HyperLogLog and MinHash

A Union for Intersections

Andrew Pascoe
andrew.pascoe@adroll.com

April 25, 2013

Introduction

At AdRoll, we have a never-ending set of questions related to Big Data. One of these problems is essentially just counting: How many unique items are in a list? How many unique items are shared between lists? On the surface this may sound trivial, but imagine how the problem scales. If I have two lists each with 100 million items, in no particular order, with duplicates, triplicates, etc. how easy is it to determine the size of the corresponding sets, and the size of their intersection?

A now classic approach to these problems is to use Bloom filters, which allow us to test for membership. If an element is not already a member in the Bloom filter, add it, and increment your count by one. Unfortunately, Bloom filters can start to have relatively large sizes when you set a low false positive rate. Say, for example, that we are expecting 100 million insertions and want a 1% false positive rate. Using the formula:

$$m = -\frac{n \log p}{(\log 2)^2}$$

We would need 958505838 bits, or about 114MB, if we use the optimal number of hashes, which is seven. That may not seem awful, but AdRoll has a lot of customers, and we want a lot of such counts for each. It starts to add up.

So we started researching other options.

HyperLogLog++

Our first stop was the Google paper "HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm," by Heule, Nunkesser, and Hall. The authors suggest improvements to the original HyperLogLog algorithm (by Flajolet, et. al.), which is a means of counting uniques in a stream. It's all quite clever, and I'll give a brief overview of how it works. The heavy math behind it can be found in the paper.

The Algorithm

We start by selecting a precision p. Using this, we initialize a byte array M of length 2^p with all zeros. That's the basic setup and data structure behind the system. From there, we take 64 bit hashes of every element coming through in our stream, and divide up the hash like so:

$$\overbrace{101\ldots011}^{p\text{ bits}}\overbrace{001\ldots110}^{64-p\text{ bits}}$$

The first chunk of bits gets interpreted as a particular bin in our byte array. We use the second chunk by counting the number of leading zeros and adding one (so we'd have three in this case). If this number is bigger than the number in the selected bin, we put the number in the bin. That's it. We know that putting in the same element twice isn't going to change anything, because the hash is going to be the same. By the end of the process, we have a byte array full of a bunch of numbers. Ok... cool? Where does that get us?

Well, imagine we just had one bin. Assuming our hash function evenly distributes elements across its range, the maximum number of leading zeros we see after hashing everything tells us a bit about the size of our set. If our answer is 64 leading zeros + 1, it's possible that we got very (un)lucky, and the one element in the set just happened to have gotten hashed to 0... *or* it's also possible that we had a giant set, and we managed to hit an element that got hashed to 0 on the way. (Note that by adding one to the number of leading zeros, if a bin has value zero in it, we know it never got touched.)

By breaking everything up into a variety of bins, we can average out over all the bins to get an estimate of the size of the set. Yes, it is a probabilistic approach. It's

entirely possible that with 65536 bins, we had a set of 65536 elements that managed to each hash in such a way to hit each bin only once and the second chunk of the hash was 0 every time. We'll end up grossly overestimating the size of the set. But this is also extremely unlikely.

So how do we go about averaging this out, anyway? Easy!

$$E = 2^{2p}\left(2^p \int_0^\infty \left(\log_2\left(\frac{2+u}{1+u}\right)\right)^{2^p} \mathrm{d}u\right)^{-1}\left(\sum_{i=0}^{2^p-1} 2^{-M[i]}\right)^{-1}$$

Uh... maybe not. Fortunately, though, everything left of the sum term only depends on p, so we can rely on a table of constants. The Google paper suggests for the second term (because it's convergent):

p	second term
4	0.673
5	0.697
6	0.709
> 7	$0.7213/(1 + 1.079/2^p)$

Obviously, more accuracy can be achieved by evaluating the constants yourself. Anyway, we're still not done. There are some additional improvements to be had.

For one, this raw algorithm is not great at estimating small cardinalities. We can account for this in two ways:

1. We count the number of bins equal to zero (call it V), and if this number is positive, we return: $E = 2^p \log\left(\frac{2^p}{V}\right)$.
2. (If we have $E < 5 \cdot 2^p$, then we estimate the bias for our estimate under the given precision. The Google paper has an addendum of empirically derived values to use for precisions four through eighteen. AdRoll uses these values in practice.

The paper has some additional optimizations that can be made, such as a sparse representation of the byte array, and difference encoding to shrink the size even further. It complicates the code considerably, and they only come in handy for small cardinalities, so we here at AdRoll pass on them.

Unions and Folding

One of the nice things about HyperLogLog and HyperLogLog++ is that unions are lossless. That is, if we have two byte arrays from different sets, we just need to look at each bin and take the maximum value. This allows us to compute the cardinalities of these sets in a distributed fashion, which really speeds up our processing time.

Of course, this only really works if each byte array has the same precision. However, it is possible to take a HyperLogLog++ structure and reduce its precision so it can be unioned with another. Let's take a look at our original hash for an element again:

$$\overbrace{101\ldots011}^{p\text{ bits}}\overbrace{001\ldots110}^{64-p\text{ bits}}$$

If we lowered our precision by one, we'd have:

$$\overbrace{101\ldots01}^{p-1\text{ bits}}\overbrace{1001\ldots110}^{64-(p-1)\text{ bits}}$$

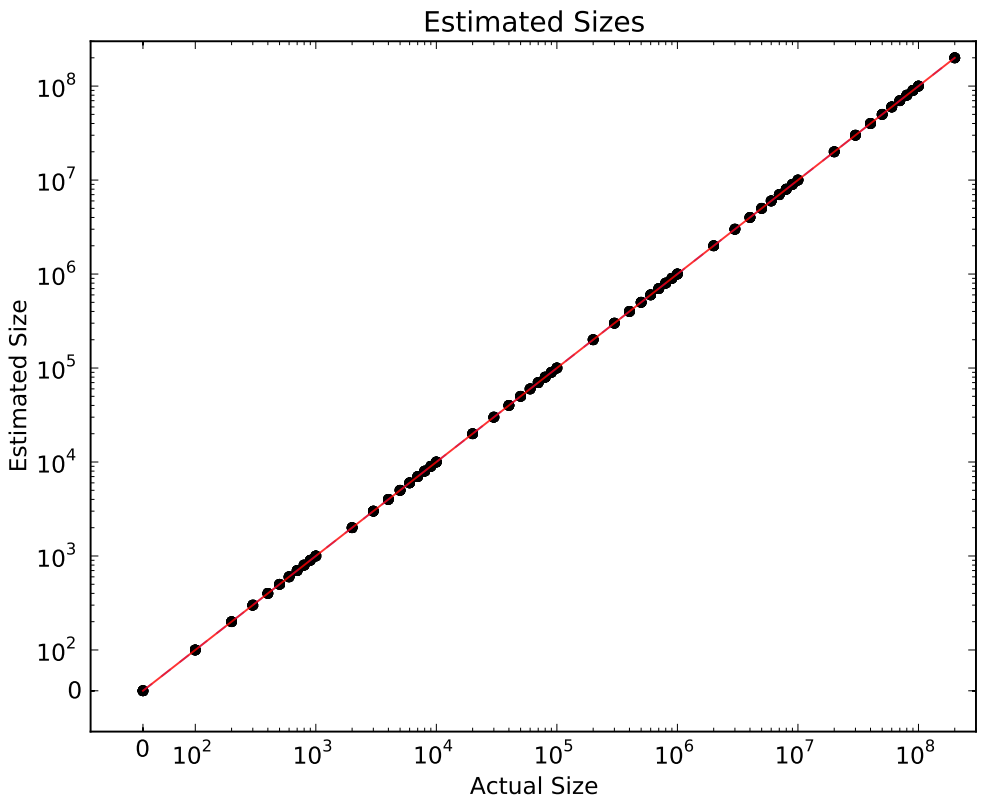
Thus, the contents of bin n in the smaller precision array can only be related to the contents of bins $2n$ and $2n+1$ in the higher precision array. If they're both zero, nothing made it into those bins, so nothing would have made it into bin n. We output zero. If they're not both zero, we just add one to what's in bin $2n$. Note that because $2n+1$ is odd, the number of leading zeros had that element gotten hashed into bin n would be zero, so we would've output one. On the other hand, since $2n$ is even, any element getting hashed into bin n would have picked up an extra leading zero, so we would have to add one to the contents of bin $2n$ anyway.

Using this scheme, we can just recurse down to our necessary lower precision, and take unions at will. Intersections are another matter entirely, and we'll get to that later.

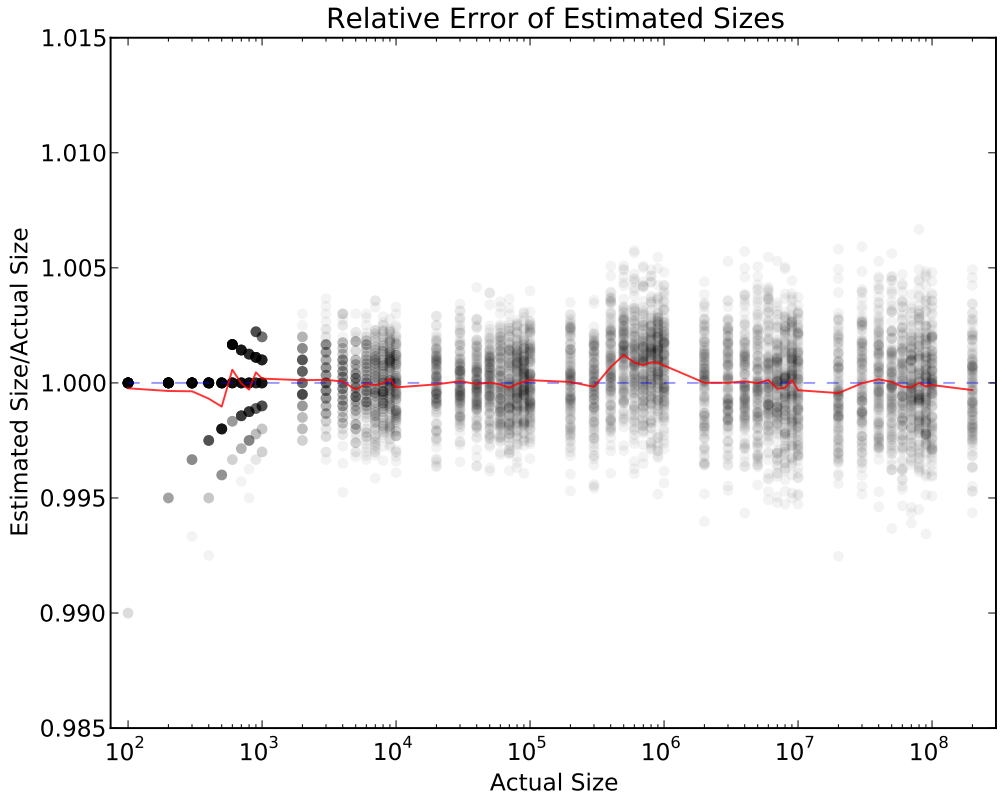
Empirical Performance

There's already enough literature out there about the performance of HyperLogLog and HyperLogLog++, but since I was running MinHash tests anyway (as you can read in the next part of this paper), I decided to evaluate HyperLogLog++ as well.

For sizes in 0, 100, 200, ..., 1000, 2000, ..., 10000, 20000, ...100 million, and 200 million, I ran HyperLogLog++ 128 times for each with $p = 18$.



Note the logarithmic scales on both axes. This is a very encouraging graph, of course, but it also makes sense to look at the relative errors too:



I added an alpha channel to the points so it's easier to see the distribution of the errors. As you can see, from small sets to very large ones, we can expect to be within 1% of the actual size of the set. Our averages work out well too. Appendix B has graphs with 3σ ranges for $p = 4, \ldots, 18$.

MinHash

One of the drawbacks of HyperLogLog implementations is that we can't really get intersections. We'll never get this to happen because elements get lost in the fray, masked by elements that hashed to values with more leading zeros. But what we *can* do is get an estimate of the size of arbitrary intersections by tacking on an additional structure: MinHash.

