DOI:10.1145/3080008

Article development led by acmqueue
queue.acm.org

The approximate approach is often faster and more efficient.

BY GRAHAM CORMODE

Data Sketching

DO YOU EVER feel overwhelmed by an unending stream of information? It can seem like a barrage of new email and text messages demands constant attention, and there are also phone calls to pick up, articles to read, and knocks on the door to answer. Putting these pieces together to keep track of what is important can be a real challenge.

The same information overload is a concern in many computational settings. Telecommunications companies, for example, want to keep track of the activity on their networks, to identify overall network health and spot anomalies or changes in behavior. Yet, the scale of events occurring is huge: many millions of network events per hour, per network element. While new technologies allow the scale and granularity of events being monitored to increase by orders of magnitude, the capacity of computing elements (processors, memory, and disks) to make sense of these is barely increasing. Even on a small scale, the amount of information may be too large to store in an impoverished setting (say, an embedded device) or to keep conveniently in fast storage.

In response to this challenge, the model of streaming data processing has grown in popularity. The aim is no longer to capture, store, and index every minute event, but rather to process each observation quickly in order to create a summary of the current state. Following its processing, an event is dropped and is no longer accessible. The summary that is retained is often referred to as a *sketch* of the data.

Coping with the vast scale of information means making compromises: The description of the world is approximate rather than exact; the nature of queries to be answered must be decided in advance rather than after the fact; and some questions are now insoluble. The ability to process vast quantities of data at blinding speeds with modest resources, however, can more than make up for these limitations.

As a consequence, streaming methods have been adopted in a number of domains, starting with telecommunications but spreading to search engines, social networks, finance, and time-series analysis. These ideas are also finding application in areas using traditional approaches, but where the rough-and-ready sketching approach is more cost effective. Successful applications of sketching involve a mixture of algorithmic tricks, systems know-how, and mathematical insight, and have led to new research contributions in each of these areas.

This article introduces the ideas behind sketching, with a focus on algorithmic innovations. It describes some algorithmic developments in the abstract, followed by the steps needed to put them into practice, with examples. The article also looks at four novel algorithmic ideas and discusses some emerging areas.

Simply Sampling

When faced with a large amount of information to process, there may be

PHOTO BY TAFFPIXTURE

a strong temptation just to ignore it entirely. A slightly more principled approach is just to ignore *most of it*—that is, take a small number of examples from the full dataset, perform the computation on this subset, and then try to extrapolate to the full dataset. To give a good estimation, the examples must be randomly chosen. This is the realm of sampling.

There are many variations of sampling, but this article uses the most basic: uniform random sampling. Consider a large collection of customer records. Randomly selecting a small number of records provides the sample. Then various questions can be answered accurately by looking only at the sample: for example, estimating what fraction of customers live in a certain city or have bought a certain product.

The method. To flesh this out, let's fill in a few gaps. First, how big should the sample be to supply good answers? With standard statistical results, for questions like those in the customer records example, the standard error of a sample of size `s` is proportional to $1/\sqrt{s}$. Roughly speaking, this means that in estimating a proportion from the sample, the error would be expected to look like $\pm 1/\sqrt{s}$. Therefore, looking at the voting intention of a subset of 1,000 voters produces an opinion poll whose error is approximately 3%—providing high confidence (but not certainty) that the true answer is within 3% of the result on the sample, assuming the sample was drawn randomly and the participants responded honestly. Increasing the size of the sample causes the error to decrease in a predictable, albeit expensive, way: reducing the margin of error of an opinion poll to 0.3% would require contacting 100,000 voters.

Second, how should the sample be drawn? Simply taking the first `s` records is not guaranteed to be random; there may be clustering through the data. You need to ensure every record has an equal chance of being included in the sample. This can be achieved by using standard random-number generators to pick which records to include in the sample. A common trick is to attach a random number to each record, then sort the data based on this random tag and take the first `s` records in the sorted order. This works fine, as long as sorting the full dataset is not too costly.

Finally, how do you maintain the sample as new items are arriving? A simple approach is to pick every record with probability `p`, for some chosen value of `p`. When a new record comes, pick a random fraction between 0 and 1, and if it is smaller than `p`, put the record in the sample. The problem with this approach is that you do not know in advance what `p` should be. In the previous analysis a fixed sample size s was desired, and using a fixed sampling rate p means there are too few elements initially, but then too many as more records arrive.

