HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm

Stefan Heule
ETH Zurich and Google, Inc.
stheule@ethz.ch

Marc Nunkesser
Google, Inc.
marcnunkesser
@google.com

Alexander Hall
Google, Inc.
alexhall@google.com

ABSTRACT

Cardinality estimation has a wide range of applications and is of particular importance in database systems. Various algorithms have been proposed in the past, and the HyperLogLog algorithm is one of them. In this paper, we present a series of improvements to this algorithm that reduce its memory requirements and significantly increase its accuracy for an important range of cardinalities. We have implemented our proposed algorithm for a system at Google and evaluated it empirically, comparing it to the original HyperLogLog algorithm. Like HyperLogLog, our improved algorithm parallelizes perfectly and computes the cardinality estimate in a single pass.

1. INTRODUCTION

Cardinality estimation is the task of determining the number of distinct elements in a data stream. While the cardinality can be easily computed using space linear in the cardinality, for many applications, this is completely impractical and requires too much memory. Therefore, many algorithms that *approximate* the cardinality while using less resources have been developed. These algorithms play an important role in network monitoring systems, data mining applications, as well as database systems.

At Google, various data analysis systems such as Sawzall [15], Dremel [13] and PowerDrill [9] estimate the cardinality of very large data sets every day, for example to determine the number of distinct search queries on `google.com` over a time period. Such queries represent a hard challenge in terms of computational resources, and memory in particular: For the PowerDrill system, a non-negligible fraction of queries historically could not be computed because they exceeded the available memory.

In this paper we present a series of improvements to the HyperLogLog algorithm by Flajolet et. al. [7] that estimates cardinalities efficiently. Our improvements decrease memory usage as well as increase the accuracy of the estimate significantly for a range of important cardinalities. We evaluate all improvements empirically and compare with the HyperLogLog algorithm from [7]. Our changes to the algorithm are generally applicable and not specific to our system. Like HyperLogLog, our proposed improved algorithm parallelizes perfectly and computes the cardinality estimate in a single pass.

Outline. The remainder of this paper is organized as follows; we first justify our algorithm choice and summarize related work in Section 2. In Section 3 we give background information on our practical use cases at Google and list the requirements for a cardinality estimation algorithm in this context. In Section 4 we present the HyperLogLog algorithm from [7] that is the basis of our improved algorithm. In Section 5 we describe the improvements we made to the algorithm, as well as evaluate each of them empirically. Section 6 explains advantages of HyperLogLog for dictionary encodings in column stores. Finally, we conclude in Section 7.

2. RELATED WORK AND ALGORITHM CHOICE

Starting with work by Flajolet and Martin [6] a lot of research has been devoted to the cardinality estimation problem. See [3, 14] for an overview and a categorization of available algorithms. There are two popular models to describe and analyse cardinality estimation algorithms: Firstly, the data streaming (ε, δ)-model [10, 11] analyses the necessary space to get a $(1 \pm \varepsilon)$-approximation with a fixed success probability of δ, for example $\delta = 2/3$, for cardinalities in $\{1, \ldots, n\}$. Secondly it is possible to analyse the relative accuracy defined as the standard error of the estimator [3].

The algorithm previously implemented at Google [13, 15, 9] was MinCount, which is presented as *algorithm one* in [2]. It has an accuracy of $1.0/\sqrt{m}$, where m is the maximum number of hash values maintained (and thus linear in the required memory). In the (ε, δ)-model, this algorithm needs $O(\varepsilon^{-2} \log n)$ space, and it is near exact for cardinalities up to m (modulo hash collisions). See [8] for the statistical analysis and suggestions for making the algorithm more computationally efficient.

The algorithm presented by Kane et al. [11] meets the lower bound on space of $\Omega(e^{-2} + \log n)$ in the (ε, δ)-model [10] and is optimal in that sense. However, the algorithm is complex and an actual implementation and its maintenance seems out of reach in a practical system.

In [1], the authors compare six cardinality estimation algorithms including MinCount and LogLog [4] combined with LinearCounting [16] for small cardinalities. The latter algorithm comes out as the winner in that comparison. In [14] the authors compare 12 algorithms analytically and the most promising 8 experimentally on a single data set with $1.9 \cdot 10^6$ distinct elements, among them LogLog, LinearCounting and MultiresolutionBitmap [5], a multiscale version of LinearCounting. LinearCounting is recommended as the algorithm of choice for the tested cardinality. LogLog is shown to have a better accuracy than all other algorithms except LinearCounting and MultiresolutionBitmap on their input data. We are interested in estimating multisets of much larger cardinalities well beyond 10^9, for which LinearCounting is no longer attractive, as it requires too much memory for an accurate estimate. MultiresolutionBitmap has similar problems and needs $O(\varepsilon^{-2} \log n)$ space in the (ε, δ)-model, which is growing faster than the memory usage of LogLog. The authors of the study also had problems to run MultiresolutionBitmap with a fixed given amount of memory.

