KHyperLogLog

Estimating Reidentifiability and Joinability of Large Data at Scale

Pern Hui Chia[1], Damien Desfontaines[1,2], Irippuge Milinda Perera[1],
Daniel Simmons-Marengo[1], Chao Li[1],
Wei-Yen Day[1], Qiushi Wang[1], Miguel Guevara[1]

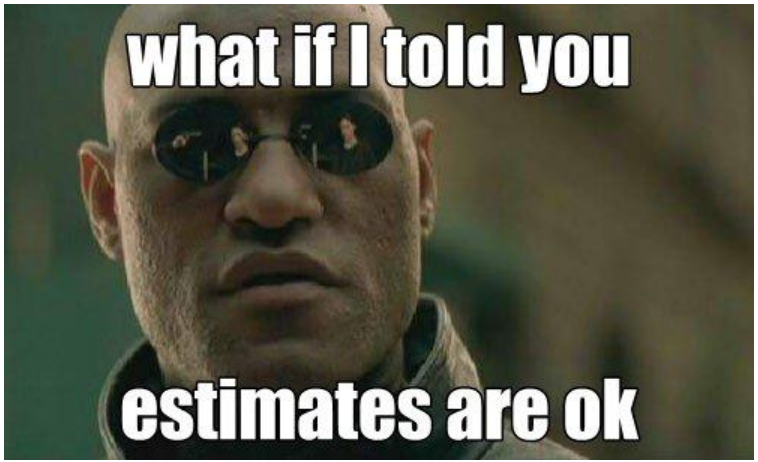
[1] Google, [2] ETH Zurich

Google ETH zürich

Is data reidentifying?

Are data sets joinable?

Problem: Data is LARGE



Google ETH zürich

Background on Approximate Counting

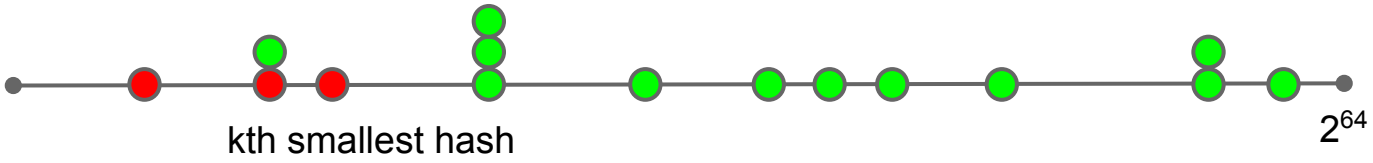
Google ETH zürich

Approximate Counting

- sublinear memory
  - compact data structures (sketches)
  - often bounded size (KBs)
- streaming algorithm
- easily parallelizable
- bounded error rate

K Min Values [1]

- Use a uniform hash function
- Hash input values, keep **K** smallest hashes
- Extrapolate from the density of **K smallest hashes**

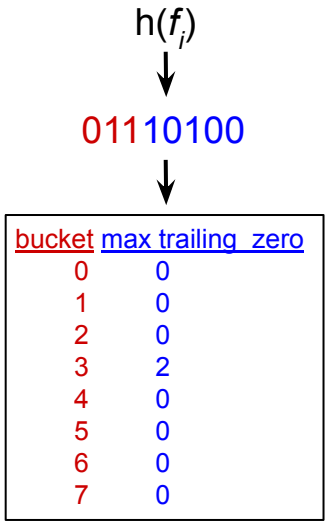


Error rate ~ 1/ sqrt(K)

With K=1024, 64-bit hash: error rate ~3%, size ~8KB

HyperLogLog [2]

- Hash input values ⇒ keep max count of trailing 0's
- High number of trailing 0's ⇒ has seen a lot of unique values
- High variance ⇒ average over M counts



Error rate ~ 1/ sqrt(M)

