Sketch of the Day: HyperLogLog — Cornerstone of a Big Data Infrastructure

Intro

In the Zipfian world of AK, the HyperLogLog distinct value (DV) sketch reigns supreme. This DV sketch is the workhorse behind the majority of our DV counters (and we're not alone) and enables us to have a real time, in memory data store with incredibly high throughput. HLL was conceived of by Flajolet et. al. in the phenomenal paper HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. This sketch extends upon the earlier Loglog Counting of Large Cardinalities (Durand et. al.) which in turn is based on ~~the seminal AMS work~~ FM-85, Flajolet and Martin's original work on probabilistic counting. *(Many thanks to Jérémie Lumbroso for the correction of the history here. I am very much looking forward to his upcoming introduction to probabilistic counting in Flajolet's complete works.)* UPDATE – Rob has recently published a blog about PCSA, a direct precursor to LogLog counting which is filled with interesting thoughts. There have been a few posts on HLL recently so I thought I would dive into the intuition behind the sketch and into some of the details.

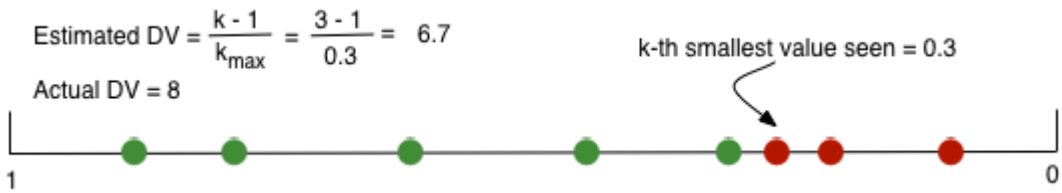
Just like all the other DV sketches, HyperLogLog looks for interesting things in the hashed values of your incoming data. However, unlike other DV sketches HLL is based on bit pattern observables as opposed to KMV (and others) which are based on order statistics of a stream. As Flajolet himself states:

> *Bit-pattern observables*: these are based on certain patterns of bits occurring at the beginning of the (binary) S-values. For instance, observing in the stream S at the beginning of a string a bit- pattern $0^{\rho-1}1$ is more or less a likely indication that the cardinality n of S is at least 2^{ρ}.
>
> *Order statistics observables*: these are based on order statistics, like the smallest (real) values, that appear in S. For instance, if X = min(S), we may legitimately hope that n is roughly of the order of 1/X...

In my mind HyperLogLog is really composed of two insights: Lots of crappy things are sometimes better than one really good thing; and bit pattern observables tell you a lot about a stream. We're going to look at each component in turn.

Bad Estimator

Even though the literature refers to the HyperLogLog sketch as a different family of estimator than KMV I think they are very similar. It's useful to understand the approach of HLL by reviewing the KMV sketch. Recall that KMV stores the smallest k values that you have seen in a stream. From these k values you get an estimate of the number of distinct elements you have seen so far.



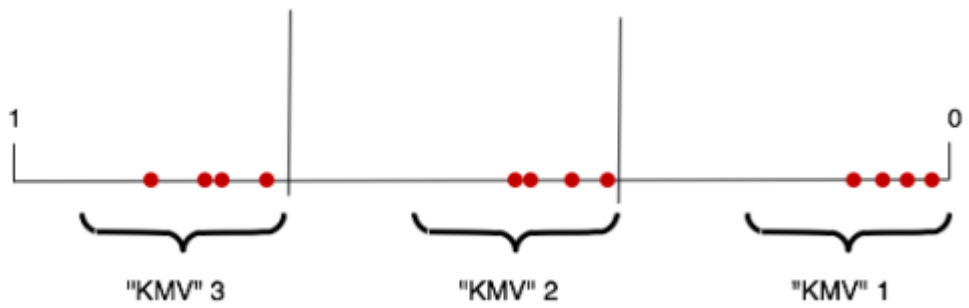
HLL also stores something similar to the smallest values ever seen. To see how this works it's useful to ask "How could we make the KMV sketch smaller?" KMV stores the actual value of the incoming numbers. So you have to store k 64 bit values which is tiny, but not that tiny. What if we just stored the "rank" of the numbers? Let's say the number 94103 comes through (I'll use base 10 here to make things easier). That number is basically $9 * 10^4$ plus some stuff. So, let's just store the exponent, i.e. 4. In this way I get an approximation of the size of numbers I have seen so far. That turns the original KMV algorithm into only having to store the numbers 1-19 (since $2^{64} \approx 10^{19}$) which is a whole lot less than 2^{64}numbers. Of course, this estimate will be much worse than storing the actual values.

