What are Bloom filters?

A tale of code, dinner, and a favour with unexpected consequences.

Jamie Talbot Follow
Jul 15, 2015 · 16 min read

Genesis

"So how do you know whether someone's read a post already?" asks Sarah, Medium's legal counsel. We're at dinner for my birthday, yet somehow have gotten onto the topic of work yet again. Sarah is referring to Medium's personalised reading list—a recommendation system that I helped build—which suggested new and interesting posts to users when they visited Medium's homepage.

I sigh. I'm hungry and the main course has just arrived—venison glazed in honey, served with a sweet potato hash. The problem is, I find describing technical solutions like this really interesting, and Sarah knows it. If she wasn't vegetarian, I'd suspect it was a ploy to distract me so she could get her hands on my dinner. My wife shakes her head with a rueful smile and turns to chat to another friend.

I put my fork down, and begin to explain to her all about the Bloom filter, an ingenious data structure invented 50 years ago, and how we use it at Medium to answer Sarah's question. Most of the rest of the dinner table ignores us, and no-one bats an eyelid when I begin drawing on napkins to illustrate the concept. All of this has happened before and all of it will happen again.

I pause for a second, and look up. "And what does monkey hash to?" I ask Brad, a Median sitting across from me at the dinner table.

"Monkeys hash to 10, 2, and 1," he says promptly, and I nod approvingly. Sarah is non-plussed. Two years ago—before Sarah joined—I gave a short presentation on Bloom filters to Medium's then 25 staff. I'm impressed Brad remembers this made-up fact, about a made-up hash function, which was made-up to illustrate the possibility of false positives. But more on that later.

. . .

Hashing

To understand Bloom filters, you first have to understand hashing. To understand hashing, you *don't* have to understand maths, which is good, because I'm a middling mathematician at best. I sit at the median, maybe.

A hash is like a fingerprint for data. A hash function takes your data —which can be any length—as an input, and gives you back an identifier of a smaller (usually), fixed (usually) length, which you can use to index or compare or identify the data.

This is a hash of my full name:

```
b3f9b3a3504ccb29c4183730a42c8d56
```

Most hash functions are *one-way* operations, which is to say you can figure out an identifier from the data, but you typically can't do the reverse. You couldn't look at that hash above, for example, and know that my full name is *James Graham Michael Swanson Talbot*. Reversible hash functions do exist, but they tend to be pretty trivial, and not particularly useful.

There are a few properties which are desirable in a useful hash function. The most important one is that the same input must always hash to the same output. Really, that's the defining feature. If the same input gave you a different output each time, it wouldn't be very useful as an identifier.

Ideally, the output values should be distributed uniformly—that is, each possible output should be equally likely. For a lot of use-cases it is also important that the outputs be distributed randomly; similar inputs should not give similar outputs. For some use-cases—usually in security—it is also important to minimise collisions, meaning, as far as possible, each input should give a unique output.

Finally, in most cases (though not always), you want it to be fast. No one wants to wait for anything these days, and that includes programmers.

The combination of hash function properties that you care about will vary wildly depending on the task for which you are using them. And hash functions are used in a *lot* of tasks. Shazam uses hashes to figure out what song you're listening to. If you need to find duplicates in a set of things, you might use a hash. Databases? Giant hashes. This post on Medium was delivered to you securely—you can probably see a green padlock on the browser address bar that confirms this. Hashes are important in that process too.

You may even have heard of some of these algorithms. SHA-1—Secure Hash Algorithm—was devised by the NSA, and for a long time was used to secure lots of Internet communication. MD5 is another popular one, often used to prove that the file you're downloading is the one you think it is, and not some malware written by the NSA to infect your computer and spy on you. (In fact, MD5 was the hash function I used to encode my name further up the page.)

Some hash functions have exotic and nonsensical names—this is computing after all—and so you also get things like CityHash, MurmurHash, and SpookyHash.

