ChatGPT

HyperLogLog: Counting Unique Items the Clever Way

Introduction

Counting the number of **unique** elements in a huge dataset can be surprisingly hard. A straightforward approach (e.g. keeping a set of seen IDs) needs memory roughly proportional to the number of distinct items. For instance, storing 1 billion 64-byte IDs could eat up ~64 GB of RAM [1]! For web giants like Google this might be feasible, but for most applications it's impractical. This is where **probabilistic** data structures come in. One such structure, **HyperLogLog (HLL)**, can estimate the number of distinct elements (the *cardinality*) using **tiny fixed memory** at the cost of a small error. In fact, HLL can count billions of distinct items with about **2% error using only 1.5 KB of memory** [2]. It's a brilliant trade-off for big data scenarios where an approximate answer is good enough.

Building Intuition: Coin Flips and Rare Streaks

Imagine a crowd of people each flipping a fair coin repeatedly until they get the first **heads**. They then report how many **tails** they got in a row before that first heads [3]. If only a few people do this, the longest streak of tails will likely be short (maybe 0, 1, or 2 tails). But if **millions** of people participate, it's quite likely *someone* will get an impressively long run of tails (perhaps 10, 15, or even more) [4]. In general, the more participants, the higher the chance that one of them hits a long tail streak. The **core idea**: observing a *rare* event (like many tails in a row) is a clue that you've seen a lot of trials/people [5]. A very long maximum run suggests a large number of participants.

Why is that? The probability of flipping, say, 5 tails in a row before a heads is $(1/2)^5 = 1/32$ (about 3.1%) [6] [7]. That is a pretty rare outcome for one person. But with dozens of people flipping coins, the *chance that somebody* hits 5 tails in a row increases. In fact, statistically, if ~32 people attempt this, you'd expect about one of them to get 5 tails in a row (since $1/32 \approx 3.1\%$ chance each) [7]. If the longest streak observed is 5, a reasonable guess is that on the order of $2^5 = 32$ people were flipping. Likewise, a longest run of 10 tails (probability $1/2^{10} = 1/1024$) hints at roughly $2^{10} \approx 1024$ participants, and so on [8] [9]. The rarer the pattern, the more trials are likely to have been run to see it.

Binary event series e with $Z(e)$ leading zeros	Longest series of leading zeros $Z(e)$	Leading zeros series probability $P(Z(e)) = (½)^{Z(e)}$	Estimated number of runs $N(e)_{est} = 2^{Z(e)}$
1 0 1 1 0 0 0 ...	0	$(½)^0 = 100\%$	$2^0 = 1$
0 1 0 0 1 0 1 ...	1	$(½)^1 = 50\%$	$2^1 = 2$
0 0 1 0 1 0 0 ...	2	$(½)^2 = 25\%$	$2^2 = 4$
0 0 0 1 1 0 0 ...	3	$(½)^3 = 12.5\%$	$2^3 = 8$
0 0 0 0 1 1 0 ...	4	$(½)^4 = 6.25\%$	$2^4 = 16$
0 0 0 0 0 1 0 ...	5	$(½)^5 = 3.125\%$	$2^5 = 32$
0 0 0 0 0 0 1 ...	6	$(½)^6 = 1.5625\%$	$2^6 = 64$
0 0 0 0 0 0 0 ...	7	$(½)^7 = 0.78125\%$	$2^7 = 128$

Empirical relationship between the longest run of "tails" observed and the number of people flipping. A run of L consecutive tails has probability $1/2^L$. So if you see a longest run of L=5 tails, you'd estimate on the order of $2^5=32$ people participated (because such a streak would appear only about once in 32 coin-flip sequences) [7].

In essence, **long streaks are the giveaway**. This coin-flip analogy underpins HyperLogLog. Each person flipping coins is like an item in our dataset, and a tail = 0, head = 1 sequence corresponds to a random binary string. If we see an item whose binary representation (we'll soon hash actual data into binary) starts with a long run of 0s (e.g. `0000001001...`), that's a *rare event*. Observing a 0-run of length *L* suggests on the order of 2^L unique items have been seen.

If you're already comfortable with this concept, feel free to skip ahead to the next section. If you'd like to see a quick simulation of this idea, read on for a brief experiment.

