AN OVERVIEW OF CARDINALITY ESTIMATION ALGORITHMS

Cardinality Estimation

- How many unique elements are in a set?
- In SQL:
  - SELECT COUNT(DISTINCT ip_addr) AS Cardinality
  - Fine for thousands of records, very slow for billions
- Rather than calculating the exact cardinality, ***estimate*** it

Cardinality Estimation Goals

- Both online and offline calculation are valid use cases
- Memory usage must be controlled
  - Especially for online calculation!
- Error rates must be predictable and configurable depending on the situation at hand

Use Cases

- A frequent query at Google: how many unique IP addresses visited Gmail today?
  - How many from Fort Collins, CO?
- In a given range of temperature readings, how many were unique?
- If the cardinality of a user's outgoing connections is high, could they be infected with malware?
- How many unique words are in *Hamlet*?

Algorithms

- Bloom Filter
- Linear Counting
- Probabilistic Counting
  - HyperLogLog
  - HyperLogLog++

Bloom Filter

- Recall: bloom filters tell us whether an element is a member of a set
  - False positives possible, no false negatives
- The process:
  1. Insert incoming values into our bloom filter
  2. If the inserted value is not in the filter, increment the cardinality counter
- Much more compact than using a bit vector and hash function, at the cost of accuracy

Bloom Filter: Issues

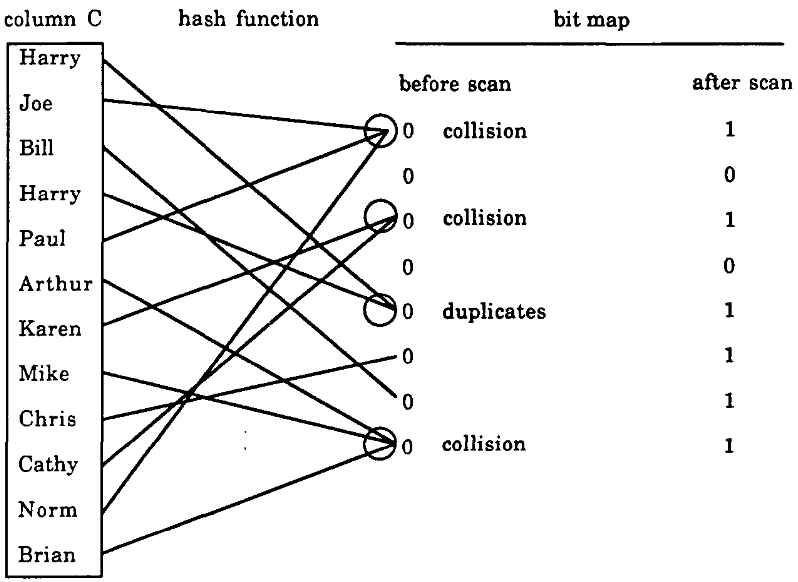
- We need to have an idea of how big our set is ahead of time
  - Bit vectors are allocated up front
- Difficult to resize (but possible)
- Error rates can fluctuate
  - As the number of elements increases, accuracy will decrease
  - Causes cyclic accuracy levels

Linear Counting

- Allocate a bit vector of M bits
  - Adjust M based on the expected upper bound for cardinality
- Apply a hash function on incoming elements
- Use the hash value to map to a bit in the vector, and set it to **1**
- Cardinality = M * log(M/Z);
  - Where 'Z' is the number of 'zero bits'

Linear Counting: Implications

- Very accurate for small cardinalities
  - Becomes less efficient as we scale up
- Error is determined by frequency of hash collisions
- Can be compressed to further reduce space



Probabilistic Counting Algorithms

- Assume we have a set of random binary integers
- Inspecting the bits, what is the probability that a given integer ends in Z zeroes?
  - $1 / 2^Z$
- 10111010 = 50%
  - 10111100 = 25%
    - 10011000 = 12.5%
- This means the likely cardinality is 2^Z
- Fun fact: counting the number of trailing zeroes in a binary number is hardware accelerated

However…

- If you were flipping a coin and told me the longest run of 'heads' you've seen is 3
  - I'd assume you weren't flipping the coin for very long
- Let's say you sat down and flipped a coin 10 times, all landing 'heads.'
  - Apart from possibly indicating a two-headed coin, this would cause my "coin flipping time" estimate to be waaaaay off
- Besides all this, who counts unique **random** integers?

