary search trees use one key per node to direct the search/insert/delete in two different directions to the next level of the tree (<key and >key), B-trees use many more keys per node. In the rest of the text, we sometimes use the term 'pivot' interchangeably with the key in a B-tree node.

Specifically, a B-tree of order $d$ has in each node at least $d$ keys (with the branching factor $d+1$) and at most $2d$ keys (with the branching factor $2d+1$.) Branching factors of nodes can

[6] G. S. Brodal and R. Fagerberg, "Lower bounds for external memory dictionaries," in Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, Baltimore, Maryland, 2003.
[7] P. O'Neil, E. Cheng, D. Gawlick and E. O'Neil, "The Log-Structured Merge-Tree (LSM-tree)," Acta Informatica, vol. 33, no. 4, pp. 351-385, 1996.
[8] C. Luo and M. J. Carey, "LSM-based storage techniques: a survey," VLDB Journal , vol. 29, p. 393–418, 2020.
[9] Y. Matsunobu, S. Dong and H. Lee, "MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph," Proceedings of the VLDB Endowment, vol. 13, no. 12, pp. 3217-3230, 2020.

differ depending on how many keys they house. The only node that does not need to obey the requirement on the minimum number of keys is the root --- the root can have fewer than $d$ keys, but not more than $2d$ keys.

The value of $d$ in a B-tree node determines the size of the node, as the node always has the space to accommodate $2d$ keys and $2d + 1$ pointers to the subtrees below regardless of how many keys it actually stores. As we will see, nodes will commonly have some empty space.

To understand the internal structure of B-tree nodes, see Figure 10.2. The Figure shows two nodes in a B-tree of order 2. On the left, we see a minimally filled node with 2 keys and 3 pointers to children that are non-null. On the right, we see a full node, with 4 keys and 5 pointers to children that are non-null.

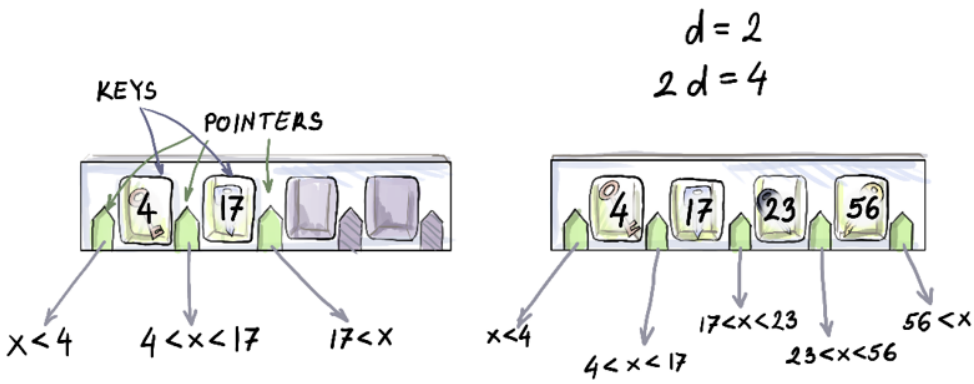



**Figure 10.2: A structure of a node in a B-tree. Each node has space to accommodate 2d keys and 2d+1 pointers, no matter how full it is. The pivots direct the search, that is, when searching for the element x, we compare it with the pivots, and the next node (i.e., pointer to it) is chosen based on the value of the key. The node on the left is minimally filled, and the node on the right is maximally filled.**

Each key inside a B-tree node, aside from the value used to route the query to the next level, also contains a pointer to the row location into the table, as we showed in Figure 10.1. We hide this detail from Figure 10.2 onwards and only show the key, as we will not refer to original database tables in the remainder of the chapter. We will just assume that the moment we locate a key in the node of the tree that is the answer to the query, we will have the necessary information to automatically jump to the table to fetch the rest of the record. However, it is important to understand that there is a great deal of bookkeeping and wiring that takes place within each node of a B-tree that consumes a large fraction of its space. These considerations are important when choosing the node size. Generally, throughout this chapter, we will assume that node size is related to the block size $B$, so $d$ can be considered some fraction of block size, e.g., $B/2$, $B/4$, etc.

Consider the case when the block size $B = 1024$. If pointers leading to the next level of the tree take up roughly half of the space, then the remaining half is left for the keys and

pointers to the table. Let us assume that this space is again evenly split, so keys take up $B/4$ space, and pointers into the table take up remaining $B/4$. Even though the original node houses 1024 words and could theoretically store 1024 keys, it in fact stores up to 256 keys. Still a bargain, considering that such a B-tree with all nodes at full capacity can in 4 levels store over 4 billion keys in the leaf level. In practice, many B-trees have much larger nodes, sometimes on the order of megabytes.

### 10.3.1 B-tree balancing

To maintain a low search cost, just like with balanced binary search trees, we need to be sure that no single root-to-leaf path in a B-tree becomes too long. B-trees have this problem nicely solved, in that every root-to-leaf path is always of the same length. B-tree is flat from the bottom, i.e., all leaves reside on the same level.

Insert and delete operations can violate the size limitations of a node, such as when inserting into a full node or deleting from the minimally occupied node. When this happens, we might split an overly full node into two and redistribute the keys, or we might join two nodes that do not have enough keys. This might trigger the changes up the tree, requiring splits/merges on upper levels, where in the extreme case, the tree could ultimately grow or shrink from the top. That is, we might end up splitting the root into two nodes and imposing a new root above, or we might bring down the existing root to the lower level, merging it with nodes below it.

If this sounds confusing, don't worry, we will visualize and describe these operations in detail soon. For now, it is important to understand that B-tree depth grows and shrinks from the top, while leaves all stay on the same level on the bottom. Contrast this with the binary search trees, where a new element is inserted as a leaf on the bottom of the tree, without imposing that all leaves be at the same level.

Now let us see the mechanics of lookup and insert/delete operations.

### 10.3.2 Lookup

The lookup algorithm is fairly simple and it mimics the logic of a lookup in a binary search tree. To perform the lookup, we first read in the root node of the B-tree, and find where the query key belongs in the sorted order among the root keys. In case our key equals one of the root keys, we return `True`, otherwise we follow the appropriate pointer down the tree applying the same algorithm recursively until we either return `True` or reach a null pointer and return `False`. If the element is found before reaching the leaves, then there is no need to go further down the tree. Depending on the implementation, we might prefer to have a stored value returned instead of a Boolean, but the idea is the same.

Because the upper levels of the tree are often small enough and reside in RAM, we might save some I/Os while searching the upper levels of the tree. Most keys, however, will reside in the lower levels, so the probability of the queried key be in one of the nodes in RAM is fairly small.

In the worst case of a lookup, we might need to read in every block on the root-to-leaf path, making lookup cost $O(\log_d N)$ for the B-tree of order $d$. Because generally, we will make an assumption that $d = \theta(B)$, that means our worst-case lookup cost will be $O(\log_B N)$. The fact that some nodes will be emptier than others will not disturb the asymptotics, as the branching factor will still be $\theta(B)$.

The worst case will happen when an item we are searching for is in the leaf level, so we need to examine every block on the root-to-leaf path to the element in order to find it. The worst case occurs often when you consider that the majority of elements reside in leaves, and also, that the worst case occurs also in the common case when the lookup reports the element as not present.

### 10.3.3 Insert

Insert is somewhat more involved than a lookup. First, we perform a lookup to find the leaf where a given element should be inserted (we always insert into a leaf). If the leaf is not full (has $< 2d$ keys), then the element is simply added to the right position in the appropriate leaf and the modified node is written back to disk. Consider an example shown in Figure 10.3 below, where 80 is inserted into a B-tree of order $d = 2$. Aside from adding the element to the leaf, no other changes to the tree are required.

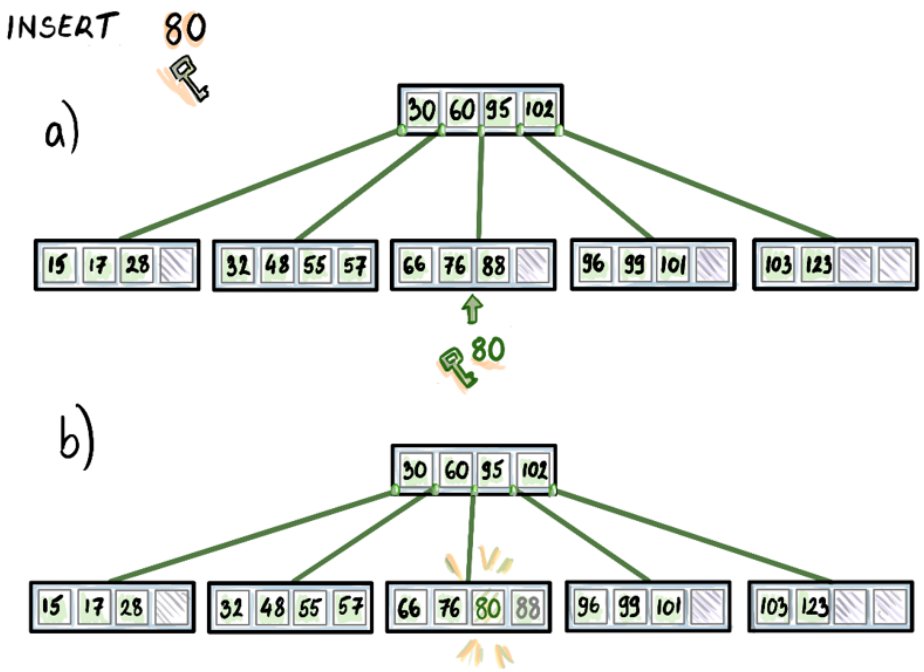



Figure 10.3: Insertion into B-tree when the designated leaf is not full.

However, it might happen that the designated leaf is full (already has $2d$ keys), in which case after the placement of the new key, the node will have $2d + 1$ keys. Because it now violates the size limit, the overfull leaf is broken into two leaves in the following way: the left leaf will contain the smallest $d$ keys, the right leaf will contain the largest $d$ keys, and the median key will be inserted into the parent node to serve as the separator of the two newly created nodes. This process might trigger further splits up the tree. For example, if all nodes on the given root-to-leaf path are full, all nodes will be split going up to the root, including the root, and the tree will grow in height by 1.

Such is the example shown in Figure 10.4 where we insert 69 into the B-tree of order $d = 2$. After 69 is placed into the leaf, the leaf has 5 elements, and it splits into two leaves that will be separated by the median 76; 2 elements go to the left leaf, and 2 elements go to the right leaf, and 76 is inserted into the parent to serve as a separator between the two newly formed leaves. In this case, the parent is the root of the tree and it is also full, so the insert triggers a new split. The root splits into two nodes each with 2 keys, and the median gets promoted to a newly created root.

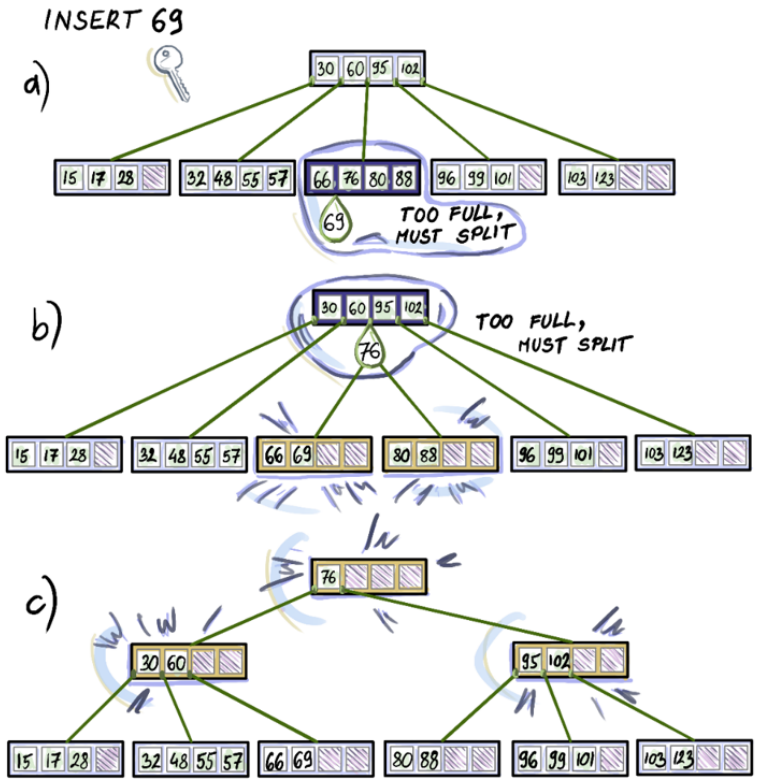



Figure 10.4: Insertion into a full node that results in B-tree growing in depth by 1.

What is the cost of the insert operation? We can divide the total cost of insert into the lookup cost --- cost required to find the place where to insert, and modification cost --- node splits, redistributing the keys, etc. The lookup cost is always $O(\log_B N)$ I/Os as we need to reach the leaf every time when performing a lookup required by an insert. The modification cost varies depending on how far we need to perform splits up the tree, but in the worst case, this cost is $O(1)$ I/Os per level of the tree. Creating a new node and moving over some of the keys does not require more than an access to and writing to a constant number of blocks. Therefore, in the worst case, the modification cost does not asymptotically degrade the total insertion cost as it is also at most $O(\log_B N)$.

Note, though, that because B-tree has very large nodes, the gap between a minimally filled node ($d$ keys) and a full node ($2d$ keys) is quite large. This means that node splits triggered by inserting into a full node do not happen quite as often. Some insertion patterns might trigger many node splits, for example, a lot of inserts into the same leaf will make a B-tree suffer. This is why many practical B-tree implementations attempt to recognize when this type of inserts takes place, and treat them differently (they are inserted in one big batch, etc.)

### 10.3.4 Delete

Deletion of an element from a B-tree is somewhat analogous to the insert. Deletion, however, has two following cases:

1. We are deleting a key from an internal node
2. We are deleting a key from a leaf

The deletion algorithm reduces both cases to the 2nd case in the following manner: if the key to be deleted sits at an internal node, it gets removed from its node and its successor is placed in its location. To remind ourselves, a successor of an element $x$ in the tree is the smallest element in the tree that is larger than $x$.

You might want to stop here and assure yourself that every key in a non-leaf node in a B-tree a) must have a successor, and b) that a successor of an arbitrary key in an internal node must reside in a leaf. We can find a successor of a key $x$ in an internal node by following the pointer $p$ just right of the key, and finding a minimum of the subtree pointed to by $p$. Visually, by following $p$, we make one right, and then we keep making left until we reach a leaf. The leftmost (smallest) key in that leaf is $x$'s successor.

By replacing an element with its successor (the way 60 gets replaced by 66 in Figure 10.5), we maintain the same number of keys in the internal node from which the deleted element comes, and also we uphold the lexicographical order of elements in the tree, so no problems there. However, we just lost an element from a leaf.

How do we delete from a leaf? If the leaf $y$ contains more than $d$ keys, then the element can be safely removed from the leaf and that's it (see the example of deleting 99 in Figure 10.5.)

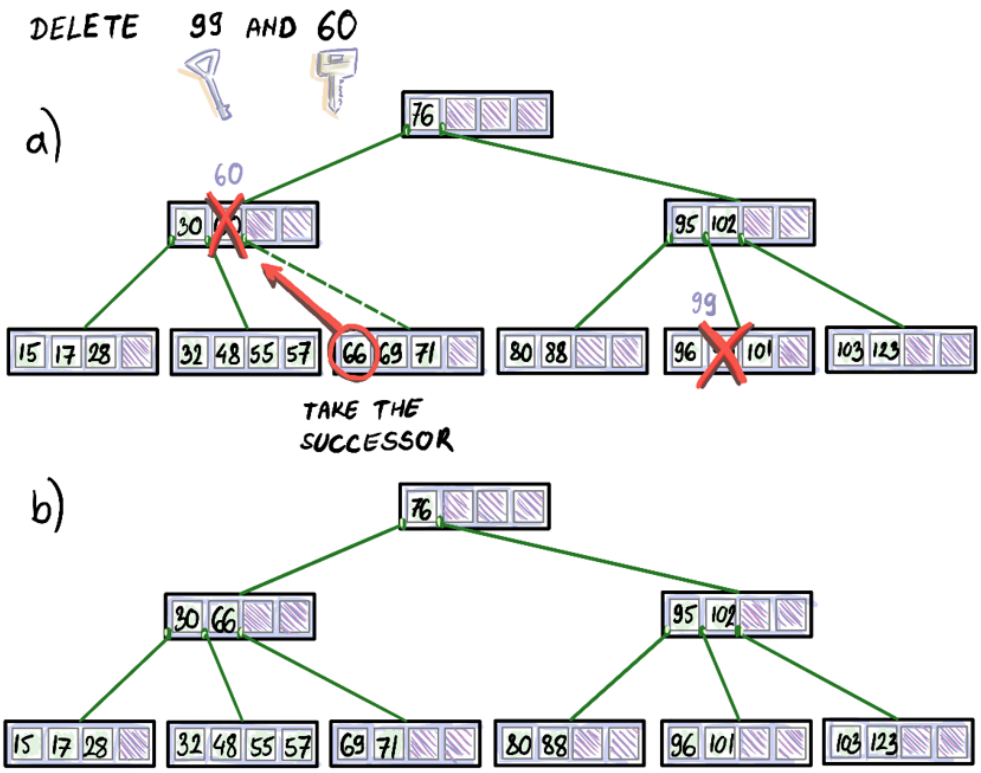



Figure 10.5: Handling deletions from an internal node vs. a leaf.

