It PROBABLY Works

**PROBABILISTIC
ALGORITHMS ARE
ALL AROUND US—
NOT ONLY ARE THEY
ACCEPTABLE, BUT
SOME PROGRAMMERS
ACTUALLY SEEK OUT
CHANCES TO USE THEM.**

TYLER MCMULLEN, FASTLY

Probabilistic algorithms exist to solve problems that are either impossible or unrealistic (too expensive, too time-consuming, etc.) to solve precisely. In an ideal world, you would never actually need to use probabilistic algorithms. To programmers who are not familiar with them, the idea can be positively nerve-racking: “How do I know that it will actually work? What if it’s inexplicably wrong? How can I debug it? Maybe we should just punt on this problem, or buy a whole lot more servers...”

However, to those who either deeply understand probability theory or at least have used and observed the behavior of probabilistic algorithms in large-scale production environments, these algorithms are not only acceptable, but it’s also worth seeking out opportunities to use them. This is because they can help solve problems and create systems that are less expensive, more predictable, and can do things that couldn’t be done otherwise.

The world is probabilistic—or at least it’s way too complex for us to model 100 percent accurately. Networks, and especially the Internet, are also probabilistic. Whether or

not a particular packet will get where it needs to go is not something we can have complete knowledge of. Knowing which paths it will take through the many networks it will traverse in getting from one side of the world to the other is even less certain. Systems that run across asynchronous/lossy networks are necessarily probabilistic. Like it or not, probability is all around us.

Uncertainty is the core of the problem that uninitiated developers have with probabilistic algorithms, leading them to believe that the algorithms may work poorly or be impossible to debug. That idea, however, is not correct. Probabilistic algorithms incorporate an element of randomness (they're also referred to as *randomized algorithms*), which is quantifiable. It can be analyzed, allowing strong predictions about the algorithm's behavior to be made. There is, in fact, very little uncertainty about how the algorithms will behave.

This article addresses two types of probabilistic algorithms: those that explicitly introduce randomness through a **rand()** call, and those that convert input data into a uniform distribution to achieve a similar effect. I'll discuss specific algorithms that fall into these classes, including LogLog (and its better-known successor HyperLogLog) and Bimodal Multicast, and the problems they attempt to solve.

In a third type of probabilistic algorithm, randomness is intrinsic to the data. This applies to problems such as GPS tracking, self-driving cars, sensor networks, and missile-guidance systems. For example, sensor networks often try to make a statement about the network as a whole despite having only partial data. In the case of self-driving cars and

missile-guidance systems, the data is in continuous flux. By the time a program has processed the data, the current state has changed. So it's already wrong and that must be taken into account. This article does not discuss this type of algorithm, but for more information on the set of problems in this category and the algorithms to solve them, it's helpful to read up on Kalman filters¹¹ and estimation algorithms in general.

COUNT-DISTINCT

The count-distinct problem involves counting the number of unique items in a stream that may contain duplicates. More formally: find the cardinality of a multiset. Many problems fall into this category. For example: How many unique IPs have connected to a server over the past hour? How many different words are in a huge corpus of text? How many different users visited a popular Web site in a day? Or even, how many unique URLs have been requested through an HTTP proxy?

The naive solution to this problem is easy. You could easily have written the code in your head before getting to the end of the previous sentence. Here is some Python code that illustrates the solution:

```
...
def count_distinct(stream):
    seen = set()
    for item in stream:
        seen.add(item)
    return len(seen)
...
```

As with many problems, the difficulties with count-distinct come with scale. It may be easy and cheap to count a thousand or even a million unique users, IPs, URLs, or words, but how about 100 million? Still not too bad. How about 100 million per server across thousands of servers? Now it's starting to get interesting.

The problem now is: perform a count-distinct across 1,000 servers, which could be as many as 100 million unique items each, and find the cardinality of the union of their result. In other words: distributed count-distinct. Let's consider a naive solution.

