Cardinality Estimation of Distinct Items - An Overview

Sebastian Balsam

December 1, 2025

Abstract

In this paper I want to give an overview of different solutions for the Count Distinct Problem...Add more content.

1 Introduction

The cardinality estimation problem or count-distinct problem is about finding the number of distinct elements in a data stream with repeated elements. These elements could be URL's, unique users, sensor data or IP addresses passing through a data stream. A simple solution would be to use a list and add an item to this list each time we encounter an unseen item. With this approach we can give an exact answer to the query, how many distinct elements have been seen. Unfortunately if millions of distinct elements are present in a data stream, with this approach, we will soon hit storage boundaries and the performance will deteriorate.

As we often do not need an exact answer, streaming algorithms have been developed, that give an approximation that is mostly good enough and bound to a fixed storage size. There different approaches to solve this problem. Sampling techniques are used as an estimate by sampling a subset of items in the set and use the subset as estimator, e.g. the CVM algorithm by Chakraborty et al. [Chakraborty et al., 2023]. In its simplest form if we take a sample size of N_0 for a set of size N, we get an estimate by counting the distinct items times N/N_0. While sampling methods generally give a good estimate, they can fail in certain situations. If rare elements are not in the sampled set, they would not be counted and we would underestimate. Another problem is replicating structures in the data. If we sample every 10th item, and accidentally every 10th item is different, while other items are mostly the same, we would overestimate. For database optimization, machine learning techniques have been used recently for cardinality estimation [Liu et al., 2015, Woltmann et al., 2019, Schwabe and Acosta, 2024].

The technique I want to focus on in this paper are stochastic sketch techniques that implement a certain data structure and an accompanying algorithm to store information efficient for cardinality estimation. I will follow the development chronologically from the first algorithm of Flajolet and Martin in 1985 [Flajolet and Martin, 1985], to the HyperLogLog algorithm in 2007 and the HyperLogLog++ algorighm in 2013 to the HLL-TailCut algorithm from 2023.

TODO: Give an overview of the used techniques. Tell what I want to say and what I leave out.

2 Flajolet and Martin

Before the advent of the internet there were not many data stream applications as we have today. But databases existed and began to grow in size. As table sizes grew, evaluation strategies became important, how to handle such big data tables for join operations. Query optimizers were developed that could find the optimal strategy.

In this context Flajolet and Martin developed in 1985 a method to estimate the number of distinct items in a set in a space efficient way[Flajolet and Martin, 1985]. The idea behind the method is to use a hash function that maps n items uniformly to integers. If we look at the registerary representation of these integers, and check the longest runs of zeros, in about $n/2$ cases, we have a '0' at any position. In about $n/4$ cases, we have two '0's consecutively. With a chance of $1/8$ we have three '0' in a row, and so on. That means if we start for example at the right and count the zeros for each element in the stream, based on maximal number we have seen so far, we can estimate the number of items. When we add an item, we use the algorithm as described in Algorithm 1 to add set the position of the rightmost '1' in a bitmap.

Flajolet and Martin found that they get better result, if an average of multiple (m) registers of bitmaps are used. This is equivalent with running an experiment multiple times to get results closer to the expected value. With a higher number of registers available for the average, the estimation quality raises, but at the same time more memory to store the bitmap is needed. They use the hash value to decide the register or register (by using the modulo operation) and save the rightmost '1' of the registerary representation in this register.

Algorithm 1 Adding an item in the Flajolet-Martin algorithm.

```
m ← Number of registers
function φ(val)
    return position of the first 1 bit in val.
end function
function ADDITEM(x)
    hashedx ← hash(x)
    α ← hashedx mod m
    index ← φ(hashedx div m)
    BITMAP[α][index] = 1
end function
```

When an estimate is needed, an average of the leftmost zeros over all registers are used, as shown in algorithm 2. The

