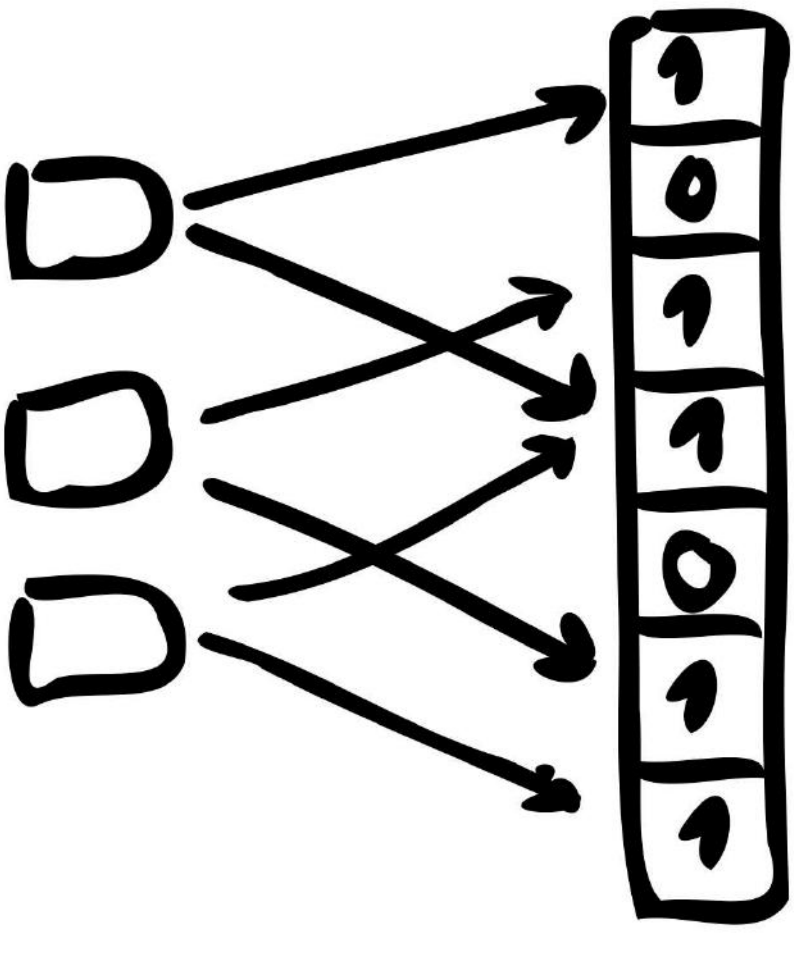
PROBABILISTIC

DATA STRUCTURES AND ALGORITHMS FOR BIG DATA APPLICATIONS



ANDRII GAKHOV

PROBABILISTIC DATA STRUCTURES AND ALGORITHMS

FOR BIG DATA APPLICATIONS

ANDRII GAKHOV

gaKhov

Probabilistic Data Structures and Algorithms
for Big Data Applications

1st edition, 2019

Bibliographic information published by the Deutsche Nationalbibliothek:

The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available on the Internet at http://dnb.dnb.de.

The paperback edition is printed and published by
BoD — Books on Demand GmbH
22848 Norderstedt
Germany

ISBN (paperback): 978-37-48190-48-6

ASIN (ebook): B07MYKTY8W

To my wife Larysa
and my son Gabriel.

Table of Contents

Preface

Big data is characterized by three fundamental dimensions: *Volume*, *Velocity*, and *Variety*, **The Three V's of Big Data**. The *Volume* expresses the amount of data, *Velocity* describes the speed at which data is arriving and being processed, and *Variety* refers to the number of types of data.

The data could come from anywhere, including social media, various sensors, financial transactions, etc. IBM has stated[1] that people create **2.5 quintillion** bytes of data **every day**, this number is growing constantly and most of it cannot be stored and is usually wasted without being processed. Today, it is not uncommon to process terabyte- or petabyte-sized corpora and gigabit-rate streams.

On the other hand, nowadays every company wants to fully understand the data it has, in order to find value and act on it. This led to the rapid growth in the Big Data Software market. However, the traditional technologies which include data structures and algorithms, become ineffective when dealing with Big Data. Therefore,

[1] What Is Big Data? `https://www.ibm.com/software/data/bigdata/what-is-big-data.html`

many software practitioners, again and again, refer to computer science for the most appropriate solutions and one option is to use probabilistic data structures and algorithms.

Probabilistic data structures is a common name for data structures based mostly on different hashing techniques. Unlike regular (or deterministic) data structures, they always provide approximated answers but with reliable ways to estimate possible errors. Fortunately, the potential losses and errors are fully compensated for by extremely low memory requirements, constant query time, and scaling, the factors that become essential in Big Data applications.

About this book

The purpose of this book is to introduce technology practitioners which includes software architects and developers, as well as technology decision makers to probabilistic data structures and algorithms. Reading this book, you will get a theoretical and practical understanding of probabilistic data structures and learn about their common uses.

This is not a book for scientists, but to gain the most out of it you will need to have basic mathematical knowledge and an understanding of the general theory of data structures and algorithms. If you do not have any “computer science” experience, it is highly recommended you read *Introduction to Algorithms* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (MIT), which provides a comprehensive introduction to the modern study of computer algorithms.

While it is impossible to cover all the existing amazing solutions, this book is to highlight their common ideas and important areas of application, including membership querying, counting, stream mining, and similarity estimation.

Organization of the book

This book consists of six chapters, each preceded by an introduction and followed by a brief summary and bibliography for further reading relating to that chapter. Every chapter is dedicated to one particular problem in Big Data applications, it starts with an in-depth explanation of the problem and follows by introducing data structures and algorithms that can be used to solve it efficiently.

The first chapter gives a brief overview of popular hash functions and hash tables that are widely used in probabilistic data structures. Chapter 2 is devoted to approximate membership queries, the most well-known use case of such structures. In chapter 3 data structures that help to estimate the number of unique elements are discussed. Chapters 4 and 5 are dedicated to important frequency- and rank-related metrics computations in streaming applications. Chapter 6 consists of data structures and algorithms to solve similarity problems, particularly — the nearest neighbor search.

This book on the Web

You can find errata, examples, and additional information at https://pdsa.gakhov.com. If you have a comment, technical question about the book, would like to report an error you found, or any other issue, send email to pdsa@gakhov.com.

In case you are also interested in Cython implementation that includes many of the data structures and algorithms from this book, please check out our free and open-source Python library called PDSA at https://github.com/gakhov/pdsa. Everybody is welcome to contribute at any time.

About the author

Andrii Gakhov is a mathematician and software engineer holding a Ph.D. in mathematical modeling and numerical methods. He has been a teacher in the School of Computer Science at V. Karazin Kharkiv National University in Ukraine for a number of years and currently works as a software practitioner for ferret go GmbH, the leading community moderation, automation, and analytics company in Germany. His fields of interests include machine learning, stream mining, and data analysis.

The best way to reach the author is via Twitter @gakhov or by visiting his webpage at https://www.gakhov.com.

Acknowledgments

The author would like to thank Asmir Mustafic, Jean Vancoppenolle, and Eugen Martynov for the contribution to reviewing this book and for their useful recommendations. Big gratitude to academia reviewers Dr. Kateryna Nesvit and Dr. Dharavath Ramesh for their invaluable suggestions and remarks.

Special thanks to Ted Dunning, the author of the t-digest algorithm, for a very precise review of the corresponding chapter, the insightful questions, and discussion.

Finally, thanks to all the people who provided feedback and helped make this book possible.

1 Hashing

Hashing plays the central role in probabilistic data structures as they use it for randomization and compact representation of the data. A *hash function* compresses blocks of input data of an arbitrary size by generating an identifier of a smaller (and in most cases fixed) size, called the *hash value* or simply the *hash*.

The choice of hash functions is crucial to avoid bias. Although the selection decision is mostly based on the input and particular use cases, there are certain common properties that a hash function should fulfill in order to be applicable for hash-based selection.

Hash functions compress the input, therefore, cases where they generate the same hash values for two different blocks of data are unavoidable and known as *hash collisions*.

In 1979 J. Lawrence Carter and Mark Wegman proposed the *universal hash functions* whose mathematical properties can guarantee a low expected number of collisions, even if the input data are chosen randomly from the *universe*.

The universal hash functions family H maps elements of the universe to the range $\{0, 1, \ldots, m-1\}$ and guarantees that by randomly picking a hash function from the family the probability of collisions is limited:

$$\Pr\Big(h(x) = h(y)\Big) \leq \frac{1}{m}, \text{ for any } x, y : x \neq y. \tag{1.1}$$

Thus, the random choice of a hash function from the family with property (1.1) is precisely the same as choosing an element uniformly at random.

An important universal hash functions family, designed to hash integers, can be defined as

$$h_{\{k,q\}}(x) = ((k \cdot x + q) \bmod p) \bmod m, \tag{1.2}$$

where k and q are randomly chosen integers modulo p with $k \neq 0$. The value of p should be selected as a prime $p \geq m$, and the common choice is to take one of the known Mersenne prime numbers, e.g., for $m = 10^9$ we choose $p = \mathrm{M}_{31} = 2^{31} - 1 \approx 2 \cdot 10^9$.

Many applications can use the simpler version of the family (1.2):

$$h_{\{k\}}(x) = (k \cdot x \bmod p) \bmod m, \tag{1.3}$$

this is only approximately universal, but still provides a good probability of collisions smaller than $\frac{2}{m}$ in expectation.

However, the above families of hash functions are limited to integers, that is not enough for most practical applications which require to hash variable-sized vectors and are in demand of fast and reliable hash functions with certain guaranteed properties.

There are many classes of hash functions used in practice and the choice mainly depends on their design and particular use. In the current chapter we provide an overview of popular hash functions and simple data structures that are prevalent in various probabilistic data structures.

1.1 Cryptographic hash functions

Practically, *cryptographic hash functions* are defined as fixed mappings from variable input bit strings to fixed length output bit strings.

As stated previously, hash collisions are unavoidable, but a secure hash function is required to be *collision resistant*, meaning that it should be hard to find collisions. Of course, a collision can be found accidentally or computed in advance. This is why such a class of functions always requires mathematical proofs.

Cryptographic hash functions are very important in cryptography and are used in many applications such as digital signatures, authentication schemas, and message integrity.

There are three main requirements that cryptographic hashes are expected to satisfy:

- *Work factor* — to make brute force inversion hard, a cryptographic hash should be computationally expensive.
- *Sticky state* — cryptographic hash should not have a state in which it can stick for a plausible input pattern.
- *Diffusion* — every output bit of a cryptographic hash should be an equally complex function of every input bit.

Theoretically, cryptographic functions can be further divided into *keyed hash functions*, that use a secret key, and *unkeyed hash functions*, which do not. Probabilistic data structures use only unkeyed hash functions, which include *One–Way hash functions*, *Collision Resistant hash functions*, and *Universal One–Way hash functions*. These functions differ only in some additional properties.

One–Way hash functions satisfy the following requirements:

- They can be applied to blocks of data of any length (of course, in practice, it's bounded by some huge constant).
- They produce a fixed-length output.
- They should have *preimage resistance* (*one-way property*) — it should be computationally infeasible to find an input which hashes to the specified output.

Additionally, for Collision Resistant hash functions it should be extremely unlikely for two different inputs to generate the same hash value.

If not collision resistant, Universal One–Way hash functions need to be *target collision resistant* or *second-preimage collision resistant* — it should be computationally infeasible to find a second distinct input that hashes to the same output as the specified input.

Note, that being collision resistant implies that the function is second-preimage resistant, but the generic complexity of finding a second-preimage resistance function is much higher than finding a colliding pair.

Because of their design (particularly, the work factor requirement), cryptographic hash functions are much slower than non-cryptographic ones. For instance, the function SHA–1, discussed below, is in the order of 540 MiB/second[1], but the popular non-cryptographic functions are in the order of 2500 MiB/second and more.

[1]Crypto++ 6.0.0 Benchmarks `https://www.cryptopp.com/benchmarks.html`

Message–Digest Algorithms

The popular *Message–Digest Algorithm*, MD5, was invented by Ron Rivest in 1991 to replace the old MD4 standard. It is a cryptographic hash algorithm, defined in IETF RFC 1321, that takes a message of an arbitrary length and produces as an output the unique 128-bit hash of the input.

The MD5 algorithm is based on the Merkle–Damgård schema. At the first stage, it converts the input of an arbitrary size to a number of blocks of a fixed size (512-bit blocks or sixteen 32-bit words) using an *MD–compliant padding* function. Afterwards, such blocks are processed one by one using a special compression function and every next block uses the result of the previous output. To make the compression secure, the algorithm applies Merkle–Damgård strengthening, then the padding uses the encoded length of the original message. The final MD5 hash digest is the 128-bit value generated after the processing of the last block.

The MD5 algorithm is often used to verify the integrity of a file — instead of confirming that the file is unchanged by examining its raw data, it is enough to compare the MD5 hashes.

As stated in Vulnerability Note VU#836068[2], the MD5 algorithm is vulnerable to collision attacks. The discovered weaknesses in the algorithm allow for the construction of different messages with the same MD5 hash. As a result, attackers can generate cryptographic tokens or other data that illegitimately appears authentic. It is not advisable to use it as a secure cryptographic algorithm anymore, however, such vulnerability doesn't have a big impact for probabilistic data structures and can still be used.

Secure Hash Algorithms

Secure Hash Algorithms were developed by the US National Security Agency (NSA) and published by the National Institute of Standards and

[2] VU#836068 `http://www.kb.cert.org/vuls/id/836068`

Technology (NIST). The first algorithm from the family, called SHA–0, was published in 1993 and quickly replaced by its successor SHA–1, which became widely accepted globally. SHA–1 produces a longer 160-bit (20-byte) hash value, while its security has been increased by fixing the weaknesses of SHA–0.

SHA–1 was widely used for years in various applications, and most websites were signed using algorithms based on it. However, in 2005 a weakness in SHA–1 was discovered, so in 2010 NIST deprecated it for government use and it also got deprecated on the Internet since 2011. Same as with MD5, the found weaknesses didn't impinge on its usage as a hash function for probabilistic data structures.

SHA–2 was published in 2001 and included six hash functions with varying digest sizes: SHA–224, SHA–256, SHA–384, SHA–512, and others. SHA–2 is stronger than SHA–1 and attacks made against SHA–2 are unlikely to happen with current computing power.

RadioGatún

The cryptographic hash function family called RadioGatún was presented at the Second Cryptographic Hash Workshop in 2006 [Be06]. The design of RadioGatún improved the known *Panama* hash function.

Similar to other popular hash functions, the input is split into a sequence of blocks which are injected into the algorithm's internal state using a special function, that is followed by an iterative application of a single non-cryptographic round function (called the *belt-and-mill* round function). At every round, the state is represented as two parts, the belt and the mill, that are treated differently by the round function. The application of the round function consists of four operations in parallel: 1) non-linear function applied to the mill, 2) simple linear function applied to the belt, 3) feedforward some bits of the mill to the belt in a linear way, 4) feedforward some bits of the belt to the mill in a linear way. After injection of all input blocks, the algorithm performs a number of rounds without input or output (blank rounds) after which a part of the state is returned as the final hash value.

Among the family, RadioGatún64, with 64-bit words, is the default choice and is optimal for 64-bit platforms. For best performance on 32-bit platforms, RadioGatún32, with 32-bit words, can also be used.

For the same clock frequency, RadioGatún32 is claimed to be 12 times faster than SHA–256 for long inputs, and 3.2 times faster for short inputs,while having fewer gates. RadioGatún64 is even 24 times faster than SHA–256 for long inputs but has about 50% more logic gates.

1.2 Non-Cryptographic hash functions

In contrast to cryptographic hash functions, non-cryptographic functions are not designed to fend off attacks aimed at finding a collision, hence don't require security and high collision resistance.

Such functions simply have to be fast and guarantee a low probability of collisions, allowing a lot of data to be quickly hashed with a reasonable error probability.

Fowler/Noll/Vo

The basis of the *Fowler/Noll/Vo* (FNV or FNV1) non-cryptographic hash algorithm was taken from an idea sent, as a reviewer comment, to the IEEE POSIX P1003.2 committee by Glenn Fowler and Phong Vo back in 1991 and afterward improved on by Landon Curt Noll [Fo18].

The FNV algorithm maintains an internal state that is initialized to a special offset basis. After that, it iterates over the input blocks of 8 bits and performs the multiplication of the state on some large numerical constant, called the *FNV Prime*, followed by applying logical exclusive OR (XOR) to the input block. After the last input is processed, the resulting value of the state is reported as the hash.

The FNV Prime and the offset basis constants are design parameters and depend on the bit length of the produced hash values. As mentioned by Landon Curt Noll, the selection of the primes is the part of the magic of the FNV algorithm, and some primes do hash better than others for the same hash size.

The FNV1a alternate algorithm, that currently has to be preferred, is a minor variation of the FNV algorithm that differs only in the order of the internal XOR and multiplication operations. Although FNV1a uses the same parameters and the FNV Prime as the FNV1, its XOR–folding provides slightly better dispersion without interfering with the CPU performance.

Currently, the FNV family includes algorithms for 32-, 64-, 128-, 256-, 512-, and 1024-bit hash values.

The FNV is very simple to implement, but its high dispersion of the hash values makes them well suited for hashing nearly identical strings. It is widely used in DNS servers, Twitter, database indexing hashes, web search engines, and many other places. Some years ago, the 32-bit version of the FNV1a was recommended as the hash algorithm for IPv6 flow label generation [An12].

MurmurHash

Another well-known family of hash functions, called *MurmurHash*, was published by Austin Appleby in 2008 and finalized as the MurmurHash3 algorithm in 2011 [Ap11].

The MurmurHash algorithms use a special probabilistic technique for approximating the global optimum to find a hash function that mixes the bits of the input value in the best way to produce the bits of the output hash. The various generations of the algorithm differ mainly in their mixing functions.

The algorithm is claimed to be twice as fast as the speed-optimized *lookup3* hash function[3]. MurmurHash3 includes 32- and 64-bit versions for x86 and x64 platforms.

Currently, MurmurHash3 is one of the most popular algorithms and is used in Apache Hadoop, Apache Cassandra, Elasticsearch, libstdc++, nginx, and others.

CityHash and FarmHash

In 2011, Google published a new family of hash functions for strings, called *CityHash*, developed by Geoff Pike and Jyrki Alakuijala [Pi11]. CityHash functions are simple non-cryptographic hash functions that are based on the MurmurHash2 algorithm.

The CityHash family were developed with the focus on short strings (e.g., up to 64 bytes) that have the most interest in probabilistic data structures and hash tables. It includes 32-, 64-, 128- and 256-bit versions. For such short strings, the 64-bit version CityHash64 is faster than MurmurHash and outperforms the 128-bit CityHash128. However, while for long strings with at least a few hundred bytes the CityHash128 is preferred over other hash functions of the CityHash family, in practice, it is better to use MurmurHash3 instead.

One of the downsides of the CityHash is that it is fairly complex and leads to non-optimal behavior on different compilers that can significantly degrade its speed.

In 2014 Google published a successor to CityHash called the *FarmHash*, developed by Geoff Pike [Pi14]. The new algorithm included most of the techniques used in CityHash (and, unfortunately, inherited its complexity) and the new generation of MurmurHash. FarmHash functions mix the input bits thoroughly, but it is not enough to be used in cryptography.

The FarmHash uses CPU specific optimizations and still requires tuning of the compiler to get the best performance and is platform dependent.

[3]Hash Functions and Block Ciphers `https://burtleburtle.net/bob/hash/`

Notably, the computed hash values also differ across platforms.

The FarmHash functions come in many versions, and the 64-bit version Farm64 outperforms algorithms such as CityHash, MurmurHash3, and FNV in tests on many platforms, including mobile phones.

1.3 Hash tables

A *hash table* is a dictionary data structure that is comprised of unordered associative array of length m whose entries are called buckets and are indexed by a key in the range $\{0, 1, \ldots, m - 1\}$. To insert an element into the hash table, a hash function is used to compute the key that is utilized to select the appropriate bucket to store the value.

Typically, the universe from which we draw the input elements is much bigger than the capacity m of the hash table, hence collisions in keys are unavoidable. Additionally, when the number of elements in the hash table grows, the number of collisions rises as well.

The critical concept of hash tables is the *load factor* α, the ratio of the number of used keys n to the table's total length m:

$$\alpha := \frac{n}{m}.$$

The load factor is a measure of how full the hash table is and since n cannot exceed the capacity of the hash table it is upper bounded by one. When α approaches its maximal value, the probability of collision increases significantly which can necessitate an increase in capacity.

All hash table implementations need to address the problem of collisions and provide a strategy on how to handle them. There are two main techniques:

- *Closed addressing* — to store collided elements under the same keys in a secondary data structure.

- *Open addressing* — to store collided elements in positions other than their preferred positions and provide a way to address them.

The closed addressing technique is the most obvious way to resolve collisions. There are many different implementations, for instance, *separate chaining* that stores collided elements in a linked list, *perfect hashing* that uses special hash functions and secondary hash tables of different lengths.

Instead of creating a secondary data structure in either form, it is possible to resolve collisions by storing the collided elements elsewhere in the primary table and providing an algorithm on how to address them. Since the address of the element is not known from the beginning, this technique is known as open addressing.

Now we will cover two open addressing implementations that are useful in the probabilistic data structures listed in this book.

Linear probing

One of the most straightforward hash table implementations that uses open addressing is the *Linear probing* algorithm, invented by Gene Amdahl, Elaine M. McGraw, and Arthur Samuel in 1954 and analyzed by Donald Knuth in 1963. The idea of the algorithm is to place collided elements into the next empty bucket. Its name originates from the fact that the final position of the element will be linearly shifted from the preferable bucket since we probe one bucket after another.

A LinearProbing hash table can be seen as a circular array that stores indexed values in buckets. To insert a new element x, we compute its key $k = h(x)$ using a single hash function h. If the bucket that corresponds to that key is non-empty and contains a different value, meaning a collision, we keep looking clockwise at the next buckets until we find a free space where we can index the element x. Monitoring of the load factor of the hash table can guarantee that we will definitely find a free space at some point.

Similarly, when we want to lookup for some element x, we compute its key k using the same hash function h and start checking the buckets clockwise, starting at the preferable bucket with the key $k = h(x)$, until we found the wanted element x or the first empty bucket appears, resulting in the decision that the element is not in the table.

Example 1.1: Linear probing

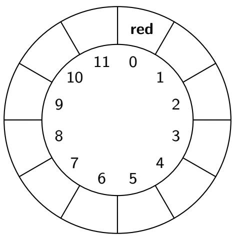
Consider a LinearProbing hash table of length $m = 12$ and a hash function based on 32-bit MurmurHash3 that maps the universe to the range $\{0, 1, \ldots, m-1\}$:

$$h(x) \coloneqq \text{MurmurHash3}(x) \bmod m.$$

Suppose that we want to store different names of colors in the hash table, starting from ***red***. The value of the hash function for the element is

$$h = h(\boldsymbol{red}) = 2352586584 \bmod 12 = 0.$$

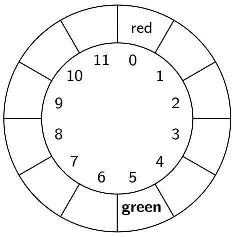
Since the LinearProbing hash table is empty at the beginning, the bucket with the key $k = 0$ contains no elements, therefore we just index the element there:



Next, we take the element ***green***, whose hash value is

$$h = h(\boldsymbol{green}) = 150831125 \bmod 12 = 5.$$

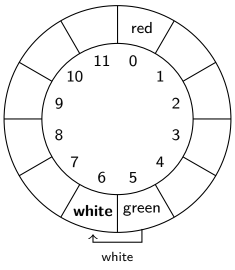
The key is $k = 5$, as this bucket is empty we again freely store the element.



Now, consider the element ***white***. Its hash value is

$$h = h(\boldsymbol{white}) = 16728905 \bmod 12 = 5.$$

The preferable bucket for that element is the one with the key $k = 5$. However, the bucket is already occupied by a different element, meaning a collision has appeared. In this case, we apply the Linear probing algorithm and try to find the next empty bucket going clockwise from the preferable bucket position. Fortunately, the next bucket, under key $k = 6$ is free and we store the element ***white*** there.



When we lookup for the element ***white*** in the LINEARPROBING hash table, we first check its preferable bucket, with the key $k = 5$. Since that bucket contains a value that differs from the element, we start checking buckets in a clockwise direction, starting from the key $k + 1 = 6$. Fortunately, the next bucket with the key $k = 6$ contains the wanted value and we can conclude that the element is present in the hash table.

The algorithm requires O(1) time for each operation, as long as the LINEARPROBING hash table is not full (the load factor is strictly less than one). The longest probe sequence in Linear probing is of expected length $O(\log n)$.

The Linear probing algorithm is very sensitive to the choice of the hash function h because it must provide ideal uniform distribution. Unfortunately, in practice, it is not possible, and the performance of the algorithm degrades rapidly as the actual distribution diverges. To address this problem, a variety of techniques for additional randomization are widely used.

Cuckoo hashing

Another implementation of open addressing is *Cuckoo hashing*, introduced by Rasmus Pagh and Flemming Friche Rodler in 2001 and published in 2004 [Pa04]. The main idea of the algorithm is to use two hash functions instead of one.

The Cuckoo hash table is an array of buckets, where instead of one preferable bucket as in Linear probing and many other algorithms, each element has two candidate buckets determined by two different hash functions.

To index a new element x into the Cuckoo table, we compute keys for two candidate buckets with the hash functions h_1 and h_2. If at least one of those buckets is empty, we insert the element into that bucket. Otherwise, we randomly choose one of those buckets and store element x there, while moving the element from that bucket to its alternative candidate bucket. We repeat this procedure until an empty bucket is found, or until a maximum number of displacements is reached. If there are no empty buckets, the hash table is considered full.

Although Cuckoo hashing may execute a sequence of displacements, it keeps the constant time $O(1)$ to be finished.

The lookup procedure is straightforward and can be done in constant time. We simply need to determine the candidate buckets for the input element by computing its hashes h_1 and h_2 and check if such an element is present in one of those buckets. The deletion procedure can be performed in a similar way.

Example 1.2: Cuckoo hashing

Consider a Cuckoo hash table of length $m = 12$ with two 32-bit hash functions MurmurHash3 and FNV1a that produce values in the range $\{0, 1, \dots, m-1\}$:

$$h_1(x) := \text{MurmurHash3}(x) \bmod m,$$
$$h_2(x) := \text{FNV1a}(x) \bmod m.$$

Like in Example 1.1, we index color names in the hash table starting with ***red***. The keys of the candidate bucket we obtain by applying those hash functions:

$$h_1(\boldsymbol{red}) = 2352586584 \bmod 12 = 0,$$
$$h_2(\boldsymbol{red}) = 1089765596 \bmod 12 = 8.$$

The Cuckoo hash table is empty, so we use one of the candidate buckets, for instance, the bucket with the key $k = h_1(\boldsymbol{red}) = 0$ and index the element.

0	1	2	3	4	5	6	7	8	9	10	11
red											

Next, we index element ***black*** whose candidate buckets are $h_1(\boldsymbol{black}) = 6$ and $h_2(\boldsymbol{black}) = 0$. Since the bucket with the key $k = 0$ is occupied by another element, we can only index it into the alternative bucket $k = 6$, which is free.

0	1	2	3	4	5	6	7	8	9	10	11
red						**black**					

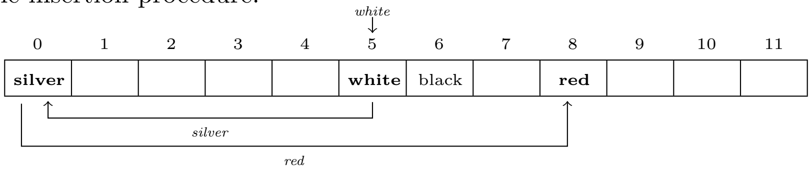
There is a similar situation with the element ***silver*** with $h_1(\boldsymbol{silver}) = 5$ and $h_2(\boldsymbol{silver}) = 0$. We store this element in the bucket with the key $k = 5$ since 0 is occupied.

0	1	2	3	4	5	6	7	8	9	10	11
red					**silver**	black					

Now consider the element ***white***. The hash values of this element are

$$h_1(\textbf{\textit{white}}) = 16728905 \bmod 12 = 5,$$
$$h_2(\textbf{\textit{white}}) = 3724674918 \bmod 12 = 6.$$

As we can see, both candidate buckets for this element are occupied, and we have to perform the displacements according to the Cuckoo hashing schema. First, randomly select one of the candidate buckets, let's say the bucket with the key $k = 5$ and put the element ***white*** into it. The element ***silver*** from the bucket 5 has to be relocated to its alternative bucket, which is 0. As we can see, the bucket with the key 0 is not empty; therefore, we store element ***silver*** and move element ***red*** from that bucket to its other candidate bucket. Fortunately, the alternative bucket with the key 8 for element ***red*** is free and after storing it in that bucket, we finish the insertion procedure.



For instance, when we want to lookup the element ***silver***, we check only its candidate buckets, which are 5 and 0, as we computed earlier. Since this element is present in one of them, in the bucket with the key 0 in this case, we conclude that the element ***silver*** is present in the Cuckoo hash table.

Cuckoo hashing ensures high space occupancy but requires the length of the hash table to be slightly larger than the space needed to keep all elements. A modification of the Cuckoo hash schema is used in a probabilistic data structure called the Cuckoo filter, which we will describe in detail in the next chapter.

Conclusion

In this chapter we covered an overview of hashing, its problems and importance in data structures. We discussed cryptographic versus non-cryptographic hash functions, reviewed a list of the functions that are most used in practice, and learned about universal hashing which is very important theoretically. As an application of the hash functions we have considered hash tables, which are simple data structures that map keys to values and answer membership queries. We studied examples of open addressing hash tables that we will use in the next chapters for probabilistic data structures.

If you are interested in more information about the material covered here, please take a look at the list of references that follows this chapter.

In the next chapter we will be discussing first probabilistic data structures and studying extensions of hash tables, called filters, that are used to answer membership queries under requirements that are common for Big Data applications, such as when storage is at a premium and the speed of lookups must be as fast as possible.

Bibliography

[An12] Anderson, L., et al. (2012) “Comparing hash function algorithms for the IPv6 flow label”, *Computer Science Technical Reports*, 2012.

[Ap11] Appleby, A. (2011) “MurmurHash”, *sites.google.com*, https://sites.google.com/site/murmurhash/, Accessed Sept. 18, 2018.

[Ap16] Appleby, A. (2016) “SMHasher”, *github.com*, https://github.com/aappleby/smhasher, Accessed Sept. 18, 2018.

[Be06] Bertoni, G., et al. (2006) “RadioGatún, a belt-and-mill hash function”, Presented at the Second Cryptographic Hash Workshop, Santa Barbara - August 24–25, 2006.

[Fo18] Fowler, G., et al. (2018) “The FNV Non-Cryptographic Hash Algorithm”, *IETF Internet-Draft.* Version 15, https://tools.ietf.org/html/draft-eastlake-fnv-15, Accessed Sept. 18, 2018.

[Fr84] Fredman, M. L., Komlós, J., and Szemerédi, E. (1984) “Storing a Sparse Table with 0(1) Worst Case Access Time”, *Journal of* *the ACM (JACM)*, Vol. 31 (3), pp. 538–544.

[Fu09] Fuhr, T., Peyrin, T. (2009) “Cryptoanalysis of RadioGatún”, In: Dunkelman O. (eds) Fast Software Encryption. *Lecture Notes in Computer Science*, Vol. 5665, Springer, Heidelberg

[He85] Heileman, G. L., Luo, W. (1985) “How Caching Affects Hashing”, *Proceedings of the 7th Workshop on Algorithm Enginnering and Experiments (ALENEX)*, pp. 141–154.

[Pa04] Pagh, R., Rodler, F. F. (2004) “Cuckoo hashing”, *Journal of Algorithms*, Vol. 51 (2), pp. 122–144.

[Pi14] Pike, G. (2014) “FarmHash, a family of hash functions”, *github.com*,
https://github.com/google/farmhash, Accessed Sept. 18, 2018.

[Pi11] Pike, G., Alakuijala, J. (2011) “CityHash, a family of hash functions for strings”, *github.com*,
https://github.com/google/cityhash, Accessed Sept. 18, 2018.

2

Membership

A *membership problem* for a dataset is a task to decide whether some element belongs to the dataset or not. For small sets, it could be solved by direct lookup and subsequent comparison of the given element to each element in the set. However, such a naive approach depends on the number of elements in the set and takes on average $\mathrm{O}(\log n)$ comparisons (on pre-sorted data), where n is the total number of elements. It is obvious that for huge sets of elements, which are operated by Big Data applications, this approach is not efficient and requires too much time and $\mathrm{O}(n)$ memory to store the elements.

Possible workaround solutions like chunking such sets and running comparisons in parallel can help in the reduction of computation time. However, it is not always applicable because for big data processing to store such huge sets of elements is almost an unachievable task.

On the other hand, in many cases, it isn't necessary to know exactly which element from the set has been matched, only that *a match has been made* and, therefore, it is possible to store only signatures of the elements rather than the whole value.

Example 2.1: Safe-browsing problem

Imagine, we develop a web-browser and notice that some URLs are known to contain malware, thus we want to alert users (or even prevent them from visiting) if they try to navigate to those pages. An immediate solution, that minimizes the network traffic, is to store all such URLs in the application and after the user enters the URL just check if it's not known as malware and can be safely navigated to.

Such a naive implementation will work quite well while the number of bad URLs is small. That is unfortunately not the case for real-world applications. After some time, we will need a special structure that can store bad URLs (or, ideally, only some information about them) without growing linearly in size when a new URL is introduced. Other requirements include that it should support the check of whether a URL is listed and it should be as fast as possible since we don't want users to wait for a long time.

Applications of the membership problem are not specific to pure computer science and play an essential role in various branches.

Example 2.2: DNA sequences (Stranneheim et al., 2010)

One important issue in metagenomic studies is the classification of sequences either as "novel" or belonging to a known genome, i.e., filtering out data that has been seen before.

A preprocessing step that executes membership queries, if performed efficiently, can reduce the complexity of the data before more careful analysis is performed.

The problem of fast lookup can be solved using hashing, which is also the simplest way to do that. With a hash function, every element of the dataset can be hashed into a hash table that maintains a (sorted) list of hash values. However, such an approach yields a small probability of errors (caused by possible hash collisions) and requires about $O(\log n)$ bits per each hashed element, which can still be unfeasible in practice for huge datasets.

In this chapter, we consider popular alternatives to regular hash tables that require less space, make faster lookups, and maintain smaller error probabilities. Such space-efficient data structures help to handle a big volume of data and allow for the execution of membership queries with good performance.

We start on the famous Bloom filter, then learn about its extensions and modifications, and finally, study its modern alternatives.

2.1 Bloom filter

The simplest and most well-known data structure that solves the membership problem is the *Bloom filter* which was proposed by Burton Howard Bloom in 1970. It is a space-efficient probabilistic data structure for representing a dataset $\mathbb{D} = \{x_1, x_2, \ldots, x_n\}$ of n elements that supports only two operations:

- *Adding* an element into the set, and
- *Testing* whether an element is or is not a member of the set.

The Bloom filter can store a large set very efficiently by discarding the identity of the elements; it stores only an (almost) unique set of bits corresponding to some number of hash functions that are applied to the element by the algorithm.

Practically, the Bloom filter is represented by a bit array and can be described by its length m and number of different hash functions $\{h_i\}_{i=1}^{k}$.

It is assumed that m is proportional to the number of expected elements n, while k is much smaller than m.

Hash functions h_i should be independent and uniformly distributed. In this way, we randomize the hash values uniformly (you can think of it as using hash functions as some kind of random-number generator) in the filter and decrease the probability of hash collisions.

Such an approach drastically reduces the storage space and, regardless of the number of elements in the data structure and their size, requires a constant number of bits by reserving a few bits per element.

The BloomFilter data structure is a bit array of length m where at the beginning all bits are equal to zero, meaning the filter is empty. To insert an element x into the filter, for every hash function h_k we compute its value $j = h_k(x)$ on the element x and set the corresponding bit j in the filter to one. Note, it is possible that some bits can be set multiple times due to hash collisions.

Algorithm 2.1: Adding element to the Bloom filter

```
Input: Element $x \in \mathbb{D}$
Input: Bloom filter with $k$ hash functions $\{h_i\}_{i=1}^{k}$
for $i \leftarrow 1$ to $k$ do
    $j \leftarrow h_i(x)$
    BloomFilter[$j$] $\leftarrow 1$
```

Example 2.3: Add elements to the filter

Consider a Bloom filter with length $m = 10$ and two 32-bit hash functions MurmurHash3 and FNV1a to produce values in the range $\{0, 1, \ldots, m-1\}$:

$$h_1(x) := \text{MurmurHash3}(x) \bmod m,$$
$$h_2(x) := \text{FNV1a}(x) \bmod m.$$

The empty filter has the following form:

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

As an example, we insert the names of capital cities into the filter. Let's start with ***Copenhagen*** and in order to find the corresponding bits in the filter we compute its hash values:

$$h_1(\boldsymbol{Copenhagen}) = \text{MurmurHash3}(\boldsymbol{Copenhagen}) \bmod 10 = 7,$$
$$h_2(\boldsymbol{Copenhagen}) = \text{FNV1a}(\boldsymbol{Copenhagen}) \bmod 10 = 3.$$

Hence, we need to set bits 3 and 7 in the filter:

0	1	2	3	4	5	6	7	8	9
0	0	0	**1**	0	0	0	**1**	0	0

It is possible that different elements can share corresponding bits, for instance let's add another element, ***Dublin***, to the filter:

$$h_1(\boldsymbol{Dublin}) = \text{MurmurHash3}(\boldsymbol{Dublin}) \bmod 10 = 1,$$
$$h_2(\boldsymbol{Dublin}) = \text{FNV1a}(\boldsymbol{Dublin}) \bmod 10 = 3.$$

As you can see, its corresponding bit-positions in the filter are 1 and 3, where only bit 1 hasn't been set yet (meaning that some element in the filter that is not ***Dublin*** has 3 as one of its corresponding bits):

0	1	2	3	4	5	6	7	8	9
0	**1**	0	1	0	0	0	1	0	0

When we need to test if the given element x is in the filter, we compute all k hash functions $h_i = \{h_i(x)\}_{i=1}^{k}$ and check bits in the corresponding positions. If *all* bits are set to one, then the element x **may exist** in the filter. Otherwise, the element x is **definitely not** in the filter. The uncertainty about the element's existence originates from the possibility of situations when some bits are set by different previously added elements (as we saw in Example 2.3) or, due to hard collisions, when all hash functions collide accidentally.

Algorithm 2.2: Testing element in the Bloom filter

```
Input: Element $x \in \mathbb{D}$
Input: Bloom filter with $k$ hash functions $\{h_i\}_{i=1}^{k}$
Output: False if element not found and True if element may exist
for $i \leftarrow 1$ to $k$ do
    $j \leftarrow h_i(x)$
    if BloomFilter$[j] \neq 1$ then
        return False
return True
```

Example 2.4: Test elements in the filter

Consider the Bloom filter from Example 2.3 with two indexed elements, ***Copenhagen*** and ***Dublin***:

0	1	2	3	4	5	6	7	8	9
0	1	0	1	0	0	0	1	0	0

To test if the element ***Copenhagen*** is in the filter, we again need to compute its hash values $h_1(\textbf{\textit{Copenhagen}}) = 7$ and $h_2(\textbf{\textit{Copenhagen}}) = 3$. After that, we check the corresponding bits in the filter and see that both of them are set to one, therefore we claim that ***Copenhagen*** **may exist** in the filter.

Now we will consider an element ***Rome*** and compute its hashes in order to find the corresponding bits in the filter:

$$h_1(\textbf{\textit{Rome}}) = \text{MurmurHash3}(\textbf{\textit{Rome}}) \bmod 10 = 5,$$
$$h_2(\textbf{\textit{Rome}}) = \text{FNV1a}(\textbf{\textit{Rome}}) \bmod 10 = 6.$$

Thus, checking the bits 5 and 6, we see that bit 5 isn't set, therefore the element ***Rome*** is **definitely not** in the filter and we don't even need to check bit 6.

