Cardinality Estimation of Distinct Items - A Review

Sebastian Balsam

December 15, 2025

Abstract

In this paper I want to give an overview of different solutions for the Count Distinct Problem. I will follow the development from the first algorithm by Flajolet and Martin, to the HyperLogLog algorithm and further optimizations to the Extended HyperLogLog algorithm, to increase the estimation quality while at the same time decrease the memory consumption of these algorithms. I have implemented most of the described algorithms to confirm the algorithms estimations and compare their results. This paper shows the development of the algorithm over the last 40 years with the main changes and improvements.

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 2017, the martingale setting of HyperLogLog in 2017 and the ExtendedHyperLogLog in 2023. All of the discussed methods are based on the original method by Flajolet and Martin and we can follow the different problems and solutions researchers have found to improve the algorithm further and further. From a science history perspective, this paper presents in a chronological order all the improvements that have been madeo, often with surprising ideas. There are other methods that solve the given problem, as mentioned before, but I want to focus specifically on the history of improvements of the Flajolet & Martin algorithm.

I will omit any mathematical proofs, as they can be found in the original sources, but I will specify any equations that are necessary for understanding the algorithm. Most of the algorithms are implemented by myself in the accompanying repository [Balsam, 2025] and have been used to verify and confirm the problems and results of the algorithms.

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]. Often this method is called Probabilistic Counting with Stochastic Averaging (PCSA) in the literature, I will refer to it as FM85. The idea behind the method is to use a hash function that maps n items uniformly to integers. If we look at the binary 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 (by using the modulo operation) and save the rightmost '1' of the binary representation in this register.

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

```
 1: $m \leftarrow$ Number of registers
 2: function $\phi(val)$
 3:     return position of the first 1 bit in val.
 4: end function
 5: function ADDITEM($x$)
 6:     $hashedx \leftarrow hash(x)$
 7:     $\alpha \leftarrow hashedx$ mod $m$
 8:     $index \leftarrow \phi(hashedx$ div $m)$
 9:     $BITMAP[\alpha][index] = 1$
10: end function
```

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

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

When an estimate is needed, an average of the leftmost zeros over all registers are used, as shown in algorithm 2. The 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 $= 1$kB 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