Each server could keep a set of "seen" items, just as in the previous code example. When each finishes its individual count-distinct run, it sends the entire contents of that "seen" set to a central server, which can then take the union of all the sets and return the cardinality of the combined set.

```
...
def combined_cardinality(seen_sets):
    combined = set()
    for seen in seen_sets:
        combined |= seen
    return len(combined)
...
```

Again, the naive solution is delightfully simple and obvious, but it comes at a steep cost. If each server is seeing as many as 100 million unique items, then the size of that "seen" list is going to be significant. In the case of URLs, the average length is around 76 characters. This would put the list at approximately 7.6 GB of data per server. Even if each of

the URLs could be hashed down to a 64-bit integer, the size would be 800 MB. That's much more manageable, but what if there are thousands of servers? Each of those servers sends its "seen" list to a central server. In the case of just 1,000 servers, that's 800 GB of data to dump into the **combined_cardinality** function.

If you need to do this distributed count-distinct often, then you're either going to need to install one of those "big data" systems (and hire a team to run it) or find a different solution.

Enter HyperLogLog. This algorithm takes a probabilistic approach to the count-distinct problem. Its core is based on two observations: first, the probability of any particular bit being set in the binary representation of a random number is 50 percent; second, the probability of two independent events, **A** and **B**, both occurring is **P(A)*P(B)**. Thus, if the probability that any one specific bit is set in that random number is 50 percent, then the probability that any *two* specific bits are set is 25 percent, and the probability that any *three* specific bits are set is 12.5 percent. You get the picture.

The other bit of Probability Theory 101 to remember is that the expected number of trials until an event occurs is **1 / P(event)**. Therefore, if **P(one specific bit set)** is 50 percent, then its expected number of trials is two. For two specific bits, the expectation is four; for three, eight; and so on.

Input values are generally not uniformly distributed random numbers, so you need a way of transforming the input values into values that resemble a uniform distribution—in other words, a hash function. A word of warning: in some applications of hash functions the

distribution is not critical to the correctness of the system. HyperLogLog is very sensitive to this, however. If certain bits of the hash function output are significantly more likely than others to be set or unset, it completely throws off the math that underlies the algorithm. (I've had good success with MurmurHash3.¹³ To see how your preferred hash function stacks up, take a look at SMHasher,¹⁵ which does a great job of finding flaws in hash functions.)

The first step, then, is to take the input item and hash it to a set of bits. Then count the number of leading bits that are set and keep a record of the *maximum* number of leading bits that have been set so far. This would provide a rough estimate of the number of unique items processed. The probability of seeing one leading bit set is 50 percent, so you would “expect” that the number of unique items is two. With three leading bits set, then, on average, you would see eight unique items.

The way to improve the estimate—and the unique idea behind HyperLogLog—is to divide the incoming items into a series of “buckets” based on their trailing bits. Remember the maximum leading number of set bits for each of the buckets: by dividing the incoming items into buckets like this, you can develop a much more accurate estimate of the total cardinality. If you have **m** buckets and **n** unique items, then on average each bucket will have seen **n/m** unique items. Therefore, taking the mean of those buckets provides a reasonable estimate of **log₂(n/m)**, from which you can easily generate an estimate.

Further, HyperLogLog has the excellent property of

allowing you to take separate HyperLogLog instances of the same number of buckets, as well as the maximum of each of their corresponding buckets, and end up with a HyperLogLog that is the *union* of those two previous HyperLogLogs.

That's a rough sketch of how the HyperLogLog algorithm works. The primary change HyperLogLog introduces is the use of a harmonic mean instead of an arithmetic mean. (For further information, read the original LogLog⁶ and HyperLogLog⁸ papers.)

Consider the naive implementation of the distributed count-distinct problem. Assume that you have 1,000 servers and 100 million unique items per server. During every run of the algorithm, the central server will need to process 800 GB of data. This also implies that 800 GB of data will need to traverse the network. How does HyperLogLog change this?