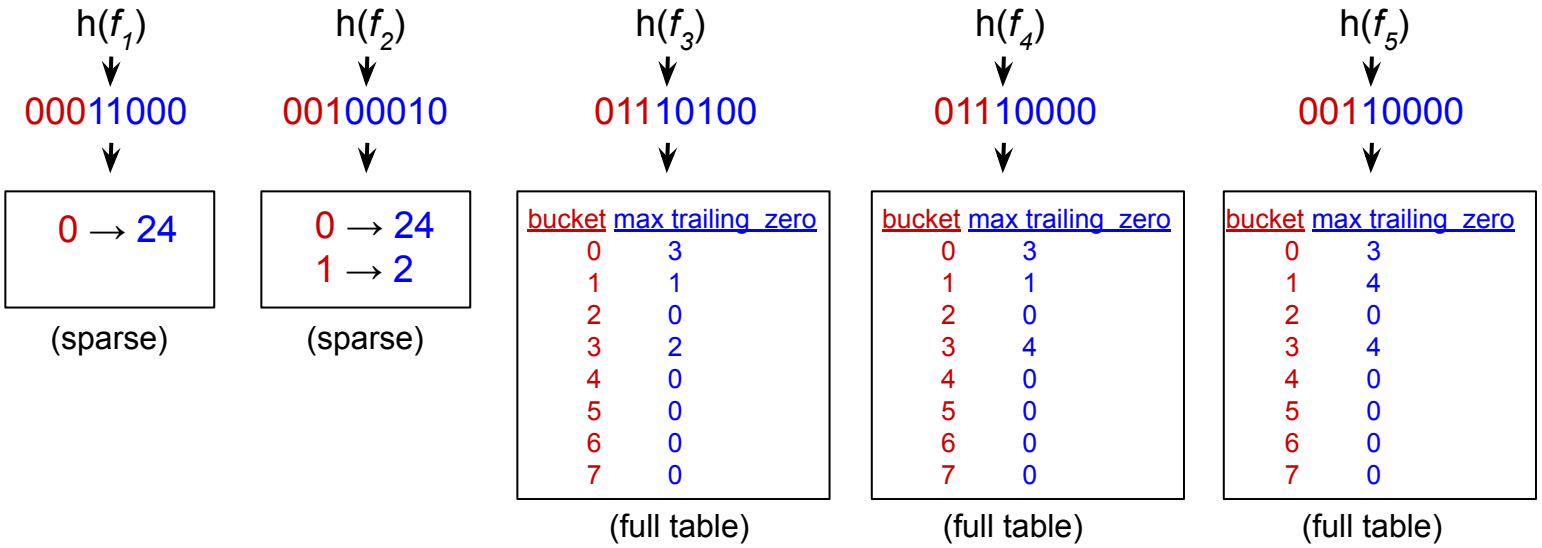
With M=1024, 64-bit hash:

error rate ~3%, size ~1KB

HyperLogLog++ [3]

- Sparse representation for low cardinalities

  ⇒ better estimates + less memory



But cardinality estimate is not enough

Introducing **KHyperLogLog**

to estimate reidentifiability and joinability

Google ETH zürich

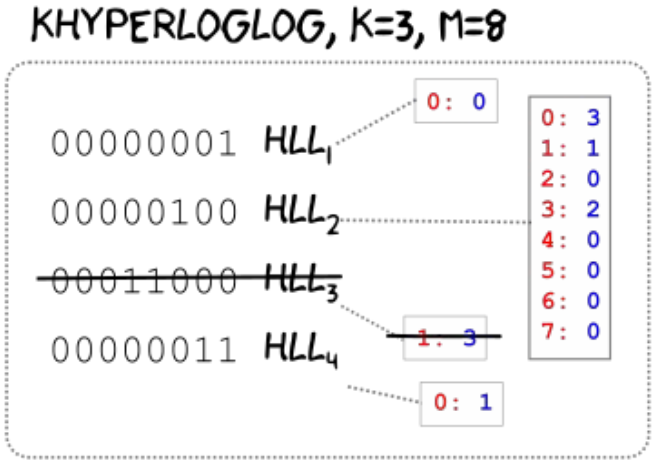
KHyperLogLog (KHLL)