The Algorithm

First let's talk about what MinHash does. It computes an estimate of something called the Jaccard Index, which is defined as:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Intuitively, the Jaccard Index is a measure of how similar two sets are, and generally we see MinHash used to compare two sets. However, we can break this interpretation and extend the notion:

$$J(A_1, A_2, \ldots, A_n) = \frac{|\bigcap_{i=1}^{n} A_i|}{|\bigcup_{i=1}^{n} A_i|}$$

(Sorry, CS folks: I'm a math guy, and starting our indices with one is the preference.) Now to see how MinHash works, we're going to start with the idea of a simple membership table:

$a \in \bigcup A_i$	A_1	A_2	$\ldots$	A_n
1	1	0	$\ldots$	0
2	0	1	$\ldots$	1
3	1	1	$\ldots$	1
4	1	1	$\ldots$	0
$\vdots$	$\vdots$	$\vdots$	$\ldots$	$\vdots$
N	1	1	$\ldots$	1

The rows we care about are where all of the entries are one, so that $a \in \bigcap A_i$. The number of such rows is, of course, $x = |\bigcap A_i|$. Now let's think about permutations. We know that there are $N!$ permutations of all of the rows. The number of permutations that start with any given row is $(N-1)!$. Then, the proportion of the number of permutations that start with all ones on the first row is:

$$\frac{x(N-1)!}{N!} = \frac{x}{N} = \frac{|\bigcap A_i|}{|\bigcup A_i|} = J(A_1, A_2, \ldots, A_n)$$

That means that if we permute the rows, the probability that the first row is all ones is equal to our extended Jaccard Index. Of course, we can't examine all of these

permutations, and even computing a single permutation would be prohibitive for large N. Instead, we can rely on a set of hash functions, where each hash function serves as a substitute for a permutation. We just need good hash functions that don't collide, and have nice, even distributions.

Suppose we have k separate hash functions. We hash every element of every set as it comes through, and we keep track of the minimum hash for each hash function. We let y be our tally of when all of our minima are equal:

$$y = \sum_{i=1}^{k} \mathbf{1}_{\{\text{True}\}}(\min\{h_i(A_1)\} = \min\{h_i(A_2)\} = \ldots = \min\{h_i(A_n)\})$$

That's kind of a weird indicator function up there, but you get the idea. Anyway, from here we can say that:

$$J(A_1, A_2, \ldots, A_n) \approx \frac{y}{k}$$

This is one variant of MinHash. Another variant attempts to minimize the number of hash functions we have to compute. Instead of having k hash functions, we keep k of the minimum values for each set, and only have one hash function. Essentially, we work off of one permutation of the membership table, and go down through k rows. This is like getting a random sample of $\bigcup A_i$. Then we count the number of rows that are all ones. Of course, we can do this without actually building a table. From a set perspective, we have:

$$J(A_1, A_2, \ldots, A_n) \approx \frac{y}{k} = \frac{|\min_k \{\bigcup \min_k\{h(A_i)\}\} \cap (\bigcap \min_k\{h(A_i)\})|}{k}$$

Perhaps that looks like a nightmare, but it's not so bad. From all of the sets, get all their minima, and look at the bottom k. Count how many of these bottom k are in all the individual lists for each set. Divide by k, and we're done.

As we said, we can't really get intersections out of HyperLogLog, but we can get an estimate for the sizes of intersections. That's because we know that:

$$\left|\bigcap A_i\right| = J(A_1, A_2, \ldots, A_n) \cdot \left|\bigcup A_i\right| \approx \text{MinHash Result} \cdot \text{HyperLogLog Result}$$

Hey, we were hashing stuff anyway when doing the HyperLogLog algorithm. We might as well keep track of our k lowest hashes while we're at it, and we can get some intersection estimates to boot. Note that this is all still easily distributable. When we go to union two HyperLogLogs, we can also union the MinHash portions of each object too, because:

$$\min_k \{h(A \cup B)\} = \min_k \left\{ \min_k \{h(A)\} \cup \min_k \{h(B)\} \right\}$$

But how good is all this, and how should we pick our k? Let's assume that the hash function works well as a permutation, and we don't have to worry about collisions. Then the probability mass function for getting r rows of all ones is a binomial distribution. (Actually, it's hypergeometric, but we will assume that $k \ll N$.) We would have:

$$\mathbf{P}\left(y; k, J(A_1, \ldots, A_n)\right) = \binom{k}{y} J(A_1, \ldots, A_n)^y (1 - J(A_1, \ldots, A_n))^{k-y}$$

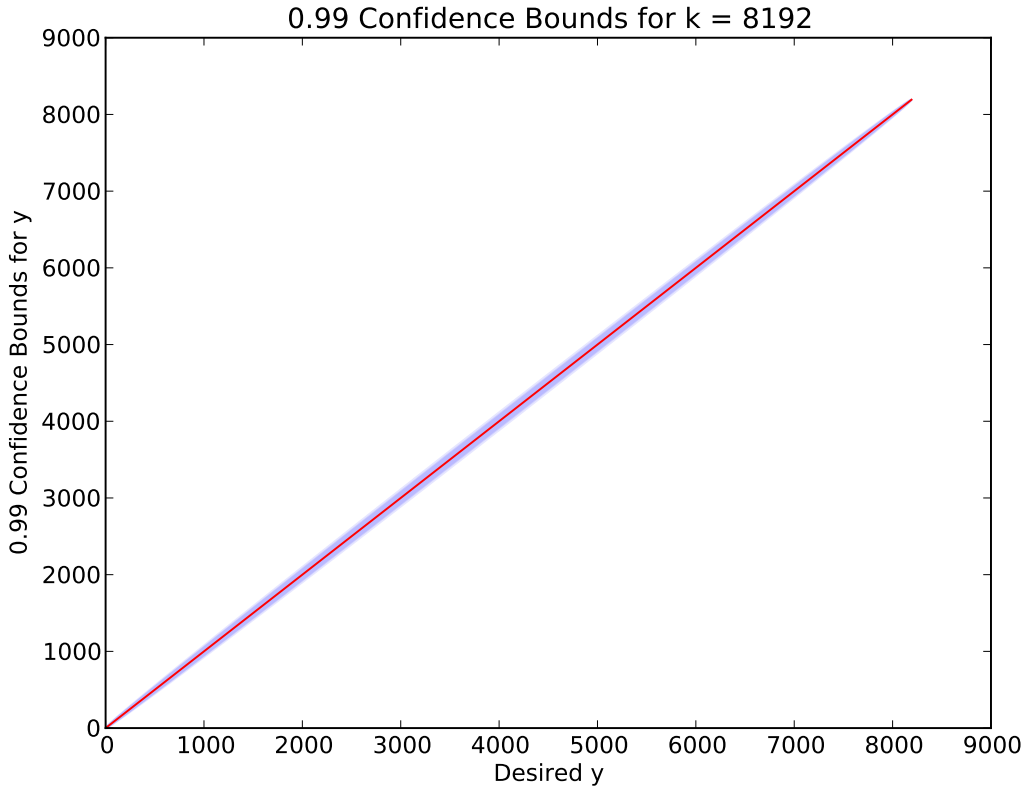
Now, obviously the larger k is, the more resolution we can gain into the extended Jaccard Index. But by using the cumulative distribution function:

$$F(x; k, J(A_1, \ldots, A_n)) = \sum_{m=0}^{\lfloor x \rfloor} \binom{k}{m} J(A_1, \ldots, A_n)^m (1 - J(A_1, \ldots, A_n))^{k-m}$$

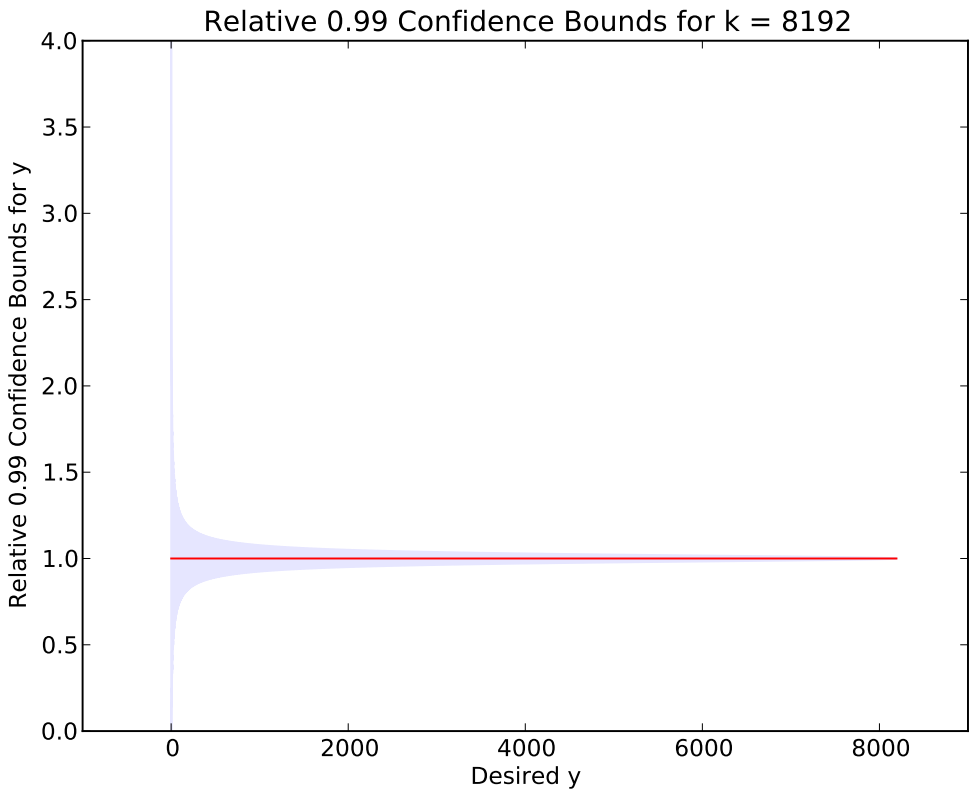
We can use the inverse to find confidence intervals. If we set our desired confidence to $1 - \delta$, then we would have our confidence interval be:

$$\left[F^{-1}\left(\frac{\delta}{2}; k, J(A_1, \ldots, A_n)\right),\ F^{-1}\left(1 - \frac{\delta}{2}; k, J(A_1, \ldots, A_n)\right) \right]$$

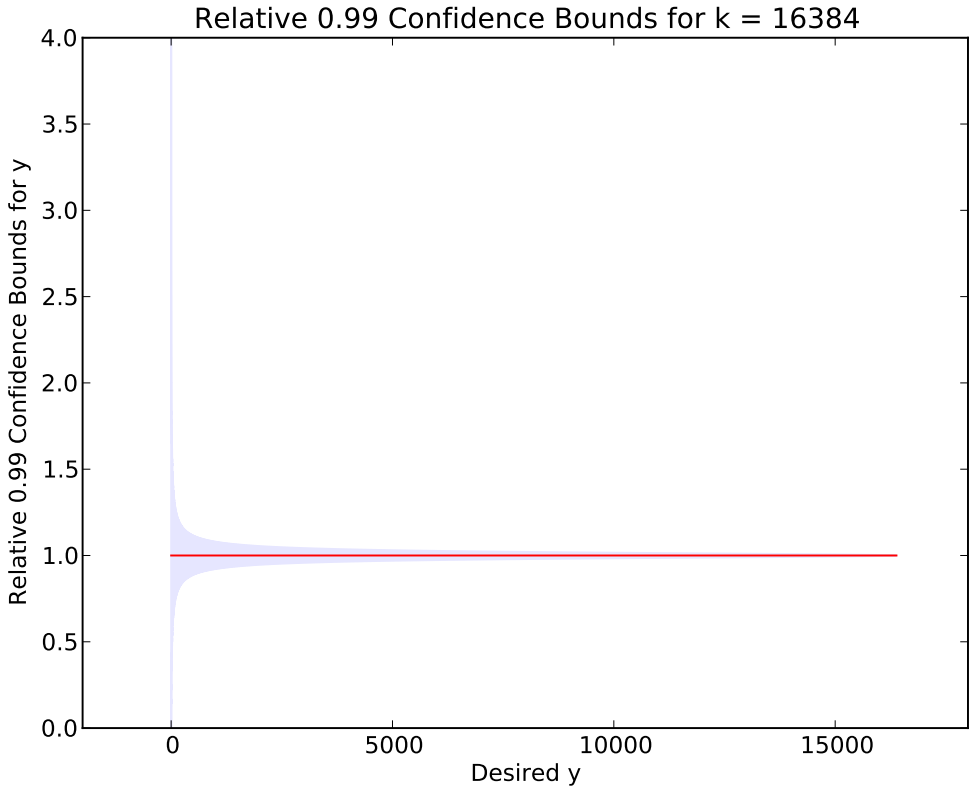
Here's a graph of these bounds for $k = 8192$. I've sampled at extended Jaccard Indices with k in the denominator, so it's easy to visualize the y we're looking for:



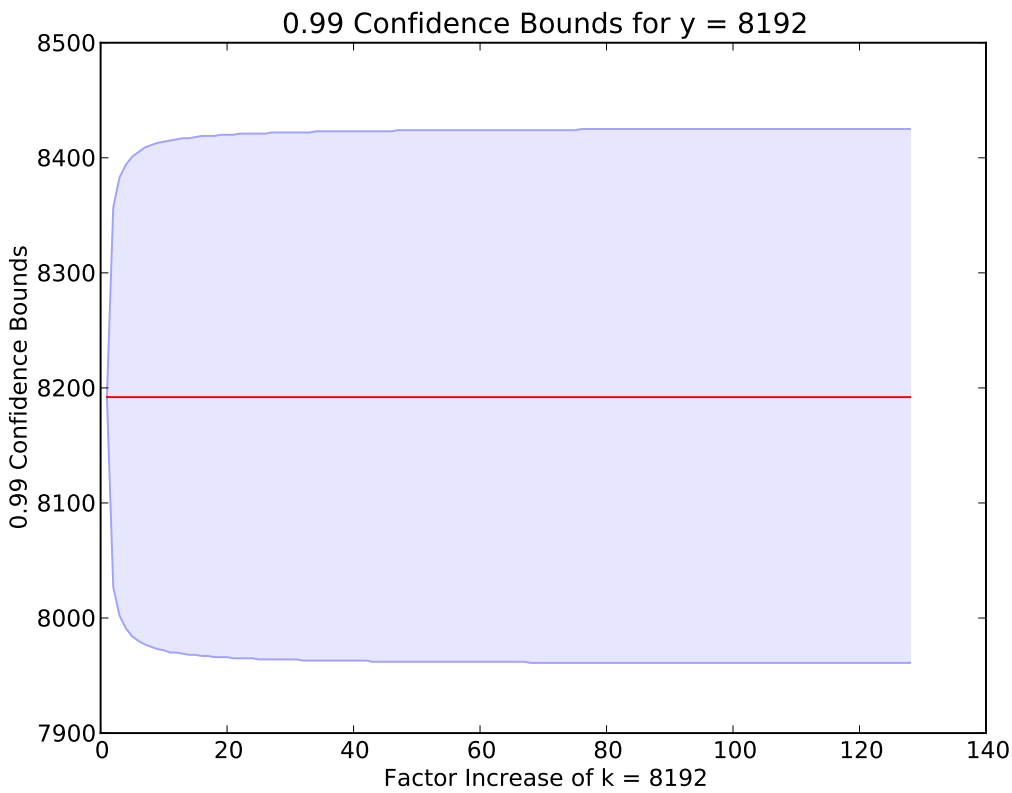
We can see that even at 99% confidence, we are going to estimate the extended Jaccard Index quite closely. However, our relative error can be quite high for low values of y, as the following graph demonstrates:



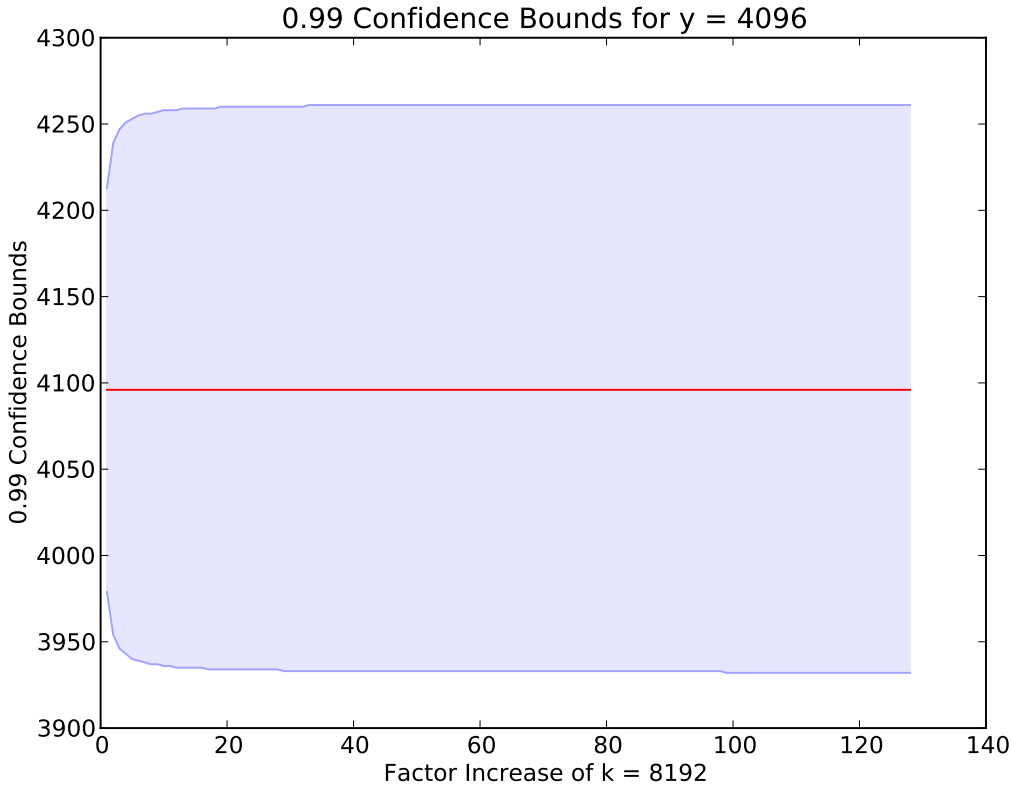
If we double k to 16384, we get:



The take away is that increasing k benefits measurements of low extended Jaccard Indices the most. This is because the confidence interval bounds change very slowly with increasing k for most y. For example, this is the change of bounds for $y = 8192$, which has the most accuracy to lose:



And for $y = 4096$:



But since we're keeping y fixed and increasing k, the corresponding extended Jaccard Index is shrinking linearly. Therefore, if we want to select a k for a given accuracy, we can first determine the lowest extended Jaccard Index we want to measure accurately, and specify how accurate we want to be with certain confidence. Since we know that the confidence bounds don't change too much, we can work off of a table for $k = 8192$:

Expected Relative Error Bounds
for $k = 8192$

$J(A_1, \ldots, A_n)$	$Confidence$		
–	0.90	0.95	0.99
0.001	0.5869	0.7090	0.9531
0.01	0.1841	0.2207	0.2939
0.05	0.0791	0.0962	0.1255
0.1	0.0547	0.0657	0.0864

Thus, if we wanted our error bounds for an extended Jaccard Index of 0.0005 to be within 0.9231 with 0.99 confidence, we could take $k = 16384$. Of course, without the use of a table, we would calculate for relative error α:

$$k = \inf\left\{x : \ \sup\left\{s \geq x : \ \left|\frac{F^{-1}\left(1 - \frac{\delta}{2}; s, J(A_1, \ldots, A_n)\right)}{J(A_1, \ldots, A_n)s} - 1\right|\right\} < \alpha\right\}$$

Appendix A has some Python code using the scipy package that calculates this k. There are also some additional tables attached, which are more practical from an accuracy standpoint. As stated, one of AdRoll's concerns is space requirements, and so for us, it makes sense to determine our accuracy in terms of storage.

It is also possible to use Chernoff bounds and the Sampling Theorem to set k (but I still prefer the above method, as it's significantly tighter than the Chernoff bounds):

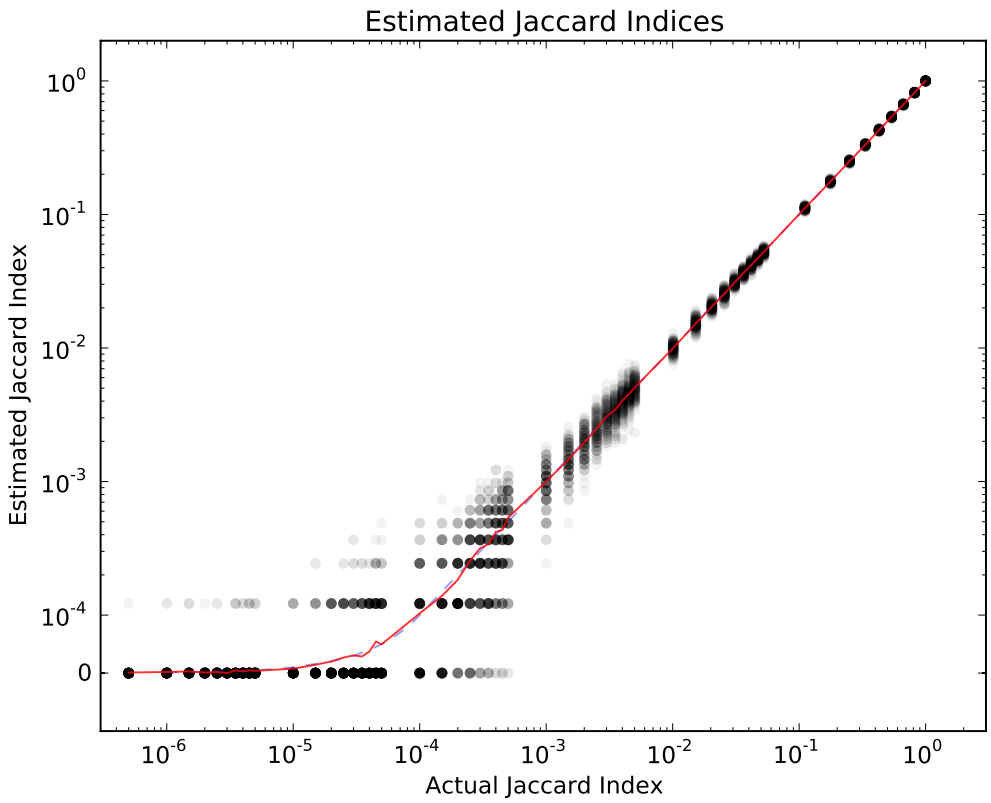
$$k \geq \frac{2 + \epsilon}{\epsilon^2} \log \frac{2}{\delta}$$

Empirical Performance

First, we're going to start by considering $k = 8192$. This value may have seemed very arbitrary before, but it's a little less arbitrary than it looks: storage-wise, it takes up as much space as 25% of a corresponding HyperLogLog++ structure at $p = 18$.

