CS 686: Special Topics in Big Data

Counting Streams

Lecture 24

Today's Schedule

- Cluster status and P2 Update
- Bloom Filters
- Cardinality Estimation

Today's Schedule

- **Cluster status and P2 Update**
- Bloom Filters
- Cardinality Estimation

Cluster Status

- Things are back up and running, so you can pretend Friday's lecture never happened ☺
- But now you also have the option of doing local analysis on your own MapReduce install
  - Use the compressed 30% sample dataset

Clarifying our (many) Datasets

- We now have three datasets:
  1. nam_mini – this is for testing your code
  2. nam_2015 – the original data, hosted on bass
  3. nam_2015_S – a 30% sample of the full dataset on bass
- You can use #2 or #3 to answer the questions, but make sure you specify which dataset you're using!

Thinking Ahead

- P3 will now include Deliverable II
- We'll be using Hadoop **and** Spark
- You'll also have the chance to choose your own dataset to analyze
- Some ideas:
  - http://academictorrents.com
  - Earth on AWS: https://aws.amazon.com/earth/
  - Potentially (if large enough): http://archive.ics.uci.edu/ml/index.php

Today's Schedule

- Cluster status and P2 Update
- **Bloom Filters**
- Cardinality Estimation

Bloom Filter (1/2)

- Compact data structure to test for set membership
- Supports two functions:
  - put(data) → places some information in the filter
  - get(data) → reports the **probability** of whether or not the data was put in the filter
- May produce false positives but **never** false negatives
  - If the bloom filter says it wasn't inserted, then it definitely wasn't!

Bloom Filter (2/2)

- Bloom filters give us two answers:
  - Maybe (with a probability)
  - Definitely not
- You can think of it as a HashMap that throws away the keys and values
  - We can't get the data back out of the bloom filter, but it is **very** compact in memory!
- Great when acting as a gatekeeper to a high-latency data source (such as disks!)

Building a Bloom Filter (1/2)

- To implement a bloom filter, we need:
  - An array of bits
  - Multiple hash functions
- When putting an item in the filter, the data is passed to the hash functions
- The **hash space** of each function is mapped to our array of bits
  - Very much like a DHT

Building a Bloom Filter (2/2)

- We take the position in the hash space, map it to our array of bits, and then set the corresponding bit to **1**
  - Repeat for all hash functions
- To perform a lookup, we repeat the process
  - If all the positions in the bit array are set to **1**, then we can return a "maybe"
  - If **any** of the positions are a **0**, then we return "no"

Collisions

- If the size of our bit array is small, then there is a good chance of **collisions**
  - Two inputs map to the same bit:
    - hash(my_dog.jpg) → bit #3
    - hash(secret_passwords.txt) → bit #3
- This is the source of **uncertainty** in bloom filters
- How do we decide how large to make our filters?

Pareto Principle

- The 80/20 rule
- 80% of the effects come from 20% of the causes
- Pareto's observation: in Italy, 20% of the population controls 80% of the land
- 80% of complaints are made by 20% of the customers
- So: shoot for bit arrays sized at 20% of your input
  - Reasonable starting point

Sizing our Bloom Filters

- If we can accept some uncertainty, we can maintain even smaller bit arrays in memory
- Fewer than 10 bits per element are required for a 1% false positive probability, independent of the size or number of elements in the set
  - -- Bonomi et al., "*An Improved Construction for Counting Bloom Filters*"

Demo Use Case

- One big issue with our DFS is scalability
- What happens when we have too many files to fit all their names into memory on the Controller?
  - We could write the file names and locations to disk, but then retrievals would be much slower
- Instead, we can use bloom filters to predict whether a node has the file we're looking for or not

Indexing via Bloom Filter

- We maintain a bloom filter for each storage node and evaluate lookups against the filter
  - This process is parallelizable and fast
- If the filter returns "no," then we can safely exclude that node from our search
- If it's a "yes," we'll start our search with the highest-probability node
  - We can even make requests in parallel
  - Downside: this costs additional network I/O