On the other hand, if the leaf $y$ has $d$ keys, removing a key will cause an underflow. We then turn to left/right neighbor of $y$ to see whether we can borrow some keys from it. If the left or the right neighbor has more than $d$ keys, then we can borrow at least one key to make up for an underflow.

Ideally, if one of the neighbors has ample keys, we would like to evenly split keys between that neighbor and leaf y, but by re-arranging elements among leaves, the separator also changes. In the example shown in Figure 10.6 where we delete 69, after 69 is removed, the left node contains 32,48,55,57, the separator is 66, and the right node contains 71. We re-arrange these elements so that the left node contains 32, 48 and 55, the separator becomes 57, and the right node contents become 66, 71.

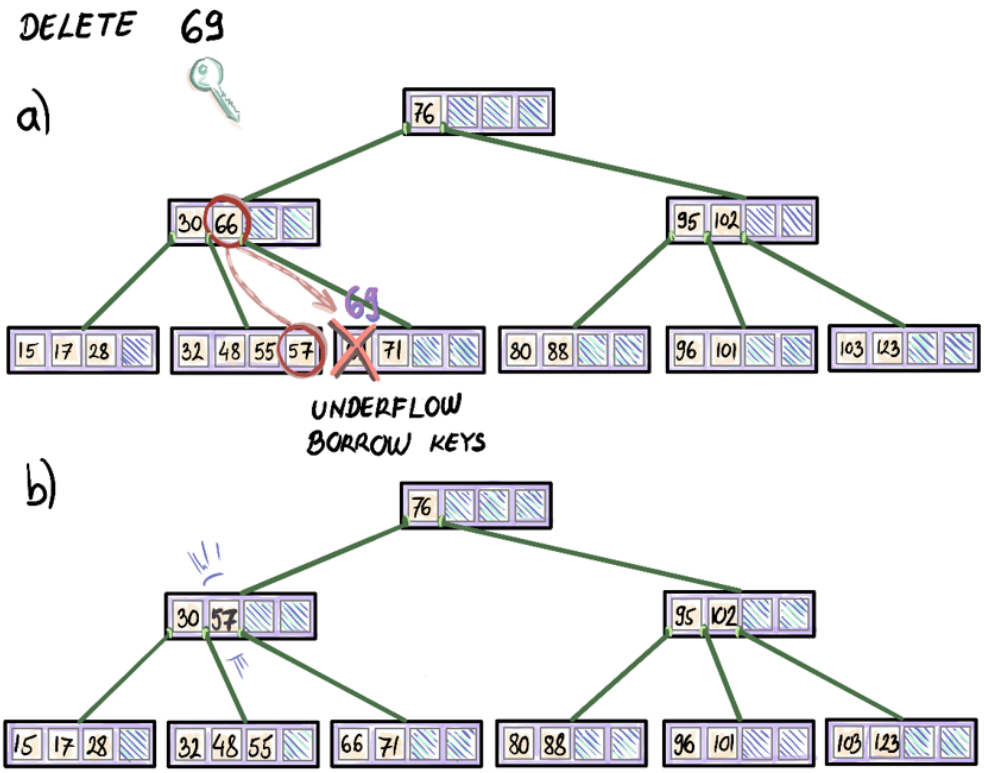



Figure 10.6: Deletion of an element can cause an underflow. If neighbors are not at their minimum capacity, the node borrows keys from one of the neighbors.

It might not be apparent on a B-tree of order $d = 2$, but redistributing elements evenly between two leaves is very important (e.g., consider nodes that have thousands of keys). By redistributing evenly, we are pushing farther into the future the next potential redistribution of keys.

It might happen that both neighbors are at the minimum capacity and can not lend any keys. In that case, we concatenate leaves. We concatenate the leaf $y$ (now it contains $d - 1$ keys) with a neighbor of our choosing (contains $d$ keys) and the earlier separator between the two leaves to form a new node that contains $2d$ keys, thus forming a full node. By bringing the separator down, we effectively delete a key from an internal node, which might trigger further redistribution of keys or node concatenations up the tree.

Consider an example in Figure 10.7 where the deletion of 88 causes concatenation at the leaf level. Once the leaf is concatenated with its right neighbor, the previous separator of the two leaves (95) comes down into the new concatenated node. That triggers an underflow on the

2nd level of the tree, causing another concatenation to happen, and ultimately reducing the depth of B-tree by 1.

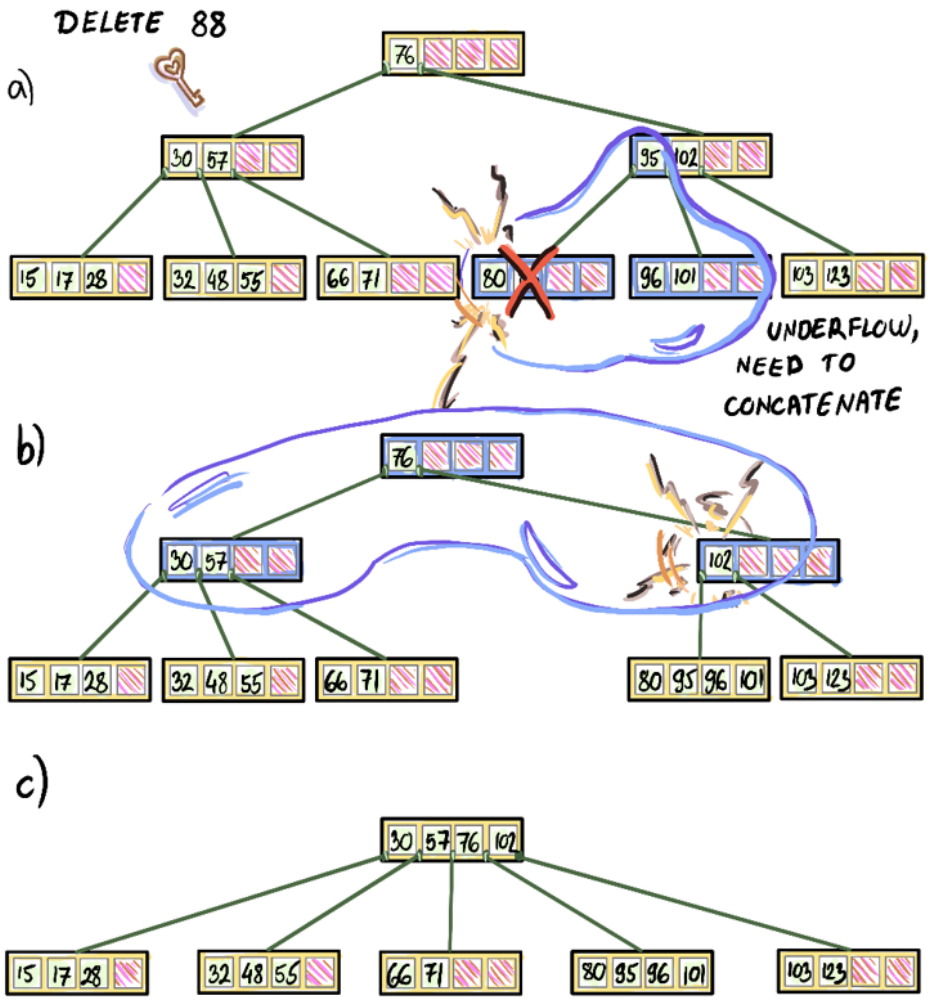



**Figure 10.7: Deletion of an element from a leaf might cause a node concatenation, if neighbors are at their minimum capacity. The concatenations can propagate up the tree and ultimately reduce the tree height by 1.**

The deletion, just like insertion, requires a lookup of a key to be deleted, and potentially node modifications. Similarly to insertions, the modification cost to the tree during a deletion does not asymptotically endanger the total cost, and it amounts to $O(\log_B N)$.

Even though we analyze operations asymptotically in a B-tree, the depth of a B-tree is rarely above 6 or 7. The upper levels of a B-tree can also often fit in main memory. For instance, for the node where $d = 512$, and a total node size is on the order of a couple of kilobytes, a standard RAM memory might fit 2 or 3 top levels of the tree. This saves us I/Os accesses to be used only for the lower 2-4 levels of the tree.

### 10.3.5 B⁺-trees

Most implementations of B-trees in use today are in fact $B^+$-trees. The main difference between the plain B-trees we just described and the $B^+$-trees is that $B^+$-trees hold all of their data in leaves. The internal nodes contain keys whose main purpose is to route the query to the correct leaf but they do not necessarily reflect the contents of the actual dataset. This also means that all queries cost $\theta(\log_B N)$, as even if we encounter a queried key in an internal node during search, we still continue the search all the way to the leaf.

There are a couple of reasons why we benefit from this design. First of all, the leaf level is organized in a linked list as shown in the Figure 10.8. This allows fast sequential access over sorted order of data and fast range queries. Consider how in a classical B-tree, a range query or a need to scan all data in the sorted order triggers an in-order traversal of a tree, which jumps up and down various tree levels. If the tree is laid out on disk in the level-by-level fashion, then switching between levels invokes an inevitable random access. Range queries and the need to output all data in one fast scan is important in systems such as databases and file systems. Therefore, the ability to provide fast traversals in a $B^+$-tree comes particularly handy.

In addition, not maintaining pointers to the actual data in the internal nodes frees a lot of space to store more keys. This gives a higher branching factor and a lower depth, thus resulting in a reduced number of I/Os for common operations than in the classical B-tree.

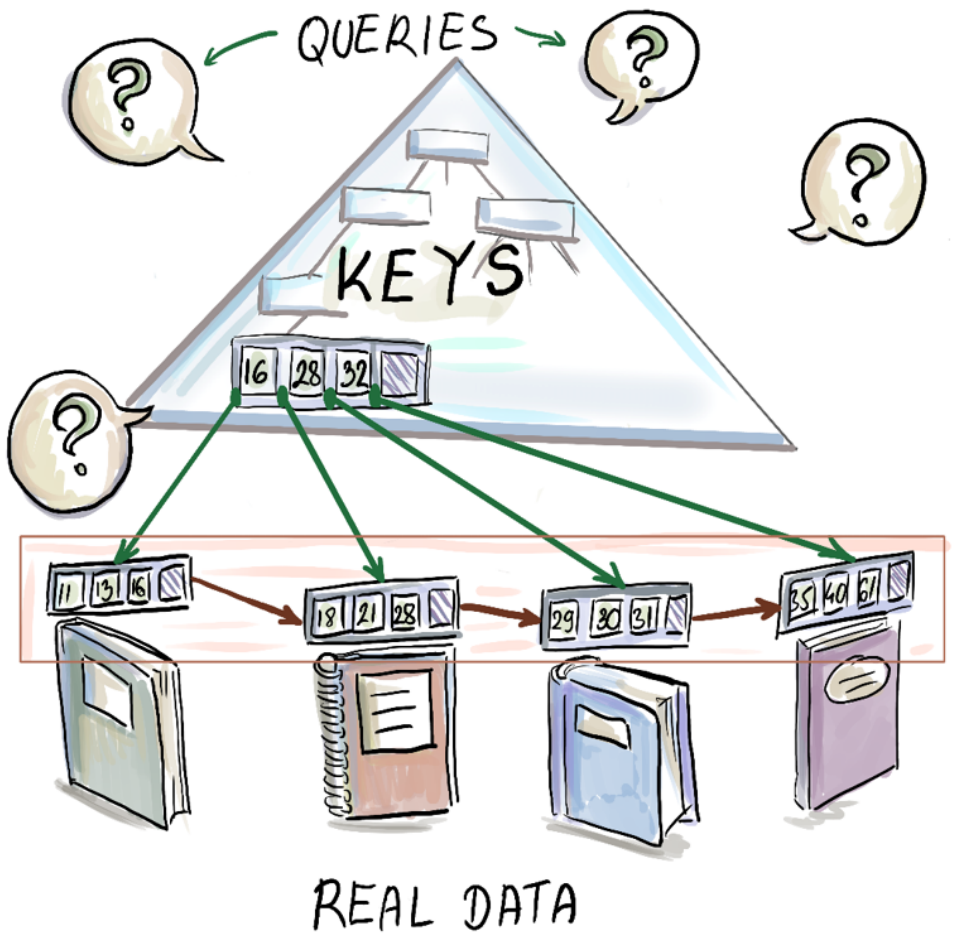



Figure 10.8: A $B^+$-tree organization. Internal nodes are simple and contain keys that route queries to the leaves, where more detailed information on each key is contained.

### 10.3.6 How operations on a $B^+$-tree are different

Initially, a $B^+$-tree can be built in a way that keys from internal nodes are duplicated versions of actual data keys. In other words, in the beginning, all the items existent in the leaves are also existent in the internal nodes. However, when an item is deleted, it is only deleted from a leaf, but it stays in the internal node as a guidepost (unless there is no need for that separator key anymore due to node merge, etc.) For example, if the item 28 were to be deleted from the $B^+$-tree in Figure 10.8, it would only get removed from the leaf level, but would stay as a separator between the two leaves in the internal node.

Similarly, when during a more complicated insert, a node is split into two, and a key from the leaf is promoted to the internal level, in a $B^+$-tree, it is duplicated so that it still stays on the leaf level and it is promoted to the internal node level to serve as a separator.

Searches always go all the way down to the leaf level of the tree, so we can not have the case where a search incurs zero I/Os, because the element was found in one of the higher levels of the tree that were cached in RAM. On the other hand, once a particular element is found, the successor operation runs in amortized $O(1/B)$ time, because every $\theta(B)$ operations, we fetch a new block, and all other times, the successor operation is free. As we will see in the later part of the chapter, a $B^+$-tree is a useful component to use to build larger data structures where components are occasionally merged. In this case, the ability to quickly scan down all the data of two components, and merge them in a mergesort-like fashion significantly boosts the performance.

### 10.3.7 Use case: B-trees in MySQL (and many other places)

B-trees form the foundation for many database engines such as PostgreSQL[10], MySQL[11] and many others. File systems, such as Apple's filesystem HFS+[12], and BTRFS by Linux[13] all use B-trees. If your company is running some kind of database, most likely, it is a B-tree-based database. Many of these implementations are actually $B^+$-trees.

As an example (and for a change), consider an online application that does not deal with an enormous dataset. The website contains the information on all self storage facility records in the US. The database contains about 50,000 self storage facilities, but in fact, each facility has a large number of storage types that can be rented (different categories of storage dimensions, different features such as whether the facility is climate controlled, whether there is elevator access, whether there is a promotion on the price), so in effect, we have about 10 million individual records including historical records.

Users can go to the site to check the availability of particular types of storage in their neighborhood and they can filter using various criteria (e.g., by zip code, storage size, etc.) Every day over 100,000 new rentals are signed, so we can assume an even larger number of queries are posed to the site.

On the other hand, modifications to the database do happen, but not nearly at the same pace as queries --- for instance, a facility might close or a new one might open; also, pricing information can change, but all this happens at the rate of a couple times a week. To make searches fast, we should store the database as a B-tree, and we can build indices on different columns in the table (e.g., zip code.)

[10] P. G. D. Group, "PostgreSQL 13.3 Documentation," [Online]. Available: https://www.postgresql.org/docs/13/index.html. [Accessed 08 August 2021].
[11] O. Corporation, "The Physical Structure of an InnoDB Index," Oracle, 2021. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/innodb-physical-structure.html. [Accessed 08 August 2021].
[12] A. Inc, "TechnicalNote TN1150, HFS Plus Volume Format," Apple Inc., 2016. [Online]. Available: https://developer.apple.com/library/archive/technotes/tn/tn1150.html#BTrees. [Accessed 08 August 2021].
[13] P. LLC, "Percona," Percona LLC, [Online]. Available: https://www.percona.com/software/mysql-database/percona-tokudb. [Accessed 08 August 2021].