Two level counting

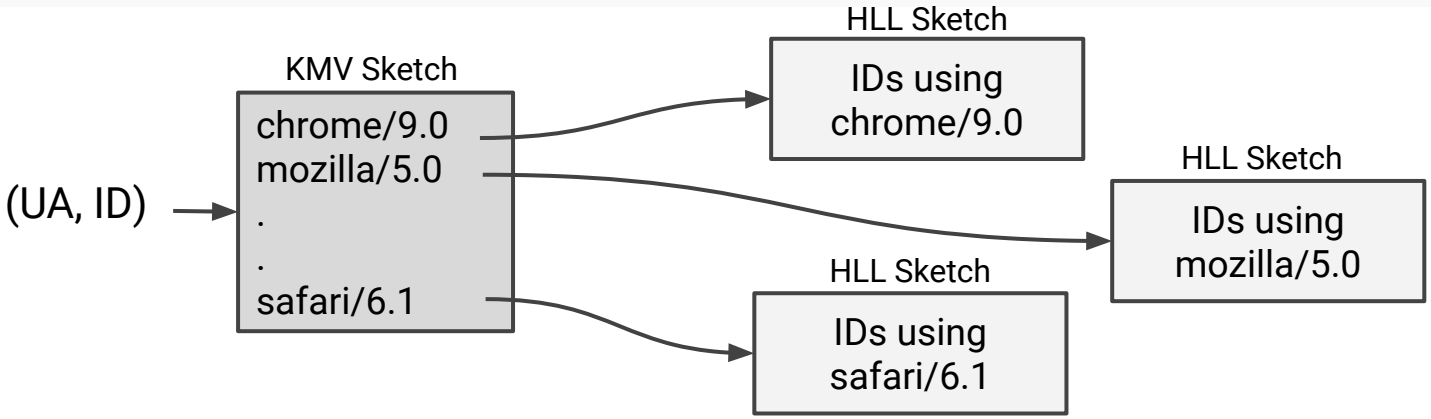
- 1st level → K Min Values
- 2nd level → HyperLogLog++$_{\text{HalfByte}}$

Ex: estimate uniqueness of User Agent (UA) strings

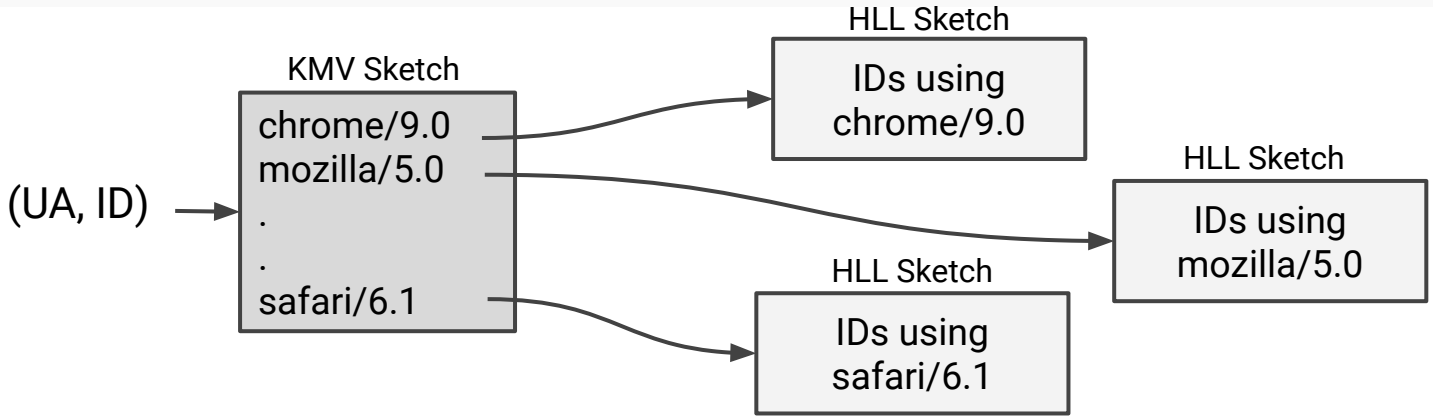
h(UA)	h(ID)
00000001	00000011
00000100	00011000
00000100	00100010
00000100	01110100
00011000	00101000
00000011	00011010