According to the analysis done by the authors of the original paper, you can expect accuracy of about 2 percent while using roughly 1.5 KB of space for the buckets. Each server keeps its own 1.5-KB HyperLogLog, then sends it to the central server. For 1,000 servers, the central server is processing about 1.5 MB of data per run of the distributed count-distinct. In other words, you're processing only about 0.0002 percent of the data you were processing for the naive solution.

This completely changes the economics of the problem. Suddenly you could run many of these, and run them more frequently. You wouldn't need a big data system; you wouldn't even need more than one server—all for the cost of a 2 percent skew in your estimates.

NEAREST-NEIGHBOR SEARCH

The nearest-neighbor search problem involves finding items that are similar within a set—for example, books, music, products, photos, and videos. This is actually a twofold problem: What does *similar* mean, and how do you find items that are similar to each other?

The first half of the problem is known as *feature extraction*. It defines what is being compared about the items. For example, if you were to compare two books character by character, then the character and its location would be your features. In practice, that's not terribly useful. Instead, you might choose to extract the relative count of each unique word and use that as your set of features for the book. Imagine the following quote is a “book”:

One is still what one is going to cease to be
and already what one is going to become.
One lives one's death, one dies one's life.

— *Jean-Paul Sartre*

Extracting the counts of each word from this “book” would result in a “bag of words,” visualized in this way:

{ already: 1, and: 1, be: 1, become: 1, cease: 1,
death: 1, die: 1, going: 2, is: 3, life: 1, lives: 1,
one: 7, still: 1, to: 3, what: 2 }

That's just one way of doing feature extraction. You could choose to extract pairs of words instead:

{ already_what: 1, and_already: 1, be_and:
1, become_one: 1, cease_to: 1, death_one:
1, ... }

In all cases, however, the output of feature extraction can be represented as a *vector*. A book with 10,000 unique words could be represented as one vector in a 10,000-dimensional space.

That might seem odd, but representing the features this way allows for some nifty maneuvers. For example, if you think of each book as a single point in an n -dimensional space, you could measure similarity by calculating the Euclidean distance⁷ between them. You could also measure the angle between the vectors. Cosine similarity³ works well for that technique.

There are many techniques for measuring similarity, but they all fall prey to an insidious problem: *the curse of dimensionality*.⁴ Succinctly, the curse of dimensionality is a group of phenomena that occurs when attempting to work with data in a very high-dimensional space due to the volume of space increasing exponentially with each dimension. The problem is that the amount of work necessary to calculate the Euclidean distance or the cosine similarity of two vectors grows in relation to the number of dimensions. Doing those calculations on vectors with many dimensions is very time-consuming.

Enter another probabilistic algorithm: LSH (locality-sensitive hashing). The idea behind LSH is basically the opposite of traditional hashing. In traditional hashing, two input values that are similar but not the same need to have very different outputs. In LSH, two inputs that are similar should have similar outputs, but in a much, much smaller

amount of data. In practice, I've used LSH to convert million-dimensional vectors to 512-bit hashes, and it worked well.

Clearly, if high-dimensional vectors are being shrunk down to a small "signature," then data will be lost. That is where the probabilistic part comes in.

There are a number of different ways to generate an LSH. The following example is one of the simpler ways.

Let's say we have a corpus of book texts. We have already extracted a vocabulary with 100,000 entries. Thus, the feature vectors for each of these books will have 100,000 dimensions. Let's also say that we've decided to make a 256-bit LSH function. Now we generate 256 completely random 100,000-dimension vectors. These random vectors are the basis of our LSH function.

The LSH function computes the dot product of the input vector and each of the random vectors that were generated ahead of time. For each random vector, if the dot product is positive, we record a 1 bit. If it is negative, we record a 0 bit. We end up with a 256-bit output hash.

The best way to think about this is that the random vectors effectively "divide" the vector space. By answering whether the input vector is greater than or less than each of them, we can approximate its "angle" in that space. Therefore, what this LSH function actually does is produce a hash that can be used to approximate the cosine-similarity between any two of the hashes. The cosine-similarity of two books will be proportional to the Hamming distance between the hashes of them.