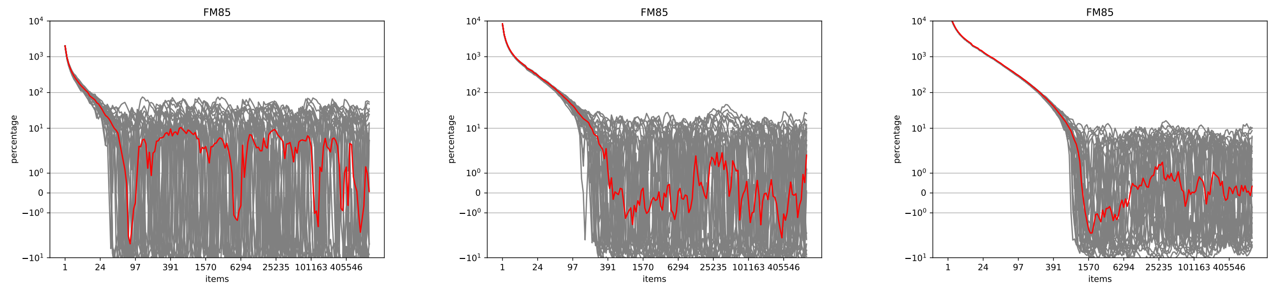
A first refinement of the Flajolet & Martin algorithm came in 2003 by Durand and Flajolet [Durand and Flajolet, 2003]. Already in this paper the memory usage and the standard error could be reduced. They introduced a truncation rule to only use 70% of the smallest values in the registers, since they 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 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: $m \leftarrow$ Number of registers
 2: function $\phi(val)$
 3:     return position of the first 1 bit in val.
 4: end function
 5: function ADDITEM($x$)
 6:     $hashedx \leftarrow hash(x)$
 7:     $j \leftarrow hashedx$ mod $m$
 8:     $w \leftarrow \phi(hashedx$ div $m)$
 9:     $M[j] := max(M[j], \phi(w))$
10: end function
```

Algorithm 3 shows my implementation of the algorithm



(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.

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

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

Since each register will have n/m elements, when n is the number of counted elements so far, with the harmonic mean of these registers mH would be about n/m. Therefore, $m^2 H$ 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 as shown in Figure 2. If the estimate E is below $\frac{5}{2}m$, they use linear counting and get a correction of

$$E^* = m \log(\frac{m}{V}) \tag{2}$$

with V being 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 number of empty registers. As most of the registers are empty anyway

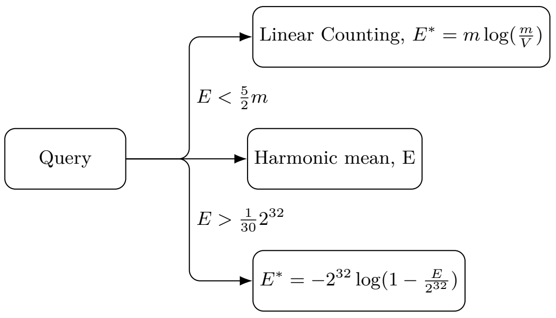

Query
Linear Counting, $E^* = m\log(\frac{m}{V})$
$E < \frac{5}{2}m$
Harmonic mean, E
$E > \frac{1}{30}2^{32}$
$E^* = -2^{32}\log(1 - \frac{E}{2^{32}})$

Figure 2: When an estimation for HLL is queried, corrections are made for very small and very large cardinalities, while most of the estimates are generated by the harmonic mean calculation.

this gives a result, much closer to the true value of the number of items For very high numbers, where hash collisions are more common, they give another correction, based on the probability of hash collisions.

Figure 3 shows the results for one million items, and we can see, that the results are good, even for few items.

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

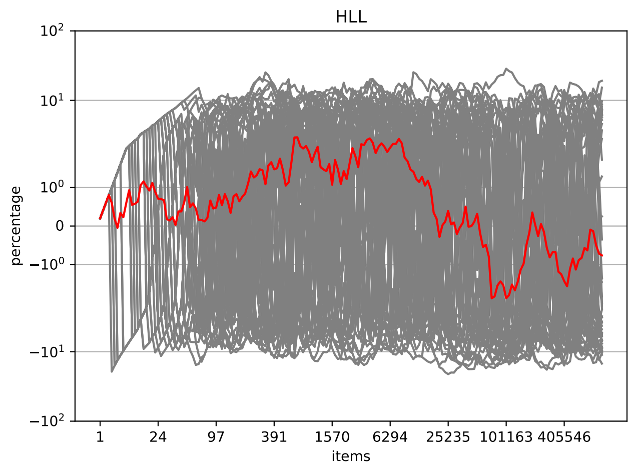


Figure 3: The results of the error in percentage of one million items for the HyperLogLog algorithm over 30 independent runs for $m = 256$. By using linear counting for lower cardinalities, the problem of FM85 in this area is solved.

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 also not needed anymore. For small cardinalities below $\frac{5}{2}m$, they keep the linear counting correction of HLL as in equation (2).

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 5a 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 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, since we can not use sparse representations.