Presented this way, the question has the appearance of an algorithmic puzzle, and indeed this was a common question in technical interviews for many years. One can come up with clever solutions that incrementally adjust p as new records arrive. A simple and elegant way to maintain a sample is to adapt the idea of random tags. Attach to each record a random tag, and define the sample to be the s records with the smallest tag values. As new records arrive, the tag values decide whether to add the new record to the sample (and to remove an old item to keep the sample size fixed at s).

Discussion and applications. Sampling methods are so ubiquitous that there are many examples to consider. One simple case is within database systems. It is common for the database management system to keep a sample of large relations for the purpose of query planning. When determining how to execute a query, evaluating different strategies provides an estimate of how much data reduction may occur at each step, with some uncertainty of course. Another example comes from the area of data integration and linkage, in which a subproblem is to test whether two columns from separate tables can relate to the same set of entities. Comparing the columns in full can be time consuming, especially when you want to test all pairs of columns for compatibility. Comparing a small sample is often sufficient to determine whether the columns have any chance of relating to the same entities.

Entire books have been written on the theory and practice of sampling, particularly around schemes that try to sample the more important elements preferentially, to reduce the error in estimating from the sample. For a good survey with a computational perspective, see *Synopses for Massive Data: Samples, Histograms, Wavelets and Sketches*.[11]

Given the simplicity and generality of sampling, why would any other method be needed to summarize data? It turns out that sampling is not well suited for some questions. Any question that requires detailed knowledge of individual records in the data cannot be answered by sampling. For example, if you want to know whether one specific individual is among your customers, then a sample will leave you uncertain. If the customer is not in the sample, you do not know whether this is because that person is not in the data or because he or she did not happen to be sampled. A question like this ultimately needs all the presence information to be recorded and is answered by highly compact encodings such as the Bloom filter (described later).

A more complex example is when the question involves determining the cardinality of quantities. In a dataset that has many different values, how many distinct values of a certain type are there? For example, how many distinct surnames are in a particular customer dataset? Using a sample does not reveal this information. Let's say in a sample size of 1,000 out of one million records, 900 surnames occur just once among the sampled names. What can you conclude about the popularity of these names in the rest of the dataset? It might be that almost every other name in the full dataset is also unique. Or it might be that each of the unique names in the sample reoccurs tens or hundreds of times in the remainder of the data. With the sampled information there is no way to distinguish between these two cases, which leads to huge confidence intervals on these kinds of statistics. Tracking information about cardinalities, and omitting duplicates, is addressed by techniques such as HyperLogLog, addressed later.

Finally, there are quantities that samples can estimate, but for which better special-purpose sketches exist. Recall that the standard error of a sample of size s is $1/\sqrt{s}$. For problems such as estimating the frequency of a particular attribute (such as city of residence), you can build a sketch of size s so the error it guarantees is proportional to $1/s$. This is considerably stronger than the sampling guarantee and only improves as we devote more space s to the sketch. The Count-Min sketch described later in this article has this property. One limitation is that the attribute of interest must be specified in advance of setting up the sketch, while a sample allows you to evaluate a query for any recorded attribute of the sampled items.

Because of its flexibility, sampling is a powerful and natural way of building a sketch of a large dataset. There are many different approaches to sampling that aim to get the most out of the sample or to target different types of queries that the sample may be used to answer.[11] Here, more information is presented about less flexible methods that address some of these limitations of sampling.

Summarizing Sets with Bloom Filters

The Bloom filter is a compact data structure that summarizes a set of items. Any computer science data-structures class is littered with examples of "dictionary" data structures, such as arrays, linked lists, hash tables, and many esoteric variants of balanced tree structures. The common feature of these structures is that they can all answer "membership questions" of the form: Is a certain item stored in the structure or not? The Bloom filter can also respond to such membership questions. The answers given by the structure, however, are either "the item has definitely not been stored" or "the item has *probably* been stored." This introduction of uncertainty over the state of an item (it might be thought of as introducing potential false positives) allows the filter to use an amount of space that is much smaller than its exact relatives. The filter also does not allow listing the items that have been placed into it. Instead, you can pose membership questions only for specific items.