Resizing a Bloom Filter

- If our false positive probability starts to go up, we need to resize the bloom filter
- Generally accomplished by creating a new bloom filter and then inserting the values back in
  - Big downside in situations where we can't predict how many values we'll see

Today's Schedule

- Cluster status and P2 Update
- Bloom Filters
- **Cardinality Estimation**

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
  - If gmail says my search returned 6 results +/- a million, then it's not such a useful metric

Use Cases

- A frequent query at Google: how many unique IP addresses visited Gmail today?
  - How many from San Francisco?
- 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++

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

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

Algorithms

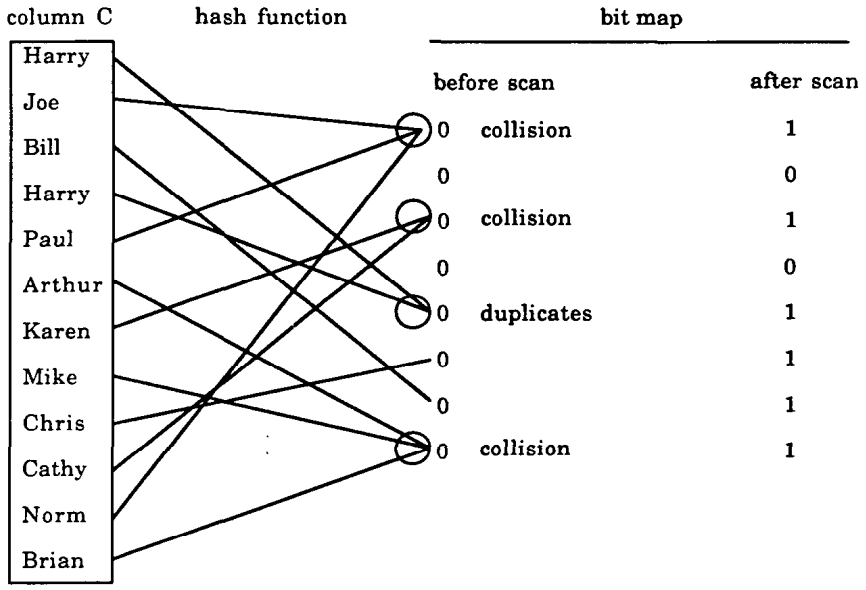
- Bloom Filter
- **Linear Counting**
- Probabilistic Counting
  - HyperLogLog
  - HyperLogLog++

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



Algorithms

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

Probabilistic Counting (1/2)

- 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%

Probabilistic Counting (2/2)

- This means the likely cardinality is 2^Z
- Another way of looking at it: counting based on how **rare** an event is
- Fun fact: counting the number of trailing zeroes in a binary number is hardware accelerated
- Related: recall our discussion on Bitcoin: finding a particular number of **leading** zeros in a hash

Coin Flip

- Let's say you are flipping a coin, and I want to estimate how long you've been flipping it
- If you tell me the longest run of 'heads' was 3, then I will guess you didn't flip the coin very long
- If it was something like 20, then you must've been at it for a really long time!

However…

- 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
- To fix this, I'd divide up your workload
  - Rather than reporting a single value, I'll give you 10 pieces of paper to record the number of heads/tails on