To summarize HLL++, they use 6-bit instead of 5 bits, but a sparse representation for small cardinalities to save memory. The process of counting a new item is otherwise unchanged and done, as proposed in the HLL algorithm. Using 6-bits leads to better estimations of higher cardinalities, so a correction for high cardinalities is not needed anymore. With empirical bias corrections they improve the estimation by adjusting the bias and prevent spikes.

5 LogLogBeta

In 2016, [Qin et al., 2016] used the results of HLL++ to improve the algorithm even further. While HLL++ still needed linear counting for small cardinalities and a bias correction to handle the transition between linear counting and the usual harmonic mean calculation, LogLog-β (LogLogBeta) simplifies this process to one function for the query operation. Otherwise, LogLogBeta also uses 6 bits and a sparse representation as HLL++ does.

The modified version LogLogBeta proposes uses a function of m and z (the number of empty registers), that also uses precalculated coefficients for different values of m, that had been found empirically to correct the bias. The update of the registers, when new items are counted is still done in the same way as proposed for HLL and HLL++, but the query for the number of items is now simplified to:

$$E = \frac{\alpha_m M(m-z)}{\beta(m,z) + \sum_{i=0}^{m-1} 2^{-M[i]}}$$

With $z = 0$, $\beta(m, 0) = 0$ and this formula is the same as Flajolet used in the HLL algorithm equation (1), so the changes apply only for situation where some of the registers are 0. This is therefore a simplification and correction for small cardinalities. The calculation for a given cardinality c for $\beta(m, z)$ is given as an approximation of

$$\beta(m,z) = \frac{\alpha_m m(m-z)}{c} - \sum_{i=0}^{m-1} 2^{-M[i]}$$

by calculating

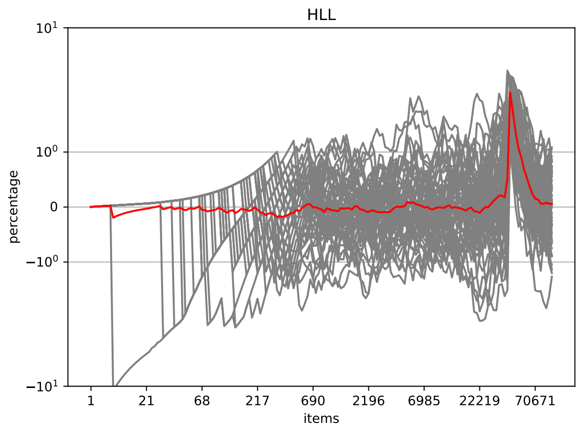
$$\beta(m,z) = \beta_0(m)z + \beta_1(m)z_l + \ldots + \beta_k(m)z_l^k$$

with $z_l = \log(z+1)$. Here β_k can be precalculated for different values for m, which makes this calculation very efficient.

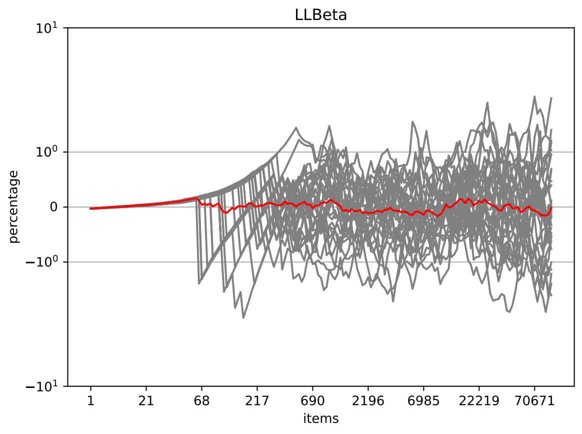
As we can see in Figure 5b, the spike at the transition between linear counting and the harmonic mean estimation is no longer visible. In comparison to HLL++ the LogLogBeta algorithm might not be an improvement of estimation quality, but is a great simplification of the algorithm and also solves the problem of the transition between linear counting and harmonic mean estimation.

6 HLL-TailCut

Already in [Durand and Flajolet, 2003], Flajolet only used the lowest 70% of the registers. This idea has been taken up by [Xuao et al., 2017]. If we take a look at the histogram



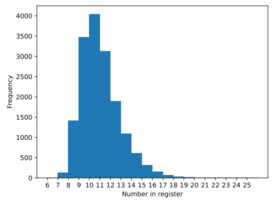
(a) HLL with spike at transition.



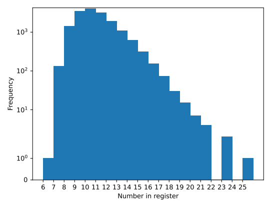
(b) LogLogBeta without a spike.