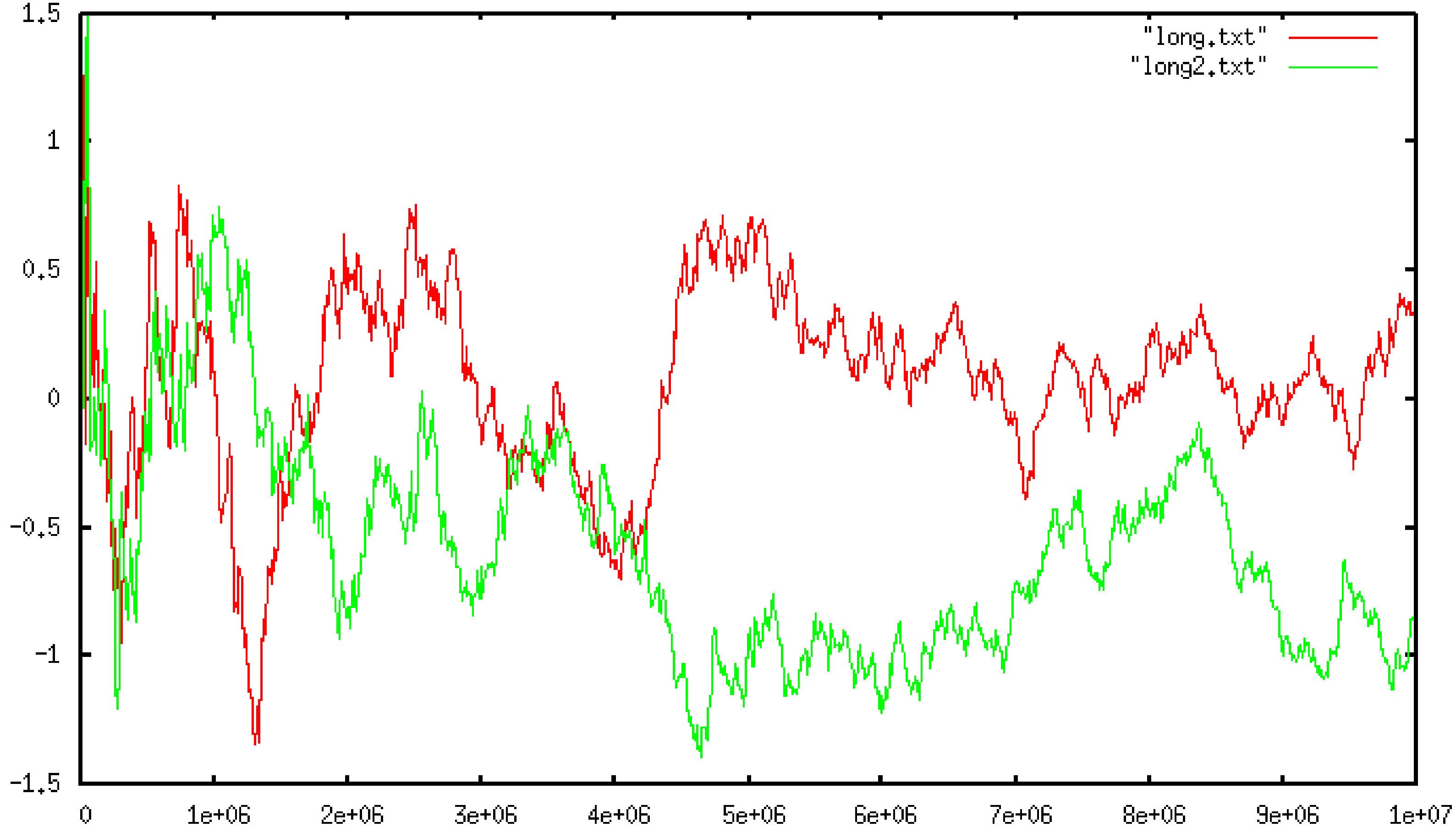
HyperLogLog

- Hash incoming values to 'randomize' them
  - Reference implementation uses a 32 bit hash function
- Instead of just counting trailing (or leading) zeroes, maintain a set of registers
  - These divide incoming values up into several samples
  - Now if I have 10 registers and you flip your two-headed coin 10 times, I still make an accurate estimate
  - ***Stochastic Averaging***
- Average the results across sample sets

HyperLogLog Benefits

- With R registers, the standard error of HLL is:
  - 1.04/sqrt(R)
  - Makes configuration simple
- With an accuracy level of 2%, cardinalities up to 10^9 can be calculated with 1.5 KB of memory
  - Using this algorithm online is very space-efficient!