Quick Simulation (Coin Flips)

Let's simulate the coin flip experiment in Python to cement this intuition. In the code below, we have 1000 people flip a coin until heads and record the longest tails-run observed:

```
import random

def toss_until_heads():
    count = 0
    # Flip until a head (1) appears
    while random.random() < 0.5:  # 50% chance for tail
        count += 1
    return count

# Simulate 1,000 people flipping until heads
runs = [toss_until_heads() for _ in range(1000)]
print("Longest tail-run among 1000 people:", max(runs))
```

Running this a few times will yield varying results (because it's random), but typically the longest run for 1000 people might be around 9 or 10 tails in a row. For instance, one run might output:

```
Longest tail-run among 1000 people: 10
```

It's not a coincidence that 10 is close to $\log_2(1000) \approx 9.97$. In general, with `N` people, the longest tail-run length will be in the ballpark of $\log_2 N`. This simple experiment confirms our earlier reasoning: more participants ⇒ higher chance of a longer streak of tails.

From Coin Flips to Hashing: Making Data Look Random

How do we apply this idea to arbitrary data (like user IDs or URLs) instead of coins? We use a **hash function** to turn each item into a random-looking sequence of bits [10]. A good hash (e.g. SHA-1 or MurmurHash) will spread your data uniformly across 0/1 bit strings. Think of hashing as each data element generating its own private "coin flip sequence": by hashing the item, we get a binary string where each bit is like a fair coin flip. In this binary “fingerprint” [11], we can look at the prefix of the string as a series of coin tosses. For each item’s hash: - If it starts with `1` (binary `...`), that’s like getting heads immediately (0 tails in a row). - If it starts with `01...`, that corresponds to one tail then a head. - If it starts with `0001...`, that's three tails then a head, a rarer pattern. - If we ever found something that hashed to `000...0001...` with 20 zeros before a 1, that would be extremely rare (chance $1/2^{20}$) and would strongly suggest we’ve seen on the order of $2^{20} \approx 1,048,576$ distinct items to get such an unlikely hash [8].

Crucially, this works only if the hash function is **uniform** – every item should have an equal chance of yielding any particular bit pattern. If our data isn’t uniformly distributed (e.g. many IDs all start with `0000` in binary), hashing fixes that by randomizing the bits [12]. We assume from here on that our hash function is good, so each item’s hash is essentially a random binary number.

A Single-Counter Estimator (Flajolet-Martin Algorithm)

With the above insight, you might design a very simple distinct counter like this: as you stream through items, **track the longest run of leading zeros** you encounter in any item’s hash. If the longest leading-zero sequence seen is length *R*, output 2^R as the estimated number of unique items [13]. For example, if after processing all data the longest leading-zero prefix found was `00000` (L=5), we’d estimate about $2^5 = 32$ distinct elements [7]. If the longest prefix was `0000000000` (L=10), estimate ~1024, etc. This approach was essentially proposed in 1985 by Flajolet and Martin, pioneers of streaming algorithms.

We can sketch this approach in code form for clarity:

```
max_zero_run = 0
for item in data:
    x = hash(item)              # get a hash (int) for the item
    zeros = count_leading_zeros(x)
    max_zero_run = max(max_zero_run, zeros)
estimate = 2 ** max_zero_run    # estimated cardinality
```

This method uses only a few bytes of storage (just one number for `max_zero_run`!), making it **extremely memory-efficient**. However, it has a big drawback: it’s **too sensitive to luck** [14] [15]. If by chance the very

first item's hash starts with a long string of zeros, our estimator jumps to a wildly high value even though we've seen only one item. Conversely, you might process thousands of items yet never see a long zero-run purely by bad luck, leading to an underestimate. In other words, this single-counter estimator has a **high variance** [16] —one "unlucky" or "lucky" item can throw it off.

Interactive tip: If you feel you already grasp why a single longest-zero count is noisy, you can skip ahead. Otherwise, consider this: run the above process multiple times on the same dataset but using different hash functions (or simulate by random numbers). You'd likely get slightly different estimates each time, because which item happens to produce the longest zero prefix can vary. We need a way to **smooth out** that luck.