In the next section, we touch upon the mathematical foundations of optimality behind B-tree searches. This section is primarily intended for mathematically curious readers and otherwise can be skipped.

## 10.4 Math bit: Why are B-tree lookups optimal in external memory?

To answer the question of optimal way to query in the external memory, let's back up to RAM and the optimal searching in RAM. We know that binary search trees (as well as binary search on a sorted array) can perform queries optimally in $\sim\log_2 N$ comparisons, in other words, the lower bound for searching in RAM is $\Omega(\log_2 N)$ comparisons. But how do we know this? In other words, how do we know that someone won't come along one of these days having invented a new algorithm faster than binary search?

To answer this, we need to produce a lower bound argument that places all potential algorithms under one umbrella of procedures that perform a sequence of comparisons (e.g., $a < 3$?), whose answers can be 'Yes' or 'No', and analyze the amount of information we learn from each answer. That is, we are operating in the world of algorithms that can only perform comparisons (otherwise hash tables can beat our searching lower bound). Then we compute the minimum number of questions that this generally defined procedure has to ask in order to solve the problem.

To illustrate the point, let's turn to a children's game that might feel familiar: say that you imagined a number $x$ between 1 and 1,000,000, and your friend is trying to guess the number. They are allowed to ask questions such as: Is $x$ smaller, larger or equal to 30,000?, and you need to give them a truthful answer. If $x$ equals the number they mention, the game stops, otherwise, you respond whether their guess is too high or too low and the game continues until they guess correctly. The goal for them is to guess the right number is the smallest number of questions possible.

Without much thought, you can conclude that the best choice for them of a first question is whether $x$ is smaller, larger or equal to 500,000. This way, even in the worst case, the space of potential options is reduced from 1,000,000 to 500,000. If your friend chose a smaller or a larger number, that would be of benefit to you but not to your friend, as you can cater your responses to whichever options leaves more candidate numbers while remaining consistent in your answers (e.g., if they ask whether the number is smaller, equal or larger than 900,000 as the 1st question, you will definitely answer "Smaller").

The conclusion is that one question/comparison helps us cut down on the number of options by at most a factor of 2 --- a question might cut down on the number of options by a smaller factor or none at all if it is not designed well, but *the most* that it can help us is reduce options by a factor of 2. This means that to go from the search space of $N$ to 1, we need to ask at least $\Omega(\log_2 N)$ questions, so with $N = 1{,}000{,}000$, our game is called 20 Questions Game. Now let's translate this analogy into external memory.

While in RAM we were counting the questions, i.e., the number of comparisons the algorithm has to make to solve the problem, in the external-memory model, we are counting I/Os.

Consequently, we need to compute what is the maximum benefit, i.e., how much information we learn from one memory block input.

Because a memory transfer contains at most $B$ elements, inputting one block is like changing the game to let our friend ask a bit more complicated question involving $B$ values. An example of such questions with $B = 4$, could be: Where would you place $x$ between the following four numbers: 23, 31, 56, 88? If $x$ equals one of the numbers, the game stops, otherwise you would have 5 options for your answer: $x < 23$, $23 < x < 31$, $31 < x < 56$, $56 < x < 88$, and $x > 88$) and game continues until one of the offered numbers equals $x$. In this case, we can also suggest what would be the optimal first question for our friend if say $B = 4$? It would be 4 evenly spaced out numbers, so that whichever of the 5 options we choose, the space in between is equal. This prevents us from prolonging the game longer than it has to last.

In theoretical computer science, this proof technique is called an *adversary argument*. In our game, we are the adversary, because provided with some $B$ elements, we will always choose to place $x$ in the sub-space that allows the game to go on for the longest. This is how we test the worst case of an algorithm. The only way to decidedly win against the adversary is for the algorithm to make all sub-spaces of equal size. When we deploy our algorithms into the real-world, we do not have real-life adversaries, rather, the adversary metaphor is there to help us realize the asymptotic complexity of a problem.

So, the best that could happen is if $B$ elements in the block help reduce the number of options by a factor of $B + 1$. Again, note that our friend can make up a bad block, which would enable us, the adversary, to cut down on space by a factor smaller than $B + 1$ (see Figure 10.9 to see how one can make a good/bad block.)

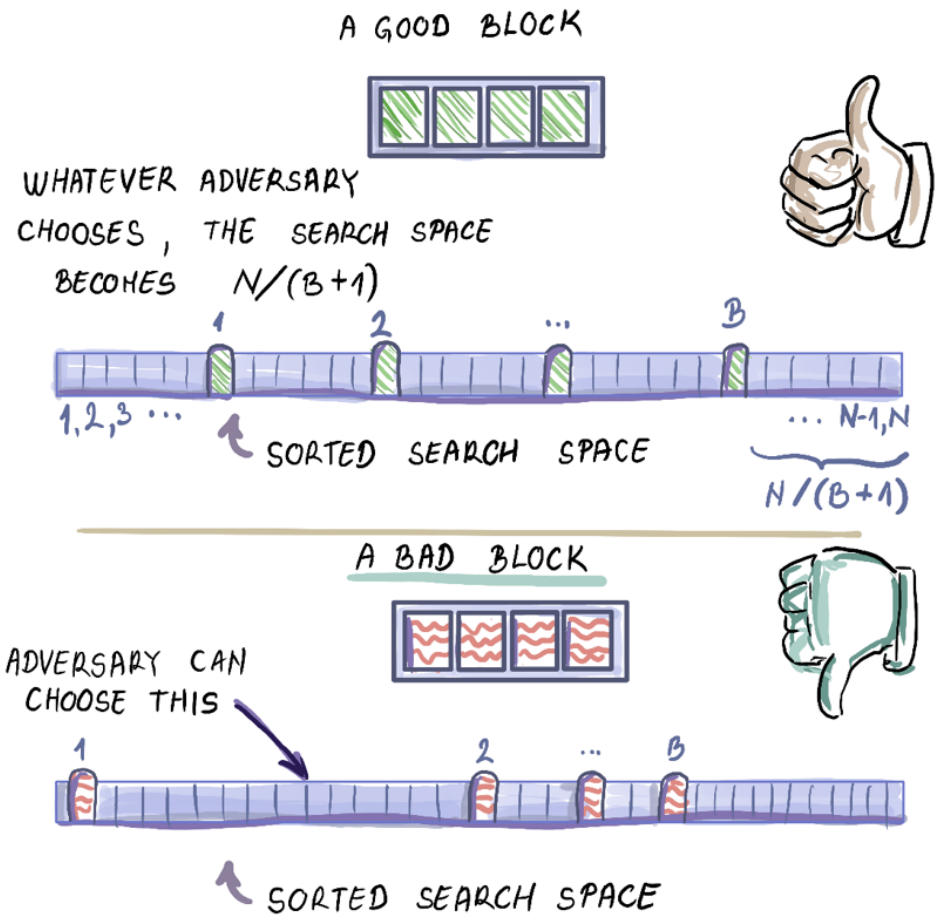



Figure 10.9: The lower bound calculation assumes good blocks.

Because each I/O helps us reduce the total number of options by at most $B+1$, that means we need $\Omega(\log_{B+1} N) = \Omega(\log_B N)$ I/Os to perform search in external memory, and B-tree lookup meets this lower bound.

### 10.4.1 Why B-tree inserts/deletes are not optimal in external memory?

Now that we know that B-trees are optimal with respect to queries, let's look at modify operations, such as insert and delete that require the same asymptotic number of memory transfers as the lookup.

Inserts and deletes, however, are essentially different from lookups, because an insert operation does not require an immediate proof that a new element has been stored into a

leaf. Similarly, a delete operation does not need an immediate confirmation that an element has been physically removed from the tree. The only confirmation comes as a result of a later lookup, when it should result in a 'Yes' on an inserted element and 'No' on a deleted element. Lookup is the only operation that requires an immediate feedback, and as such, it can not be delayed. Inserts and deletes, on the other hand, can be delayed and buffered. This way, a data structure can process these operations more efficiently in batches. We will see two such data structures in the rest of the chapter: $B^\varepsilon$-trees and LSM-trees.

## 10.5 $B^\varepsilon$-trees

$B^\varepsilon$-tree was devised by Brodal and Fagerberg[14],[15] as a data structure that embodies the tradeoff between the speed of inserts and lookups in external memory. The tradeoff is reflected in the range of values of parameter $\varepsilon = [0,1]$ that can be tuned, and when $\varepsilon = 0$, the data structure is fully optimized for inserts/deletes; when $\varepsilon = 1$, it is fully optimized for lookups (it is a B-tree.)

However, when we talk about $B^\varepsilon$-trees in this chapter, we will commonly refer to the "middle-ground" data structure that occurs at $\varepsilon = 1/2$. This value of $\varepsilon$ is interesting because at that point of the spectrum, we get a data structure with lookups that are only constant-factor worse than B-trees, and inserts that are asymptotically better than B-trees. This means that for write-optimized workloads, $B^\varepsilon$-tree is a significantly better fit than a B-tree, while still maintaining asymptotically optimal lookups.

### 10.5.1 $B^\varepsilon$-tree: How it works

The key design feature of $B^\varepsilon$-trees is that aside from keys, each internal node features a buffer. The purpose of buffers is to temporarily store inserts and deletes that act as messages on their way to the designated leaf. Namely, a delete operation in a $B^\varepsilon$-tree does not work the way it works in a B-tree, by directly going to the location of the element and physically removing it. Instead, a tombstone message "Delete $x$" is initially inserted into the buffer of the root node, and it gradually moves down the buffers along the root-to-leaf path to the leaf that stores $x$. Once the tombstone message reaches the leaf containing $x$, $x$ is physically removed from the tree along with the tombstone message. The analogous process exists with inserts.

Like in $B^+$-tree, in a $B^\varepsilon$-tree all items reside in leaves, so all inserts and deletes eventually affect the leaves, and keys in internal nodes are there only as pivots to direct the search. Insert/delete messages wait in the buffer until enough other messages have been collected to be flushed together to one of the children in just 1 I/O. This is in contrast to B-trees, where a single insert/delete triggers uses 1 I/O to descend to the next level of the tree, and consequently, a number of I/Os to complete the operation. By delaying operations in a $B^\varepsilon$-

[14] G. S. Brodal and R. Fagerberg, "Lower bounds for external memory dictionaries," in Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, Baltimore, Maryland, 2003.
[15] M. A. Bender, M. Farach-Colton, W. Jannen, R. Johnson, B. C. Kuszmaul, D. E. Porter, J. Yuan and Y. Zhan, "An Introduction to Be-trees and Write-Optimization," ;login: , vol. 40, no. 5, 2015.

tree, we can make them run faster in the amortized sense. Later we will see how keeping around all these messages affects our lookup algorithm.

Internal structure of a node in the $B^\varepsilon$–tree is following: each node contains $B^\varepsilon$ keys and the remaining $B - B^\varepsilon$ space is used for a buffer (see Figure 10.10 for a node where $B = 16$, and $\varepsilon = 1/2$.) For our common setup of $\varepsilon = 1/2$, we have $\sqrt{B}$ keys, and $B - \sqrt{B}$ buffer space. The buffer, therefore, occupies the most of the node space. Also, note that the depth of $B^\varepsilon$-tree is dictated by the node structure, where $\sqrt{B}$ keys per node gives us the tree depth of $\log_{\sqrt{B}} N = 2\log_B N$. Even though the number of keys is significantly smaller, the depth of the tree is only twice as that of a B-tree. This will slightly affect the performance of lookups as we have sacrificed node space to accommodate buffers, but the asymptotic lookup cost will remain equal to that in a B-tree.

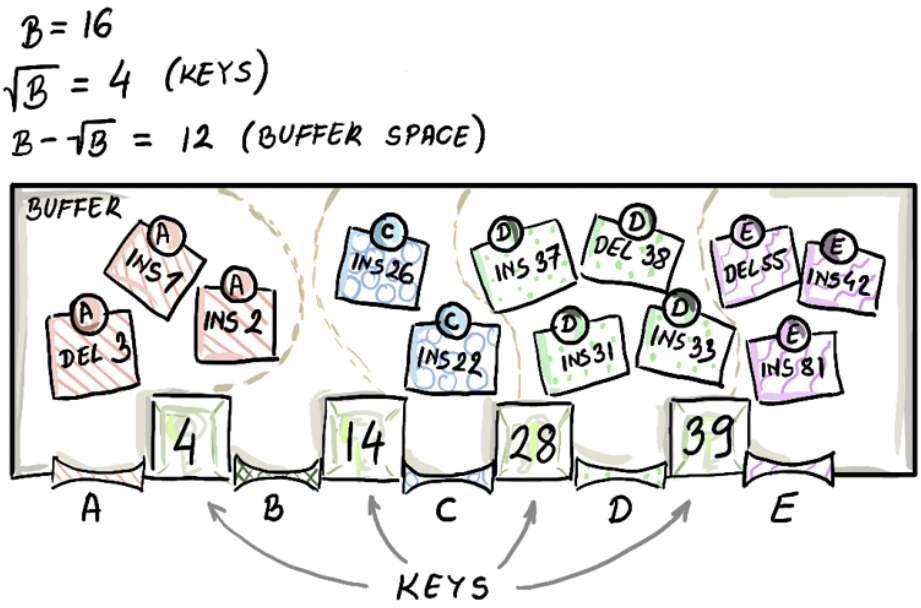



**Figure 10.10: A node in a $B^\varepsilon$-tree has keys and buffers. Currently, the buffer is full and can not accommodate any more updates.**

### 10.5.2 Buffering mechanics

One may think of a buffer as a single area in the node where messages accumulate, and once buffer becomes overly full, we flush it. That is, we only flush the elements destined for the child that has most pending updates. For this reason, perhaps a cleaner way to think of a buffer visually is splitting it into $B^\varepsilon + 1$ different sub-buffers, where each sub-buffer holds the messages destined for one particular child based on the keys' values (such as in Figure 10.10.) We do not explicitly partition the space between sub-buffers, and different sub-

buffers can share each other's space so that the flush is triggered only after the whole buffer has been filled. However, once the buffer is full, only the fullest sub-buffer gets flushed. Buffer is usually implemented as a balanced binary search tree, where we can quickly add and traverse the items and keep them in the sorted order. If implementing sub-buffers as separate binary search trees, we should only worry about their total size not to overshoot the buffer capacity, not the individual sizes of sub-buffers.

Figure 10.11 is showing the moment when a buffer overflows after a new update ("Del 8") and it gets flushed. Messages from the fullest sub-buffer get flushed and distributed to appropriate sub-buffers at the child. Other messages from the buffer stay in the buffer (for instance, "Del 8", which triggered the flush does not get flushed.)

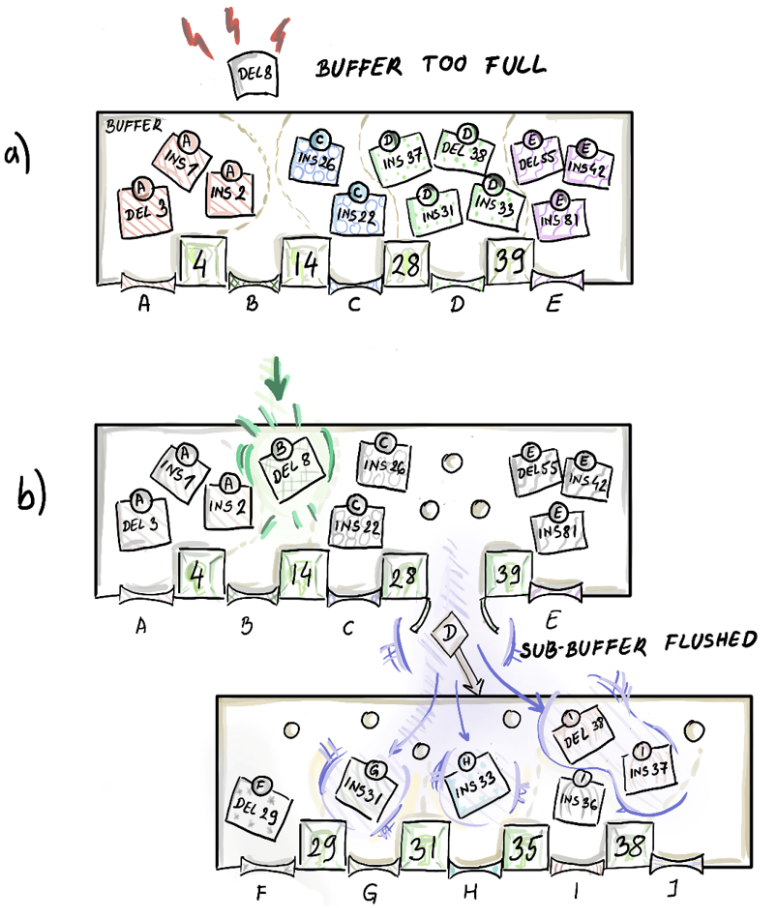



**Figure 10.11: When a buffer becomes too full, we flush the fullest sub-buffer to the appropriate node**

In Figure 10.11, the child node already had some of its earlier updates waiting in the buffer ('Del 29' and 'Ins 36'), however, together with the incoming messages, it does not overshoot the buffer capacity, so the process stops here. One important detail not shown is that each update message has a timestamp associated with it. Timestamp helps us reconstruct the history, and thus perform the lookup algorithm correctly.