However, the filter can also result in a false positive answer. Consider element ***Berlin***, whose hash values are

$$h_1(\textbf{\textit{Berlin}}) = \text{MurmurHash3}(\textbf{\textit{Berlin}}) \bmod 10 = 1,$$
$$h_2(\textbf{\textit{Berlin}}) = \text{FNV1a}(\textbf{\textit{Berlin}}) \bmod 10 = 7.$$

The corresponding bits 1 and 7 are both set in the filter, hence the result of the test function is that the element **may exist** in the filter. At the same time, we know that we have not added that element and this is an example of a hash collision. Note, in this particular case bit 1 was set by $h_1(\boldsymbol{Dublin})$ and bit 7 by $h_1(\boldsymbol{Copenhagen})$.

If each hash function $\{h_i\}_{i=1}^{k}$ can be computed in a constant time (which is true for all the most popular hash functions), the time to add a new element or test an element is a fixed constant $\mathrm{O}(k)$ and independent from the filter's length m and the number of elements in the filter.

The performance of the Bloom filter is highly dependent on the chosen hash functions. A hash function with a good uniformity will reduce the practically observed false positive rate. On the other hand, the faster the computation of each hash function, the smaller the overall time of each operation, and it is therefore recommended to avoid cryptographic hash functions.

Example 2.5: Prevent compromised passwords (Spafford, 1991)

Consider a web service registration page where we want to prevent users from choosing weak and compromised passwords. Note, that in the Dark Web it is possible to find hundreds of millions[1] hacked passwords that can be used in a dictionary attack, a brute force attack that makes repetitive attempts to defeat an authentication by trying all values from a prearranged listing. Thus; every time the user types a new password, we would like to ensure it is not in such a list. However, along with the lack of security related to storing raw passwords, we don't want to maintain a huge dataset that grows linearly with every newly added password which will slow down our lookups (as in traditional databases).

Therefore, the usage of a space-efficient Bloom filter is essential. The false positive event, in this case, is a situation where we mistakenly think that the entered password is unsuitable. In such rare cases, we need to ask the user to type another password and that usually doesn't hurt.

[1] 1.4 Billion Clear Text Credentials Discovered `https://medium.com/4iqdelvedeep/3131d0a1ae14`

Count unique elements in the filter

A method to estimate the number of unique elements indexed into the filter was proposed by S. Joshua Swamidass and Pierre Baldi and, in fact, is an extension of the Linear Counting algorithm that is discussed in the next chapter. Using the information about the number of set bits in the filter and the probability of each bit to be set, it provides a simple formula to approximate the number of elements in the filter. Since two identical elements added into the filter won't change the number of set bits, such approximation gives an estimation for the number of unique elements (known as *cardinality*).

Algorithm 2.3: Counting unique elements in the Bloom filter

```
Input: Bloom filter of length $m$ with $k$ hash functions
Output: Number of unique elements in the filter
$\mathrm{N} \leftarrow \underset{j=1 \ldots m}{\mathtt{count}}(\textsc{BloomFilter}[j] = 1)$
if $\mathrm{N} < k$ then
    return $0$
if $\mathrm{N} = k$ then
    return $1$
if $\mathrm{N} = m$ then
    return $\frac{m}{k}$
return $-\frac{m}{k} \cdot \ln\left(1 - \frac{\mathrm{N}}{m}\right)$
```

Properties

False positives are possible. As has already been mentioned, the Bloom filter doesn't store elements and hardly relies on the calculated hashes which are stored all together in a bit array. Such space-efficient representation can lead to situations where some element is not a member (wasn't added to the filter), but the algorithm returns like it is. Such an event is called a false positive and can occur because of hash collisions or due to mess in stored bits — in the test operation there is no prior knowledge of whether the particular bit has been set by the same hash function as the one we compare with.

The Bloom filter principle [Br04]: *Wherever a list or set is used, and space is at a premium, consider using a Bloom filter if the effect of false positives can be mitigated.*

Fortunately, such false positive situations rarely happen and their probability $\mathrm{P_{fp}}$ can be easily estimated (actually, this is a lower bound):

$$\mathrm{P_{fp}} \approx (1 - e^{-\frac{kn}{m}})^k. \tag{2.1}$$

As we can see from (2.1), under the fixed number of expected elements n, the probability of false positives depends on the choice of k and m. It is a clear trade-off between the length of the filter, the number of hash functions, and the probability of such events.

In the extreme case, when the filter is full (meaning all bits are set), every lookup will yield a (false) positive response. It means that the choice of m depends on the (estimated) number of elements n that are expected to be added, and m should be quite large compared to n.

In practice, the length of the filter m, under given false positive probability $\mathrm{P_{fp}}$ and the expected number of elements n, can be determined by the formula:

$$m = -\frac{n \ln \mathrm{P_{fp}}}{(\ln 2)^2}. \tag{2.2}$$

Thus, a filter must grow linearly with the number of elements to keep the fixed false positive probability.

For the given ratio of $\frac{m}{n}$, meaning the number of allocated bits per element, the false positive probability can be tuned by choosing the number of hash functions k. The optimal choice of k is computed by minimizing the probability of false positives in (2.1):

$$k = \frac{m}{n} \ln 2. \tag{2.3}$$

In other words, the optimal number of hash functions k is about 0.7 times the number of bits per element. Since k must be an integer, the smaller sub-optimal values are preferred.

Some widely used near-optimal solutions can be found below.

Table 2.1: Near-optimal choices of parameters

k	$\frac{m}{n}$	$\mathrm{P_{fp}}$
4	6	0.0561
6	8	0.0215
8	12	0.00314
11	16	0.000458

Example 2.6: Parameters estimation

According to (2.2), to support the false positive probability $\mathrm{P_{fp}} = 1\%$ the filter has to be 10 times longer than the expected number of elements n and use 6 hash functions. However, the length of the filter doesn't depend on the size of elements themselves and stays the same for elements of different natures.

Bloom filters can be seen as a generalization of hash tables. In fact, a filter with only one hash function is equivalent to a hash table. However, using many hash functions, Bloom filters can maintain the constant false positive probability even for a fixed number of bits per element, while hash tables cannot.

False negatives are not possible. In contrast to the situation above, if the Bloom filter returns that a particular element isn't a member, then it's definitely not a member of the set:

$$\mathrm{P_{fn}} = 0. \tag{2.4}$$

Works well while it fits in memory. As already mentioned above, the probability of false positives can be decreased by allocating more memory, this is why people tend to create bigger filters (with larger m).

However, such classical Bloom filters work well while they fit in the main memory. As soon as they grow too big and have to be moved on disk, they immediately hit the problem induced by the design — uniformly distributed hash functions produce random corresponding indices which need to be randomly accessed every time — very unpleasant for disks with rotating platters and moving heads (for solid-state storage devices it is much better, but still not perfect).

Example 2.7: Required memory

According to (2.2), to handle 1 billion elements and keep the probability of false positive events at about 2% using the optimal number of hash functions, we need to choose a filter of $m = -10^9 \cdot \ln(0.02)/(\ln 2)^2 \approx 8.14 \cdot 10^9$ bits long that is roughly 1 GB of memory.

Two different Bloom filters of the same length can be merged only if they also have the same hash functions. In this case, the merge is just a bitwise OR operation and the result is a full equivalent to the Bloom filter built for the union of those two sets of elements. The intersection of those two Bloom filters is also possible and can be done by bitwise XOR, however, the result can have a higher probability of collisions.

Unfortunately, it is not possible to adjust the size of a Bloom filter when it runs out of space without recomputing all the hashes that are already in the filter, that is unlikely to be feasible in Big Data applications.

Example 2.8: Cache sharing (Fan, 2000)

Consider a list of distributed caching proxies $\mathrm{P}_1, \mathrm{P}_2, \ldots, \mathrm{P}_n$ in a network that shares their caches. If the content of the requested URL is stored on a proxy P_i, then the proxy returns it without an actual call to the remote server, otherwise, the content will be retrieved, stored locally, and sent to the client.

With the goal of minimizing the network traffic and distributing the storage, we can set up a routing in that proxy network and attempt forward to a proxy that already has the content stored, if any, otherwise, call the remote server. Since requests from the client can come to any proxy, there is a problem sharing the routing list on each proxy and when it's changed exchanging it within the network or merging, if necessary.

The Bloom filter is a natural choice to store such routing lists and perform fast membership queries. It is also easily transferable in the network due to its small size.

The false positive event, in this case, is that some proxy P_i assumes that another proxy P_j may have the content for the requested URL, but in fact, it doesn't. P_i routes traffic to P_j and asks it to return the content, therefore P_j has to call the remote server. As a result, it produces some additional network traffic and stores redundant local copies for such content, which is completely acceptable.

Deletion is not possible. To delete a particular element from the Bloom filter it would need to unset its corresponding k bits in the bit array. Unfortunately, a single bit could correspond to multiple elements due to hash collisions and shared bits between elements.

A number of extensions have been developed that support deletion of elements, but they always cost through space and speed. This is why the classical Bloom filter is so fast and space-efficient.

Fortunately, missing deletion support is not a problem for many real-world applications, but if you really need it you have to go for modifications of the Bloom filter, for example the Counting Bloom filter.

2.2 Counting Bloom filter

The most popular extension of the classical Bloom filter that supports deletion is the *Counting Bloom filter*, proposed by Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder in 2000 [Fa00]. Building on the classical Bloom filter algorithm, it introduces an array of m counters $\{\mathrm{C}_j\}_{j=1}^{m}$ corresponding to each bit in the filter's array.

The Counting Bloom filter allows approximating the number of times each element has been seen in the filter by incrementing the corresponding counter every time the element is added. The associated CountingBloomFilter data structure contains a bit array and the array of counters of length m, all initialized to zeros.

When we insert a new element into CountingBloomFilter, we first compute its corresponding bit-positions, then for each position we increment the associated counter and, only if it changes from zero to one, set the bit in the filter, similar to the step of the classical Algorithm 2.1.

Algorithm 2.4: Adding element to the Counting Bloom filter

```
Input: Element $x \in \mathbb{D}$
Input: Counting Bloom filter with $m$ counters $\{C_j\}_{j=1}^{m}$ and $k$ hash
       functions $\{h_i\}_{i=1}^{k}$
for $i \leftarrow 1$ to $k$ do
    $j \leftarrow h_i(x)$
    $C_j \leftarrow C_j + 1$
    if $C_j = 1$ then
        CountingBloomFilter[$j$] $\leftarrow 1$
```

The test operation looks exactly the same as for the classical Bloom filter Algorithm 2.2 since we don't need to check counters at all. The amount of time required to test an element is comparable to the classical algorithm because the filters' bit arrays are the same.

Algorithm 2.5: Testing element in the Counting Bloom filter

```
Input: Element $x \in \mathbb{D}$
Input: Counting Bloom filter with $k$ hash functions $\{h_i\}_{i=1}^{k}$
Output: False if element not found and True if element may exist
for $i \leftarrow 1$ to $k$ do
    $j \leftarrow h_i(x)$
    if CountingBloomFilter[$j$] $\neq 1$ then
        return False
return True
```

The deletion is quite similar to the insertion but in reverse. To delete an element x, we compute all k hash values $h_i = \{h_i(x)\}_{i=1}^{k}$ and decrease the corresponding counters. If the counter changes its value from one to zero, the corresponding bit in the bit-array has to be unset.

Algorithm 2.6: Deleting element from the Counting Bloom filter

```
Input: Element $x \in \mathbb{D}$
Input: Counting Bloom filter with $m$ counters $\{C_j\}_{j=1}^{m}$ and $k$ hash
       functions $\{h_i\}_{i=1}^{k}$
for $i \leftarrow 1$ to $k$ do
    $j \leftarrow h_i(x)$
    $C_j \leftarrow C_j - 1$
    if $C_j = 0$ then
        $\textsc{CountingBloomFilter}[j] \leftarrow 0$
```

Algorithm 2.6 presupposes that element x exists (or may exist) in the filter, therefore it might be necessary to test elements before decreasing the corresponding counters.

Properties

The Counting Bloom filter inherits all the properties of the classical Bloom filter, including false positive error estimation and recommendations for the optimal choice of m and k given by the dependencies (2.2) and (2.3).

Naturally, Counting Bloom filters are much bigger than classical Bloom filters because additional memory has to be allocated for the counters even if most of them are zeros. Therefore, it is important to estimate how large such counters can become and how their size depends on the filter's length m and the number of hash functions k.

Assuming that every counter C has a capacity level N, the probability that the value goes over that capacity level (known as *overflow probability*) in a Counting Bloom filter of length m with the optimal choice of k from the relation (2.3) is

$$\Pr\Big(\max(\mathrm{C}) \geq \mathrm{N}\Big) \leq m \cdot \left(\frac{e \ln 2}{\mathrm{N}}\right)^{\mathrm{N}}. \tag{2.5}$$

For instance, for 4-bit counters ($\mathrm{N} = 16$) the overflow probability given by formula (2.5) becomes

$$\Pr\Big(\max(\mathrm{C}) \geq 16\Big) \leq m \cdot 1.37 \cdot 10^{-15}.$$

In other words, if we allocate 4 bits per counter, the probability of overflow for practical values of m (e.g., a few billion bit-positions) during the initial insertion into the filter is extremely small. After many deletions and insertions, the probability can become a bit bigger, but is still small enough for the practical usage.

To prevent arithmetic overflow (i.e., incrementing a counter that already has the maximum possible value), each counter in the array must be sufficiently large in order to retain the properties of the Bloom filters. In practice, the counter consists of 4 or more bits and a Counting Bloom filter, therefore, requires four times more space than a classical Bloom filter.

It depends on the application, but if a 4-bit counter ever exceeds the value of 15 we can simply "freeze" it and let it stay at 15. After many deletions, this might lead to a situation where the Counting Bloom filter produces a false negative response (the counter becomes zero when it shouldn't be), but the probability of such a chain of events is so low that it is much more likely that our application will be rebooted and the filter re-created.

However, it is possible to design a more complex version of the Counting Bloom filter with smaller counters (e.g., 2-bit) that uses less space and is still practically useful by adopting an approach similar to closed addressing in hash tables and introducing a secondary hash table to manage overflowing counters.

Therefore, the Counting Bloom filter supports only *probabilistically correct deletions* because there is a non-zero probability of error as soon as some counter goes above its maximal size.

Despite the noted peculiarities, Counting Bloom filters are extensively used by Apache Hadoop and Apache Spark in MapReduce applications to speed up the processing of huge datasets on big clusters by helping to reduce their volume.

2.3 Quotient filter

When the classical Bloom filter and its modifications do not fit into the main memory they are entirely unfriendly to storage due to the requirements of multiple random accesses for any operation. One of the data structures that supports the basic operations of Bloom filters, but with better data locality and requiring only a small number of contiguous disk accesses, is the *Quotient filter*, presented by Michael Bender et al. in 2011 [Be11].

The filter achieves comparable performance regarding space and time, but additionally supports deletions and can be dynamically resized or merged. The name of this data structure originates from an arithmetic quotient which is a result of the division operation.

The Quotient filter represents a dataset $\mathbb{D} = \{x_1, x_2, \ldots, x_n\}$ by storing a p-bit fingerprint for each of them and requires only one hash function to generate such fingerprints. In order to support enough randomness, the hash function should generate uniformly and independently distributed fingerprints.

Each fingerprint f in the algorithm is partitioned into q most significant bits (the *quotient*) and r least significant bits (the *remainder*) using the quotienting technique, suggested by Donald Knuth[2].

[2]D. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, 1973

Algorithm 2.7: Quotienting technique

```
Input: Fingerprint $f$
Output: Quotient $f_q$ and remainder $f_r$
$f_r \leftarrow f \bmod 2^r$
$f_q \leftarrow \left\lfloor \frac{f}{2^r} \right\rfloor$
return $f_q$, $f_r$
```

Practically, to improve spatial locality, the QUOTIENTFILTER data structure is represented by a compact open addressing hash table with $m = 2^q$ buckets where the remainder f_r is stored in a bucket indexed by the quotient f_q. Possible collisions are resolved by Linear probing.

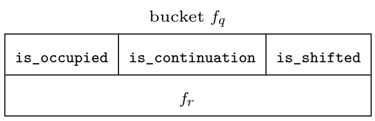
Given a remainder f_r in bucket f_q, the full fingerprint can be uniquely reconstructed as

$$f = f_q \cdot 2^r + f_r.$$

Each bucket contains three metadata bits, all unset at the beginning: `is_occupied`, `is_continuation`, and `is_shifted` — these play an important role in navigating the data structure.

- `is_occupied` is set when the bucket j is the canonical bucket $(f_q = j)$ for some fingerprint f stored somewhere in the filter.
- `is_continuation` is set when the bucket is occupied, but not by the first of the remainders that belong to the same bucket.
- `is_shifted` is set when the remainder in the bucket is not in its canonical bucket.

Figure 2.1: Bucket in the Quotient filter



When two different fingerprints f and f' have the same quotient (meaning $f_q = f'_q$), it is a soft collision that can be resolved by the Linear probing strategy that we discussed early. In Quotient filter it is implemented by storing all the remainders of fingerprints with the same quotient contiguously in a *run*. If necessary, a remainder can be shifted forward from its original location and stored in a subsequent bucket, wrapping around at the end of the array.

Algorithm 2.8: Using right shift to empty buckets

```
Input: Bucket index $k$
Input: Quotient filter of length $m$
$prev \leftarrow \textsc{QuotientFilter}[k]$
$i \leftarrow k + 1$
while True do
    if $\mathrm{QF}[i] = \mathrm{NULL}$ then
        $\mathrm{QF}[i] \leftarrow prev[i]$
        $\mathrm{QF}[i]$.is_continuation $\leftarrow 1$
        $\mathrm{QF}[i]$.is_shifted $\leftarrow 1$
        return QF
    else
        $curr \leftarrow \mathrm{QF}[i]$
        $\mathrm{QF}[i] \leftarrow prev$
        $\mathrm{QF}[i]$.is_continuation $\leftarrow prev$.is_continuation
        $\mathrm{QF}[i]$.is_shifted $\leftarrow prev$.is_shifted
        $prev \leftarrow curr$
        $prev$.is_continuation $\leftarrow curr$.is_continuation
        $prev$.is_shifted $\leftarrow curr$.is_shifted
    $i \leftarrow i + 1$
    if $i > m$ then
        $i \leftarrow 0$
```

The sequence of one or more consecutive runs with no empty buckets in between is called a *cluster*. All clusters are immediately preceded by an empty bucket and the `is_shifted` bit of its first value is never set.

The internal hash table is compactly stored in an array to reduce the required memory and achieve better data locality; however, this makes the navigation through it quite complex.

Consider a scan function that is designed to find the run. It starts by walking backward from the canonical bucket for f to find the beginning of the cluster. As soon as the cluster's start is found, it goes forward again to find the location of the first remainder for the bucket f_q, that is the actual start of the run r_{start}.

Algorithm 2.9: Scanning the Quotient filter to find the run

```
Input: Canonical bucket index $f_q$, Quotient filter
$j \leftarrow f_q$
while QF[$j$].is_shifted $= 1$ do
    $j \leftarrow j - 1$
$r_{\text{start}} \leftarrow j$
while $j \neq f_q$ do
    /* skip all elements in the current run and find the next occupied bucket */
    repeat
        $r_{\text{start}} \leftarrow r_{\text{start}} + 1$
    until QF[$r_{\text{start}}$].is_continuation $\neq 1$
    repeat
        $j \leftarrow j + 1$
    until QF[$j$].is_occupied $= 1$
$r_{\text{end}} \leftarrow r_{\text{start}}$
repeat
    $r_{\text{end}} \leftarrow r_{\text{end}} + 1$
until QF[$r_{\text{end}}$].is_continuation $\neq 1$
return $r_{\text{start}}$, $r_{\text{end}}$
```

When we want to insert a new element into QuotientFilter, we first calculate its quotient and remainder. If the canonical bucket for the element isn't occupied, it can immediately be inserted using the insertion procedure given by Algorithm 2.10. Otherwise, before insertion, it is necessary to find an appropriate bucket with the scan function from Algorithm 2.9. Once the correct bucket is found, actual insertion still requires the appropriate merging of the f_r to the sequence of already stored elements, that may need shifting right of subsequent elements and updating the corresponding metadata bits respectively.

With the mentioned selection strategy for the appropriate bucket and the right shift function given by Algorithm 2.8, we can formulate the complete insertion procedure below.

Algorithm 2.10: Adding element to the Quotient filter

```
Input: Element $x \in \mathbb{D}$
Input: Quotient filter with hash function $h$
$f \leftarrow h(x)$
$f_q, f_r \leftarrow f$
if $\mathrm{QF}[f_q]$.is_occupied $\neq 1$ and $\mathrm{QF}[f_q]$ is empty then
    $\mathrm{QF}[f_q] \leftarrow f_r$
    $\mathrm{QF}[f_q]$.is_occupied $\leftarrow 1$
    return True
$\mathrm{QF}[f_q]$.is_occupied $\leftarrow 1$
$r_{\text{start}}, r_{\text{end}} \leftarrow$ **Scan**$(\mathrm{QF}, f_q)$
for $i \leftarrow r_{\text{start}}$ to $r_{\text{end}}$ do
    if $\mathrm{QF}[i] = f_r$ then
        /* $f_r$ already exists */
        return True
    else if $\mathrm{QF}[i] > f_r$ then
        /* insert $f_r$ in the bucket $i$ and shift others */
        $\mathrm{QF} \leftarrow$ **ShiftRight**$(\mathrm{QF}, i)$
        $\mathrm{QF}[i] \leftarrow f_r$
        return True
/* the run should be extended with the new element */
$\mathrm{QF} \leftarrow$ **ShiftRight**$(\mathrm{QF}, r_{\text{end}} + 1)$
$\mathrm{QF}[r_{\text{end}} + 1] \leftarrow f_r$
return True
```

According to the Linear probing schema, the length of most runs is O(1) and, the authors of the filter noted, it is highly likely that all runs have length $O(\log m)$.

Example 2.9: Add elements to the filter

Consider a Quotient filter with 16-bit fingerprints produced by the 32-bit version of the MurmurHash3 hash function:

$$h(x) := \text{MurmurHash3}(x) \bmod 16.$$

For the bucketing we reserve $q = 3$ most significant bits, hence the size of the QuotientFilter is $m = 2^3 = 8$, and the rest $p = 13$ bits we store into the chosen buckets.

Like in Example 2.3 we start indexing names of capitals and the first element to add into the filter is ***Copenhagen***. We need to compute its fingerprint using the hash function h:

$$f = h(\boldsymbol{Copenhagen}) = 4248224207.$$

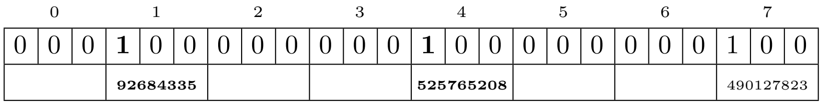
According to Algorithm 2.7, the quotient and remainder are

$$f_q = \left\lfloor \frac{f}{2^{13}} \right\rfloor = 7,$$
$$f_r = f \bmod 2^{13} = 490127823.$$

The canonical bucket for the element ***Copenhagen*** is $j = f_q = 7$ where we want to index its remainder f_r. Insertion at this point is straightforward since all buckets are not occupied and we insert $f_r = 490127823$ in the bucket with index 7 and set the `is_occupied` bit:

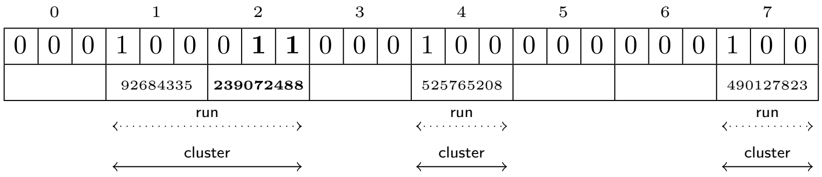
	0			1			2			3			4			5			6			7	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0
																					490127823		

In the same way, we index the elements ***Lisbon***, that has the fingerprint $f = 629555247$ and the canonical bucket 1, and ***Paris*** with the fingerprint $f = 2673248856$ and the canonical bucket 4. Since those canonical buckets are free, we insert remainders accordingly and set the `is_occupied` bits:

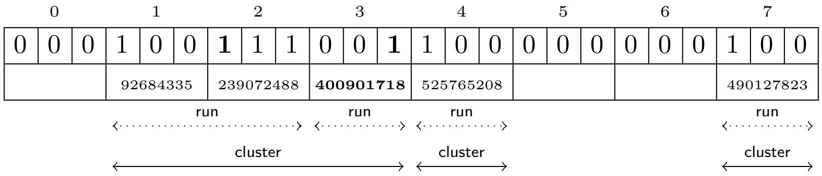


Next, we add element ***Stockholm*** with fingerprint $f = 775943400$, getting its canonical bucket $j = f_q = 1$ and the remainder $f_r = 239072488$. However, the canonical bucket 1 already has its `is_occupied` bit set, meaning that it is already occupied by the remainder of another element (element ***Lisbon*** in this case).

Since the `is_shifted` and `is_continuation` bits are not set, we are at the beginning of the cluster that is also the start of the run. The remainder f_q is bigger than the already indexed value 92684335, therefore it should be stored into the next available bucket, being bucket 2 and those `is_shifted` and `is_continuation` bits have to be set. However, the `is_occupied` bit for bucket 2 remains unchanged, because there is no stored remainder that has it as a canonical bucket.



The next element is ***Zagreb*** whose fingerprint $f = 1474643542$, canonical bucket $j = 2$ and remainder $f_r = 400901718$. Unfortunately, bucket 2 is already in use by some shifted value as is shown by the set `is_shifted` bit, however the `is_occupied` bit is not set. Thus, the value f_r has to be shifted right as well, into the next available bucket, which is bucket 3 in this case.

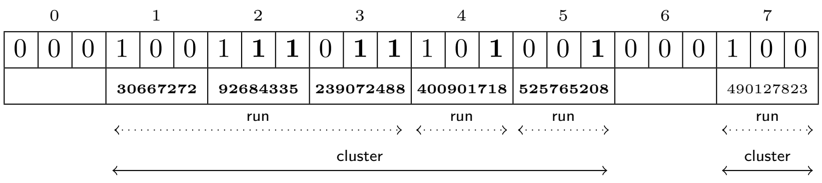


We set the `is_shifted` bit to indicate that the bucket contains a value shifted from its canonical position, but keep the `is_continuation` bit unset since it is the first element associated with that canonical bucket. Additionally, we set the `is_occupied` bit for bucket 2 to remember that there is at least one stored remainder that has it as its canonical bucket.

Finally, let's add element ***Warsaw*** with fingerprint $f = 567538184$, whose quotient and remainder are

$$f_q = \left\lfloor \frac{f}{2^{13}} \right\rfloor = 1,$$
$$f_r = f \bmod 2^{13} = 30667272.$$

The canonical bucket $j = f_q = 1$ is already occupied according to the set `is_occupied` bit. However, other bits are not set, meaning that we are at the beginning of the cluster, that is also the start of the run. The remainder f_q is smaller than the indexed value 92684335; thus it should be indexed into the canonical bucket, and all other remainders have to be shifted and marked as a continuation. In this case, the shifting also affects the remainders from other runs, forcing us to shift them as well, set shifted bits and mirror the continuation bits if they were set for their current positions.



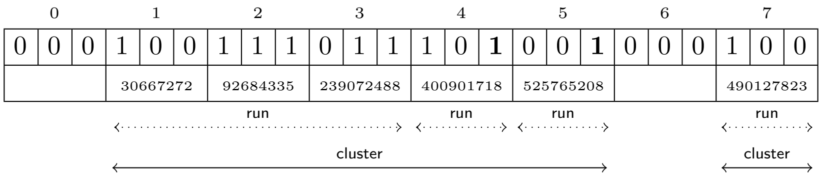
Testing for elements can be completed in the same way as insertion. We check if the canonical bucket for the tested element has at least one associated remainder somewhere in the filter by observing the `is_occupied` bit. If the bit is not set, we can conclude that element is **definitely not** in the filter. Otherwise, we scan the filter using the scan procedure given by Algorithm 2.9 to find the appropriate run for the bucket. Next, within that run, we compare stored remainders with the remainder of the tested element taking into account that they are all sorted. If such a remainder is found, we can report that the element **may exist** in the filter.

Algorithm 2.11: Testing element in the Quotient filter

```
Input: Element $x \in \mathbb{D}$
Input: Quotient filter with hash function $h$
Output: False if element not found and True if element may exist
$f \leftarrow h(x)$
$f_q, f_r \leftarrow f$
if $\mathrm{QF}[f_q]$.is_occupied $\neq 1$ then
    return False
else
    $r_{\mathrm{start}}, r_{\mathrm{end}} \leftarrow$ Scan$(\mathrm{QF}, f_q)$
    /* search for $f_r$ within the run */
    for $i \leftarrow r_{\mathrm{start}}$ to $r_{\mathrm{end}}$ do
        if $\mathrm{QF}[i] = f_r$ then
            return True
    return False
```

Example 2.10: Test elements in the filter

Consider the QuotientFilter data structure that we built in Example 2.9:



Let's test the element ***Paris***, with quotient $f_q = 4$ and remainder $f_q = 525765208$ as calculated earlier. Bucket 4 is already occupied, meaning there is at least one remainder somewhere in the filter that has it as the canonical bucket. We cannot at this point compare the value from the bucket with the f_r because the `is_shifted` bit is set and we need to find a run that corresponds to canonical bucket 4 in the current cluster.

Thus, we scan from bucket 4 to the left and count buckets with set `is_occupied` bits until we reach the start of the cluster. In our example, the cluster starts at bucket 1, and there are two occupied buckets (buckets 1 and 2) located left of bucket 4. Therefore, our run is the third in the cluster and we need to scan right from the beginning of the cluster (bucket 1) until we reach that run by counting buckets with unset `is_continuation` bits. Finally, we find that the run starts within bucket 5, and start comparing stored remainders, taking into account that they are sorted in ascending order.

The value in bucket 5 exactly matches the remainder $f_q = 525765208$, thus we can conclude that element ***Paris*** may exist in the filter.

Deletions in a Quotient filter are handled in a very similar way to the addition of a new element. However, since all remainders of fingerprints with the same quotient are stored contiguously according to their numerical order, removal of a remainder from the cluster must shift all fingerprints to fill the “empty” entry after deletion and modify the metadata bits respectively.

Algorithm 2.12: Using left shift to fill empty buckets

```
Input: Bucket index $k$
Input: Quotient filter of length $m$
$i \leftarrow k + 1$
while $\mathrm{QF}[i] \neq \mathrm{NULL}$ do
    $\mathrm{QF}[i-1] \leftarrow \mathrm{QF}[i]$
    $\mathrm{QF}[i-1]$.is_continuation $\leftarrow \mathrm{QF}[i]$.is_continuation
    $\mathrm{QF}[i-1]$.is_shifted $\leftarrow \mathrm{QF}[i]$.is_shifted
    $\mathrm{QF}[i] \leftarrow \mathrm{NULL}$
    $\mathrm{QF}[i]$.is_continuation $\leftarrow 0$
    $\mathrm{QF}[i]$.is_shifted $\leftarrow 0$
    $i \leftarrow i + 1$
    if $i > m$ then
        $i \leftarrow 0$
```

Firstly, it is necessary to check if the canonical bucket is already occupied, otherwise the element is definitely not in the filter and we can stop here. Afterward we use the scan procedure to find the appropriate bucket and delete the requested element (if it exists) and shift the subsequent elements and update the corresponding metadata bits. Note, that if the deleted remainder was the last for its canonical bucket, we also unset the `is_occupied` bit.

Algorithm 2.13: Deleting element from the Quotient filter

```
Input: Element $x \in \mathbb{D}$
Input: Quotient filter with hash function $h$
Output: False if element not found and True otherwise
$f \leftarrow h(x)$
$f_q, f_r \leftarrow f$
if $\mathrm{QF}[f_q]$.is_occupied $\neq 1$ then
    return True
$r_{\text{start}}, r_{\text{end}} \leftarrow$ Scan$(\mathrm{QF}, f_q)$
for $i \leftarrow r_{\text{start}}$ to $r_{\text{end}}$ do
    if $\mathrm{QF}[i] = f_r$ then
        /* element found and can be deleted */
        $\mathrm{QF}[i] \leftarrow \mathrm{NULL}$
        if $r_{\text{start}} = r_{\text{end}}$ then
            $\mathrm{QF}[i]$.is_occupied $\leftarrow 0$
        else if $i < r_{\text{end}}$ then
            $\mathrm{QF} \leftarrow$ ShiftLeft$(\mathrm{QF}, i + 1)$
        return True
return False
```

Properties

False positives are possible. The Quotient filter data structure is a compact representation of a multi-set of fingerprints, and its false positive rate is a function of the hash function h and the number of elements n added into the filter.

Moreover, two different elements could have the same values for both the remainder and quotient, that is called a *hard collision*. Due to such extremely rare events, it is possible that false positive responses can occur and their probability P_{fp} is upper bounded by

$$\mathrm{P}_{\mathrm{fp}} \approx 1 - e^{-\frac{n}{2^p}} \leq \frac{n}{2^p}. \tag{2.6}$$

The formula (2.6) shows that, under the fixed number of expected elements n, there is a trade-off between the probability of false positives P_{fp} and the length of the fingerprint p.

Practical implementations of Quotient filters use 32- and 64-bit fingerprints.

Similar to other hash tables, the load factor is very important in the Quotient filter and we want to allocate at least as many buckets as we expect elements, meaning we choose the number of buckets m as

$$m := 2^q > n, \tag{2.7}$$

and the length of the remainder r can be calculated from (2.6) as

$$r = \left\lceil \log\left(-\frac{n}{2^q} \cdot \frac{1}{\ln\left(1 - \mathrm{P}_{\mathrm{fp}}\right)}\right) \right\rceil. \tag{2.8}$$

False negatives are not possible. Same as with other data structures in this chapter, if the Quotient filter finds an element is not a member, then it is definitely not a member of the set:

$$\mathrm{P}_{\mathrm{fn}} = 0. \tag{2.9}$$

The Quotient filter is about 20% bigger than the Bloom filter, but faster because each access requires evaluating only a single hash function and all data are stored in contiguous blocks. Tests in a Quotient filter incur a single cache miss, as opposed to at least two in expectation for the Bloom filter algorithm.

Results of in-RAM performance comparisons in [Be12] show that Quotient filters can handle 2.4 million inserts per second while Bloom filters are limited to about 0.69 million. However, with tests for elements, they are almost at the same level of about 2 million per second.

Example 2.11: Required memory

As stated in (2.7), to handle 1 billion elements, the Quotient filter has to contain at least 2^{30} buckets, meaning we cannot use fingerprints shorter than 31 bits. If we want to keep the probability of false positive events at about 2%, the number of bits for the remainder can be found from the formula (2.8) as

$$r = \left\lceil \log\left(-\frac{10^9}{2^30} \cdot \frac{1}{\ln(1-0.02)} \right) \right\rceil = 6.$$

Therefore, the required length of the fingerprint is $p = q + r = 36$ bits, where the first 30 bits are used for the bucketing and the rest 6 bits are stored in the appropriate bucket. Since every bucket additionally contains three metadata bits, the total size of the Quotient filter is $9 \cdot 2^{30}$ bits that is roughly 1.2 GB of memory.

The Quotient filter can restore fingerprints from the stored data, therefore, support deletion, merging, and resizing. The merge doesn't affect the false positive rates of the filters and deletion in a Quotient filter is always correct in contrast to the Counting Bloom filter that supports only probabilistically correct deletions.

Resizing of the Quotient filter (both, shrunk and expanded) can be done by iterating over the filter and copying each fingerprint into a newly allocated data structure without the need to re-hash. Two or more Quotient filters can be merged using an algorithm similar to the merge sort, the divide-and-conquer sorting algorithm invented by John von Neumann. Thus, all input filters can be scanned in parallel and the merged result is written to the output filter.

The time required to perform a test, addition or deletion in a Quotient filter is dominated by the time to scan backward and forward.

However, the Quotient filter is designed with a focus on big data (e.g., 1 billion elements for 64-bit hash function) and for small- or medium-sized datasets its complexity can diminish the benefits.

2.4 Cuckoo filter

Most modifications of the classical Bloom filter that support deletions degrade either in space or performance. In order to handle this problem Bin Fan, David Anderson, Michael Kaminsky, and Michael Mitzenmacher in 2014 [Fa14] proposed the *Cuckoo filter* which was a compact variant of the Cuckoo hash table that we discussed earlier, but adjusted to store only fingerprints of some length p for each inserted element, instead of key-value pairs.

Cuckoo filters are easier to implement, they support dynamic additions and deletions, while using less space and achieving even higher performance than other Bloom filter modifications in many practical applications.

The CuckooFilter data structure is represented as a multi-way associative Cuckoo hash table with m buckets each of which can store up to b values. However, with the standard Cuckoo hashing, to insert a new element, it is necessary to access the original existing elements in order to determine where to relocate stored values if space is needed for new ones. However, the Cuckoo filter only stores fingerprints and there is no way to restore the original elements and re-hash them to find their new bucket in the hash table.

With the purpose of overcoming this limitation and still employing the Cuckoo hashing, the Cuckoo filter algorithm uses the *Partial–Key Cuckoo hashing*, which allows the new bucket of the existing element to be derived by its fingerprint without knowing the original element itself.

According to that schema, for each element x to be inserted, the algorithm computes a p-bit fingerprint f and the indexes of two candidate buckets as follows:

$$\begin{aligned} i &= h(x) \bmod m, \\ j &= (i \oplus (h(f) \bmod m)) \bmod m. \end{aligned} \tag{2.10}$$

In order to distribute elements uniformly in the hash table, the fingerprint f is additionally hashed with a hash function h before the XOR calculation in formula (2.10).

When the fingerprints length p is small compared to the filter length m, the XOR operation alters only that small number of lower bits, but most of the higher order bits stay the same. This simply implies that elements shifted from their primary buckets tend to be found close to each other in their alternate buckets and distribution in the hash table is going to be skewed which influences the efficiency of the filter.

Hashing the fingerprints ensures that these elements are relocated to buckets in entirely different parts of the hash table, hence reducing hash collisions and improving the table utilization.

The exclusive disjunction (XOR) operation $\oplus$ in the formula (2.10) ensures an important property, that by knowing the current element's buckets k it is possible to compute its alternate bucket k^* without restoring the original element:

$$k^* = (k \oplus h(f)) \bmod m. \tag{2.11}$$

To add a new element x into the Cuckoo filter, we compute indices for two candidate buckets with (2.10). If at least one of those buckets is empty, we insert the element into that bucket. Otherwise, we randomly choose one of those buckets and store element x there, while moving the element from that bucket to its alternative candidate bucket using (2.11). We repeat this procedure until an empty bucket is found, or until a maximum number of displacements is reached. If there are no empty buckets, the filter is considered full.

Algorithm 2.14: Adding element to the Cuckoo filter

```
Input: Element $x \in \mathbb{D}$
Input: Cuckoo filter with fingerprinting and hash function $h$
Output: True if element has been added and False otherwise
$f \leftarrow$ fingerprint$(x)$
$i \leftarrow h(x)$
$j \leftarrow i \oplus h(f)$
if CuckooFilter$[i]$ has empty space then
    CuckooFilter$[i]$.add$(f)$
    return True
else if CuckooFilter$[j]$ has empty space then
    CuckooFilter$[j]$.add$(f)$
    return True
$k \leftarrow$ sample$(\{i, j\})$
for $n \leftarrow 0$ to MaxIter do
    $x \leftarrow$ sample(CuckooFilter$[k]$)
    swap $f$ and the fingerprint stored in entry $x$
    $k = k \oplus h(f)$
    if CuckooFilter$[k]$ has empty space then
        CuckooFilter$[k]$.add$(f)$
        return True
return False
```

Example 2.12: Add elements to the filter

Consider a CuckooFilter data structure of length $m = 8$ that, for simplicity, stores only one $p = 16$-bit fingerprint per bucket. We use a single 32-bit MurmurHash3 hash function to compute the fingerprints and the bucket indices.