The method. To understand the filter, it is helpful to think of a simple exact solution to the membership problem. Suppose you want to keep track of which of a million possible items you have seen, and each one is helpfully labeled with its ID number (an integer between one and a million). Then you can keep an array of one million bits, initialized to all 0s. Every time you see an item i, you just set the ith bit in the array to 1. A lookup query for item j is correspondingly straightforward: just see whether bit j is a 1 or a 0. The structure is very compact: 125KB will suffice if you pack the bits into memory.

Real data, however, is rarely this nicely structured. In general, you might have a much larger set of possible inputs—think again of the names of customers, where the number of possible name strings is huge. You can nevertheless adapt your bit-array approach by borrowing from a different dictionary structure. Imagine the bit array is a hash table: you will use a hash function h to map from the space of inputs onto the range of indices for your table. That is, given input i, you now set bit hi to 1. Of course, now you have to worry about hash collisions in which multiple entries might map onto the same bit. A traditional hash table can handle this, as you can keep information about the entries in the table. If you stick to your guns and keep the bits only in the bit array, however, false positives will result: if you look up item i, it may be that entry h_i is set to 1, but i has not been seen; instead, there is some item j that was seen, where $h(i) = h(j)$.

Can you fix this while sticking to a bit array? Not entirely, but you can make it less likely. Rather than just hashing each item i once, with a single hash function, use a collection of k hash functions $h_1, h_2, \dots h_k$, and map i with each of them in turn. All the bits corresponding to $h_1(i), h_2(i) \dots h_k(i)$ are set to 1. Now to test membership of j, check all the entries it is hashed to, and say no if any of them are 0.

There's clearly a trade-off here: Initially, adding extra hash functions reduces the chances of a false positive as more things need to "go wrong" for an incorrect answer to be given. As more and more hash functions are added, however, the bit array gets fuller and fuller of 1 values, and therefore collisions are more likely. This trade-off can be analyzed mathematically, and the sweet spot found that minimizes the chance of a false positive. The analysis works by assuming that the hash functions look completely random (which is a reasonable assumption in practice), and by looking at the chance that an arbitrary element not in the set is reported as present.

If n distinct items are being stored in a Bloom filter of size m, and k hash functions are used, then the chance of a membership query that should receive a negative answer yielding a false positive is approximately $\exp(k \ln(1 \exp(kn/m)))$.[4] While extensive study of this expression may not be rewarding in the short term, some simple analysis shows that this rate is minimized by picking $k = (m/n) \ln 2$. This corresponds to the case when about half the bits in the filter are 1 and half are 0.

For this to work, the number of bits in the filter should be some multiple of the number of items that you expect to store in it. A common setting is $m = 10n$ and $k = 7$, which means a false positive rate below 1%. Note that there is no magic here that can compress data beyond information-theoretical limits: under these parameters, the Bloom filter uses about 10 bits per item and must use space proportional to the number of different items stored. This is a modest savings when representing integer values but is a considerable benefit when the items stored have large descriptions—say, arbitrary strings such as URLs. Storing these in a traditional structure such as a hash table or balanced search tree would consume tens or hundreds of bytes per item. A simple example is shown in Figure 1, where an item i is mapped by $k = 3$ hash functions to a filter of size $m = 12$, and these entries are set to 1.

Discussion and applications. The possibility of false positives needs to be handled carefully. Bloom filters are at their most attractive when the consequence of a false positive is not the introduction of an error in a computation, but rather when it causes some additional work that does not adversely impact the overall performance of the system. A good example comes in the context of browsing the Web. It is now common for Web browsers to warn users if they are attempting to visit a site that is known to host malware. Checking the URL against a database of "bad" URLs does this. The database is large enough, and URLs are long enough, that keeping the full database, as part of the browser would be unwieldy, especially on mobile devices.

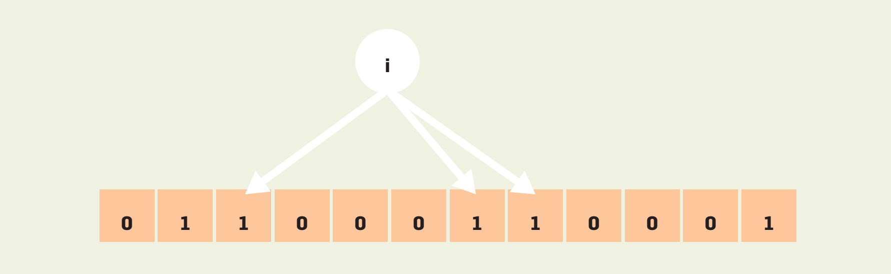
Instead, a Bloom filter encoding of the database can be included with the browser, and each URL visited can be checked against it. The consequence of a false positive is that the browser may believe that an innocent site is on the bad list. To handle this, the browser can contact the database authority and check whether the full URL is on the list. Hence, false positives are removed at the cost of a remote database lookup.