Error Consistency



Pitfalls

- After cardinalities of 10^9, hash collisions become more frequent and we lose our tight accuracy bounds
- The algorithm does not cope well with small cardinalities
- To deal with these issues, Google has introduced HyperLogLog++

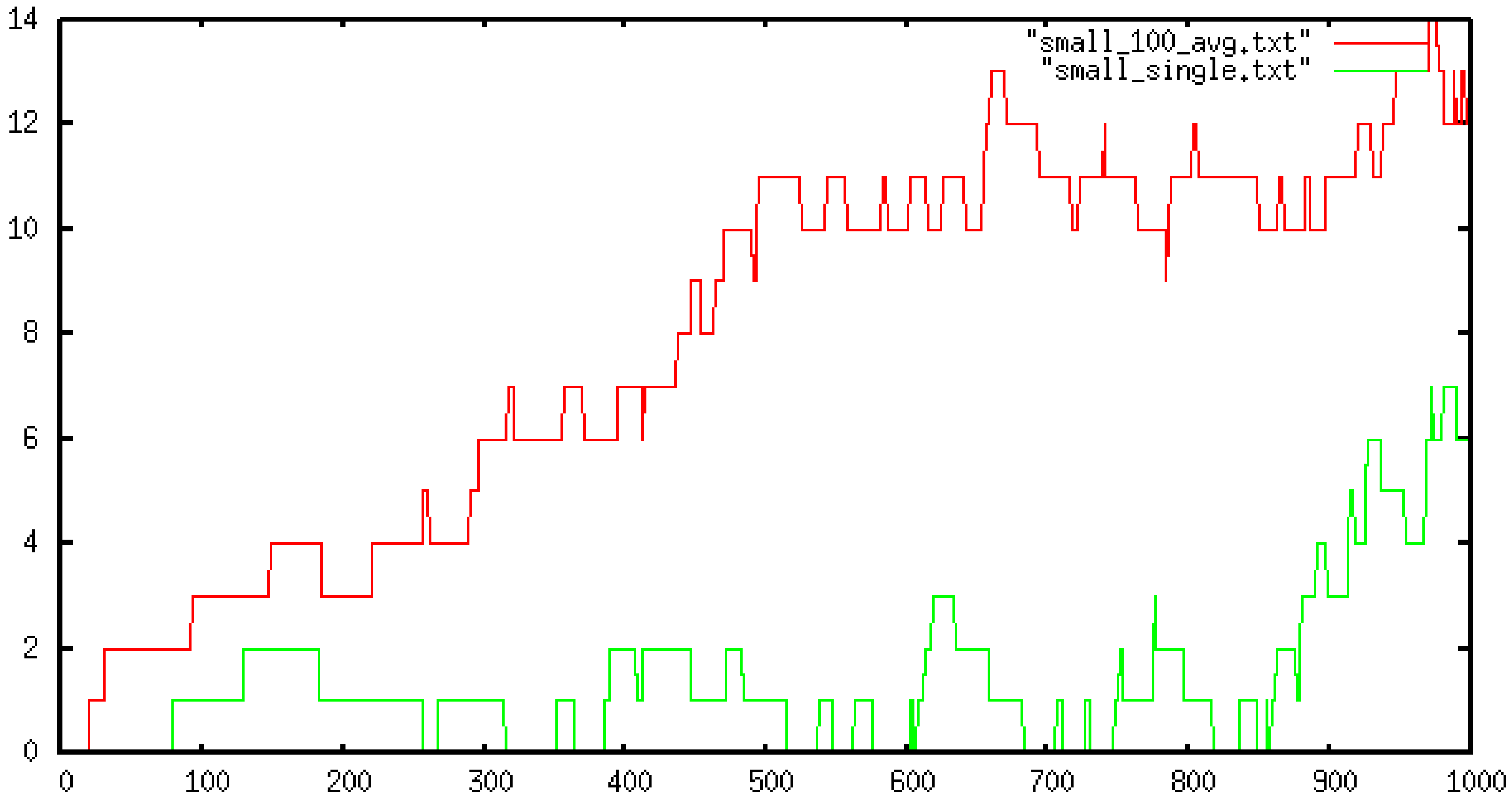
64 Bit Hash Function

- The hash function in HLL is limited to 32 bits
  - This limits us to cardinalities of 10^9 before collisions start to be a problem
  - HLL implements special logic to deal with cardinalities near 2^{32}