Similar to other examples, we insert the names of capital cities, starting with element ***Copenhagen***, whose p-bit fingerprint is

$$f = \text{MurmurHash3}(\textbf{\textit{Copenhagen}}) \bmod 2^p = 49615.$$

When its primary bucket i according to the formula (2.10) is

$$i = \text{MurmurHash3}(\boldsymbol{Copenhagen}) \bmod m = 7,$$

and the alternate bucket j can be derived from i and the fingerprint f as follows:

$$j = (i \oplus \text{MurmurHash3}(f)) \bmod m = (1 \oplus 34475545) \bmod 10 = 0.$$

Thus, we can index the fingerprint f into either bucket 7 or 0, and, since the filter is empty, we use the primary bucket:

0	1	2	3	4	5	6	7	8	9
							49615		

Similarly, we index the element ***Athens*** with fingerprint $f = 27356$ and the candidate buckets 0 and 7. The primary bucket 0 isn't occupied and allows the storing of the fingerprint freely:

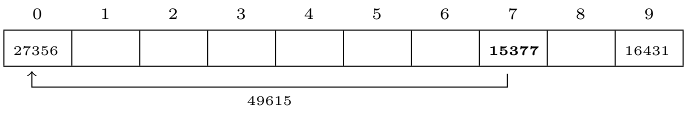
0	1	2	3	4	5	6	7	8	9
27356							49615		

Consider element ***Lisbon***, whose fingerprint is $f = 16431$ and the candidate buckets are 7 and 9. We start with the primary bucket 7, but it is already occupied in the CuckooFilter and it is at its maximum capacity of one, hence we check the alternate bucket 9, which is empty, and we store the fingerprint there:

0	1	2	3	4	5	6	7	8	9
27356							49615		**16431**

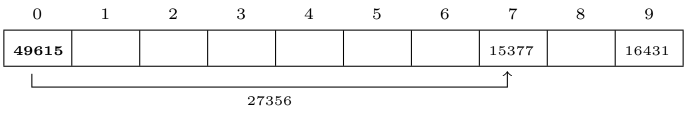
Next, consider the element ***Helsinki***. It has fingerprint $f = 15377$ and both bucket indices equal to 7. Note, that such an index collision is more likely for small filters, as we have in this example, then for real ones. Bucket 7 is occupied and cannot accept more than one element, therefore we need to start the relocation procedure in the filter. First, we start with the bucket k and swap the value 49615 from bucket 7 with the value f, then, relocate that value to a new bucket k that is derived from it by formula (2.11):

$$k = (7 \oplus \text{MurmurHash3}(49615)) \bmod 10 = 0.$$



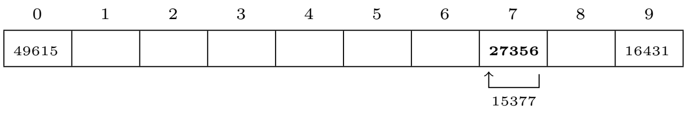
Unfortunately, bucket 0 already contains value 27356 and we swap it with the 49615, and need to compute the new bucket index for it:

$$k = (0 \oplus \text{MurmurHash3}(27356)) \bmod 10 = 7.$$



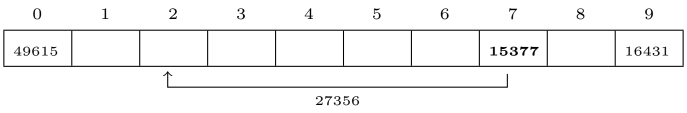
We are back to bucket 7 which isn't empty and we are required to repeat the relocation procedure once again. First, we store value 27356 in the bucket, and then compute a new bucket for the value 15377:

$$k = (7 \oplus \text{MurmurHash3}(15377)) \bmod 10 = 7.$$

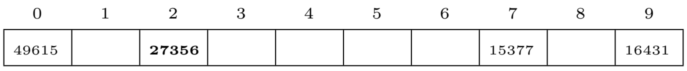


Due to the index collision that we mentioned earlier for that fingerprint, we return to bucket 7 again and store value 15377 in it while relocating the value 27356 to a new bucket k:

$$k = (7 \oplus \text{MurmurHash3}(27356)) \bmod 10 = 2.$$



Fortunately, bucket 2 is empty, so we can store value 27356 and finish the insertion procedure.



The testing of element existence in the filter is straightforward. First, for the tested element, we compute the fingerprint and its candidate buckets. If the fingerprint is present in either bucket, we conclude that the element **may exist**. Otherwise, it is **definitely not** in the filter.

Algorithm 2.15: Testing element in the Cuckoo filter

```
Input: Element $x \in \mathbb{D}$
Input: Cuckoo filter with fingerprinting and hash function $h$
Output: False if element not found and True if element may exist
$f \leftarrow$ fingerprint($x$)
$i \leftarrow h(x)$
$j \leftarrow i \oplus h(f)$
if $f \in$ CuckooFilter[$i$] or $f \in$ CuckooFilter[$j$] then
    return True
return False
```

Example 2.13: Test elements in the filter

Consider the CuckooFilter data structure that we built in Example 2.12:

0	1	2	3	4	5	6	7	8	9
49615		27356					15377		16431

Let's test the element ***Lisbon*** whose candidate buckets are 7 and 9 and fingerprint is $f = 16431$ as we computed above. We can find value 16431 in bucket 9 and conclude that the element ***Lisbon*** may exist in the filter.

In contrast, consider the element ***Oslo***, that has fingerprint $f = 53104$ and candidate buckets 0 and 6. As we can see, there is no such value in these buckets, therefore element ***Oslo*** is definitely not in the filter.

In order to delete an element, we build the fingerprint, then compute the indices of the candidate buckets and lookup for the fingerprint there. If it matches any existing values in either bucket, one copy of the fingerprint is removed from that bucket.

Algorithm 2.16: Deleting element from the Cuckoo filter

```
Input: Element $x \in \mathbb{D}$
Input: Cuckoo filter with fingerprinting and hash function $h$
Output: True if element has been deleted and False otherwise
$f \leftarrow$ fingerprint($x$)
$i \leftarrow h(x)$
$j \leftarrow i \oplus h(f)$
if $f \in$ CuckooFilter[$i$] then
    CuckooFilter[$i$].drop($f$)
    return True
else if $f \in$ CuckooFilter[$j$] then
    CuckooFilter[$j$].drop($f$)
    return True
return False
```

Properties

When the Cuckoo filter needs to support deletion, it must store multiple copies of the same value or arrange counters for each stored value. However, both approaches induce probabilistically correct, one – due to limited bucket capacity (we cannot store more than $2b$ same values in the table), and another because the counters overflow, as it was explained in Counting Bloom filter. However, a non-deletable Cuckoo filter doesn't have this problem and much more space efficient because it doesn't need to remember identical values that were added multiple times.

False positives are possible. It is possible that different elements could share the same fingerprint, but in most cases they have different candidate buckets, thus can still be differentiated. However, when the candidate buckets are also the same for those elements, a *hard collision* occurs. Due to such extremely rare events, the filter can end up in false positive responses and their probability P_{fp} is

$$P_{fp} = 1 - \left(1 - \frac{1}{2^p}\right)^{2b} \approx \frac{2b}{2^p}. \tag{2.12}$$

The formula (2.12) shows that, under the fixed number of expected elements n, there is a trade-off between the probability of false positives P_{fp} and the bucket size b, that can be compensated by the length of the fingerprints p. Intuitively, if the fingerprints are sufficiently long, Partial–Key Cuckoo hashing is a good approximation to standard Cuckoo hashing, but longer fingerprints affect the required space.

Thus, the recommended fingerprint length p can be estimated as

$$p \geq \left\lceil \log \frac{2b}{P_{fp}} \right\rceil, \tag{2.13}$$

and if we want to store in m buckets of size b at least as many values as the number of input elements, the length of the filter is lower bounded by

$$m \geq \left\lceil \frac{n}{b} \right\rceil. \tag{2.14}$$

False negatives are not possible. Similar to others, if the Cuckoo filter finds that an element is not a member, then it is definitely not a member of the set:

$$P_{fn} = 0. \tag{2.15}$$

Cuckoo filters ensure high space occupancy because they refine earlier element-placement decisions when adding new elements. However, they have a maximum capacity, which is expressed as a *load factor* α. After reaching the maximum feasible load factor, insertions are non-trivially and increasingly likely to fail, hence the hash table must expand to store more elements.

The average number of bits per element β can be defined as the ratio between the length of the fingerprints and the load factor α that, under the fixed false positive probability P_{fp}, can be estimated as

$$\beta \leq \frac{1}{\alpha} \cdot \left\lceil \log \frac{2b}{P_{fp}} \right\rceil. \tag{2.16}$$

Since Cuckoo hashing schema uses two hash functions, the load factor with buckets of size $b = 1$ is 50% as the hash table is directly mapped. However, increasing the bucket size allows to improve table occupancy, for instance, for $b = 2$ and $b = 4$ the load factors are 84% and 95%, correspondingly.

The experimental study [Fa14] showed that buckets of size $b \in \{1, 2, 3, 4\}$ are enough for practically important cases.

Example 2.14: Required memory

For instance, we want to handle 1 billion elements with a Cuckoo filter, keeping the probability of false positive events at about 2% and table occupancy at 84%. To support such load factor we choose the size of buckets as $b = 2$, meaning the length of the filter $m = 2^{29}$, according to (2.14).

As stated in (2.13), the minimal fingerprint length is

$$p = \left\lceil \log \frac{4}{0.02} \right\rceil = 8.$$

Therefore, the required length of the fingerprint is 8 bits and the total size of the Cuckoo filter is $2 \cdot 8 \cdot 2^{29}$ bits that is roughly 1.07 GB of memory.

To compare the space requirements with other studied filters, we can use $b = 1$ that achieves 50% of table occupancy and requires the filter of length $m = 2^{30}$. According to (2.13), we need to use 9-bits fingerprints, that results in about 0.94 GB of memory.

In fact, Cuckoo filters use a similar approach as the *d-left Counting Bloom filter* [Bo06], but they achieve better space efficiency and much more straightforward to implement. For applications that store many elements and target moderately low false positive rates (less than 3%), Cuckoo filters offer lower space overhead than even space-optimized Bloom filters.

However, when the Cuckoo filter is at its maximum capacity, the underlying hash table must be extended, until then it is not possible to add new elements. In contrast, with the Bloom filter it is still possible to keep inserting new elements at the cost of the increased false positive rate.

Conclusion

This chapter is dedicated to membership problems and we learned how traditional hash tables can be replaced or extended to be practically applicable for big data handling. We have learned the most well-known probabilistic data structure called the Bloom filter and discussed its strength and weaknesses, then considered its modifications that are widely used in practice. Additionally, we studied the modern alternatives that have better data locality, support more operations, and are tuned for good performance for large datasets.

If you are interested in more information about the material covered here or want to read the original papers, please take a look at the list of references that follows this chapter.

In the next chapter we study the problem of determining the number of distinct elements in a dataset, that can be challenging for big data and also requires probabilistic approaches to be efficiently solved.

Bibliography

[Al07] Almeida, P., et al. (2007) “Scalable Bloom Filters”, *Information Processing Letters*, Vol. 101 (6), pp. 255–261.

[Be11] Bender, M., et al. (2011) “Don’t Thrash: How to Cache your Hash on Flash”, *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems*, Portland, pp. 1, USENIX Association Berkeley.

[Be12] Bender, M., et al. (2012) “Don’t Thrash: How to Cache your Hash on Flash”, *Proceedings of the VLDB Endowment*, Vol. 5 (11), pp. 1627–1637.

[Bl70] Bloom, B.H. (1970) “Space/Time Trade-offs in Hash Coding with Allowable Errors”, *Communications of the ACM*, Vol. 13 (7), pp. 422–426.

[Bo06] Bonomi, F., et al. (2006) “An Improved Construction for Counting Bloom Filters”, *Proceedings of the 14th conference on Annual European Symposium*, Vol. 14, pp. 684–695.

[Br04] Broder, A., Mitzenmacher, M. (2004) “Network Applications of Bloom Filters: A Survey”, *Internet Mathematics*, Vol. 1 (4), pp. 485–509.

[Fa00] Fan, L., et al. (2000) “Summary cache: a scalable wide-area web cache sharing protocol”, *Journal IEEE/ACM Transactions on Networking*, Vol. 8 (3), pp. 281–293.

[Fa14] Fan, B., et al. (2014) “Cuckoo Filter: Practically Better Than Bloom”, *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, Sydney, Australia — December 02–05, 2014, pp. 75–88, ACM New York, NY.

[Ki08] Kirsch, A., Mitzenmacher, M. (2008) “Less hashing, same performance: Building a better Bloom filter”, *Journal Random Structures & Algorithms*, Vol. 33 (2), pp. 187–218.

[Mi02] Mitzenmacher, M. (2002) “Compressed Bloom filters”, *IEEE/ACM Transactions on Networking*, Vol. 10 (5), pp. 604–612.

[Ta12] Tarkoma, S., et al. (2012) “Theory and Practice of Bloom Filters for Distributed Systems”, *IEEE Communications Surveys and Tutorials*, Vol. 14 (1), pp. 131–155.

3 Cardinality

The *cardinality* estimation problem is a task to find the number of distinct elements in a dataset where duplicates are present. Traditionally, to determine the exact cardinality of a set, classical methods build a list of all elements, and use sorting and search to avoid listing elements multiple times. Counting the number of elements in that list gives the accurate number of the unique elements, but it has a time complexity of $O(N \cdot \log N)$, where N is the number of all elements including duplicates, and requires auxiliary linear memory, that is unlikely to be feasible for Big Data applications that operate huge datasets of large cardinalities.

Example 3.1: Unique visitors

One of the valuable KPIs for any website is the number of unique visitors that have visited it over a specified period of time. For simplicity, we assume that unique visitors use different IP addresses, therefore we need to calculate the number of unique IPs which according to IPv6 Internet Protocol version are represented by 128-bit strings. Is this an easy task? Can we just use the classical methods to count the number exactly? That depends on the popularity of the website.

Consider traffic statistics for March 2017 of the top three most popular retail websites in the United States: *amazon.com*, *ebay.com* and *walmart.com*. According to SimilarWeb[1], the average number of visits to those websites was about 1.44 billion and the average number of pages viewed per visit was 8.24. Therefore, the statistics for March 2017 include about 12 billion IP addresses at 128-bit each, meaning a total size of 192 GB.

If we assume that every 10th of those visitors was unique, we can expect cardinality of such a set at about 144 million and the memory required to store the list of unique elements is 23 GB.

Another example illustrates the challenge of cardinality estimation for scientific researches.

Example 3.2: DNA analysis (Giroire, 2006)

One of the long-standing tasks in human genome research is to study correlations in DNA sequences. DNA molecules include two paired strands, each made up of four chemical DNA-base units, marked A (adenine), G (guanine), C (cytosine), and T (thymine). The human genome contains about 3 billion such base pairs. Sequencing means determining the exact order of the base pairs in a segment of DNA.

From a mathematical point of view, a DNA sequence can be considered a string of symbols A, G, C, T which can be as long as you want, and we can consider them as an example of a potentially infinite dataset.

The correlation measuring problem can be formulated as a task of

[1] Traffic Overview `https://www.similarweb.com/website/amazon.com?competitors=ebay.com`

determining the number of distinct substrings of some fixed size in a piece of DNA. The idea is that a sequence with a few distinct substrings is more correlated than a sequence of the same size but with more distinct substrings.

Such experiments demand multiple runs on many huge files and to speed up the research they require only limited or even constant memory and small execution time, which is unfeasible with exact counting algorithms.

Thus, the possible gains of the accurate cardinality estimation are neglected by large time processing and memory requirements. Big Data applications shell use more practical approaches, mostly based on various probabilistic algorithms, even if they can provide only approximated answers.

While processing data, it is important to understand the size of the dataset and the number of possible distinct elements.

Consider the potentially infinite sequence of 1-letter strings a, d, s, $\ldots$, which is based on letters from the English alphabet. The cardinality can be easily estimated and it is upper bounded by the number of letters, which is 26 in the modern English language. Obviously, in this case, there is no need to apply any probabilistic approach and a naive dictionary-based solution of exact cardinality calculation works very well.

To approach the cardinality problem, many of the popular probabilistic methods are influenced by the ideas of the Bloom filter algorithm, they operate hash values of elements, then observe common patterns in their distribution, and make reasoned "guesses" about the number of unique elements without the need to store all of them.

3.1 Linear Counting

As a first probabilistic approach to the cardinality problem, we consider the linear-time probabilistic counting algorithm, the *Linear Counting* algorithm. The original ideas were proposed by Morton Astrahan, Mario Schkolnick, and Kyu-Young Whang in 1987 [As87] and the practical algorithm was published by Kyu-Young Whang, Brad Vander-Zanden, and Howard Taylor in 1990 [Wh90].

The immediate improvement to the classical exact methods was to hash elements with some hash function h, which out-of-the-box can eliminate duplicates without the need to sort elements with a payout of introducing some probability of error due to possible hash collisions (we cannot distinguish duplicates and “accidental duplicates”). Thus, using such a hash table, only a proper scan procedure is required to implement a simple algorithm that already outperforms the classical approach.

However, for datasets with huge cardinalities, such hash tables could be quite large and require memory that grows linearly with the number of distinct elements in the set. For systems with limited memory, it will require disk or distributed storage at some point, which drastically reduces the benefits of hash tables due to slow disk or network access.

Similar to the Bloom filter idea, to work-around such an issue, the Linear Counting algorithm doesn’t store the hash values themselves, but instead their corresponding bits, replacing the hash table with a bit array LinearCounter of length m. It is assumed that m still is proportional to the expected number of distinct elements n, but requires only 1 bit per element which is feasible for most cases.

In the beginning, all bits in LinearCounter are equal to zero. To add a new element x into such a data structure, we compute its hash value $h(x)$ and set the corresponding bit to one in the counter.

Algorithm 3.1: Adding element to the Linear counter

```
Input: Element $x \in \mathbb{D}$
Input: Linear counter with hash function $h$
$j \leftarrow h(x)$
if LinearCounter[$j$] $= 0$ then
    LinearCounter[$j$] $\leftarrow 1$
```

Since only one hash function h is used, we can expect many additional hard collisions when two different hash values set the same bit in the array. Thus, the exact (or even near-exact) number of distinct elements can no longer be directly obtained from such a sketch.

The idea of the algorithm leads to distributing elements into buckets (bits indexed by hash values) and keeps a LinearCounter bit array indicating which buckets are hit. Observing the number of hits in the array leads to the estimate of the cardinality.

In the first step of the Linear Counting algorithm, we build our LinearCounter data structure as is shown in Algorithm 3.1. Having such a sketch, the cardinality can be estimated using the observed fraction of empty bits V by the formula:

$$n \approx -m \cdot \ln \mathrm{V}. \tag{3.1}$$

We see clearly now how collisions impact on the cardinality estimation in the Linear Counting algorithm — each collision reduces the number of bits that have to be set, making the observed fraction of unset bits bigger than the real value. If there were no hash collisions, the final count of set bits would be the desired cardinality. However, collisions are unavoidable and the formula (3.1) actually gives an overestimation of the exact cardinality and, since the cardinality is an integer value, we prefer to round its result to the nearest smaller integer.

Thus, we can formulate the complete counting algorithm as below.

Algorithm 3.2: Estimating cardinality with Linear Counting

```
Input: Dataset $\mathbb{D}$
Output: Cardinality estimation
LinearCounter[$i$] $\leftarrow 0, i = 0 \ldots m-1$
for $x \in \mathbb{D}$ do
    Add($e$, LinearCounter)
Z $\leftarrow \underset{i=0 \ldots m-1}{\mathtt{count}}$ (LinearCounter[$i$] $= 0$)
return $\lfloor -m \cdot \ln \frac{\mathrm{Z}}{m} \rfloor$
```

Example 3.3: Linear Counting algorithm

Consider a dataset that contains 20 names of capital cities extracted from recent news articles: ***Berlin***, *Berlin*, ***Paris***, *Berlin*, ***Lisbon***, ***Kiev***, *Paris*, ***London***, ***Rome***, ***Athens***, ***Madrid***, ***Vienna***, *Rome*, *Rome*, *Lisbon*, *Berlin*, *Paris*, *London*, *Kiev*, ***Washington***.

For such small cardinalities (actual cardinality is 10) to have a standard error about 10% we need to choose the length of the LinearCounter data structure at least as the expected number of unique elements, thus let's choose $m = 2^4$. As the hash function h with values in $\{0, 1, \dots, 2^4-1\}$ we use a function based on 32-bit MurmurHash3 defined as

$$h(x) := \text{MurmurHash3}(x) \bmod m,$$

and cities hash values can be found in the table below.

City	h(City)
Athens	12
Berlin	7
Kiev	13
Lisbon	15
London	14

City	h(City)
Madrid	14
Paris	8
Rome	1
Vienna	6
Washington	11

As we can see, the cities ***London*** and ***Madrid*** share the same value, but such collisions are expected and completely natural. The LinearCounter data structure has the following view:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	0	0	0	0	1	1	1	0	0	1	1	1	1	1

According to the Linear Counting algorithm, we calculate the fraction V of empty bits in the LinearCounter:

$$\text{V} = \frac{9}{16} = 0.5625$$

and the estimated cardinality is

$$n \approx -16 \cdot \ln 0.5625 \approx 9.206,$$

which is pretty close to the exact number 10.

Properties

If the hash function h can be computed in a constant time (which is true for the most popular hash functions), the time to process every element is a fixed constant O(N), where N is the total number of elements, including duplicates. Thus, the algorithm has O(N) time complexity.

As for any other probabilistic algorithm, there is a number of parameters that can be tuned to influence its performance.

The expected accuracy of the estimation depends on the bit array size m and its ratio to the number of distinct elements $\alpha = \frac{m}{n}$, called the *load factor*. Unless $\alpha \ \geq 1$ ($m > n$ is not a practically interesting case), there is a non-zero probability P_{full} that LinearCounter bit array becomes full, called the *fill-up probability*, that fatally distorts the algorithm and blows up the expression (3.1). The probability P_{full} depends on the load factor and, consequently, on the size m that should be selected big enough to have the fill-up probability negligible.

The standard error δ is a measure of the variability of the estimate provided by Linear Counting and there is a trade-off between it and the bit array size m. Decreasing the standard error results in more precise estimates, but increases the required memory.

Table 3.1: Trade-off between accuracy and bit array size

n	m	
	$\delta = 1\%$	$\delta = 10\%$
1000	5329	268
10000	7960	1709
100000	26729	12744
1000000	154171	100880
10000000	1096582	831809
100000000	8571013	7061760

The dependency on choosing m is quite complex and has no analytical solution. However, for a widely acceptable fill-up probability $\mathrm{P}_{full} = 0.7\%$

the algorithm authors have provided precomputed values that are given in Table 3.1 and can be used as references.

Since the fill-up probability is never zero, the bit array very rarely becomes full and distorts Algorithm 3.2. When working with small datasets, we can re-index all elements with a different hash function or increase the size of LinearCounter. Unfortunately, such solutions won't work for huge datasets and, together with quite high time complexity, require a search for alternatives.

However, Linear Counting performs very well when the cardinality of the dataset being measured is not extremely big and can be used to improve other algorithms, developed to provide the best possible behavior for huge cardinalities.

In the Linear Counting algorithm, the estimation of the cardinality is approximately proportional to the exact value, this is why the term "linear" is used. In the next section, we consider an alternative algorithm that could be classified as "logarithmic" counting since it is based on estimations that are logarithms of the true cardinality.

3.2 Probabilistic Counting

One of the counting algorithms that is based on the idea of observing common patterns in hashed representations of indexed elements is a class of *Probabilistic Counting* algorithms invented by Philippe Flajolet and G. Nigel Martin in 1985 [Fl85].

As usual, every element is pre-processed by applying a hash function h that transforms elements into integers sufficiently uniformly distributed over a scalar range $\{0, 1, \dots, 2^{\mathrm{M}} - 1\}$ or, equivalently, over the set of binary strings[2] of length M:

$$h(x) = i = \sum_{k=0}^{\mathrm{M}-1} i_k \cdot 2^k := (i_0 i_1 \dots i_{\mathrm{M}-1})_2\,, \, i_k \in \{0, 1\}.$$

[2]We use the "LSB 0" numbering scheme and start at zero for the least significant bit (LSB)

Flajolet and Martin noticed that patterns:

$$0^k 1 := \overbrace{00\ldots0}^{k \text{ times}}1$$

should appear in such binary strings with probability $2^{-(k+1)}$ and, if recorded for each indexed element, can play the role of a cardinality estimator.

Every pattern can be associated with its index, called *rank*, that is calculated by the formula:

$$\operatorname{rank}(i) = \begin{cases} \min\limits_{i_k \neq 0} k, & \text{for } i > 0, \\ \mathrm{M} & \text{for i} = 0 \end{cases} \tag{3.2}$$

and simply equivalent to the left-most position of 1, known as the least significant 1-bit position.

Example 3.4: Rank calculation

Consider an 8-bit long integer number 42 that has the following binary representation using the "LSB 0" numbering scheme:

$$42 = 0\cdot2^0 + 1\cdot2^1 + 0\cdot2^2 + 1\cdot2^3 + 0\cdot2^4 + 1\cdot2^5 + 0\cdot2^6 + 0\cdot2^7 = (0\mathbf{1}010100)_2\,.$$

Thus, the ones appear at positions 1, 3, and 5, therefore, according to the definition (3.2), the rank(42) is equal to:

$$\operatorname{rank}(42) = \min(1, 3, 5) = 1.$$

The occurrences of the 0^k1 pattern, or simply $\operatorname{rank}(\cdot) = k$, in binary representations of hash values of each indexed element, can be compactly stored in a simple data structure COUNTER, also known as a *FM Sketch*, that is represented as a bit array of length M.

At the start, all bits in COUNTER are equal to zero. When we need to add a new element x into the data structure, we compute its hash value using the hash function h, then calculate $\operatorname{rank}(x)$ and set the corresponding bit to one in the array, as stated in the algorithm below.

Algorithm 3.3: Adding element to simple counter

```
Input: Element $x \in \mathbb{D}$
Input: Simple counter with hash function $h$
$j \leftarrow$ rank$(h(x))$
if $\textsc{Counter}[j] = 0$ then
    $\textsc{Counter}[j] \leftarrow 1$
```

In this way, the one in the COUNTER at some position j means that the pattern 0^j1 has been observed at least once amongst the hashed values of all indexed elements.

Example 3.5: Build a simple counter

Consider the same dataset as in Example 3.3 that contains 20 names of capital cities extracted from recent news articles: ***Berlin***, *Berlin*, ***Paris***, *Berlin*, ***Lisbon***, ***Kiev***, *Paris*, ***London***, ***Rome***, ***Athens***, ***Madrid***, ***Vienna***, *Rome*, *Rome*, *Lisbon*, *Berlin*, *Paris*, *London*, *Kiev*, ***Washington***.

As the hash function h we can use 32-bit MurmurHash3, that maps elements to values from $\{0, 1, \ldots, 2^{32} - 1\}$, therefore we can use a simple counter COUNTER of length $M = 32$. Using the hash values already computed in Example 3.3 and the definition (3.2), we calculate ranks for each element:

City	h(City)	rank
Athens	4161497820	2
Berlin	3680793991	0
Kiev	3491299693	0
Lisbon	629555247	0
London	3450927422	1
Madrid	2970154142	1
Paris	2673248856	3
Rome	50122705	0
Vienna	3271070806	1
Washington	4039747979	0

Thus, the COUNTER has the following form:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Let's stress a very interesting theoretical observation. Based on the uniform distribution of the values, if n is the exact number of the distinct elements indexed so far, then we can expect that one in the first position can appear in about $\frac{n}{2}$ cases, in the second position in about $\frac{n}{2^2}$ cases, and so on. Thus, if $j \gg \log_2 n$, then the probability of discovering one in the j-th position is close to zero, hence the COUNTER[j] will almost certainly be zero. Similarly, for $j \ll \log_2 n$ the COUNTER[j] will almost certainly be one. If value j is around the $\log_2 n$, then the probability to observe one or zero in that position is about the same.

Thus, the left-most position R of zero in the COUNTER after inserting all elements from the dataset can be used as an indicator of $\log_2 n$. In fact, a correction factor φ is required and the cardinality estimation can be done by the formula:

$$n \approx \frac{1}{\varphi} 2^{\mathrm{R}}, \tag{3.3}$$

where $\varphi \approx 0.77351$.

Flajolet and Martin have chosen to use the least significant 0-bit position (the left-most position of 0) as the estimation of cardinality and built their algorithm based on it. However, from the observation above we can see, that the most significant 1-bit position (the right-most position of 1) can be used for the same purpose; however, it has a flatter distribution that leads to a bigger standard error.

The algorithm to compute the left-most position of zero in a simple counter can be formulated as follows.

Algorithm 3.4: Computing the left-most zero position

```
Input: Simple counter of length M
Output: The left-most position of zero
for $j \leftarrow 0$ to $M - 1$ do
    if $\text{COUNTER}[j] = 0$ then
        return $j$
return M
```

Example 3.6: Cardinality estimation with simple counter

Consider the COUNTER from Example 3.5 and compute the estimated number of distinct elements.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	**0**	0	0	0	0	0	0	0	0	0	0	0

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Using Algorithm 3.4, in the COUNTER the left-most value 0 appears in position $R = 4$, therefore, according to the formula (3.3), the cardinality estimation is

$$n \approx \frac{1}{0.77351} 2^4 \approx 20.68.$$

The exact cardinality of the set is 10, meaning the computed estimation has a huge error due to the fact that the values of R are integers and for very close ranks we can obtain results that differ in some binary orders of magnitude. For instance, in our example, $R = 3$ would give an almost perfect estimation of 10.34.

Theoretically, the cardinality estimation based on a single simple counter can provide very close expected values, but it has quite a high variance that usually corresponds, as we also observed in Example 3.6, to the unpractical standard error δ of one binary order of magnitude.

Obviously, the weakness of the one-counter approach is that there is a lack of highly confident estimations for the cardinality (in fact, it makes its prediction based on a single estimation only).

Thereby, the natural extension of the algorithm is to have many simple counters and, consequently, increase the number of estimations. The final prediction n can be obtained by averaging the predictions R_k from those counters $\{\textsc{Counter}_k\}_{k=0}^{m-1}$.

Thus, the modified formula (3.3) of the Probabilistic Counting algorithm has the form:

$$n \approx \frac{1}{\varphi} 2^{\bar{\mathrm{R}}} = \frac{1}{\varphi} 2^{\frac{1}{m} \sum_{k=0}^{m-1} \mathrm{R}_k}, \tag{3.4}$$

and the cardinality n will have the same-quality estimated value, but with a much smaller variance.

The obvious practical disadvantage to building m independent simple counters is the requirement to compute values of m different hash functions that, given that a single hash function can be computed in $\mathrm{O}(1)$, has $\mathrm{O}(m)$ time complexity and quite high CPU costs.

The solution to optimizing the Probabilistic Counting algorithm is to apply a special procedure, called *stochastic averaging*, when m hash functions are replaced by only one but its value split by quotient and remainder, which are used to update a single counter per element. The remainder r is used to choose one out of m counters and quotient q to calculate the rank and find the appropriate index to be updated in that counter.

Algorithm 3.5: Using stochastic averaging to update counters

```
Input: Element $x \in \mathbb{D}$
Input: Array of $m$ simple counters with hash function $h$
$r \leftarrow h(x) \bmod m$
$q \leftarrow h(x) \text{ div } m := \left\lfloor \frac{h(x)}{m} \right\rfloor$
$j \leftarrow \texttt{rank}(q)$
if $\textsc{Counter}_r[j] = 0$ then
    $\textsc{Counter}_r[j] \leftarrow 1$
```

Applying the stochastic averaging Algorithm 3.5 to the Probabilistic Counting, under the assumption that quotient-based distribution of elements is fair enough, we may expect that $\frac{n}{m}$ elements have been indexed by each simple counter $\{\textsc{Counter}_k\}_{k=0}^{m-1}$, therefore the formula (3.4) is a good estimation for $\frac{n}{m}$ (not n directly):

$$n \approx \frac{m}{\varphi} 2^{\bar{\mathrm{R}}} = \frac{m}{\varphi} 2^{\frac{1}{m} \sum\limits_{k=0}^{m-1} \mathrm{R}_k}. \tag{3.5}$$

Algorithm 3.6: Flajolet–Martin algorithm (PCSA)

```
Input: Dataset $\mathbb{D}$
Input: Array of $m$ simple counters with hash function $h$
Output: Cardinality estimation
for $x \in \mathbb{D}$ do
    $r \leftarrow h(x) \bmod m$
    $q \leftarrow h(x) \operatorname{div} m$
    $j \leftarrow$ rank($q$)
    if $\textsc{Counter}_r[j] = 0$ then
        $\textsc{Counter}_r[j] \leftarrow 1$
$\mathrm{S} \leftarrow 0$
for r $\leftarrow 0$ to $m-1$ do
    $\mathrm{R} \leftarrow$ **LeftMostZero**($\textsc{Counter}_r$)
    $\mathrm{S} \leftarrow \mathrm{S} + \mathrm{R}$
return $\frac{m}{\varphi} \cdot 2^{\frac{1}{m}\mathrm{S}}$
```

The corresponding Algorithm 3.6 is called the *Probabilistic Counting algorithm with stochastic averaging* (PCSA) and is also known as the *Flajolet–Martin algorithm*. In comparison to its version with m hash functions, it reduces the time complexity for each element to about O(1).

Example 3.7: Cardinality estimation with stochastic averaging

Consider the dataset and the hash values computed in Example 3.5 and apply a stochastic averaging technique simulating $m = 3$ hash functions. We use the remainder r to choose one out of three counters and the quotient q to calculate the rank.

City	h(City)	r	q	rank(q)
Athens	4161497820	0	1387165940	2
Berlin	3680793991	1	1226931330	1
Kiev	3491299693	1	1163766564	2
Lisbon	629555247	0	209851749	0
London	3450927422	2	1150309140	2
Madrid	2970154142	2	990051380	2
Paris	2673248856	0	891082952	3
Rome	50122705	1	16707568	4
Vienna	3271070806	1	1090356935	0
Washington	4039747979	2	1346582659	0

Every counter handles information for about one-third of the cities, therefore, the distribution is fair enough. After indexing all elements and setting the appropriate bits in the corresponding counters, our counters have the following forms.

COUNTER$_0$

0	**1**	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	**0**	1	1	0	0	0	0	0	0	0	0	0	0	0	0

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

COUNTER$_1$

0	1	2	**3**	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	**0**	1	0	0	0	0	0	0	0	0	0	0	0

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

COUNTER$_2$

0	**1**	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	**0**	1	0	0	0	0	0	0	0	0	0	0	0	0	0

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The left-most positions of zero for each counter (highlighted above) are $\mathrm{R}_0 = 1$, $\mathrm{R}_1 = 3$ and $\mathrm{R}_2 = 1$. Thus, the estimation of the cardinality according to formula (3.5) is

$$n \approx \frac{3}{\varphi} 2^{\frac{1}{3}\sum\limits_{k=0}^{2} \mathrm{R}_k} \approx \frac{3}{0.77351} 2^{\frac{1+3+1}{3}} \approx 12.31.$$

The computed estimation is very close to the true cardinality value of 10, and even without using too many counters, it notably outperforms the estimation from Example 3.6.

Properties

The Flajolet–Martin algorithm works well for datasets with large cardinalities and produces good approximations when $\frac{n}{m} > 20$. However, additional non-linearities can appear in the algorithm for small cardinalities that usually require special corrections.

One possible correction to the algorithm was proposed by Björn Scheuermann and Martin Mauve in 2007 [Sc07] which adjusted the formula (3.5) by adding a term that corrects it for small cardinalities and quickly converges to zero for large cardinalities:

$$n \approx \frac{m}{\varphi} \left(2^{\bar{\mathrm{R}}} - 2^{-\varkappa \cdot \bar{\mathrm{R}}} \right), \tag{3.6}$$

where $\varkappa \approx 1.75$.

The standard error δ of the Flajolet–Martin algorithm is inversely related to the number of used counters and can be approximated as

$$\delta \approx \frac{0.78}{\sqrt{m}}. \tag{3.7}$$

The reference values of the standard error for the widely used number of counters can be found in Table 3.2.

The length M of each counter Counter can be selected in a way that:

$$M > \log_2\left(\frac{n}{m}\right) + 4, \tag{3.8}$$

thus, practicaly used M = 32 is enough to count cardinalities well beyond 10^9 using 64 counters.

Table 3.2: Trade-off between accuracy and storage (M = 32)

m	Storage	δ
64	256 bytes	9.7%
256	1.024 KB	4.8%
1024	4.1 KB	2.4%

The simple counters that have been built for different datasets can be easily merged together, that results in a Counter for the union of those datasets. Such merging is trivial and can be done by applying a bitwise OR operation.

Like the Bloom filter, the Probabilistic Counting algorithms do not support deletions. But, following the approach used in the Counting Bloom filter, their inner bit arrays can be extended by counters and they will support probabilistically correct deletions. However, the increased storage requirements have to be taken into account.

3.3 LogLog and HyperLogLog

The most popular probabilistic algorithms to estimate cardinality used in practice are the LogLog family of algorithms that includes the *LogLog* algorithm, proposed by Marianne Durand and Philippe Flajolet in 2003 [Du03], and its successors *HyperLogLog* and *HyperLogLog++*.

The algorithms use an approach that is similar to the Probabilistic Counting algorithm in a way that estimation of the cardinality n is done by observing the maximum number of leading zeros in the binary representation of values. They all require an auxiliary memory and perform a single pass over the data to produce an estimate of the cardinality.

As usual, every element in the dataset is pre-processed by applying a hash function h that transforms elements into integers sufficiently uniformly distributed over a scalar range $\{0, 1, \ldots, 2^{\mathrm{M}}-1\}$ or, equivalently, over the set of binary strings[3] of length M:

$$h(x) = i = \sum_{k=0}^{\mathrm{M}-1} i_k \cdot 2^k := (i_0 i_1 \ldots i_{\mathrm{M}-1})_2\, , i_k \in \{0, 1\}.$$

The steps of the algorithms are similar to PCSA, which we reproduce here once again. First, it splits the initial dataset or input stream into some number of subsets, each of these is indexed by one of m simple counters. Then, according to the stochastic averaging, because there is a single hash function, we choose the counter for the particular element x using one part of its hash value $h(x)$, while another part is used to update the corresponding counter.

All algorithms discussed here are based on the observation of the patterns 0^k1 that occur at the beginning of the values for the particular counter, and associate each pattern with its index, called rank. The rank is equivalent to the least significant 1-bit position in the binary representation of the hash value of indexed element and can be calculated by the formula (3.2). Each simple counter builds its own cardinality observation based on the seen ranks, the final estimation of the cardinality is produced from such observations using an evaluation function.

In regards to storage, the counters in the Probabilistic Counting algorithm are relatively costly to maintain, but the LogLog algorithm

[3] We use the “LSB 0” numbering scheme and start at zero for the least significant bit (LSB)

suggests a more storage-efficient solution together with a better evaluation function and bias correction approach.

LogLog algorithm

The basic idea of the LogLog algorithm starts with the computation of ranks for each input element based on a single hash function h. Since we can expect that $\frac{n}{2^k}$ elements can have $\texttt{rank}(\cdot) = k$, where n is the total number of elements indexed into a counter, the maximal observed rank can provide a good indication of the value of $\log_2 n$:

$$\mathrm{R} = \max_{x \in \mathbb{D}}(\texttt{rank}(x)) \approx \log_2 n. \tag{3.9}$$

However, such estimation has an error of about ± 1.87 binary orders of magnitude, which is unpractical. To reduce the error, the LogLog algorithm uses a bucketing technique based on the stochastic averaging and splits the dataset into $m = 2^p$ subsets $\mathrm{S}_0, \mathrm{S}_1, \cdots, \mathrm{S}_{m-1}$, where the precision parameter p defines the number of bits used in navigation.

Thus, for every element x from the dataset, the first p bits of the M-bit hash value $h(x)$ can be taken to find out the index j of the appropriate subset:

$$j = (i_0 i_1 \ldots i_{p-1})_2\,,$$

and the rest $(\mathrm{M} - p)$ bits are indexed into the corresponding counter $\textsc{Counter}[j]$ to compute the rank and get the observation R_j according to formula (3.9).

Under fair distribution, every subset receives $\frac{n}{m}$ elements, therefore observations R_j from the counters $\{\textsc{Counter}[j]\}_{j=0}^{m-1}$ can provide an indication of the value of $\log_2 \frac{n}{m}$, and using their arithmetic mean with some bias correction, we can reduce a single observation variance:

$$n = \alpha_m \cdot m \cdot 2^{\frac{1}{m}\sum\limits_{j=0}^{m-1} \mathrm{R}_j}, \tag{3.10}$$

where $\alpha_m = \left(\Gamma\left(-\frac{1}{m}\right) \cdot \frac{1-2^{\frac{1}{m}}}{\log 2}\right)^m$, $\Gamma(\cdot)$ is the gamma function. However, for most practical cases $m \geq 64$ it is enough to just use $\alpha_m \approx 0.39701$.