Notice the effect of the Bloom filter: it gives the all clear to most URLs and incurs a slight delay for a small fraction (or when a bad URL is visited). This is preferable both to the solution of keeping a copy of the database with the browser and to doing a remote lookup for every URL visited. Browsers such as Chrome and Firefox have adopted this concept. Current versions of Chrome use a variation of the Bloom filter based on more directly encoding a list of hashed URLs, since the local copy does not have to be updated dynamically and more space can be saved this way.

The Bloom filter was introduced in 1970 as a compact way of storing a dictionary, when space was really at a premium.[3] As computer memory grew, it seemed that the filter was no longer needed. With the rapid growth of the Web, however, a host of applications for the filter have been devised since around the turn of the century.[4] Many of these applications have the flavor of the preceding example: the filter gives a fast answer to lookup queries, and positive answers may be double-checked in an authoritative reference.

Bloom filters have been widely used to avoid storing unpopular items in caches. This enforces the rule that an item is added to the cache only if it has

Figure 1. Bloom filter with K=3, M=12.



been seen before. The Bloom filter is used to compactly represent the set of items that have been seen. The consequence of a false positive is that a small fraction of rare items might also be stored in the cache, contradicting the letter of the rule. Many large distributed databases (Google's Bigtable, Apache's Cassandra and HBase) use Bloom filters as indexes on distributed chunks of data. They use the filter to keep track of which rows or columns of the database are stored on disk, thus avoiding a (costly) disk access for nonexistent attributes.

Counting with Count-Min Sketch

Perhaps the canonical data summarization problem is the most trivial: to count the number of items of a certain type that have been observed, you do not need to retain each item. Instead, a simple counter suffices, incremented with each observation. The counter has to be of sufficient bit depth in order to cope with the magnitude of events observed. When the number of events gets truly huge, ideas such as Robert Morris's approximate counter can be used to provide such a counter in fewer bits[12] (another example of a sketch).

When there are different types of items, and you want to count each type, the natural approach is to allocate a counter for each item. When the number of item types grows huge, however, you encounter difficulties. It may not be practical to allocate a counter for each item type. Even if it is, when the number of counters exceeds the capacity of fast memory, the time cost of incrementing the relevant counter may become too high. For example, a social network such as Twitter may wish to track how often a tweet is viewed when displayed via an external website. There are billions of Web pages, each of which could in principle link to one or more tweets, so allocating counters for each is infeasible and unnecessary. Instead, it is natural to look for a more compact way to encode counts of items, possibly with some tolerable loss of fidelity.

The Count-Min sketch is a data structure that allows this trade-off to be made. It encodes a potentially massive number of item types in a small array. The guarantee is that large counts will be preserved fairly accurately, while small counts may incur greater (relative) error. This means it is good for applications where you are interested in the head of a distribution and less so in its tail.

The method. At first glance, the sketch looks quite like a Bloom filter, as it involves the use of an array and a set of hash functions. There are significant differences in the details, however. The sketch is formed by an array of counters and a set of hash functions that map items into the array. More precisely, the array is treated as a sequence of rows, and each item is mapped by the first hash function into the first row, by the second hash function into the second row, and so on (note that this is in contrast to the Bloom filter, which allows the hash functions to map onto overlapping ranges). An item is processed by mapping it to each row in turn via the corresponding hash function and incrementing the counters to which it is mapped.

Given an item, the sketch allows its count to be estimated. This follows a similar outline to processing an update: inspect the counter in the first row where the item was mapped by the first hash function, and the counter in the second row where it was mapped by the second hash, and so on. Each row has a counter that has been incremented by every occurrence of the item. The counter was also potentially incremented by occurrences of other items that were mapped to the same location, however, since collisions are expected. Given the collection of counters containing the desired count, plus noise, the best guess at the true count of the desired item is to take the smallest of these counters as your estimate.

Figure 2. Count-min sketch data structure with four rows, nine columns.