HyperLogLog: Many Registers for Accuracy

HyperLogLog improves accuracy by not putting all its eggs in one basket. Instead of one global counter for the longest run, HLL uses **many independent counters** and then combines them in a smart way [17] [18]. These counters are often called **registers** or buckets. The idea is to hash each item and use part of the hash to decide **which register** to update, and use the rest of the hash to determine the **zero-run length**. Each register will effectively get a random subset of the items, and will record the longest zero-prefix seen among *its* subset [19] [20].

Here's how HLL works step by step:

1. **Setup Registers:** Decide on *m* registers (say *m* = 1024, or 2^10). We need *m* to be a power of two for ease of computation. Give each register an index (0 to m–1) and initialize them all to 0 [17]. The register value will represent "the longest leading-zero run seen in this bucket so far."

2. **Process Each Item:** For each incoming item:

3. Compute a hash of the item (a 32-bit or 64-bit binary string).
4. **Split the hash:** Use the first **p** bits of the hash to choose a register (if *m = 2^p*, we need p bits for the index). There are $m = 2^p$ possible registers, and each item effectively picks one at random by its hash [19]. For example, if p=10 (m=1024) and an item's hash in binary starts with `0110 1010 10...`, that corresponds to register index `0110101010` (which is 0x**6AA** in hex, or 1706 in decimal).
5. **Count leading zeros:** Look at the remaining bits of the hash (after the first p bits). Count how many zeros are at the start of this remaining sequence until you hit a `1`. Let that count be *k* (we count *zero bits in a row* in the **value portion** of the hash). This *k* is our measure of how "rare" this item's hash was for that register [21] (longer zero-run = rarer pattern).

6. **Update the register:** If the observed *k* is larger than the current value in that register, update the register to *k*. Otherwise, leave it. Each register keeps the maximum zero-run length seen among items mapped to it [20].

7. **Combine the Registers:** After processing all items, we have *m* registers, each with a number (the max zero-run in that slice of the data). Now we need to combine these into one overall estimate. We **do not** simply take the maximum (that would just be back to the single-counter method) and a plain average of the register values isn't ideal either (because a few registers might have very high values

by luck) [18]. HLL uses a kind of **smart averaging** – specifically, a harmonic mean – to aggregate the registers [22]. In practice, the algorithm computes:

$$\displaystyle E = \alpha_m \cdot m^2 \cdot \left(\sum_{j=1}^{m} 2^{-M[j]}\right)^{-1},$$

where $M[j]$ is the value in register *j*, and α_m is a constant that depends on *m*. This formula is a mouthful, but intuitively it: (a) takes $2^{-M[j]}$ for each register (which is like an *estimate* of $1/\text{(items in that register)}$), (b) averages those, and (c) inverts that average, with a scaling factor α_m to correct bias [23]. The harmonic mean down-weights registers that had abnormally high values (since those contribute very small 2^{-M}) and gives more weight to the typical registers. The result E is our raw estimate of the number of distinct items.

1. **Apply Corrections:** The raw estimate E is very good for medium-to-large cardinalities. HLL includes some corrections at extreme ends [24] [25]:
2. If E is very small (relative to *m*), a lot of registers will be zero (never saw even a single 1 in their value bits). In this case, an alternative **linear counting** correction is used: roughly, if *V* registers are still 0, one can estimate $E \approx m \ln(m/V)$ (this gives a better estimate when $E \ll m$).
3. If E is extremely large (close to the range of the hash, e.g. approaching 2^{32} for a 32-bit hash space), then hash **collisions** become non-negligible. An optional correction can adjust for the fact that once you've seen *almost* every possible hash value, new items are more likely to collide than produce a new zero-run record. In practice, this matters only when you're counting tens of billions or more with a 32-bit hash. For most use cases, you won't hit this scenario if you choose a 64-bit hash.

The end result of all this is a single number, **E**, which is HLL's estimate of the distinct count.