Algorithm 3.7: Estimating cardinality with LogLog

```
Input: Dataset $\mathbb{D}$
Input: Array of $m$ LogLog counters with hash function $h$
Output: Cardinality estimation
$\textsc{Counter}[j] \leftarrow 0, j = 0 \ldots m-1$
for $x \in \mathbb{D}$ do
    $i \leftarrow h(x) := (i_0 i_1 \ldots i_{\mathrm{M}-1})_2\,, i_k \in \{0, 1\}$
    $j \leftarrow (i_0 i_1 \ldots i_{p-1})_2$
    $r \leftarrow$ rank$((i_p i_{p+1} \ldots i_{\mathrm{M}-1})_2)$
    $\textsc{Counter}[j] \leftarrow \max(\textsc{Counter}[j], r)$
$\mathrm{R} \leftarrow \frac{1}{m} \sum_{k=0}^{m-1} \textsc{Counter}[j]$
return $\alpha_m \cdot m \cdot 2^{\mathrm{R}}$
```

Properties

The standard error δ of the LogLog algorithm is inversely related to the number of used counters m and can be closely approximated as

$$\delta \approx \frac{1.3}{\sqrt{m}}. \tag{3.11}$$

Hence, for $m = 256$ the standard error is about 8% and for $m = 1024$ it decreases to about 4%.

The storage requirements of the LogLog algorithm can be estimated as $O(\log_2 \log_2 n)$ bits of storage if counts till n are needed. More precisely, the total space required by the algorithm in order to count to n is $m \cdot \log_2 \log_2 \frac{n}{m} (1 + O(1))$.

In comparison to the Probabilistic Counting algorithm where each counter requires 16 or 32 bits, the LogLog algorithm requires much smaller counters $\{\textsc{Counter}[j]\}_{j=0}^{m-1}$, usually of 5 bits each. However, while the LogLog algorithm provides better storage-efficiency than the Probabilistic Counting algorithm, it is slightly less accurate.

Assume that we need to count cardinalities till 2^{30}, that is about 1 billion, with an accuracy of about 4%. As already mentioned, for such a standard error, $m = 1024$ buckets are required, each of which will receive roughly $\frac{n}{m} = 2^{20}$ elements.

The $\log_2\left(\log_2 2^{20}\right) \approx 4.32$, therefore, it is enough to allocate about 5 bits per bucket (i.e., a value less than 32). Hence, to estimate cardinalities up to about 10^9 with the standard error of 4%, the algorithm requires 1024 buckets of 5 bits, which is 640 bytes in total.

HyperLogLog algorithm

An improvement of the LogLog algorithm, called *HyperLogLog*, was proposed by Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier in 2007 [Fl07]. The HyperLogLog algorithm uses 32-bit hash function, a different evaluation function, and various bias corrections.

Similar to the LogLog algorithm, HyperLogLog uses randomization to approximate the cardinality of a dataset and has been designed to handle cardinalities up to 10^9 with a single 32-bit hash function h splitting the dataset into $m = 2^p$ subsets, with precision $p \in 4 \ldots 16$.

Additionally, the evaluation function differentiates the HyperLogLog algorithm from the standard LogLog. The original LogLog algorithm uses the geometric mean while the HyperLogLog uses a function that is based on a normalized version of the harmonic mean:

$$\hat{n} \approx \alpha_m \cdot m^2 \cdot \left(\sum_{j=0}^{m-1} 2^{-\textsc{Counter}[j]}\right), \tag{3.12}$$

where

$$\alpha_m = \left(m \int_0^\infty \left(\log_2\left(\frac{2+x}{1+x}\right)\right)^m dx\right)^{-1}.$$

The approximate values of α_m can be found in Table 3.3.

The intuition behind using the harmonic mean is that it reduces the variance due to its property to tame skewed probability distributions.

Table 3.3: α_m for most used values of m

m	α_m
2^4	0.673
2^5	0.697
2^6	0.709
$\geq 2^7$	$\frac{0.7213 \cdot m}{m+1.079}$

However, the estimation (3.12), requires a correction for small and large ranges due to non-linear errors. Flajolet et al. empirically found that for small cardinalities $n < \frac{5}{2}m$ to achieve better estimates the HyperLogLog algorithm can be corrected with Linear Counting using a number of non-zero Counter[j] counters (if a counter has a zero value, we can say with certainty that the particular subset is empty).

Thus, for different ranges of cardinality, expressed as intervals on the estimate $\hat{n}$ computed by formula (3.12), the algorithm provides the following corrections:

$$n = \begin{cases} \textsc{LinearCounter}, & \hat{n} \leq \frac{5}{2}m \text{ and } \exists j : \textsc{Counter}[j] \neq 0 \\ -2^{32} \log\left(1 - \frac{\hat{n}}{2^{32}}\right), & \hat{n} > \frac{1}{30}2^{32} \\ \hat{n}, & \textit{otherwise}. \end{cases} \tag{3.13}$$

However, for $n = 0$ the correction it seems is not enough and the algorithm always returns roughly $0.7m$.

Since the HyperLogLog algorithm uses a 32-bit hash function, when cardinality approaches $2^{32} \approx 4 \cdot 10^9$ the hash function almost reaches its limit and the probability of collisions increases. For such large ranges, the HyperLogLog algorithm estimates the number of different hash values and uses it to approximate the cardinality. However, in practice, there is a danger that a higher number just cannot be represented and will be lost, impacting the accuracy.

Consider a hash function that maps the universe to values of M bits. At most such a function can encode 2^{M} different values and if the estimated cardinality n approaches such a limit, the hash collisions become more and more probable.

There is no evidence that some popular hash functions (e.g., MurmurHash3, MD5, SHA–1, SHA–256) perform significantly better than others in HyperLogLog algorithms or its modifications.

The complete HyperLogLog algorithm is shown below.

Algorithm 3.8: Estimating cardinality with HyperLogLog

```
Input: Dataset $\mathbb{D}$
Input: Array of $m$ LogLog counters with hash function $h$
Output: Cardinality estimation
$\textsc{Counter}[j] \leftarrow 0, j = 0 \dots m-1$
for $x \in \mathbb{D}$ do
    $i \leftarrow h(x) := (i_0 i_1 \dots i_{31})_2, i_k \in \{0, 1\}$
    $j \leftarrow (i_0 i_1 \dots i_{p-1})_2$
    $r \leftarrow \mathtt{rank}((i_p i_{p+1} \dots i_{31})_2)$
    $\textsc{Counter}[j] \leftarrow \max(\textsc{Counter}[j], r)$
$\mathrm{R} \leftarrow \sum_{k=0}^{m-1} 2^{-\textsc{Counter}[j]}$
$\hat{n} = \alpha_m \cdot m^2 \cdot \frac{1}{\mathrm{R}}$
$n \leftarrow \hat{n}$
if $\hat{n} \leq \frac{5}{2} m$ then
    $\mathrm{Z} \leftarrow \underset{j=0 \dots m-1}{\mathtt{count}} (\textsc{Counter}[j] = 0)$
    if $\mathrm{Z} \neq 0$ then
        $n \leftarrow m \cdot \log(\frac{m}{\mathrm{Z}})$
else if $\hat{n} > \frac{1}{30} 2^{32}$ then
    $n \leftarrow -2^{32} \cdot \log\left(1 - \frac{n}{2^{32}}\right)$
return $n$
```

Properties

Similar to the LogLog algorithm, there is a clear trade-off between the standard error δ and the number of counters m:

$$\delta \approx \frac{1.04}{\sqrt{m}}.$$

The memory requirement does not grow linearly with the number of elements (unlike, e.g., the Linear Counting algorithm), allocating $(\mathrm{M}-p)$ bits for the hash values and having $m = 2^p$ counters in total, the required memory is

$$\lceil \log_2(\mathrm{M}+1-p) \rceil \cdot 2^p \text{ bits }, \tag{3.14}$$

moreover, since the algorithm uses only 32-bit hash functions and the precision $p \in 4 \ldots 16$, the memory requirements for the HyperLogLog data structure is $5 \cdot 2^p$ bits.

Therefore, the HyperLogLog algorithm makes it possible to estimate cardinalities well beyond 10^9 with a typical accuracy of 2% while using a memory of only 1.5 KB.

For instance, the well-known in-memory database Redis maintains[4] HyperLogLog data structures of 12 KB that approximate cardinalities with a standard error of 0.81%.

While HyperLogLog, in comparison to LogLog, improved the cardinality estimation for small datasets, it still overestimates the real cardinalities in such cases.

The variants of the HyperLogLog algorithms are implemented in well-known databases such as Amazon Redshift, Redis, Apache CouchDB, Riak, and others.

[4]PFCOUNT in Redis `https://redis.io/commands/pfcount`

HyperLogLog++ algorithm

After some time, in 2013 [He13], an improved version of HyperLogLog was developed, the *HyperLogLog++* algorithm, published by Stefan Heule, Marc Nunkesser, and Alexander Hall and focused on large cardinalities and better bias correction.

The most noticeable improvement of the HyperLogLog++ algorithm is the usage of a 64-bit hash function. Clearly, the longer the output values of the hash function, the more different elements can be encoded. Such improvement allows to estimate cardinalities far larger than 10^9 unique elements, but when the cardinality approaches $2^{64} \approx 1.8 \cdot 10^{19}$, hash collisions become a problem for the HyperLogLog++ as well.

The HyperLogLog++ algorithm uses exactly the same evaluation function given by (3.12). However, it improves the bias correction. The authors of the algorithm performed a series of experiments to measure the bias and found that for $n \leq 5m$ the bias of the original HyperLogLog algorithm could be further corrected using empirical data collected over the experiments.

Additional to the original article, Heule et al. provided[5] empirically determined values to improve the bias correction in the algorithm – arrays of raw cardinality estimates rawEstimateData and related biases biasData. Of course, it is not feasible to cover every possible case, so the rawEstimateData provides an array of 200 interpolation points, storing the average raw estimate measured at this point over 5000 different datasets. biasData contains about 200 measured biases that correspond with the rawEstimateData. Both arrays are zero-indexed and contain precomputed values for all supported precisions $p \in 4 \ldots 18$, where the zero index in the arrays corresponds to the precision value 4. As an example, for $m = 2^{10}$ and $p = 10$ the needed data can be found in rawEstimateData[6] and biasData[6].

The bias correction procedure in the HyperLogLog++ algorithm can be formalized as follows.

[5] Appendix to HyperLogLog in Practice `http://goo.gl/iU8Ig`

Algorithm 3.9: Correcting bias in HyperLogLog++

```
Input: Estimate $\hat{n}$ with precision $p$
Output: Bias-corrected cardinality estimate
$n_{\text{low}} \leftarrow 0$, $n_{\text{up}} \leftarrow 0$, $j_{\text{low}} \leftarrow 0$, $j_{\text{up}} \leftarrow 0$
for $j \leftarrow 0$ to length($\textsc{rawEstimateData}[p-4]$) do
    if $\textsc{rawEstimateData}[p-4][j] \geq \hat{n}$ then
        $j_{\text{low}} \leftarrow j-1$, $j_{\text{up}} \leftarrow j$
        $n_{\text{low}} \leftarrow \textsc{rawEstimateData}[p-4][j_{\text{low}}]$
        $n_{\text{up}} \leftarrow \textsc{rawEstimateData}[p-4][j_{\text{up}}]$
        break
$b_{\text{low}} \leftarrow \textsc{biasData}[p-4][j_{\text{low}}]$
$b_{\text{up}} \leftarrow \textsc{biasData}[p-4][j_{\text{up}}]$
$y = \text{interpolate}\left((n_{\text{low}}, n_{\text{low}} - b_{\text{low}}), (n_{\text{up}}, n_{\text{up}} - b_{\text{up}})\right)$
return $y(\hat{n})$
```

Example 3.8: Bias correction using empirical values

As an example, assume that we have computed the cardinality estimation $\hat{n} = 2018.34$ using the formula (3.12) and want to correct it for the precision $p = 10$ $(m = 2^{10})$.

First, we check the $\textsc{rawEstimateData}[6]$ array and determine that such a value $\hat{n}$ falls in the interval between values with indices 73 and 74 of that array, where $\textsc{rawEstimateData}[6][73] = 2003.1804$ and $\textsc{rawEstimateData}[6][74] = 2026.071$:

$$2003.1804 \leq \hat{n} \leq 2026.071.$$

Thus, we need to retrieve biases from the $\textsc{biasData}[6]$ that are indexed at the same positions 73 and 74, which are $\textsc{biasData}[6][73] = 134.1804$ and $\textsc{biasData}[6][74] = 131.071$.

The correct estimation is in the interval:

$$[2003.1804 - 134.1804, 2026.071 - 131.071] = [1869.0, 1895.0]$$

and to compute the corrected approximation, we can interpolate that values, e.g., using k-nearest neighbor search or just by a linear interpolation $y(x) = a \cdot x + b$, where $y(2003.1804) = 1869.0$ and $y(2026.071) = 1895.0$.

Thus, using simple calculations, the interpolation line is

$$y = 1.135837 \cdot x - 406.28725,$$

and the interpolated value for our cardinality estimation is

$$n = y(\hat{n}) = y(2018.34) \approx 1886.218.$$

According to experiments performed by the authors of the HyperLogLog++, the estimate n_{lin} built according to the Linear Counting algorithm is still better for small cardinalities even comparing to the bias-corrected value n. Therefore, if at least one empty counter exists, the algorithm additionally computes the linear estimate and uses a list of empirical thresholds, that can be found in Table 3.4, to choose which evaluation should be preferred. In such a case, the bias-corrected value n is used only when the linear estimate n_{lin} falls above the threshold $\varkappa_m$ for the current m.

Example 3.9: Bias correction with the threshold

Consider Example 3.8, where for $m = 2^{10}$ we computed the bias-corrected value $n \approx 1886.218$. In order to determine whether or not we should prefer this value to the estimation by Linear Counting, we need to find out the number of empty counters Z in HyperLogLog++ data structure. Because we do not have that value in our example, assume it is Z = 73.

Thus, the linear estimation according to the formula (3.1) is

$$n_{\text{lin}} = 2^{10} \cdot \log\left(\frac{2^{10}}{73}\right) \approx 2704.$$

Next, we compare the n_{lin} to the threshold $\varkappa_m = 900$ from Table 3.4, which is far below the computed value, therefore, we prefer the bias-corrected estimate n to the Linear Counting estimate n_{lin}.

Table 3.4: Empirical thresholds $\varkappa_m$ for the supported precision values

p	m	$\varkappa_m$
4	2^4	10
5	2^5	20
6	2^6	40
7	2^7	80
8	2^8	220
9	2^9	400
10	2^{10}	900
11	2^{11}	1800
12	2^{12}	3100
13	2^{13}	6500
14	2^{14}	11500
15	2^{15}	20000
16	2^{16}	50000
17	2^{17}	120000
18	2^{18}	350000

The complete HyperLogLog++ algorithm is shown below.

Algorithm 3.10: Estimating cardinality with HyperLogLog++

```
Input: Dataset $\mathbb{D}$
Input: Array of $m$ LogLog counters with hash function $h$
Output: Cardinality estimation
$\textsc{Counter}[j] \leftarrow 0, j = 0 \dots m-1$
for $x \in \mathbb{D}$ do
    $i \leftarrow h(x) := (i_0 i_1 \dots i_{63})_2, i_k \in \{0,1\}$
    $j \leftarrow (i_0 i_1 \dots i_{p-1})_2$
    $r \leftarrow \texttt{rank}((i_p i_{p+1} \dots i_{63})_2)$
    $\textsc{Counter}[j] \leftarrow \max(\textsc{Counter}[j], r)$
$\mathrm{R} \leftarrow \sum_{k=0}^{m-1} 2^{-\textsc{Counter}[j]}$
$\hat{n} = \alpha_m \cdot m^2 \cdot \frac{1}{\mathrm{R}}$
$n \leftarrow \hat{n}$
if $\hat{n} \leq 5m$ then
    $n \leftarrow \textbf{CorrectBias}(\hat{n})$
$\mathrm{Z} \leftarrow \underset{j=0\dots m-1}{\texttt{count}} (\textsc{Counter}[j] = 0)$
if $\mathrm{Z} \neq 0$ then
    $n_{\text{lin}} \leftarrow m \cdot \log \frac{m}{\mathrm{Z}}$
    if $n_{\text{lin}} \leq \varkappa_m$ then
        $n \leftarrow n_{\text{lin}}$
return $n$
```

Properties

The accuracy of HyperLogLog++ is better than HyperLogLog for a large range of cardinalities and equally good for the rest. For cardinalities between 12000 and 61000, the bias correction allows for a lower error and avoids a spike in the error when switching between sub-algorithms.

However, since HyperLogLog++ doesn't need to store hash values, just one plus the maximum size of the number of leading zeros, the memory requirements don't grow significantly compared to HyperLogLog and, according to (3.14), it requires only $6 \cdot 2^p$ bits.

The HyperLogLog++ algorithm can be used to estimate cardinalities of about $7.9 \cdot 10^9$ elements with a typical error rate of 1.625%, using 2.56 KB of memory[6].

As mentioned earlier, the algorithm uses the stochastic averaging approach and splits the dataset into $m = 2^p$ subsets $\{\mathrm{S}_j\}_{j=0}^{m-1}$, each of which has associated counters $\{\textsc{Counter}[j]\}_{j=0}^{m-1}$, every counter handles information about $\frac{n}{m}$ elements. Heule et al. noticed that for $n \ll m$ most counters are never used and don't need to be stored, therefore the storage can benefit from a sparse representation. If the cardinality n is much smaller than m, then HyperLogLog++ requires significantly less memory than its predecessors.

The HyperLogLog++ algorithm in a sparse version stores only pairs $(j, \textsc{Counter}[j])$, representing them as a single integer by concatenating their bit patterns. All such pairs are stored in a single sorted list of integers. Since we always compute the maximal rank, we don't need to store different pairs with the same index, instead only the pair with the maximal index has to be stored. In practice, to provide a better experience one can maintain another unsorted list for fast insertions that have to be periodically sorted and merged into the primary list. If such a list requires more memory than the dense representation of the counters, it can be easily converted to the dense form. Additionally, to make

[6]Micha Gorelick and Ian Ozsvald, High Performance Python, 2014

the sparse representation even more space friendly, the HyperLogLog++ algorithm proposes different compression techniques using variable length encoding and difference encoding for the integers, therefore storing only the first pair and differences from its value.

Currently, the HyperLogLog++ algorithm is widely used in many popular applications, including Google BigQuery and Elasticsearch.

Conclusion

In this chapter we covered various probabilistic approaches to counting unique elements in huge datasets. We have discussed the difficulties that appear in cardinality estimation tasks and learned a simple solution that could approximate the small cardinalities quite well. Further, we have studied the family of algorithms based on an observation of certain patterns in the hashed representations of elements from the dataset which is followed by many improvements and modifications that have become industry standard today for estimating cardinalities of almost any range.

If you are interested in more information about the material covered here or want to read the original papers, please take a look at the list of references that follows this chapter.

In the next chapter we consider streaming applications and study the efficient probabilistic algorithms to estimate frequencies of elements, find heavy hitters and trending elements in data streams.

Bibliography

[As87] Astrahan, M.M., Schkolnick, M., Whang, K.-Y. (1987) "Approximating the number of unique values of an attribute without sorting", *Journal Information Systems*, Vol. 12 (1), pp. 11–15, Oxford, UK.

[Du03] Durand, M., Flajolet, P. (2003) "Loglog Counting of Large Cardinalities (Extended Abstract)", In: G. Di Battista and U. Zwick (Eds.) – ESA 2003. *Lecture Notes in Computer Science*, Vol. 2832, pp. 605–617, Springer, Heidelberg.

[Fl85] Flajolet, P., Martin, G.N. (1985) "Probabilistic Counting Algorithms for Data Base Applications", *Journal of Computer and System Sciences*, Vol. 31 (2), pp. 182–209.

[Fl07] Flajolet, P., et al. (2007) "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm", *Proceedings of the 2007 International Conference on Analysis of Algorithms*, Juan les Pins, France – June 17-22, 2007, pp. 127–146.

[He13] Heule, S., et al. (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*, Genoa, Italy —- March 18-22, 2013, pp. 683–692, ACM New York, NY.

[Sc07] Scheuermann, B., Mauve, M. (2007) “Near-optimal compression of probabilistic counting sketches for networking applications”, *Proceedings of the 4th ACM SIGACT-SIGOPS International Workshop on Foundation of Mobile Computing (DIAL M-POMC)*, Portland, Oregon, USA. – August 16, 2007.

[Wh90] Whang, K.-Y., Vander-Zanden, B.T., Taylor H.M. (1990) “A Linear-Time Probabilistic Counting Algorithm for Database Applications”, *Journal ACM Transactions on Database Systems*, Vol. 15 (2), pp. 208–229.

4 Frequency

Many important problems with streaming applications that operate large data streams are related to the estimation of the *frequencies* of elements, including determining the most frequent element or detecting the trending ones over some period of time.

As seen in other problems, when data streams are large enough (they can be seen as an infinite sequence of elements) and have a big number of distinct elements, the usual solutions, like sorting or keeping counters for every element, are not possible anymore. It is also important to note that in most cases it isn't feasible to store and re-process such sequences, therefore *one-pass* data stream algorithms are required.

If the data stream is large but has a low cardinality (contains only a small number of distinct elements), it is enough to maintain exact frequency counters using one counter per distinct element, in these cases, there is no need for special algorithms.

The specificity of Big Data applications that handle large data streams requests that appropriate data structures and algorithms fulfill the following requirements:

- Make one pass through the data.
- Have sublinear space (polylogarithmic at most), meaning they don't grow as fast as the input stream does.
- Support fast and simple updates with some guarantee of accuracy.

Because of the space restrictions, it is clear that such structures need to operate data in a compressed form that is some summary of the data stream (e.g., sketch) and makes it not possible to compute most functions over the stream precisely, therefore probabilistic approximation is needed.

Let's start with formal definitions. By *data stream* $\mathbb{D} = \{x_1, x_2, \ldots, x_n\}$ we mean a sequence of elements of any nature, assuming that the number of elements n is very large, e.g., billions, and there is an unknown large number of distinct elements. If the stream is truly infinite, $\mathbb{D}$ can be seen as a substream if viewed in a time window.

With an approach to estimating frequencies of elements in a huge data stream, we can address the common problem of finding the list of high-frequency elements in a stream, known as the *Frequent problem*.

When we look for an element that occurs more than $\frac{n}{2}$ times in data stream $\mathbb{D}$, we consider the *Majority problem* that was formulated as a research problem by J. Strother Moore in the Journal of Algorithms in 1981 [Bo81]. We can postulate that such an element exists in the stream, which is not always true, and, by definition, it is clear that it can be only one element for the given data stream, which is called the *majority element*.

Obviously, this is mostly a toy problem, but it gives us a better understanding of the frequency problems involved with data streams.

One of the most complex problems that appears in practice while working with big data is the problem of finding the top k most frequent elements in the stream that occur more than $\frac{n}{k}$ times, also known as *heavy hitters*, where $k \ll n$ these are usually 10, 100 or 1000.

Searching for heavy hitters presupposes that some elements can occur significantly more often than others in the data stream, otherwise there is no sense in solving this problem.

This might be surprising, but it was proven that there is no algorithm that can solve the *Heavy hitters problem* in one pass using a sublinear space.

Many practical applications are connected to the Heavy hitters problem, including search, log mining, network analysis, traffic engineering, and anomaly detection. For instance, we might want to determine the heaviest k users (for a desired value of $k \ll n$) for a high-traffic website. However, some users may have nearly equal load and getting an exact answer to this question is impossible using limited space.

In practice, we consider the ε–Heavy hitters problem — an ε-approximation of the Heavy hitters problem, that results in elements that occur at least $\frac{n}{k}$ time with a guaranteed occurrence at least $\frac{n}{k} - \varepsilon \cdot n$, where $\varepsilon > 0$ and is small. For instance, for $\varepsilon = \frac{1}{2k} > 0$ the output of the ε–Heavy hitters problem will be elements with frequencies at least $\frac{n}{k}$ and a guarantee that they occur at least $\frac{n}{k} - \varepsilon \cdot n = \frac{n}{k} - \frac{n}{2k} = \frac{n}{2k}$ times.

For small data streams (regardless of the number of unique elements) it is enough to just sort the elements and, using a linear scan, find elements that occur at least $\frac{n}{k}$ these will be the heavy hitters.

In an arbitrary data stream $\mathbb{D}$, there are from zero to k heavy hitters and, in contrast to the Majority problem, it is much more likely that for some k at least one heavy hitter exists, while the majority element doesn't. Therefore, the Majority problem could be seen as a particular case of the Heavy hitters problem with the requirement that such a majority element exists and $k \approx 2 - \varepsilon$, where $\varepsilon > 0$ and is small.

Example 4.1: DNS DDoS attack detection (Afek et al., 2016)

A distributed denial-of-service (DDoS) attack includes many systems flooding the resources of the targeted system, typically by sending a large number of queries from a botnet. One popular target is the Domain Name System (DNS) that plays the role of a "phonebook" of the Internet, providing the translation between easy to remember domain names and IP addresses of websites.

DNS queries are considered a data stream where each element has an associated domain to resolve. Going further we can group the queries using their top-level domain and by investigating the heaviest domains in the query stream we can detect the randomized DNS Flood when queries for many different non-existent subdomains of the same primary domain are issued.

Another interesting task in streaming applications, called the *Max–Change problem*, is to determine elements whose frequencies changed the most across different data streams or time windows. This problem has a practical importance for search engines since the queries whose frequency changes most between two consecutive time periods can indicate which topics are increasing or decreasing in popularity at the fastest rate.

Example 4.2: Trending Twitter hashtags

A hashtag is used to index a topic on Twitter and allows people to easily follow items they are interested in. Hashtags are usually written with a # symbol in front.

Every second about 6000 tweets[1] are created on Twitter, that is roughly 500 billion items daily. Most of these tweets are linked with one or more hashtags and to keep abreast of all the latest events it is important to determine the most popular topics of the day.

This can be done by processing the data stream of tweets, estimating the frequencies of each hashtag, and finding the most frequent values. Additionally, it might be useful to compare the frequencies of yesterday's and today's values to determine the topics that are trending, e.g., those that have the most increased frequencies since yesterday.

Here we study various approaches to solve frequency-related problems in Big Data streams. We start with very simple deterministic algorithms and afterward learn modern probabilistic alternatives that can efficiently address real-world problems.

4.1 Majority algorithm

Without any additional investigation, it is possible to suggest a linear-time solution for the Majority problem because the majority element (of course, with the assumption that it exists) is the median. The disadvantage is that it requires multiple passes through the stream and, therefore, is not suitable for Big Data streams.

The *Majority algorithm*, also known as the *Boyer–Moore Majority Vote algorithm*, was invented by Bob Boyer and J. Strother Moore in 1981 [Bo81] to solve the Majority problem in a single pass through the data stream. A similar solution was independently proposed by Michael J. Fischer and Steven L. Salzberg in 1982 [Fi82].

The data structure for the Majority algorithm is fairly simple and is a pair made up of an integer counter and the so-called *monitored element*: $\mathrm{S} = (c, x^*)$. Therefore, it requires a constant amount of memory, but its size varies and depends on the size of the elements.

Such a data structure supports only one simple update operation,

[1]Twitter Usage Statistics `https://www.internetlivestats.com/twitter-statistics/`

which updates the counter and selects the candidate for the monitored element based on its previous state and current element x.

Algorithm 4.1: Updating the Majority data structure

```
Input: Element $x \in \mathbb{D}$
if $c = 0$ then
    $x^* \leftarrow x$
if $x = x^*$ then
    $c \leftarrow c + 1$
else
    $c \leftarrow c - 1$
```

With this type of data structure, it is simple to describe the algorithm. For every element x in the stream $\mathbb{D}$ the algorithm triggers the update procedure given by Algorithm 4.1 and, under the requirement that the majority element exists, it returns the last monitored element as the majority element. Note, that the value of the counter is **not** the frequency of the majority element.

Algorithm 4.2: MAJORITY ALGORITHM

```
Input: Data stream $\mathbb{D}$
Output: Majority element
$c \leftarrow 0$
$x^* \leftarrow$ NULL
for $x \in \mathbb{D}$ do
    UPDATE($x$)
return $x^*$
```

In the Majority algorithm, every "non-majority" value that follows can decrease the counter c or even reset it to 0, which forces the re-election of the monitored element x^*. From a non-precise view, it might be unclear how such an algorithm ends up with the correct value and whether there is a danger that for some cases all majority values could be eliminated.

The following "non-majority" values can wipe out only one copy of the previous majority element, but as there has to be more than $\frac{n}{2}$ majority values in the data stream, there will not be enough "non-majority" values and at least one copy of the majority value will be left at the end. This also explains why the returned value of the counter cannot be used as an approximation of the majority element's frequency.

When the majority element doesn't exist, the output of the Majority algorithm is an arbitrary element of the data stream. Therefore, applying such an algorithm when we are uncertain about the existence of the majority element would require another pass through the data stream with a simple counter to verify that the element given by Algorithm 4.2 is actually the majority element that occurs more then $\frac{n}{2}$ times.

Example 4.3: Majority algorithm

Consider a dataset of $n = 10$ elements: $\{4, 4, 3, 5, 6, 4, 4, 4, 4, 2\}$, where the obvious majority element is $x = 4$ as it occurs 6 times out of 10.

According to the algorithm, we allocate a pair $\mathrm{S} = (c, x^*) = (0, \mathrm{NULL})$ and start consuming elements from the dataset. The first element is $x_1 = 4$ and, since our counter c is empty, we store it as the monitored element $x^* = 4$ and increase the counter: $c = 1$. The next element x_2 is 4 again, which is equal to the monitored element, so we just increase the counter: $c = 2$. The third input element is $x_3 = 3$ that is different from the $x^* = 4$, thus we decrement our counter: $c = 1$. Similarly, after processing $x_4 = 5$, we decrement the counter again and it becomes zero: $c = 0$.

Next, we process element $x_5 = 6$ and, since the current counter value is zero, we update the monitored element $x^* = 6$ and set its counter: $c = 1$. However, it doesn't stay too long and after handling elements $x_6 = 4$ and $x_7 = 4$, the monitored element becomes $x^* = 4$ again with counter $c = 1$. The next element will increment counter $c = 2$, but the last element is not equal to 4 and the counter will be decremented again to the value $c = 1$.

In the end, the correct majority element 4 is left as the monitored element. However, note that the remaining counter c is not a frequency estimator and contains a completely different value.

The Majority algorithm is the most popular algorithm for undergraduate classes due to its simplicity. In the next section we study its extention that can already solve the Frequent and Heavy hitters problems.

4.2 Frequent algorithm

A generalization of the Majority algorithm, known as the *Frequent algorithm*, was proposed by Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro in 2002 [De02], years after the original algorithm. At some point, it was discovered that the algorithm was actually the same as the algorithm published by Jayadev Misra and David Gries in 1982 [Mi82], known now as the *Misra–Gries algorithm.*

The Frequent algorithm is designed to address the Heavy hitters problem and instead of keeping only one counter like in the Majority algorithm, the Frequent data structure consists of a set of monitored elements X^* and an array of p counters $\mathrm{C} = \{c_i\}_{i=1}^{p}$.

Whenever we process a new element from the data stream, we first check if it is already monitored. However, if the element is new, we add it into the X^* only when we have room in the set since we maintain at most p elements. If the element wasn't added, we still want to reflect its presence in the stream by decrementing the counters of all elements in the set of monitored elements. When the element already exists in the X^*, we just increment its associated counter. At the end of the procedure, we go through the list and pop up all elements whose counters hit zero.

In the original article, Misra and Gries used balanced search trees to represent the Frequent data structure, however, future researchers preferred to use hash tables and implement it as a dictionary.

Algorithm 4.3: Updating the Frequent data structure

```
Input: Element $x \in \mathbb{D}$
Input: Frequent data structure with $p$ counters
if $x \notin X^*$ then
    if $\exists m : c_m = 0$ then
        $x^*_m \leftarrow x$
if $x \in X^*$ then
    $\exists m : x^*_m = x$
    $c_m \leftarrow c_m + 1$
else
    for $j \leftarrow 1$ to p do
        if $c_j > 0$ then
            $c_j \leftarrow c_j - 1$
for $j \leftarrow 1$ to p do
    if $c_j = 0$ then
        $X^* \leftarrow X^* \setminus \{x^*_j\}$
```

The Frequent algorithm uses the FREQUENT data structure of length p to discover elements that occur at least $\frac{n}{p+1}$ times in the data stream of length n. Thus, to determine up to $k-1$ heavy hitter elements that occur at least $\frac{n}{k}$ times in the data stream we need to use $p = k - 1$ counters.

Algorithm 4.4: Frequent algorithm

```
Input: Data stream $\mathbb{D}$
Input: Frequent data structure with $k - 1$ counters
Output: Heavy hitters elements
$C := \{c_i\}_{i=1}^{k-1}, c_i \leftarrow 0$
$X^* \leftarrow \varnothing$
for $x \in \mathbb{D}$ do
    Update($x$)
return $X^*$
```

The intuition behind the Frequent algorithm is very similar to the Majority algorithm given the requirement that heavy hitter elements occur more than $\frac{n}{k}$ times.

Example 4.4: Find heavy hitters with the Frequent algorithm

Consider a data stream of $n = 18$ elements:

$$\{4, 4, 4, 4, 6, 2, 3, 5, 4, 4, 3, 3, 4, 2, 3, 3, 3, 2\}.$$

To identify heavy hitter elements that occur in the data stream at least $\frac{n}{3} = 6$ times, we allocate a Frequent data structure of $p = 2$ counters and use Algorithm 4.4 to identify at most two of three possible heavy hitters.

	1	2
X^*		
C	0	0

We start with element 4. Since it isn't in the X^* and there are no elements in the data structure, we freely add element 4 in the set of monitored elements and increment the associated counter $c_1 = 1$.

	1	2
X^*	4	
C	1	0

Similarly, we process the next three elements that are also equal to 4. Because this element is already in the X^*, we simply increment its counter c_1.

	1	2
X^*	4	
C	4	0

The next element is 6 which is not monitored yet. Since the set X^* has space, we insert element 6 into it and set the counter $c_2 = 1$.

	1	2
X^*	4	6
C	4	1

Next, we take element 2 which is also not in the X^*, however we have no room in the set and cannot add it. Otherwise, we decrement counters of all elements currently in the X^*. According to Algorithm 4.4 we also need to remove from the monitored set elements whose counters hit zero. In our example, this is element 6 which is therefore removed from the set.

	1	2
X^*	4	
C	3	0

Next, we consume element 3 from the data stream. This element is not in the X^* and since the set has enough space, we just add it in and set the associated counter $c_2 = 1$.

	1	2
X^*	4	3
C	3	1

Continuing in a similar way, we process all remaining elements and the final data structure becomes:

	1	2
X^*	4	3
C	3	3

Thus, the identified heavy hitters are elements 4 and 3. However, the counters do not reflect the actual frequencies on the elements in the data stream, as we also noted for the Majority algorithm.

Properties

The time cost of the algorithm is dominated by the O(1) dictionary operations per update and the cost of decrementing counts. To optimize the speed of the algorithm, all counters can be decremented at once, in constant time by organizing them in a sorted order and using *difference encoding*, where the only information stored is about how much larger the particular counter is compared to the next smallest one. Minimizing significant movements in the order while incrementing and decrementing the counter means all equal counters can be grouped. With such an optimized data structure each counter no longer needs to store a value, but rather its group. Thus, the Frequent algorithm can be augmented to run in O(1) time.

Actually, even without any probabilistic approach, the algorithm delivers at most $k-1$ candidates in the Heavy hitters problem. However, it is focused on the determining the high-frequency elements without their correct frequencies approximations. Therefore, if we want to estimate frequencies of the elements, the second pass through the data stream is required which is unfeasible in most cases of handling huge data streams.

Thus, in the next sections we continue studying solutions for frequency-related problems with very effective probabilistic data structures that are perfectly suited to Big Data streams.

4.3 Count Sketch

A space-efficient algorithm that is used to solve many frequency-related data stream problems is the *Count Sketch* that was proposed by Moses Charikar, Kevin Chen, and Martin Farach-Colton in 2002 [Ch02]. They had a practical requirement to create a space-efficient data structure that could easily maintain approximate counts of high-frequency elements in a data stream.

In order to better understand the problem that Count Sketch solves we note that the idea of a Counting Bloom filter could also be used to compute frequencies of elements in a data stream, however, it is not enough to build precise frequency estimators.

Consider a data structure with an array $\mathrm{C} = \{c^i\}_{i=1}^m$ of m counters and p hash functions $h_1, h_2, \ldots, h_p$ that map from elements to $\{1, 2, \ldots, m\}$. The indexing of element x from the data stream into such a data structure, like with the Counting Bloom filter, includes computing $\{h_j(x)\}_{j=1}^p$ and incrementing the corresponding counters $c^{h_j(x)}, j = 1 \ldots p$ in the array.

When we need to find a frequency $f(x)$ of element x, we simply compute the values of each hash function for that element and get values of the corresponding counters $c^1, c^2, \dots, c^m$ that play the role of frequency estimations.

However, because the counters are never decremented and the hash functions use the same array, it is clear that such estimations will be bigger than the real frequency $f(x)$ of the element:

$$f(x) \le c^i, i = 1 \dots m,$$

the inequality is the result of possible hash collisions when different elements update the same counters. In other words, the one-sided error common to our estimations making them all upper bound estimates.

The idea of Count Sketch is to solve this problem by building lower bound as well as the upper bound estimations. To prevent situations when collisions with high-frequency elements spoil most estimates of lower frequency elements, this requires a random decision when to decrease the counter and when to increment it. In order to reduce the variance it additionally takes the median of those estimations.

The CountSketch data structure designed to store the frequencies of m high frequency elements consists of a $p \times m$ array of counters $\{c^i_j\}$ that can be seen as an array of p hash tables, each of m buckets. Additionally, it uses p hash functions $h_1, h_2, \dots, h_p$ that map from elements to $\{1, 2, \dots, m\}$ and p hash functions $s_1, s_2, \dots, s_p$ that map from elements to $\{+1, -1\}$ in order to support both side approximation to the real frequency value. It is assumed that hash functions h_i and s_i are pairwise independent and independent of each other.

The data structure allows counters to be updated for each indexed element and estimates the number of times the element has been seen in the past, this is used as the frequency estimation for the element. Every time we index a new element x, the counters $c_j^{h_j(x)}$ for each row j of the sketch can be either incremented or decremented, based on the values of $s_j(x)$. Therefore, it is possible that the counters overestimate the frequency of the element x, as well as underestimate it.

Algorithm 4.5: Updating the Count sketch

```
Input: Element $x \in \mathbb{D}$
Input: Count sketch with $p \times m$ counters
for $j \leftarrow 1$ to $p$ do
    $i \leftarrow h_j(x)$
    $c_j^i \leftarrow c_j^i + s_j(x) \cdot 1$
```

Assuming that every hash function $\{h_j\}_{j=1}^p$ and $\{s_j\}_{j=1}^p$ can be computed in a constant time, the running time for the update procedure given by Algorithm 4.5 is $O(p)$.

Example 4.5: Build Count sketch

Consider a dataset of $n = 18$ elements:

$$\{4, 4, 4, 4, 2, 3, 5, 4, 6, 4, 3, 3, 4, 2, 3, 3, 3, 2\}$$

and let's build a CountSketch data structure of $m = 5$ counters using $p = 3$ hash functions based on MurmurHash3, FNV1a and MD5 to decide which counter to update:

$$h_1(x) := \text{MurmurHash3}(x) \bmod 5 + 1,$$
$$h_2(x) := \text{FNV1a}(x) \bmod 5 + 1,$$
$$h_3(x) := \text{MD5}(x) \bmod 5 + 1,$$

and three hash functions to determine the direction of the update:

$$s_1(x) := \text{MurmurHash3}(x) \bmod 2 \ ? -1 : 1,$$
$$s_2(x) := \text{FNV1a}(x) \bmod 2 \ ? -1 : 1,$$
$$s_3(x) := \text{MD5}(x) \bmod 2 \ ? -1 : 1.$$

In the beginning, the CountSketch data structure consists of zeros:

	1	2	3	4	5
h_1	0	0	0	0	0
h_2	0	0	0	0	0
h_3	0	0	0	0	0

We start processing elements from the dataset and the first element is 4. According to Algorithm 4.5, we compute its hash values $h_1(4)$, $h_2(4)$, and $h_3(4)$ to determine counters that have to be updated:

$$i_1 = h_1(4) = 3,\ \ i_2 = h_2(4) = 3,\ \ i_3 = h_2(4) = 1.$$

In this case, two hash functions deliver the same value, but since we maintain dedicated lists of counters for each hash function this is not a problem. To determine the direction of updates we compute hash values $s_1(4)$, $s_2(4)$, and $s_2(4)$:

$$s_1(4) = 1,\ \ s_2(4) = 1,\ \ s_2(4) = -1.$$

Thus, we increment counters c_1^3 and c_2^3, while decrementing counter c_3^1. The resulting CountSketch data structure becomes as follows.