### 10.5.3 Inserts and deletes

Now that we have the buffering mechanism under our belt, let's see the whole insert/delete process through. An insert/delete message is initially placed in the buffer at the root of the tree. If the message does not trigger any buffer overflows, we are done. Otherwise, if the message triggered the root buffer to flush, we flush it, and we perform any cascading flushes down the tree potentially going all the way down to the leaf level with appropriate inserts/deletes at the leaf.

If a leaf level is reached during flushes, we perform an insert/delete of messages that have arrived to the leaf level by physically inserting/adding an element like we do in a $B^+$-tree, and by eliminating those insert/delete messages from their buffer. Nodes in a $B^\varepsilon$-tree have the same sort of "order $d$" property like B-trees. When a leaf overflows its capacity, it splits in the same way in which it is done in $B^+$-trees. When it becomes too empty, it gets merged in the same way as in $B^+$-trees. So a typical insert/delete that progresses down buffers and eventually reaches the leaf might then trigger a split/merge operation at the leaf, that might in turn, trigger new splits/merges up the tree. This whole process works just like we described it for a B-tree, except that now, we split/merge not only the keys, but also re-distribute the messages in buffers.

### 10.5.4 Lookups

The searches in the $B^\varepsilon$-tree operate similarly to those in a B-tree, in that they follow the root-to-leaf path to the leaf that might contain the queried element. However, a $B^\varepsilon$-tree lookup also has to be mindful of the insert/delete messages it encounters along its path, as they affect the final lookup result.

For instance, let's say we are looking for an element 10 that was inserted in past, however, recently a delete operation was issued for it. If no other operations were issued in regards to 10 since, our lookup should report the element not present. However, the element might still exist in the leaf of the tree, as the tombstone message may not have reached it yet.

For this reason, a lookup operation has to collect all messages (with their timestamps) that relate to the queried element on the root-to-leaf path to the element. Then when it determines whether the element is present in the leaf or not, then it applies any potential insert/delete messages in the correct chronological order. In Figure 10.12, a lookup of an element 7 collects messages on its root-to-leaf path and upon reaching the leaf level and applying all the messages, it concludes that 7 is present.

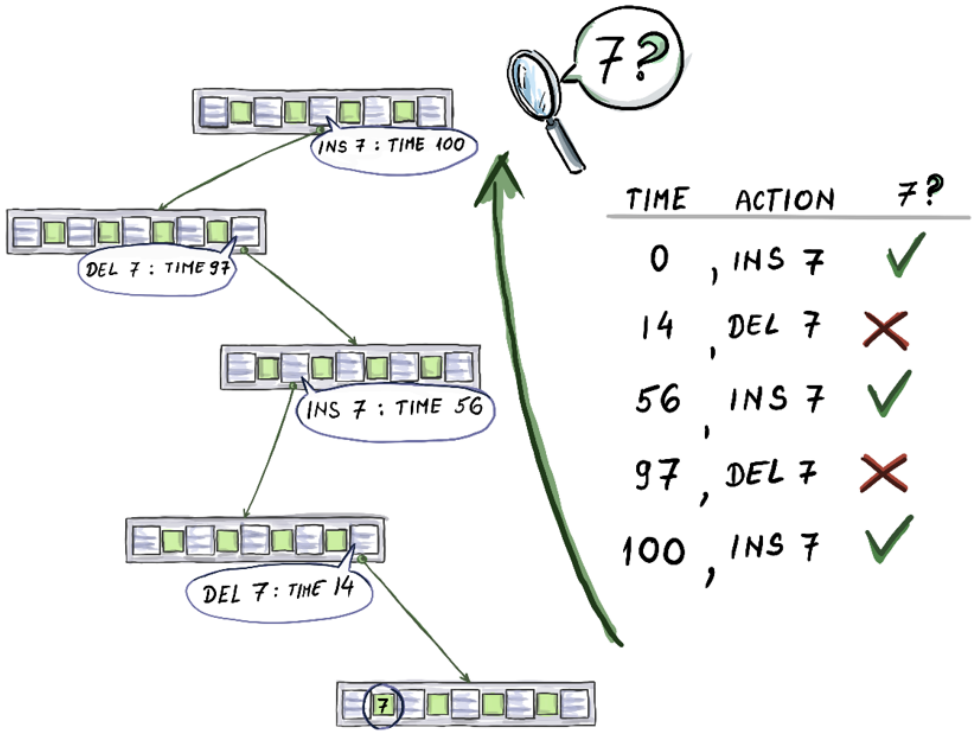



Figure 10.12: Insert and delete messages on the root-to-leaf path to item 7.

Keep in mind that a lookup never triggers flushing any buffers. It only internally collects relevant messages so that it can correctly answer the query. All the work in relation to flushing buffers is left to the insert/delete operations.

### 10.5.5 Cost Analysis

In this section, we analyze the cost of lookup, insert and delete operations in the $B^\varepsilon$-tree. We focus on the analysis for the middle-ground data structure that is of interest to us ($\varepsilon = 1/2$) even though it is easy to generalize for any value of $\varepsilon$.

$B^\varepsilon$-tree has $O(\log_B N)$ levels, so the lookup has to read in $O(\log_B N)$ nodes on its root-to-leaf path. In that sense, a lookup costs asymptotically the same as that of a lookup in a B-tree. More precisely, it is twice as slow because the $B^\varepsilon$-tree is twice as deep.

Inserts and deletes can be analyzed together as they work similarly. First we need to analyze how much it costs for one message to descend from one level of the tree to the next. This depends on how many elements travel together in one I/O when the buffer is flushed. When the buffer becomes full, the fullest sub-buffer is at least as full as all other sub-buffers, hence

it contains at least $(B-\sqrt{B})/(\sqrt{B}+1) \sim \sqrt{B}$ messages. This means that in 1 I/O, we transport roughly $\sqrt{B}$ updates to the next level of the tree, therefore each update costs $O(1/\sqrt{B})$ per level. The tree has $O(\log_B N)$ levels, so one insert/delete overall costs $O((\log_B N)/\sqrt{B})$ I/Os, a factor of $\sqrt{B}$ cheaper than in a B-tree! An example of this is shown in Figure 10.11, where $B = 16$, but we flushed 4 elements (fullest sub-buffer has 4 items), so per item, we used ¼ of an I/O. If we consider that $B$ is often expressed in thousands or even millions, being a factor $\sqrt{B}$ cheaper can represent a significant reduction in cost.

Other than the buffering cost, there is also the classical cost of physical splits and merges of nodes that are like those in a B-tree. But this time around, we need to analyze this cost more carefully, considering that we do not want to overshoot $O((\log_B N)/\sqrt{B})$ insert/delete cost. In other words, with a $B$-tree, we could be more loose with the analysis and assume that in the worst case, 1 split or merge happens per level, and that cost would still be covered by an already existent insert/lookup cost of just traversing down the tree. With the $B^\varepsilon$-tree, we need to be more frugal. Luckily, the number of expected splits and merges does work in our favor.

Consider the worst possible workload of all inserts headed towards one leaf --- this workload maximizes the number of node splits. Starting from the tree of order $d = \theta(\sqrt{B})$ whose nodes are minimally filled, every $\theta(\sqrt{B})$ inserts, we need to make the a node split. After $\theta(\sqrt{B})$ such splits --- which also constitute inserts into a higher level, we would need to make a split on one level above. That is, only after $\theta((\sqrt{B})^2)$ insert/deletes, we would affect the level above the leaf level. However, that cost is already "covered" by the cost incurring the splits on the leaf level much more often. And so on, one could continue to make an argument for higher levels, but the bottom line is that the cost incurred by splitting and merging is negligible and it is constant I/O cost amortized. This means that the cost of splits/merges in $B^\varepsilon$-tree is dominated by the cost of flushing and transporting messages down the tree.

### 10.5.6 $B^\varepsilon$-tree: The spectrum of data structures

As mentioned before, depending on how $\varepsilon$ is chosen, we might get either a better lookup or a better insert performance than in our common setup when $\varepsilon = 1/2$. Figure 10.13 shows three points on the spectrum, where (a) shows a B-tree (all keys, no buffers), (b) $B^\varepsilon$-tree at $\varepsilon = 1/2$ (some keys and majority buffer space), and (c) buffered repository tree (one key and all buffer space.)

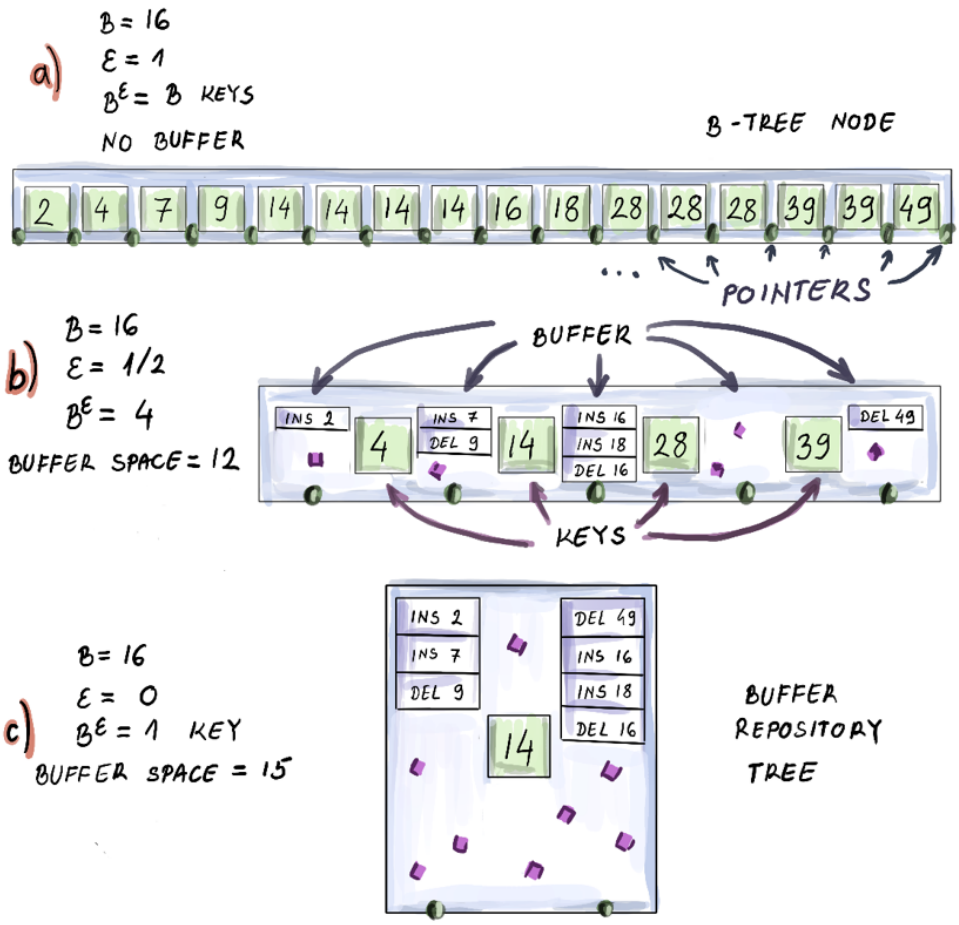



**Figure 10.13: The spectrum of $B^\varepsilon$-tree data structures, from most read optimized to most write optimized.**

Buffer repository tree is an interesting data structure, which allows us to optimize insert/delete performance even more than $B^\varepsilon$-tree with $\varepsilon = 1/2$. Because the buffer is large and messages at each node can be directed in only 2 different directions, then $\theta(B)$ items can carpool together to the next level, bringing the insert performance down to $O(1/B)$ per level, and $O((\log_2 N)/B)$ I/Os in total (the buffer repository tree has the $O(\log_2 N)$ levels just like binary search tree, which makes it infeasible for the lookup performance.

### 10.5.7 Use case: $B^\varepsilon$-trees in TokuDB

$B^\varepsilon$-trees have been implemented by the Percona TokuDB storage engine for Percona Server for MySQL[16]. Similarly, there has been implementations of file systems that run $B^\varepsilon$-trees underneath, such as BetrsFS[17]. Because $B^\varepsilon$-trees help inserts get better, this in turn can help make index maintenance easier and faster, thus allowing multiple indexes to co-exist without inserts becoming too slow. So somewhat ironically, the story is that we make lookups worse to help the inserts who, in turn, again help lookups.

A typical use case where $B^\varepsilon$-trees might prove useful is in highly dynamic applications where both inserts and searches need to be fast. Consider the following highly-performant application: your company hosts web requests for the largest publisher of online magazines. Users constantly load new content and react to that content (e.g., by adding new comments) and at the same time a large amount new content and articles is being posted, modified, and simultaneously queried.

The challenge of this type of application is to post new content that is up-to-date, but not at the cost of slowing down customer's reading experience. Similar use case scenario arises with social networks, where new content should be ingested at a high rate, however, the content needs to be quickly loaded to users as well.

### 10.5.8 Make haste slowly, the I/O way

One of the major differences between the B-trees and $B^\varepsilon$-trees is that B-trees perform in-place updates, that is, when, for example, a modify/insert/delete operation arrives, the change is immediately incorporated right where the relevant element resides. $B^\varepsilon$-trees, on the other hand, perform out-of-place updates, where a modify/insert/delete message is incorporated into the data structure in a different place from the one where the element in question resides. There is no rush to immediately find the element and apply the needed change to it. Note that out-of-place updates increase the amount of space required by the data structure, in that the number of items in a data structure is measured not by the number of distinct elements, but by the number of updates to it. We store not just the items, but messages in relation to those items.

However, note that the out-of-place feature is exactly what helps the modify/insert/delete operations to be faster than in the B-tree. In order to perform an update, we do not need to search for the exact location of the element immediately and burn a lot of I/Os in the process. Updates take their time to travel down the tree when it's cheapest to descend down the tree. Inserts/deletes wait longer to be applied but exactly for that reason are faster --- this is because we do not measure the efficiency of operations in the time taken to apply the operation but in the number of I/Os required for the change to be applied. In the next

[16] P. LLC, "Percona," Percona LLC, [Online]. Available: https://www.percona.com/software/mysql-database/percona-tokudb. [Accessed 08 August 2021].

[17] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. Bender, M. Farach-Colton, R. Johnson, B. Kuszmaul and D. E. Porter, "BetrFS: A Right-Optimized Write-Optimized File System," in Proceedings of the 13th USENIX Conference on File and Storage Technologies, Santa Clara, California, 2015.

section, we will see a data structure that takes the notion of fast writes (and out-of-place writes) even further than $B^\varepsilon$-tree.

## 10.6 Log-Structured Merge-trees (LSM-trees)

To understand how LSM-trees came about, let's start by developing a simple write-optimized data structure right here ourselves. What would be an optimal way to implement an external-memory index with blazing fast out-of-place inserts/deletes without regard to the speed of lookups?

Just logging insert/delete messages in one sequential log comes to mind. One way to envision such a data structure is to have an in-memory buffer where messages carrying inserts, deletes or modifications to records are accumulated. Once the buffer is full, we flush it in the sequential fashion to a location on disk. Then we again fill the in-memory buffer with writes (when we say writes, we mean inserts, deletes and modify operations), and flush the memory contents by appending the new stuff to the end of the log.

This simple system guarantees an ideal insert/delete performance of $1/B$ per item --- the amortized cost of just writing the items to disk, so it is not hard to see why we can not do better than that. Of course, queries would be terrible as we would need to scan the whole file on disk to answer a query ($N/B$ I/Os).