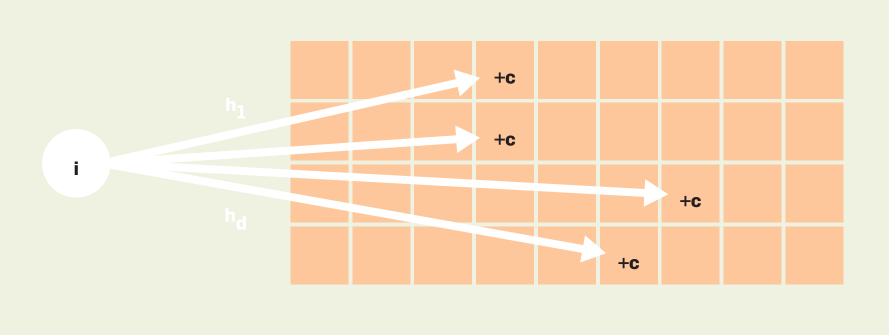


Figure 2 shows the update process: an item i is mapped to one entry in each row j by the hash function h_j, and the update of c is added to each entry. It can also be seen as modeling the query process: a query for the same item i will result in the same set of locations being probed, and the smallest value returned as the answer.

Discussion and applications. As with the Bloom filter, the sketch achieves a compact representation of the input, with a trade-off in accuracy. Both provide some probability of an unsatisfactory answer. With a Bloom filter, the answers are binary, so there is some chance of a false positive response; with a Count-Min sketch, the answers are frequencies, so there is some chance of an inflated answer.

What may be surprising at first is that the obtained estimate is very good. Mathematically, it can be shown that there is a good chance that the returned estimate is close to the correct value. The quality of the estimate depends on the number of rows in the sketch (each additional row halves the probability of a bad estimate) and on the number of columns (doubling the number of columns halves the scale of the noise in the estimate). These guarantees follow from the random selection of hash functions and do not rely on any structure or pattern in the data distribution that is being summarized. For a sketch of size s, the error is proportional to $1/s$. This is an improvement over the case for sampling where, as noted earlier, the corresponding behavior is proportional to $1/\sqrt{s}$.

Just as Bloom filters are best suited for the cases where false positives can be tolerated and mitigated, Count-Min sketches are best suited for handling a slight inflation of frequency. This means, in particular, they do not apply to cases where a Bloom filter might be used: if it matters a lot whether an item has been seen or not, then the uncertainty that the Count-Min sketch introduces will obscure this level of precision. The sketches are very good for tracking which items exceed a given popularity threshold, however. In particular, while the size of a Bloom filter must remain proportional to the size of the input it is representing, a Count-Min sketch can be much more compressive: its size can be considered to be independent of the input size, depending instead on the desired accuracy guarantee only (that is, to achieve a target accuracy of ε, fix a sketch size of s proportional to 1/ε that does not vary over the course of processing data).

The Twitter scenario mentioned previously is a good example. Tracking the number of views that a tweet receives across each occurrence in different websites creates a large enough volume of data to be difficult to manage. Moreover, the existence of some uncertainty in this application seems acceptable: the consequences of inflating the popularity of one website for one tweet are minimal. Using a sketch for each tweet consumes only moderately more space than the tweet and associated metadata, and allows tracking which venues attract the most attention for the tweet. Hence, a kilobyte or so of space is sufficient to track the percentage of views from different locations, with an error of less than one percentage point, say.

Since their introduction over a decade ago,[7] Count-Min sketches have found applications in systems that track frequency statistics, such as popularity of content within different groups—say, online videos among different sets of users, or which destinations are popular for nodes within a communications network. Sketches are used in telecommunications networks where the volume of data passing along links is immense and is never stored. Summarizing network traffic distribution allows hotspots to be detected, informing network-planning decisions and allowing configuration errors and floods to be detected and debugged.[6] Since the sketch compactly encodes a frequency distribution, it can also be used to detect when a shift in popularities occurs, as a simple example of anomaly detection.

Successful applications of sketching involve a mixture of algorthmic tricks, systems know-how, and mathematical insight, and have led to new research contributions in each of these areas.