	1	2	3	4	5
h_1	0	0	1	0	0
h_2	0	0	1	0	0
h_3	-1	0	0	0	0

The next three elements are 4 too, hence we increment or decrement the same counters three more times:

	1	2	3	4	5
h_1	0	0	4	0	0
h_2	0	0	4	0	0
h_3	-4	0	0	0	0

Next element in the dataset is 2 and its corresponding indices are $i_1 = 3$, $i_2 = 2$, and $i_3 = 3$. The values of direction hash functions are $s_1(2) = 1$, $s_2(2) = 1$, and $s_3(2) = -1$, so we increment counters c_1^3 and c_2^2, and decrement c_3^3. Note, that there is a soft collision and element 2 changes (in the same direction) the counter used by element 4. This makes the value in the counter c_1^3 overestimate the real value for both elements.

	1	2	3	4	5
h_1	0	0	5	0	0
h_2	0	1	4	0	0
h_3	-4	0	-1	0	0

In the same way, we process all remaining elements. For element 3 we decrement counters c_1^1 and c_2^3, and increment counter c_3^4; for element 5 we decrement counters c_1^3 and c_2^4, and increment c_3^4; for element 6 we decrement counters c_1^4 and c_3^3, and increment c_2^1.

The final CountSketch has the following form:

	1	2	3	4	5
h_1	-6	0	9	-1	0
h_2	1	3	1	-1	0
h_3	-7	0	-4	7	0

It is known in probability theory that the usual procedure for building better approximations from a number of randomly distributed trials is to use the mean and median. The Count Sketch algorithm, to compute the final estimation of the frequency, uses the median because it is robust and less sensitive to outliers.

Algorithm 4.6: Estimating frequency with the Count Sketch

```
Input: Element $x \in \mathbb{D}$
Input: Count–Min sketch with $p \times m$ counters
Output: Frequency estimation
$\hat{f} := \{\hat{f}_j\}_{i=1}^{p}$
for $j \leftarrow 1$ to $p$ do
    $i \leftarrow h_j(x)$
    $\hat{f}_j \leftarrow s_j(x) \cdot c_j^i$
return median$(\hat{f}_1, \hat{f}_2, \ldots, \hat{f}_p)$
```

The update time for each element is $O(p)$ and to find the median of p elements we spend some linear time using one of the selection algorithms, therefore the overall query time is $O(p)$.

Example 4.6: Frequency estimation with Count Sketch

Consider the data structure that we built in Example 4.5:

	1	2	3	4	5
h_1	-6	0	9	-1	0
h_2	1	3	1	-1	0
h_3	-7	0	-4	7	0

Let's estimate the frequency of element 4 whose corresponding counters are c_1^3, c_2^3, and c_3^1, the update directions are $s_1(4) = 1$, $s_2(4) = 1$, and $s_2(4) = -1$, as we determined earlier. Using Algorithm 4.6, as the estimation we calculate the median of weighted values of those counters:

$$\hat{f} = \text{median}(s_1(4) \cdot c_1^3, s_2(4) \cdot c_2^3, s_3(4) \cdot c_3^1) = \text{median}(9, 1, 7) = 7.$$

Thus, the estimated frequency of element 4 is 7, that is also the correct count from the dataset.

Now, consider element 2 with the corresponding counters c_1^3, c_2^2, and c_3^3 with values of direction hash functions $s_1(2) = 1$, $s_2(2) = 1$, and $s_3(2) = -1$.

Thus, the frequency estimation for element 2 is

$$\hat{f} = \text{median}(s_1(2) \cdot c_1^3, s_2(2) \cdot c_2^2, s_3(2) \cdot c_3^3) = \text{median}(9, 3, 4) = 4,$$

which overestimates the real value 3.

The Count Sketch algorithm can be used to find the top k most frequent elements, known as the Frequent problem. In a single pass through the data stream, as well as the regular $p \times m$ array of counters and the hash functions $\{h_j\}_{j=1}^p$ and $\{s_j\}_{j=1}^p$, we maintain a set X^* of k high-frequency elements. We first index every element x from the data stream to the CountSketch data structure according to Algorithm 4.5. Then, if the element is not in the set X^* and there is capacity to add it, we insert the element. Otherwise, we estimate frequency with Algorithm 4.6 and if it is greater than the smallest frequency in the set, we add element x to X^* while removing the element with the smallest frequency.

Algorithm 4.7: Getting frequent elements with the Count Sketch

```
Input: Data stream $\mathbb{D}$
Input: Count–Min sketch with $p \times m$ counters
Output: Top frequent elements
$X^* \leftarrow \varnothing$
for $x \in \mathbb{D}$ do
    Update($x$)
    if $x \in X^*$ then
        continue
    if $|X^*| < k$ then
        $X^* \leftarrow X^* \cup \{x\}$
    else
        $\hat{f} \leftarrow$ Frequency($x$)
        $(x^*_{\min}, \hat{f}^*_{\min}) \leftarrow \min_{x^* \in X^*} (\textbf{Frequency}(x^*))$
        if $\hat{f} > \hat{f}^*_{\min}$ then
            $X^* \leftarrow X^* \cup \{x\} \setminus \{x^*_{\min}\}$
return $X^*$
```

Example 4.7: Most frequent elements with Count Sketch

Consider the same setup as in Example 4.5 and search for $k = 3$ most frequent elements in the dataset:

$$\{4, 4, 4, 4, 2, 3, 5, 4, 6, 4, 3, 3, 4, 2, 3, 3, 3, 2\}.$$

According to Algorithm 4.7, additional to the CountSketch data structure, we create a set X^* to store frequent candidates.

We start consuming the dataset and the first element is 4, so, as we know from Example 4.5, we need to increment counters c_1^3 and c_2^3, while decrementing counter c_3^1.

	1	2	3	4	5
h_1	0	0	1	0	0
h_2	0	0	1	0	0
h_3	-1	0	0	0	0

The set X^* is empty, thus we freely insert element 4 into it: $\mathrm{X}^* = [4]$.

The next three elements in the data stream are also equal to 4, so we index them into the data structure without any changes to X^*.

	1	2	3	4	5
h_1	0	0	4	0	0
h_2	0	0	4	0	0
h_3	-4	0	0	0	0

The next element is 2 and we increment counters c_1^3 and c_2^2, and decrement c_3^3, as we determined earlier. This element is not in the set of the most frequent candidates and since X^* has enough capacity, we add element 2 into the set: $\mathrm{X}^* = [4, 2]$.

	1	2	3	4	5
h_1	0	0	5	0	0
h_2	0	1	4	0	0
h_3	-4	0	-1	0	0

The next input element is 3. To index it into the CountSketch data structure we decrement counters c_1^1 and c_2^3, and increment counter c_3^4. Since the set X^* contains only two elements out of three possibles, we add element 3 into the set: $\mathrm{X}^* = [4, 2, 3]$.

	1	2	3	4	5
h_1	-1	0	5	0	0
h_2	0	1	3	0	0
h_3	-4	0	-1	1	0

Next, we take element 5 from the dataset and update the sketch by decrementing the counters c_1^3 and c_2^4, and incrementing c_3^4.

	1	2	3	4	5
h_1	-1	0	4	0	0
h_2	0	1	3	-1	0
h_3	-4	0	-1	2	0

Element 5 is not in the set X^* which has reached its maximum capacity of $k = 3$ monitored elements. Thus, we need to estimate the frequencies of elements in the set and of element 5 using Algorithm 4.6 for the current CountSketch data structure.

$$\begin{aligned}
\hat{f}(5) &= \text{median}(-c_1^3, -c_2^4, c_3^4) = \text{median}(-4, 1, 2) = 1,\\
\hat{f}(4) &= \text{median}(c_1^3, c_2^3, -c_3^1) = \text{median}(4, 3, 4) = 4,\\
\hat{f}(2) &= \text{median}(c_1^3, c_2^2, -c_3^3) = \text{median}(4, 1, 1) = 1,\\
\hat{f}(3) &= \text{median}(-c_1^1, -c_2^3, c_3^4) = \text{median}(1, -3, 2) = 1.
\end{aligned}$$

Therefore, the estimated frequency of the current element 5 doesn't exceed the minimum frequency of elements in the set, so we don't change the set of monitored elements: $X^* = [4, 2, 3]$.

In a similar manner we handle all remaining elements from the dataset and after processing the last one, the CountSketch data structure has the following form:

	1	2	3	4	5
h_1	-6	0	9	-1	0
h_2	1	3	1	-1	0
h_3	-7	0	-4	7	0

and the set of the three most frequent elements is

$$X^* = [4, 2, 3].$$

Note, that the most frequent elements in X^* are not ordered and to estimate their frequencies we can use Algorithm 4.6.

In the same way we can address the Heavy hitters problem. To find k heavy hitters we maintain a counter N of already processed elements, using which, we calculate the frequency threshold $f^* = \frac{N}{k}$ every time a new element is indexed. If the estimated frequency of the current element is above the threshold, we insert it into the heap X^* as a candidate for heavy hitters. Additionally, on every step we remove elements from the heap whose stored frequency fall below the actual threshold f^*.

Algorithm 4.8: Determining heavy hitters with the Count Sketch

```
Input: Data stream $\mathbb{D}$
Input: Count sketch with $p \times m$ counters
Output: Heavy hitters
$\mathrm{N} \leftarrow 0$, $\mathrm{X}^* \leftarrow \varnothing$
for $x \in \mathbb{D}$ do
    $\mathrm{N} \leftarrow \mathrm{N} + 1$
    $\mathbf{Update}(x)$
    $\hat{f} \leftarrow \mathbf{Frequency}(x)$
    $f^* \leftarrow \frac{\mathrm{N}}{k}$
    if $\hat{f} \geq f^*$ then
        $\mathrm{X}^* \leftarrow \mathrm{X}^* \cup \{(x, f)\}$
    for $(x^*, \hat{f}) \in \mathrm{X}^*$ do
        if $\hat{f} \leq f^*$ then
            $\mathrm{X}^* \leftarrow \mathrm{X}^* \setminus \{(x^*, \hat{f})\}$
return $\mathrm{X}^*$
```

The Count Sketch algorithm can also be applied to finding elements with the largest frequency change, otherwise known as the Max-Change Problem. Having data streams of two comparable periods, we can build a CountSketch data structure for each of them and maintain the heap X^* of elements with the largest differences. Every time new elements are indexed, we estimate their frequencies using Algorithm 4.6 and update the heap to keep only elements with the most change. Finally, the algorithm outputs k elements with the largest values of frequency change.

Properties

The Count Sketch provides the guarantee that the estimation error for frequencies is not bigger than $\varepsilon \cdot n$ with probability at least $1 - \delta$. The increasing number of hash functions p decreases the probability of a bad estimate and for the desired standard error δ the recommendation on the number of hash functions, that correspond to the rows in the CountSketch, is

$$p = \left\lceil \ln \frac{1}{\delta} \right\rceil. \qquad (4.1)$$

The bigger m, the less likely that collisions will happen, meaning a lower estimation error $\varepsilon \cdot n$. At the same time, with bigger p more estimators are used to calculate the final value, which makes it more reliable. The recommendation on the number of counters m is

$$m \approx \left\lceil \frac{2.71828}{\varepsilon^2} \right\rceil. \qquad (4.2)$$

The overall space required by the Count Sketch data structure is $O(m \cdot p + 2p)$, because we keep a count matrix sized $p \times m$ and two hash functions per row.

If two Count Sketch data structures have the same size m, they can be easily added to and subtracted from each other, this is useful for distributed stream processing.

There are implementations of Count Sketch for Apache Hive and other data warehouse software, but modern applications prefer to use its successor, the Count–Min Sketch algorithm, due to it requiring less space and execution time.

4.4 Count–Min Sketch

Count–Min Sketch is a simple space-efficient probabilistic data structure that is used to estimate frequencies of elements in data streams and can address the Heavy hitters problem. It was presented in 2003 [Co03] by Graham Cormode and Shan Muthukrishnan and published in 2005 [Co05].

As we saw in the previous section for Count Sketch, the main obstacle to the direct application of the Counting Bloom filter in frequency estimation tasks is that it shares a single array of counters for all hash functions and, consequently, suffers from hard and soft collisions.

The quality of the estimation is hardly affected by the probability of hash collisions even though they lead to overestimations for counters. However, when the number of elements in the data stream is huge, collisions with high-frequency elements are almost certain and this makes such an approximation useless due to the large overestimation of all counters.

By treating this problem as a lack of highly confident estimates to compute frequency with sufficient precision, the Count–Min Sketch algorithm replaces the single array of m counters with a hash table of p arrays of m counters and, instead of updating each counter by every element, lets the elements update different subsets of counters, one per hash table. The purpose of m is to compress the data stream $\mathbb{D} = \{x_1, x_2, \ldots, x_n\}$ and because $m \ll n$ this is a “lossy” compression that leads to errors. To reduce these errors, the algorithm introduces many independent trials by using p hash functions with a dedicated array of m counters for each.

The CMSketch is a space-efficient data structure that consists of a $p \times m$ array of counters $\{c_j^i\}$, where p pairwise independent hash functions $h_1, h_2, \ldots, h_p$ map the universe to the range $\{1, 2, \ldots, m\}$.

Such a simple data structure allows for the indexing of elements from the data stream, results in updating counters, and can provide the number of times every particular element has been indexed, which can be seen as the frequency estimation for the element.

Algorithm 4.9: Updating the Count–Min sketch

```
Input: Element $x \in \mathbb{D}$
Input: Count–Min sketch with $p \times m$ counters
for $j \leftarrow 1$ to $p$ do
    $i \leftarrow h_j(x)$
    $c_j^i \leftarrow c_j^i + 1$
```

Assuming that every hash function $\{h_j\}_{j=1}^{p}$ can be computed in a constant time, the running time for the update procedure given by Algorithm 4.9 is $\mathrm{O}(p)$.

Example 4.8: Build Count–Min sketch

Consider the dataset of $n = 18$ elements from Example 4.5:

$$\{4, 4, 4, 4, 2, 3, 5, 4, 6, 4, 3, 3, 4, 2, 3, 3, 3, 2\}$$

and let's build a Count–Min sketch of $m = 4$ counters using $p = 2$ hash functions based on MurmurHash3 and FNV1a:

$$h_1(x) := \text{MurmurHash3}(x) \bmod 4 + 1,$$
$$h_2(x) := \text{FNV1a}(x) \bmod 4 + 1.$$

In the beginning, the CMSketch data structure consists of zeros:

	1	2	3	4
h_1	0	0	0	0
h_2	0	0	0	0

We start consuming elements from the dataset. The first element is 4 and, according to Algorithm 4.9, we compute its hash values to determine counters that have to be updated:

$$i_1 = h_1(4) = 4,$$
$$i_2 = h_2(4) = 4.$$

Note that both hash functions deliver the same value, but since we maintain dedicated arrays of counters for each hash function this is not a problem. Thus, we increment counters c_1^4 and c_2^4 and the CMSketch is as below.

	1	2	3	4
h_1	0	0	0	1
h_2	0	0	0	1

The next three elements are all equal to 4, hence we update the same counters:

	1	2	3	4
h_1	0	0	0	4
h_2	0	0	0	4

Next element in the dataset is 2 and its corresponding indices are $i_1 = 4$ and $i_2 = 1$, so we increment counters c_1^4 and c_2^1. Note that there is a soft collision and element 2 changes the counter that is also used by element 4. This makes the value in the counter c_1^4 overestimate the real value for both elements.

$$\begin{array}{c|c|c|c|c|}
 & 1 & 2 & 3 & 4 \\
\hline
h_1 & 0 & 0 & 0 & 5 \\
\hline
h_2 & 1 & 0 & 0 & 4 \\
\hline
\end{array}$$

In the same way, we process all remaining elements and update counters c_1^1 and c_2^3 for element 3, counters c_1^1 and c_2^1 for element 5, and c_1^1 and c_2^2 for element 6. Note, that both counters for element 6 collide with other elements, thus we can expect that its value will be overestimated.

The final CMSKETCH data structure has the following form:

$$\begin{array}{c|c|c|c|c|}
 & 1 & 2 & 3 & 4 \\
\hline
h_1 & 8 & 0 & 0 & 10 \\
\hline
h_2 & 1 & 4 & 6 & 7 \\
\hline
\end{array}$$

Every time element x is indexed, the same counters $c_j^{h_j(x)}$ are incremented for each row j of the sketch and since they are never decremented, those counters provide the upper bound for the frequencies:

$$f(x) \leq c_j^{h_j(x)}, j = 1, 2, \ldots, p.$$

While counters cannot underestimate the real frequency $f(x)$, they generally overestimate it because $m \ll n$ and there are a lot of collisions such that $h_j(x) = h_j(y)$ for $x \neq y$, meaning that when element y is indexed into CMSKETCH the counter for element x is also incremented.

As a result, there are p estimations that suffer from a one-sided error (all of them are overestimations of the real value). The usual procedure for building a better approximation from a number of estimations is averaging, but this error can make the estimation even worse. Obviously, the best estimation in this case is the smallest one.

Algorithm 4.10: Estimating frequency with Count–Min Sketch

```
Input: Element $x \in \mathbb{D}$
Input: Count–Min sketch with $p \times m$ counters
Output: Frequency estimation
$\hat{f} := \{\hat{f}_j\}_{i=1}^{p}$
for $j \leftarrow 1$ to $p$ do
    $i \leftarrow h_j(x)$
    $\hat{f}_j \leftarrow c_j^i$
return $\min(\hat{f}_1, \hat{f}_2, \ldots, \hat{f}_p)$
```

The minimum of p elements can be found in linear time, and therefore the running time of the frequency estimation procedure given by Algorithm 4.10 is $O(p)$, the same as for an update.

Example 4.9: Frequency estimation with the Count–Min Sketch

Consider the CMSketch data structure that we built in Example 4.9:

	1	2	3	4
h_1	8	0	0	10
h_2	1	4	6	7

Let's estimate the frequency of element 4 whose corresponding counters are c_1^4 and c_2^4, as we determined earlier. Using Algorithm 4.10, as the estimation we calculate the minimum of those counters:

$$\hat{f} = \min(c_1^4, c_2^4) = \min(10, 7) = 7.$$

Thus, the estimated frequency of element 4 is 7, that is also the correct count from the dataset.

Now, consider element 6 with corresponding counters c_1^1 and c_2^2. However, as we already noted in Example 4.9, both of them are also used by other elements due to collisions. Thus, the frequency estimation for element 6 is

$$\hat{f} = \min(c_1^1, c_2^2) = \min(8, 4) = 4,$$

that significantly overestimates the real value of 1. If we want to maintain better accuracy and make such collisions rare, we need to have more hash functions and counters that increase the computational time and storage.

Knowing how to estimate the frequencies of elements, lets the Count–Min Sketch algorithm determine the most frequent elements. Similar to the Count Sketch, the simplest approach requires the maintenance of a set of candidates for the most frequent elements as well as the main CMSketch data structure. Then, we go over the data stream and update the sketch with all the elements seen thus far. If the element is not in the set that still contains less than k elements, we simply add it. However, if the set is at its maximal capacity, we add the current element only if its estimated frequency exceeds the minimum frequency in the set by replacing the element with the smallest frequency. In the end, elements in the X^* are considered the most frequent elements in the data stream.

In a similar way, the CMSketch data structure can address the Heavy hitters problem, described earlier. In a single pass through the data stream, additional to the regular $p \times m$ array of counters C and p hash functions, we allocate a single counter N that stores the number of elements seen thus far, and maintains a heap X^* of up to k potential heavy hitters. We use frequency threshold $f^* = \frac{\mathrm{N}}{k}$ to decide whether element is a heavy hitter. For every element x in the data stream, we execute an update procedure followed by the frequency estimation, and if $\hat{f}(x) \geq f^*$, then element is qualified as a heavy hitter candidate. If the element is not in the heap yet, we store it and its frequency together, otherwise we update the stored frequency with the new value.

The counter N increases with every processed element, when this grows, the estimated frequency for some elements in the heap become less than f^* and these elements must be removed from the heap at every step. At the end of the processing, all elements in the heap are considered heavy hitters. According to the definition, there are at most k heavy hitters in the data stream.

Algorithm 4.11: Getting heavy hitters with the Count–Min Sketch

```
Input: Data stream $\mathbb{D}$
Input: Count–Min sketch with $p \times m$ counters
Output: Heavy hitters
$\mathrm{N} \leftarrow 0, \mathrm{X}^* \leftarrow \varnothing$
for $x \in \mathbb{D}$ do
    $\mathrm{N} \leftarrow \mathrm{N} + 1$
    Update($x$)
    $\hat{f} \leftarrow$ Frequency($x$)
    $f^* \leftarrow \frac{\mathrm{N}}{k}$
    if $\hat{f} \geq f^*$ then
        $\mathrm{X}^* \leftarrow \mathrm{X}^* \cup \{(x, f)\}$
    for $(x^*, \hat{f}) \in \mathrm{X}^*$ do
        if $\hat{f} \leq f^*$ then
            $\mathrm{X}^* \leftarrow \mathrm{X}^* \setminus \{(x^*, \hat{f})\}$
return $\mathrm{X}^*$
```

Maintaining a heap for the ε–Heavy hitters problem with $\varepsilon = \frac{1}{2k}$ requires $O(\log \frac{1}{\varepsilon})$ additional work per element.

Example 4.10: Heavy hitters with Count–Min Sketch

Consider the same setup as in Example 4.9 and search for $k = 3$ heavy hitters while processing the dataset.

$$\{4, 4, 4, 4, 2, 3, 5, 4, 6, 4, 3, 3, 4, 2, 3, 3, 3, 2\}.$$

According to Algorithm 4.11, additional to the CMSketch data structure we create a counter N of processed elements and a heap X^* that stores up to k heavy hitters candidates. We will skip the details of updating counters and frequency estimation because these steps are the same as in examples above.

We start consuming the dataset and the first element is 4, so we increment the corresponding counters c_1^4 and c_2^4.

	1	2	3	4
h_1	0	0	0	1
h_2	0	0	0	1

At this point, we have processed $\mathrm{N} = 1$ element, so the threshold f^* for the heap X^* is $\frac{1}{3}$. The frequency estimation for element 4 from CMSketch is 1 and this is above the threshold, hence we add this element and its frequency to the heap: $\mathrm{X}^* = [(4, 1)]$.

Next element is 4 again and we increment the same counters.

	1	2	3	4
h_1	0	0	0	2
h_2	0	0	0	2

However, since we already processed $\mathrm{N} = 2$ elements, the threshold f^* is changed to $\frac{2}{3}$. The current frequency estimation for element 4 is 2 which is still above the threshold and since the element is already in the heap we just update its frequency: $\mathrm{X}^* = [(4, 2)]$.

In a similar manner we process the next 14 elements (up to $\mathrm{N} = 16$). There are no changes to the number of elements in the heap and element 4 is the only heavy hitter candidate so far: $\mathrm{X}^* = [(4, 7)]$. The CMSketch data structure has the following form:

	1	2	3	4
h_1	7	0	0	9
h_2	1	3	6	7

Next element in the dataset is 3, whose counters c_1^1 and c_2^3 we increment.

	1	2	3	4
h_1	7	0	0	9
h_2	1	3	6	7

At this moment, there are $\mathrm{N} = 17$ processed elements, so the frequency threshold is $f^* = \frac{17}{3} \approx 5.33$. The estimated frequency of element 3 is $\hat{f} = \min(7, 6) = 6$ that is above the threshold, therefore we add it to the heap: $\mathrm{X}^* = [(4, 7), (3, 6)]$. All elements in the heap have large enough frequencies, hence we don't remove any of them.

The last element in the dataset is 2 whose frequency is below the threshold $f^* = \frac{18}{3} = 6$. Therefore, there are no changes to the heap at this step and the final list of heavy hitters is

$$\mathrm{X}^* = [(4,7),(3,6)].$$

Properties

The Count–Min Sketch is approximate and probabilistic at the same time, therefore two parameters, the error ε in answering the paricular query and the error probability δ, affect the space and time requirements. In fact, it provides the guarantee that the estimation error for frequencies will not exceed $\varepsilon \cdot n$ with probability at least $1-\delta$.

Similar to the Count Sketch, the increasing number of hash functions p decreases the probability of a bad estimate. For the desired standard error δ, the recommendation for the number of hash functions that correspond to the rows in the CMSketch data structure is

$$p = \left\lceil \ln \frac{1}{\delta} \right\rceil. \tag{4.3}$$

The bigger m, the less likely collisions will happen, thus the overestimation error $\varepsilon \cdot n$ will be lower. At the same time, with bigger p more estimations are used to calculate the final minimal value, which makes it more reliable. Thus, the recommendation on the number of counters m is

$$m \approx \left\lceil \frac{2.71828}{\varepsilon} \right\rceil, \tag{4.4}$$

and comparison with (4.2) shows that the Count–Min Sketch is more space-friendly than the Count Sketch.

Since the CMSketch data structure consists of a two-dimensional array sized $p \times m$ and uses p hash functions, it requires $\mathrm{O}(m \cdot p + p)$ space, assuming that every hash function is stored in $\mathrm{O}(1)$ space.

Example 4.11: Estimate required space

According to condition (4.3), to have the standard error δ around 1%, at least $p = \lceil \ln \frac{1}{0.01} \rceil = 5$ hash functions are required. For instance, we expect 10 million ($n = 10^7$) elements to be indexed and allow the fixed overestimate of 10. Thus, we need $\varepsilon = \frac{10}{10^7} = 10^{-6}$ and the recommended number of counters is

$$m = \frac{2.71828}{10^{-6}} \approx 2718280.$$

Thus, the CMSketch data structure needs to keep the counter array sized 5×2718280 and, having 32-bit integer counters, the whole data structure requires 54.4 MB of memory.

Two Count–Min sketches of the same size can be easily merged together by simple matrix addition resulting in a data structure for the union of their datasets. As a result, the Count–Min Sketch is useful in MapReduce and parallel streaming tasks for Big Data applications.

Big data is characterized by a large amount of data that comes at high speed, which makes space and update time significant. Fortunately, the practical implementations of the Count–Min Sketch consume only up to a few hundreds of megabytes of memory and can handle dozens of millions of updates per second.

The Count–Min Sketch is widely used for tasks on traffic analysis and in-stream mining applications running on distributed stream processing frameworks including Apache Spark, Apache Storm, Apache Flink, and others. There are also implementations for popular databases such as Redis and PostgreSQL.

Conclusion

In this chapter we discussed the problem of determining frequencies of elements in continuous and potentially infinite streams, these often have to be processed by Big Data applications. We started with formulating many important frequency-related problems that can be solved using the data structures and algorithms from this chapter. Starting with the very simple problem of the majority element, we moved onto learning how to solve the very complex problems of finding the most frequent elements and heavy hitters.

If you are interested in more information about the material covered here or want to read the original papers, please take a look at the list of references that follows this chapter.

In the next chapter, we continue working with data streams and consider probabilistic algorithms that can be employed to compute rank characteristics such as quantiles and their particular types including percentiles and quartiles.

Bibliography

[Bo81] Boyer, R., Moore, J. (1981) “MJRTY - A Fast Majority Vote Algorithm”, *Technical Report 1981-32*, Institute for Computing Science, University of Texas, Austin.

[Ch02] Charikar, M., Chen, K., Farach-Colton, M. (2002) “Finding Frequent Items in Data Streams”, *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, pp. 693–703, Springer, Heidelberg.

[Co09] Cormode, G. (2009) “Count-min sketch”, In: Ling Liu, M. Tamer Özsu (Eds.) – *Encyclopedia of Database Systems*, pp. 511–516, Springer, Heidelberg.

[Co03] Cormode, G., Muthukrishnan, S. (2003) “What’s hot and what’s not: Tracking most frequent items dynamically”, *Proceedings of the 22th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, San Diego, California — June 09–11, 2003, pp. 296–306, ACM New York, NY.

[Co05] Cormode, G., Muthukrishnan, S. (2005) “An Improved Data Stream Summary: The Count–Min Sketch and its Applications”, *Journal of Algorithms*, Vol. 55 (1), pp. 58–75.

[Co08] Cormode, G., Hadjieleftheriou, M. (2008) “Finding frequent items in data streams”, *Proceedings of the VLDB Endowment*, Vol. 1 (2), pp. 1530–1541.

[De02] Erik, D., Demaine, E.D., López-Ortiz, A., Munro, J.I. (2002) “Frequency Estimation of Internet Packet Streams with Limited Space”, In: R. Möhring and R. Raman (Eds.) – ESA 2002. *Lecture Notes in Computer Science*, Vol. 2461, pp. 348–360, Springer, Heidelberg.

[Fi82] Fischer, M.J., Salzberg, S.L. (1982) “Finding a Majority Among N Votes: Solution to Problem 81-5”, *Journal of Algorithms*, Vol. 3, pp. 376-–379.

[Mi82] Misra, J., Gries, D. (1982) “Finding repeated elements”, *Science of Computer Programming*, Vol. 2 (2), pp. 143–152.

[Mu05] Muthukrishnan, S. (2005) “Data Streams: Algorithms and Applications”, *Foundations and Trends in Theoretical Computer Science*, Vol. 1 (2), pp. 117–236.

5 Rank

Large volumes of unstructured data easily overwhelm the human ability to understand it and makes data summarization by computing statistical quantities one of the most necessary tasks to perform with data. In this chapter, we investigate algorithms and data structures to calculate rank-based characteristics of the data using a small amount of memory and one pass through the data.

The most commonly used rank characteristics are quantiles. Formally, the q-*quantile* $(0 \leq q \leq 1)$ is an element of the sequence where a q fraction of elements from the sequence are less or equal to it, and the remaining $(1-q)$ are greater or equal. Moreover, if the sequence has n elements, we say that a q-quantile element is an element of the sequence those *rank* is $q \cdot n$. Percentiles are just quantiles that divide the sorted sequence into 100 equal parts, hence the 95th percentile is the same as the 0.95-quantile. The 0- and 1-quantiles are the minimal and the maximal elements in the sequence, respectively. The 0.5-quantile is known as the median.

As was proven by Ian Munro and Michael Paterson[1], to find a particular quantile *exactly* in p passes through the data requires $O(n^{\frac{1}{p}})$ memory. This means that any one-pass algorithm cannot guarantee to produce the precise value of the quantile in sublinear space. This motivates a search for algorithms that compute approximate quantiles.

In practice, having an error in a quantile calculation is often tolerable because they are usually estimated for noisy input data and approximate the unknown data distributions. Thus, in most cases we are interested in the ε–approximate q-quantile, meaning an element with its rank in $[(q-\varepsilon) \cdot n, (q+\varepsilon) \cdot n]$, where n is the number of elements and $0 < \varepsilon < 1$ is an error parameter. Note that more than one element could qualify.

Estimation of various rank characteristics like quantile summaries plays an important role in streaming outlier detection methods. For instance, if we monitor online e-commerce transactions to detect credit card fraud, we are interested in unusual payment locations, those that don't fit in the 99th percentile of usual location distribution for our customers.

Example 5.1: Fraud detection (Perlich et al., 2007)

Financial fraud remains one of the most critical issues facing the financial industry. For instance, in 2015, global credit and debit card fraud resulted in losses amounting to \$21.84 billion[2].

Many applications have been built to search for and identify the signs of financial fraud. Such applications frequently use numerous specific variables whose "degree of outlyingness" is examined for every observation. For instance, variables such as the total amount spent on a credit card

[1]Selection and sorting with limited storage, Theoretical Computer Science, Vol.12 (1980)

and the amount spent per day can be used.

For every observation, the degree of outlyingness can be approximated by the quantiles of some spending distribution. Thus, the suspected fraudulent observations can be identified as outliers through comparison to some high quantiles, e.g., the 0.95-quantile.

Another huge application domain of rank summaries is web traffic monitoring. Investigating the summaries means problems can be detected early, without inspecting the actual data.

Example 5.2: Website monitoring (Buragohain & Suri, 2009)

Big websites handle millions of users every single day. For instance, in September 2017 Wikipedia processed about 500 million hits per day[3] across all its languages, that is roughly 5.7 thousands requests per second, using more than 300 servers around the globe.

One of the most critical issues in a website's performance is latency, the delay between when the content was created and the time it was transferred to the visitor. Since the distribution of the latency values is typically skewed, the monitoring usually is built by tracking some particular high-quantities or percentiles. The most common questions are:

- What is the latency for 95% of requests for a single web server?
- What is the latency for 99% of requests for the entire website?
- What was the latency for 95% of requests for the entire website in the last 15 minutes?

While all these questions can be answered with the quantile computation, technically they have differences that might require the application of different methods. For instance, while for the first question, a summary can be computed per single stream, the second question requires distributed algorithms that can compute statistics on many streams' data. In contrast, the third question requires only a subset of the stream's data defined by a time window and such a subset will always change.

[2]Credit Card & Debit Card Fraud Statistics `https://wallethub.com/edu/statistics/25725/`

The task to find q-quantile, or, in other words, elements from a sorted sequence of n elements whose rank is $q \cdot n$, where $q \in (0, 1)$, is called a *Quantile query*. The Median query is a special case of the quantile query with $q = 0.5$.

The problem of quantile calculation is not new and is already well developed in classical computation. However, it has new challenges for unbounded streams, which are common for Big Data applications, when limited memory is available, and only a single pass through the data is possible. The Count–Min Sketch algorithm, previously introduced in Chapter 4, allows for the computation of approximate values of quantiles but requires much more memory than the algorithms that will be discussed in this chapter.

Alternatively, we can search for the rank of the given element in a sorted sequence of n elements, known as an *Inverse quantile query*. With $\text{rank}(x)$ and the total number of elements n, it is easy to compute the corresponding quantile q:

$$q = \frac{1}{n} \cdot \text{rank}(x).$$

For many applications, it is also important to find the number of elements from a sorted sequence of n elements that are in some given range $[a, b]$, often referred to as a *Range query*. In fact, to calculate such a number, it is enough to compute the ranks of the range's boundaries and return their difference.

In this chapter, we start with a randomized sampling algorithm, then continue with a simple tree-based q-digest and, finally, study the modern t-digest algorithm that uses clustering for efficient estimation of rank-based statistics in unbounded streams.

[3]Wikipedia Page Views `https://stats.wikimedia.org/EN/TablesPageViewsMonthlyCombined.htm`

5.1 Random sampling

The *random sampling* technique, selecting without replacement a random subset of the data, can be found in many algorithms in computer science. For rank problems, this technique can be used to report quantiles computed on samples, as an approximation to the quantiles of the whole data stream.

The distinct advantage is that such samples are much smaller, in fact, often rank quantiles queries can be answered using classical deterministic algorithms. However, to have some prior guarantees on the error of such an approximation, the random sample has to be taken in a special way, which could even be data dependent.

Additional problem that may occur with classical sampling is that many sampling schemas require prior knowledge of the size of the dataset, that is problematic for the continuous streams often used in Big Data applications. One of the possible solutions is the simple *reservoir sampling* technique, developed by Jeffrey Vitter in 1985, that allowed for the generation of a sample without such knowledge, but if we wanted to apply it directly to the Quantile problem the memory requirements would be quite significant.

The *Random sampling* algorithm, often referred to as MRL, was published by Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce Lindsay in 1999 [Ma99] and addressed the problem of the correct sampling and quantile estimation. It consists of a non-uniform sampling technique and deterministic quantile finding algorithm.

To support continuous data streams processing with little space requirements, Manku et al. suggested a non-uniform modification of reservoir sampling where elements that appear earlier in the sequence are included with higher probability than others. Such a modification has better space-efficiency and considerably more accurate than the original reservoir sampling.

The main disadvantages of the MRL algorithm are that its configuration parameters are determined by solving a complicated optimization problem and it uses some complex procedures. In this section, we investigate a simpler version of the MRL algorithm that was proposed by Ge Luo, Lu Wang, Ke Yi, and Graham Cormode in 2013 [Wa13], [Lu16], and denoted in the original articles as *Random*.

The Random algorithm processes data from the data stream in chunks of variable sizes and performs a sampling on them that produces the non-uniform sampling at the end.

In order to store samples of the elements, the algorithm maintains a data structure SampleBuffers that consists of b simple data units $\mathrm{B}_1, \mathrm{B}_2, \ldots \mathrm{B}_b$, called *buffers*, each of these stores at most k elements and can be associated with some level L at which it was populated.

The level parameter L reflects the probability that the elements are drawn and depends on the number of elements n that have been processed so far and the maximum allowed *height* h of the tree that represents the sequence of operations carried out the algorithm:

$$\mathrm{L} = \mathrm{L}(n, h) = \max\left(0, \left\lceil \log \frac{n}{k \cdot 2^{h-1}} \right\rceil\right), \tag{5.1}$$

where $\mathrm{L}(0, h) = 0$.

To populate an empty buffer $\mathrm{B}_i^{\mathrm{L}}, i \in 0 \ldots b$ at level L, we choose k random elements from $k \cdot 2^{\mathrm{L}}$ consequent input elements, one per block of 2^{L}, and store them in $\mathrm{B}_i^{\mathrm{L}}$. At the end of the procedure, the buffer might have less than k elements because the input sequence did not have enough elements, but if at least one element is in the buffer, it is labeled as *full*.

The probability that a particular element from the incoming data stream is selected and stored into a buffer directly depends on the level L since it controls the size of the chunk 2^{L} from which elements are drawn. This is the practical implementation of the non-uniform sampling used in the algorithm.

Algorithm 5.1: Populating empty buffers

```
Input: Data stream $\mathbb{D}$
Input: Empty buffer $\mathrm{B}^{\mathrm{L}}$ of size $k$ at level L
Output: Populated buffer $\mathrm{B}^{\mathrm{L}}$ and its label
for $i \leftarrow 0$ to $k-1$ do
    $\mathrm{S} \leftarrow$ next($2^{\mathrm{L}}, \mathbb{D}$) // read next $2^{\mathrm{L}}$ elements from $\mathbb{D}$
    if $\mathrm{S} = \varnothing$ then
        break
    $x \leftarrow$ sample($\{s \in \mathrm{S}\}$) // randomly choose one element from S
    $\mathrm{B}^{\mathrm{L}} \leftarrow \mathrm{B}^{\mathrm{L}} \cup \{x\}$
$label \leftarrow$ empty
if count($\mathrm{B}^{\mathrm{L}}$) $> 0$ then
    $label \leftarrow$ full
return $\mathrm{B}^{\mathrm{L}}$, $label$
```

Two buffers from the same level L can be collapsed, merged to reclaim buffer space, which results in a new buffer of the same size at level $\mathrm{L}+1$. To collapse two buffers, we sort the sequence of the elements from both, and randomly select half of the elements, for example, by choosing all the elements at either odd or even positions. The collapsed buffers are marked as *empty*, and the output buffer as *full*.

Algorithm 5.2: Collapsing two non-empty buffers

```
Input: Non-empty bufferts $\mathrm{B}_i^{\mathrm{L}}$, $\mathrm{B}_j^{\mathrm{L}}$ of size $k$ at level L
Output: Populated buffer $\mathrm{B}^{\mathrm{L}+1}$ at level L and its label
$\mathrm{S} \leftarrow$ sort($\mathrm{B}_i^{\mathrm{L}} \cup \mathrm{B}_j^{\mathrm{L}}$)
free($\mathrm{B}_i^{\mathrm{L}}$)
free($\mathrm{B}_j^{\mathrm{L}}$)
$\mathrm{B}^{\mathrm{L}+1} \leftarrow$ sample($\mathrm{S}, k$) // randomly choose $k$ elements from joined buffers
return $\mathrm{B}^{\mathrm{L}+1}$, full
```

The collapse operation requires $\mathrm{O}(k \cdot \log k)$ for sorting the buffers and the subsequent buffer population can be performed in $\mathrm{O}(k)$ time.