Now let's try to slightly modify this idea without hurting the write performance. Let's say every time the memory buffer gets filled up, we internally sort all the items in the memory buffer --- in order to do this, a buffer can be some sort of a balanced binary tree --- and flush the sorted range to a separate file or a table on disk. The next time the buffer gets filled up, we again sort all in-memory data and flush it to another table next to the first table, and so on. Now we have a somewhat more organized system, with many separate tables of data where each table is internally ordered. The inserts/deletes still run optimally in $O(1/B)$ amortized per element, as buffer sorting takes place in internal memory and does not need any additional I/Os. Queries are not astronomically better: now we have to examine each table to locate updates related to an item of interest. If the table size is similar to the main memory size $M$, and the total number of updates is $N$, we will have a total of $N/M$ tables. Because each table is sorted, we can use binary search, for example, to guide the search within individual tables, which helps us avoid full linear scan of a table. This gives us $O\left(\frac{N}{M} * \log_2\left(\frac{M}{B}\right)\right)$ I/Os cost per query.

Let's slightly improve our simple design: considering that tables are immutable (we would not be modifying them after flushing to disk), then why not build a $B^+$-tree index on top of each table, and improve the query performance to $O\left(\frac{N}{M} * \log_B\left(\frac{M}{B}\right)\right)$ I/Os. Here is our resulting data structure in Figure 10.14:

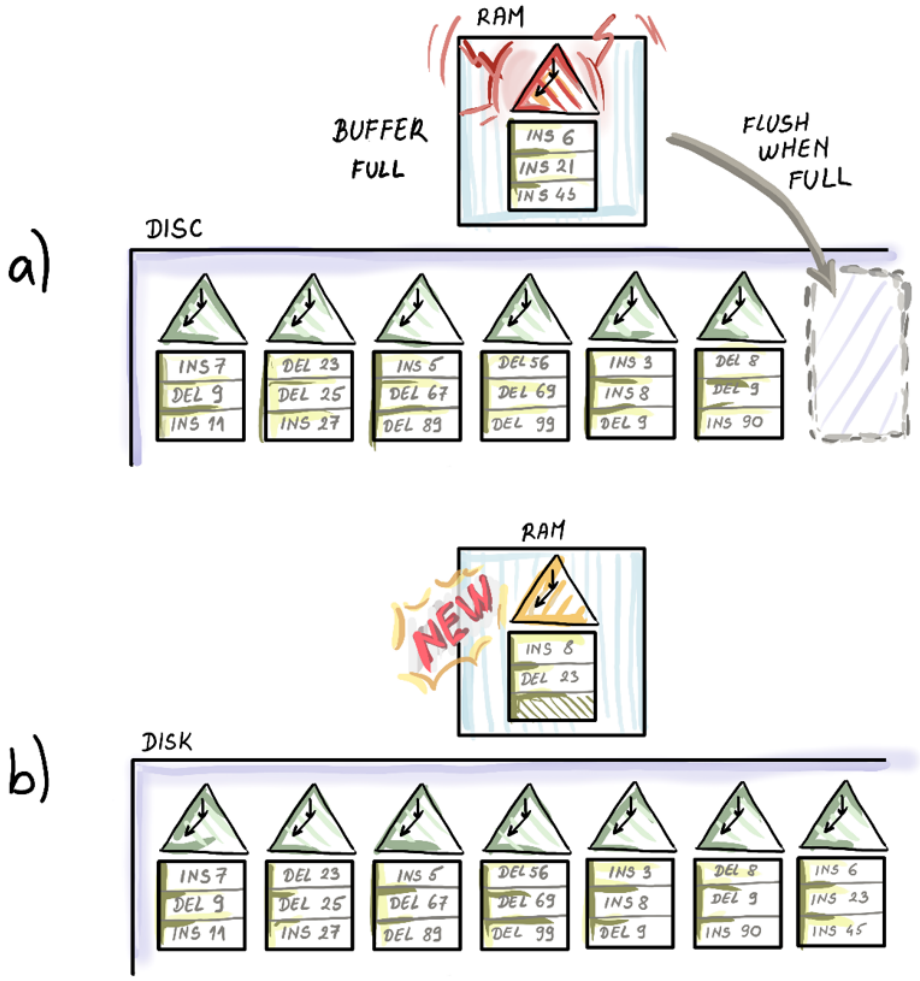



Figure 10.14: Our simple write-optimized data structure that is not the LMS-tree.

Over time, a growing number of tables will exacerbate an already poor lookup performance. Maintaining a number of Bloom filters in RAM (a use case covered in Chapter 3), one per table, could help eliminate the disk lookup on tables that do not contain updates for the queried element. However, if you recall, Bloom filters also grow linearly with the total size of data, so before we know, we would be inundating the RAM with a ton of mini Bloom filters. Bloom filters buy us time, but not for long, if we are dealing with ridiculously high insert rates.

The log-structured-merge tree (LSM-tree) is a data structure devised in 1996 that embeds the idea of our simplified write-optimized structure and adds to it a mechanism that limits the number of tables on disk by occasionally merging and compacting them. LSM-tree has been successfully implemented in a number of write-optimized databases such as LevelDB used by Google, RocksDB used by Facebook and others. Let's see how LSM-trees work.

### 10.6.1 LSM-tree: How it works

There is a number of variants of the basic design of LSM-tree, and many different implementations[18]. Originally, LSM-tree is made out of $k$ components $C_0, C_1, \ldots, C_{k-1}$ where $C_0$ is in internal memory and all other components are on disk. However, there are a couple of important differences from our simplified data structure from before: in an LSM-tree, we assume that the size of $C_0$ is on the order of memory size $M$, and $C_1$ is by a factor $f$ (usually $f \geq 2$) larger than component $C_0$. In fact, the ratio $f$ is maintained between the sizes of any two consecutive components, so the component sizes, in the increasing order, are $M$, $fM$, $f^2M$, etc.

On-disk components were originally envisioned as $B^+$-trees, but as we will see, in modern implementations, different data structures are used such as skip lists, or simple sorted key-value tables and files. The exponential increase in sizes between components guarantees that we will have a manageable number of components to later query. The largest component should be able to store all $N$ elements, so the total number of components is $k = O(\log_f(N/M))$, if the smallest component is of size $\theta(M)$.

Each component sits at its own level, and each level has a limit on its maximum capacity. When the upper capacity threshold is violated at one level, the corresponding component $C_i$ gets merged into the component $C_{i+1}$. This in turn might fill up the component $C_{i+1}$, and cause cascading merges with levels below. In the original LSM-tree design, this is achieved by actually merging the range of keys from the smaller component into the larger component. Modern LSM-tree implementations favor the approach where once written components (also called runs) are immutable. So even if the final effect of merging level $C_i$ into $C_{i+1}$ is the same as in the original LSM-tree merging approach, modern merging policy between levels never mutates the structures once written. Instead, it creates a new merged component and garbage-collects the old ones. Obviating the details of how things are physically merged on disk, Figure 10.15 shows an example of an LSM-tree and the merging policy we just described, commonly known as *leveling merge policy.*

In the example in Figure 10.15, we set $M = 4$ and $f = 2$. On the left side of the figure, we capture a snapshot of the data structure at some point of the workload processing. The component $C_1$ is full and has to be merged into $C_2$. To merge $C_1$ into $C_2$, we sequentially scan the range of items in $C_1$ and $C_2$, and merge them in the fashion in which they would be merged during mergesort --- we can do this because items inside individual components are sorted. Before the merge, $C_2$ had 8 items and now it has 16 (the right side of the figure). The component $C_2$ will also be merged into $C_3$ as it has reached the maximum capacity.

[18] C. Luo and M. J. Carey, "LSM-based storage techniques: a survey," VLDB Journal , vol. 29, p. 393–418, 2020.

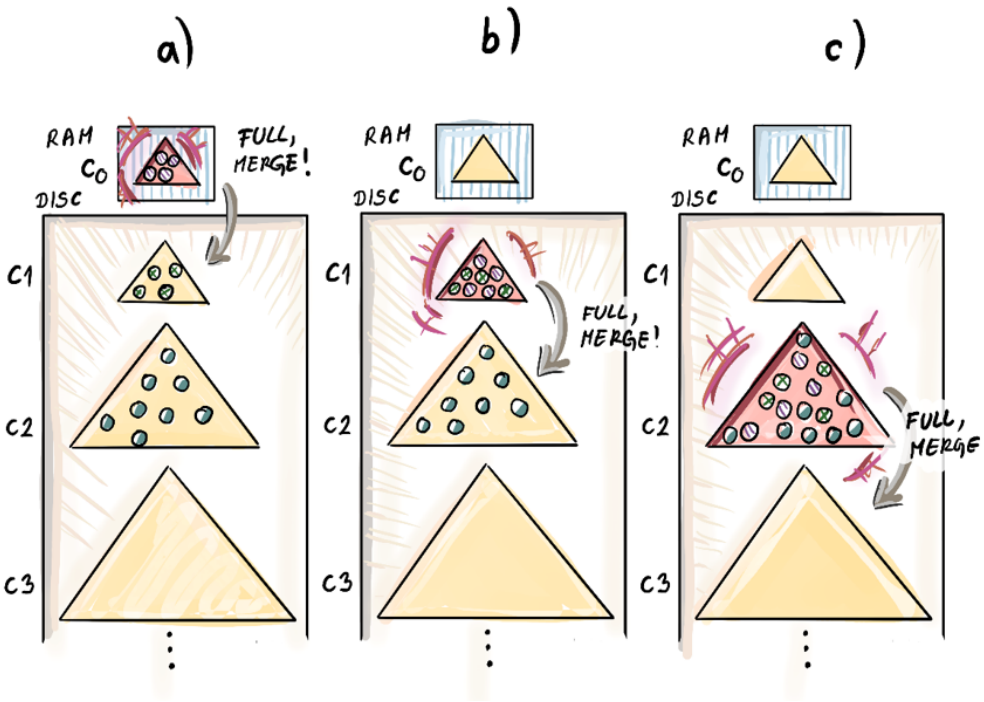



Figure 10.15: Merging smaller component into a larger one in leveling merge policy of an LSM-tree. The example shows a B+-tree-like unit components at each level, but modern LSM-tree implementations use various data structures or even simple tables and files instead of B+-trees.

Note that when we merge in this way, for one element to descend to the next level, we write that piece of data many times --- once when it is actually being merged into the lower level, and then later when other items are being merged into its level. This process gets repeated for each level, and this so-called *write amplification* effect is more pronounced with larger growth factors, as we need to merge the smaller component $f$ times into the larger one in order to fill it up. Write amplification is a term used to measure how much data is written inside data structure per unit item inserted. It is safe to say that with leveling merge policy, write amplification is fairly high.

*Tiering merge policy* is another popular mechanism for the component compaction in modern LSM-tree implementations. In this policy, tiers are equivalent to levels, except that each tier contains $f$ components of the same size. Once $f$ components at tier $i$ fill up, they all get merged into one new component in the tier $i+1$. This way, in order to descend to a new tier, each item gets written only once. See an example of tiering merge policy in Figure 10.16, where we use sorted runs instead of $B^+$-trees as components and $f = 2$. In this example, two tables at tier 1 become full and get merged into one table at tier 2. This process of merging

is done in a fast sequential manner. It is not shown in figure, but now 2 full components at Tier 2 will get merged into one component at Tier 3.

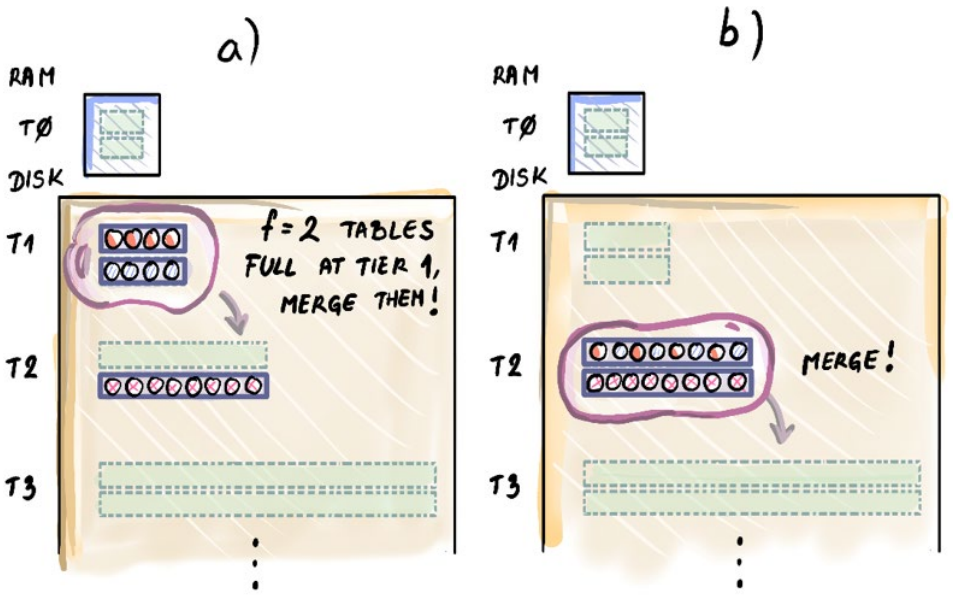



**Figure 10.16: Tiering merge policy in an LSM-tree with $f = 2$. When $f$ components at tier $i$ get filled up, they merge into one component at tier $i + 1$.**

### 10.6.2 LSM-tree cost analysis

Note that with both merging policies, we lose some of the initial $O(1/B)$-type write performance during compaction. In leveling merge policy, because each item is written about $O(f)$ times in order to descend one level, the total cost for one item to descend one level below is $O(f/B)$, and the total cost to descend to the bottom of the LSM-tree is the cost accumulated over all levels: $O((f/B)\log_f(N/M))$ I/Os. In tiering merge policy, we only need $O(1/B)$ per level, and accumulated over all levels, we need $O(\log_f(N/M))$ I/Os, a factor $f$ less than in leveling merge policy.

Queries, however, tell a different story. Without regards to the size of the component itself, we need about a constant number of I/Os per component to check the presence of an item. This can be achieved even if the component itself or a run is much larger than a block. An example are String Sorted Tables (SSTs) that contain sorted key-value pairs of data, and in addition to it, a small index of keys. So if we first want to find out where a potential item might lie in the table, we fetch the index of the individual table (component) that is small enough to fit in a block. By learning where the item resides, we then need just one more I/O to fetch the item. So overall, one component needs one I/O.

In leveling merging policy, because we have the number of components equal to the number of levels, we need $O(\log_f(N/M))$ I/Os for a lookup. In tiering merge policy, we can have by up to a factor of $f$ more components to check, shooting up the query cost in this merging policy. However, the biggest gains in the query performance come with the use of Bloom filters that help redirect the query to the right table, thus bringing the query performance to $O(1)$ in most cases. This is the optimization that can be applied to both merging policies and this time it works because the total number of components is logarithmic in the total size of the dataset, thus maintaining those Bloom filters in main memory is quite manageable. Range queries are not as lucky as they can not benefit so well from Bloom filters as point queries.

### 10.6.3 Use case: LSM-trees in Cassandra

LSM-trees have been implemented in a number of large-database engines, such as Cassandra, LevelDB, RocksDB etc. Specifically, Cassandra's tiering merge LSM-tree uses Bloom filters to avoid unnecessary disk seeks. Figure 10.17 is a throwback to our Chapter 3, where we hailed applications of Bloom filters in distributed storage contexts. The tables shown in the figure below show an LSM-tree whose components are tables. You can envision that with $f = 4$, tables SST 1-4 are tier 1, SST 5-8 are tier 2, etc.

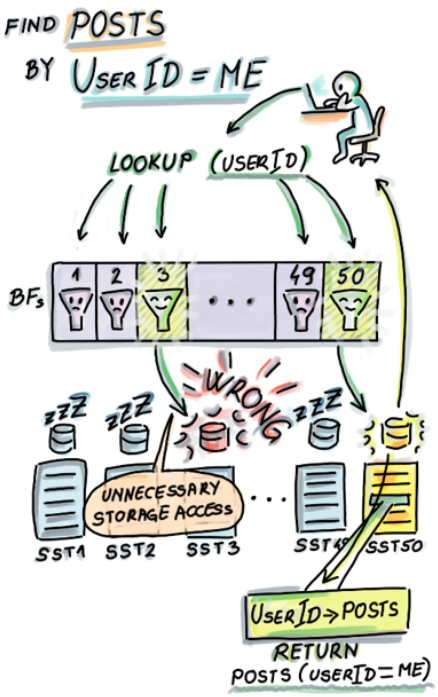



**Figure 10.17: LSM-trees use Bloom filters to eliminate unnecessary disk seeks to tables that do not contain queried items.**

A typical business case for an LSM-tree would be an application that needs blindingly fast write performance. An example of such an application is a backup product that takes data snapshots of data at regular intervals and stores petabytes of data, but rarely revisits history. Another use case would be a traffic-monitoring network application that witnesses hundreds of millions request per hour. The requests are stored in the database, but there is rarely a case of an explicit search for a particular request.