RELIABLE BROADCAST

Reliably broadcasting messages among a large number of peers across a high-latency lossy network (i.e., the Internet) is a difficult problem. It is, in fact, the exact problem that we had to solve at Fastly in order to create our Instant Purging system. Normally when faced with this problem, engineers will attempt to solve it with a centralized, single source of truth. That system then either pushes updates to all servers or is pulled from by all servers. Relying on that single source of truth, however, also makes it a single source of failure. If that centralized system goes away, or is inaccessible to all or a subset of the servers, then it is game over for the system's reliability.

Are there alternative solutions? At first glance, this problem appears to be one that could be solved with *atomic broadcast*.⁵ One of the tenets of atomic broadcast, however, is the property of *total order*. In other words, if you have two messages, A and B, every server always sees them in the same order. While that property can be useful to some applications, it introduces a *head-of-line blocking* problem.¹⁰ If message A comes before message B, but message A is delayed getting to a server, then every server must wait to receive B until that one server receives A. In our particular application, the order is irrelevant, and the potential additional latency caused by head-of-line blocking is unacceptable.

We kept looking for alternatives and eventually decided on an algorithm called Bimodal Multicast, a multiphase

protocol that reliably and probabilistically distributes messages over a network of servers. The first phase consists of unreliably broadcasting a message from one peer to all others. The mechanism for that broadcast is not terribly important—you could use IP multicast if it's available, for example. What is important is that the messages get out to all servers as quickly as possible, even if they have a chance of being lost along the way.

The second phase of this algorithm is a *gossip protocol*⁹ used for *anti-entropy*. Every so often each server independently chooses at random another server in the network. It sends that chosen server a digest of its current state, essentially a summarized version of everything that server has seen until this point. The specific format of that digest is not a dictated part of the protocol, but the idea is to send the summary in such a way that when the server receives the digest it can determine which messages it has missed. It will then respond with the list of messages it has not yet received, and the gossip-initiating server will resend those messages.

This system is probabilistic in that servers choose which other servers to gossip with at random. It's possible that the server that they choose to gossip with in some round may be missing the same messages that they are. In that case, they won't immediately recover the missing messages. Even though the way that missing messages propagate through the system is random, it is well understood and follows a classic "epidemic" pattern. The speed and probability with

which a lost message will make its way through the network can be tuned by changing the frequency of gossip as well as the length of time that servers remember old messages.

The version of this system that is at the core of our purging system is tuned such that the probability of losing messages permanently is astronomically unlikely—a likelihood of about 10^{-312} percent. At the current rate of purges, we would expect to lose a message as a result of the randomness in the algorithm once every 3×10^{300} years. In other words, “with high probability” is entirely sufficient, as long as you know what that probability is.

Many problems that are hard to solve precisely become quite feasible with probabilistic algorithms. Consistent hashing² is often applied in load balancing and distributed databases. The Mincut algorithm¹² can be applied to problems in network routing. Bloom filters¹ exist in many databases and network applications. Quicksort¹⁴ is practically the standard sort algorithm in use today. Even basic run-of-the-mill hash tables are actually probabilistic. A programmer’s day-to-day work is filled with probabilistic algorithms and data structures. They tend not to be noticed because they work.

Indeed, probabilistic algorithms are all around us. They can help solve hard problems without resorting to racks upon racks of machines. They can help solve problems that are otherwise practically unsolvable. Probabilistic algorithms should be at the forefront of the minds of those designing and building large systems.