Finally, the process of building the SampleBuffers data structure consists of a series of buffer population steps and collapse operations.

We start with every buffer labeled as empty. Processing of the input stream starts with setting the active level L using the formula (5.1), this is equal to zero at the beginning since there are no processed elements yet. If there is an empty buffer B, we populate it using Algorithm 5.1 by reading $k \cdot 2^{L}$ elements from the stream. When all buffers become full, we find the lowest level that contains at least two buffers and collapse two that have been randomly selected.

The total number of collapse operations is $O\left(\frac{n}{k}\right)$ throughout the entire data stream, which is about O(1) for each update. The sorting takes $O(\log k)$ for each update. Thus, the amortized time is $O(\log k)$.

Example 5.3: Build Sample Buffers

Consider a dataset of 25 integers:

$$\{0, 0, 3, 4, 1, 6, 0, 5, 2, 0, 3, 3, 2, 3, 0, 2, 5, 0, 3, 1, 0, 3, 1, 6, 1\}.$$

To illustrate the process of handling a data stream, we use the height $h = 3$ and maintain $b = 4$ buffers: B_1, B_2, B_3, B_4 of $k = 4$ elements each. Thus, simplifying the formula (5.1), the active level can be calculated as

$$L = L(n) = \max\left(0, \lceil \log(n) - 4 \rceil\right).$$

In the beginning, the number of processed elements $n = 0$, hence we start from $L = 0$ and read the first $N_1 = 4$ elements from the input stream $\{0, 0, 3, 4\}$, and populate an empty buffer, say B_1. Since the capacity of the buffer is also 4, we don't need to draw random elements and all of them are stored.

$B_1^{\mathbf{0}}$				B_2				B_3				B_4			
0	0	3	4												

Secondly, we again need to define the active level. The number of processed elements is $n = N_1 = 4$ and the active level remains zero: $L = L(4) = \max(0, 2 - 4) = 0$. We read the next $N_2 = 4$ elements $\{1, 6, 0, 5\}$ and in the same way, populate buffer B_2.

$B_1^{\mathbf{0}}$				$B_2^{\mathbf{0}}$				B_3				B_4			
0	0	3	4	1	6	0	5								

Thus, we have already processed $n = N_1 + N_2 = 8$ elements, the current level $L = L(8) = \max(0, 3-4) = 0$, and we index the next $N_3 = 4$ elements $\{2, 0, 3, 3\}$ to buffer B_3.

$B_1^{\mathbf{0}}$				$B_2^{\mathbf{0}}$				$B_3^{\mathbf{0}}$				B_4			
0	0	3	4	1	6	0	5	2	0	3	3				

Likewise, after processing $n = N_1 + N_2 + N_3 = 12$ elements, the active level is still zero, we again read the next $N_4 = 4$ elements $\{2, 3, 0, 2\}$ and populate the only remaining empty buffer B_2.

$B_1^{\mathbf{0}}$				$B_2^{\mathbf{0}}$				$B_3^{\mathbf{0}}$				$B_4^{\mathbf{0}}$			
0	0	3	4	1	6	0	5	2	0	3	3	2	3	0	2

At this point we have no empty buffers left, hence we need to perform the collapse operation. The lowest layer that has at least two buffers is level 0 from which we randomly select two buffers, for instance, B_2^0 and B_3^0. First, we merge all elements from these buffers and sort them:

$$\{1, 6, 0, 5\} \cup \{2, 0, 3, 3\} = \{1, 6, 0, 5, 2, 0, 3, 3\} \to \{0, 0, 1, 2, 3, 3, 5, 6\}.$$

Next, we free buffers B_2^0 and B_3^0, and populate buffer B_3 at level 1 with 50% of their former elements, for simplicity let's take the odd elements.

$B_1^{\mathbf{0}}$				B_2				$B_3^{\mathbf{1}}$				$B_4^{\mathbf{0}}$			
0	0	3	4					0	1	3	5	2	3	0	2

Thus, we have already processed $n = N_1 + N_2 + N_3 + N_4 = 16$ elements, but the active layer remains zero, and we populate B_2 with the next $N_5 = 4$ elements from the data stream: $\{5, 0, 3, 1\}$.

$B_1^{\mathbf{0}}$				$B_2^{\mathbf{0}}$				$B_3^{\mathbf{1}}$				$B_4^{\mathbf{0}}$			
0	0	3	4	5	0	3	1	0	1	3	5	2	3	0	2

Once again there are no empty buffers, thus we need to perform another collapse.

Level 0 contains three full buffers and we randomly choose two of them, e.g., B_1^0 and B_4^0, then merge and sort all their elements:

$$\{0,0,3,4\} \cup \{2,3,0,2\} = \{0,0,3,4,2,3,0,2\} \to \{0,0,0,2,2,3,3,4\}.$$

We label buffers B_1^0 and B_4^0 as empty and populate buffer B_4 at level 1 with 50% of their elements by taking elements in even positions.

B_1				$B_2^{\mathbf{0}}$				$B_3^{\mathbf{1}}$				$B_4^{\mathbf{1}}$			
				5	0	3	1	0	1	3	5	0	2	3	4

At the next step, we have already processed $n = N_1 + N_2 + N_3 + N_4 + N_5 = 20$, thus the active level $L = L(20) = 4.32 - 4 = 1$, and we read the next $N_6 = 4 \cdot 2^1 = 8$ elements from the data stream. In this case, there are not enough elements left in the data stream, we read $\{0,3,1,6,1\}$ and populate B_1 by sampling one element from each group of two elements.

$B_1^{\mathbf{1}}$				$B_2^{\mathbf{0}}$				$B_3^{\mathbf{1}}$				$B_4^{\mathbf{1}}$			
3	1	1		5	0	3	1	0	1	3	5	0	2	3	4

Finally, we have built the resulting data structure SampleBuffers.

With SampleBuffers it is possible to answer the Inverse quantile query and the rank of the given element x can be estimated as weighted by the layer sum of counts of elements smaller than x for each non-empty buffer:

$$\operatorname{rank}(x) = \sum_{i=1}^{k} 2^{L(B_i)} \cdot |\{e < x | e \in B_i^{L(B_i)}\}|. \tag{5.2}$$

Example 5.4: Inverse quantile query with Random sampling

Consider the data stream from Example 5.3 and perform the Inverse quantile query to estimate the rank of element 4.

The data structure SampleBuffers has the following form.

$B_1^{\mathbf{1}}$				$B_2^{\mathbf{0}}$				$B_3^{\mathbf{1}}$				$B_4^{\mathbf{1}}$			
3	1	1		5	0	3	1	0	1	3	5	0	2	3	4

Using the formula (5.2) we calculate the rank:

$$\mathrm{rank}(4) = 2^1 \cdot 3 + 2^0 \cdot 3 + 2^1 \cdot 3 + 2^1 \cdot 3 = 21.$$

Thus, for element 4 the estimated rank(4) = 21.

To answer the Quantile query and find the q-quantile from the SampleBuffers data structure, we simply need to search for an element whose estimated rank, derived from formula (5.2), is closest to $q \cdot n$. In fact, we need to ask a number of Inverse quantile queries for each of the elements in the data structure, but we can use the binary search to speed up the process, and stop as soon as we find a value that is close enough.

Example 5.5: Quantile query with Random sampling

Consider the data stream from Example 5.3 and calculate the 0.65-quantile.

The total number of elements in the data structure SampleBuffers is $n = 25$, so our boundary value is $q \cdot n = 0.65 \cdot 25 = 16.25$.

$B_1^{\mathbf{1}}$				$B_2^{\mathbf{0}}$				$B_3^{\mathbf{1}}$				$B_4^{\mathbf{1}}$			
3	1	1		5	0	3	1	0	1	3	5	0	2	3	4

There are elements $\{0, 1, 2, 3, 4, 5\}$ in the SampleBuffers. We start with element 0 and estimate its rank that, according to the formula (5.2), equals to zero: rank(0) = 0. Next, we check element 1 and its rank estimation is rank(1) = 5. The rank of element 2 is rank(2) = 12, while rank(3) = 14. And we already know from Example 5.4 rank(4) = 21. Finally, the rank of element 5 is rank(5) = 23.

Thus, the closest element to the boundary value 16.25 is element 3 with rank(3) = 14. We report element 3 as an approximation of the 0.65-quantile.

Note that we could speed up the process by using a binary search over the sorted number of elements, taking into account that rank is a monotonic function of its argument.

Properties

To compute the ε–approximate of q-quantile, the Random requires a fixed amount of memory that is proportional to $b \cdot k$ and depends only on ε. With given approximation error, we can have the computation tree with height $h = \log \frac{1}{\varepsilon}$, and the optimial number of buffers is

$$b = \log \frac{1}{\varepsilon} + 1,$$

while the size of each buffer is

$$k = \frac{1}{\varepsilon}\sqrt{\log \frac{1}{\varepsilon}}.$$

Being probabilistic, the Random algorithm correctly reports quantile approximates with constant error probability that is bounded by $\frac{1}{2}\varepsilon$ and originates from random sampling and random merging steps.

5.2 q-digest

Quantile digest, or *q-digest*, is a tree-based stream summary algorithm that was proposed by Nisheeth Shrivastava, Chiranjeeb Buragohain, Divyakant Agrawal, and Subhash Suri in 2004 [Sh04] for use in the context of monitoring distributed data from sensors.

The q-digest addresses the quantile computation problem as a histogram problem when data are summarized by some number of buckets. The algorithm maintains a set of such buckets in a tree-like Q-DIGEST data structure, merges small buckets, and splits the big ones. It is a lossy deterministic algorithm, however, we consider it very useful and important for our narration.

The algorithm works with integer values within some known range. The binary partition of the integer range $[0, \mathrm{N}-1]$ can be represented as a virtual full and complete binary tree, whose root element corresponds to the whole range $[0, \mathrm{N}-1]$, its left and right children have ranges $\left[0, \left\lfloor \frac{\mathrm{N}-1}{2} \right\rfloor\right]$

and $\left[\left\lfloor \frac{N-1}{2} \right\rfloor + 1, N-1\right]$, and, iteratively, the leaf nodes represent single integer values. The depth of the tree is $\log N$.

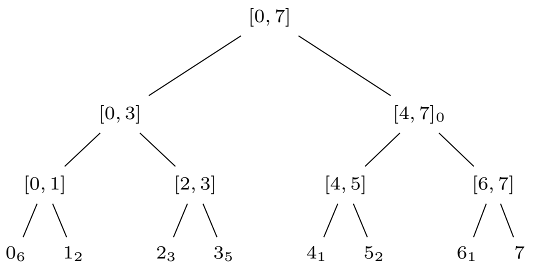
Every node ν in such a binary tree is a bucket that has an associated range $[\nu_{\min}, \nu_{\max}]$. Additionally, we associate counters ν_{count} for each bucket to represent the number of elements (including duplicates) that are indexed in it.

Example 5.6: Binary partitioning for q-digest

Consider a dataset of $n = 20$ integers from range $[0, 7]$ that we investigated in Example 5.3:

$$\{0, 0, 3, 4, 1, 6, 0, 5, 2, 0, 3, 3, 2, 3, 0, 2, 5, 0, 3, 1\}.$$

By binary partitioning the range we build the following binary tree and bucket the input data:



The leaf nodes from left to right represent elements from $[0, N-1]$ and the index numbers indicate the frequencies of the elements in the dataset.

Thus, the internal representation of the data consists of the frequencies with which the stored elements were observed. At worst, storage limitations mean we have to store such data as $O(n)$ or $O(N)$, whichever is smaller. Note that in practice such binary trees are likely to be very sparse and imbalanced, therefore storing it in raw form without compression is quite inefficient.

The q-digest algorithm proposes a way to compress and compactly store such a binary partition tree. Its data structure Q-DIGEST encodes information about the distribution of elements and represents a version of the binary tree that includes only those buckets ν that satisfy the following *digest property*:

$$\begin{cases} \nu_{\text{count}} \leq \lfloor \frac{n}{\sigma} \rfloor, & \text{(except leaf buckets)} \\ \nu_{\text{count}} + \nu^{p}_{\text{count}} + \nu^{s}_{\text{count}} > \lfloor \frac{n}{\sigma} \rfloor, & \text{(except the root)} \end{cases} \tag{5.3}$$

where ν^p is the parent and ν^s is the sibling of ν; n is the total number of elements, and $\sigma \in [1, n]$ is a design parameter responsible for the level of compression.

The exception from this property is the root and leaf buckets. The root bucket can violate the digest property (5.3), however, still be included in the Q-DIGEST data structure. The leaf buckets with counts bigger than the *boundary value* $\lfloor \frac{n}{\sigma} \rfloor$ (frequent elements) are included as well.

In fact, the digest property defines a compromise between including a few top-level and broad buckets, and many small buckets that contain information about a few non-frequent elements.

Simplifying, the first constraint in the digest property (5.3) excludes buckets unless they are leaf nodes which contain counts about high-frequency elements because for such buckets it is worth storing child elements and having more precise counters.

On the other hand, according to the second constraint, if two adjacent buckets, which are siblings, have low counts, then we want to avoid having two separate counters for each of them and it is better to merge them into their parent and achieve the required degree of compression.

Thus, the construction of the Q-DIGEST requires hierarchical merging and reduction of the buckets, going through all buckets bottom-up and checking if any of them violate the digest property. In practice, since we are only going bottom-up the second constraint to be checked.

Except for the root bucket, for every bucket ν that violates the digest property, we merge its subtree by compressing counts from it, its parent ν^p and sibling ν^s, and promote them to the parent bucket:

$$\nu^{p}_{\text{count}} = \nu^{p}_{\text{count}} + \nu_{\text{count}} + \nu^{s}_{\text{count}},$$

while excluding the bucket ν and its sibling ν^s from the Q-DIGEST.

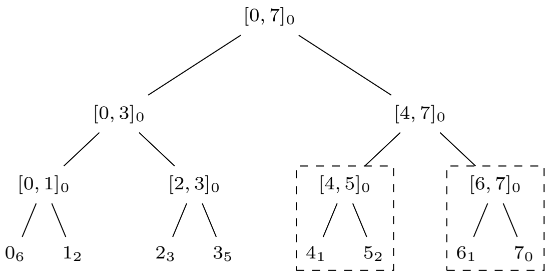
Algorithm 5.3: Compressing q-digest

```
Input: q-digest data structure of a range $[0, N-1]$
Input: Compression factor $\sigma$
Output: Compressed q-digest data structure
$level \leftarrow \log N - 1$
while $level > 0$ do
    for $\nu \in$ Q-DIGEST$[level]$ do
        if $\nu_{count} + \nu^{p}_{count} + \nu^{s}_{count} \leq \lfloor \frac{n}{\sigma} \rfloor$ then
            $\nu^{p}_{count} \leftarrow \nu^{p}_{count} + \nu_{count} + \nu^{s}_{count}$
            Q-DIGEST $\leftarrow$ Q-DIGEST $\setminus \{\nu, \nu^{s}\}$
    $level \leftarrow level - 1$
return Q-DIGEST
```

The compression takes $O(m \cdot \log N)$ time, where $m = |\text{Q-DIGEST}|$ is the number of buckets in the data structure; thus, the theoretical update cost per element is about $O(\log N)$. In practice, however, the update takes more time because every element is inserted into the leaf node first, and then, during the compress operation, the algorithm needs to find its appropriate position in the Q-DIGEST by moving the element up one step at a time.

Example 5.7: Compress tree with q-digest

Consider the dataset of $n = 20$ elements from Example 5.6 where the frequencies for non-observed buckets default to zero.

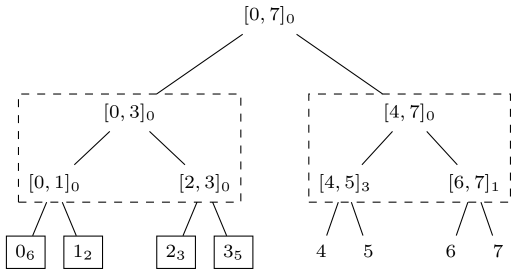


Let's assume we want to achieve a compression with $\sigma = 5$, then the boundary value is

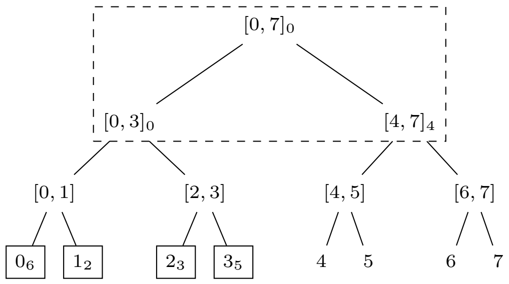
$$\left\lfloor \frac{n}{\sigma} \right\rfloor = \left\lfloor \frac{20}{5} \right\rfloor = 4.$$

Going bottom-up, consider the fourth level first, where only buckets from 0 to 3 satisfy the second condition of the digest property (5.3). According to Algorithm 5.3, the children of buckets $[4, 5]$ and $[6, 7]$, that together violate the digest property, have to be merged to their parents and excluded from the Q-DIGEST.

Thus, the Q-DIGEST at this stage is (buckets in solid-line boxes are included in the compressed data structure):



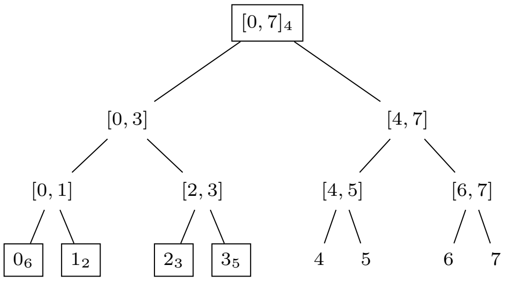
Further, at the third level, all buckets violate the constraints (5.3). Therefore, we also compress them to their parents and don't include in the Q-DIGEST:



At the second level, we check the digest property for two children of the root bucket, which again violate the constraints (5.3) since their total counts do not exceed the boundary value and, consequently, they have to be merged to the parent.

For the root element, it is not necessary to check the digest property since it, as we declared earlier, is always included in the compressed Q-DIGEST if it has non-zero associated counts.

Hence, the final version of the compressed q-DIGEST data structure is as follows:



As we can see in this example, the compressed q-DIGEST data structure requires storing only five buckets with non-zero counts.

Because we always go bottom-up (and never top-down), check the digest property, and make the decision about merging buckets only once during the procedure, it is not necessary that all buckets from the compressed q-digest satisfy the digest property after compression. For instance, changes (e.g., merging to a parent) in some buckets on the top levels of the tree could make already included buckets violate the constraints of the digest property (5.3). However, in practice, this behavior does not decrease the accuracy of the algorithm and, at worst, produces a less optimal data structure that consumes more memory than is theoretically expected.

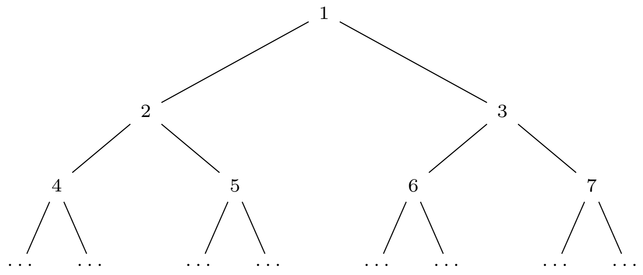
Putting that all together, we can formulate the complete q-digest algorithm for the arbitrary dataset as below.

Algorithm 5.4: q-digest algorithm

```
Input: Dataset $\mathbb{D}$ with elements from range $[0, N-1]$
Input: Compression factor $\sigma$
Output: Compressed q-digest data structure
Q-DIGEST $\leftarrow$ BinaryPartitionTree($\mathbb{D}$, $[0, N-1]$)
return Compress(Q-DIGEST, $N$, $\sigma$)
```

To optimize the representation of the Q-DIGEST data structure, the buckets in the associated binary tree can be enumerated in a right-to-left, top-to-bottom manner:

Figure 5.1: Buckets enumeration



As soon as all buckets ν are enumerated, it is easy to restore the corresponding range $[\nu_{\min}, \nu_{\max}]$ even if we only know its index i.

Algorithm 5.5: Restoring bucket range $[\nu_{\min}, \nu_{\max}]$

```
Input: Bucket index $i$
Output: Bucket range
$level \leftarrow \lfloor \log(i) \rfloor$
$n \leftarrow 2^{level-1}$ // number of buckets on the level
$m \leftarrow i \mod n$ // position of the bucket on the level
return $\left\lceil \frac{N}{n} \cdot m \right\rceil, \left\lfloor \frac{N}{n} \cdot (m+1) \right\rfloor$
```

In this way, we can build a linear representation of the Q-DIGEST data structure — an array of buckets, where each bucket is just a 2-tuple of its number and the associated counts. For example, the compressed Q-DIGEST from Example 5.7 has the following linear representation: $\Big\langle (1,4), (8,6), (9,2), (10,3), (11,5) \Big\rangle$.

Two q-digests with the same compression factor σ and element ranges can be easily merged which allows for the processing of Big Data streams in a distributed fashion. The idea is to take the union of their sets of stored buckets and add the counts of the buckets with the same range, sum the total number of elements, and afterward run the compression algorithm.

The q-digest algorithm can be used to answer the Quantile query and find the q-quantile from the Q-DIGEST data structure. At first, we obtain a sorted sequence S by ordering the buckets in increasing order of their $\nu_{\max}$ values, breaking ties with smaller values first. After that, we can scan the sequence S from the beginning and add the counts of buckets as they are seen. As soon as for some bucket ν^* this sum, that is the rank estimation for the bucket, becomes larger than $q \cdot n$, its $\nu^*_{\max}$ is reported as the estimate to the q-quantile.

Algorithm 5.6: Answering Quantile queries with q-digest

```
Input: q-digest data structure
Input: Value $q \in [0, 1]$
Output: $q$-quantile
S ← sort(Q-DIGEST)
$rank \leftarrow 0$
for $(\nu, \text{count}) \in S$ do
    $rank \leftarrow rank + count$
    if $rank \geq q \cdot n$ then
        return $\nu_{\max}$
```

There are at least $q \cdot n$ buckets whose max values are less than $\nu^*_{\max}$, therefore the rank of bucket ν^* is at least $q \cdot n$.

It is possible to have an error in calculation of the ε–approximate of q-quantile if values less than $\nu^*_{\max}$ are present in the ancestors of bucket ν^* because in this case they will not be counted by Algorithm 5.6. Analytically, such an error is bounded by $\varepsilon \cdot n$ and the algorithm reports a rank in $[q \cdot n, (q + \varepsilon) \cdot n]$ interval; thus, it never underestimates the exact value of the q-quantile.

Example 5.8: Quantile query with q-digest

We perform the Quantile query to calculate the 0.65-quantile from the Q-DIGEST data structure built in Example 5.7 whose linear representation has the following form:

$$\Big\langle (1,4),(8,6),(9,2),(10,3),(11,5) \Big\rangle.$$

Thus, the sorted sequence of buckets is

$$\mathrm{S} = \Big\langle (8,6),(9,2),(10,3),(11,5),(1,4) \Big\rangle.$$

According to the algorithm, going from the beginning, we sum counts of the buckets until the total becomes larger than $0.65 \cdot n = 13$. In the current Q-DIGEST, we exceed that boundary value at the bucket $(11,5)$ that corresponds to leaf element 3.

Thus, the q-digest estimation of the 0.65-quantile (or 65th percentile) for the dataset of Example 5.7 is the element 3.

In a similar manner can be addressed the Inverse quantile query. We build a sorted sequence S of the buckets and traverse it from the beginning while maintaining the running sum of counts from seen buckets. The rank of the given element x can be estimated as the sum of the counts of the buckets ν for which $x > \nu_{\max}$.

Algorithm 5.7: Answering Inverse quantile queries with q-digest

```
Input: Element $x$
Input: q-digest data structure
Output: Rank of element
$S \leftarrow$ sort(Q-DIGEST)
$rank \leftarrow 0$
for $(\nu, \text{count}) \in S$ do
    if $x > \nu_{\max}$ then
        $rank \leftarrow rank + count$
return $rank$
```

As in the Quantile query above, the rank obtained here lies within the interval $[\mathrm{rank}(x), \mathrm{rank}(x) + \varepsilon \cdot n]$.

As we already mentioned, to answer the Range query it is enough to perform two Inverse quantile queries to find the ranks and the difference between the range borders a and b. The maximum error for the Range query in q-digest can be estimated as $2\varepsilon \cdot n$.

Algorithm 5.8: Answering Range queries with q-digest

```
Input: Range $[a, b]$
Input: q-digest data structure
Output: Number of elements in range
$r_a \leftarrow$ **InverseQuantileQuery**($a$, Q-DIGEST)
$r_b \leftarrow$ **InverseQuantileQuery**($b$, Q-DIGEST)
return $r_b - r_a$
```

Properties

The q-digest algorithm is a lossy algorithm; it compresses information about low-frequency elements while carefully preserving information about high-frequency ones. Therefore, it provides a good approximation schema when there are wide variations in frequencies of different elements. The q-digest algorithm can provide information about the distribution of elements values, but not the information concerning where those values have occurred.

There is a clear trade-off between the accuracy of the algorithm and the memory required to store the Q-DIGEST data structure, that is controlled by the compression factor σ. Thus, for the given range $[0, \mathrm{N}]$, we can expect at most $3 \cdot \sigma$ stored buckets and the error in ε–approximate q-quantile computation is upper bounded:

$$\varepsilon \leq \frac{\log \mathrm{N}}{\sigma}.$$

The core property of the q-digest is that it is adaptive to the data and builds buckets of almost equal weights. In contrast to the traditional histogram, q-digest allows overlapping buckets which makes it possible to answer consensus queries (e.g., the frequent values).

The main problems in practical applications of the q-digest algorithm is that it can handle only integer elements, requires their range being known in advance, and suffer from significant errors for extreme quantiles.

5.3 t-digest

One of the modern alternatives to the accurate online accumulation of rank-based statistics is called *t-digest* and was proposed by Ted Dunning and Otmar Ertl in 2014 [Du14]. The t-digest algorithm allows estimating quantiles in unbounded streams with a focus on extreme values such as 0.99-quantile. This is ongoing research and the algorithm periodically gets improvement updates based on its practical applications [Du18].

The t-digest summarizes the input data stream $\mathbb{D}$ in varying-sized clusters $\{\mathcal{C}_i\}_{i=1}^{m}$ that allows it to maintain a good accuracy in quantile computation while processing a large amount of data. Every such cluster $\mathcal{C}_i$ represents a subset of input elements and sized to ensure it is not too large to be able to estimate quantiles by interpolation, but not too small to prevent ending up with too many clusters.

Every cluster $\mathcal{C}_i$ is defined by the centroid c_i, a data point at the center of the cluster, that is the mean of the input elements that contribute to this cluster, and the number of such elements c_i^{count}. The T-DIGEST data structure is an array of such weighted centroids $\{(c_1, c_1^{\text{count}}), (c_2, c_2^{\text{count}}), \dots (c_m, c_m^{\text{count}})\}$ that are sorted in ascending order. From this sorted sequence we can estimate the maximal quantile value that corresponds to each centroid c_i:

$$q(c_i) = \frac{1}{n} \sum_{j<i} c_j^{\text{count}} + \frac{1}{n} c_i^{\text{count}}, \tag{5.4}$$

where $n = \sum\limits_{j=1}^{m} c_j^{\text{count}}$ is the total number of indexed elements in the data structure.

Thus, according to (5.4), every cluster $\mathcal{C}_i$ in T-DIGEST data structure is responsible for a certain range of quantile values $(q(c_{i-1}), q(c_i)]$, whose length depends on the cluster size, the number of elements that contribute to this cluster. Correct cluster sizing has a direct influence on the accuracy and in the t-digest algorithm it is provided by a non-decreasing *scale function*. Such a function $k = k(q, \sigma)$ takes into account desired compression σ and scales quantile values q differently based on how far their are from the extrema such as $q = 0$ and $q = 1$. The good choice of the scale function is crucial and there are alternative functions with different trade-offs in terms of accuracy [Du18a]. For instance, one of commonly used functions is

$$k(q, \sigma) = \frac{\sigma}{2\pi} \arcsin(2q - 1), \tag{5.5}$$

where the compression parameter $\sigma > 1$ (bigger values correspond to less compression).

With respect to the chosen scale function $k = k(q, \sigma)$, for every cluster $\mathcal{C}_i$ that is associated with its centroid c_i in the T-DIGEST data structure, we can define the *k-size*, denoted as $\mathrm{K}(c_i)$, which expresses the scaled length of the quantile range for the cluster:

$$\mathrm{K}(c_i) \coloneqq k\left(q(c_i), \sigma\right) - k\left(q(c_{i-1}), \sigma\right), i = 2 \ldots m \tag{5.6}$$

where $\mathrm{K}(c_1) \coloneqq k\left(q(c_1), \sigma\right)$.

To restrict the number of elements in a cluster in a way that depends on the quantile values it is responsible for, we can restrict its k-size and with non-linear scale functions we result in non-uniform clusters, having larger cluster sizes for the middle-range quantiles and smaller near the extrema (up to singleton clusters that contain only one element). Moreover, the t-digest algorithm is designed to build a *fully-merged* T-DIGEST data structure, meaning all clusters $\{\mathcal{C}_j\}_{j=1}^m$ satisfy the *digest property*:

$$\begin{cases} \mathrm{K}(c_i) \leq 1, & \text{(except singleton clusters)} \\ \mathrm{K}(c_i) + \mathrm{K}(c_{i+1}) > 1, & \end{cases} \tag{5.7}$$

which not only restricts the k-size of each cluster, but also ensures that any two adjacent clusters cannot be further merged.

In practice, to fulfill the constraints of the digest property (5.7), we do not need to recompute the k-size for each cluster on every change. Instead, since all centroids are sorted in the T-DIGEST, the number of elements in the cluster $\mathcal{C}_i$ can be limited by choosing a boundary for its estimated maximal quantile value:

$$q_{\text{limit}} = k^{-1}\left(k(q(c_i), \sigma) + 1, \sigma\right), \tag{5.8}$$

that for the scale function (5.5) has the following form:

$$q_{\text{limit}} = \frac{1}{2}\left[1 + \sin\left(\arcsin\left(2 \cdot q(c_i) - 1\right) + \frac{2\pi}{\sigma}\right)\right]. \tag{5.9}$$

Having the rules (5.8) to restrict the number of elements per cluster in T-DIGEST, we can formulate the merging t-digest algorithm as Algorithm 5.9, which is similar to the regular clustering procedure. To summarize input sequence of weighted data points $\{(x_1, x_1^{\text{count}}), (x_2, x_2^{\text{count}}), \ldots (x_b, x_b^{\text{count}})\}$, we sort them together with all centroids from the T-DIGEST data structure and, making a single pass through the resulting sequence X, we attempt to merge them successively if the digest property is not violated. We start with the left-most centroid, take its cluster as the current candidate cluster, and compute its boundary value q_{limit} by (5.8). Then, sequentially processing all centroids from X, we estimate their approximate quantile values and compare them to the boundary value. If absorbing of the pending centroid does not exceed the boundary value, we merge it into the candidate cluster and continue with the next centroid from the sequence X. Otherwise, meaning the maximal capacity of the candidate cluster is reached and no new elements can be added, we persist the current candidate cluster in the T-DIGEST data structure, emit a new candidate cluster with the pending centroid, and recompute the quantile boundary value q_{limit}. At the end, we receive a fully-merged T-DIGEST data structure.

Algorithm 5.9: Merging elements to t-digest

```
Input: Buffer B with elements $\{(x_1, x_1^{\text{count}}), (x_2, x_2^{\text{count}}), \ldots (x_b, x_b^{\text{count}})\}$
Input: t-digest data structure T-DIGEST
Input: Compression parameter $\sigma > 1$, scale function $k$
Input: t-digest data structure with merged buffer
$X \leftarrow$ sort(T-DIGEST $\cup$ B)
T-DIGEST $\leftarrow \varnothing$
$m \leftarrow$ count(X), $n \leftarrow \sum_{x_i \in X} x_i^{\text{count}}$
$c \leftarrow x_1,\ q_c \leftarrow 0$
$q_{\text{limit}} \leftarrow k^{-1}(k(q_c, \sigma) + 1, \sigma)$
for $i \leftarrow 2$ to $m$ do
    $\hat{q} \leftarrow q_c + \frac{1}{n} c^{\text{count}} + \frac{1}{n} x_i^{\text{count}}$
    if $\hat{q} \leq q_{\text{limit}}$ then
        $c^{\text{count}} \leftarrow c^{\text{count}} + x_i^{\text{count}}$
        $c \leftarrow c + x_i^{\text{count}} \cdot \frac{x_i - c}{c^{\text{count}}}$
        continue
    T-DIGEST $\leftarrow$ T-DIGEST $\cup \{(c, c^{\text{count}})\}$
    $q_c \leftarrow q_c + \frac{1}{n} c^{\text{count}}$
    $q_{\text{limit}} \leftarrow k^{-1}(k(q_c, \sigma) + 1, \sigma)$
    $c \leftarrow x_i$
T-DIGEST $\leftarrow$ T-DIGEST $\cup \{(c, c^{\text{count}})\}$ // last cluster usually is a singleton
return T-DIGEST
```

As we can see, every time when a new candidate cluster is emitted the boundary value q_{limit} has to be recomputed which involves expensive computation of the scale function and its inverse, according to (5.8) and (5.9). Luckily, the number of clusters is not too large in practice and various techniques have been suggested in order to optimize the boundary estimation, such as using efficient approximations for components of scale functions or roughly estimating the maximal number of elements that can be summarized into each cluster.

The complete algorithm to index a continuous data stream is based on the idea of processing streaming data by buffers of some fixed size and continuously merge them into the T-DIGEST data structure.

Algorithm 5.10: Stream processing with t-digest

```
Input: Data stream $\mathbb{D} = \{x_1, x_2, \ldots,\}$
Input: Buffer size $b$, compression parameter $\sigma > 1$, scale function $k$
Input: t-digest data structure
T-DIGEST $\leftarrow \varnothing$
while $\mathbb{D}$ do
    $\mathrm{B} \leftarrow \{(x_1, 1), (x_2, 1), \ldots, (x_b, 1)\}$
    T-DIGEST $\leftarrow$ Merge(T-DIGEST, B, $\sigma$, $k$)
return T-DIGEST
```

Note, the runtime costs of the buffer-and-merge Algorithm 5.10 are shared between frequent inserts of input elements into the buffer and rare calls of Algorithm 5.9. Since the inserts are cheap, the overall costs are dominated by the sort and the scale function invocations in the merging sub-algorithm that are amortized over several insertions.

Example 5.9: Indexing data stream with t-digest

Consider the dataset of $n = 20$ integers from Example 5.6:

$$\{0, 0, 3, 4, 1, 6, 0, 5, 2, 0, 3, 3, 2, 3, 0, 2, 5, 0, 3, 1\}.$$

As an example, let's take the compression parameter $\sigma = 5$, the buffer size $b = 10$, and the scale function as given by (5.5).

We populate the buffer B with the first ten elements from the input:

$$\mathrm{B} = \Big\langle (0, 1), (0, 1), (3, 1), (4, 1), (1, 1), (6, 1), (0, 1), (5, 1), (2, 1), (0, 1) \Big\rangle.$$

According to Algorithm 5.10, we need to join elements from the buffer and the centroids that are in the T-DIGEST. However, since the t-digest data structure is empty, the list of candidate centroids X contains only $n = 10$ elements from B, which we sort in ascending order:

$$\mathrm{X} = \Big\langle (0, 1), (0, 1), (0, 1), (0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1) \Big\rangle,$$

We select our first candidate cluster by taking the left-most centroid $(0, 1)$ and compute its quantile boundary value q_{limit} using the formula (5.9):

$$q_{\text{limit}} = \frac{1}{2} \left[1 + \sin\left(\arcsin(-1) + \frac{2\pi}{5} \right) \right] = 0.34549.$$

Then, we take the next element from X, which is again $(0, 1)$, and evaluate if it can be merged to the candidate cluster without violating the digest property. In our case it requires to estimate the maximal quantile value $\hat{q}$ of this merged cluster:

$$\hat{q} = \frac{(1+1)}{10} = 0.2,$$

and because it is below the quantile boundary value q_{limit}, we can freely merge the element $(0, 1)$ into the current candidate cluster, which now has $c^{\text{count}} = 2$ elements but, since the elements are identical, the centroid stays the same $c = 0$.

Similarly, we are able to merge the next element $(0, 1)$ to the candidate cluster, that changes only the number of summarized elements $c^{\text{count}} = 3$.

Next, we get the fourth element from X, which is $(0, 1)$, and follow the same procedure as above to check if it can also be merged into the candidate cluster. However, the estimated maximal quantile $\hat{q}$ of such merged cluster will become:

$$\hat{q} = \frac{(3+1)}{10} = 0.4,$$

that exceeds the current boundary value $q_{\text{limit}} = 0.34549$. Thus, we stop our attempts to absorb other centroids by the candidate cluster, store it into the T-DIGEST data structure

$$\text{T-DIGEST} = \Big\langle (0, 3) \Big\rangle,$$

and remember its maximal quantile value as $q = \frac{c^{\text{count}}}{n} = 0.3$.

From this moment we start to build a new candidate cluster from the current pending element $(0, 1)$, having $c = 0$ and $c^{\text{count}} = 1$, and its quantile boundary value is

$$q_{\text{limit}} = \frac{1}{2}\left[1 + \sin\left(\arcsin\left(2 \cdot 0.3 - 1\right) + \frac{2\pi}{5}\right)\right] = 0.874025.$$

Next, we take element $(1, 1)$ and if we merge it to the current candidate cluster, the estimated maximal quantile $\hat{q}$ will be

$$\hat{q} = 0.3 + \frac{(1+1)}{10} = 0.5,$$

that doesn't exceed the current boundary value. Thus, we merge the element $(1, 1)$ to the current cluster whose counts increase to $c^{\text{count}} = 2$ and the centroid becomes

$$c = 0 + \frac{1 - 0}{2} = 0.5.$$

Similarly, we process all the rest of the elements from X and the T-DIGEST data structure grows into:

$$\text{T-DIGEST} = \Big\langle (0, 3), (2, 5), (5, 1), (6, 1) \Big\rangle.$$

Continue processing the dataset, we fill a new buffer:

$$\mathrm{B} = \Big\langle (3, 1), (3, 1), (2, 1), (3, 1), (0, 1), (2, 1), (5, 1), (0, 1), (3, 1), (1, 1) \Big\rangle,$$

join it with the T-DIGEST data structure, and sort the resulting sequence X in ascending order of centroids:

$$\begin{aligned}\mathrm{X} = \Big\langle & (0, 3), (0, 1), (0, 1), (1, 1), (2, 5), (2, 1), (2, 1), \\ & (3, 1), (3, 1), (3, 1), (3, 1), (5, 1), (5, 1), (6, 1) \Big\rangle.\end{aligned}$$

Having that, we flush the T-DIGEST data structure and starting from the left-most elements attempt to merge sequentially elements and store clusters into T-DIGEST that they cannot be further merged.

At the end, the resulting T-DIGEST data structure consists of $m = 5$ clusters and has the following view:

$$\text{T-DIGEST} = \Big\langle (0.1667, 6), (2.36364, 11), (5, 1), (5, 1), (6, 1) \Big\rangle.$$

Because we lost information about exact elements clustered together (except the singleton clusters, where the centroid is the initial element), the T-DIGEST data structure provides a lossy representation of the data stream and to estimate quantiles and answer the Quantile query, we need to make an interpolation, taking into account distribution of clusters that has been produced by the chosen scale function.