## 10.7 Summary

- Database index is a data structure built on top of a database table meant to speed up query performance for large tables. Indexes are built using data structures that can perform efficient searches in the external-memory setting.
- B-trees form the backbone of the most widespread storage engines such as MySQL. B-tree is an optimal data structure to perform lookups on disk. B-tree nodes are large and are usually related to the block size. All operations in a B-tree are logarithmic with the base of $B$. When nodes in B-trees become too full/too empty, nodes can be split/merged, and the tree grows upwards.
- B-trees are read-optimized, and there are other data structures that are better suited to write-heavy workloads. Inserts and deletes are operations that, unlike searches, can be delayed, or processed in batches together. Write-optimized data structures use this idea to delay and buffer insert/delete operations to achieve much faster inserts than B-trees.
- $B^\varepsilon$-tree is a write-optimized data structure whose inserts/deletes are asymptotically faster than B-trees, and lookups only by a constant slower than B-trees. $B^\varepsilon$-trees employ buffer at their nodes to temporarily store insert/delete messages so that they can be at a convenient moment processed in a batch. The value of a parameter $\varepsilon$ determines the extent to which the data structure prefers writes over reads.
- LSM-tree is a write-optimized data structure that consists of sorted runs that occasionally get merged in a fast sequential fashion. LSM-trees can achieve extremely fast updates at the cost of lookups that are slower both than those of a B-tree and $B^\varepsilon$-tree.

# 11

# *External-Memory Sorting*

**This chapter covers**

- **Understanding the importance of efficient sorting on disk**
- **Revising two most classical in-RAM sorting algorithms: mergesort and quicksort**
- **Learning how external-memory mergesort works**
- **Understanding how external-memory quicksort works**
- **Understanding the relationship between searching and sorting in internal vs external memory**

In the previous chapter, we learned about different ways to design indexes in databases. Indexes embody the very fundamental problem of searching in computer science. Another fundamental problem that crops up in databases --- and pretty much everywhere --- is sorting. Just think of how many times you used the function `sort()` in your code to order a set of data.

Aside from obvious applications of sorting, there is a large number of algorithms that use sorting as its subroutine. For instance, in Chapter 2, we discussed the problem of deduplication, i.e., eliminating duplicates, and talked about various efficient hashing solutions. Hashing gives a good average-case performance, however, if we are aiming at the best worst-case performance involving actual element comparisons (not matching via hashing), eliminating duplicates requires sorting of data. This does not mean that the optimal algorithm for deduplication must explicitly sort, but it needs to perform at least the amount of work required to sort --- so we might as well solve it by sorting and then scanning the array for duplicates. Another, just slightly different version of this problem is the element distinctness problem that takes an unordered array as an input, and asks to output 'Yes' if all elements are distinct in an array and 'No' otherwise. Element distinctness also requires

sorting in a similar way deduplication does, in that, the optimal algorithm for this problem will have the runtime at least as high as that of sorting ($\Omega(n \log_2 n)$).

In this chapter, we will first talk about different contexts where sorting comes up and the challenges that arise when sorting huge files with main memory of limited size. Then we will explore two well known sorting algorithms, mergesort and quicksort, more specifically , we will explore how to adapt them to external memory. We will do so gradually, so as to demonstrate algorithmic tricks that can be useful in other sorting-like problems. Lastly, we will show how to analyze lower bounds for sorting in internal and external memory. Using this tool, we will be able to ascertain that the external-memory mergesort is an optimal algorithm for sorting in external memory.

## 11.1 Sorting use cases

Sorting is common across applications in many domains. In the world of geometry, sorting points by coordinates is quite common and is needed for many fundamental routines, such as computation of the closest pair of points in a 2D plane, sweep line algorithms and others. Consider the following application of sorting in computational geometry and robotics.

### 11.1.1 Robot motion planning

Imagine you are designing a robot that needs to move around the kitchen table avoiding obstacles (let's say the robot needs to pick up the crumbs from the table). The robot has a map of objects and their 2D footprints in its neighborhood which should aid the robot in moving seamlessly around the table without crashing into objects on the table. Objects can be of various shapes and the actual footprints might be complicated shapes that can make the motion planning difficult, so to simplify computation, instead of actual footprints, the robot computes what we call the convex hull of the 2D footprint of each object. Basically, if we took a giftwrapping paper and started wrapping it around the 2D footprint, we would obtain a convex hull --- the smallest convex polygon that contains the footprint (see Figure 11.1 for clarification).

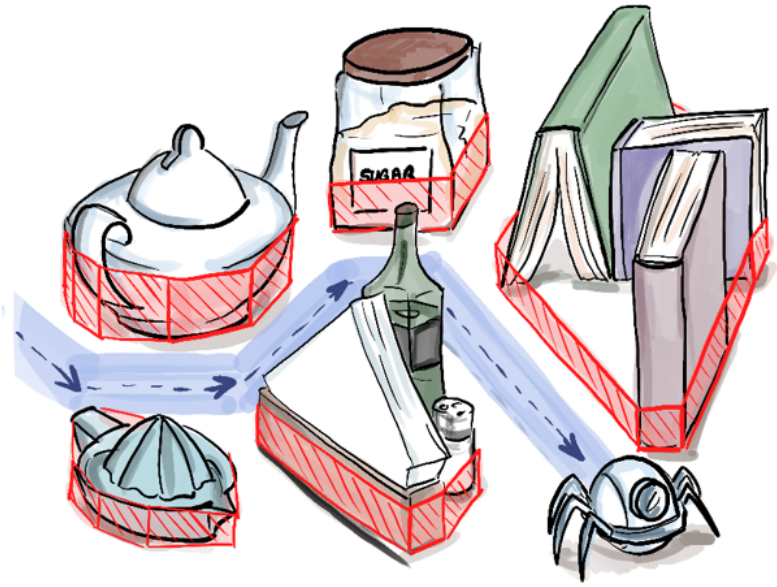



Figure 11.1: Robot motion planning algorithms often involve computing convex hulls of the nearby objects.

Many convex hull algorithms use sorting, where they sort the points along $x$ and $y$ coordinates. One way to visualize one popular convex hull algorithm is in fact to imagine wrapping a gift paper around the 2D footprint of the object. The process of figuring out which corner to wrap around next involves sorting the angles from the current corner to other corners of the footprint. For more details, check out Jarvis March, or Gift Wrap algorithm for convex hulls. (Many other algorithms for computing convex hulls, without the giftwrap in their name, also use sorting.)

Databases also use sorting extensively to create indexes, to perform group-by operations, sort query outputs, etc.[1] Aside from using sorting to implement basic database operations, large databases commonly need to order data according to some criteria that involves computations on a number of different columns. Consider an example of a following bioinformatics database as an application of sorting.

### 11.1.2 Cancer genomics

You have a large database of genomes (a complete genetic code of an individual) that you would like to order according to proclivity to a particular type of cancer. You are using your database to test a hypothesis from a recent study that the frequency of certain sequences X and Y within the genome play a role in cancer incidence and sequence X does do twice as much as sequence Y. To do so, we order genomes according to the evaluation score that

[1] G. Graefe, "Implementing Sorting in Database Systems," ACM Computing Surveys, vol. 38, pp. 1-37, 2006.

uses the number of occurrences of said sequences, and use the evaluation score as the input to the comparison function, as shown in example in Figure 11.2:

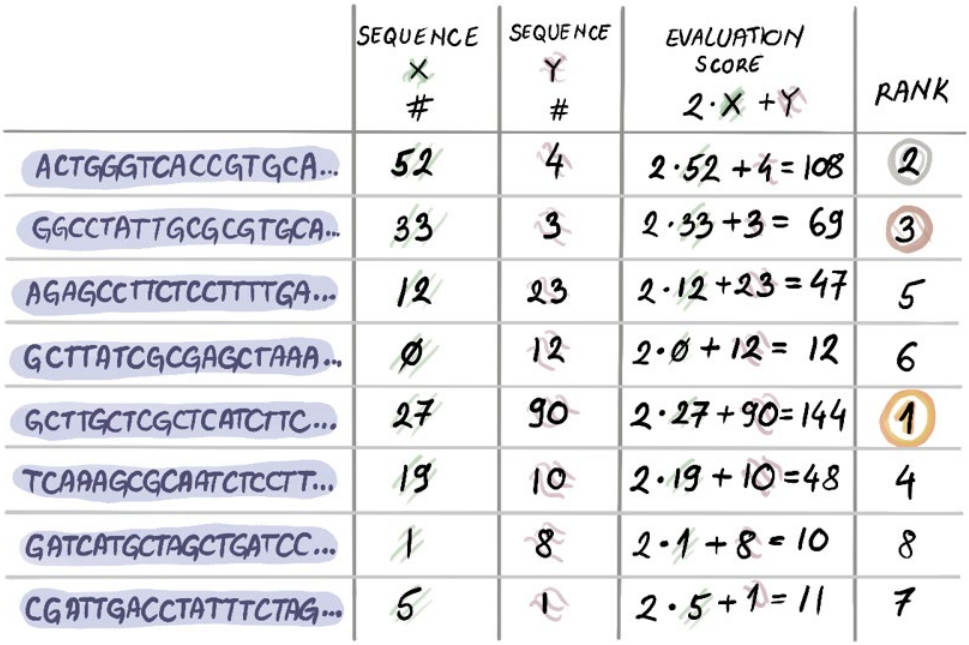



**Figure 11.2: In bioinformatics, genomes are often ordered according to various criteria. In this particular case, we are ordering genomes by the number of times sequences X and Y have occurred. The importance of sequence X's presence is valued as twice as that of sequence Y's presence.**

When sorting, it is common to provide a customized comparison function which we used to define the notions of "less than" and "equal than". This is especially helpful with non-primitive data types, where the ordering between items is something context-specific and more complex. Python's sort function, for example, allows one to pass in a customized comparison function.

Given particular ranges and types of data, size of a dataset, and many other parameters, a different sorting algorithm might apply. Research on sorting and a number of different implementations of sorting algorithms are quite extensive. A comprehensive review of sorting deserves a chapter or a book of its own, and with the exception of a couple of algorithms that we will review as we go along in this chapter, we will not discuss many intricacies of sorting.

What we will focus on is the aspect of sorting when data becomes too large to fit into RAM. When we have a large file to sort that sits on disk, and only a small chunk can fit into main

memory at one time, the main issue becomes how we define the high-level sorting procedure that will sort the whole file while being able to work with only a small portion of data simultaneously. Specifically, figuring out how to do this while minimizing the number of disk transfers will be the focus of this chapter.

## 11.2 Challenges of sorting in external memory: An example

Consider working for a hosting company that collects data on web requests for their clients. Say you want to order all requests in past month to determine the distribution of access times and find the requests that took the longest. Your company collects a lot of data and data is organized into one large table where each row represents one request, and all its associated information: IP address, browser, access time, etc. So, in total, you have a file of roughly 512GB that needs to be sorted, but you dispose of only 4GB RAM.

The first thing that comes to mind is that we can sort 4GB worth of data at one time. If we partition the original file into chunks of 4GB, and read each whole chunk into main memory, sort it and write it back, we get a partially sorted dataset.

This step of creating mini sorted lists is in fact a great starting point to apply mergesort algorithm, only in external memory. In future text, we will sometimes use the term 2-way mergesort to refer to the traditional mergesort algorithm as means of contrasting with the multi-way mergesort we will be developing for the external memory. Let's see how 2-way mergesort (or just, mergesort) works when blindly translated to external memory.

But first, a quick review: 2-way mergesort in RAM works by trivially partitioning the array into small sub-arrays top-down until of size 1, and does all the work by merging those arrays bottom-up, one pair at a time. Merging is effectively sorting. This recursive merging turns $n$ sorted lists of size 1 into $n/2$ sorted lists of size 2, $n/4$ lists of size 4, etc and then at last 1 list of size $n$. Mergesort runtime is described using the following recursive formula $T(n)$ that, when the recursion is unrolled, represents the number of comparisons required by mergesort:

$$T(n) = 2T(n/2) + O(n)$$

The $O(n)$ term represents the time required to merge at one level (for instance, $n/2$ lists of size 2 into $n/4$ lists of size 4.) The base case of recursion is $T(1) = 1$ because sorting a list of one element is trivial. Unwrapping this recursion using Master method, or a simple tree built by unraveling the recursion, we obtain that mergesort runtime is $O(n \log_2 n)$.

### 11.2.1 2-way mergesort in external memory

Before we start thinking about how to adapt 2-way mergesort to external memory, we first review the parameters we use for the analysis of algorithms in external memory. The value of $N$ represents the input size (number of records), $M$ represents the total size of main memory, and $B$ is the block size.

The benefit of external memory when it comes to sorting is that with one sweep over all data (N/B block transfers), we can get N/M sorted lists of size M so trivially partitioning into lists smaller than M does not make much sense. Naturally, this will translate in the base case of our algorithm. After creating $N/M$ sorted lists each of size $M$, the algorithm works in the analogous way to internal mergesort, where we merge pairs of lists --- see an example with sorting cards in Figure 11.3a.

When merging two sorted lists, we are often unable to hold both lists in their entirety in main memory simultaneously, but all we need in order to perform merging is to have one block of each of the two lists into main memory and picking out the smallest remaining element among the two blocks until one block is fully exhausted, then we read the next block from the list. The process is similar to merging k sorted lists from Chapter 9, a figure we repeat here (Figure 11.4), but in 2-way external mergesort, $k = 2$.

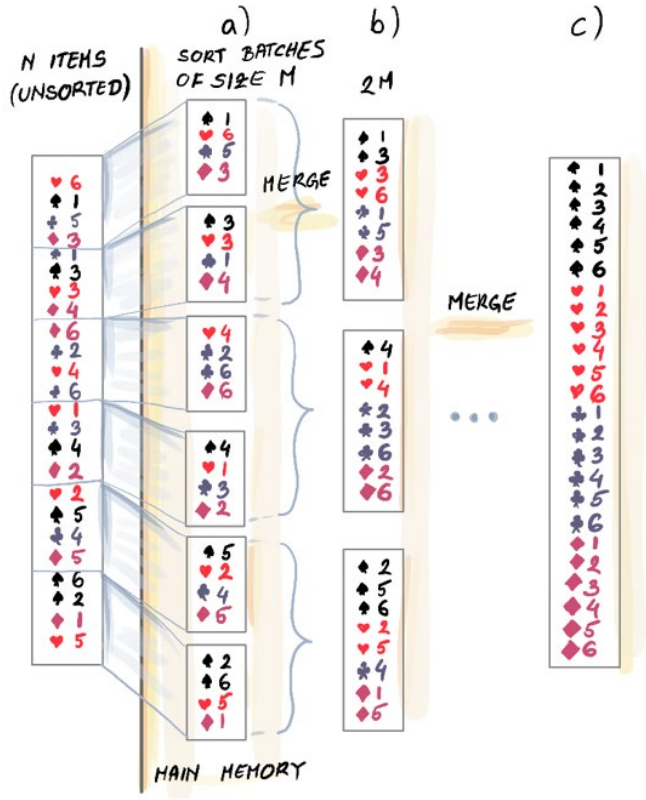



**Figure 11.3. 2-way mergesort adapted to external memory. In the first run, N elements are processed in N/M batches of size M, where each batch of size M is sorted. This is our "base case" for external-memory mergesort (internal memory mergesort ordinarily begins from lists of size 1). Then one by one, pairs of lists of size M are processed and merged into lists of size 2M, then 4M, and so on. Eventually we arrive at the final**

list of size $N$.

This means that the runtime of 2-way external mergesort is:

$$T_{2ext}(N) = 2T_{2ext}\left(\frac{N}{2}\right) + O\left(\frac{N}{B}\right)$$

and the base case $T_{2ext}(M) = O(M/B)$, the number of transfers required just to read the data. The total sorting cost dominates the linear cost of creating initial sorted lists of size $M$, so we do not include it in the formula. To understand what happens in 2-way external mergesort, it is important to understand that each read of all of data costs $N/B$ transfers. Each sweep over entire data that increases the list size by a factor of 2 (and cuts the number of lists by a factor of 2) needs $N/B$ I/Os. To get from $N/M$ lists of size $M$ to 1 list of size $N$ by doubling the list size every run, we need $O(\log_2 \frac{N}{M})$ such sweeps. This analysis (as well as unwrapping of the recursion above) gives us $O\left(\frac{N}{B}\log_2 \frac{N}{M}\right)$ I/Os.

**Exercise 11.1**

Analyze the number of block requests needed to sort the request data from an earlier example using 2-way mergesort in external memory. Some of the common block sizes are 8KB-64KB.