Counting Distinct Items with HyperLogLog

Another basic problem is keeping track of how many different items have been seen out of a large set of possibilities. For example, a Web publisher might want to track how many different people have been exposed to a particular advertisement. In this case, you would not want to count the same viewer more than once. When the number of possible items is not too large, keeping a list, or a binary array, is a natural solution. As the number of possible items becomes very large, the space needed by these methods grows proportional to the number of items tracked. Switching to an approximate method such as a Bloom filter means the space remains proportional to the number of distinct items, although the constants are improved.

Could you hope to do better? If you just counted the total number of items, without removing duplicates, then a simple counter would suffice, using a number of bits that is proportional to the logarithm of the number of items encountered. If only there were a way to know which items were new, and count only those, then you could achieve this cost.

The HyperLogLog (HLL) algorithm promises something even stronger: the cost needs to depend only on the logarithm of the logarithm of the quantity computed. Of course, there are some scaling constants that mean the space needed is not quite so tiny as this might suggest, but the net result is that quantities can be estimated with high precision (say, up to a 1%–2% error) with a couple of kilobytes of space.

The method. The essence of this method is to use hash functions applied to item identifiers to determine how to update counters so that duplicate items are treated identically. A Bloom filter has a similar property: attempting to insert an item already represented within a Bloom filter means setting a number of bits to 1 that are already recording 1 values. One approach is to keep a Bloom filter and look at the final density of 1s and 0s to estimate the number of distinct items represented (taking into account collisions under hash functions). This still requires space proportional to the number of items and is the basis of early approaches to this problem.[15]

To break this linearity, a different approach to building a binary counter is needed. Instead of adding 1 to the counter for each item, you could add 1 with a probability of one-half, 2 with a probability of one-fourth, 4 with a probability of 1/8th, and so on. This use of randomness decreases the reliability of the counter, but you can check that the expected count corresponds to the true number of items encountered. This makes more sense when using hash functions. Apply a hash function `g` to each item `i`, with the same distribution: `g` maps items to `j` with probability 2^{-j} (say, by taking the number of leading zero bits in the binary expansion of a uniform hash value). You can then keep a set of bits indicating which `j` values have been seen so far. This is the essence of the early Flajolet-Martin approach to tracking the number of distinct items.[8] Here a logarithmic number of bits is needed, as there are only this many distinct `j` values expected.

The HLL method reduces the number of bits further by retaining only the highest `j` value that has been seen when applying the hash function. This might be expected to be correlated to the cardinality, although with high variation for example, there might be only a single item seen, which happens to hash to a large value. To reduce this variation, the items are partitioned into groups using a second hash function (so the same item is always placed in the same group), and information about the largest hash in each group is retained. Each group yields an estimate of the local cardinality; these are all combined to obtain an estimate of the total cardinality.

A first effort would be to take the mean of the estimates, but this still allows one large estimate to skew the result; instead, the harmonic mean is used to reduce this effect. By hashing to `s` separate groups, the standard error is proportional to $1/\sqrt{s}$. A small example is shown in Figure 3. The figure shows a small example HLL sketch with $s = 3$ groups. Consider five distinct items `a, b, c, d, e` with their related hash values. From this, the following array is obtained:

3	2	1

The estimate is obtained by taking 2 to the power of each of the array entries and computing the sum of the reciprocals of these values, obtaining $1/8 + 1/4 + 1/2 = 7/8$ in this case. The final estimate is made by multiplying $\alpha_s s^2$ by the reciprocal of this sum. Here, α_s is a scaling constant that depends on `s`. $\alpha_3 = 0.5305$, so 5.46 is obtained as the estimate—close to the true value of 5.

The analysis of the algorithm is rather technical, but the proof is in the deployment: the algorithm has been widely adopted and applied in practice.

Discussion and applications. One example of HLL's use is in tracking the viewership of online advertising. Across many websites and different advertisements, trillions of view events may occur every day. Advertisers are interested in the number of "uniques:" how many different people (or rather, browsing devices) have been exposed to the content. Collecting and marshaling this data is not infeasible, but rather unwieldy, especially if it is desired to do more advanced queries (say, to count how many uniques saw both of two particular advertisements). Use of HLL sketches allows this kind of query to be answered directly by combining the two sketches rather than trawling through the full data. Sketches have been put to use for this purpose, where the small amount of uncertainty from the use of randomness is comparable to other sources of error, such as dropped data or measurement failure.