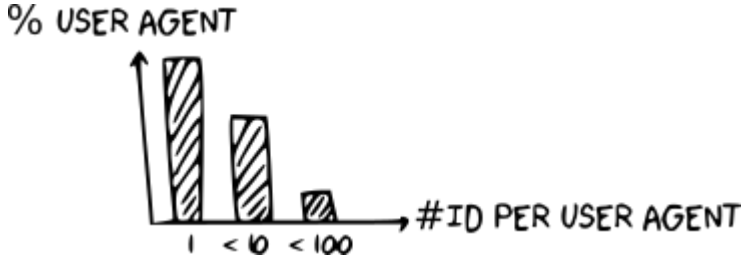
Uniqueness Distribution

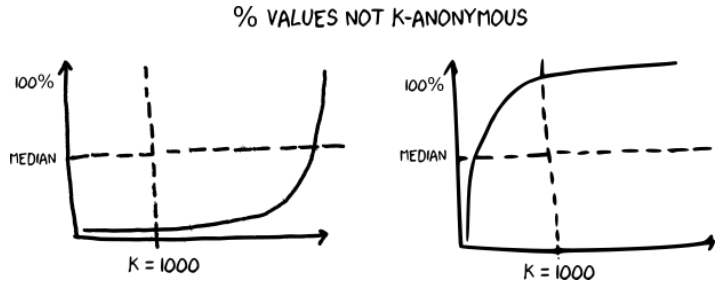


Uniqueness Distribution

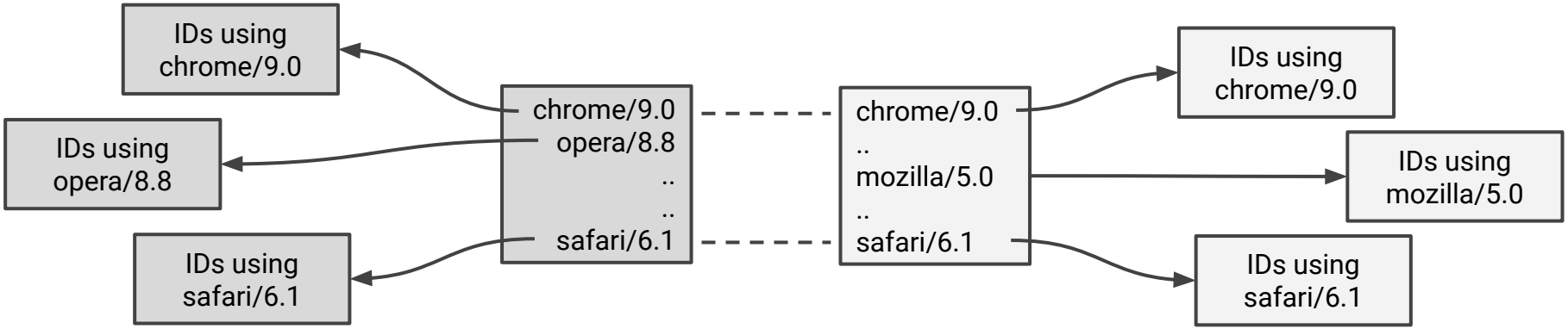


From a KHLL sketch, we can estimate

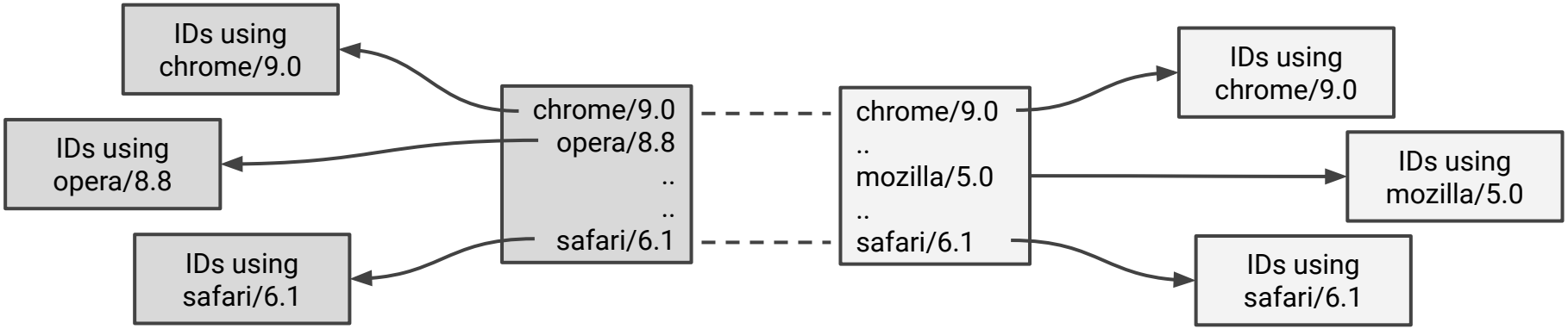




Containment & Joinability



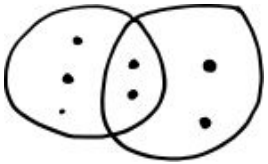
Containment & Joinability



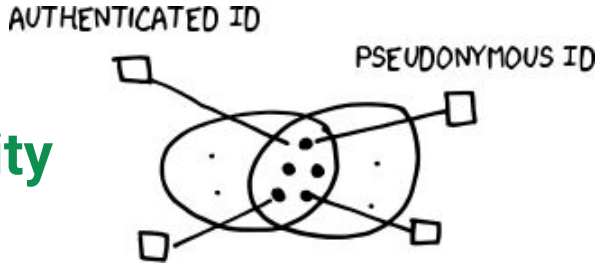