Algorithm 5.11: Answering Quantile queries with t-digest

```
Input: t-digest data structure with $m$ clusters
Input: Value $q \in [0,1]$
Output: $q$-quantile
$n \leftarrow \sum_{j=1}^{m} c_j^{\text{count}}$
if $n \cdot q < 1$ then
    return $c_1$
if $n \cdot q > n - \frac{1}{2} c_m^{\text{count}}$ then
    return $c_m$
/* at this point, we can be sure that $\exists i \in [1,m) : q(c_i) + \frac{1}{2n} c_{i+1}^{\text{count}} > q$,   */
/* so the searched quantile is somewhere between $c_i$ and $c_{i+1}$                                          */
if $c_i^{\text{count}} = 1$ and $q(c_i) > q$ then
    return $c_i$
if $c_{i+1}^{\text{count}} = 1$ and $q(c_{i+1}) - \frac{1}{n} \leq q$ then
    return $c_{i+1}$
$\Delta_{\text{left}} \leftarrow (c_i^{\text{count}} = 1)\ ?\ 1 : 0$
$\Delta_{\text{right}} \leftarrow (c_{i+1}^{\text{count}} = 1)\ ?\ 1 : 0$
$w_{\text{left}} \leftarrow n \cdot q - n \cdot q(c_i) + \frac{c_i^{\text{count}} - \Delta_{\text{left}}}{2}$
$w_{\text{right}} \leftarrow n \cdot q(c_i) - n \cdot q + \frac{c_{i+1}^{\text{count}} - \Delta_{\text{right}}}{2}$
return $\frac{c_i \cdot w_{\text{right}} + c_{i+1} \cdot w_{\text{left}}}{w_{\text{left}} + w_{\text{right}}}$
```

Thus, to find the q-quantile from the T-DIGEST data structure, we calculate the rank of the searched element x in this sorted sequence, which is $n \cdot q$, where n is the total number of elements summarized into t-digest data structure. If this rank is below one, we report the centroid c_1 as the quantile. Similarly, if the rank is within a half of the last cluster of the the maximal count or even above, we return c_m that the maximal element in the digest. Otherwise, we search for clusters $\mathcal{C}_i$ and $\mathcal{C}_{i+1}$ whose estimated quantile values given by formula (5.4) encircle the given quantile value q. When the left cluster $\mathcal{C}_i$ is a singleton and its maximal quantile surpasses q, we return centroid c_i as the quantile that is the element that was actually summarized into the cluster. Similarly, if the right cluster $\mathcal{C}_{i+1}$ is a singleton and its estimated minimal quantile value falls behind the given quantile value q, meaning this cluster is responsible for the searched quantile, we report centroid c_{i+1} as the best estimation to it. Otherwise, we compute weights by evaluating the contribution of each such cluster and build an interpolation by taking the weighted average of the centroids from both clusters, which is reported as the q-quantile.

The quantile estimation algorithm is dependent on the choice of the scale function and with more aggressive functions that produce a bigger tail of singleton clusters at the edges, it can be tuned to improve accuracy for extreme quantiles [Du18]. Additionally, it is advised to persist the minimal and maximal elements during the indexing to use them in interpolation.

Example 5.10: Quantile query with t-digest

We perform the Quantile query to calculate the 0.65-quantile from the T-DIGEST data structure built in Example 5.9:

$$\text{T-DIGEST} = \Big\langle (0.1667, 6), (2.36364, 11), (5, 1), (5, 1), (6, 1) \Big\rangle.$$

The total number of elements in the T-DIGEST is the sum of counts from all clusters, that is $n = 20$ in our case. The rank of the searched quantile x is $n \cdot q = 20 \cdot 0.65 = 13$, which is neither smaller than one, nor too close to n, therefore, we start searching for two consecutive clusters $\mathcal{C}_i$ and $\mathcal{C}_{i+1}$, whose centroids will encircle the searched quantile x. In the current T-DIGEST data structure, these are $\mathcal{C}_2$ and $\mathcal{C}_3$, because

$$q(c_2) + \frac{1}{2 \cdot 20} c_3^{\text{count}} = \frac{6 + 11}{20} + \frac{1}{40} = 0.875 > 0.65,$$

as it required according to Algorithm 5.11.

Since the cluster $\mathcal{C}_3$ is a singleton, meaning it has $c_3^{\text{count}} = 1$, we need to compare its maximal quantile value to the searched quantile value q in order to check if its centroid can be the best fit. Thus, we calculate

$$q(c_3) + \frac{1}{20} = \frac{6 + 11 + 1}{20} + \frac{1}{20} = 0.95,$$

that is significantly surpasses the value $q = 0.65$ and we conclude that the actual quantile is located somewhere in between the centroids c_3 and c_2, but remember that the right cluster is a singleton by setting $\Delta_{\text{right}} = 1$.

Therefore, the searched quantile can by estimated by the interpolation using the weighted average of the centroids, where the weights are

$$\begin{aligned} w_{\text{left}} &= 20 \cdot 0.65 - 20 \cdot q(c_2) + \frac{c_2^{\text{count}} - 0}{2} = 13 - 20 \cdot \frac{6+11}{20} + \frac{11}{2} = 1.5, \\ w_{\text{right}} &= 20 \cdot q(c_2) - 20 \cdot 0.65 + \frac{c_3^{\text{count}} - 1}{2} = 20 \cdot \frac{6+11}{20} - 13 + \frac{1-1}{2} = 4. \end{aligned}$$

Finally, the estimated 0.65-quantile for the dataset of Example 5.9 is

$$x = \frac{c_2 \cdot w_{\text{right}} + c_3 \cdot w_{\text{left}}}{w_{\text{left}} + w_{\text{right}}} = \frac{2.36364 \cdot 4 + 5 \cdot 1.5}{1.5 + 4} = 3.08,$$

which is pretty close to the exact value of 3 for that dataset.

Similar to the Quantile query, we can use the T-DIGEST data structure to answer the Inverse quantile query and find the rank of some given element x. We start with the comparison of the element x to the minimal and maximal centroids in the T-DIGEST data structure, which are the left-most and right-most clusters, accordingly. If x falls outside that range, we just report either 1 or the total number of elements n as the estimated rank value, depending on the side the element appear. Otherwise, we search for the element x through the centroids and if such clusters are found, we accumulate their counts and report the rank(x) as the rank of the cluster with the smallest index adjusted by that amount. If neither of the checks above are succeeded, we can be sure that the element x falls in between centroids of some consecutive clusters, say (c_i, c_{i+1}), and its rank is already at least $n \cdot q(c_i)$. If both of these clusters are singletons, meaning their centroids are exactly the input elements that were summarized, we do not need to correct that value and return it as the searched rank. When only one of those clusters is a singleton, we fine-tune the rank by the scaled contribution of another cluster to get the final value. Otherwise, we build an interpolation using the both cluster sizes and adjust the guaranteed rank value.

Algorithm 5.12: Answering Inverse quantile queries with t-digest

```
Input: Element $x$
Input: t-digest data structure with $m$ clusters
Output: Rank of element
$n \leftarrow \sum_{j=1}^{m} c_j^{\text{count}}$
if $x < c_1$ then
    return 1
if $x > c_m$ then
    return $n$
/* check if $x$ is one of the centroids */
if $\exists j : c_j = x$ then
    $\text{J} \leftarrow \{j : c_j = x\}, i^* \leftarrow \min(\text{J})$
    return $n \cdot q(c_{i^*}) - c_{i^*}^{\text{count}} + \frac{1}{2} \sum_{j \in \text{J}} c_j^{\text{count}}$
/* at this point, we can be sure that $\exists i \in [1, m) : x \in (c_i, c_{i+1})$ */
$rank \leftarrow n \cdot q(c_i)$
if $c_i^{\text{count}} = 1$ and $c_{i+1}^{\text{count}} = 1$ then
    return $rank$
$\hat{x} \leftarrow \frac{x - c_i}{c_{i+1} - c_i}$
if $c_i^{\text{count}} = 1$ then
    return $rank + \frac{\hat{x}}{2} \cdot c_{i+1}^{\text{count}}$
if $c_{i+1}^{\text{count}} = 1$ then
    return $rank - \frac{(1-\hat{x})}{2} \cdot c_i^{\text{count}}$
return $rank + \frac{\hat{x}}{2} \cdot c_{i+1}^{\text{count}} - \frac{(1-\hat{x})}{2} \cdot c_i^{\text{count}}$
```

As mentioned earlier, to answer the Range query, it is simple enough to perform two Inverse quantile queries and find the difference between the ranks of the range borders.

Properties

There is a clear trade-off between the size of the T-DIGEST data structure as controlled by the compression parameter σ, the speed, and the accuracy to which the quantiles are estimated. Thus, with a smaller value of σ and a large buffer size b, we can achieve higher speed with constant memory usage. For highest accuracy, it is preferred to use larger σ to have less compression and a bigger buffer (e.g., $10 \times \sigma$), while for the smallest memory — a smaller buffer and larger values of the compression parameter σ.

As shown by the t-digest authors, when using the scale function (5.5), the number of clusters m in the T-DIGEST data structure that satisfies the digest property (5.7) and indexed $n \geq \frac{\sigma}{2}$ elements is in the range of

$$\left\lfloor \frac{\sigma}{2} \right\rfloor \leq m \leq \lceil \sigma \rceil. \tag{5.10}$$

Example 5.11: Estimate required space

For example, we want to index at least $n = 1000$ elements with the compression parameter $\sigma = 100$. Therefore, according to (5.10), we can expect from 50 to 100 clusters in the fully-merged T-DIGEST.

In the T-DIGEST data structure each cluster is represented by its centroid and the number of indexed elements. Thus, having 32-bit counters and double precision 64-bit floating point number for the centroid value, the entire centroid requires 12 bytes of memory and the whole data structure fits in about 1.2 KB of memory.

For high accuracy we typically use buffer ten times bigger the compression parameters and having $b = 10 \cdot \sigma = 1000$ we can allocate smaller 16-bit counters and double precision 64-bit floating point numbers for elements in the buffer, that end up in additional 10 KB of memory in runtime.

The t-digest algorithm maintains accuracy ε in q-quantile estimation that is proportional to $q \cdot (1-q)$ and, in contrast to other algorithms which maintain only the constant absolute error, in the t-digest the relative error is bounded that makes it resistant to significant errors for extreme quantiles. The advantage of the t-digest over the q-digest is also that it can handle floating point values while the q-digest, as we have already seen, is limited to integers only.

Two t-digest data structures can be easily merged using the same algorithm, but the resulting data structure is not the same as the T-DIGEST built for the joined input stream. However, the empirical results show that it provides a good estimation to that value, so it is possible to compose t-digests for different parts of the data stream in parallel and combine them to answer rank queries. This makes the algorithm parallel friendly and useful in MapReduce and stream mining tasks for Big Data applications.

The t-digest algorithm has become more and more popular these days. For instance, it is used in the percentiles aggregation in Elasticsearch and also available in stream-lib and Apache Mahout.

Conclusion

In this chapter we covered efficient algorithms and data structures that are widely used to calculate rank-based characteristics of the data using a small amount of memory. We studied a popular sampling algorithm, well-known tree-based stream summary algorithm as well as its modern alternative that is based on one-dimensional clustering. With these algorithms we can find ranks of elements in a data stream, various quantiles and execute range queries.

If you are interested in more information about the material covered here or want to read the original papers, please take a look at the list of references that follows this chapter.

In the next chapter we consider the similarity problem, one of the fundamentals in data analysis. We study different similarity definitions and efficient probabilistic algorithms that approach the problem of ascertaining the most similar documents for a given document across huge datasets.

Bibliography

[Ma99] Manku, G., et al. (1999) “Random sampling techniques for space efficient online computation of order statistics of large datasets”, *Proceedings of the 1999 ACM SIGMOD International conference on Management of data*, Philadelphia, Pennsylvania, USA - May 31–June 03, 1999, pp. 251–262, ACM New York, NY.

[Co06] Cormode, G., et al. (2006) “Space- and time-efficient deterministic algorithms for biased quantiles over data streams”, *Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, Chicago, IL — June 26–28, 2006, pp. 263–272, ACM New York, NY.

[Co08] Cormode, G., Hadjieleftheriou, M. (2008) “Finding frequent items in data streams”, *Proceedings of the VLDB Endowment*, Vol. 1 (2), pp. 1530–1541.

[Du18a] Dunning, T. (2018) “The Size of a t-Digest”, *github.com*, https://github.com/tdunning/t-digest/blob/779ab7b/docs/t-digest-paper/sizing.pdf, Accessed Jan. 19, 2019.

[Du14] Dunning, T., Ertl, O. (2014) “Computing Extremely Accurate Quantiles Using t-Digests”, *github.com*, https://github.com/tdunning/t-digest/blob/t-digest-1.0/docs/theory/t-digest-paper/histo.pdf, Accessed Jan. 12, 2019.

[Du18] Dunning, T., Ertl, O. (2018) “Computing Extremely Accurate Quantiles Using t-Digests”, *github.com*, https://github.com/tdunning/t-digest/blob/779ab7b/docs/t-digest-paper/histo.pdf, Accessed Jan. 19, 2019.

[Gr01] Greenwald, M., Khanna, S. (2001) “Space-Efficient Online Computation of Quantile Summaries”, *Proceedings of the 2001 ACM SIGMOD International conference on Management of data*, Santa Barbara, California, USA - May 21–24, 2001, pp. 58–66, ACM New York, NY.

[Lu16] Luo, G., Wang, L., Yi, K. et al. (2016) “Quantiles over data streams: experimental comparisons, new analyses, and further improvements”, *The VLDB Journal*, Vol. 25 (4), pp. 449–472.

[Sh04] Shrivastava, N., et al. (2004) “Medians and Beyond: New Aggregation Techniques for Sensor Networks”, *Proceedings of the 2nd International conference on Embedded networked sensor systems*, Baltimore, MD, USA - November 03–05, 2004, pp. 58–66, ACM New York, NY.

[Wa13] Wang, L., et al. (2013) “Quantiles over data streams: an experimental study”, *Proceedings of the 2013 ACM SIGMOD International conference on Management of data*, New York, NY, USA - June 22–27, 2013, 2013, pp. 737–748, ACM New York, NY.

6 Similarity

Similarity is a fundamental data analysis problem that has attracted a lot of research effort in the last two decades. While talking about relations of two documents[1], we are mostly interested in concepts such as *roughly the same* and in finding a way to express *similarity* numerically.

The similarity plays an important role for Big Data applications and can be used to reduce the processing time and computation efforts. For instance, with its help, we can eliminate data that has already been processed even if it doesn't have the same form as before. Another example is the development of different sampling techniques to handle

[1] "Documents" can be objects of any nature, e.g., texts, images, etc.

large volumes of data, that are sometimes unfeasible to process. When handling data from a number of classes, instead of just taking every n-th document from the dataset (which can result in unbalanced processing of the classes), we can develop a similarity measure to group documents of one class together and process equal subsets from each class to keep the processing balanced.

Example 6.1: DNA sequences (Xie et al., 2015)

The rapid development of DNA sequencing technologies in recent years has led to a huge number of discovered DNA sequences. Evaluation of the similarity between them is a crucial starting point for analyzing genomic information and has a wide range of applications. However, DNA databases have a huge number of documents, where the same data can be stored in various different forms and an efficient search for similar sequences is essential.

The most well-known similarity-related problem is to find a *nearest neighbor* for a given document, meaning the document that is most similar to it across the dataset. Having an efficient algorithm for the nearest neighbor search in a large database can speed up, by several orders of magnitude, many important applications like document retrieval, image matching, etc.

The naive solution is to use a linear scan, iterate over all existing documents, and compare them to the given document. Such an approach guarantees to find the exact nearest neighbor of any query object, but requires $O(n)$ time, where the number of pairs n is huge. In high-dimensional spaces, the problem of the nearest neighbor search becomes even more difficult.

Thus, we are looking for sublinear time solutions that approximately find the nearest neighbor, that is suitable in most practical cases. In practice, we are interested in solving an approximate nearest neighbor problem or, more formally, a ε–*Nearest neighbor problem* to find with some high probability $1-\varepsilon$ the nearest neighbor for a given document in a large database.

The immediate application of the nearest neighbor search is the detection of duplicates (exact and non-exact), a task to find documents that at some level are similar to the given document.

Example 6.2: Intellectual property (Broder et al., 1997)

The detection of duplicates, illegal copies, or modifications, is very important in intellectual properties protection and plagiarism prevention.

Given a source document, we can perform a nearest neighbor search to find other documents that are similar to it, in whole or part, that have been substantially copied or minorly edited.

Another important application of the nearest neighbor problem is *clustering*, a task of grouping documents in a way that documents in the group (cluster) are more similar to each other than to other documents outside the group, in other words, to group the nearest documents together.

Conceptually, to find similar documents in a dataset, we need to compare each document to each document, which requires the evaluation of about the quadratic number of pairs. Thus, for 1 million documents there are about 500 billion ($5 \cdot 10^{11}$) pairs and, judging 10^6 pairs per second, it takes almost six days to process all those documents, which is unpractical.

Since the similarity problem itself is fuzzy, it is natural to use probabilistic algorithms to solve it fast and efficiently.

Jaccard (resemblance) similarity

While it is not immediately clear how to express the similarity between documents of arbitrary representation, mathematics has already developed a solid theory for set similarity. Thus, representing documents as collections of some features, the document similarity problem can be converted to a set intersection problem and evaluated, for instance, by a random sampling that can be done independently for each document.

There are many different ways to represent a document of any nature as a set. Generally speaking, we need to identify important document characteristics that describe it in the best way and represent the document as a simple collection of those *features*. To be able to compare documents with higher effectiveness, it is important to define a *canonical collection of features* which stay the same for documents that differ only in information that is usually ignored as meaningless (e.g., for text documents, we often ignore punctuation, capitalizations, formatting, and so on). The step of the preprocessing of documents to their canonical form is called *document normalization*.

Example 6.3: Features for music tracks

In the task of finding audio matches, we want to use features that are robust to the common types of abuse that are performed on audio before it reaches our ears. For instance, we can note the peaks in the spectrum and encode their positions in time and space as a collection of signatures that describe the particular audio.

In contrast, for songs, we can extract features based on mel-frequency cepstral coefficients (MFCCs), which are a short-time spectral decomposition of a musical clip that conveys the general frequency characteristics important to human hearing. Representing a song as a collection of MFCC frames, we can consider two songs similar if they have the same frames regardless of the order.

Another example shows how text documents can be treated.

Example 6.4: Shingling technique for text documents

For text documents, the most well-known method to represent them as collections of features is *shingling*, where *shingle* is a contiguous subsequence contained in a document. Specifically, every document can be associated with a collection of w*-shingles*, that includes all shingles of some predefined size w contained in the document.

For example, consider a text document *"The quick brown fox jumps over the lazy dog"*. We can build shingles of size $w = 6$ from the sequence of characters which are

"the qu", "he qui", "e quic", " quick", "quick ", "uick b", "ick br", "ck bro", "k brow", and so on.

Another approach is to use word tokenization, for that our example can be reduced to a simple split of the document by spaces, and we build shingles from the sequence of words. For example, 3-shingles (3-grams) will be

"the quick brown", "quick brown fox", "brown fox jumps", "fox jumps over", "jumps over the", "over the lazy", "the lazy dog".

Unfortunately, the length of the shingles can vary by a wide range, and it can be tough to allocate a space-efficient data structure.

Instead, we can convert shingles to fixed-length entities by applying a classical hash function that hashes to the desired number of bits, e.g., 8-bit values. This approach has some additional tiny probability of collision, but can drastically reduce the required space.

If two documents d_A and d_B are represented as collections of features, we can mathematically calculate their resemblance as a *Jaccard similarity* $\mathrm{J}(d_\mathrm{A}, d_\mathrm{B})$ which indicates the ratio of common features in both documents, and produces a number between zero and one, such that it is close to one for the documents that are roughly the same:

$$\mathrm{J}(d_\mathrm{A}, d_\mathrm{B}) = \frac{|d_\mathrm{A} \cap d_\mathrm{B}|}{|d_\mathrm{A} \cup d_\mathrm{B}|}. \tag{6.1}$$

The Jaccard similarity of exact duplicates is equal to one, and we can consider documents as nearest neighbors if their resemblance exceeds a certain given threshold $0 < \theta < 1$.

In reality with high volumes of documents to compute the Jaccard similarity for, it suffices to keep a relatively small fixed-size sketch for each document. Such sketches can be produced very fast (linear on document size) and, given two sketches, the Jaccard similarity can be computed in linear time based on the size of the sketches.

Example 6.5: Jaccard similarity

Medical symptoms can be naturally used as features for diseases. Consider five well-known illnesses together with their most common symptoms[2]:

	Disease	Symptoms
d_1	allergic rhinitis	*sneezing, itchiness, runny nose*
d_2	common cold	*runny nose, sore throat, headache, muscle aches, cough, sneezing, fever, loss of taste*
d_3	flu	*fever, aching body, feeling tired, cough, sore throat, headache, difficulty sleeping, loss of appetite, diarrhea, nausea*
d_4	measles	*runny nose, cough, red eyes, fever, greyish-white spots, rash*
d_5	roseola	*fever, runny nose, cough, diarrhea, loss of appetite, swollen glands, rash*

Intuitively, we can expect that the *common cold* is a bit more similar to the *flu*, than to *roseola*; *roseola* should be similar to *measles*, and *allergic rhinitis* should be quite different from the others. Let's compute Jaccard similarities for these documents.

Documents d_2 and d_3 have 14 different symptoms in total while sharing only 4 of them (*cough*, *fever*, *headache*, *sore throat*); thus, the similarity is equal to 0.2857, which is about 29%:

$$\mathrm{J}(d_2, d_3) = \frac{4}{14} = 0.2857.$$

Next, we compare documents d_4 and d_5 that have 9 different symptoms in total and 4 in common, so the similarity is 44%:

$$\mathrm{J}(d_4, d_5) = \frac{4}{9} = 0.44.$$

Comparing d_1 to d_3 gives us no common symptoms, so $\mathrm{J}(d_1, d_3) = 0$ and they are two different diseases that cannot be accidentally mixed up.

Once every document is represented as a collection of features, we have the set of all features across all documents which is called

[2]Find more conditions and treatments at NHS Choices `https://www.nhs.uk`

the *universal set* Ω. This feature set can be seen as a bit-array, where set bits indicate that the corresponding feature from the universal set is present in the document.

The universal set is usually much bigger than the collection of features from a particular document, therefore the document bit-arrays have much more unset bits that set ones (very sparse).

Example 6.6: Document bit-array

Consider the list of diseases from Example 6.5. The universal set Ω for these documents includes all the different symptoms mentioned in the documents (in practice, it should consist of all possible medical symptoms). We can enumerate those symptoms in some particular order, e.g., alphabetically:

Index	Symptom
0	aching body
1	cough
2	diarrhea
3	difficulty sleeping
4	feeling tired
5	fever
6	greyish-white spots
7	headache
8	itchiness
9	loss of appetite
10	loss of taste
11	muscle aches
12	nausea
13	rash
14	red eyes
15	runny nose
16	sneezing
17	sore throat
18	swollen glands

The bit-array that corresponds to document d_3 (*flu*) and the order of features has the following form:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	1	1	1	1	1	0	1	0	1	0	0	1	0	0	0	0	1	0

Thus, the set bit in position 5 (corresponding to *fever*) means that it is a symptom for the *flu*, while the unset bit in position 13 indicates that *rash* is not a symptom.

The *Jaccard similarity* between two document bit-arrays is the ratio between the number of bits that are set for both documents (i.e., have ones in the same bit-positions) to the number of bits that are set for either one or the other document.

A binary representation of the documents encodes only the fact of the feature's existence in the document, but cannot answer questions about how frequently the feature appears and does not support feature prioritizing. For instance, in Example 6.5 many diseases have a *cough*, *fever*, and *runny nose* as symptoms because it is just the way our body protects itself regardless of the particular illness. However, this makes many different diseases a bit more similar to each other and to identify "truly" similar documents we need to use different approaches, for instance the TF–IDF model, which prioritize more unique terms between documents, and represents documents as dense vectors of the features' weights. Unfortunately, the Jaccard similarity defined by (6.1) cannot be applied in this case, and we need to go for other similarity definitions, such as the Ruzicka similarity or the cosine similarity.

Cosine similarity

Another view on mathematical formalization of documents is to represent them as *dense* vectors of weighted features, where the weights could highlight the importance of the features.

Text documents are the main targets of such formalization due to the popularity of the Vector Space Model[3], that provides a representation for such documents as dense vectors of identifiers. For instance, the *term frequency – inverse document frequency model* (TF–IDF) considers documents as dense vectors of term weights, built as a relative frequency of the term in the document (term frequency, TF), normalized by the relative number of documents in the dataset that contains the term (inverse document frequency, IDF).

[3]G. Salton et al., A vector space model for automatic indexing, 1975

Example 6.7: Vector Space Model

Consider the documents from Example 6.5 and let's build a real-valued representation of them using the TF–IDF model. We treat every symptom s_j as a term and compute its weight w_j for the document based on the occurrence of the term in the dataset. The idea is to prioritize terms that occur more in the particular document but are very rare in the whole dataset, which could be an indicator that they better characterize the document. In our case, all symptoms occur exactly one or zero times in the documents, so instead of using the pure frequency, we use the features frequency adjusted for document length, the *relative frequency* f_j^d of the symptom in the document. To make the results more visual, we additionally scale the output and round the weights to integers:

$$w_j = 100 \cdot f_j^d \cdot \log \frac{n}{n_j},$$

where n_j is the number of documents that contain feature s_j and n is the total number of documents in the dataset.

Like in Example 6.6, we can define a universal set Ω and enumerate all the different symptoms alphabetically. Thus, we end up with 19 unique features which will induct the dimensionality of our document vectors.

Feature	Symptom	Number of documents
s_0	aching body	1
s_1	cough	4
s_2	diarrhea	1
s_3	difficulty sleeping	1
s_4	feeling tired	1
s_5	fever	4
s_6	greyish-white spots	1
s_7	headache	2
s_8	itchiness	1
s_9	loss of appetite	2
s_{10}	loss of taste	1
s_{11}	muscle aches	1
s_{12}	nausea	1
s_{13}	rash	2

Feature	Symptom	Number of documents
s_{14}	red eyes	1
s_{15}	runny nose	4
s_{16}	sneezing	2
s_{17}	sore throat	2
s_{18}	swollen glands	1

Consider document d_3 (*flu*) and build its representation as a vector of weights of the document features. The feature s_0 (*aching body*) is one out of the 10 features for document d_3, its relative frequency is $f_0^3 = \frac{1}{10} = 0.1$; since no other document from the dataset contains that feature, $n_0 = 1$ and the total number of documents $n = 5$:

$$w_0^3 = 0.1 \cdot \log \frac{5}{1} \approx 16.$$

Similarly, feature s_1 (*cough*) appears once in the document, so $f_1^3 = 0.1$, but it is contained also in four documents in the dataset, so $n_1 = 4$ and its weight is

$$w_1^3 = 1 \cdot \log \frac{5}{4} \approx 2.$$

We can continue processing all features and if some feature from the universal set is not present in the document, its weight is equal to zero regardless of other counts.

Thus, the final real-valued vector representation of document d_3 is

16	2	9	16	16	2	0	9	0	9	0	0	16	0	0	0	0	9	0

One of the popular similarity measures in the area of document vectors, the *cosine similarity* $c(d_A, d_B)$, is the value of the angle $\alpha = \alpha(d_A, d_B)$ between two documents that are represented as two non-zero vectors:

$$c(d_A, d_B) = \cos(\alpha) = \frac{d_A \cdot d_B}{\|d_A\|_2 \cdot \|d_B\|_2}. \tag{6.2}$$

The cosine similarity focuses on the orientation of the document vectors, not on their magnitude. If two document vectors are orthogonal in the space (therefore, such documents are completely non-related), the angle between them is 90° and the cosine similarity is *cos* 90° = 0.

On the other hand, if the angle between the document vectors is close to $0°$, the documents are roughly the same and their cosine similarity is close to one.

Although the cosine function can take values from $[-1, 1]$, in most information retrieval problems document vectors have only positive components, so the angle doesn't exceed $90°$ and the cosine similarity only has values from $[0, 1]$.

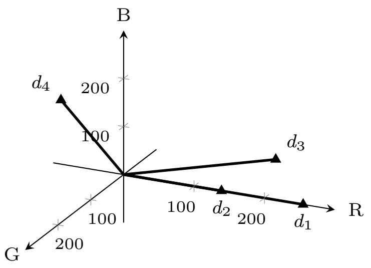
Example 6.8: Cosine similarity

Consider the RGB color space that defines the chromaticity of *red* (R), *green* (G), and *blue* (B). Every supported color can be represented with non-negative 8-bit values of R, G, and B. For instance, *red* has the maximum value 255 in the R channel and zeros in other channels.

Many people natively can identify similar colors, but let's estimate their cosine similarities. Consider the list of colors in the table below.

	Color	R	G	B
d_1	red	255	0	0
d_2	dark red	139	0	0
d_3	ruby	224	17	95
d_4	deep sky blue	0	191	255

Alternatively, we can draw them as position vectors in three-dimensional RBG space.



We compute the similarity between documents d_1 (*red*) and d_2 (*dark red*) which intuitively must be very similar:

$$c(d_1, d_2) = \frac{255 \cdot 139 + 0 \cdot 0 + 0 \cdot 0}{\sqrt{255^2 + 0^2 + 0^2} \cdot \sqrt{139^2 + 0^2 + 0^2}} = 1.$$

Thus, according to the cosine similarity, the documents are exactly the same, this is due to the fact that for cosine similarity only the orientation is important (the fact, that both of them have values only in the R channel), but not the magnitude of the vectors (the actual value in the channel).

Next, consider the document d_4 (*deep sky blue*), that must be strong contrast to d_1 (*red*), and compute the cosine similarity between them:

$$c(d_1, d_4) = \frac{255 \cdot 0 + 0 \cdot 191 + 0 \cdot 255}{255 \cdot 255} = 0.$$

Definitely, these documents are orthogonal which confirms the zero cosine similarity.

Now, consider the document d_3 (*ruby*), that has values in all channels, and find which color it is more similar to:

$$c(d_3, d_4) = \frac{224 \cdot 0 + 17 \cdot 191 + 95 \cdot 255}{\sqrt{224^2 + 17^2 + 95^2} \cdot \sqrt{0^2 + 191^2 + 255^2}} = 0.32,$$

and

$$c(d_3, d_1) = \frac{224 \cdot 255 + 17 \cdot 0 + 95 \cdot 0}{\sqrt{224^2 + 17^2 + 95^2} \cdot 255} = 0.92.$$

Thus, *ruby* is more similar to red than to *deep sky blue*, which is predictable since it is a representation of the color of the cut and polished ruby gemstone and is a shade of red.

Now we study a generic framework for efficient searching of near-duplicate documents and then we go to its well-known implementations regarding different definitions of similarity.

6.1 Locality–Sensitive Hashing

Locality–Sensitive Hashing (LSH) was proposed by Piotr Indyk and Rajeev Motwani in 1998 [In98] as a family of functions with the property that similar input objects (from the domain of such functions) have a higher probability of colliding in the range space than dissimilar ones.

Intuitively, the LSH is based on the simple idea that if two documents of any nature are close together, then, after applying those hash functions, the resulting hash values of these documents will remain close as well.

Locality-sensitive hash functions radically differ from conventional hash functions because they have the goal of maximizing the probability of a collision of similar items, while others try to minimize it. If we consider two documents that are different just by a single byte and apply any conventional hash function, for instance, MurmurHash3 or MD5, the hash values will be completely different, because the goal of those hash functions is to maintain a low probability of collision.

In order to construct locality-sensitive hash functions that preserve similarity between documents, it is necessary to know how to measure such similarity $\mathrm{Sim}(d_\mathrm{A}, d_\mathrm{B})$ and distinguish similar objects using a certain threshold θ.

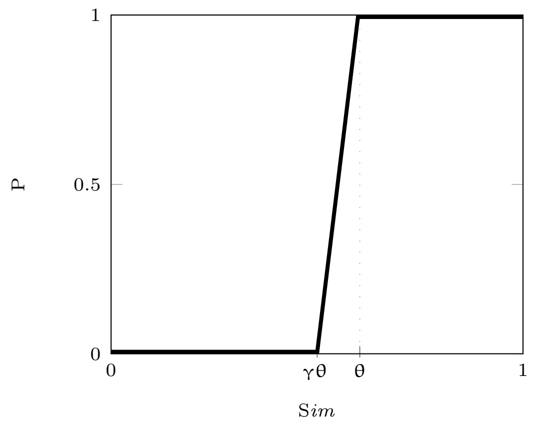
The similarity measure should be chosen based on the particular practical problem and different similarity measures induce different LSH function families. However, not every similarity measure can be used to build locality-sensitive hash functions, for instance, it has been proven that it is impossible to construct them for such popular metrics as the Dice coefficient and the Overlap coefficient.

The locality-sensitive hash function h is a function that maps every document from the dataset, presuming similarity between documents in the way that the probability of collision P is higher for similar documents:

$$\begin{cases} \Pr\left(h(d_{\mathrm{A}}) = h(d_{\mathrm{B}})\right) \geq p_1, & \text{if } \mathrm{Sim}(d_{\mathrm{A}}, d_{\mathrm{B}}) \geq \theta, \\ \Pr\left(h(d_{\mathrm{A}}) = h(d_{\mathrm{B}})\right) \leq p_2, & \text{if } \mathrm{Sim}(d_{\mathrm{A}}, d_{\mathrm{B}}) \leq \gamma\theta, \end{cases} \tag{6.3}$$

where $0 < \gamma < 1$ and $0 \leq p_2 < p_1 \leq 1$.

The closer γ is to one, the better the function, the smaller error in the similarity detection.



The Locality–Sensitive Hashing algorithm is a generic schema that solves similarity problems with the help of locality-sensitive hash functions that have been built for the chosen similarity measure.

Algorithm 6.1: Locality-sensitive bucketing

```
Input: Dataset $\mathbb{D} = \{d_1, d_2, \dots d_n\}$
Input: Family of LSH functions $\mathrm{H}^{\theta}_{\mathrm{Sim}}$
Output: LSH hash table with documents grouped into buckets
$\mathrm{T} \leftarrow \varnothing$
$h \sim \mathrm{H}^{\theta}_{\mathrm{Sim}}$
for $d \in \mathrm{D}$ do
    $key \leftarrow h(d)$
    $\mathrm{T}(key) \leftarrow \mathrm{T}(key) \cup \{d\}$
return $\mathrm{T}$
```

The simple idea is to map documents, using locality-sensitive hash functions, to a limited number of buckets where similar documents appear in the same bucket with a higher probability. Such buckets can be organized in a hash table, where each of them is indexed by its hash value, and we can search for near documents via a hash table lookup.

However, as we see from (6.3), locality-sensitive hash functions are not exact, which means false positive and false negative events can occur.

False positive events in this generic schema occur when two dissimilar documents (whose similarity measure does not exceed the threshold θ) appear in the same bucket. This type of error can be eliminated by calculating the exact similarities for documents in the bucket and comparing them to the given threshold.

More difficult are false negatives, when two similar documents end up in different buckets. This cannot be avoided, but to minimize their number we can build k different hash tables using randomly selected distinct LSH functions from the same family that map to the same set of bucket keys. In other words, we are increasing the number of estimators for each bucket that can boost the accuracy.

Algorithm 6.2: Finding similar documents

```
Input: Document $d$, dataset $\mathbb{D}$
Input: LSH hash table T with documents grouped into buckets
Input: Similarity threshold $\theta$
Output: Similar documents
$S \leftarrow \varnothing$
for $key \in T$ do
    if $d \notin T(key)$ then
        continue
    for $c \in T(key)$ do
        if $\mathrm{Sim}(d, c) \geq \theta$ then
            $S \leftarrow S \cup \{c\}$
return S
```

Since locality-sensitive hash functions focus on preserving similarities, we can expect that hash functions will map similar documents to the same bucket at least once. The resulting buckets can be built from the documents that appear together at least once.

Of course, this technique increases the number of false positive errors, but they can be eliminated with high confidence, as we described above.

Algorithm 6.3: Locality–Sensitive Hashing algorithm

```
Input: Document $d$
Input: LSH hash table T with documents grouped into buckets
Input: Similarity threshold $\theta$
Output: Similar documents
$S \leftarrow \varnothing$
for $i \leftarrow 1$ to $k$ do
    $T_i \leftarrow \textbf{Bucketing}(\mathbb{D}, H^{\theta}_{\text{Sim}})$
$T := \bigcup_{i=1}^{k} T_i$
for $key \in T$ do
    if $d \notin T(key)$ then
        continue
    for $c \in T(key)$ do
        if $\text{Sim}(d, c) \geq \theta$ then
            $S \leftarrow S \cup \{c\}$
return S
```

The performance of the LSH algorithm depends on a proper choice of θ and k. Bad choices for these parameters could result in too few documents in the hash buckets leading to incorrect grouping, or too many documents leading to an increased time for exact similarity computation at the final step.

The Locality–Sensitive Hashing algorithm is a framework to solve the Nearest neighbor problem, it has different implementations based on the chosen similarity measure. For instance, for the regular Euclidian distance it can be implemented as Random Projections, for Jaccard similarity as minwise hashing (MinHash), and for cosine similarity as SimHash that we study in detail in the next few sections.

Nearest neighbors search

When we need to find the nearest neighbors for a given document from the dataset distributed into buckets by the LSH algorithm, we apply the same locality-sensitive hash functions to that document and get the number of relevant mapped buckets. Documents in those buckets are the candidates for the nearest neighbors, and we compute the exact similarity between them and the given document, filtering by the comparison to the similarity threshold θ.

In practical applications, there is a huge number of documents and the problem of searching in a LSH hash table becomes challenging. There are many approaches to handling it, but most of them introduce additional measures that can help to store hash table keys in an optimized order to improve the table lookup.

For instance, *SortingKeys–LSH*, invented by Yingfan Liu et al. in 2014 [Li14], improves the search by minimizing random I/O operations when retrieving candidate documents. The authors defined a custom distance measure for the hash table keys and proposed to sort those keys in a special linear order associated with that distance. Following that order, the candidate documents can be stored closely in the memory or on the disk. When a new document arrives, we need to retrieve only the documents for the close hashes according to the introduced distance measure, and can find the candidates faster due to the reduction of random I/O operations and higher search accuracy.

6.2 MinHash

The most well-known implementation of the Locality–Sensitive Hashing schema for Jaccard similarity is minwise hashing, or simply MinHash, proposed by Andrei Broder in 1997 [Br97], which includes a similarity-preserving hash function family and an algorithm for near-duplicates detection. Initially used in AltaVista search engine to detect duplicate web pages [Br00], today it is widely adopted in the search industry with numerous applications including large-scale machine learning systems, content matching for online advertising, and others.

The idea is to represent documents as short fixed-length signatures while preserving the similarity and efficiently compare them.

MinHash signatures

For every document d_i in the form of a document bit-array, the *MinHash value* is the position of the left-most set bit, in some permuted order of the index (some order of the features). Thus, by each permutation π, we can define a different MinHash value $\min(\pi(d_i))$.

Example 6.9: MinHash value

Consider the document bit-array built for the document d_3 (*flu*) in Example 6.6.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	1	1	1	1	1	0	1	0	1	0	0	1	0	0	0	0	1	0

For instance, let's take a random permutation of the index $0 \ldots 18$:

$$\pi = \{16, 13, 12, 4, 17, 10, 1, 2, 9, 14, 8, 5, 15, 3, 6, 18, 11, 7, 0\},$$

that corresponds to the following features order:

Index	Symptom
16	sneezing
13	rash
12	nausea
4	feeling tired
17	sore throat
10	loss of taste
1	cough
2	diarrhea
9	loss of appetite
14	red eyes

Index	Symptom
8	itchiness
5	fever
15	runny nose
3	difficulty sleeping
6	greyish-white spots
18	swollen glands
11	muscle aches
7	headache
0	aching body

Thus, the document bit-array, indexed in the permuted order is

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	0	**1**	1	1	0	1	1	1	0	0	1	0	1	0	0	0	1	1

After the bits are re-ordered according to the permutation π, the position of the left-most 1-bit for d_3 is 2, thus the MinHash value for the document d_3 equals 2:

$$\min(\pi(d_3)) = 2.$$

Instead of relying on a single MinHash value, variability can be reduced by building the *MinHash signature* of length k for each document d_i, which is a vector of k MinHash values computed using k random permutations $\pi_1, \pi_2, \ldots, \pi_k$ of the bit-array index. The length of the signatures k is independent of the size n of the universal set Ω and has to be chosen based on the allowed probability of error and given similarity threshold.

The list of signatures that have been built for each document $\{d_i\}_{i=1}^{n}$ create a *signature matrix* $\textsc{MinHashSig}$ $k \times n$, that is the primary data structure of the MinHash algorithm. The rows in the signature matrix correspond to the permutations and the columns correspond to the documents. It is important to highlight here, that to build the signature matrix we must use the same collection of permutations and apply them in the exact order.

The signature matrix $\textsc{MinHashSig}$ is a dense matrix with integer values, and the number of columns is equal to the number of documents in the dataset. However, the number of rows in the signature matrix is much less compared to the number of features in the universal set Ω, so this is more storage-efficient than the binary representations of the documents.

Unfortunately, in practice, it is unfeasible to permute a large index explicitly; even picking a random permutation of millions or billions of integers is time-consuming, and the additional necessary sorting of the index would take even more time.