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

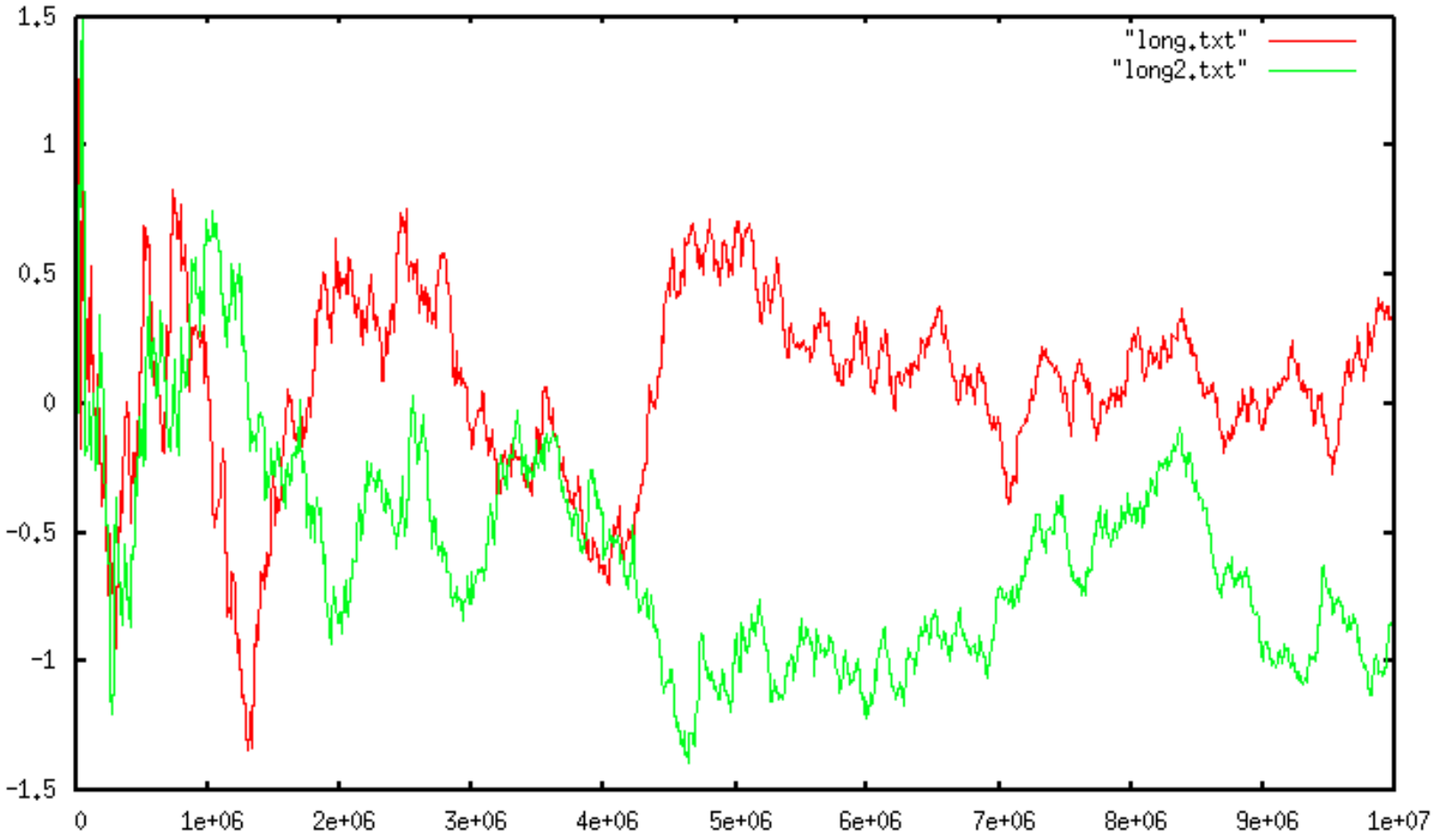
Which Sample?

- We split up the hash:
- One part is used to determine how many zeros were produced
- The other part is used to determine which **register** to update

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 is very space-efficient!