- Swapping this with a 64 bit hash instead:
  - Results in a small increase in memory usage
  - Pushes our upper bound to 2^{64}
  - Eliminates the edge case logic

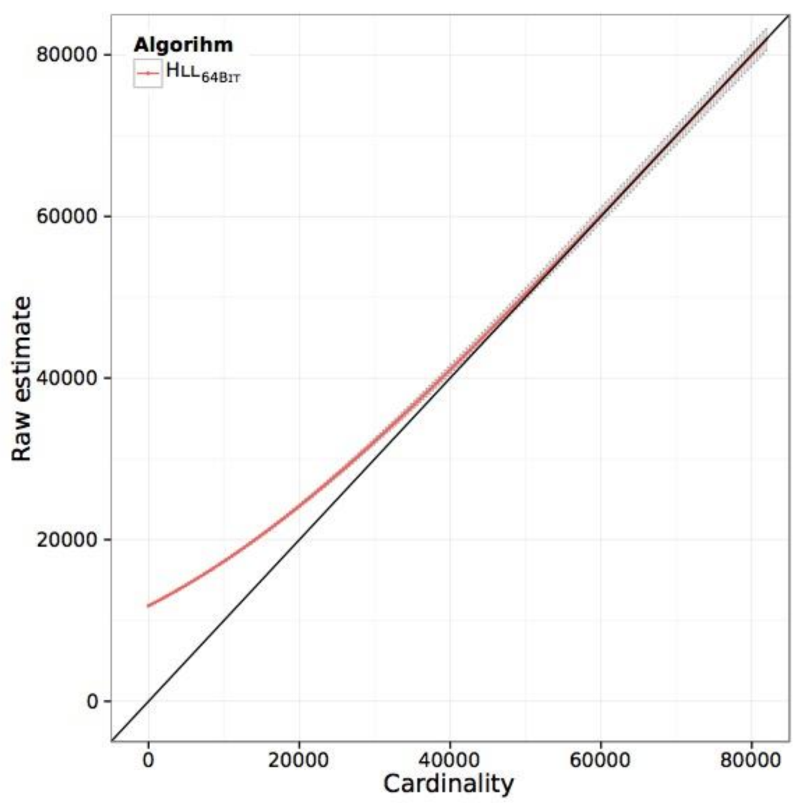
Error Rates

- With very small datasets, HLL produces large error rates
- "SuperLogLog" attempts to mathematically correct this issue
  - …with limited success
- Alternative: use Linear Counting for small cardinalities
  - HLL registers are tweaked slightly to act as linear counting bit vectors