Bit Pattern Observables

In actuality HLL, just like all the other DV sketches, uses hashes of the incoming data in base 2. And instead of storing the "rank" of the incoming numbers HLL uses a nice trick of looking for runs of zeroes in the hash values. These runs of zeroes are an example of "bit pattern observables". This concept is similar to recording the longest run of heads in a series of coin flips and using that to guess the number of times the coin was flipped. For instance, if you told me that you spent some time this afternoon flipping a coin and the longest run of heads you saw was 2 I could guess you didn't flip the coin very many times. However, if you told me you saw a run of 100 heads in a row I would gather you were flipping the coin for quite a while. This "bit pattern observable", the run of heads, gives me information about the stream of data it was pulled from. An interesting thing to note is just how probable long runs of heads are. As Mark Shilling points out, you can almost always tell the difference between a human generated set of coin flips and an actual one, due to humans not generating long runs. (The world of coin flipping seems to be a deep and crazy pit.) Disclaimer: The only thing I am trying to motivate here is that by keeping a very small piece of information (the longest run of heads) I can get some understanding of what has happened in a stream. Of course, you could probably guess that even though we have now reduced the storage of our sketch the DV estimate is pretty crummy. But what if we kept more than one of them?

Stochastic Averaging

In order to improve the estimate, the HLL algorithm stores many estimators instead of one and averages the results. However, in order to do this you would have to hash the incoming data through a bunch of independent hash functions. This approach isn't a very good idea since hashing each value a bunch of times is expensive and finding good independent hash families is quite difficult in practice. The work around for this is to just use one hash function and "split up" the input into m buckets while maintaining the observable (longest run of zeroes) for each bucket. This procedure is called stochastic averaging. You could do this split in KMV as well and it's easier to visualize. For an m of 3 it would look like:



To break the input into the m buckets, Durand suggests using the first few (k) bits of the hash value as an index into a bucket and compute the longest run of zeroes (R) on what is left over. For example, if your incoming datum looks like *010100000110* and *k = 3* you could use the 3 rightmost bits, *110*, to tell you which register to update ($110_2 = 6$) and from the remaining bits, *010100000*, you could take the longest run of zeroes (up to some max), which in this case is 5. In order to compute the number of distinct values in the stream you would just take the average of all of the m buckets:

$$DV_{LL} = \text{constant} * m * 2^{\overline{R}}$$

Here $\overline{R}$ is the average of the values R in all the buckets. The formula above is actually the estimator for the LogLog algorithm, not HyperLogLog. To get HLL, you need one more piece...

Harmonic Mean

A fundamental insight that Flajolet had to improve LogLog into HyperLogLog was that he noticed the distribution of the values in the m registers is skewed to the right, and there can be some dramatic outliers that really mess up the average used in LogLog (see Fig. 1 below). He and Durand knew this when they wrote LogLog and did a bunch of hand-wavey stuff (like cut off the top 30% of the register values) to create what he called the "SuperLogLog", but in retrospect this seems kind of dumb. He fixed this in HLL by tossing out the odd rules in SuperLogLog and deciding to take the harmonic mean of the DV estimates. The harmonic mean tends to throw out extreme values and behave well in this type of environment. This seems like an obvious thing to do. I'm a bit surprised they didn't try this in the LogLog paper, but perhaps the math is harder to deal with when using the harmonic mean vs the geometric mean.