Error Consistency



Source: http://antirez.com/news/75

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 introduced HyperLogLog++

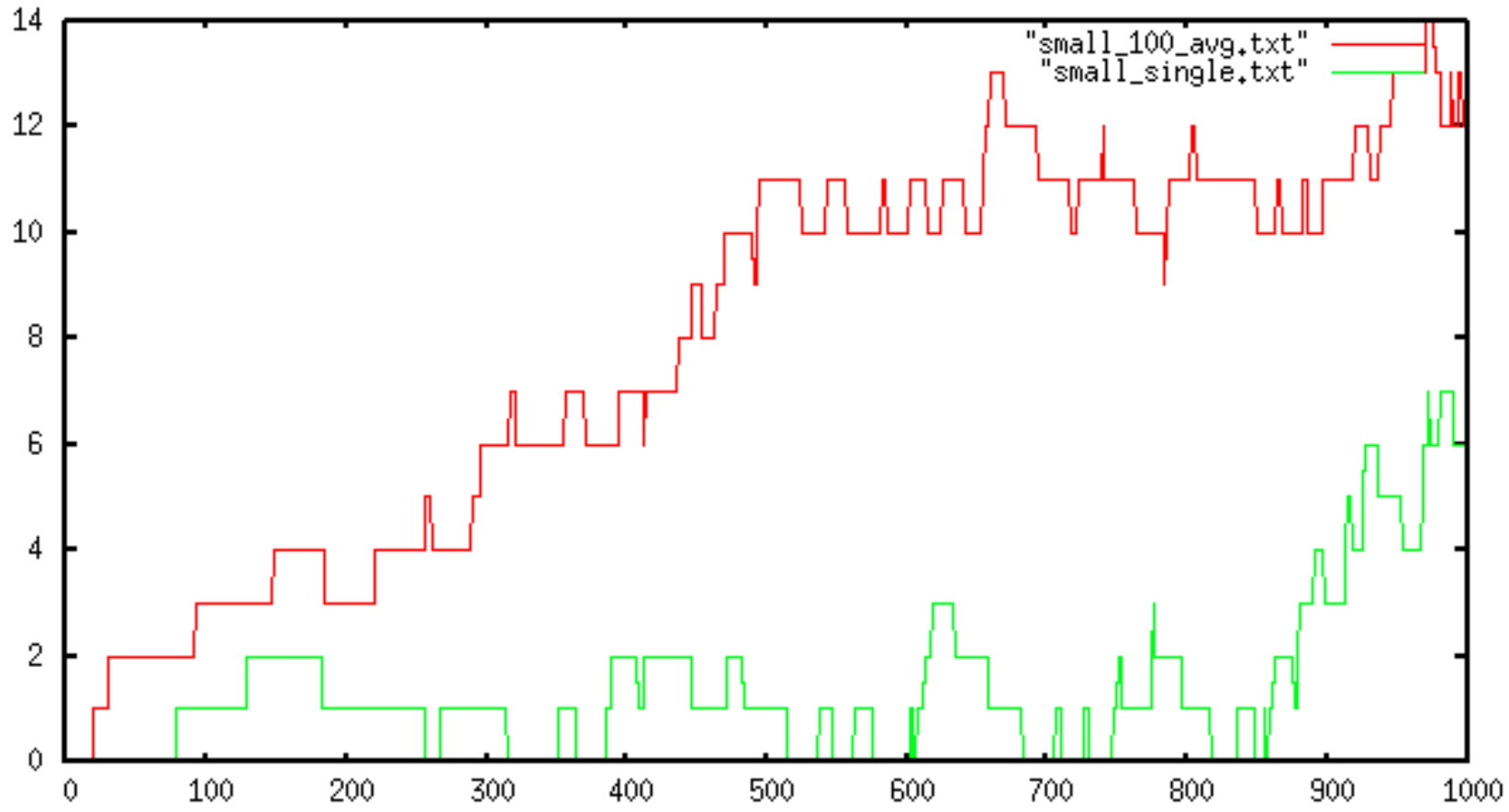
64 Bit Hash Function

- 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

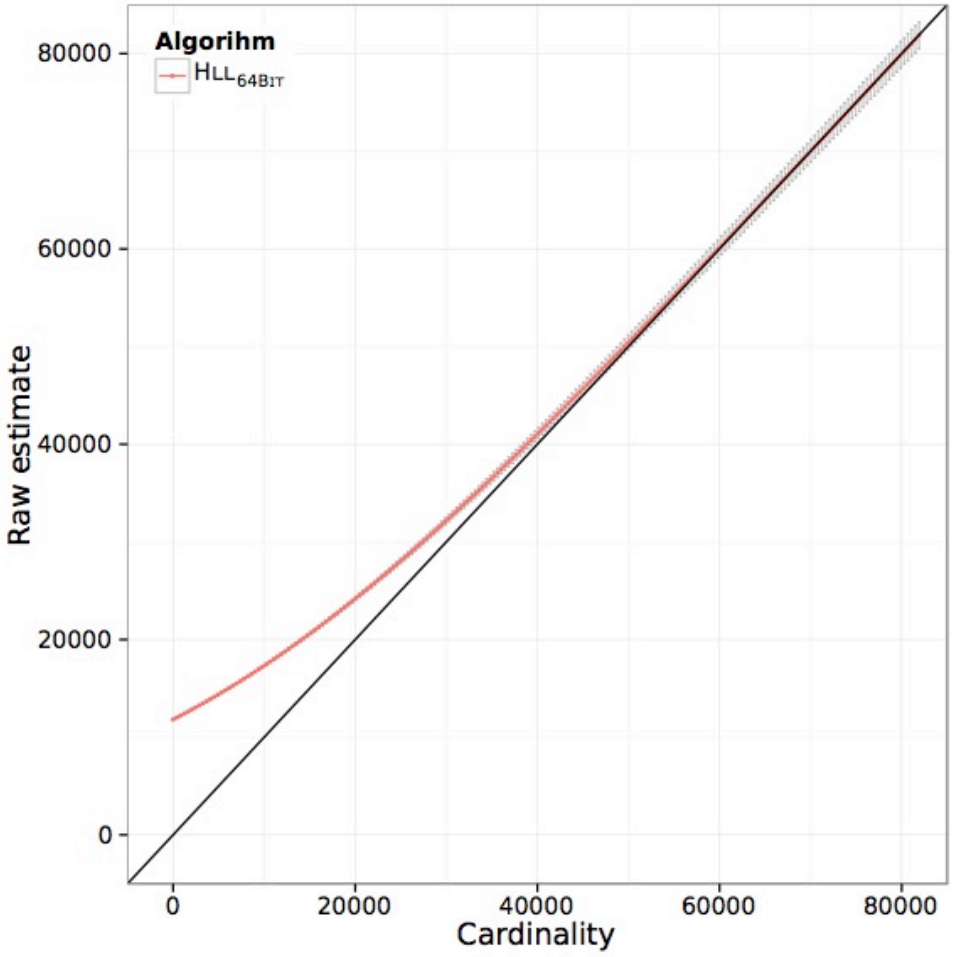
- With very small datasets, HLL produces large error rates
- "SuperLogLog" attempts to mathematically correct this issue
  - …with limited success
- Alternative: use Linear Counting for small cardinalities
  - HLL registers are tweaked slightly to act as linear counting bit vectors