Algorithm 2 Get an estimate in the Flajolet-Martin algorithm.

```
1: m ← Number of registers
2: function QUERY
3:     ρ ← 0.77351, S ← 0
4:     max ← 32                                  ▷ 32 bit per register
5:     for i := 0 to m − 1 do
6:         j ← 0
7:         while BITMAP[i][j] == 1 and j < max do
8:             j ← j + 1
9:         end while
10:        S ← S + j
11:    end for
12:    return int(m/ρ · 2^{S/m})
13: end function
```

magic number $\rho = 0.77351$ is used, as the expected Value of R - the position of leftmost zero in the bitmap is

$$\mathbb{E}(R) \approx \log_2 \rho n$$

We sum up the registers and as each register only counted $1/m$ items, we get an estimate with:

$$E = \frac{1}{\rho} m \cdot 2^{\frac{S}{m}}.$$

The results for this algorithm are shown in Figure 1. The memory consumption for the algorithm is $m \cdot 32$. That means with $m = 256$, we can produce estimates with about 5% standard error for higher cardinalities and have to use $256 \cdot 32 = 8192$ bit $=$ 1kB of memory to safely count up to 100 million items, before the quality decreases.

Figure 1 shows the results of my implementation of the FM algorithm of a cardinality up to one million items for different values of m. We can see that the algorithm results in strong overestimation below 1500 items for $m = 256$. After that, we only have a deviation on average of about 1 percent of the true number of items. This makes sense, since we use 256 registers, each new item is only stored in one of the registers, and many would remain empty in the beginning. With $256/0.77 = 332$ and $s^{n/265} = 1$ for low n we overestimate in the beginning. The solution for numbers below 1500 items is simply to count these items exactly in an array, and only use Flajolet & Martin above a certain number.

The value of m - the number of registers is important for accuracy. Even though we have more overestimation for less items counted, we get a better estimation for a higher number of items. With $m = 265$, Flajolet and Marin report a bias of 0.0073 and a standard error of 4.65%. For $m = 16$, we have a bias of 1.0104, about the same, but a standard error of 19.63%. These numbers match my own experiments. Generally, as estimated later in [Durand and Flajolet, 2003], the standard error is about $1.3/\sqrt{m}$.

3 HyperLogLog

A first refinement of the Flajolet & Martin algorithm came in 2003 by Durand and Flajolet [Durand and Flajolet, 2003]. Already in this paper the memory usage and the standard error could be reduced. They introduced a truncation rule to only use 70% of the smallest values in the registers, since they found that the arithmetic mean would often overestimate. Later, in 2007 Flajolet [Flajolet et al., 2007] refined the algorithm even more to improve the accuracy even further to an algorithm called HyperLogLog (HLL).

When we try to estimate the number of items, in the FM85 algorithm, we look for the maximum number of '1' in the bit registers. All the other '1' before and any bits after this position are not used. So instead of saving the complete bit register, it should be enough to just save the maximum number. In addition, since we have a maximum of 32 bits originally in the register, we only need 5 bits ($2^5 = 32$) for this number to be stored for each register. This is an additional memory reduction. Instead of 1kB of memory as in the original FM85 algorithm, the HLL algorithm only need 160 bytes.

Algorithm 3 Adding an item in the HyperLogLog algorithm.

```
1: m ← Number of registers
2: function φ(val)
3:     return position of the first 1 bit in val.
4: end function
5: function ADDITEM(x)
6:     hashedx ← hash(x)
7:     j ← hashedx mod m
8:     w ← φ(hashedx div m)
9:     M[j] := max(M[j], φ(w))
10: end function
```

Algorithm 3 shows my implementation of the algorithm to add an item. The original version used the first b bits with $m = 2^b$ to identify the register and used the remaining bits to evaluate the maximum numbers of zeros. As my implementation is done in python and integer values don't have a fixed size, I used the modulo operation (the last b bits) to identify the register index and div m (the equivalent of a bit shift 2^m) as the remaining value to get the position of the first 1-bit. This should not change the outcome, and as our results are comparable with the results of the HyperLogLog paper, this should not be a problem.

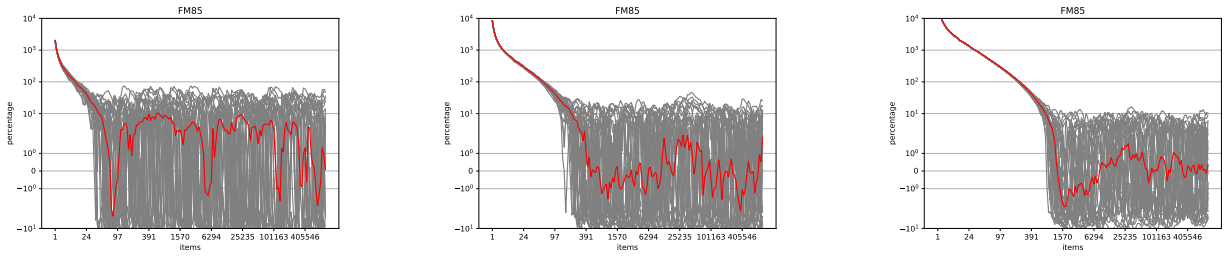
As the original algorithm would overestimate when calculating the average over all registers. Only using the lowest 70% was one method to improve the results. For HLL, Flajolet used the harmonic mean given by