Figure 4: Figure (a) shows a spike at the transition between linear counting and harmonic mean estimation in HLL for $m = 16384 = 2^{14}$ at $\frac{2}{5}2^{14}$. With the HLL++ and LogLogBeta algorithm, this spike is eliminated as shown in figure (b).

of the numbers of values in the saved registers as shown in Figure **??**, we can see that the histogram has a long tail to the right, with few registers of high values. These registers are not contributing very much to the result in the calculation of the mean. But at the same time, they take precious memory. The general idea is to cut registers with higher values, which can save memory. If we only store registers up to a size of 16, we only need 4 bits.



(a) Histogram



(b) Histogram with log scale

Figure 5: This histogram shows the frequency of the numbers in the registers for LLBeta with $m = 2^{16}$ for 10 Million items. Even though, they are not visible in the left figure, there are values up to 28, although with a very low frequency, as shown in the log scaled histogram on the right. That means we have a long tail to the right side with registers of little information as shown on the right with the log scale.

Registers with a maximum value of 16, can count up to $2^{16} = 65536$. But since we distribute the counted items on m registers, we have $m \cdot 2^{16}$. As m is bound to a given estimation error, and we would like to minimize this error, values for $m = 2^{14}$ are quite common to get an estimation error of about 0.4%. This means we can store up to $2^{14} \cdot 2^{16} = 10^9$ items, although we get wrong results already before due to hash collisions. Nevertheless, this should be enough for most applications.

Xuao et al. went further and could truncate the size to even $3 \cdot m$ bits by using a base register B below the truncated registers M, as most of the registers would share the same base register with increasing items that are counted. They claimed they could reduce the error to $1.0/\sqrt{m}$, but [Pettie and Wang, 2020] showed, that this is unfortunately not true, the error is constant and independent of m. But [The Apache Foundation, 2019] has an implementation which uses 4 bits. The method of tail cutting works and can reduce the amount needed for memory with 20% (from 5 bits to 4 bits).

7 Mergeble sketches and Martingale settings

In 2007, Daniel Ting [Ting, 2014] found another method for better predictions. If the different items are presented as a stream, we can view this stream as a Markov process with martingale properties, that means the expected value of the next observation is equal to the most recent value. We can now use the sketch as a predictor with Markov probabilities. The changes in the algorithm are minimal. Every time the sketch is updated, we calculate the probability with

$$q(S) = \frac{1}{m}\sum_{i=1}^{m} 2^{-S[i]}$$

and we can update the martingale Estimator N with

$$N_{new} = N + \frac{1}{q(S)}.$$