There are literally thousands of named hashing functions. Some are secure, but comparatively slow to calculate. Some are very fast, but have more collisions. Some are close to perfectly uniformly distributed, but very hard to implement. You get the idea. If there's one rule in programming it's this: there will always be trade-offs.

While you don't have to be a genius to understand hashes, you do have to be a pretty exceptional mathematician or computer scientist to *create* one—or at least one that's useful.

Take this elegant, impenetrable piece of code by Paul Hsieh, for example, which he calls SuperFastHash (excluding some helper code):

```
uint32_t SuperFastHash (const char * data,
int len) {
  uint32_t hash = len, tmp;
  int rem;

  if (len <= 0 || data == NULL) return 0;
  rem = len & 3;
  len >>= 2;

  /* Main loop */
  for (;len > 0; len--) {
    hash += get16bits (data);
    tmp = (get16bits (data + 2) << 11) ^
hash;
    hash = (hash << 16) ^ tmp;
    data += 2 * sizeof (uint16_t);
    hash += hash >> 11;
  }

  /* Handle end cases */
  switch (rem) {
    case 3: hash += get16bits (data);
      hash ^= hash << 16;
      hash ^= ((signed char)data[sizeof
(uint16_t)]) << 18;
      hash += hash >> 11;
      break;
    case 2: hash += get16bits (data);
      hash ^= hash << 11;
      hash += hash >> 17;
      break;
    case 1: hash += (signed char)*data;
      hash ^= hash << 10;
      hash += hash >> 1;
  }
  /* Force "avalanching" of final 127 bits
*/
  hash ^= hash << 3;
  hash += hash >> 5;
  hash ^= hash << 4;
  hash += hash >> 17;
  hash ^= hash << 25;
  hash += hash >> 6;

  return hash;
}
```

This is a series of fast operations that do things like addition and multiplication, inverting of ones and zeroes, munging and melding parts of the data with itself, shifting bits this way and that. And at the end of it, you get an almost perfectly unique fingerprint of the input data. Why pick those operations, with those numbers, in that order? Beats me. Paul Hsieh might not be a wizard, but I've seen no evidence of that.

What matters is, we have a way of taking any data, like the contents of a YouTube video, or an MP3 file, or the word "monkey", and getting back a fingerprint that is predictable in length and unique—mostly—to that item.

. . .

A quick favour

This is all Marcin's fault, really. Marcin, if you didn't know already, is obsessed with typography. This admirable trait makes him an easy mark if you need any tasks doing that involve Medium's fonts, or icons, which are also a font. Sometime last year, I was working on a feature which required a new icon. I was keen to get it deployed to the world before the weekend deadline, so I sent him a quick message at 6pm:

"Any chance you can get a new icon into our font sometime tomorrow? I want to ship *feature X.*"

“Hmm. And what will you do for me?” he asks, seemingly nonchalantly. (Now, recounting this exchange to you, I instead picture him rubbing his hands together in gleeful expectation like Mr. Burns.) “Will you fix a bug of my choosing? I kind of have a full day tomorrow.”

“Sure,” I say, glad that we’re going to be able to ship the feature. “Anything you like,” I promise, naïvely.

6:30pm. Slack pings with a notification. “PR is in,” he writes, announcing to me that he’s already completed the work. “I want you to fix *this*.”

. . .

This

The word *this* is interesting for JavaScript programmers, a source of confusion for newcomers to the language, and frustrating even to journeypeople who periodically forget about the language’s idiosyncratic functional scope. The gist of it is, whenever you see *this* in JavaScript code, you never quite know what it’s referring to. For programmers who are used to everything being in its proper place, this can be disconcerting. JavaScript is full of these objectionable, endearing nuances, and is possessed of a flexibility bordering on pathological. Picture a politician conversing with the lobbyist whose large donation made their election possible. JavaScript is more flexible than that.