Putting it All Together (Example)

To make this concrete, let's implement a *toy* HLL in Python and try it out on some data. We'll use a small number of registers for simplicity here:

```
import hashlib

# HLL parameters
p = 8                      # use 2^8 = 256 registers
m = 1 << p
registers = [0] * m

# Helper: count leading zeros in a 64-bit integer
def leading_zeros_64(x, bits=64):
    if x == 0:
        return bits  # if x is actually 0, we'll treat it as 64 zeros
    # find position of the highest set bit (0-indexed from LSB)
    # bit_length gives floor(log2(x)) + 1
    l = x.bit_length()
    return bits - l

# Process each item
```

```
for item in range(100_000):  # let's estimate the number of uniques in 0..99999
    # Hash the item (as bytes). We'll take 64 bits from an MD5 hash:
    h = hashlib.md5(str(item).encode()).digest()
    x = int.from_bytes(h[:8], 'big')      # 64-bit integer from hash
    idx = x & (m - 1)                     # use 8 bits for register index
    w = x >> p                            # remaining bits
    zeros = leading_zeros_64(w, 64 - p)
# count leading zeros in the remaining bits
    registers[idx] = max(registers[idx], zeros)

# Combine registers with harmonic mean
alpha_m = 0.7213/(1 + 1.079/m)  # appropriate constant for m >= 128
indicator = sum(2.0 ** -r for r in registers)
raw_E = alpha_m * (m ** 2) / indicator
print(f"Raw estimate E = {raw_E:.0f}")
```

If you run the above, you might get an output around:

```
Raw estimate E = 97_000
```

This is our HLL's estimate for 100,000 distinct integers. In one trial, it got about 97k, which is off by ~3%. If we increase *p* (more registers), the estimate tends to get closer. For example, with *p=10* (1024 registers), one trial gave `E ≈ 100,006` for the same data (practically spot on!), while *p=12* (4096 registers) yielded around 101,800 (about 1.8% high). This illustrates the key trade-off: **more registers = more accuracy, but more memory**. Formally, the standard error of HLL is about $1.04/\sqrt{m}$ [2] . So 256 registers (~1.5 kB) gives ~6.5% error; 1024 registers (~6 kB) gives ~3.25% error; 16384 registers (~96 kB) gives ~1% error, and so on.

Why harmonic mean? In our example, some registers saw longer runs than others. A plain average of register values might be pulled up by a few lucky high values. The harmonic mean (together with the calibrated constant α_m) ensures those outliers don't skew the result too much [22] . Essentially, each register provides an estimate of the form $2^{M[j]}$ for the number of items in its bucket, and the harmonic mean of these estimates (times a constant) gives a very stable overall estimate [23] .

HLL in Practice: Advantages and Use Cases

HyperLogLog's algorithm has some awesome properties for real-world use: - **Fixed, Tiny Memory Footprint:** You choose how much memory to use up front. For example, *m=2,048* registers (requiring only ~1.5 KB) is often enough to count billions of uniques with ~2% error [2] [26] . This memory usage stays **constant** no matter how many items you process! In contrast, exact methods would require memory growing with the data.
- **Speed:** HLL is fast. Each new item requires only a hash computation and a couple of bit operations to update a register. This is typically much faster than updating large data structures in memory or on disk.
- **Mergeable (Union-Friendly):** If you have two HLL sketches (say one for dataset A and one for dataset B), you can merge them to get the HLL for $A \cup B$ **without reprocessing all data**. Merging is as simple as taking

the *max* of each register from the two sketches (because each register independently tracked a subset) [27]. The result is an HLL that acts as if you had fed it all items from both sets. This property is hugely useful in distributed systems – you can count uniques on separate shards and combine the results easily. (Intersecting HLLs, on the other hand, is much trickier and approximate.) - **No Double Counting:** If the same item comes in multiple times, its hash will be the same and it will affect the same register with the same zero-run length. After the first time, subsequent additions of the exact same item won't increase any register value, so HLL naturally **de-duplicates** repeats. You don't have to explicitly check or store seen items – the algorithm inherently ignores duplicates. - **Adjustable Accuracy:** You can tune *p* (number of registers) to trade off between accuracy and memory. Need a tighter error margin? Use more registers. Can tolerate a looser estimate? Use fewer registers. The error bounds are well-understood (HLL is *probabilistic but predictable* in its error).