Given 2 KHLL sketches, we can estimate

Containment

= |A ∩ B| / |A|



Joinability



How does KHyperLogLog perform?

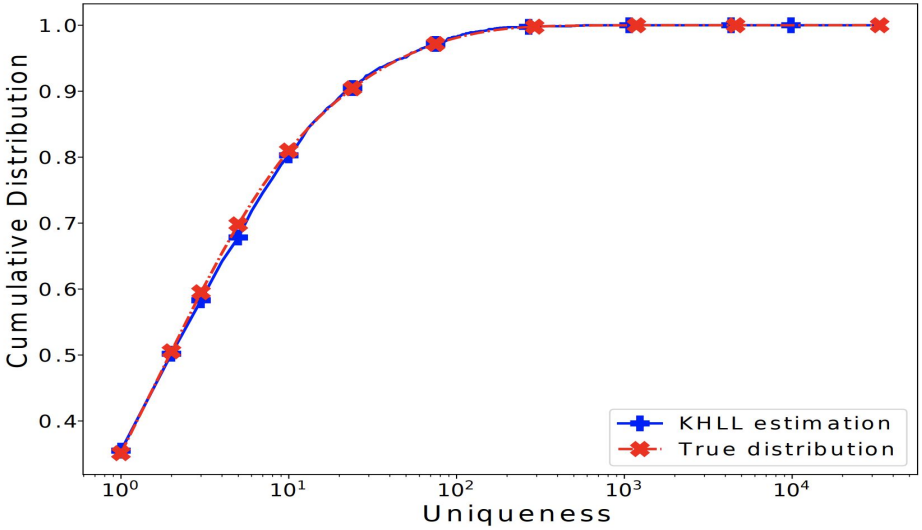
Google ETH zürich

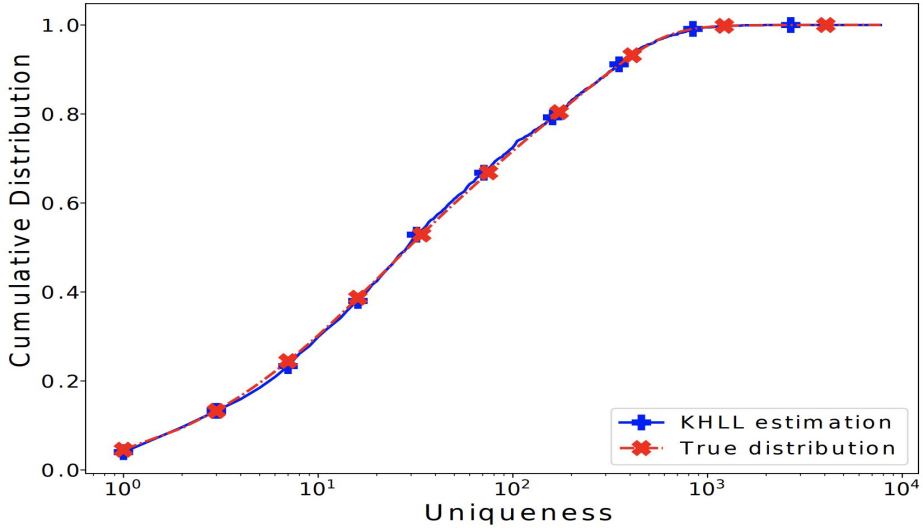
Efficiency

- KHLL vs exact counting
  - CPU performance 1 or 2 orders of magnitude faster
  - Bounded memory usage
    - With K=1024, M=512, size ~256KB
  - Scale to large data sets