For example, even for a very tiny webspam database of 350000 documents and 16 million features [Li12], the preprocessing cost for 500 independent random permutations was about 6000 seconds. However, nowadays it is not rare to find a universal set with upwards of 1 billion features. To pick a random permutation of 1 billion elements is not only slow, but just the representation of the index using 32-bit integers requires 8 GB of memory to store just *one* permutation.

Additionally, if the dataset does not fit into the main memory and we need to store it on disk, to access bits in a randomly permuted order will have the same disk issues as those discussed in the context of Bloom filters.

However, we can simulate the effect of a random permutation by a random hash function that maps indices $0 \dots m$ to exactly the same range. Some collisions can occur, but they are not important as long as k is big enough. For instance, we can use the family of universal hash functions $h_{\{a,b\}}(x)$, earlier defined by (1.2).

Example 6.10: Permutation simulation

Consider again the same bit-array that we built for document d_3 (*flu*) in Example 6.6.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	1	1	1	1	1	0	1	0	1	0	0	1	0	0	0	0	1	0

The universal set Ω for the documents has 19 features that are indexed by integers in the range $0 \dots 18$. To build signatures of length $k = 4$ for the document d_3, we select four random hash functions from the family (1.2) that map every index position $f \in 0 \dots 18$ to position $h_i(f) \in 0 \dots 18$, making the permutation of the index. In our case $m = 19$ and it is enough to choose $p = \mathrm{M}_5 = 2^5 - 1 = 31$.

$$\begin{aligned}
h_1(x) &:= ((22 \cdot x + 5) \bmod 31) \bmod 19, \\
h_2(x) &:= ((30 \cdot x + 2) \bmod 31) \bmod 19, \\
h_3(x) &:= ((21 \cdot x + 23) \bmod 31) \bmod 19, \\
h_4(x) &:= ((15 \cdot x + 6) \bmod 31) \bmod 19.
\end{aligned}$$

The corresponding permutations produced by the hash functions are

$$\begin{aligned}
h_1 &= \{5, 8, 18, 9, 0, 3, 13, 4, 7, 17, 8, 11, 2, 12, 3, 6, 16, 7, 10\},\\
h_2 &= \{2, 1, 0, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 18, 17, 16, 15\},\\
h_3 &= \{4, 13, 3, 5, 14, 4, 6, 15, 5, 7, 16, 6, 8, 17, 7, 9, 18, 8, 10\},\\
h_4 &= \{6, 2, 5, 1, 4, 0, 3, 18, 2, 17, 1, 16, 0, 15, 11, 14, 10, 13, 9\}.
\end{aligned}$$

Thus, instead of picking k random permutations, we simply compute $h_1, h_2, \ldots, h_k$ random hash functions on the rows and build the signature matrix MinHashSig out of them. Note that we need only one pass through the data to build the signature matrix in this way.

Algorithm 6.4: Building the MinHash signature matrix

```
Input: Binary document-vectors $\{d_j\}_{j=1}^{n}$
Input: Family of universal hash functions $\{h_i\}_{i=1}^{k}$
Input: Number $m$ of unique features in the universal set
Output: MinHash signature matrix
MinHashSig $\leftarrow \infty$
for $f \leftarrow 0$ to $m-1$ do
    for $i \leftarrow 1$ to $k$ do
        $h_i^f \leftarrow h_i(f)$
    for $d_j \in \mathbb{D}$ do
        if $d_j[f] \neq 1$ then
            continue
        for $i \leftarrow 1$ to $k$ do
            MinHashSig$[i-1, d_j] \leftarrow \min($MinHashSig$[i-1, d_j], h_i^f)$
return MinHashSig
```

Suppose we have a million documents and use signatures of length 200, then using 32-bit integers to represent the values we need 800 bytes per document, the entire dataset requires about 800 MB of memory storage.

Example 6.11: MinHash signature matrix

Let's build a MinHash signature matrix with signatures of length $k = 4$ using the permutations built in Example 6.10. Within the initial feature order, the bit-arrays for all documents are below.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
d_1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0	0
d_2	0	1	0	0	0	1	0	1	0	0	1	1	0	0	0	1	1	1	0
d_3	1	1	1	1	1	1	0	1	0	1	0	0	1	0	0	0	0	1	0
d_4	0	1	0	0	0	1	1	0	0	0	0	0	0	1	1	1	0	0	0
d_5	0	1	1	0	0	1	0	0	0	1	0	0	0	1	0	1	0	0	1

In the beginning, all values in the signature matrix are not set, effectively we can fill them with ∞:

	d_1	d_2	d_3	d_4	d_5
h_1	∞	∞	∞	∞	∞
h_2	∞	∞	∞	∞	∞
h_3	∞	∞	∞	∞	∞
h_4	∞	∞	∞	∞	∞

The values of hash functions for the index value 1 are $h_1^1 = h_1(1) = 5$, $h_2^1 = h_2(1) = 2$, $h_3^1 = h_3(1) = 4$, and $h_4^1 = h_4(1) = 6$. In the first position, only document d_3 has a 1-bit, thus we can update its signature values for each row, and the new values would be the minimum between the existing values of column d_3 in the signature matrix and the values of the corresponding hash functions. For instance,

$$\textsc{MinHashSig}[h_1, d_3] = \min(\textsc{MinHashSig}[h_1, d_3], h_1^1) = \min(\infty, 5) = 5.$$

Thus, the signature matrix MinHashSig after the first row has been processed is

	d_1	d_2	d_3	d_4	d_5
h_1	∞	∞	5	∞	∞
h_2	∞	∞	2	∞	∞
h_3	∞	∞	4	∞	∞
h_4	∞	∞	6	∞	∞

For the index value 2 the hash values are $h_1^2 = h_1(2) = 8$, $h_2^2 = h_2(2) = 1$, $h_3^2 = h_3(2) = 13$, and $h_4^2 = h_4(2) = 2$. In this case, all columns, except

d_1, can be updated since all those documents have the second bit set. The columns d_2, d_4, and d_5 simply get the corresponding hash values because there were no prior values for them (∞ in the signature matrix). However, for d_3 we need to compare the existing values with the current values of the hash functions to choose the smallest for each row, for instance, $\textsc{MinHashSig}[h_2, d_3] = \min(\textsc{MinHashSig}[h_2, d_3], h_2^2) = \min(2, 1) = 1$.

The matrix has the following form:

	d_1	d_2	d_3	d_4	d_5
h_1	∞	8	5	8	8
h_2	∞	1	1	1	1
h_3	∞	13	4	13	13
h_4	∞	2	2	2	2

Skipping ahead, this is the signature matrix after processing 14 index positions.

	d_1	d_2	d_3	d_4	d_5
h_1	7	3	0	3	3
h_2	6	1	0	0	0
h_3	5	4	3	4	3
h_4	2	0	0	0	0

Next, we continue on to process index value 15. At this position all documents, except d_3, have corresponding bits set. The values of the hash functions are $h_1^{15} = h_1(15) = 6$, $h_2^{15} = h_2(15) = 18$, $h_3^{15} = h_3(15) = 9$, and $h_4^{15} = h_4(15) = 14$. For instance,

$$\textsc{MinHashSig}[h_1, d_1] = \min(\textsc{MinHashSig}[h_1, d_1], h_1^{15}) = \min(7, 6) = 6,$$

which means we need to change the corresponding value in the signature matrix.

	d_1	d_2	d_3	d_4	d_5
h_1	6	3	0	3	3
h_2	6	1	0	0	0
h_3	5	4	3	4	3
h_4	2	0	0	0	0

If we process further index values, we can see that no actual updates are possible, meaning that the signature matrix above is the final one.

In fact, every permutation defines a MinHash function that is applied to the documents. It was proven that a family of such functions is an LSH family and the probability of collision over all permutations is equal to the Jaccard similarity:

$$\Pr\Big(\min(\pi(d_A)) = \min(\pi(d_B))\Big) = \mathrm{J}(d_A, d_B). \tag{6.4}$$

Thus, to estimate the Jaccard similarity between two documents, it is enough to compute the fraction of the MinHash signatures for which two corresponding columns have the same value (collide) in the signature matrix MinHashSig. While we are looking for hash collisions, it is possible that no identical values in either row are found, then we can assume that the documents are dissimilar.

Example 6.12: Similarity between signatures

Consider the signature matrix MinHashSig, that we built in Example 6.11.

	d_1	d_2	d_3	d_4	d_5
h_1	6	3	0	3	3
h_2	6	1	0	0	0
h_3	5	4	3	4	3
h_4	2	0	0	0	0

For instance, columns d_2 and d_3 share one value out of four signatures and the similarity between them is

$$\mathrm{Sim}_{\textsc{MinHashSig}}(d_2, d_3) = \frac{1}{4} = 0.25.$$

From Example 6.5 we know that the exact similarity is 0.2857, which is pretty close.

Columns d_4 and d_5 have three out of four values in common, therefore the similarity is

$$\mathrm{Sim}_{\textsc{MinHashSig}}(d_4, d_5) = \frac{3}{4} = 0.75.$$

This notably exceeds the exact Jaccard similarity value 0.44, but still indicates the high similarity between documents.

In contrast, columns d_1 and d_3 share no common values, so the similarity is 0 which is the exact value as well.

Remember, that the value we compute from the signature matrix is an approximation for the true value of the Jaccard similarity and depends on the signature length. The current length $k = 4$ is used for demonstration purposes only, and, in fact, is too small to build a close estimation with low variance according to the law of large numbers.

Properties

There is a clear trade-off between the similarity estimation error and storage. Indeed, the more MinHash functions h_i we use, the longer signatures we build, and correspondingly the lower expected error δ in the similarity estimation. However, it increases the storage requirements for the signature matrix MinHashSig and the number of required permutations which can significantly increase the computational efforts.

The practical guideline on choosing the signature length k based on the expected standard error δ is

$$k = \left\lfloor \frac{\sqrt{\theta \cdot (1-\theta)}}{\delta} + 1 \right\rfloor.$$

To store the MinHash signature of a single document using p-bit MinHash values, we need $p \cdot k$ bits per signature (for instance, $p = 32$ allows to enumerate up to $2^{32} - 1$ features) and the memory requirements for the whole MinHash signature matrix MinHashSig is $p \cdot k \cdot n$ bits.

When the number of documents n is high, storage becomes a problem for the algorithm. As a work-around to this problem, Ping Li and Arnd Christian König in 2010 [Li10] proposed a simple modification of minwise hashing, called *b-bit minwise hashing*. It provides a simple solution by storing only the lowest b bits of each p-bit MinHash value, naturally reducing the required memory for the signature matrix.

Intuitively, using fewer bits per MinHash value increases the similarity estimation variance, compared with the original minwise hashing for the same signature length k. Thus, it is necessary to increase k to maintain the same accuracy. The theoretical results [Li11] demonstrate that the signature length k should be adjusted only by a factor of about $\frac{\theta+1}{\theta}$. For most popular cases, when the resemblance is not too small (e.g., $\theta \geq 0.5$ as a common threshold), this is just two to three times bigger.

If the number of documents is large, which is the reason for using such improvements, the theoretical results suggest using $b = 1$ if the similarity threshold is $\theta \geq 0.4$, and $b \geq 2$ otherwise. Thus, even with the increased length of signatures, the total signature matrix size becomes smaller with b-bit minwise hashing.

Example 6.13: b-bit minwise hashing

As an example, for the similarity threshold $\theta = 0.5$, we can use $b = 1$, so the estimation variance will increase at most by the factor of three and, in order not to lose accuracy, it is necessary to adjust the signature length respectively. If each MinHash value has been stored initially using 32 bits, the improvement by using one-bit values is $\frac{32}{3} \approx 10.67$.

More specifically, replacing the classical MinHash algorithm that uses 32-bit MinHash values and signatures of length $k = 200$ by the 1-bit minwise hashing, we need longer signatures of length $k = 3 \cdot 200 = 600$, but greatly decrease the memory requirements from 800 bytes to 75 bytes per document.

Perhaps the most important advantage of b-bit minwise hashing are simplicity and minimal modifications to the original minwise hashing algorithm, therefore, it could be used to optimize already running systems.

Nearest neighbors search

While MinHash signatures let us represent documents in a compressed form using a space-efficient MinHashSig data structure that preserves the similarity information, there is still a quadratic number of pairs, which, as we already estimated, are unfeasible to process quickly for huge datasets of millions of documents. For instance, if a dataset consists of one million documents, the number of pairs is $5 \cdot 10^{11}$ and, doing as high as 10^7 comparisons per second, it requires about 14 hours to finish.

According to the generic LSH schema, to find the nearest neighbors for a given document, we need to take a number of independent locality-sensitive hash functions and apply them to the dataset in order to compute a key for each document, that is used to group them.

However, if documents are already represented as a MinHash signature matrix, it is enough to split all rows into b bands, select only one conventional hash function g (e.g., MurmurHash3), and apply it to the portion of each column within the band. Every band corresponds to a subset of features and we hash the documents only looking at that subset. Thus, two documents end up in the same bucket (have equal hash values) only if they are exact in that band or when a collision happens, that is rare for conventional hash functions and will be eliminated at the last step. In other words, two documents will appear in the same bucket if there is at least one band where their signature values are identical.

By choosing the number of bands b appropriately, we eliminate many document pairs with similarities below the threshold θ. Intuitively, the more similar the signatures are, the more likely they will agree on all rows in some band and become a candidate pair.

Example 6.14: MinHash LSH schema

Let's divide the signature matrix built in Example 6.11 into $b = 2$ bands with two rows each:

		d_1	d_2	d_3	d_4	d_5
band 1	h_1	6	3	0	**3**	**3**
	h_2	6	1	0	**0**	**0**
band 2	h_3	5	**4**	**3**	4	**3**
	h_4	2	**0**	**0**	0	**0**

The last two columns in band 1 are identical, so regardless of the particular hash function, the documents d_4 and d_5 become a candidate pair. Similarly, documents d_2 and d_4, d_3 and d_5 that have identical values in band 2 also become candidate pairs.

Considering the similarity threshold $\theta = 0.3$, we eliminate false positive candidate pairs by computing the exact similarities between documents in each pair and compare it with the threshold:

$$\begin{aligned} J(d_4, d_5) &= 0.44 > \theta, \\ J(d_2, d_4) &= 0.27 < \theta, \\ J(d_3, d_5) &= 0.307 > \theta. \end{aligned}$$

Only the pairs d_4 (*measles*) and d_5 (*roseola*), d_2 (*common cold*) and d_4 (*roseola*) can be returned as near-duplicates for the given threshold, which was our expectation as well.

Note that we choose the similarity threshold to eliminate duplicates quite randomly; however, there is a relation between the number of bands, the length of the signature, and the threshold.

To successfully apply a banding strategy, we need to have a recommendation for the number of bands, this is dependent on the similarity threshold θ we want to use to distinguish similar documents. Intuitively, if we have too many bands it is more likely that, for at least one small portion, many documents will become candidate pairs (an increased number of false positive errors), while for a few bands we need to compare long subsequences of signatures that are likely to differ in a few values even for documents that are very alike and we can miss many similar documents.

As soon as all candidate pairs are built, we execute the last step of the LSH schema and compute the exact similarity between the documents to eliminate false positive results.

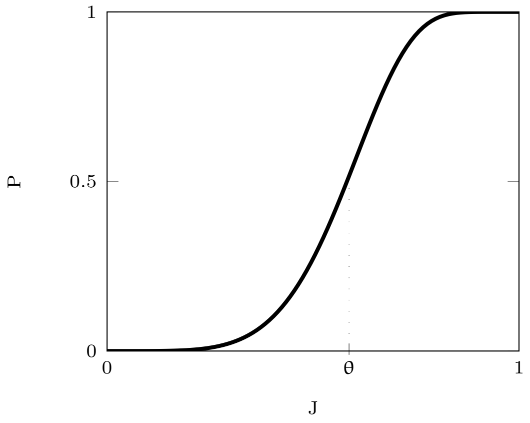
The LSH approach is very sensitive to the similarity distribution between documents in the dataset. If the dataset is skewed and most documents are similar to each other, we may find that all documents fall in one bucket, while other buckets remain empty.

Suppose, that a particular pair of documents have similarity s, then the probability P that the signatures agree in all rows of at least one band is

$$\mathrm{P} = 1 - (1 - s^{\frac{k}{b}})^b, \tag{6.5}$$

where b is the number of bands, k is the length of MinHash signatures; so the $\frac{k}{b}$ is the number of rows in each band.

The graph of the probability that documents with similarity s become candidate pairs according to (6.5) is an *S-curve*, meaning its values are very low until it reaches a *step*, then its values quickly increase and stay very high.



According to formula (6.3), we want to find the parameters when that step occurs close to the threshold θ, giving conditions for b and k:

$$\theta \approx \left(\frac{1}{b}\right)^{\frac{b}{k}}.$$

For example, the graph above is built for signatures of length $k = 50$ with $b = 10$ bands, five rows each. The approximate step value is 0.63 which is the similarity threshold at which documents are considered similar.

Generally, for the given signature length k and similarity threshold θ, the number of bands can be estimated as

$$b = \left\lfloor e^{\mathrm{W}(-k \cdot \ln \theta)} \right\rfloor, \tag{6.6}$$

where $W(\cdot)$ is the *Lambert W* function, which cannot be expressed in terms of elementary functions, but can be approximately computed as an iterative process[4]:

$$W_{\text{next}} = \frac{1}{W_{\text{prev}} + 1} \cdot (W_{\text{prev}}^2 - k \cdot \ln \theta \cdot e^{-W_{\text{prev}}}).$$

Example 6.15: Similarity threshold estimation

In the previous example, we used $b = 2$ bands with signatures of length $k = 4$. This setup corresponds to threshold $\theta = 0.707$, meaning that documents with similarity of at least 70% are likely to become a candidate pair after applying the bucketing from Example 6.14:

$$\theta \approx \left(\frac{1}{2}\right)^{\frac{2}{4}} = 0.707.$$

On the other hand, if we want to estimate the required number of buckets for $k = 4$ and the similarity threshold of 0.707 using (6.6), we need to compute the Lambert function $W(-4 \cdot \ln(0.707))$:

Iteration	W	b
1	1.3868	4
2	0.9510	2
3	0.6948	2
4	0.6933	2
5	0.6933	2

As you can see, the iterative process converges quite quickly and the recommended number of bands is 2.

However, since we are using very short signature lengths $k = 4$, the standard error $\delta = 0.11$ according to formula (6.2), and similarity for the true candidates can be approximated much lower than its true level, hence they can end up in different buckets. If we want to be more precise and maintain a standard error δ about 0.05 with the similarity threshold $\theta = 0.7$, we need to use signatures of length:

$$k = \left\lfloor \frac{\sqrt{0.7 \cdot 0.3)}}{0.05} + 1 \right\rfloor = 10.$$

[4]As an initial value we can use $W_{\text{prev}} = 0$, meaning only one band

The MinHash algorithm is very efficient for huge datasets and can be easily applied to the MapReduce computation model that makes it popular in Big Data applications. Its various implementations are available in Apache Spark, Apache Mahout, Apache Lucene and used in search engines and databases such as Elasticsearch, Apache Solr, CrateDB, and others. Google was reportedly using it for Google News personalization.

6.3 SimHash

Another popular hashing algorithm is *SimHash*, a *sign normal random projection algorithm*, that is based on the *simhash* function developed by Moses S. Charikar in 2002 [Ch02] and applied by Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma in 2007 [Ma07] to solve the problem of detection of near-duplicate web pages in Google.

From the mathematical point of view, the SimHash uses the concept of *sign random projections*. For a k-dimensional real-valued document-vector d it defines a similarity-preserving *SimHash function* family $\{h_v^{\text{sim}}\}$ that for the random vector v with components generated from i.i.d normal (i.e., $v_i \propto \mathrm{N}(0,1)$), produces a value as

$$h_v^{\text{sim}}(d) := \operatorname{sign}(v \cdot d) = \begin{cases} 1, & v \cdot d \geq 0, \\ 0, & v \cdot d < 0. \end{cases} \tag{6.7}$$

Thus, the *SimHash value* is the sign of the random projection and since the hyperplane with a normal vector v separates the multidimensional space into two half-spaces, it encodes just the information on the side (positive or negative) where the document is located.

Example 6.16: SimHash value

Consider the document-vector from Example 6.7 built for d_3 (*flu*):

16	2	9	16	16	2	0	9	0	9	0	0	16	0	0	0	0	9	0	

To compute its SimHash value, we need to build a vector v of 19 components which defines a hyperplane that separates the 19-dimensional space of documents. To build such a vector, we generate 19 random values from the normal distribution $N(0, 1)$ and use them as components of the vector (since scaling is not important in our case we scale the values by 10):

5	-1	6	15	-2	-2	16	8	-5	5	-5	-5	2	-19	-17	-6	-10	3	-9

The dot product of these two vectors is the sum of pairwise products of the corresponding components of both vectors:

$$v \cdot d = 5.12.$$

Thus, the sign of the result is positive, and the SimHash value is

$$h_v^{\text{sim}}(d) = \text{sign}(v \cdot d) = 1.$$

Notice, if two documents have an angle $\alpha = \pi$ in between, they will certainly appear in different half-spaces, and the reverse — documents with a perfect alignment that have $\alpha = 0$, definitely lie in the same half-space. Since the magnitude of the document-vectors doesn't play any role in formula (6.7), the probability that two documents d_A and d_B have the same SimHash value is equal to the probability of appearing on the same side of the hyperplane, that can be formulated using the angle between the documents $\alpha = \alpha(d_A, d_B)$ as

$$\Pr\left(h^{\text{sim}}(d_A) = h^{\text{sim}}(d_B)\right) = 1 - \frac{\alpha}{\pi} \approx \frac{\cos\alpha \; + 1}{2}, \qquad (6.8)$$

which defines the probability of hash collision for the SimHash function.

Such collision probability is closely related to the function $\cos(\alpha)$, therefore if documents are close to each other in terms of the cosine similarity (6.2), they will almost certainly collide, and vice versa. In this sense, a family of hash functions preserves the cosine similarity between documents and is the locality-sensitive function family for the cosine similarity.

SimHash signatures

The variability of using a single SimHash function, with just a single bit, is very high and to reduce it we can use p hash functions with different random vectors to produce a p-bit vector that is called a *SimHash signature*. Since every hash function preserves the similarity between documents, to estimate the similarity between signatures, we need to count the number of same-valued corresponding bits in them.

Thus, instead of working directly with long and real-valued document-vectors, the SimHash algorithm maintains a SimHashTable data structure that for every document stores its short fixed-length binary SimHash signature, which is conceptually very close to the signatures that we built in the MinHash algorithm.

Every document in the SimHashTable is represented as a p-bit binary string, which requires significantly less storage than high dimensional real-valued document-vectors, therefore it is a storage-efficient representation of the dataset.

Consider documents represented by real-valued vectors from the weights $(w_0, w_1, \ldots, w_{k-1})$ of the document features $(s_0, s_1, \ldots s_{k-1})$, or, practically speaking, we can think about documents as vectors of tuples $\{(s_j, w_j)\}_{j=0}^{k-1}$.

To build a p-bit SimHash signature for a document d, first, we hash each feature s_j using any conventional hash function h (e.g., MurmurHash3, SHA–1) into a p-bit hash value $h_j = h(s_j)$ that is going to be unique to the particular feature. After that, we start with an intermediate p-dimensional zero vector v and, iterating over hash values for all the features, we increase the i-th component v_i by the weight w_j if i-th bit of the hash value h_j is one, and decrease otherwise. At the end, when all features have been processed, we determine the signs of components of the vector v and set the corresponding bits of the final p-bit SimHash signature f to one for positive, and to zero for negative components.

Algorithm 6.5: Building the SimHash signature table

```
Input: Document-vectors $d = \{(s_j, w_j)\}_{j=0}^{k-1}$
Input: Conventional hash function $h$
Output: SimHash signature table
$v := \{v_i\}_{i=0}^{p-1}, v_i \leftarrow 0$
for $j \leftarrow 0$ to $k-1$ do
    $h_j \leftarrow \text{binary}(h(s_j))$
    for $i \leftarrow 0$ to $p-1$ do
        /* $h_j[i] \in \{0,1\}$, we either increment or decrement $v_i$ */
        $v_i \leftarrow v_i + (2 \cdot h_j[i] - 1) \cdot w_j$
return $\text{sign}(v)$
```

Example 6.17: SimHash signature table

Consider the dataset from Example 6.7. For simplicity, we build 6-bit SimHash signatures and to compute hashes from all the features of the universal set Ω, we use a randomly chosen 32-bit hash function MurmurHash3:

$$h(x) := \text{MurmurHash3}(x) \bmod 2^6.$$

Thus, the hashes of the features are

Feature	Symptom	h(s)	binary(h(s))
s_0	aching body	56	000111
s_1	cough	9	100100
s_2	diarrhea	14	011100
s_3	difficulty sleeping	41	100101
s_4	feeling tired	17	100010
s_5	fever	43	110101
s_6	greyish-white spots	7	111000
s_7	headache	5	101000
s_8	itchiness	26	010110
s_9	loss of appetite	37	101001
s_{10}	loss of taste	24	000110
s_{11}	muscle aches	13	101100

Feature	Symptom	h(s)	binary(h(s))
s_{12}	nausea	6	011000
s_{13}	rash	38	011001
s_{14}	red eyes	62	011111
s_{15}	runny nose	18	010010
s_{16}	sneezing	27	110110
s_{17}	sore throat	46	011101
s_{18}	swollen glands	4	001000

Similar to that example, we can build a real-valued representation of all documents in the dataset using feature weights:

d_1	0	0	0	0	0	0	0	0	54	0	0	0	0	0	0	7	31	0	0
d_2	0	3	0	0	0	3	0	11	0	0	20	20	0	0	0	3	11	11	0
d_3	16	2	9	16	16	2	0	9	0	9	0	0	16	0	0	0	0	9	0
d_4	0	4	0	0	0	4	27	0	0	0	0	0	0	15	27	4	0	0	0
d_5	0	3	13	0	0	3	0	0	0	13	0	0	0	13	0	3	0	0	23

Let's build a signature for document d_3 (*flu*). We start iterating over all features, using their binary representations, and for each feature, we build a value based on the document weights.

The intermediate vector v is a vector of 6 components that are all equal to zero at the beginning:

0	0	0	0	0	0

We start computing components of the vector v by iterating over all features. For instance, the binary representation of the feature s_0 has zeros in positions $0, 1$, and 2, thus we decrease the corresponding components of vector v by the feature weight $w_0^3 = 16$ that can be found in the first column for the document d_3 in the table above. For positions $3, 4$, and 5, where the feature's hash value has ones, we add the feature weights instead:

-16	-16	-16	16	16	16

In the same way, we process the feature s_1, that has ones in the positions 0 and 3, with the corresponding weight $w_1^3 = 2$:

-14	-18	-18	18	14	14

Continuing with other features, we get the final form of the vector v.

4	-32	0	4	-40	0

The actual values in the vector v are not critical and to build a signature for the document we need only the signs of the components. If a component of vector v is non-negative, the corresponding component of the signature is set to one; otherwise, it is set to zero. For document d_3 we have negative values in positions 1 and 4 only, thus the signature f^3 is

0	1	2	3	4	5
1	0	1	1	0	1

Following the same format, we process all remaining documents and the final SimHashTable is

	0	1	2	3	4	5
d_1	0	1	0	1	1	0
d_2	1	0	1	1	0	0
d_3	1	0	1	1	0	1
d_4	0	1	1	0	0	1
d_5	0	0	1	0	0	0

The probability that two signatures collide on some bit is equal to the collision probability given by formula (6.8). Therefore, two documents are considered *similar* if their signatures differ in at most p bit-positions or, in other words, the *Hamming distance* between their signatures is at most η, where η is a design parameter that is closely related to the similarity threshold θ.

The Hamming distance is widely used in information theory and can be seen as a measure of the minimum number of errors that could transform one signature into another. For binary strings, the Hamming distance is equal to the number of ones after applying the bitwise XOR operation.

Example 6.18: Hamming distance between signatures

Consider the signatures that we built in Example 6.17 and compare the documents d_4 (*measles*) and d_5 (*roseola*) using the Hamming distance:

	0	1	2	3	4	5
d_4	0	1	1	0	0	1
d_5	0	0	1	0	0	0

The corresponding bits in these two signatures differ in positions 1 and 5. Thus, the Hamming distance between them is equal to 2, meaning these documents are quite similar to each other, which is not surprising. For comparison, the exact cosine similarity between these documents is

$$c(d_2, d_4) = \cos(\alpha) = 0.17,$$

while for the current dataset the similarity threshold $\theta = 0.15$ can be considered reasonable.

Properties

While the SimHash function generates a single bit output, the MinHash function generates an integer, however, SimHash could be compared with the 1-bit minwise hashing schema that also uses a single bit output. However, it seems that the MinHash approach outperforms SimHash for high similarity thresholds [Li10].

In fact, SimHash is a dimensionality reduction technique that maps high-dimensional vectors to p-bit signatures, where p is small (usually, 32 or 64). As was shown experimentally by Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma, 64-bit signatures are enough to handle 8 billion ($\approx 2^{34}$) documents.

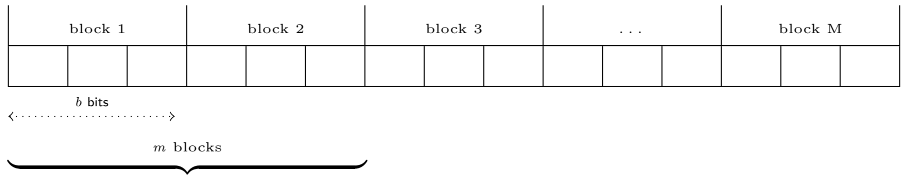
Nearest neighbors search

The SimHash algorithm lets us represent documents as a space-efficient SimHashTable of p-bit values that preserve the similarity information, but there is still a quadratic number of pairs that has to be evaluated to compute the Hamming distance and compare it with the threshold η, for huge datasets of millions of documents that is unfeasible to process in a timely manner.

Additionally, to identify the nearest neighbors for a document d with signature f^d, we need to find all signatures from the SIMHASHTABLE that differ from f^d in at most η bit-positions, which is known as the *Hamming distance range query problem*[5] and has remained difficult to solve on a large scale.

For similar documents, meaning choosing a small Hamming distance threshold η, we can use the *Block-Permuted Hamming Search* approach and split each p-bit SimHash signature into M blocks of about $b = \lceil \frac{p}{\mathrm{M}} \rceil$ consecutive bits each.

Figure 6.1: p-bit SimHash signature split into M blocks



Instead of comparing the whole signature, we can randomly choose m out of M blocks and perform search queries using the exact block-by-block comparison to the top bits of the given signature, where parameter m is a design parameter related to the Hamming distance threshold η.

Every group of selected m blocks defines a new shorter signature value, that is about $m \cdot b$ bits and, since the order of the signatures is not important, exactly $\mathrm{N} = \binom{\mathrm{M}}{m}$ transformed m-block signatures can be built for each original p-bit SimHash signature.

[5]M. Minsky and S. Papert. Perceptrons. MIT Press, 1969

Example 6.19: m-blocks SimHash signatures

Consider a 64-bit SimHash signature and define the similarity threshold in terms of the Hamming distance at $\eta = 2$.

We can split the signature into $\mathrm{M} = 5$ blocks, each block receives about $b = \lceil \frac{64}{5} \rceil = 13$ consecutive bits, for instance, 13, 13, 13, 13, and 12 bits per block. If we proceed with a block-by-block comparison with $m = 3$ blocks, the total number of ways to choose them is $\mathrm{N} = \binom{5}{3} = 10$, and the resulting 3-block signatures contain either 39 bits (or 38, for the last block of 12 bits).

Thus, for every p-bit SimHash signature in the SimHashTable we can produce $\mathrm{N} = \binom{\mathrm{M}}{m}$ m-block signatures and store them in sorted buckets $\{\mathrm{B}_i\}_{i=1}^{\mathrm{N}}$. Each bucket B_i will be associated with the particular selection of m blocks π_i and the exact number of bits in the stored signatures b_i.

Algorithm 6.6: Bucketing m-block signatures

```
Input: SimHash signature table
Input: Number N of m-block signatures
Output: Buckets with m-block signatures
for $i \leftarrow 1$ to N do
    for $f_j \in$ SimHashTable do
        $\hat{f}_j \leftarrow \pi_i(f_j)$
        $\mathrm{B}_i \leftarrow \mathrm{B}_i \cup \{\hat{f}_j\}$
    $\mathrm{B}_i \leftarrow$ sort($\mathrm{B}_i$)
return $\{\mathrm{B}_1, \mathrm{B}_2, \ldots, \mathrm{B}_\mathrm{N}\}$
```

When we need to find near-neighbors whose p-bit SimHash signatures differ in at most η bit-positions from the signature f^d of the given document d, we probe each of N buckets, that can be done in parallel. For every bucket B_i we find all m-block signatures whose b_i bits match the b_i bits of $\pi_i(f^d)$. If the total amount of signatures is 2^q, then on average 2^{q-b} such matches are expected in every bucket.

After that step, to eliminate possible false positive candidates, we compute the exact Hamming distance for each signature and check that it doesn't exceed the η.

Instead of building new shorter signatures and keeping the SimHashTable for the exact Hamming distance computation in the last step, we can permute the original SimHash signatures in a way that selects m blocks as the upper-most bits in the signature and keeps them in an untouched order. Hence, if signatures are permuted in the same way, the Hamming distance will not change, and we are still able to eliminate false positive candidates by exact distance calculation.

Algorithm 6.7: Searching for nearest neighbors

```
Input: Document $d = (s_d, w_d)$
Input: Hamming distance threshold $\eta$
Output: List of nearest neighbors
$f_d \leftarrow$ **Signature**$(d, h)$
$neighbors \leftarrow \varnothing$
for $i \leftarrow 1$ to N do
    $candidates \leftarrow \varnothing$
    for $\hat{f}_j \in \mathrm{B}_i$ do
        if $f_d[: b_i] = \hat{f}_j$ then
            $candidates \leftarrow candidates \cup \{j\}$
    for $\hat{j} \in candidates$ do
        if **HammingDistance**$(f_j, f_d) \leq \eta$ then
            $neighbors \leftarrow neighbors \cup \{d_j\}$
return $neighbors$
```

Using the binary search to find matches in each bucket, an individual probe could be done in $\mathrm{O}(b_i)$ steps, but the number of bits in each block b_i should be reasonably large to avoid checking too many signatures.

For every p-bit SimHash signature, the total number of buckets for the given Hamming distance threshold η has to be selected as $\mathrm{M} > \eta + 1$, then for a block-by-block comparison we can use $m \in [1, \mathrm{M} - \eta]$ blocks.

However, there is a clear trade-off between the number of blocks m and the number of buckets N for the fixed choice of the SimHash signature length p and the Hamming distance threshold η. If we use more blocks, therefore longer signatures, it reduces the query time because there is fewer possible matches but increases the required storage. On the other hand, with shorter signatures we can reduce the storage, but it requires the checking of more matches, which increases the query time.

To optimize storage usage, it is possible [Ma07] to compress fingerprints which can decrease the data structure size by approximately half. The compression is based on the fact that fingerprints for similar documents share some amount of bits, so we can build shorter blocks where fingerprints are encoded by storing Huffman codes for the most-significant 1-bit positions of their XOR differences.

SimHash appeared to be popular for approximate nearest neighbors searches, but it could be due to the popularity of the cosine similarity, for which SimHash can be directly applied. Same as with MinHash, the SimHash algorithm suites for the MapReduce model and is widely available, but it is mostly in independent libraries. Google was reportedly using it for near-duplicate detection in web crawling.

Conclusion

In this chapter we considered different approaches to defining similarity between documents of any nature. We have learned a very powerful framework that addresses the near-duplicate detection problem which is extremely important for many real-world applications. As to particular implementation we considered two very efficient algorithms that are widely used in the industry.

If you are interested in more information about the material covered here or want to read the original papers, please take a look at the list of references that follows this chapter.

This chapter ends our narration about probabilistic data structures and algorithms. While it is impossible to cover all the existing amazing solutions, here we wanted to highlight their common ideas and important areas of application, including efficient membership querying, counting, stream mining, and similarity estimation.

Hopefully you found this book useful and learned out of it.

Thank you very much.

Bibliography

[Br97] Broder, A.Z. (1997) “On the Resemblance and Containment of Documents”, *Proceedings of the Compression and Complexity of Sequences*, June 11–13, 1997, p. 21, IEEE Computer Society Washington, DC.

[Br00] Broder, A.Z., et al. (2000) “Min-wise independent permutations”, *Journal of Computer and System Sciences*, Vol. 60 (3), pp. 630–659.

[Ch02] Charikar, M.S. (2002) “Similarity estimation techniques from rounding algorithms”, *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, Montreal, Quebec, Canada - May 19–21, 2002, pp. 380–388.

[In98] Indyk, P., Motwani, R. (1998) “Approximate nearest neighbors: towards removing the curse of dimensionality”, *Proceedings of the 13th annual ACM symposium on Theory of computing*, Dallas, Texas, USA - May 24–26, 1998, pp. 604–613, ACM New York, NY.

[Li10] Li, P., König, A.C. (2010) “b-Bit Minwise Hashing”, *Proceedings of the 19th International Conference on World Wide Web*, April

26-30, 2010, Raleigh, North Carolina, USA - April 26–30, 2010, pp. 671–680, ACM New York, NY.

[Li11] Li, P., König, A.C. (2011) “Theory and applications of *b*-bit minwise hashing”, *Magazine Communications of the ACM*, Vol. 54 (8), pp. 101–109.

[Li12] Li, P., et al. (2012) “One Permutation Hashing”, *Proceedings of the 25th International Conference on Neural Information Processing Systems*, Lake Tahoe, Nevada - December 03–06, 2012, pp. 3113–3121, Curran Associates Inc., USA.

[Li14] Liu, Y. et al. (2014) “SK-LSH: An Efficient Index Structure for Approximate Nearest Neighbor Search”, *Proceedings of the VLDB Endowment*, Vol. 7 (9), pp. 745–756.

[Ma07] Manku, G. S., et al. (2007) “Detecting near-duplicates for web crawling”, *Proceedings of the 16th international conference on World Wide Web*, Banff, Alberta, Canada - May 08–12, 2007, pp. 141–150, ACM New York, NY.

[Sh14] Shrivastava, A., Li, P. (2014) “In Defense of MinHash Over SimHash”, *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, 22–25 April 2014, Reykjavik, Iceland - PMLR, Vol. 33, pp. 886–894.

[So11] Sood, S., Loguinov, D. (2011) “Probabilistic Near-Duplicate Detection Using Simhash”, *Proceedings of the 20th ACM international conference on Information and knowledge management*, Glasgow, Scotland, UK - October 24–28, 2011, pp. 1117–1126, ACM New York, NY.

[Wa14] Wang, J., et al. (2014) “Hashing for similarity search: A survey”, *arXiv preprint.* arXiv:1408.2927v1 [cs.DS] - Aug 13, 2014.

Index

Probabilistic data structures is a common name for data structures based mostly on different hashing techniques. Unlike regular (or deterministic) data structures, they always provide approximated answers but with reliable ways to estimate possible errors. Fortunately, the potential losses and errors are fully compensated for by extremely low memory requirements, constant query time, and scaling, the factors that become essential in Big Data applications.

While it is impossible to cover all the existing amazing solutions, this book is to highlight their common ideas and important areas of application, including membership querying, counting, stream mining, and similarity estimation.

✓ **Learn** how to solve practical issues of massive data handling

✓ **Master** the theoretical aspects of probabilistic data structures

✓ **Identify** the right data structures for your particular problems

The purpose of this book is to introduce technology practitioners which includes software architects and developers, as well as technology decision makers to probabilistic data structures and algorithms. Reading this book, you will get a theoretical and practical understanding of probabilistic data structures and learn about their common uses.

ABOUT THE AUTHOR

Andrii Gakhov is a mathematician and software engineer holding a Ph.D. in mathematical modeling and numerical methods. He has been a teacher in the School of Computer Science at V. Karazin Kharkiv National University in Ukraine for a number of years and currently works as a software practitioner for ferret go GmbH, the leading community moderation, automation, and analytics company in Germany. His fields of interests include machine learning, stream mining, and data analysis.

pdsa.gakhov.com
github.com/gakhov/pdsa
2019 © Andrii Gakhov
gaKhov

ISBN 978-37-48190-48-6
9 783748 190486
34,95€ [D]