REFERENCES

1. Bloom filters. Wikipedia; https://en.wikipedia.org/wiki/Bloom_filter.
2. Consistent hashing. Wikipedia; https://en.wikipedia.org/wiki/Consistent_hashing.
3. Cosine similarity. Wikipedia; https://en.wikipedia.org/wiki/Cosine_similarity.
4. Curse of dimensionality. Wikipedia; https://en.wikipedia.org/wiki/Curse_of_dimensionality.
5. Defago, X., Schiper, A., Urban, P. 2004. Total order broadcast and multicast algorithms: taxonomy and survey. *ACM Computing Surveys* 36(4): 372-421; <http://dl.acm.org/citation.cfm?doid=1041680.1041682>.
6. Durand, M., Flajolet, P. 2003. LogLog counting of large cardinalities. European Symposium on Algorithms; <http://www.ic.unicamp.br/~celio/peer2peer/math/bitmap-algorithms/durand03loglog.pdf>.
7. Euclidean distance. Wikipedia; https://en.wikipedia.org/wiki/Euclidean_distance.
8. Flajolet, P., Fusy, E., Gandouet, O., Meunier, F. 2007. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. Conference on Analysis of Algorithms; <http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf>.
9. Gossip protocol. Wikipedia; https://en.wikipedia.org/wiki/Gossip_protocol.
10. Head-of-line blocking problem. Wikipedia; https://en.wikipedia.org/wiki/Head-of-line_blocking.

11. Kalman, R. E. 1960. A new approach to linear filtering and prediction problems. *Transactions of the ASME – Journal of Basic Engineering* 82 (series D): 35-45; <http://www.cs.unc.edu/~welch/kalman/kalmanPaper.html>.
12. Mincut algorithm. Wikipedia; https://en.wikipedia.org/wiki/Karger%27s_algorithm.
13. MurmurHash3; <https://code.google.com/p/smhasher/wiki/MurmurHash3>.
14. Quicksort. Wikipedia; <https://en.wikipedia.org/wiki/Quicksort>.
15. SMHasher; <https://code.google.com/p/smhasher/>.

FURTHER INFORMATION

A more thorough explanation of the techniques for generating locality-sensitive hashes is too large a subject to be contained in this article. See the following list of recommended related articles:

Andoni, A., Indyk, P. 2008. Near-optimal hashing algorithms approximate nearest neighbor in high dimensions. *Communications of the ACM* 51(1):117-122; <http://people.csail.mit.edu/indyk/p117-andoni.pdf>.

Andoni, A., Indyk, P., Patrascu, M. 2006. On the optimality of the dimensionality reduction method. *Foundations of Computer Science*:449-458; <http://people.csail.mit.edu/mip/papers/eps2n/eps2n.pdf>

Andoni, A., Indyk, P., Nguyen, H. L., Razenshteyn, I. 2013. Beyond locality-sensitive hashing; <http://arxiv.org/pdf/1306.1547.pdf>.

Datar, M., Indyk, P., Immorlica, N., Mirrokni, V. S. 2004. Locality-sensitive hashing scheme based on p-stable distributions; <http://www.cs.princeton.edu/courses/archive/spring05/cos598E/bib/p253-datar.pdf>.

Gionis, A., Indyk, O., Motwani, R. 1999. Similarity search in high dimensions via hashing. Proceedings of the 25th International Conference on Very Large Databases; 518-529; <http://www.cs.princeton.edu/courses/archive/spring13/cos598C/Gionis.pdf>.

Indyk, P., Motwani, R. 1998. Approximate nearest neighbors: toward removing the curse of dimensionality. Proceedings of the 30th Annual ACM Symposium on Theory of Computing; 604-613; <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.38.249&rep=rep1&type=pdf>.

LOVE IT, HATE IT? LET US KNOW feedback@queue.acm.org

Tyler McMullen (<https://twitter.com/tbmcmullen>) is CTO at Fastly (fastly.com), where he's responsible for the system architecture and leads the company's technology vision. As part of the founding team, McMullen built the first versions of Fastly's Instant Purging system, API, and Real-time Analytics. Before Fastly, he worked on text analysis and recommendations at Scribd. A self-described technology curmudgeon, he has experience in everything from Web design to kernel development, and loathes all of it—especially distributed systems.

© 2015 ACM 1542-7730/15/0900 \$10.00