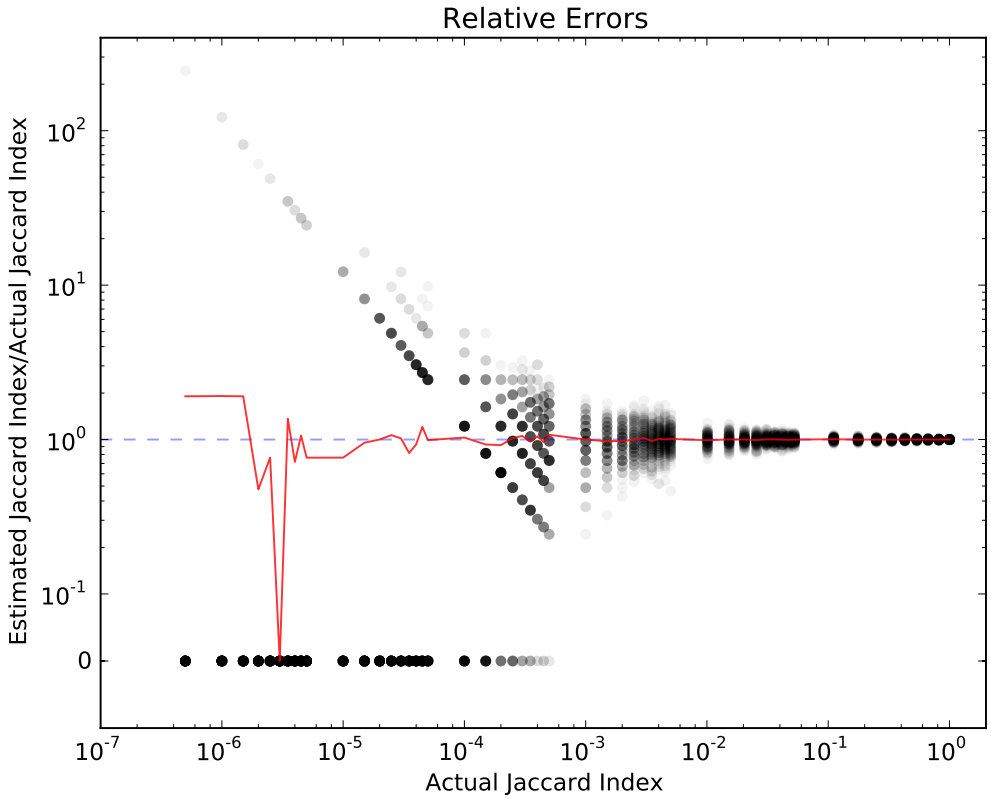
We took two sets with 100 million unique elements in each, and then started intersecting them, with intersection cardinalities of 100, 200, ..., 1000, 2000, ...10000, 20000, ...100 million. This doesn't make for particularly clean Jaccard Indices, but that's ok. We estimated their cardinalities by multiplying the estimated Jaccard Indices by the HyperLogLog++ results. That means that any errors you see here are from a practical evaluation. We did this 128 times for each.

In this first graph, we see how our estimates worked out:



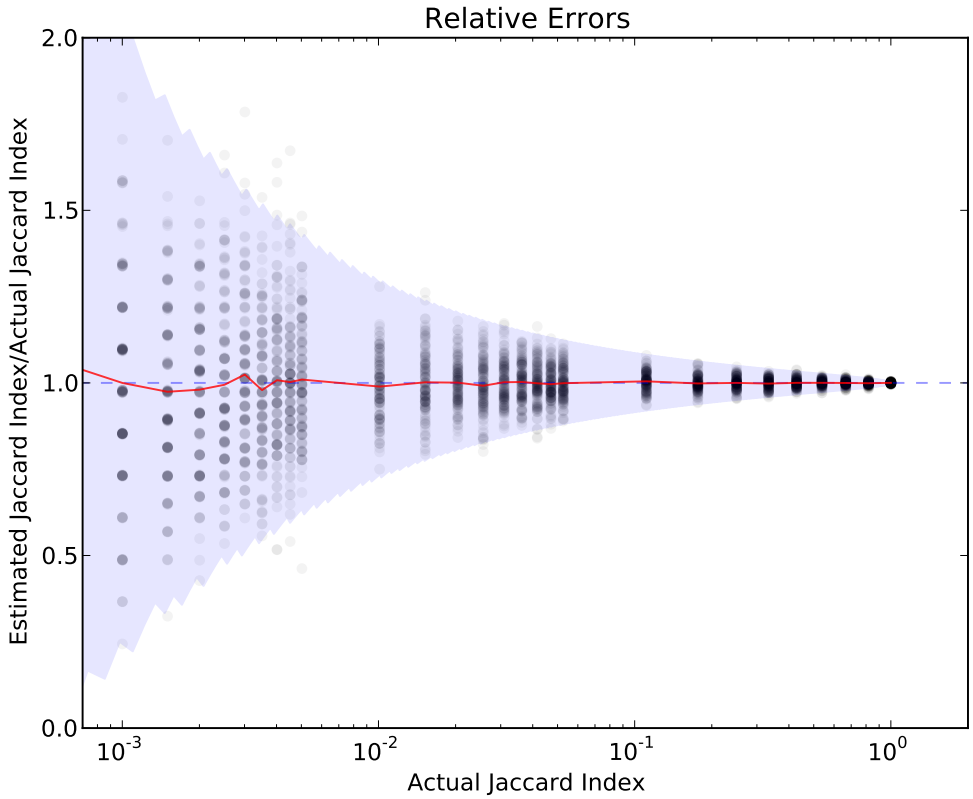
Note the log scales on both axes. On average, we estimate the Jaccard Indices pretty well. There's a blue dotted line in there which represents the ideal line, but it's pretty hard to see. I applied an alpha channel to the individual points, so you can see that for very low Jaccard Indices, we frequently output no intersection at all.

I tend to like the next graph a bit better, because it shows the relative errors, and shows the pitfalls of low Jaccard Indices.

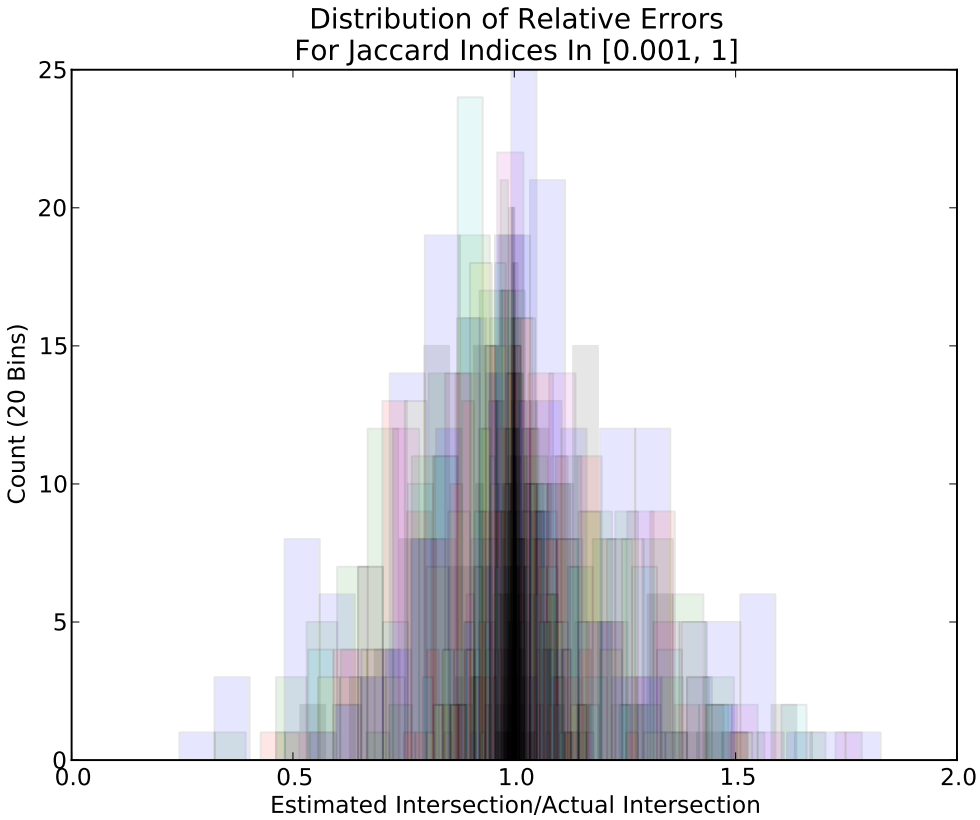


If we do get some chatter, we overstimate the size quite a bit: about 250 times what it should be in the worst case here. For a Jaccard Index of about $4 \cdot 10^{-7}$, though, I'm not too concerned if we report it as $1 \cdot 10^{-4}$ only 1% of the time.

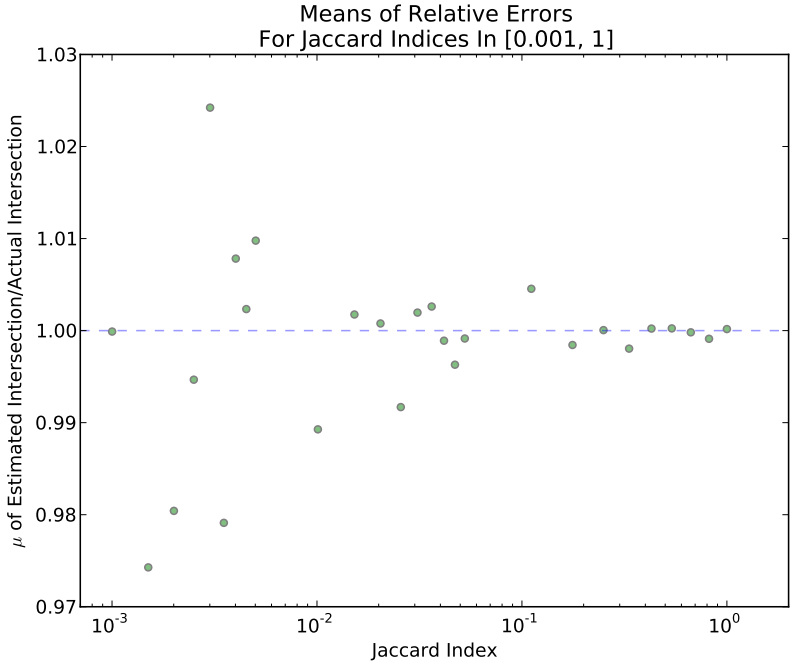
Overall, though, it looks like our errors are more predictable as soon as we hit a Jaccard Index of 0.001. To show this, here's a graph in which the log scale has been removed from the y-axis, and we only consider Jaccard Indices above this threshold:

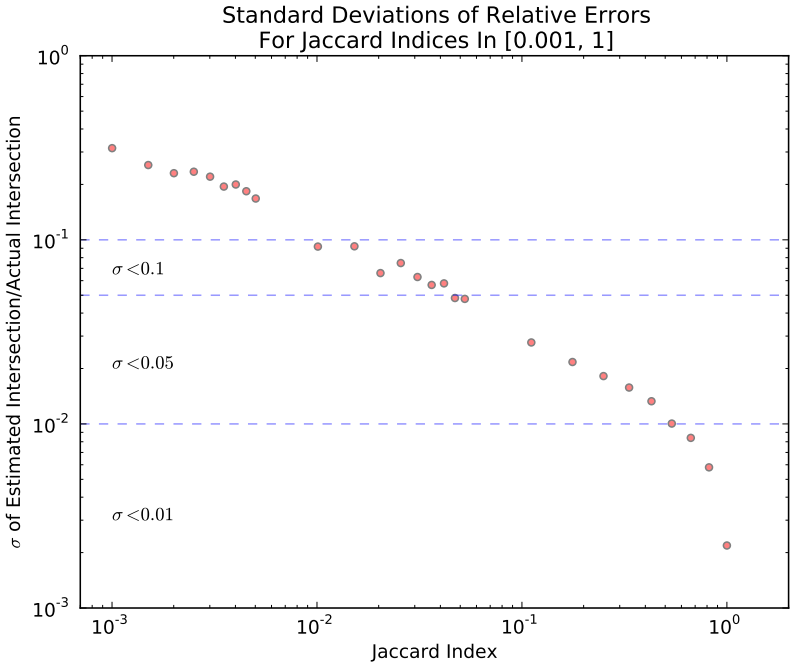


The blue region is our theoretical relative error at 99% confidence and $k = 8192$, so this experiment lines up very well with the theory. At this point, the errors look essentially normally distributed. We expect this to a certain extent, because we know that the binomial distribution can be effectively modeled by the normal distribution under certain conditions. But just as a sanity check (and also because I like this graph), here are the distributions of the errors for each Jaccard Index plotted over each other:



Here are the means and standard deviations:

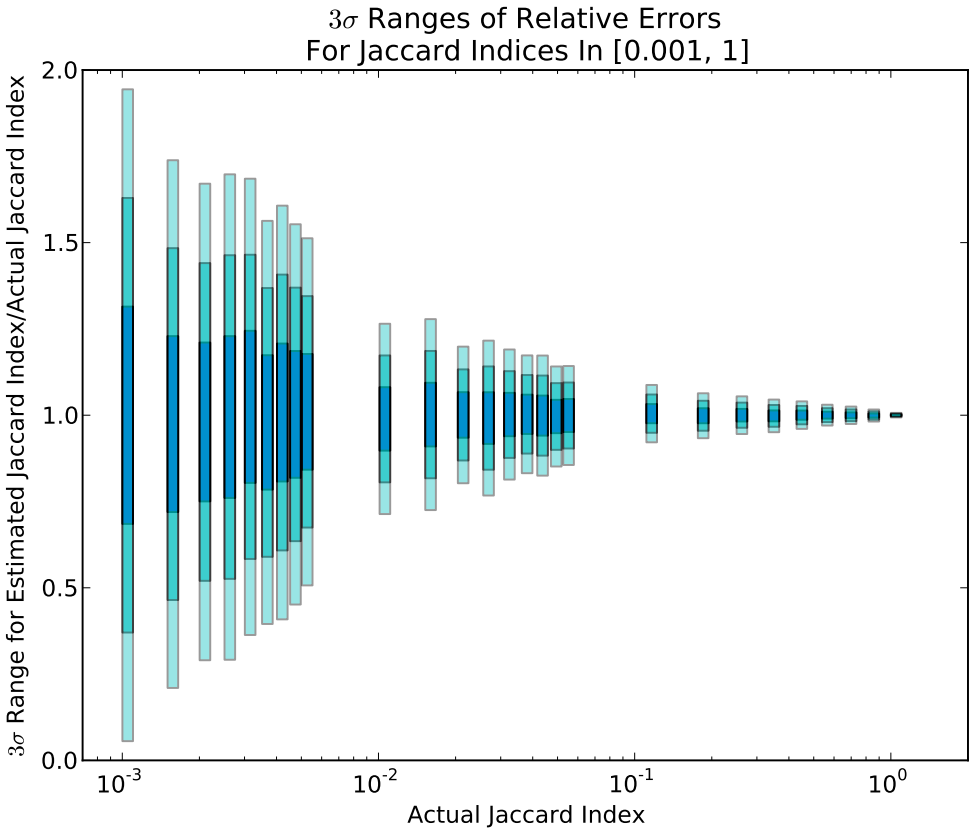




Note that we have a standard deviation at a Jaccard Index of one because Hyper-

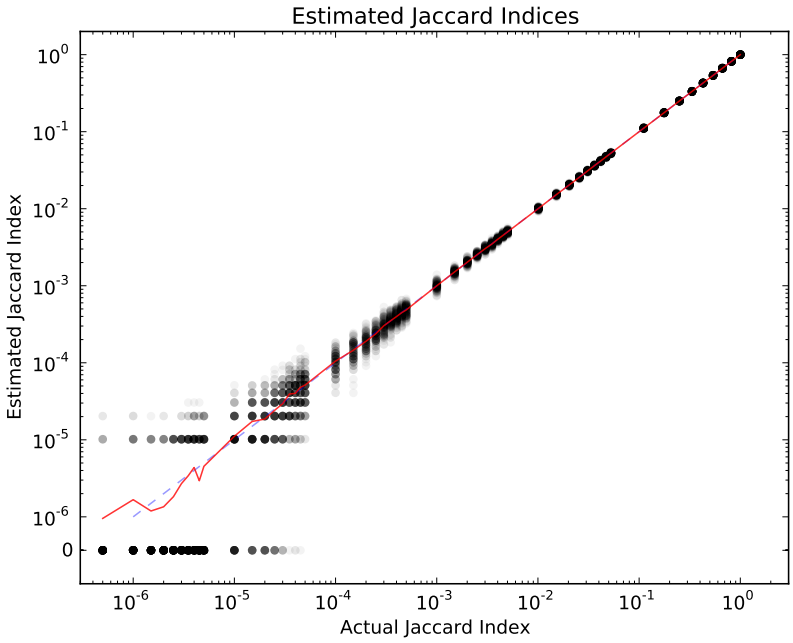
LogLog++ contributes some noise.

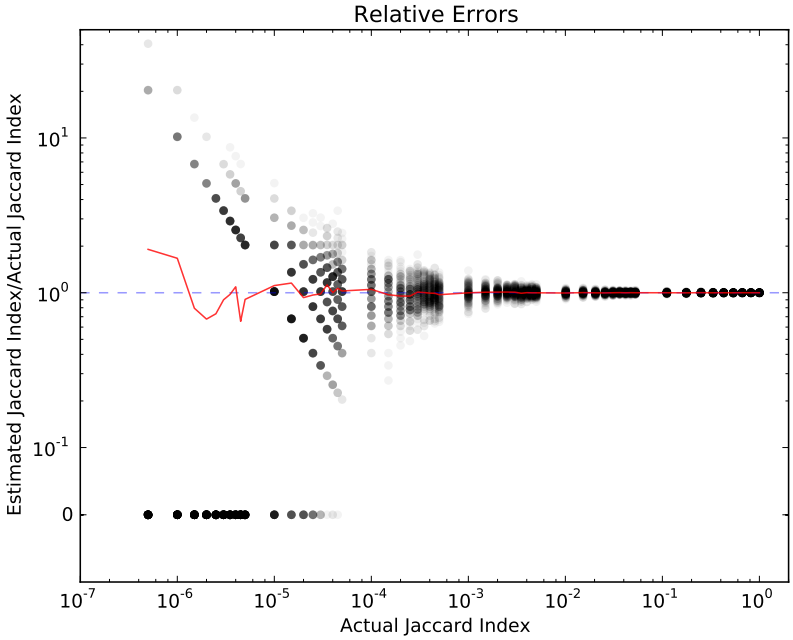
Combining the two of these graphs, we can very quickly glance and see how confident we can be in our predictions:



For our purposes, it is generally not so bad to report an 0.1% overlap as either 0% or 0.2%.

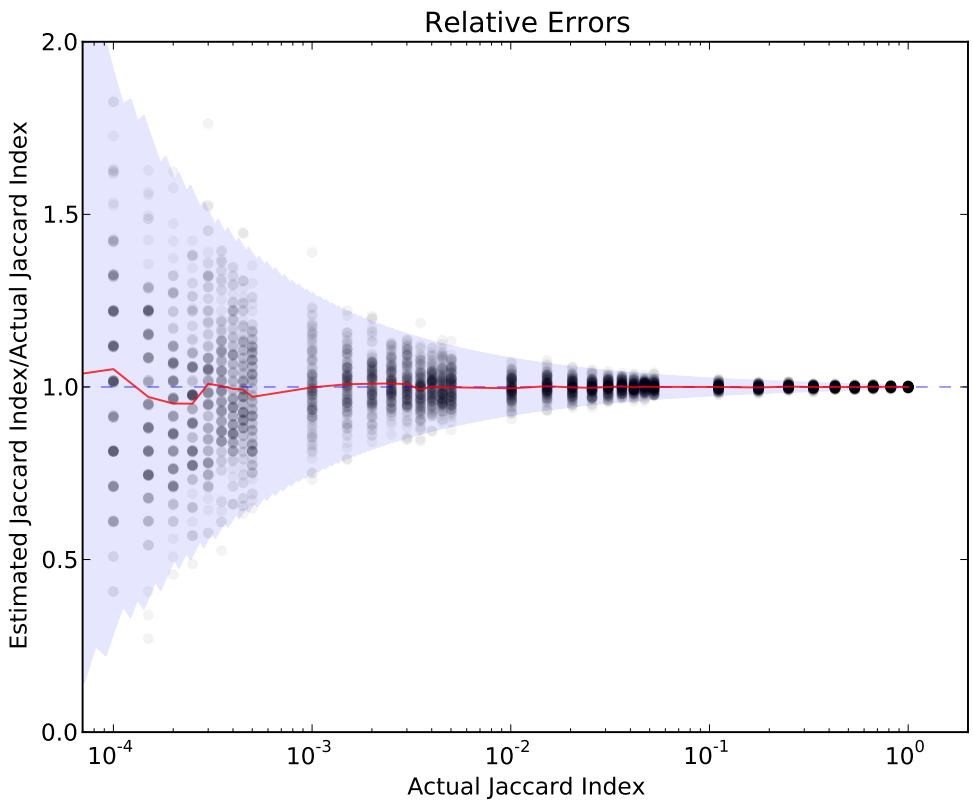
Valentino Volonghi, our Chief Architect, asked me to evaluate a data structure that would be 1MB in size. Given $p = 18$, that leaves us with 768KB left, or 98304 64-bit numbers. So with the same experimental setup, I set $k = 98304$, and here are the results.



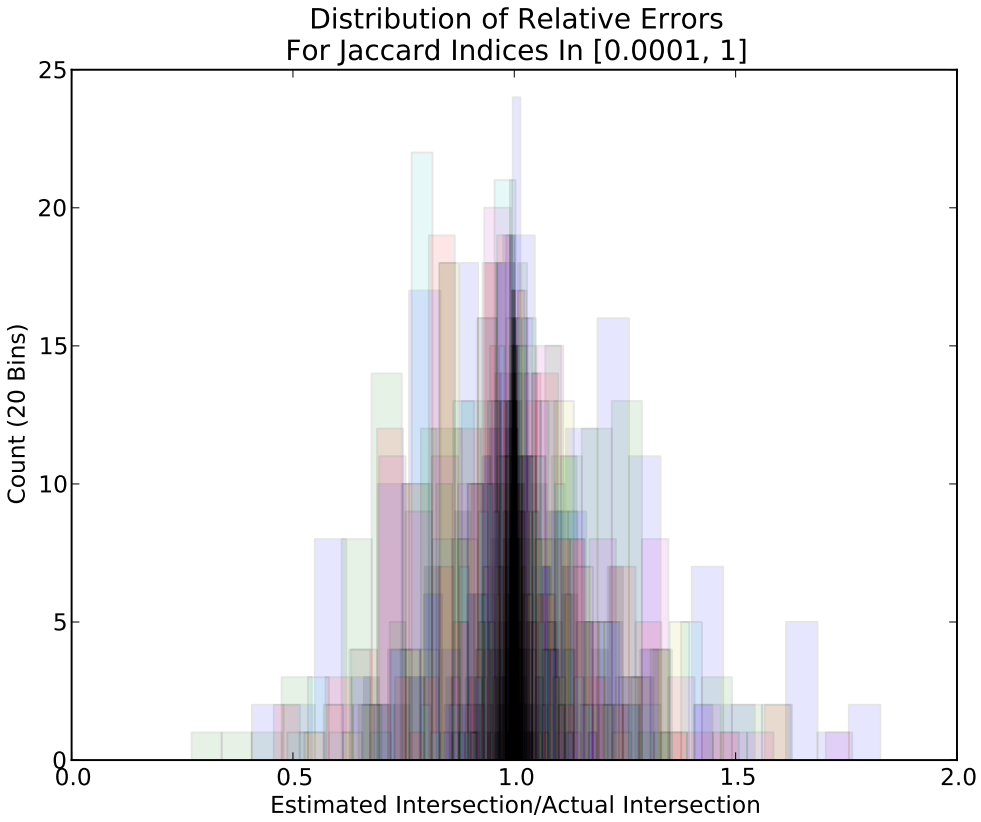


As you can see, our relative errors have dropped pretty dramatically. In addition,

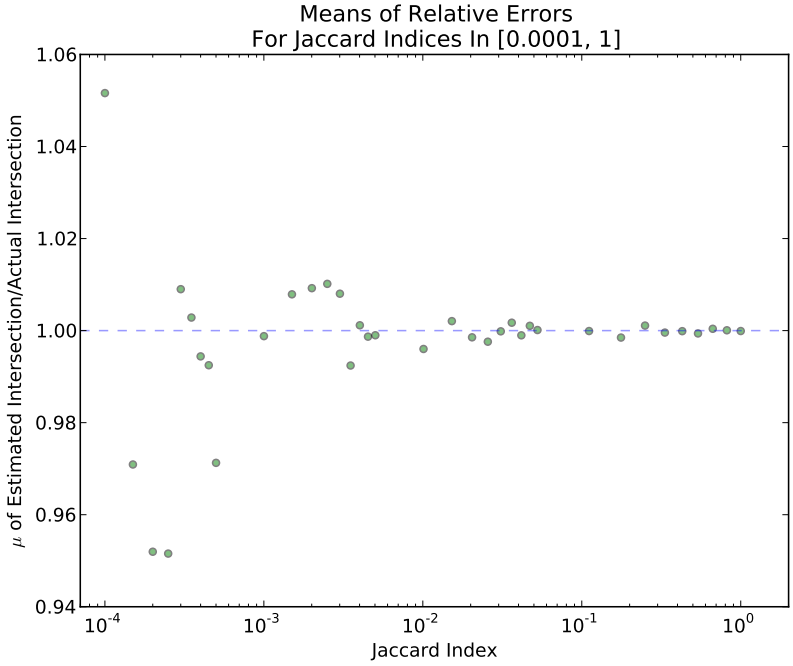
it looks like our relative errors are more or less normally distributed starting at a Jaccard Index of 0.0001, gaining an order of magnitude over $k = 8192$. Removing the log scale on the y-axis:

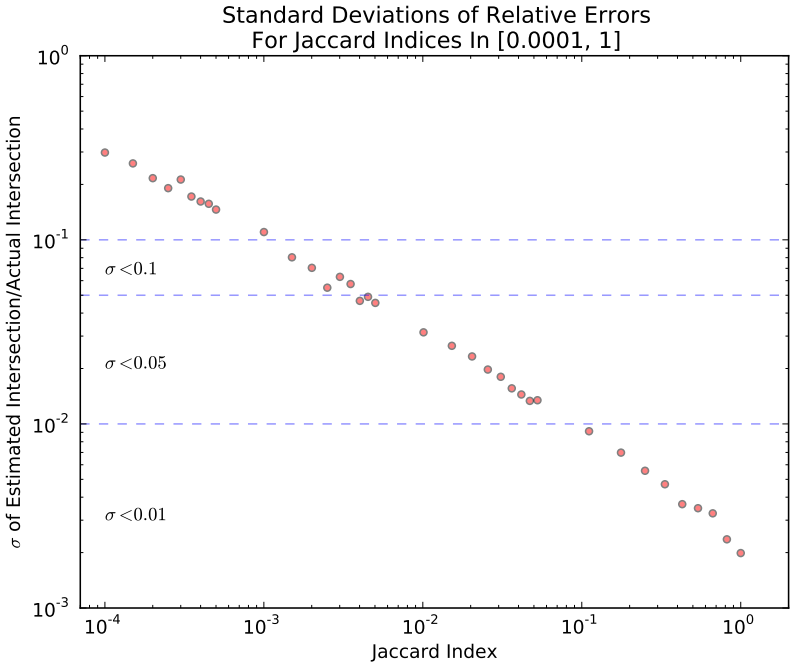


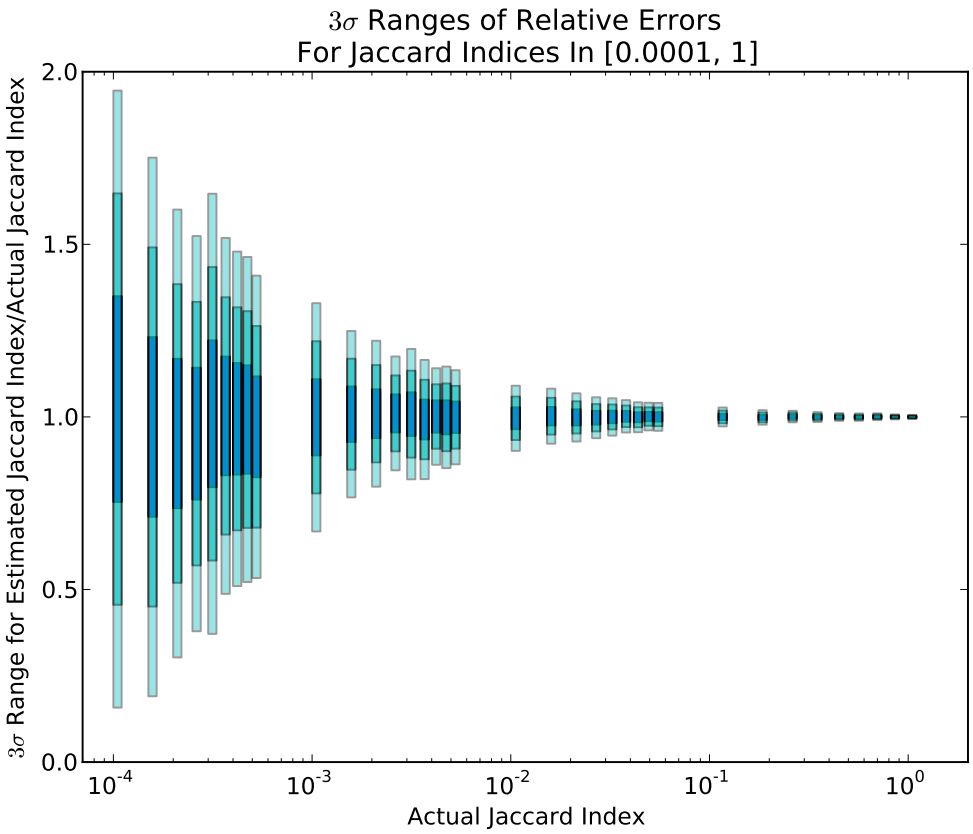
As expected, we line up well again with our theoretical bounds, calculated with a confidence of 99% and $k = 98304$. We can see the large gains made on the Jaccard Index of 0.001 from the $k = 8192$ scenario. Now it looks much more likely that instead of swinging from 0% to 0.2%, it's more likely to swing between 0.075% to 0.125%. And here's that pretty graph again, because I like it:



And now we'll finish up with means, standard deviations, and ranges:


Means of Relative Errors
For Jaccard Indices In [0.0001, 1]
μ of Estimated Intersection/Actual Intersection
1.06
1.04
1.02
1.00
0.98
0.96
0.94
10^-4
10^-3
10^-2
10^-1
10^0
Jaccard Index


Standard Deviations of Relative Errors
For Jaccard Indices In [0.0001, 1]
σ of Estimated Intersection/Actual Intersection
10^0
10^-1
10^-2
10^-3
σ <0.1
σ <0.05
σ <0.01
10^-4
10^-3
10^-2
10^-1
10^0
Jaccard Index



Conclusions

If you're already using HyperLogLog structures in your work, but have been longing for a means of intersecting them, it's worth checking out out extending your code with a MinHash implementation. While it does require extra processing power to deal with collecting all the minima, it's possible to get satisfactory performance out of the structure for a relatively low storage or memory footprint.

Last, as a final plug, I get to tackle interesting problems like this all the time throughout the course of my work. If this type of stuff fires you up, AdRoll is hiring, and we would love to hear from you.

Appendix A

Here's some Python code that uses the scipy package to find appropriate values of k for MinHash, given $J(A_1, \ldots, A_n)$, α (the relative error), and a level of confidence. It can take a while to run for high precisions and low Jaccard Indices.

```
from scipy.stats import binom

def err(ci, k, j):
    n = binom.ppf(ci, k, j)
    if n == 0:
        #this is an edge case, so we report a big error
        return 1e9
    else:
        return abs(n/(j*k) - 1)

def find_k(j, alpha, conf, k=1, maxk=1000000):
    ci = 1 - ((1 - conf)/2.0)
    e = err(ci, maxk, j)
    if e > alpha:
        #we'll never get the precision we want for this
        #range, so output the end of the range
        return (maxk, e)
    #grab bounds for search space
    kn = find_bound(j, alpha, ci, k, maxk)
    ub = find_bound(j, 0.75*alpha, ci, k, maxk)
    #start searching...
    while True:
        if kn == maxk:
            break
        broken = False
        if err(ci, kn, j) <= alpha:
            for n in range(kn, min(2*kn, ub) + 1):
                if err(ci, n, j) > alpha:
                    kn = n + 1
                    broken = True
                    break
            if not broken:
```

```
                return (kn, err(ci, kn, j))
        else:
            kn += 1
    return (maxk, err(ci, maxk, j))

def find_bound(j, alpha, ci, k, maxk):
    #just a binary search to find good a good bound
    minb = k
    maxb = maxk
    e = err(ci, maxk, j)
    while True:
        midb = int((maxb + minb)/2)
        if midb - minb < 1:
            break
        midv = err(ci, midb, j)
        if midv <= alpha:
            maxb = midb
        else:
            minb = midb
    return midb
```

Here is some sample output from the program:

0.99 Confidence

	α			
$J(A_1, \ldots, A_n)$	1.0	0.5	0.25	0.1
0.0001	90000	–	–	–
0.001	9000	31334	116800	–
0.01	900	3134	11520	68455
0.05	170	587	2208	13128
0.1	75	280	1040	6210

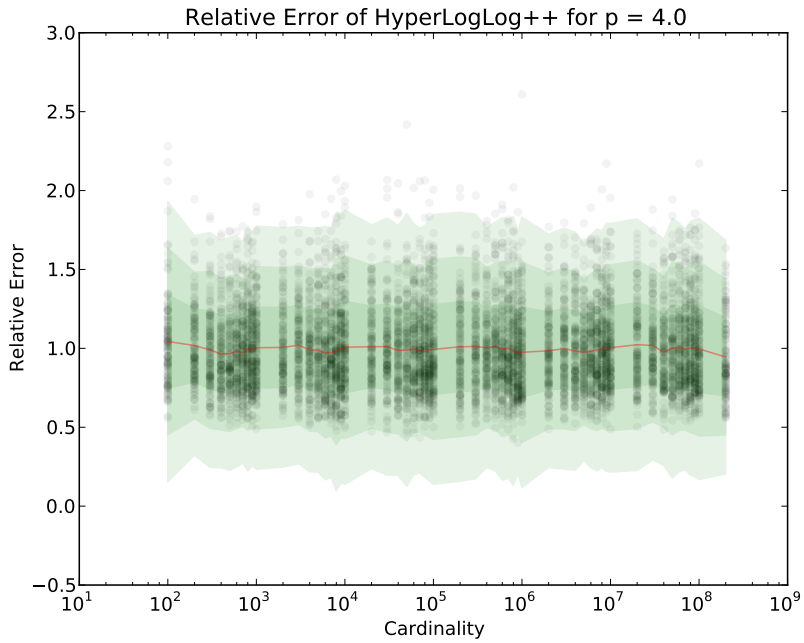
0.95 Confidence

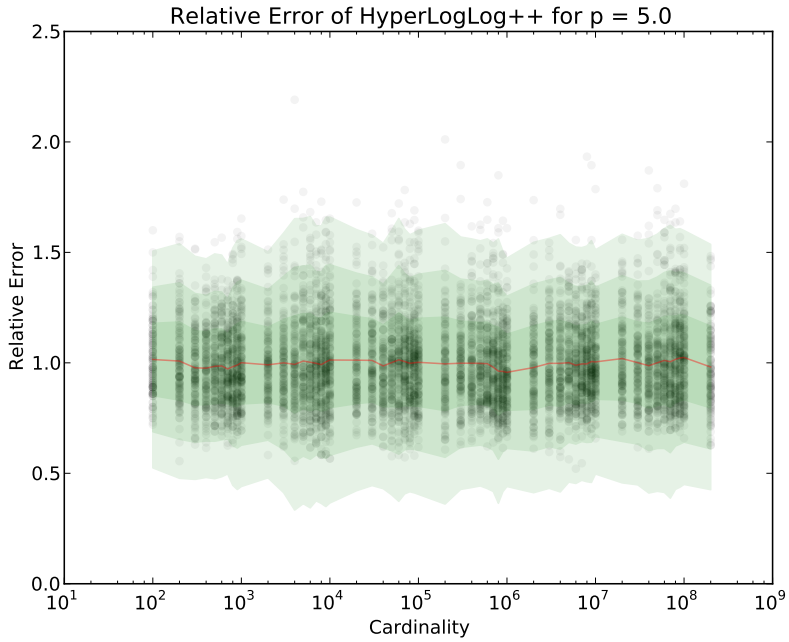
	α			
$J(A_1, \ldots, A_n)$	1.0	0.5	0.25	0.1
0.0001	55000	186667	–	–
0.001	5500	18667	68800	402728
0.01	500	1867	6800	39910
0.05	100	347	1296	7637
0.1	45	167	608	3619