Medium’s server-side codebase is primarily written in JavaScript, and the personalised reading list recommendation system abused JavaScript’s aforementioned flexibility with gusto. We discovered later that its correct performance relied, unintentionally, on a particular quirk in how asynchronous instructions were chosen for processing. It was a wonder that it worked at all. Looking back at that code… well let’s just say, nobody’s a worse programmer than your past self.

Like hashes, recommendation systems are all around us. Netflix is a recommendation system. StumbleUpon and Pandora are too. Amazon's "Because you bought *Cryptonomicon* on Kindle" feature. Facebook's "People you may know", their stream, their search results. All recommendation systems. And the tragedy of recommendation systems is that people only notice them when they're broken. Very few people say "I love Netflix's suggestions." Many more say "Ugh, Pandora played me the exact same thing yesterday." Recommendation systems are the sports referees of the computing world.

To give good suggestions, you need to collect data about your users and find interesting ways to use it. For Medium, a simple method for suggesting a story might be "because you follow the author." A more complex one might be *collaborative filtering*, which you can describe as "people who recommended lots of the stories that you recommended also recommended *this* story".

A typical recommendation system will take a bunch of these methods that make good guesses about what you like, combine their results together so you get a varied selection, then sort them so the ones with the highest score come first. It should do this very quickly. And woe betide the system that shows someone the same thing too many times, for that way lies frustration, anger, and lots of ALL CAPS emails to user support.

Medium users have written hundreds of thousands of posts. Each one of these has an ID and is stored in a database table, but the table is so big, and accessed so frequently, that it can't fit on one machine. When you ask the database for a specific post, how does it know where to find it? A hash function. The database takes the ID, hashes it to a unique value, and uses that to jump straight to the record location, just like you'd use the index of a book to jump straight to a specific page number, back when the books Amazon suggested to you were on paper and had indices.

When suggesting stories to a user, the system may come up with thousands of possibilities, but not every one will be appropriate for a user. Perhaps a story has been suggested to them too many times already. Perhaps they've already read it. We need to filter those stories out, but we don't want to retrieve thousands of records from the database to do it. We need a mechanism that is particularly good at answering the question "has this been seen before?".

And for that, we use a Bloom filter.

. . .

Returning the favour

Unfortunately for me, the *this* that Marcin is talking about is a bug that I can't help him with. He wants to me to change the ordering of how people appear in a particular list, so that your friends appear first. It's not a complex change, but the way we store our data means the list will become slow to load, and infinite scrolling will break. We are sadly stuck with what we have until we restructure our data storage, which, given our other priorities, isn't likely to happen soon. Trade-offs happen in product development too. I report back sheepishly to him that he will have to choose something else.

The Christmas break rolls around, and I forget all about my promise. And then on February 18th, this appears on Slack:

```
[mwichary]

[9:08 AM] BTW I was supposed to send you
possible issues to work on:
https://github.com/Medium/medium/issues/xxx
x
```

An elephant never forgets, nor either, apparently, do Polish designers. Incredibly, the bug he sends me this time is *literally* impossible to fix.

Partly out of guilt, but mostly because he's a fantastic conversationalist, I invite him to my birthday dinner in June. And then, a week or so ago, *this*:

```
[mwichary]

[1:24 PM] this is a periodic reminder that
you owe me this
https://github.com/Medium/medium/issues/xxx
xx or a Paul Ford-esque article.
```

So *that* escalated quickly.

Paul Ford, in case you didn't know, recently wrote an incredible treatise on programming and all that surrounds it. Funny, accessible, informative, and erudite—hell, it even had Easter Eggs. To misappropriate Greenspun's 10th rule, I'm quite confident that any sufficiently complete explanation of code and culture will now contain a poorly-articulated, mistake-ridden, plagiarised retelling of half of What Is Code. (Ladies and gentlemen, I present Exhibit A.)

No longer seeking an impossible programming task, Marcin is now content with an impossible writing task.

. . .