$$H = \frac{n}{\sum_{j=i}^{n} \frac{1}{x_i}}$$

instead of the arithmetic mean to get the average of the registers against overestimation, as the harmonic mean gives lower results. The raw estimate (before corrections) is calculated with

$$E = \alpha_m m^2 \cdot (\sum_{i=1}^{m} 2^{-M[i]})^{-1}$$

Since each register will have n/m elements, when n is the number of counted elements so far, with the harmonic mean



(a) m=16 (b) m=64 (c) m=265

Figure 1: Results for the Flajolet Martin algorithm. The percentage of deviation from the true number of items over 30 runs (in gray) are shown (m=265) up to one million items. The red line shows the average over the 30 runs.

of these registers mH would be about n/m. Therefore, m^2H should be n. Again a constant α is used to correct the bias due to hash collisions.

In addition, Flajolet introduced corrections for very low numbers where we would not have enough data for good predictions and would see overestimations as in the original FM algorithm. If the estimate E is below $\frac{5}{2}m$, they use linear counting and get a correction of

$$E^* = m \log(\frac{m}{V})$$

with V as the number of registers equal to 0. Instead of using the number of bits calculated by the harmonic mean over the registers, we estimate by counting the numer of empty registers. For very high numbers, where hash collisions are more common they give another correction based on the probability of hash collisions.

Figure 2 shows the results for one million items, and we can see, that the results are good, even for few items.

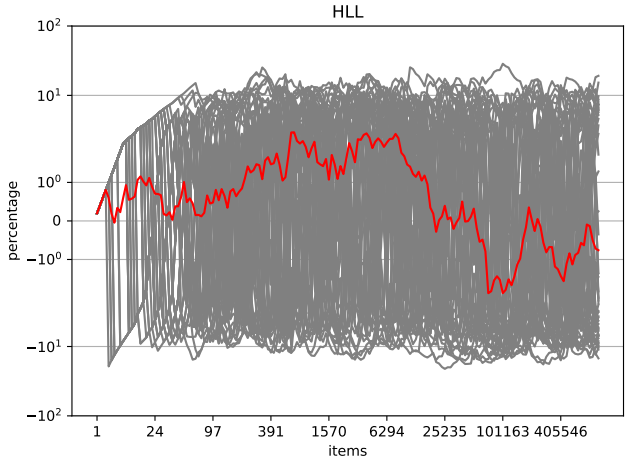


Figure 2: The results of the error in percentage of one million items for the HyperLogLog algorithm over 30 independent runs for $m = 256$.

Overall the quality of HLL exceeds the FM algorithm both in space and estimation quality and has become the standard in many applications [Redis, 2025, Dragonfly, 2025, Amazon, 2020]. It is memory efficient, is fast, as the slowest part is the hashing function, scalable up to 10^9 without loss of quality, simple to implement and has a standard error of only $1.04/\sqrt{m}$.

Originally developed for use in databases, both FM85 and HLL have become more interesting for streaming applications too. [Scheuermann and Mauve, 2007] propose a lossless compression scheme with arithmetic coding that produces results close to the theoretical optimum to perform distributed data aggregations over networks.

4 HyperLogLog++

The next improvement came in 2013 by [Heule et al., 2013] with HyperLogLog++ (HLL++). In their paper they use a 64-bit hash function, that reduces hash collisions for lower cardinalities. Instead of using 5 bits to store the maximum '1's in registers, they use 6 bits. With this change they are able to securely estimate cardinalities up to $2^{64} = 1.8 \cdot 10^{19}$. Even though this is a number that seldom should be met in real life, they propose to add an extra bit even more, if needed, as memory is cheap compared to the amount of items at this cardinality.