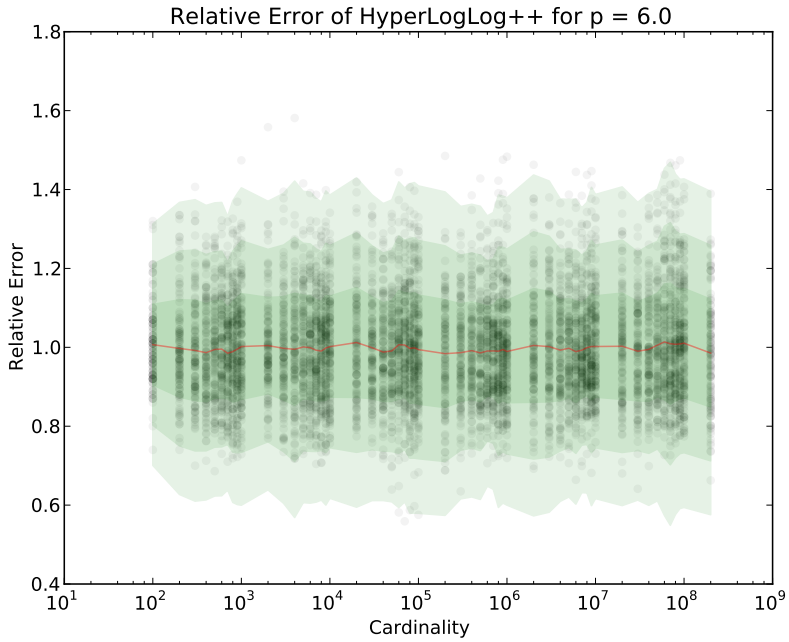
0.90 Confidence

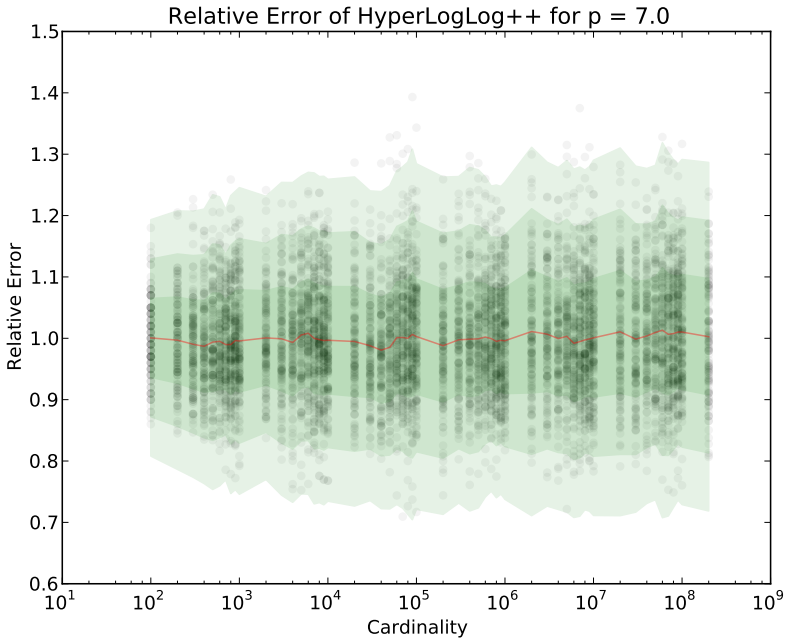
	α			
$J(A_1, \ldots, A_n)$	1.0	0.5	0.25	0.1
0.0001	40000	133334	–	–
0.001	4000	13334	48800	285455
0.01	350	1334	4800	28273
0.05	70	254	928	5419
0.1	30	114	432	2564

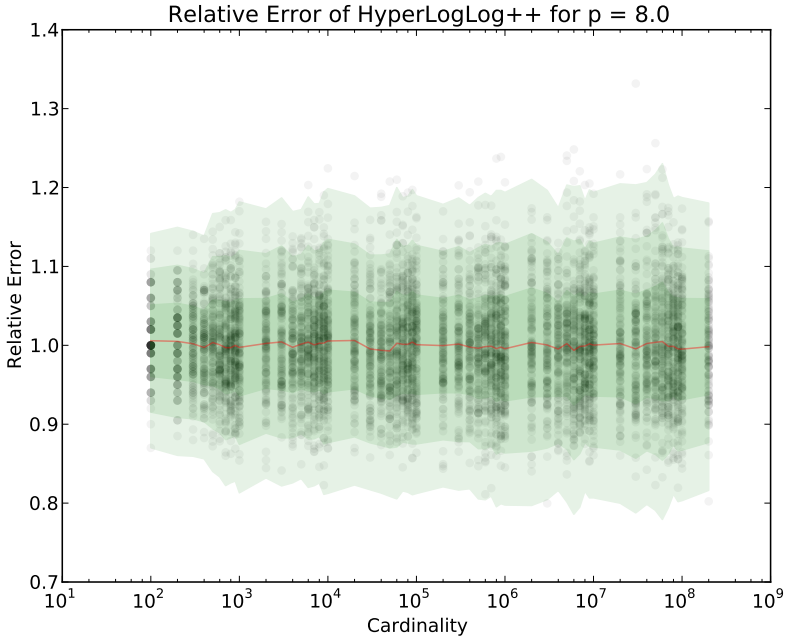
Appendix B

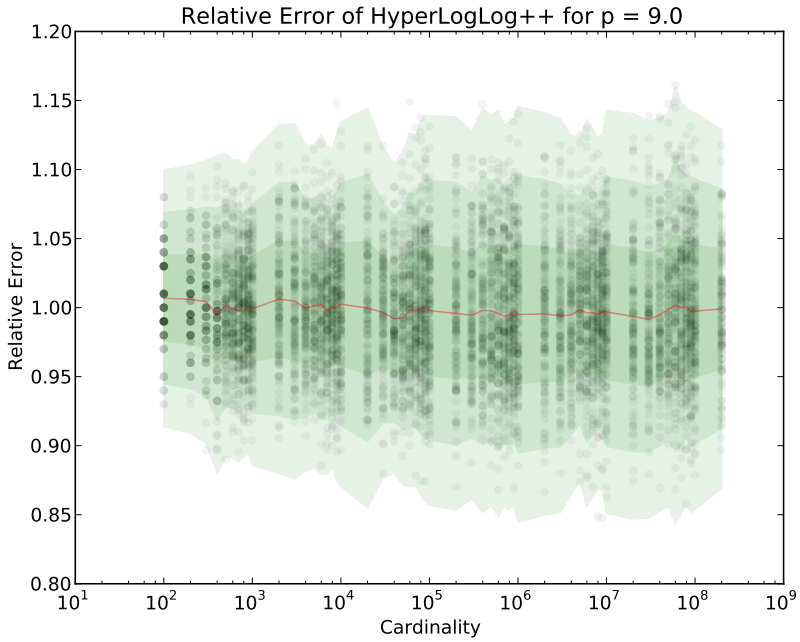

Relative Error of HyperLogLog++ for p = 4.0
Relative Error
Cardinality
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8
10^9

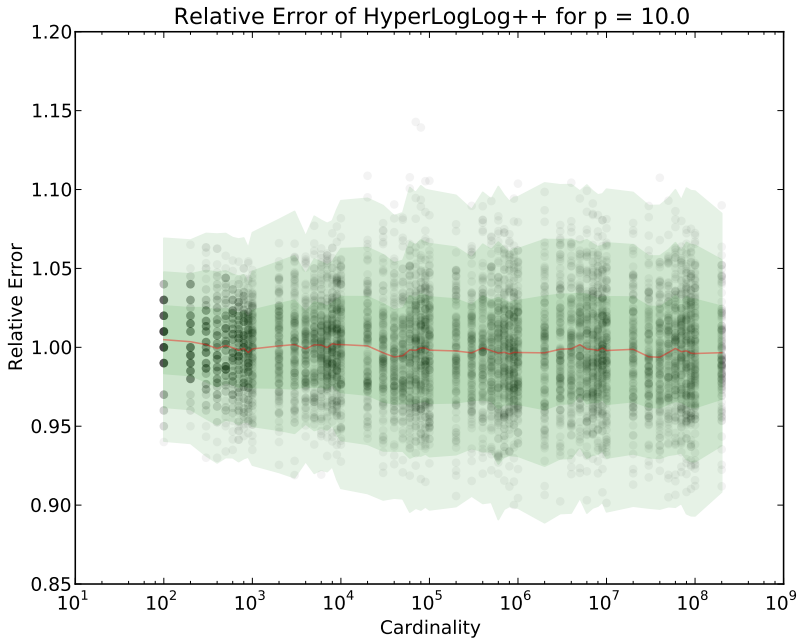

Relative Error of HyperLogLog++ for p = 5.0
Relative Error
Cardinality
2.5
2.0
1.5
1.0
0.5
0.0
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8
10^9

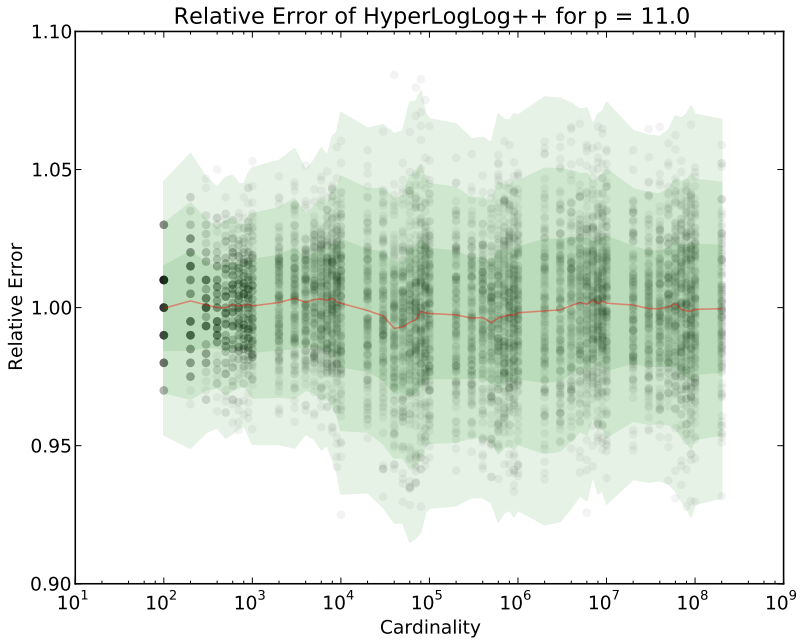

Relative Error of HyperLogLog++ for p = 6.0
Relative Error
Cardinality
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8
10^9

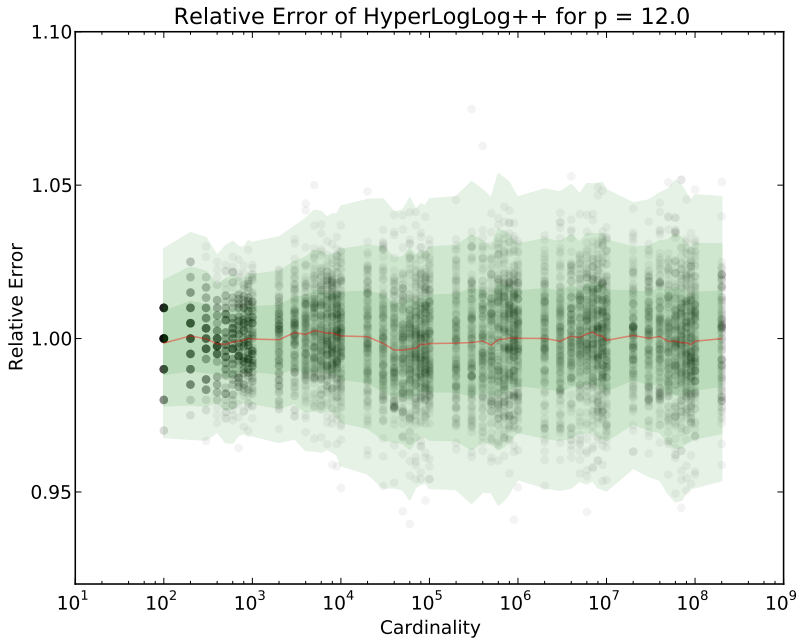

Relative Error of HyperLogLog++ for p = 7.0
Relative Error
Cardinality
1.5
1.4
1.3
1.2
1.1
1.0
0.9
0.8
0.7
0.6
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8
10^9