The idea behind this is, that the sketch in its consecutive updates forms a Markov process and this series of updates contains information that would be thrown away in a mergeable setting. But if we follow the series, we can use it to derive an estimator from the process as a whole. This is the martingale estimator, that is updated in each step for a new unseen item.

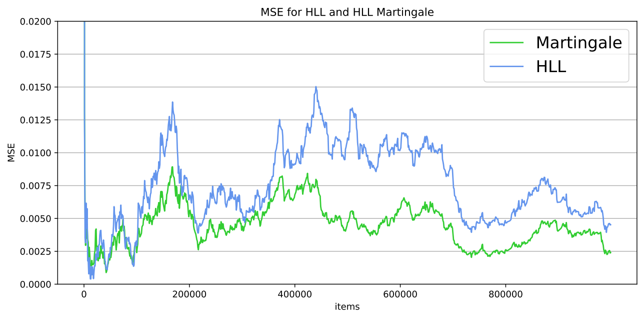


Figure 6: Mean squared error of HLL and HLL in martingale setting for cardinalities up to 10^6 and $m = 2^8$ The average MSE for EHLL is 0.0056, the avg. MSE for HLL is 0.0086.

Figure 6 shows the result for my implementation of the HLL sketch in martingale setting, and we can see that this estimation is much better than the plan HLL sketch. The plain HLL sketch needs about 1.5 more bits than the martingale version for the same estimation quality. But there is one catch to this. Since each estimate depends on the prior result, the sketch can not be run in parallel in multiple threads. We do not need any additional memory for the martingale enhancement of the sketch, as we only need to update the value of the estimator, each time we update the sketch. There is only one additional float value to store.

All other sketches that have been presented so far are mergeable sketches, that means we can count items in different threads or on different locations and can merge the result, simply by merging the registers and get a new estimation. In a martingale setting, this is no longer possible. But for most streaming applications, that would not be a problem, as long as the algorithm can follow the entire stream and we don't process different sources on multiple locations. The martingale setting is therefore a big improvement for this algorithm.

8 Using Additional Bits

Another new development came in 2021 when Tal Ohayon[Ohayon, 2021] published ExtendedHyperLogLoc (ExHLL). His method can be seen as a compromise between FM85 and HLL. Ohayon saves the highest '1' exactly as in HLL. But in addition, we also save in an additional bit for each register if the update with the highest '1' was more than a single increment. In the FM85 algorithm, we counted the number of consecutive highest values. That means a FM85 register of '111000' had the same result as '1110010'. Here the first three '1' would be used for estimation. But HLL used the highest '1' in the register. In our example, the values for estimation would be different for both registers - 3 and 6. ExHLL uses both, the maximum '1', and the information if the bit below the maximum '1' would also be a '1' or a '0'. This gives additional information that can improve the estimation quality.

Of course, since we have to use an additional bit per register, we have to use 6 bits instead 5 bits, that were used for HLL. Algorithm 4 shows the algorithm if we want to add an item to the sketch. Here we have an array C_1 of size m as we had in HLL, but in addition we keep track of the bit before the leading zero with the bitmap C_2.

Algorithm 4 Adding an item in the Extended HyperLogLog algorithm.