Time and space

The world hardly needs another definition of the word *algorithm*, but here I go anyway: *an algorithm is a set of steps to solve a problem*. Simples. Algorithms don't really have anything to do with computers, although computers turn out to be very handy algorithm processing machines. The hash functions described earlier are all implementations of algorithms, but not all algorithms are as complicated as those. In fact, many of the most important ones are actually quite straightforward.

Take Dijkstra's Algorithm—why yes, Paul Ford did indeed touch on this one—which is used for finding the shortest path between points in a graph. It's simple enough to describe in a single paragraph of plain English:

Start at your starting point, and figure out how long it takes to get to each neighbour. Then go to each neighbour and figure out how long it cumulatively takes to get to each of its neighbours. (Don't go back to points you've already visited.) If that cumulative total is shorter than the current shortest path to that point, replace the number for that point with the lower one. Repeat. Finish when you get to your ending point.

Remarkably, that simple algorithm, and modifications of it, are fundamental to allowing computers to talk to each other efficiently. Your Kindle books would take much longer to wend their way to you over Whispernet without the pioneering work of Edsger Dijkstra.

And then there's sorting, the old staple of CS101 classes around the world. There are dozens and dozens of well-understood and heavily analysed sorting algorithms. We have Quicksort, Merge sort, Bubble sort, Heapsort, and Insertion sort. And because this is computing, there's also Timsort, Stoogesort and Smoothsort—Dijkstra was responsible for that last one too.

For the personalised reading list, when we've got all our suggestions ready, and we want to sort them by score, how do we choose which sort to use? If all the sorting algorithms give you the same result, how do we know which ones are better?

There are many measures of algorithmic performance, but two of the most important ones are *time complexity* and *space complexity*.

Say you wanted to find the smallest number in a simple list of numbers. To do that, you have to look at each number once, because the last number might be the smallest. As the size of the list grows, you need to do more checking. We'd say that this algorithm has *linear* time complexity. As we're working our way through the list, we only need to remember one number—the current lowest. Because we only need to remember that one number, no matter how long the list gets, we'd say that the algorithm has *constant* space complexity.

Remember that using a hashed value lets a database jump straight to the record, like an index in a book? Computationally speaking, we say that this is a *constant* time operation, because it takes the same amount of time, on average, to find the record, no matter how big the record is, or how many records are in the table. Constant time operations are groovy because they're fast.

Most problems in computing have multiple solutions, and it's often the case that you'll have the choice between a slower algorithm that requires less space, or a faster algorithm that uses more. Which one you choose depends on the specific needs of your system. Trade-offs.

This particular trade-off is sometimes called the space-time trade-off, which always makes me think of black holes, Einstein, and Douglas Adams' Bistromathics.

. . .

Check, please

The bistro in which we are sitting is starting to empty out. The check arrives. We resist the urge to power an infinite improbability drive, split it evenly, and do the waiter a favour by paying with a single card. Credit card processing—hashes are used there too.

I managed a few bites of honey-glazed venison throughout the evening, the rest gratefully claimed one forkful at a time by my wife. The thought crosses my mind that she put Sarah up to this. Paranoia? Probably. I successfully fought off the attacks on my crème anglaise though. Venison is venison, but dessert is dessert.

Sarah has been very gracious in not allowing her eyes to glaze over, but everyone's starting to look sleepy. Fortunately, all the background is out of the way. I drain my wineglass, and finally answer the question.

. . .

In Bloom

Burton Howard Bloom, the eponymous hero of our tale, and, it has to be said, possessed of a name with a very pleasing cadence—*Bur-*

ton-Ho-ward-Bloom—studied computer science at MIT from 1963 to 1965. He invented the data structure that came to be known as the Bloom filter in 1970 while working at Computer Usage Company in Newton Upper Falls, MA.