Accuracy: Uniqueness Distribution

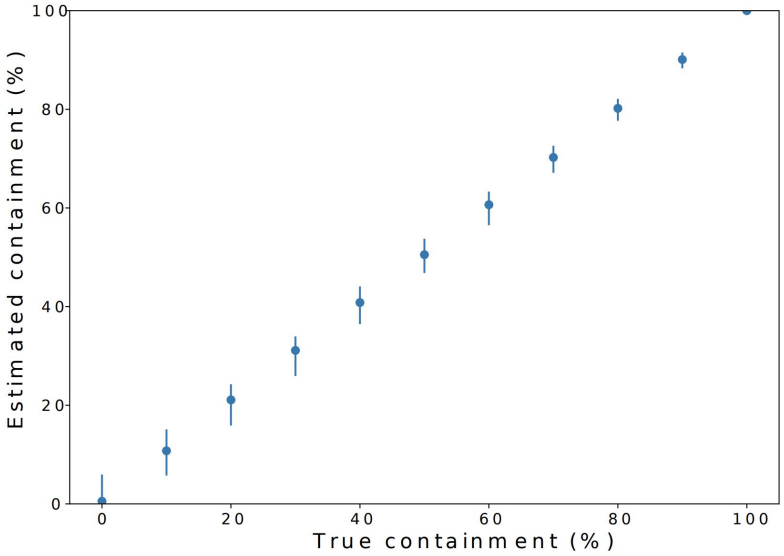
- Prepared test distributions using Netflix & US census data
- Validated accuracy of estimated distributions