With this higher number of storage, since hash collisions are not a problem anymore the correction for large cardinalities used in HLL, is not needed anymore. For small cardinalities they keep the linear counting correction of HLL.

Another improvement of HLL++ is the use of Bias correction. They show that the transition between linear counting of small values and harmonic mean estimation of higher values is not flawless. Figure 3 shows a spike at the transition. This spike is not visible for all values of m, but for some, it might be a problem. Instead, they used 200 cardinalities as interpolation points get an estimate for bias correction values by using k-nearest neighbor interpolation. This not only reduced the spikes at the transition points between linear counting and the harmonic mean estimation, but gives better general results also for higher cardinalities. I have to point out, that only the bias will be corrected, the standard error for HLL++ and HLL is about the same (except for the spike reduction).

Since 5 bits are used to store the data, Heule et al. saves

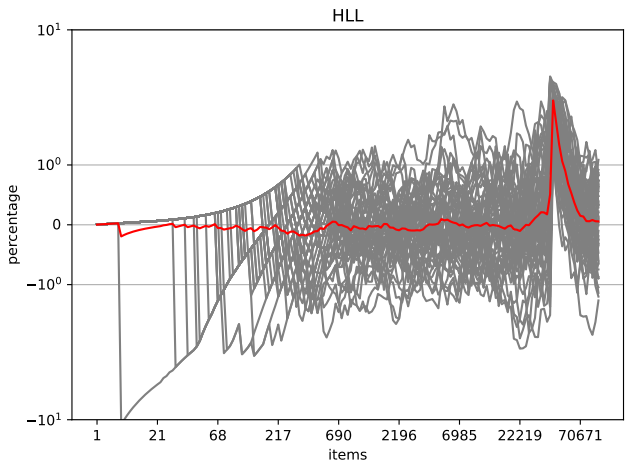


Figure 3: Spike at the transition between linear counting and harmonic mean estimation in HLL for $m = 16380$.

the number of '1's with a sparse representation as long as the size of the sparse representation is smaller than $6 \cdot m$ bits. In this way, they can keep the size requirements still small, in fact, as long as $n < m$, HLL++ requires significantly less memory than HLL. For higher cardinalities this is of course not true anymore.

To summarize HLL++, they use 6-bit instead of 5 bits, but a sparse representation for small cardinalities to save memory. Using 6-bits leads to better estimations of higher cardinalities and empirical bias correction to improve the estimation even more.

5 LogLogBeta, 2016

6 ExtendedHyperLogLog, 2023

https://arxiv.org/pdf/2106.06525

7 HLL-TailCut, 2023

https://topoer-seu.github.io/PersonalPage/csqjxiao_files/papers/INFOCOM17.pdf

8 Conclusion

Table 1: Table showing the different methods...

Algorithm	Memory	Comment
FM85	23	clever
HLL	45	here
HLL++	23	23
ExHLL	35	fsd
HLL-TailCut	23	sdf

References

[Amazon, 2020] Amazon (2020). Amazon redshift announces support for hyperloglog sketches.

[Chakraborty et al., 2023] Chakraborty, Vinodchandran, and Meel (2023). Distinct elements in streams: An algorithm for the (text) book.

[Dragonfly, 2025] Dragonfly (2025). Dragonfly db docs.

[Durand and Flajolet, 2003] Durand and Flajolet (2003). Loglog counting of large cardinalities.

[Flajolet et al., 2007] Flajolet, Fusy, Gandouet, and Meunier (2007). Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm.

[Flajolet and Martin, 1985] Flajolet, P. and Martin, G. N. (1985). Probabilistic counting algorithms for data base applications.

[Heule et al., 2013] Heule, Nunkesser, and Hall (2013). Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm.

[Liu et al., 2015] Liu, Xu, Yu, Covinelli, and ZuZarte (2015). Cardinality estimation using neural networks.

[Redis, 2025] Redis (2025). Hyperloglog docs.

[Scheuermann and Mauve, 2007] Scheuermann and Mauve (2007). Near-optimal compression of probabilistic counting sketches for networking applications.

[Schwabe and Acosta, 2024] Schwabe and Acosta (2024). Cardinality estimation over knowledge graphs with embeddings and graph neural networks.

[Woltmann et al., 2019] Woltmann, Hartmann, Thiele, Habich, and Lehner (2019). Cardinality estimation with local deep learning models.