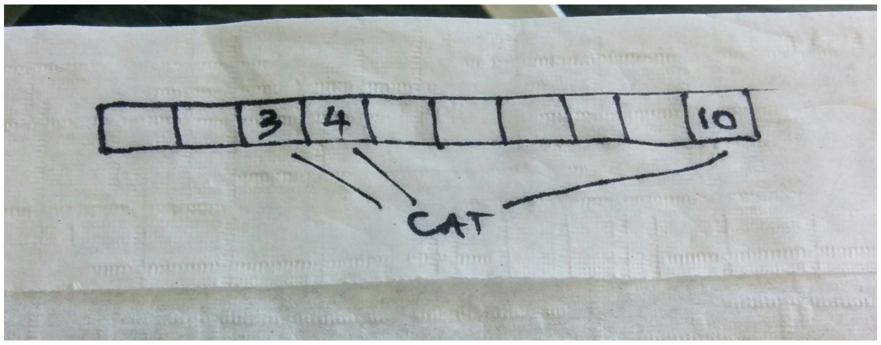
Bloom filters can very quickly answer variations on the Yes/No question "is this item in the set?", like "have I seen this item before?". There are two important caveats though. Very rarely, it will say *Yes* when the answer is actually *No* (although it will never say *No*, when the answer is actually *Yes*). You also can't remove an item from a Bloom filter. Like elephants and unrelenting Polish designers, Bloom filters never forget.

If you can accept a couple of false positives from time to time though, and if you don't need to remove items, it's a great choice. A Bloom filter has constant time complexity for both adding items *and* asking whether they are present, which makes it doubly groovy, and it requires very little space relative to the size of the items you need to store and check. It gets these properties in large part because it is based on hash functions.

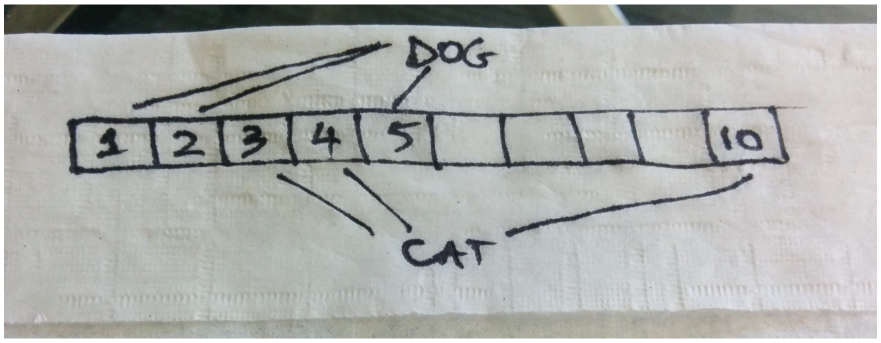
For the hash functions that Bloom filters use, collisions in the outputs don't really matter too much, as long as they're reasonably rare. It's more important for the outputs to be evenly and randomly distributed. And, of course, we want our hash functions to be fast.

You can think of a Bloom filter as a large set of numbered buckets, each one starting off empty. Let's imagine we want to keep track of the pets that Ev Williams owns. To do this, we're going to use a Bloom filter with 10 buckets and three hash functions.

We start by putting *Cat* into our Bloom filter. Internally, the filter takes *Cat* and passes it to the three hash functions, which return three identifiers. In this case, *Cat* hashes to identifiers 3, 4, and 10. The filter goes and fills up every bucket whose number matches one of the identifiers—3, 4, and 10. It's able to do this very quickly because it can jump straight to each bucket to fill it up, just like you jump straight to page numbers from an index.