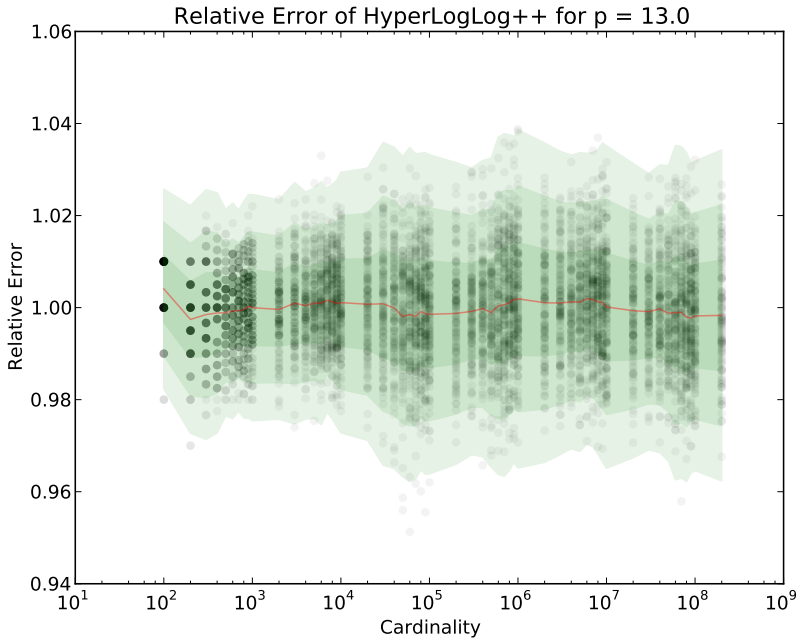

Relative Error of HyperLogLog++ for p = 8.0
Relative Error
Cardinality
1.4
1.3
1.2
1.1
1.0
0.9
0.8
0.7
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8
10^9

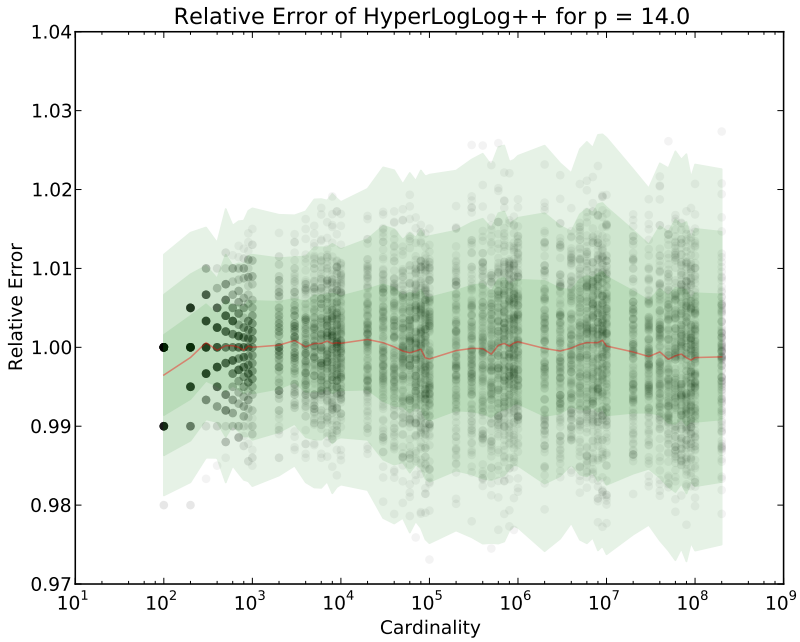

Relative Error of HyperLogLog++ for p = 9.0
Relative Error
Cardinality
1.20
1.15
1.10
1.05
1.00
0.95
0.90
0.85
0.80
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8
10^9

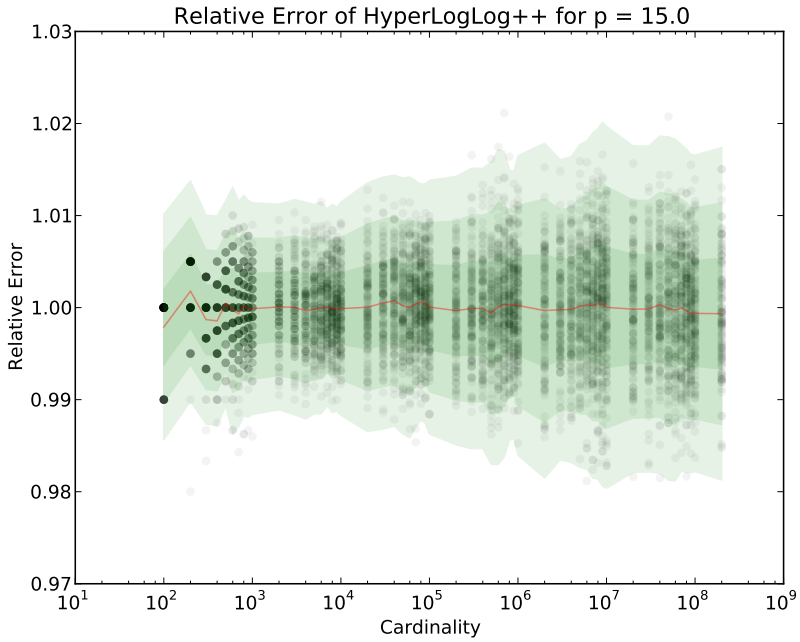

Relative Error of HyperLogLog++ for p = 10.0
Relative Error
Cardinality
1.20
1.15
1.10
1.05
1.00
0.95
0.90
0.85
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8
10^9

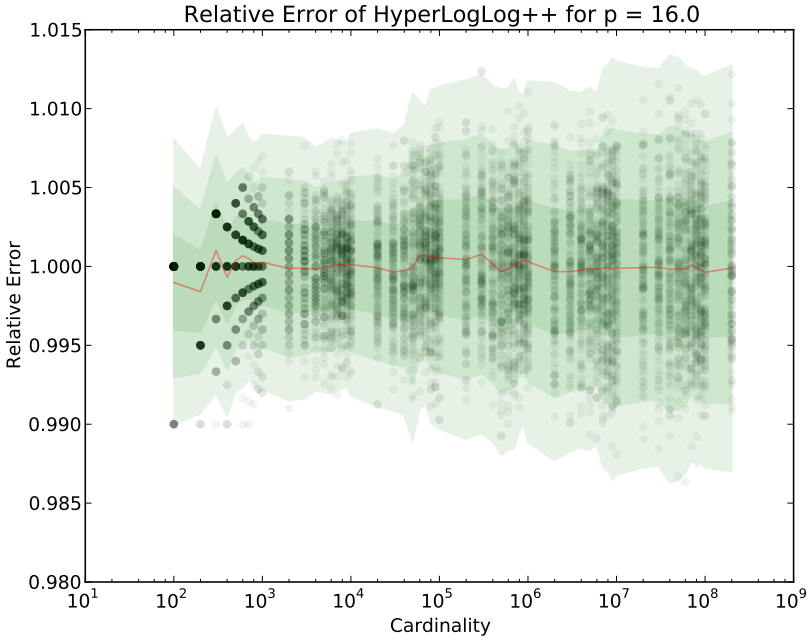

Relative Error of HyperLogLog++ for p = 11.0
Relative Error
Cardinality
1.10
1.05
1.00
0.95
0.90
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8
10^9

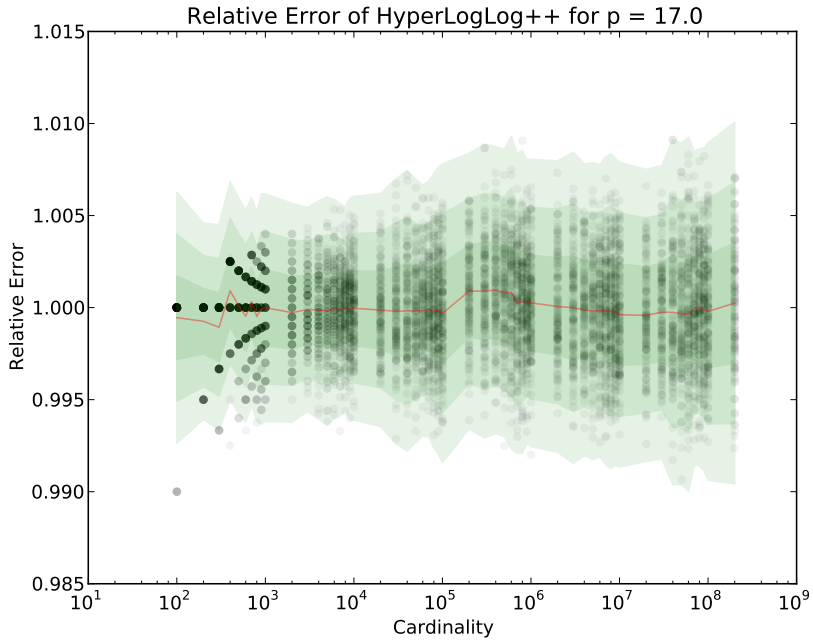

Relative Error of HyperLogLog++ for p = 12.0
Relative Error
Cardinality
1.10
1.05
1.00
0.95
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8
10^9

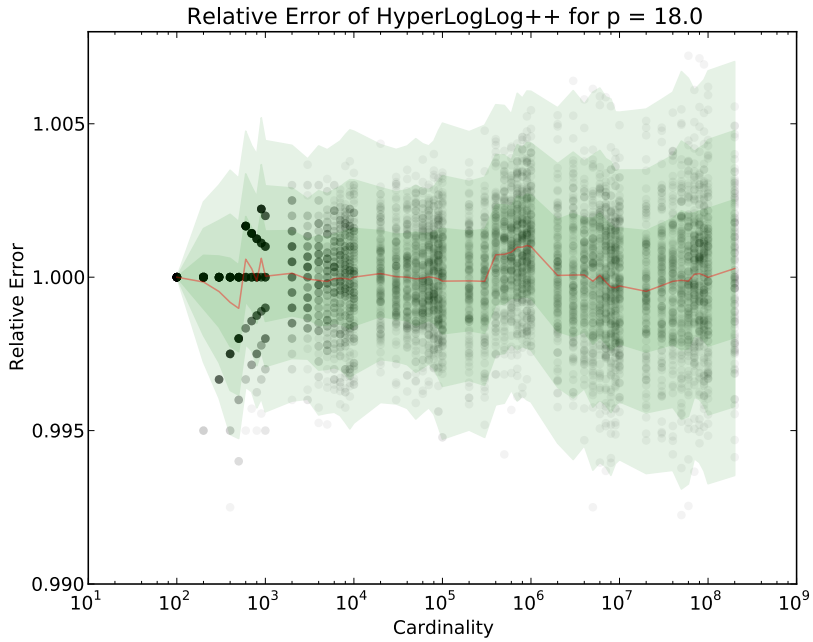

Relative Error of HyperLogLog++ for p = 13.0
Relative Error
Cardinality
1.06
1.04
1.02
1.00
0.98
0.96
0.94
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8
10^9


Relative Error of HyperLogLog++ for p = 14.0
Relative Error
Cardinality
1.04
1.03
1.02
1.01
1.00
0.99
0.98
0.97
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8
10^9


Relative Error of HyperLogLog++ for p = 15.0
Relative Error
Cardinality
1.03
1.02
1.01
1.00
0.99
0.98
0.97
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8
10^9


Relative Error of HyperLogLog++ for p = 16.0
Relative Error
Cardinality
1.015
1.010
1.005
1.000
0.995
0.990
0.985
0.980
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8
10^9


Relative Error of HyperLogLog++ for p = 17.0
Relative Error
Cardinality
1.015
1.010
1.005
1.000
0.995
0.990
0.985
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8
10^9


Relative Error of HyperLogLog++ for p = 18.0
Relative Error
Cardinality
1.005
1.000
0.995
0.990
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8
10^9