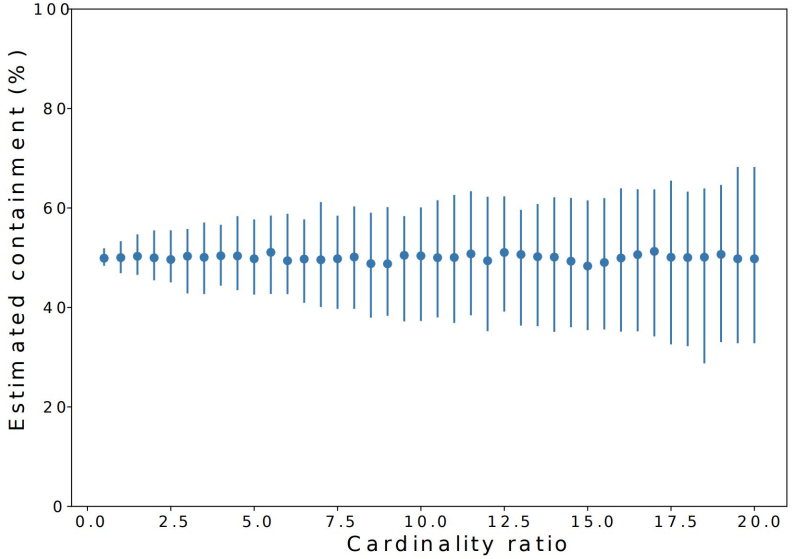


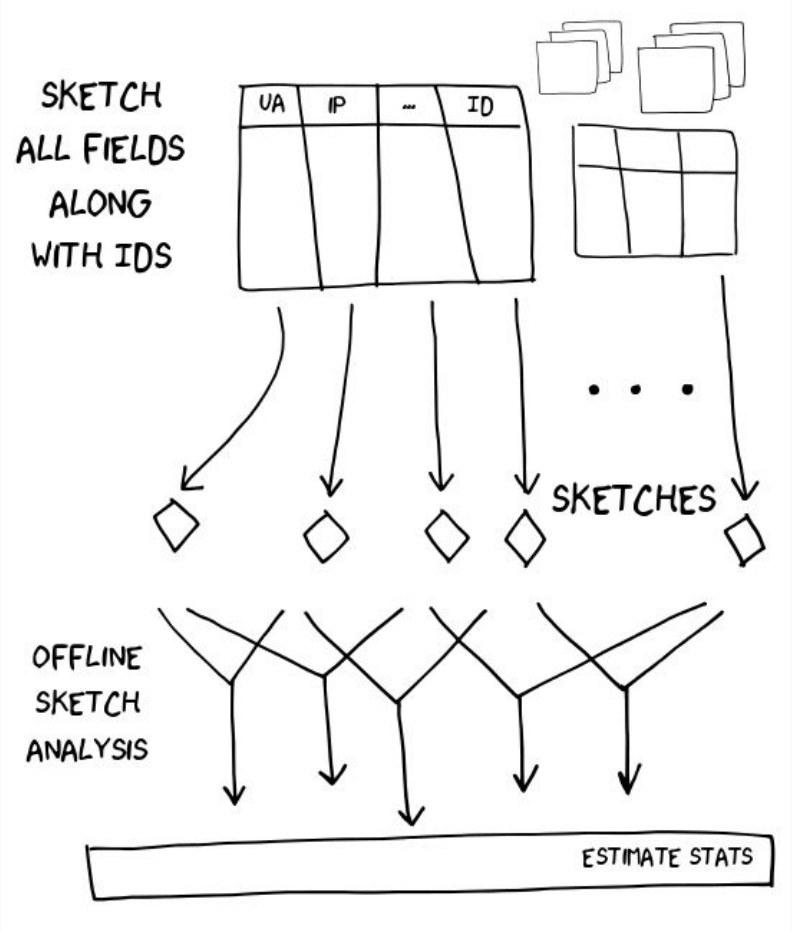


Accuracy: Containment

- Sets of equal size: ±5% error 90% of time
- But, problematic for sets of highly unequal sizes







Practical Use of KHLL

- Read data once (petabytes)
- Produce sketches
- Offline
  - Estimate reidentifiability
  - Estimate joinability

Summary

- KHLL for estimating reidentiability & joinability at scale
- Approximation errors possible, but great for
  - Understanding data e.g., for evaluating data strategies
  - Regression testing esp. detecting data joinability

Summary

- KHLL for estimating reidentiability & joinability at scale
- Approximation errors possible, but great for
  - Understanding data e.g., for evaluating data strategies
  - Regression testing esp. detecting data joinability
- Future work
  - Memory efficient but still CPU intensive
  - Applications in privacy enhancing techniques?

Thanks! Questions about **KHyperLogLog**?

Google ETH zürich

References

1. **On Synopses for Distinct-Value Estimation Under Multiset Operations.** — K. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla.
2. **HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm.** — P. Flajolet, E. Fusy, O. Gandouet and F. Meunier.
3. **HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm.** — S. Heule, M. Nunkesser, A. Hall.

HLL++$_{\text{half-byte}}$ vs HLL++

HLL++

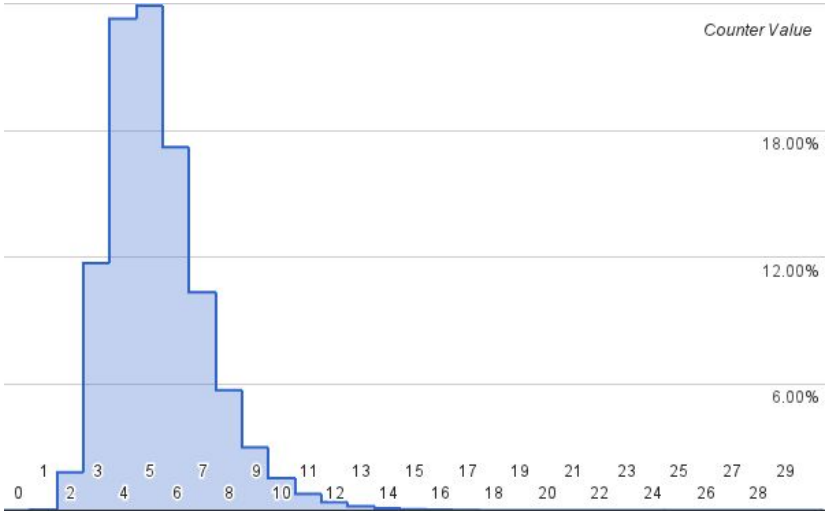
- Compressed sparse list
- Bias correction (between linear and HLL counting)

HLL++$_{\text{half-byte}}$ -- readability | cpu efficiency | accuracy

- Sparse list
- Half-byte counters

Half-byte Counters in HLL++

- Observation: counters tend to clump together



bucket	max trailing zero
0	4
1	8
2	12
3	7
4	5
5	5
6	7
7	6

→

bucket	max trailing zero
0	0
1	4
2	8
3	3
4	1
5	1
6	3
7	2

offset 4