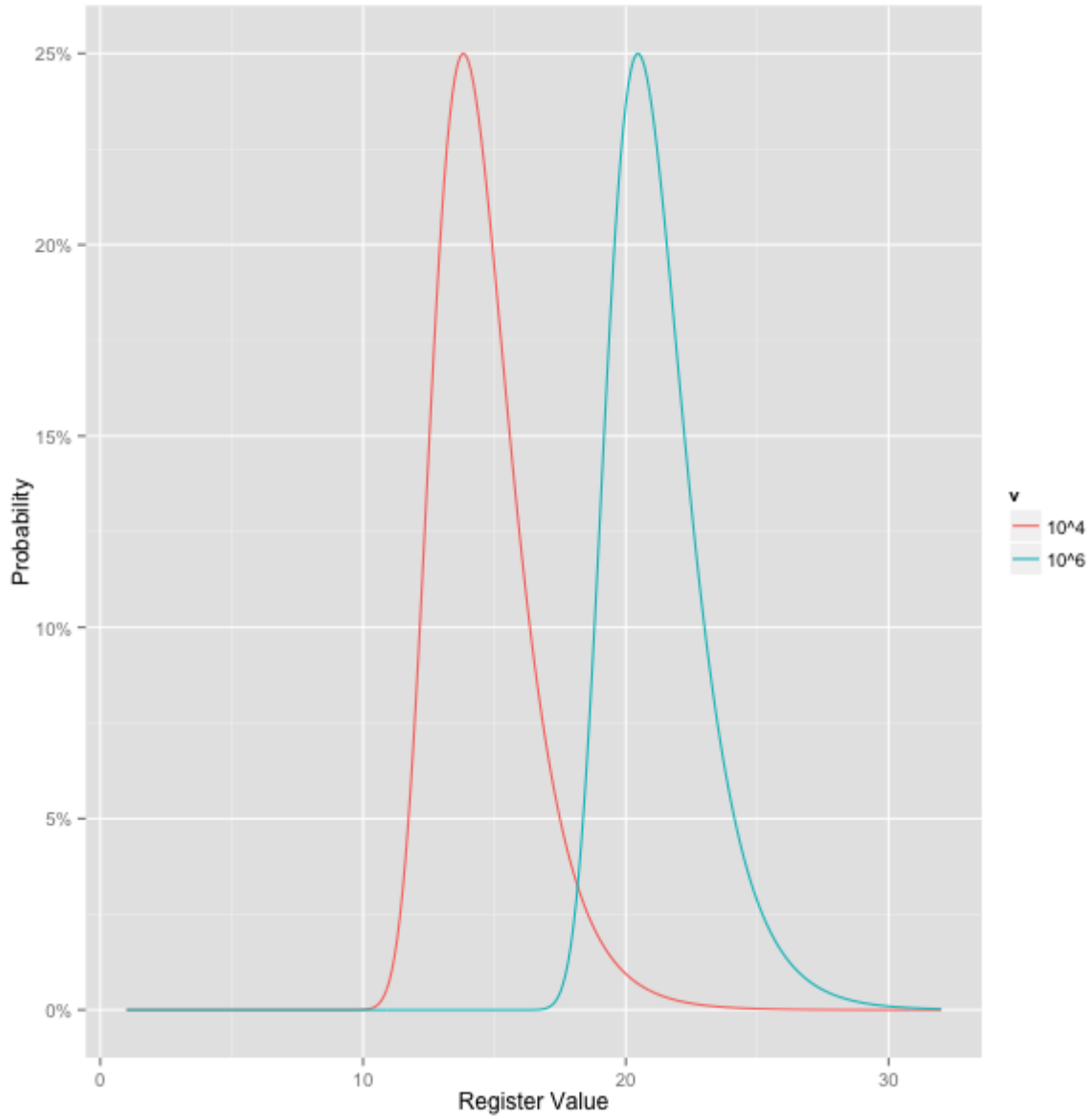


Fig. 1: The theoretical distribution of register values after v distinct values have been run through an HLL.

Throw all these pieces together and you get the HyperLogLog DV estimator:

$$DV_{HLL} = \text{constant} * m^2 * \left(\sum_{j=1}^{m} 2^{-R_j} \right)^{-1}$$

Here R_j is the longest run of zeroes in the $_i{}^{th}$ bucket.

Putting it All Together

Even with the harmonic mean Flajolet still has to introduce a few "corrections" to the algorithm. When the HLL begins counting, most of the registers are empty and it takes a while to fill them up. In this range he introduces a "small range correction". The other correction is when the HLL gets full. If a lot of distinct values have been run through an HLL the odds of collisions in your hash space increases. To correct for hash collisions Flajolet introduces the "large range collection". The final algorithm looks like (it might be easier for some of you to just look at the source in the JavaScript HLL simulation):

```
m = 2^b #with b in [4...16]
```

```
if m == 16:
    alpha = 0.673
elif m == 32:
    alpha = 0.697
elif m == 64:
    alpha = 0.709
else:
    alpha = 0.7213/(1 + 1.079/m)

registers = [0]*m # initialize m registers to 0

########################################################################################
#######
# Construct the HLL structure
for h in hashed(data):
    register_index = 1 + get_register_index( h,b ) # binary address of the rightmost b
bits
    run_length = run_of_zeros( h,b ) # length of the run of zeroes starting at bit b+1
    registers[ register_index ] = max( registers[ register_index ], run_length )

########################################################################################
#######
# Determine the cardinality
DV_est = alpha * m^2 * 1/sum( 2^ -register )  # the DV estimate

if DV_est < 5/2 * m: # small range correction
    V = count_of_zero_registers( registers ) # the number of registers equal to zero
    if V == 0:  # if none of the registers are empty, use the HLL estimate
          DV = DV_est
    else:
          DV = m * log(m/V)  # i.e. balls and bins correction

if DV_est <= ( 1/30 * 2^32 ):  # intermediate range, no correction
     DV = DV_est
if DV_est > ( 1/30 * 2^32 ):  # large range correction
     DV = -2^32 * log( 1 - DV_est/2^32)
```

Rob wrote up an awesome HLL simulation for this post. You can get a real sense of how this thing works by playing around with different values and just watching how it grows over time. Click below to see how this all fits together.