Approximate distinct counting is also widely used behind the scenes in Web-scale systems. For example, Google's Sawzall system provides a variety of sketches, including count distinct, as primitives for log data analysis.[13] Google engineers have described some of the implementation modifications made to ensure high accuracy of the HLL across the whole range of possible cardinalities.[10]

Figure 3. Example of HyperLogLog in action.

x	a	b	c	d	e
h(x)	1	2	3	1	3
g(x)	0001	0011	1010	1101	0101

A last interesting application of distinct counting is in the context of social network analysis. In 2016, Facebook set out to test the "six degrees of separation" claim within its social network. The Facebook friendship graph is sufficiently large (more than a billion nodes and hundreds of billions of edges) that maintaining detailed information about the distribution of long-range connections for each user would be infeasible. Essentially, the problem is to count, for each user, how many friends they have at distance 1, 2, 3, and so on. This would be a simple graph exploration problem, except that some friends at distance 2 are reachable by multiple paths (via different mutual friends). Hence, distinct counting is used to generate accurate statistics on reachability without double counting and to provide accurate distance distributions (the estimated number of degrees of separation in the Facebook graph is 3.57).[2]

Advanced Sketching

Roughly speaking, the four examples of sketching described in this article cover most of the current practical applications of this model of data summarization. Yet, unsurprisingly, there is a large body of research into new applications and variations of these ideas. Just around the corner are a host of new techniques for data summarization that are on the cusp of practicality. This section mentions a few of the directions that seem most promising.

Sketching for dimensionality reduction. When dealing with large high-dimensional numerical data, it is common to seek to reduce the dimensionality while preserving fidelity of the data. Assume the hard work of data wrangling and modeling is done and the data can be modeled as a massive matrix, where each row is one example point, and each column encodes an attribute of the data. A common technique is to apply PCA (principal components analysis) to extract a small number of "directions" from the data. Projecting each row of data along each of these directions yields a different representation of the data that captures most of the variation of the dataset.

One limitation of PCA is that finding the direction entails a substantial amount of work. It requires finding eigenvectors of the covariance matrix, which rapidly becomes unsustainable for large matrices. The competing approach of random projections argues that rather than finding "the best" directions, it suffices to use (a slightly larger number of) random vectors. Picking a moderate number of random directions captures a comparable amount of variation, while requiring much less computation.

The random projection of each row of the data matrix can be seen as an example of a sketch of the data. More directly, close connections exist between random projections and the sketches described earlier. The Count-Min sketch can be viewed as a random projection of sorts; moreover, the best constructions of random projections for dimensionality reduction look a lot like Count-Min sketches with some twists (such as randomly multiplying each column of the matrix by either -1 or 1). This is the basis of methods for speeding up high-dimensional machine learning, such as the Hash Kernels approach.[14]

Randomized numerical linear algebra. A grand objective for sketching is to allow arbitrary complex mathematical operations over large volumes of data to be answered approximately and quickly via sketches. While this objective appears quite a long way off, and perhaps infeasible because of some impossibility results, a number of core mathematical operations can be solved using sketching ideas, which leads to the notion of randomized numerical linear algebra. A simple example is matrix multiplication: given two large matrices A and B, you want to find their product AB. An approach using sketching is to build a dimensionality-reducing sketch of each row of A and each column of B. Combining each pair of these provides an estimate for each entry of the product. Similar to other examples, small answers are not well preserved, but large entries are accurately found.

Other problems that have been tackled in this space include regression. Here the input is a high-dimensional dataset modeled as matrix A and column vector b: each row of A is a data point, with the corresponding entry of b the value associated with the row. The goal is to find regression coefficients x that minimize $||Ax-b||_2$. An exact solution to this problem is possible but costly in terms of time as a function of the number of rows. Instead, applying sketching to matrix A solves the problem in the lower-dimensional sketch space.[5] David Woodruff provides a comprehensive mathematical survey of the state of the art in this area.[16]

Rich data: Graphs and geometry. The applications of sketching so far can be seen as summarizing data that might be thought of as a high-dimensional vector, or matrix. These mathematical abstractions capture a large number of situations, but, increasingly, a richer model of data is desired—say, to model links in a social network (best thought of as a graph) or to measure movement patterns of mobile users (best thought of as points in the plane or in 3D). Sketching ideas have been applied here also.

For graphs, there are techniques to summarize the adjacency information of each node, so that connectivity and spanning tree information can be extracted.[1] These methods provide a surprising mathematical insight that much edge data can be compressed while preserving fundamental information about the graph structure. These techniques have not found significant use in practice yet, perhaps because of high overheads in the encoding size.