But we can do better than 2-way external mergesort, and to do better, let's go back to merging $K$ sorted lists simultaneously that we introduced in Chapter 9. Figure 11.4 is instructive for how to merge sorted lists whose total size can not fit into RAM.

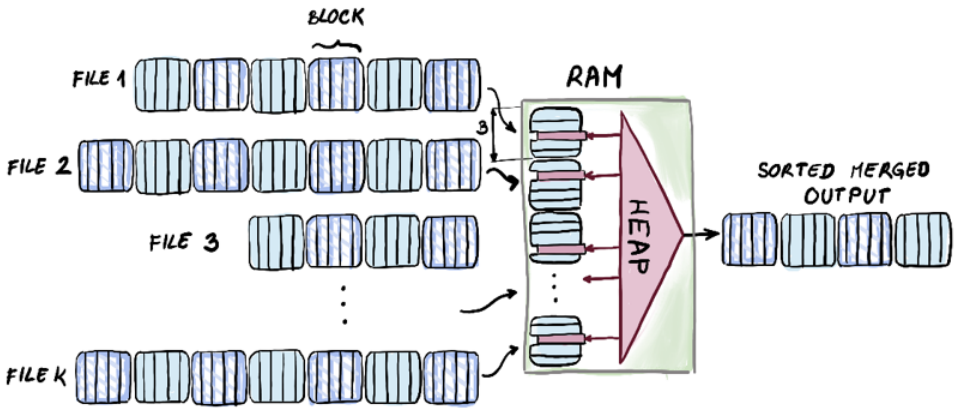



Figure 11.4: Merging $k$ sorted lists in external memory. Each list has one representative buffer block residing in memory at all times. The minimum of each block is initially inserted into a heap, from where minima are repeatedly extracted. Every time an element is extracted from the heap, the next element in line from the same block where the minimum came from gets inserted into the heap. When we run out of elements of one

particular block, we bring in the next block of the same list.

According to this figure, we can have up to $O(\frac{M}{B})$ lists being merged at one time, as every list needs just one block to represent it in RAM. Merging many lists at one time produces significant gains, as we can turn $M/B$ lists of size $x$ into 1 --- and to do this, we would use the same number of memory transfers as 2-way external mergesort uses to turn $M/B$ lists of size $x$ into $M/2B$ lists. Introducing the idea of merging many lists to the 2-way external mergesort gives way to the most popular external-memory sorting algorithm, external-memory $M/B$-way mergesort.

## 11.3 External-memory mergesort (M/B-way mergesort)

External-memory, or $M/B$-way mergesort, first introduced back in 1980s[2] employs the idea of merging many lists at once. It begins by creating the base-case $M$-sized sorted lists to be be further merged. Then it proceeds by merging $M/B$ lists at once into one list, thus increasing the list size between runs by a factor of $M/B$. In other words, we begin with lists of size $M$, then $M^2/B$, then $M^3/B$, etc, until we reach 1 list of size $N$. See Figure 11.5 for an example.

[2] A. Aggarwal and S. Vitter Jeffrey, "The input/output complexity of sorting and related problems," J Commun. ACM, vol. 31, no. 9, pp. 1116-1127, 1988.

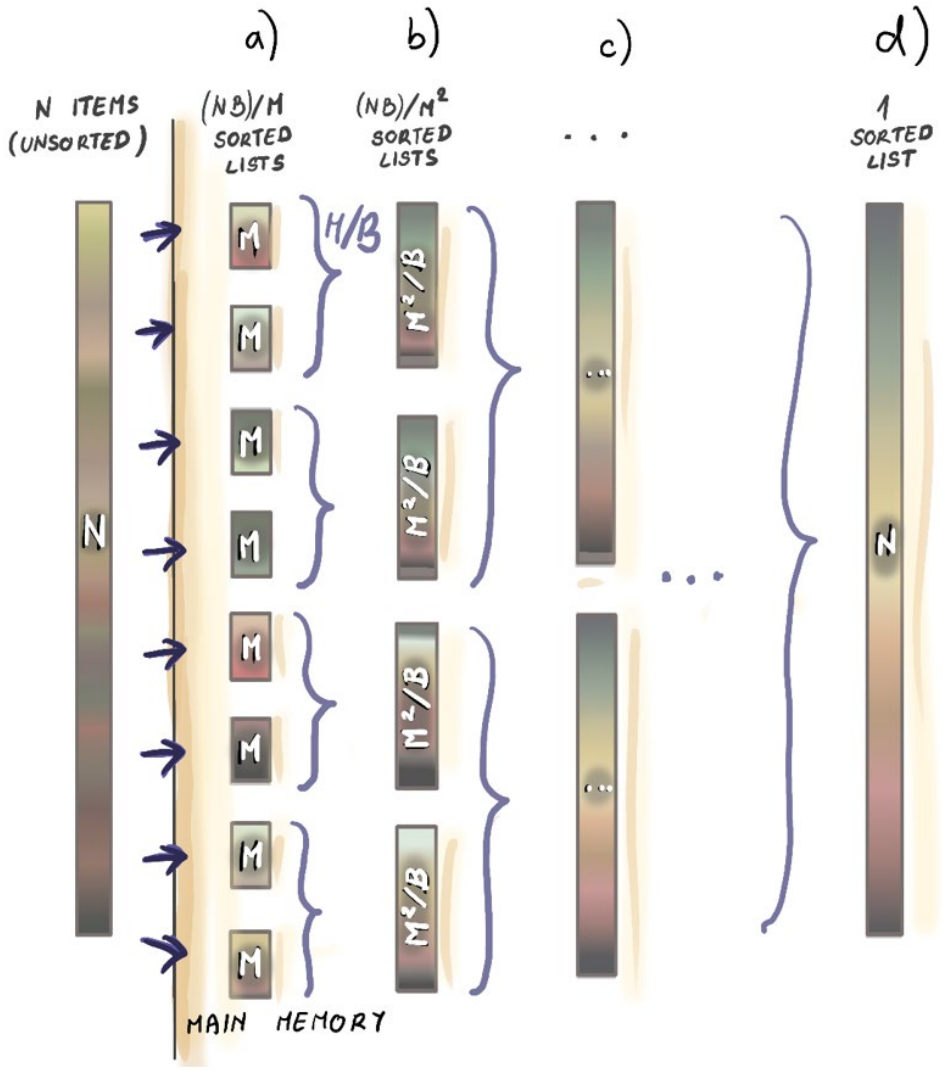



**Figure 11.5: In $M/B$-way mergesort, we begin by creating in one run $N/M$ sorted lists of size $M$. Then these lists are further merged, $M/B$ at one time, to eventually create one sorted list.**

The algorithm is like the classical internal mergesort in that it is recursive, so it trivially partitions the lists until they are of size $M$, sorts each one individually, and proceeds by recursively merging the lists. The recursive formula that describes $M/B$-way external mergesort is as follows:

$$T_{(M/B)ext}(N) = \frac{M}{B} T_{(M/B)ext}\left(\frac{N}{\frac{M}{B}}\right) + O\left(\frac{N}{B}\right)$$

The base case is the same as for the 2-way external mergesort, $T(M) = O(M/B)$. Let's unwrap this recursion. To get from lists of size $M$ to a list of size $N$ by always increasing list size by a factor of $M/B$, we need $\log_{M/B}\frac{N}{M}$ steps. Each step needs one sweep over the entire data, each of which costs $O(N/B)$ I/Os. Thus the total cost of the $M/B$-way external mergesort is $O(\frac{N}{B}\log_{M/B}\frac{N}{M})$ I/Os.

Mere expressions for runtimes might not mean much, but if we visually compare the formula for $M/B$-way external mergesort, and the 2-way external mergesort, the key difference appears in the base of the logarithm (2 vs. $M/B$). How big of a difference is this really? We are used to neglecting the base of the logarithm as it is usually a difference in a constant factor. However, here we have the parameters $M$ and $B$ involved, and we do not treat those as constants. The two runtimes differ by a factor of $\log_2\frac{M}{B}$. For many common choices of memory size and block size, the factor of difference might be even up to 30. Try and see for yourself by solving the following exercise, and comparing the results with those in Exercise 1.

**Exercise 11.2**

**Calculate how many block transfers are used by M/B-way external mergesort algorithm for our request data example. Recall that memory size is 4GB, and the total dataset size is 512GB. Use the same block size you used for Exercise 1.**

Even though in the $M/B$-way external mergesort, we attempt to optimize disk-related costs, the internal memory should not be neglected. The way we handle operations in main memory greatly affects the final execution of the algorithm. Our merging example from Chapter 9 that describes how heap can be used in main memory to merge $k$ sorted lists serves as a good example of a good use of memory in this type of algorithm. Specifically, in external mergesort, we can maintain $M/B$-sized heap to maintain the minima from each block representing its list. This way we can achieve both external-memory and internal-memory optimality.

### 11.3.1 Searching and sorting in RAM vs. external memory

Let's step back for a moment and think about the connection between searching and sorting, and how it changes when we go from internal to external memory. In internal memory, searching and sorting are highly related in the following sense: a balanced binary search tree, a data structure built for efficient searches (such as AVL tree, red-black tree, etc.) can also be used to sort data optimally. Insert (as well as lookup and delete) in a balanced binary search tree costs $O(\log_2 n)$, so $n$ inserts into the tree effectively sorts the data in $O(n \log_2 n)$.

comparisons, and one in-order traversal can output data in a linear order into an array. The per-element cost of sorting is then equal to per-element cost of searching, $O(\log_2 n)$.

Transferring that analogy to external memory, if we attempt to sort using a B-tree, we get the performance that is far from the optimal sorting algorithm --- $N$ inserts into a B-tree cost $O(N \log_B N)$ I/Os, or, if we think of top levels of a B-tree as resident in main memory, then $O(N \log_B \frac{N}{M})$ I/Os. The per-element cost of sorting is we use $M/B$-way external mergesort is only $O(\frac{1}{B} \log_{M/B} \frac{N}{M})$ I/Os, which is substantially less than time to insert into a B-tree. In other words, the per-element cost of sorting is much less than per-element cost of searching in external memory. That is, we cannot transfer the analogy to external memory.

This difference is important because it shows us that for batched problems, such as sorting, we can make good use of a large memory (merging many lists is a good example.) By batched problems, we mean problems where you need to process a ton of data and only give the result in the end. Batched lookup, for example, would mean getting a group of queries, and reporting answers to those queries once at the end. In the batched version of the lookup problem, we are optimizing the total amount of time to solve the entire problem and when this is the goal, we can think about the problem of multiple queries as a whole, ie, answering one query might help us answer another query, etc. In this setup, processing a lot of data at once can be very helpful, and large main memory can help us in doing that.

This is essentially different than if we are given the same queries, but we need to report answers one by one, and we are optimizing the sum of times taken to answer each query. This latter version feels more like the sequence of classical searching problems, and that sort of searching cannot make such a good use of a large memory except from storing top levels of a B-tree into main memory, because the outcome of comparison to elements from one block determines the next that should be brought in.

If the last two paragraphs feel to you like too much philosophy, you're probably right. But this is our last chapter, and we feel entitled to some degree of waxing philosophical. The point is, in sorting and other related problems, we truly benefit from having a large main memory whereas in some other problems, increasing main memory size might help but not that dramatically. You can assure yourself of this by looking at the runtimes of sorting vs searching and seeing how much the I/O-cost improves if main memory doubles (i.e., $M$ becomes $2M$.)

There are many other batched problems that can benefit from a large memory in the same way sorting does. So far, we have only seen the example of merging many lists as a benefit of large memory. Next, we will study the external version of quicksort, and see how large memory allows us to speed up the ordinary 2-way quicksort (that is, pivot selection) as well.

## 11.4 What about external quicksort?

Let's begin with a brief review of internal quicksort. Unlike mergesort, quicksort does most of its work top-down, by carefully partitioning data. The partition happens based on a chosen pivot, where data is further divided into elements smaller or equal to the pivot and elements

larger than pivot. Once the arrays to be partitioned become size 1, quicksort's work is effectively done.

In internal memory, quicksort has a better reputation than mergesort, and sort libraries more often employ quicksort than mergesort. This might seem unusual considering that quicksort does not offer optimal worst-case guarantees the way mergesort does. Deterministic quicksort that selects an arbitrary pivot at a fixed point (say, always from the first position in the array) can range in the performance from $O(n \log_2 n)$ to $O(n^2)$, and so does the randomized quicksort that selects the pivot at random. Yet, randomized quicksort is a much safer choice, as it effectively handles the case of almost sorted data, or any pattern in the data that might prove unfavorable for a fixed point choice of a pivot.

It is possible to force quicksort to sort in $O(n \log_2 n)$ by using median-of-medians worst-case linear-time selection algorithm[3], however, this algorithm has various practicality issues; on the other hand, to obtain the asymptotically optimal runtime, we do not need perfect medians.

One of the main benefits of quicksort is that it is an in-place algorithm, so all recursive calls actually work on the same part of the memory, the original array to be sorted. This means that we do not spend time copying over data and allocating extra memory, the factors that slow down mergesort. Saving space also saves time for the internal-memory quicksort, but let's see whether those effects translate into external memory.

In understanding how to effectively translate quicksort to external memory, our first exercise is to directly translate the ordinary 2-way quicksort, without any significant modifications to the algorithm.

### 11.4.1 External-memory 2-way quicksort

The direct adaptation of (randomized) 2-way quicksort to external memory is fairly straightforward. We randomly choose a pivot location, bring in the block containing the pivot, and then we sweep the whole file through memory block by block, deciding for each element whether it is smaller or equal or larger than the pivot. There are two buffer blocks in main memory that accumulate elements belonging to the two groups, and when a block is full on one side, we write it back to disk to its appropriate "side". After a linear number of memory transfers, we have performed one level of partition. See Figure 11.6 for how in-memory partition works:

[3] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest and R. E. Tarjan, "Time Bounds for Selection," Journal of Computer and System Sciences, vol. 7, no. 4, pp. 448-461, 1973.

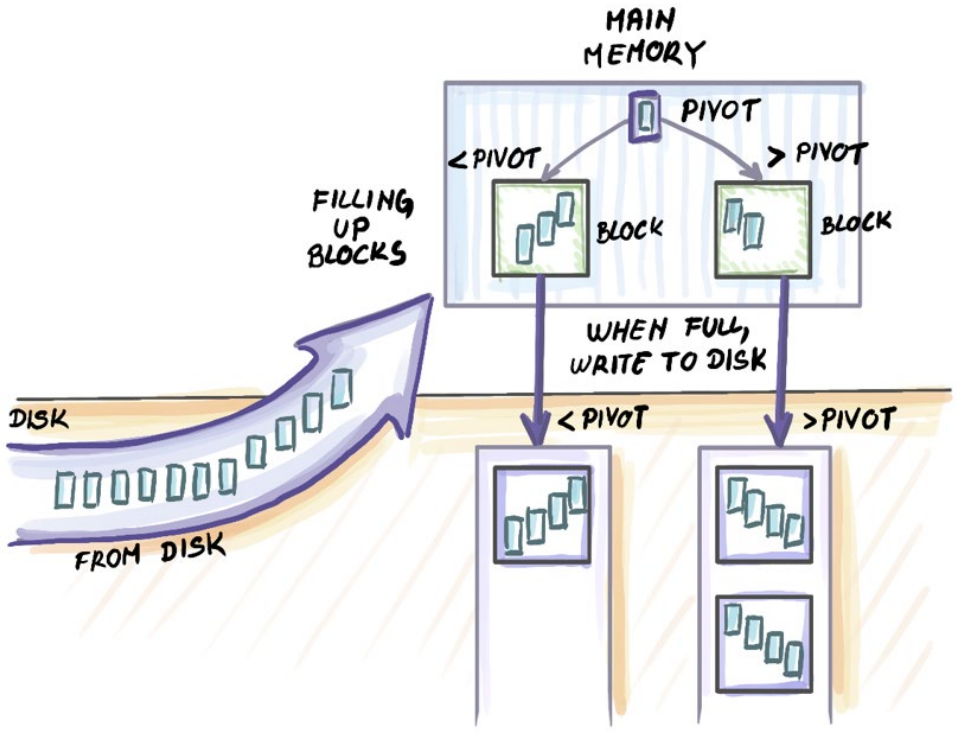



Figure 11.6: A snapshot during a partition in external- memory 2-quicksort. A data is sequentially input through main memory and each element is compared to the pivot. We have one block of buffer space to accumulate elements smaller than the pivot, and one block of buffer space to accumulate elements larger than the pivot. Once any of the buffer blocks are full, they are written back to a particular location on disk, where either left or the right side of the array is being appended. It is important, for the recursive calls later, that the elements that are smaller/larger than the pivot are all contiguously placed.