```
1: $m \leftarrow$ Number of registers
2: $C_1$ array of size $m = 2^b$ initialized with zeros
3: $C_2$ bitmap array of size $m = 2^b$ initialized with ones
4: $\phi(val)$ returns the position of the first 1 bit in val
5: function ADDITEM($x$)
6:     $hashedx \leftarrow hash(x)$
7:     $j \leftarrow hashedx \bmod m$
8:     $y \leftarrow \phi(hashedx \text{ div } m)$
9:     if $\phi(y) = C_1[j] + 1$ then
10:        $C_1[j] \leftarrow \phi(y)$
11:        $C_2[j] \leftarrow 1$
12:    else if $\phi(y) > C_1[j] + 1$ then
13:        $C_1[j] \leftarrow \phi(y)$
14:        $C_2[j] \leftarrow 0$
15:    else if $\phi(y) = C_1[j] - 1$ and $C_2[j] = 0$ then
16:        $C_2[j] \leftarrow 1$
17:    end if
18: end function
```

When the sketch is queried, we use the formula:

$$E = \gamma m^2 \cdot \left(\sum_{j=1}^{m} 2^{-C_1[j]} + (1 - C_2[j]) 2^{-C_1[j]+1}) \right)^{-1}$$

The first part is the same as equation (1), that was used in HLL. But now we use the additional $(1 - C_2[j])2^{-C_1[j]+1}$. If $C_2[j] = 1$, we get the same calculation as in HLL, as the second term becomes 0. But if $C_2[j] = 0$, we add the probability $2^{-C_1[j]+1}$ that $C_1[j] - 1$ would be used to generate the predicted value. Since we use probabilities for $C_[j]$ and possibly $C_1[j] - 1$, the correction α we used for HLL is no longer valid and the authors analyze the Expectation and variance to get the correction constant γ_m.

This algorithm is an improvement to the standard HLL, but even though the quality is much better, we have to be aware of the fact, that we still need linear counting for lower cardinalities and a correction for very large cardinalities as Flajolet used in HLL. We also still see here the spike at the transition between linear counting and the LogLog counting, that can be corrected with the methods described in HLL++ and HLLBeta.

Figure 7 shows the MSE of EHLL and HLL with values for m chosen in a way, that both methods would use the same memory ($m_{EHLL} = 6/5 m_{HLL}$). The figure shows that we get a lower error for EHLL than for HLL with the same

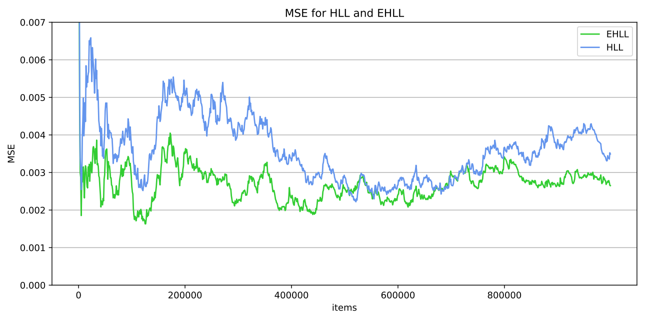


Figure 7: Mean squared error of HLL and EHLL for cardinalities up to 10^6 and $m = 2^8$ The average MSE for EHLL is 0.0037, the avg. MSE for HLL is 0.0046.

memory usage. The difference of these results are valid even in a martingale setting as Ohayon can show.

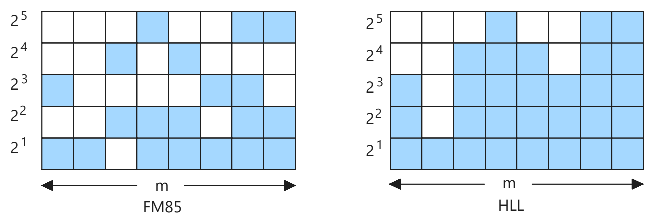


Figure 8: The left side shows the registers vertically that are used to store the sketch for FM85. FM85 only uses the lowest consecutive marked areas. The right side shows the same situation for HLL. As it only stores the maximum values, any information on lower bits are lost.

[Wang and Pettie, 2023] expanding an idea of [Ting, 2014] of viewing the sketch as a dartboard, where we would count the number of free cells of the bitmap sketch (Generalized Remaining Area - *GRA*). While FM85 stored the complete bitmap, it only used the lower marked bits and would give an estimate based on the number of consecutive lower bits as shown in Figure 8. HLL only used the maximum value for estimation to save some memory, but useful information is lost in this sketch, as we would ignore some of the empty spaces in the bitmap. Using less memory meant, we could use more registers for higher accuracy. But using additional bits below the maximum bit would be useful, even though it would also mean to need extra memory. The problem is, to find the optimal tradeoff.

Otmar Ertl builds upon this idea in [Ertl, 2024, Ertl, 2025]. In HLL we only updated the maximal rightmost '1' in the sketch. EHLL saves in addition the bit left of the rightmost '1' ($d = 1$), one bit below the maximum value. Ertl generalized this idea further. If we save the 2 bits left of the rightmost '1', we have $d = 2$. The original FM85 algorithm, that saved all bits used $d = \infty$ even though this would be the theoretical limit as only 32 bits were saved as a maximum. Ertl compares the memory usage of d with the variance of the error to find the optimal values for the memory-variance-product (MVP). $d = 2$ can be proven to be optimal to get the best ratio between memory efficiency and low variance. The algorithm of UltraLogLog and ExaLogLog also specifies corrections for small and large ranges of cardinality and analyzes performance and compression possibilities.

9 Conclusion

In this paper I wanted to show the progressive improvements, that were made on an algorithm, that already was based on an impressive idea 40 years ago. Each update was made because of a certain need to solve a problem or to improve the quality even further.