Small Cardinality Error Rates



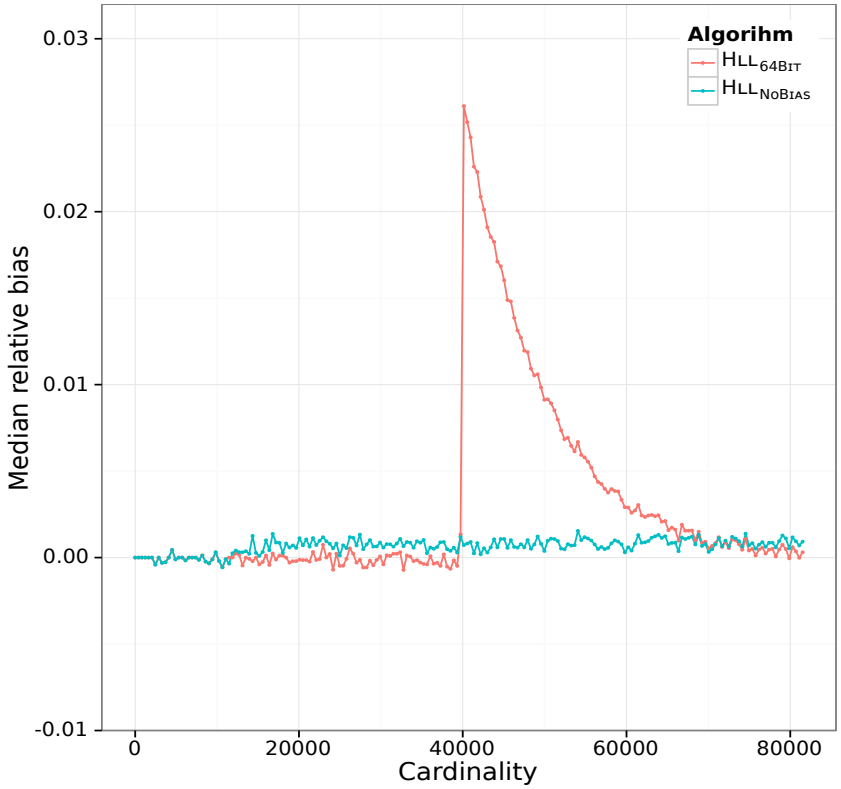
Source: http://antirez.com/news/75

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



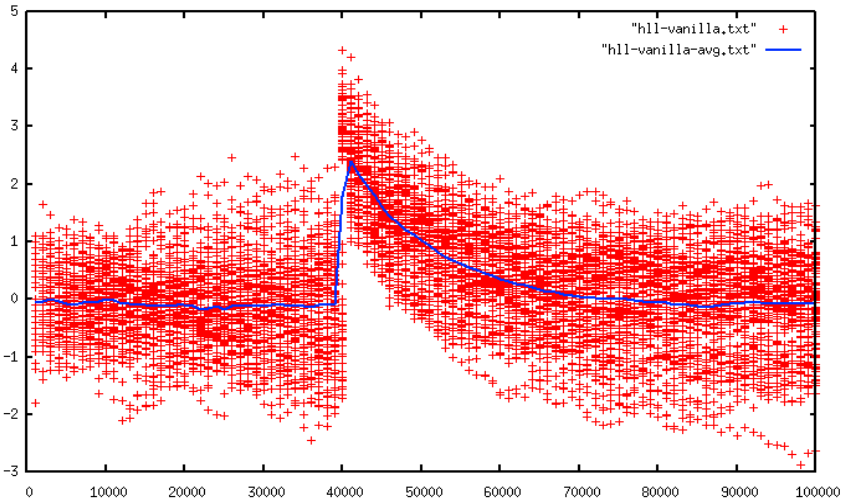
Bias Correction 1

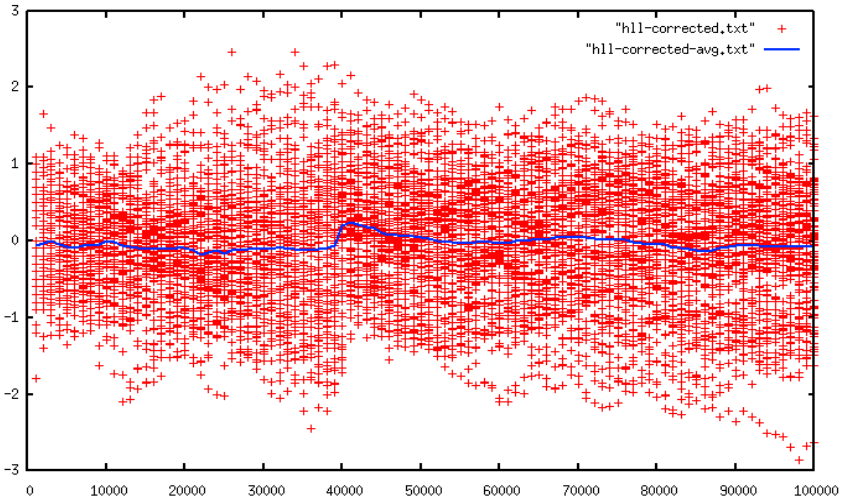
- Google calculated cardinalities for the 40-80k range depicted previously
- Using this empirical dataset, a lookup table provides estimates for cardinalities between 40-80k

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





Source: http://antirez.com/news/75

Interactive Demo

- http://content.research.neustar.biz/blog/hll.html

Wrapping Up (1/2)

- When we deal with billions of data points, sometimes the best way to provide answers fast is to **estimate**
- All of the algorithms discussed today can be used **online**
  - No need to have the entire dataset in hand
  - Update incrementally as we go!
- Estimating is only useful when we can tie a probability to it (such as the false positive probability)

Wrapping Up (2/2)

- 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