Common use cases: HLL is ideal when you need an approximate count of uniques and cannot afford to store or process all items exactly. Some examples: - *Database query optimization:* estimating the number of distinct values in a column (to plan query execution). Many databases (like Postgres, Redshift, etc.) use HLL or similar sketches under the hood for `COUNT(DISTINCT ...)` estimates.
- *Web analytics:* counting unique visitors, unique clicks, or unique search queries per day on a large-scale website [28]. Exact per-user tracking might be infeasible, but HLL can give a good estimate of "daily active users" using minimal memory.
- *Network monitoring:* counting distinct IP addresses hitting a service, or number of unique devices on a network [29].
- *Real-time streams:* in systems like Apache Flink/Beam or Kafka Streams, using HLL to keep approximate counts of unique events in sliding windows (e.g. how many unique users in the last hour) without storing every ID.

In fact, HLL is implemented in various platforms (e.g. Redis has a built-in HLL type called `PFCOUNT`, and BigQuery uses an HLL variant for `COUNT(DISTINCT)` approximations). Its ability to merge sketches makes it perfect for distributed computing scenarios.

Limitations

No tool is without trade-offs. Aside from the estimation error, a few things to note about HLL: - **No item retrieval:** HLL tells you *how many* unique items, but not *which* items. It's a **set cardinality** sketch, not a set itself. You can’t ask an HLL if it "contains" a specific element, nor can you extract the elements back. - **No deletion:** Once an item is added, you can’t remove it from the HLL sketch [30]. The data structure doesn’t store items or any reversible information (only aggregated bit patterns). So HLLs are best for accumulating counts (or unions) over data streams that only grow or for static datasets. If you need to handle deletions, other structures or a different approach is required. - **Intersection estimates:** Estimating the size of intersections (common elements between sets) is not straightforward with HLL. There are research approaches to approximate intersections by combining HLLs, but accuracy can degrade [31]. In practice, if intersection sizes are needed, one might use other methods or accept a higher error.

Despite these, for the task of *“How many distinct elements have we seen?”* in a big data context, HyperLogLog is a gold standard solution.

Summary & Key Takeaways

- **HLL is an *approximate* cardinality counter.** It won't give an exact distinct count, but with proper tuning it gets *very close* (e.g. ~2% error or better) [2] [32]. In exchange, it uses **dramatically less memory** than exact methods.
- **Probability meets practice:** The technique exploits the statistics of rare events (long streak of zeros in hashes ≈ lots of distinct items) [5] [33]. By averaging across many independent subsets (registers), it achieves a stable estimate — leveraging the law of large numbers to cancel out "luck" [18] [23]. Understanding this core intuition is key to seeing why HLL works so well.
- **Fixed tiny size:** The memory footprint is fixed **ahead of time** (you choose *m* registers). It doesn't grow with the number of items counted [26]. This makes HLL great for streaming and big data systems where you need predictable memory usage.
- **Fast and scalable:** Each item processed in O(1) time (a hash + index/update). HLL sketches can be **merged** by taking per-register maxima, enabling easy parallel or distributed counting [27].
- **Widely used:** HLL (and variants like HLL++) are used in modern databases, analytics engines, and network systems for computing unique counts on the fly.

HyperLogLog is a brilliant example of a **probabilistic algorithm** that turns a theoretical insight into a practical tool. It trades a little precision for massive gains in efficiency, allowing us to answer questions like "how many unique users visited?" on datasets so large that exact counting is impractical or impossible [34] [35]. By understanding coin flips and rare patterns, we built up to how HLL works step by step. Hopefully, this explainer demystified HLL's inner workings and showed why it's such a powerful technique in the era of big data. Happy counting!

---

[1] [3] [4] [5] [11] [14] [15] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [32] [33] [34] [35] HyperLogLog Explained | AI Transformation Leader | Enterprise Architect
https://prasadbhamidipati.in/hyperloglog-explained

[2] [10] [13] [16] HyperLogLog - Wikipedia
https://en.wikipedia.org/wiki/HyperLogLog

[6] [7] [12] [30] [31] An Introduction to HyperLogLog - Geography & Coding
https://geo.rocks/post/hyperloglog/

[8] [9] HyperLogLog-orrhea: how do HyperLogsLogs work?
https://will-keleher.com/posts/HyperLogLog-orrhea.html