Step Play Reset

Auto Scale Delay: 100 ms

Step Size: 336

Value: 18,863,110 Hash Value: 2,186,419,907

0	1	0	1	0	0	1	0	0	0	0	1	1	1	1	0	1	1	0	0	0	0	1	1

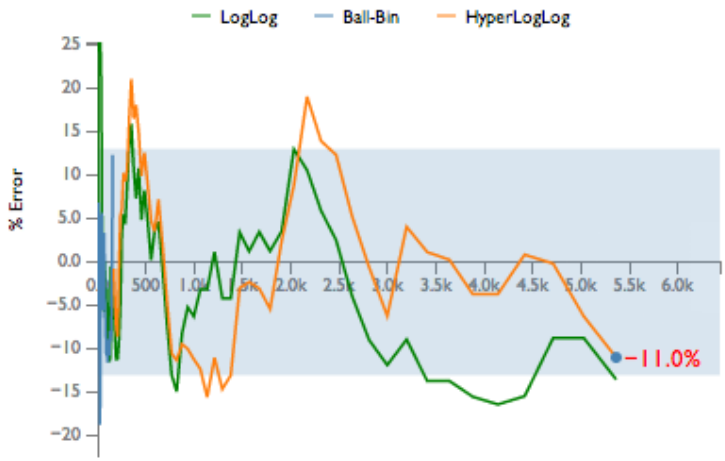
Register Value: 1 Register Index: 3

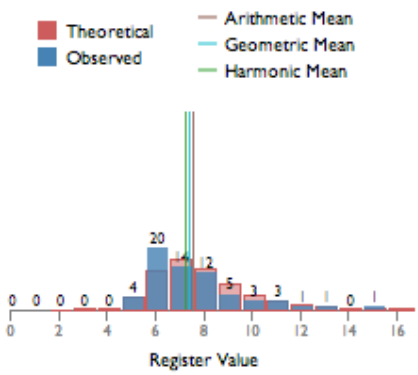
Register Values: $m = 64$

9	6	11	8	6	7	7	7	6	6	8	7	6	6	8	7
5	6	9	6	6	8	6	9	15	8	5	8	8	8	6	6
8	6	12	13	7	11	9	6	11	8	5	6	10	10	7	7
7	8	7	7	7	6	9	7	6	6	7	8	6	6	5	10

Actual Cardinality: 5,362

Algorithm	*Estimated Cardinality*	*% Error*
LogLog	4,638	−13.5
HyperLogLog	4,774	−11.0





Unions

Unions are very straightforward to compute in HLL and, like KMV, are lossless. All you need to do to combine the register values of the 2 (or n) HLL sketches is take the max of the 2 (or n) register values and assign that to the union HLL. With a little thought you should realize that this is the same thing as if you had fed in the union stream to begin with. A nice side effect about lossless unions is that HLL sketches are trivially parallelizable. This is great if, like us, you are trying to digest a firehose of data and need multiple boxes to do summarization. So, you have:

```
for i in range(0, len(R_1)):
     R_new[i] = max( R_1[i], R_2[i] )
```

To combine HLL sketches that have differing sizes read Chris's blog post about it.

Wrapping Up

In our research, and as the literature says, the HyperLogLog algorithm gives you the biggest bang for the buck for DV counting. It has the best accuracy per storage of all the DV counters to date. The biggest drawbacks we have seen are around intersections. Unlike KMV, there is no explicit intersection logic, you have to use the inclusion/exclusion principle and this gets really annoying for anything more than 3 sets. Aside from that, we've been tickled pink using HLL for our production reporting. We have even written a PostgreSQL HLL data type that supports cardinality, union, and intersection. This has enabled all kinds of efficiencies for our analytics teams as the round trips to Hadoop are less and most of the analysis can be done in SQL. We have seen a massive increase in the types of analytics that go on at AK since we have adopted a sketching infrastructure and I don't think I'm crazy saying that many big data platforms will be built this way in the future.

P.S. Sadly, Philippe Flajolet passed away in March 2011. It was actually a very sad day for us at Aggregate Knowledge because we were so deep in our HLL research at the time and would have loved to reach out to him, he seems like he would have been happy to see his theory put to practice. Based on all I've read about him I'm very sorry to have not met him in person. I'm sure his work will live on but we have definitely lost a great mind both in industry and academia. Keep counting Philippe!