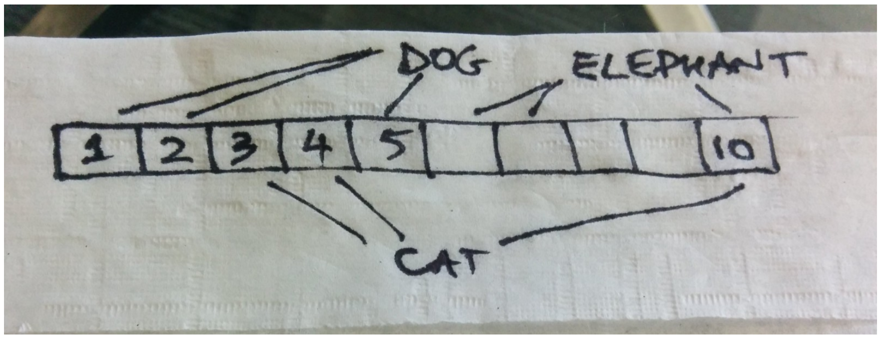


Next we put *Dog* into our filter. Using the same hash functions, *Dog* hashes to 5, 2, and 1, so those buckets get filled up too. After we've done both of these operations, buckets 1, 2, 3, 4, 5, and 10 are full.

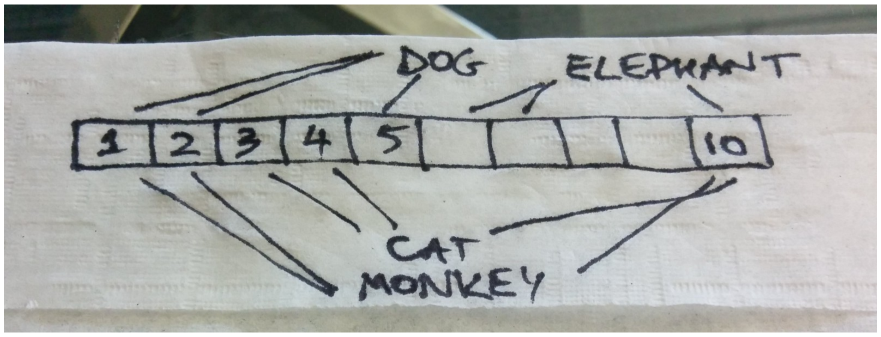


Does Ev have a cat? We can ask the filter. Because the same input always hashes to the same output, *Cat* still hashes to 3, 4, and 10, so we need to check all those buckets. Because they're all full, we're pretty sure that Ev has a cat.

So let's find out if Ev has an elephant. *Elephant* hashes to 10, 6, and 7. When we check those buckets, 10 is full, but 6 and 7 are empty. We know that if *Elephant* had been put into the filter, buckets 10, 6, and 7 would be full. So we can categorically say that Ev doesn't have an elephant. Yet.



But how do we get false positives? Imagine now that we want to see if Ev has a monkey. As Brad recalled, *Monkey* hashes to 10, 2, and 1. We check buckets 10, 2, and 1, and lo and behold, they're all full. The Bloom filter tells us that Ev has a *Monkey*, even though he doesn't. Yet.



As a corollary, a Bloom filter can't answer the question "what pets does Ev Williams have?", because it's one-way. There's no way to tell what combination of fauna, exotic or otherwise, led to the current set of filled buckets.

We can control the probability of getting a false positive by controlling the size of the Bloom filter. More space means fewer false positives. Say it with me, people: trade-offs.

. . .

Digestif

We leave the restaurant and head down the road for a nightcap.

"Ok, let me see if I have this right," says Sarah. "You want to be able to quickly exclude posts that a user has read, so that you don't suggest those posts again. And a Bloom filter is a good fit for this problem, because it will never let through a post that the user has read, and even though it might exclude a post that they *haven't* read, that's ok because they'll never know what they don't see? And it's very fast?"

"Precisely," I reply.

"That would make a good story," says Marcin.

. . .

Thanks are due to Sarah Agudo, whose interest in technology frequently sparks discussions like this, and who has an incredibly high tolerance for long-winded explanations. Thanks to Nick Santos for offering feedback on the technical portions. Thanks also, I guess, to Marcin Wichary for manipulating me into writing this, and for the small details that make reading the written word on Medium a joy. Though I feel like he owes me one now.

As in any piece of writing about code, there are certain simplifications made in deference to the story, and to clearer communication. All errors are my own. If you spot anything particularly egregious, leave a note in the margins!