Small Cardinality Error Rates

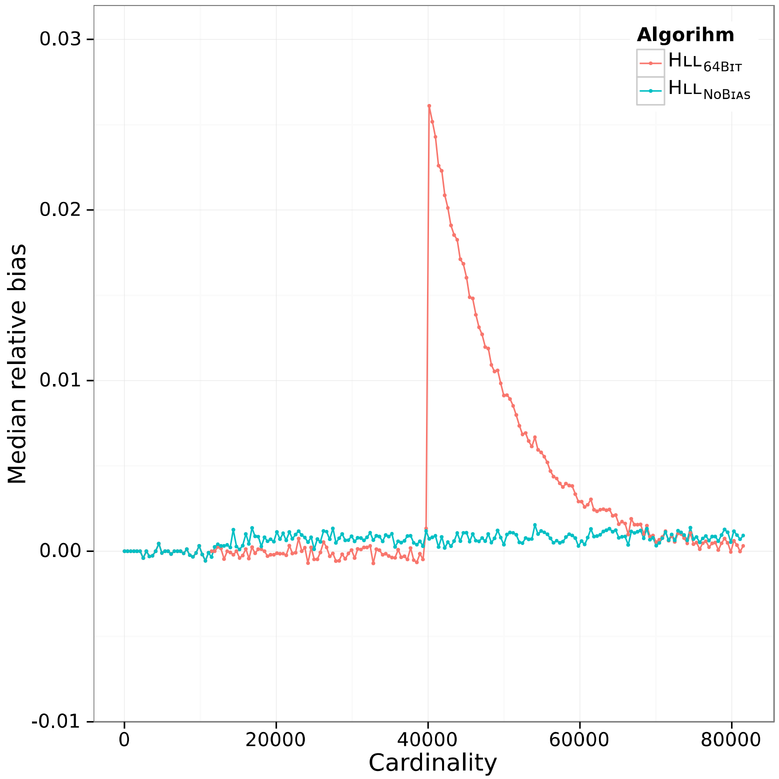


Error Rates: Another Look



Bias Correction

- Linear Counting starts consuming too much memory before HLL hits its usual accuracy levels
- Switching over to HLL early produces a small range of high error rates



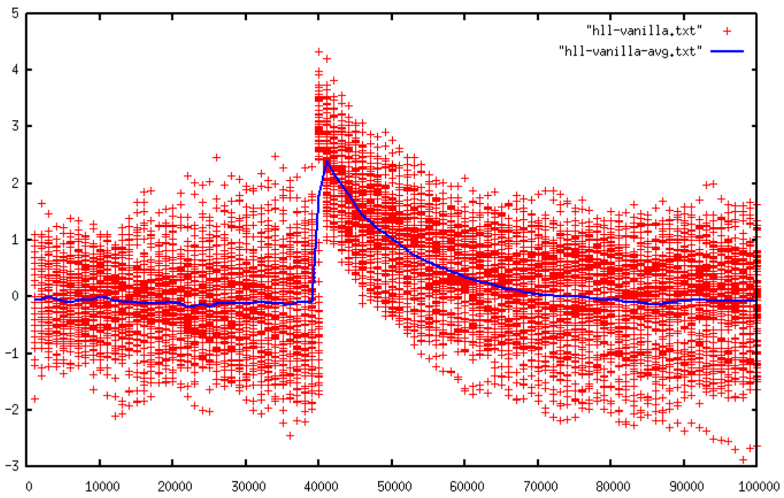
Bias Correction 1

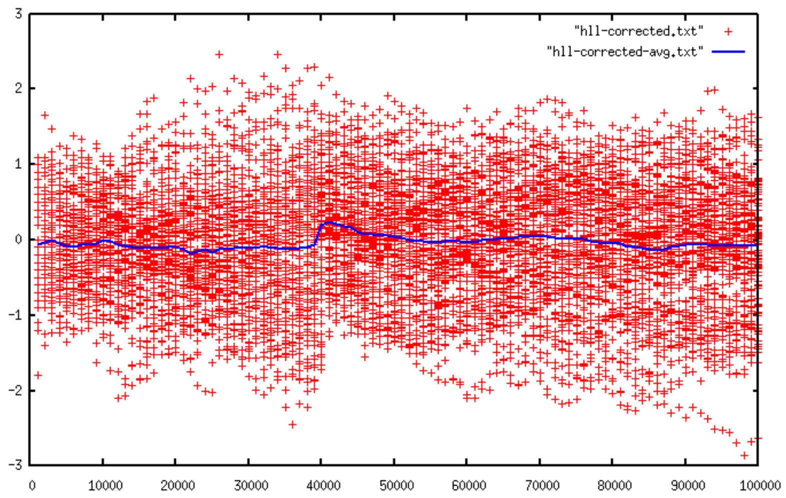
- Google calculated cardinalities for the 40-80k range depicted previously
- Using this empirical dataset, a lookup table provides estimates for cardinalities between 40-80k

Bias Correction 2

- Redis takes an alternative approach: polynomial regression
- Since the curve is fairly smooth, this allows the bias for the 40-80k range to be predicted and corrected

Redis Bias Correction





Conclusions

- Cardinality estimation has been an important topic in databases since the 70s
  - HyperLogLog (2007)
  - HyperLog++ (2013)
- Being able to estimate cardinality lets us:
  - Estimate other dataset parameters
  - Reason about data distributions
    - Optimize indexes