For geometric data, there has been much interest in solving problems such as clustering.[9] The key idea here is that clustering part of the input can capture a lot of the overall structural information, and by merging clusters together (clustering clusters) you can retain a good picture of the overall point density distribution.

Why Should You Care?

The aim of this article has been to introduce a selection of recent techniques that provide approximate answers to some general questions that often occur in data analysis and manipulation. In all cases, simple alternative approaches can provide exact answers, at the expense of keeping complete information. The examples shown here have illustrated, however, that in many cases the approximate approach can be faster and more space efficient. The use of these methods is growing. Bloom filters are sometimes said to be one of the core technologies that "big data experts" must know. At the very least, it is important to be aware of sketching techniques to test claims that solving a problem a certain way is the only option. Often, fast approximate sketch-based techniques can provide a different trade-off.

Related articles on queue.acm.org

It Probably Works
Tyler McMullen
http://queue.acm.org/detail.cfm?id=2855183

Statistics for Engineers
Heinrich Hartmann
http://queue.acm.org/detail.cfm?id=2903468

References

1. Ahn, K.J., Guha, S., McGregor, A. Analyzing graph structure via linear measurements. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, (2012).
2. Bhagat, S., Burke, M., Diuk, C., Filiz, I.O., Edunov, S. Three-and-a-half degrees of separation. Facebook Research, 2016; https://research.fb.com/three-and-a-half-degrees-of-separation/.
3. Bloom, B. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*, 7 (July 1970), 422–426.
4. Broder, M., Mitzenmacher, A. Network applications of Bloom filters: a survey. *Internet Mathematics 1*, 4 (2005), 485–509.
5. Clarkson, K.L., Woodruff, D.P. Low rank approximation and regression in input sparsity time. In *Proceedings of the ACM Symposium on Theory of Computing*, (2013), 81–90.
6. Cormode, G., Korn, F., Muthukrishnan, S., Johnson, T., Spatscheck, O., Srivastava, D. 2004. Holistic UDAFs at streaming speeds. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, (2004), 35–46.
7. Cormode, G., Muthukrishnan, S. An improved data stream summary: the Count-Min sketch and its applications. *J. Algorithms 55*, 1 (2005), 58–75.
8. Flajolet, P., Martin, G.N. 1985. Probabilistic counting. In *Proceedings of the IEEE Conference on Foundations of Computer Science*, 1985, 76–82. Also in *J. Computer and System Sciences 31*, 182–209.
9. Guha, S., Mishra, N., Motwani, R., O'Callaghan, L. Clustering data streams. In *Proceedings of the IEEE Conference on Foundations of Computer Science*, 2000.
10. Heule, S., Nunkesser, M., Hall, A. HyperLogLog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the International Conference on Extending Database Technology*, 2013.
11. Jermaine, C. Sampling techniques for massive data. Synopses for massive data: samples, histograms, wavelets and sketches. *Foundations and Trends in Databases 4*, 1–3 (2012). G. Cormode, M. Garofalakis, P. Haas, and C. Jermaine, Eds. NOW Publishers.
12. Morris, R. Counting large numbers of events in small registers. *Commun. ACM 21*, 10 (Oct. 1977), 840–842.
13. Pike, R., Dorward, S., Griesemer, R., Quinlan, S. Interpreting the data: Parallel analysis with Sawzall. *Dynamic Grids and Worldwide Computing 13*, 4 (2005), 277–298.
14. Weinberger, K.Q., Dasgupta, A., Langford, J., Smola, A.J., Attenberg, J. Feature hashing for large-scale multitask learning. In *Proceedings of the International Conference on Machine Learning*, 2009.
15. Whang, K.Y., Vander-Zanden, B.T., Taylor, H.M. A linear-time probabilistic counting algorithm for database applications. *ACM Trans. Database Systems 15*, 2 (1990, 208.
16. Woodruff, D. Sketching as a tool for numerical linear algebra. *Foundations and Trends in Theoretical Computer Science 10*, 1–2 (2014), 1–157.

Graham Cormode is a professor of computer science at the University of Warwick, U.K. Previously, he was a researcher at Bell Labs and AT&T on algorithms for data management. He received the 2017 Adams Prize for his work on data analysis.