The original FM85 algorithm had problems for lower cardinalities and general estimation error. Flajolet himself improved this with the HLL algorithm. By only storing the maximum values per register, the sketch would need less space. This means with the same space requirements, we can use more registers and thus gain better quality. He also improved the prediction quality for lower cardinalities by using linear counting, where we would not have enough information in the lower registers. But the transition between linear counting and the harmonic mean was not flawless. For certain values of m, we could see a spike with overestimation. HLL++ and HLLBeta improved the estimation and could eliminate the spike at the transition. HLL used 5 bits to save the values for each register, but in the HLL-TailCut algorithm, it could be shown, that already 4 bits can be enough to store big cardinalities without loosing quality of estimation. Only a few registers have very high numbers that would not contribute much to the estimation. They can be ignored, which means, we need fewer bits to store the maximum values. Then again, using extra bits below the maximum value for each register can be used to improve the quality even further. EHLL uses one extra bit, and UltraLogLog even two extra bits for the lowest memory-variance-product, as it can be proven that there is an optimal balance between memory and estimation quality. We also saw, that if we do not need the sketch to be mergeable, we can use a martingale estimator of the sketch, that would improve the estimation quality even more.

There are some threats for validity that have to be mentioned here for this paper. This is a narrative and not a systematic review. My choice for inclusion of journal articles is based on a subjective views, which algorithms and methods are important. My weighting for importance is mostly based on citations in later journal articles and search results in Google Scholar and Semantic Scholar.

Implementations to verify a method or to visualize a problem, mostly to generate the included figures can contain errors, even though they confirm the original results. Most of my implementations contain simplifications to visualize a certain aspect of a problem, while ignoring other points. My implementations are done in Python and I focused mostly on the error of estimation. The memory saving aspect of most methods is not implemented in my solutions (registers on a bit level), as I did not focus on it. In a lower level language

Table 1: Table showing a list of the presented methods with their estimation quality.

Algorithm	Year	Error	Memory	Comment
FM85	1985	$1.3/\sqrt{m}$	$32 \cdot m$	probabilistic counting of bits
HLL	2007	$1.04/\sqrt{m}$	$5 \cdot m$	harmonic mean, corrections for small and big cardinalities
HLL++	2013	$1.04/\sqrt{m}$	$\leq 6 \cdot m$	using 6 bits, empirical bias correction
HLL Martingale	2014	$< 1.04/\sqrt{m}$	$5 \cdot m$	non mergeable sketches
LogLogBeta	2016	$1.04/\sqrt{m}$	$\leq 6 \cdot m$	better quality for low cardinalities
HLL-TailCut	2017	$1.04/\sqrt{m}$	$4 \cdot m$	truncated registers for lower memory consumption
ExHLL	2023	$< 1.04/\sqrt{m}$	$6 \cdot m$	using 1 additional bit below maximum per register

this could be implemented very easy, but would not change the results of the included figures.

We can see, that the development of Flajolet's original sketch has not stopped in the last 40 years and researchers still find better ways to improve on the idea by adding new insights and building upon their predecessors.

References

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

[Balsam, 2025] Balsam (2025). https://github.com/sbalsam/CardinalityEstimation/.

[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.

[Ertl, 2024] Ertl, O. (2024). Ultraloglog: A practical and more space-efficient alternative to hyperloglog for approximate distinct counting.

[Ertl, 2025] Ertl, O. (2025). Exaloglog: Space-efficient and practical approximate distinct counting up to the exa-scale.

[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.

[Ohayon, 2021] Ohayon (2021). Extendedhyperloglog: Analysis of a new cardinality estimator.

[Pettie and Wang, 2020] Pettie and Wang (2020). Information theoretic limits of cardinality estimation: Fisher meets shannon.

[Qin et al., 2016] Qin, Kim, and Tung (2016). Loglog-beta and more: A new algorithm for cardinality estimation based on loglog counting.

[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.

[The Apache Foundation, 2019] The Apache Foundation (2019). Apache datasketches: A software library of stochastic streaming algorithms. https://datasketches.apache.org/.

[Ting, 2014] Ting (2014). Streamed approximate counting of distinct elements: beating optimal batch methods.

[Wang and Pettie, 2023] Wang and Pettie (2023). Better cardinality estimators for hyperloglog, pcsa, and beyond.

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

[Xuao et al., 2017] Xuao, Zhou, and Chen (2017). Better with fewer bits: Improving the performance of cardinality estimation of large data streams.