The partition step illustrated in the Figure 11.6 requires $O(N/B)$ I/Os. Then we recursively run the same algorithm on two separate pieces of files. Our base case occurs when the size of the file to be sorted is size of memory ($M$) or less. In that case, we pull the whole file and sort it in memory and write it back.

Let's assume for a moment that the pivot chosen always splits data into two equal halves. Then the recursion describing the runtime of 2-way external quicksort is identical to that one of 2-way external mergesort and it gives $O\left(\frac{N}{B}\log_2\frac{N}{M}\right)$ as the runtime.

### 11.4.2 Towards external-memory multi-way quicksort

Following the analogy of mergesort, to improve on the parallelism in this algorithm, we might think about increasing the number of pivots we find, and doing $M/B$-way partition instead of

2-way partition. Let's indulge this idea for a moment. Assuming that in $O(N/B)$ I/Os, we can find $O(M/B)$ pivots that partition the data into $O(M/B)$ sub-arrays. That would take us from this sort of recursion for 2-way external quicksort:

$$T_{2qext}(N) = 2T_{2qext}\left(\frac{N}{2}\right) + O(N/B)$$

to this type of recursion:

$$T_{(M/B)qext}(N) = \frac{M}{B}T_{(M/B)qext}\left(\frac{N}{\frac{M}{B}}\right) + O(N/B)$$

that would lead us to the runtime equivalent to that of $M/B$-way external mergesort. But not so fast… The conversion from 2-way external quicksort to multi-way external quicksort will not be that straightforward.

The main issue we are facing here is that it is not obvious how to find $O(M/B)$ good pivots and do partition in a linear number of block transfers. We could do this if we just resort to randomized pivots, but randomized pivots will not yield good partition.

Another idea is to could utilize the median-of-medians algorithm, that when transferred to external memory, requires $O(N/B)$ transfers to find one median. When applied recursively, this algorithm can find $O(M/B)$ medians in $O(\frac{N}{B}\log_2\frac{M}{B})$ I/Os --- this messes with our plan from above where we promised the partition work (the non-recursive part of the recursive formula $T_{(M/B)qext}(N)$) would be $O(N/B)$. There is, however, a way around.

### 11.4.3 Finding enough pivots

Turns out, there was a loophole in our thinking in the previous section. We said that to achieve the runtime of $M/B$-way mergesort using quicksort, we would need to do an $M/B$-way partition, i.e., be able to find $M/B$ well distributed pivots in $N/B$ I/Os. Turns out, we can get away with much fewer pivots, here's why: whatever we achieve runtime-wise with $O(M/B)$ pivots, we can also achieve (asymptotically speaking) with $\left(\frac{M}{B}\right)^c$ pivots for some constant $c$, $0 < c < 1$. So finding $\sqrt{M/B}$ pivots or even $\sqrt[3]{M/B}$ pivots still gives us the runtime asymptotically equal to that of $T_{(\frac{M}{B})qext}(N)$. Having $\sqrt{M/B}$ pivots will double the depth of the recursion tree, but this will not asymptotically affect the runtime. This will be our first relaxation of the problem: find $\sqrt{M/B}$ pivots instead of $M/B$ pivots.

The second relaxation will be one that we should have known from internal quicksort: the partition does not need to divide data into exactly equally-sized sub-partitions in order for the sorting algorithm perform asymptotically optimally. So to make our lives easier, we will translate this idea into external memory and try to find pivots that do not have to have exactly spaced-out ranks. They will be good enough in that they will separate data into $O(s)$ subarrays where $s = \sqrt{M/B}$, and all subarrays will be within a constant-factor size of each

other. So some subarrays might be twice or three times the size of other subarrays, and that's fine.

Let's take a second to understand why this will not be a problem. If we zoom back to the regular internal quicksort, recall that if every time we choose a pivot, the pivots falls exactly in the middle of the ordered array, we will get the performance of $O(n \log_2 n)$. Now if a pivot always falls somewhere in the middle half of ranks (i.e., it is never in the smallest 25% or the largest 25% of data), then the worst-case runtime is described using the following recurrence:

$$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + n$$

which also gives us $O(n \log_2 n)$. In fact, even if a pivot separates data into 1% and 99% of ranks, the runtime generated by the following recurrence:

$$T(n) = T(n/100) + T(99n/100) + n$$

still evaluates to $O(n \log_2 n)$. As long as the partitions are within constant sizes of each other, this should not give us the asymptotically worse performance than that given by perfect partitions. We will make use of this fact while finding $s$ approximate pivots for external multi-way quicksort (or, now we can call it $\sqrt{M/B}$-way quicksort). Here is how we can find $s$ approximate pivots.

### 11.4.4 Finding good enough pivots

We will split original set of $N$ elements into $N/M$ chunks, and sort each chunk. Then, from each chunk, we will select each $\alpha$th element. Take $\alpha = \frac{s}{4} = \frac{\sqrt{M/B}}{4}$. We will call the set of these selected elements $R \subseteq N$ (as in representatives); the set will have cardinality of $\sim N/\alpha$. Now we employ the median-of-medians selection algorithm recursively to find $s$ pivots in $R$.

First, we need to prove that it is possible to do this in a linear number of memory transfers. When the median-of-medians algorithm is applied recursively to the set of size $N/\alpha$ to find $s = 4\alpha$ pivots recursively, then it costs $O\left(\frac{N}{\alpha B} \log_2 4\alpha\right)$, which in total does not cost more than $O(\frac{N}{B})$ I/Os.

Next we need to show that the $s$ medians chosen from $R$ are approximate medians in $N$. The $s$ medians partition $R$ into $\sim s$ partitions of size $k = \frac{N}{s\alpha}$, and each of these elements is a representative we chose from some chunk of size $M$ in the original set. However, the chunks are not mutually ordered so one partition could have elements from different chunks. For instance, the first partition (one containing the smallest elements) might have 1 representative from the 1st chunk, 5 representatives from the 2nd chunk, 4 from the 3rd chunk, and so on. Either way, these representatives carry with themselves the elements that come before them and after them in the original set.

So the maximum of elements that $k$ representatives from a partition can carry with them is for each representative, $\alpha$ elements that come after, and for the first element in a chunk, the elements that come before it. This equals at most

$$C_1 = k * \alpha + \left(\frac{N}{M}\right) * \alpha = \frac{N}{s} + \frac{N\alpha}{M}$$

elements from the original set, and the least that one partition can carry is similarly:

$$C_2 = k * \alpha - \left(\frac{N}{M}\right) * \alpha = \frac{N}{s} - \frac{N\alpha}{M}$$

Because $s = \theta(\alpha)$, then also $C_1 = C_2$, thus showing that s medians found in $R$ are approximate and good enough medians for the original set $N$.[4]

### 11.4.5 Putting it all back together

Now that we know how to find pivots in linear time, let's see how this version of external $\sqrt{M/B}$-way quicksort would work (see Figure 11.7).

[4] This analysis is adopted from J. Erickson, "Jeff Erickson's Teaching: Lecture notes for CS473," [Online]. Available: https://jeffe.cs.illinois.edu/teaching/473/01-search+sort.pdf. [Accessed 09 2021].

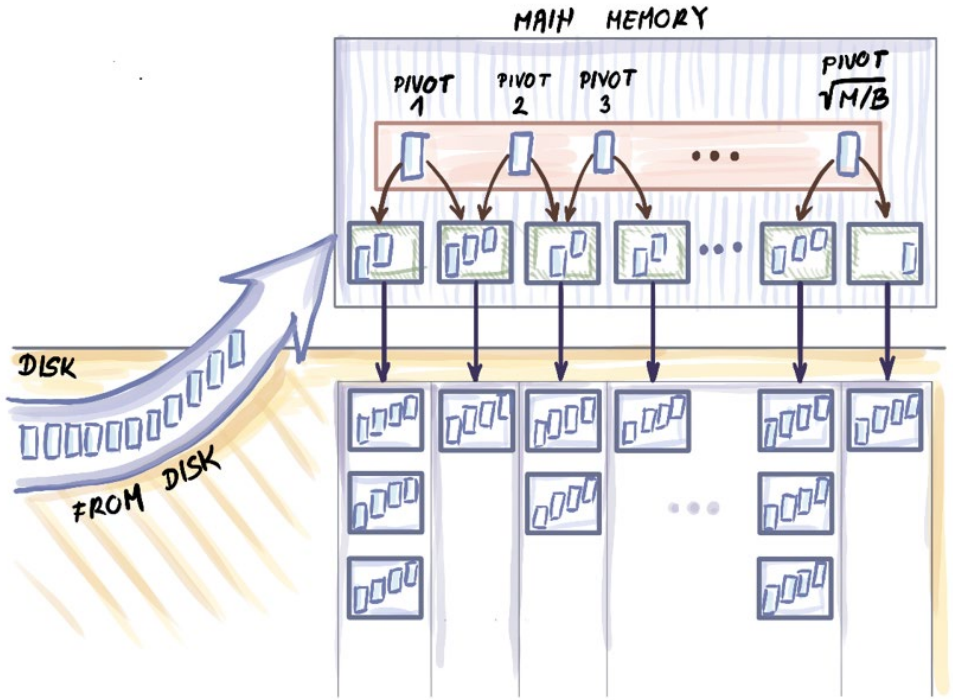



**Figure 11.7: Snapshot of external-memory multi-way quicksort. Instead of one pivot, we find $O(\sqrt{M/B})$ pivots and pull the entire dataset through memory. Based on comparisons with the pivots, each element is routed to the correct buffer block. Once the buffer block of any partition is filled, it is written back to memory and the buffer is emptied. After all data is pulled through main memory, we created $O(\sqrt{M/B})$ partitions that quicksort can again recurse on. Once a partition is of size $M$, the entire partition is read into main memory, sorted, and written back.**

It is important that we can simultaneously fit $s$ pivots into main memory, along with $s+1$ blocks that act as buffers for collecting elements. Once each block fills up, it is written back to disk. Once all elements have been processed, we recursively continue onto $s+1$ partitions.

The runtime of this algorithm equals to that of external $M/B$-way mergesort, $O(\frac{N}{B}\log_{M/B}\frac{N}{M})$ I/Os. In the next section, we will see that this bound is optimal for any external-memory sorting algorithm.

## 11.5 Math bit: Why is external-memory mergesort optimal

To understand why $M/B$-way mergesort (and $\sqrt{M/B}$-way quicksort) in external memory are optimal, it is important to first settle this question in internal memory. How do we know that the bound of $O(n \log_2 n)$ is optimal for sorting.

In the very beginning, when data is given to the sorting algorithm, we do not know which permutation of data represents the correct sorted order. One way to analyze the complexity of the sorting problem is think of how much one comparison can help us eliminate some permutations that do not represent the sorted order. For example, say we have a dataset of only 3 elements. There are $3! = 6$ potential permutations that might give us the final sorted order: $(a_1, a_2, a_3)$, $(a_1, a_3, a_2)$, $(a_2, a_1, a_3)$, $(a_2, a_3, a_1)$, $(a_3, a_1, a_2)$, $(a_3, a_2, a_1)$.

Say we compare $a_2$ to $a_3$ and learn that $a_2 < a_3$. This means that three permutations listed above, where $a_3$ comes before $a_2$, should be eliminated as potential outcomes. Because permutations are symmetric in this sense, we can assume that a good comparison can eliminate at most half of the remaining permutations. Note that if our algorithm is not good, a comparison might not cut down as much, or at all (imagine posing the same comparison over and over again.) But in the event that the algorithm is posing meaningful comparisons, we need at least $\log_2(n!)$ comparisons to get to one permutation. Simplifying this expression, we get that the lower bound for the sorting problem is $\Omega(n \log_2 n)$.

Now, how does this work in external memory? Our unit operation here is a block transfer, so the question becomes, how much can one block transfer help us reduce the number of candidate permutations. This will largely depend on the contents of a block being brought in and the contents of main memory. But for the lower bound, we are interested in what is the *most* that one block can help us during sorting. When one block is input, it has at most $B$ elements, and the memory has at most $M - B$ resident elements (see Figure 11.8).

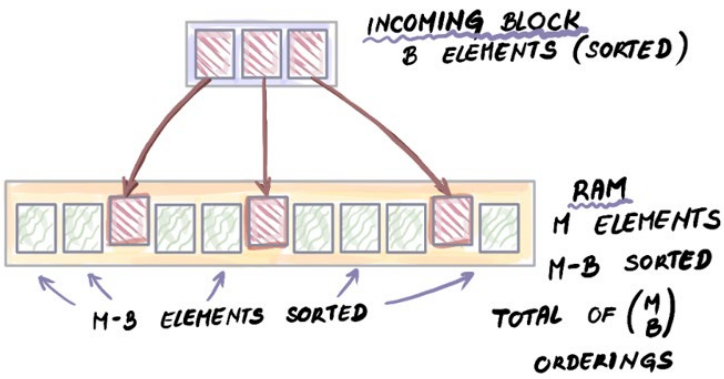



**Figure 11.8: To understand how much one block transfer can help in sorting, we analyze how many potential orderings of B elements (contents of one block) there are inside a memory-full of elements. There is a total of (M choose B) orderings, and this is a factor by which one memory transfer can reduce the total number of permutations remaining to be examined.**

To simplify the computation, we will assume that each individual block is sorted. This reduces the total number of permutations from $N!$ to $\frac{N!}{B!^{(N/B)}}$. Now that the block being brought into main memory is sorted, and the memory itself is sorted, then the total number of options for where the B elements might land is $\binom{M}{B}$. Based on these two quantities, we obtain that the lower bound for sorting in external memory is:

$$\log_{\binom{M}{B}} \frac{N!}{B!^{(N/B)}}$$

which after some algebraic manipulation comes out to $\Omega\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{M}\right)$, the bound that matches our external-memory sorting algorithms. The analysis above has been adapted from the source[5] that you can consult if you wish to understand more details about algebraic manipulation of this lower bound and lower bounds in general.

## 11.6 Wrapping up

With the sorting chapter, we have arrived to the end of this book. Whether you picked up this book because you are trying to implementing probabilistic data structures to solve a problem in a specific domain or you are beefing up on your large-scale system and algorithm design for an interview at a big-data company, we hope this book was a good investment of your time. If not, we hope you at least enjoyed the illustrations.

If you're only getting started in the field of massive-scale algorithms, our hope is that after reading this book, you have a better understanding of the range of algorithmic problems that large datasets introduce in modern-day systems --- and more importantly, that you find them exciting. We hope we convinced you that problems such as set membership, searching, sorting, cardinality estimation, sampling and database indexing for massive datasets are intriguing and challenging problems. Also, that thinking about ways of solving them helped you develop or deepen a new, more nuanced view of efficiency and performance.

Ultimately, resulting tradeoffs from limited space and time when working with large data forces us to think more creatively about problems than ever before and also embrace error and imperfection. Working with massive datasets teaches us that we can't have it all (not that we needed massive datasets to teach us this!). With a growing gap between our resources and the size of data that applications process, it is clear that the success of many applications today will be determined by how well they grapple with scalability challenges. To successfully do that, we need engineers that can wear many hats and are able to combine the algorithmic and programming know-how with the domain knowledge and mathematical underpinnings of data structures and algorithms. This book represents our small contribution to the education of such a versatile engineer.

[5] J. Erickson, "Jeff Erickson's Teaching: Lecture notes for CS473," [Online]. Available: https://jeffe.cs.illinois.edu/teaching/473/01-search+sort.pdf. [Accessed 09 2021].

## 11.7 Summary

- Sorting is one of the best-known problems in computer science, and there is a large body of research optimizing sorting algorithms for different contexts.
- When data can not fit into main memory, the sorting algorithm needs to bring in small pieces of data into main memory and sort chunk by chunk.
- $M/B$-way external mergesort is the algorithm of choice for when data is too large to fit into RAM. This algorithm merges many lists at one, thus making use of a large available memory.
- Analogously, it is possible to adapt quicksort to work optimally in external memory by choosing a larger set of pivots and thus partitioning data into many individual sub-partitions instead of just 2.

Batched problems like sorting have a cheaper per-element cost than searching in external memory. This is an important difference between RAM and external memory; in RAM, we can optimally sort by inserting into a search structure, while doing that in external memory results with a suboptimal algorithm

- To understand whether a sorting algorithm is optimal, it is important to understand how sorting lower bounds work. In case of internal memory, the key is understanding how much one comparison can contribute to eliminating permutations that are not the sorted order, and in external memory, we do the same, but by analyzing how much one block input can help us eliminate the permutations.