HyperLogLog has been proposed by Flajolet et. al. [7] and is an improvement of LogLog. It has been published after the afore-mentioned studies. Its relative error is $1.04/\sqrt{m}$ and it needs $O(\varepsilon^{-2} \log\log n + \log n)$ space in the (ε, δ)-model, where m is the number of counters (usually less than one byte in size). HyperLogLog is shown to be near optimal among algorithms that are based on order statistics. Its theoretical properties certify that it has a superior accuracy for a given fixed amount of memory over MinCount and many other practical algorithms. The fact that LogLog (with LinearCounting for small values cardinalities), on which HyperLogLog improves, performed so well in the previous experimental studies confirms our choice.

In [12], Lumbroso analyses an algorithm similar to HyperLogLog that uses the inverse of an arithmetic mean instead of the harmonic mean as evaluation function. Similar to our empirical bias correction described in Section 5.2, he performs a bias correction for his estimator that is based on a full mathematical bias analysis of its "intermediate regime".

3. PRACTICAL REQUIREMENTS FOR CARDINALITY ESTIMATION

In this section we present requirements for a cardinality estimation algorithm to be used in PowerDrill [9]. While we were driven by our particular needs, many of these requirements are more general and apply equally to other applications.

PowerDrill is a column-oriented datastore as well as an interactive graphical user interface that sends SQL-like queries to its backends. The column store uses highly memory-optimized data structures to allow low latency queries over datasets with hundreds of billions of rows. The system heavily relies on in-memory caching and to a lesser degree on the type of queries produced by the frontend. Typical queries group by one or more data fields and filter by various criteria.

As in most database systems, a user can count the number of *distinct* elements in a data set by issuing a *count distinct* query. In many cases, such a query will be grouped by a field (e.g., country, or the minute of the day) and counts the distinct elements of another field (e.g., query-text of Google searches). For this reason, a single query can lead to many count distinct computations being carried out in parallel.

On an average day, PowerDrill performs about 5 million such count distinct computations. As many as 99% of these computations yield a result of 100 or less. This can be explained partly by the fact that some groups in a group-by query can have only few values, which translates necessarily into a small cardinality. On the other extreme, about 100 computations a day yield a result greater than 1 billion.

As to the precision, while most users will be happy with a good estimate, there are some critical use cases where a very high accuracy is required. In particular, our users value the property of the previously implemented MinCount algorithm [2] that can provide near exact results up to a threshold and becomes approximate beyond that threshold.

Therefore, the key requirements for a cardinality estimation algorithm can be summarized as follows:

- *Accuracy.* For a fixed amount of memory, the algorithm should provide as accurate an estimate as possible. Especially for small cardinalities, the results should be near exact.
- *Memory efficiency.* The algorithm should use the available memory efficiently and adapt its memory usage to the cardinality. That is, the algorithm should use less than the user-specified maximum amount of memory if the cardinality to be estimated is very small.
- *Estimate large cardinalities.* Multisets with cardinalities well beyond 1 billion occur on a daily basis, and it is important that such large cardinalities can be estimated with reasonable accuracy.
- *Practicality.* The algorithm should be implementable and maintainable.

4. THE HYPERLOGLOG ALGORITHM

The HyperLogLog algorithm uses randomization to approximate the cardinality of a multiset. This randomization is achieved by using a hash function h that is applied to every element that is to be counted. The algorithm observes the maximum number of leading zeros that occur for all hash values, where intuitively hash values with more leading zeros are less likely and indicate a larger cardinality. If the bit pattern $0^{\varrho-1}1$ is observed at the beginning of a hash value, then a good estimation of the size of the multiset is 2^{ϱ} (assuming the hash function produces uniform hash values).

To reduce the large variability that such a single measurement has, a technique known as *stochastic averaging* [6] is used. To that end, the input stream of data elements S is divided into m substreams S_i of roughly equal size, using the first p bits of the hash values, where $m = 2^p$. In each substream, the maximum number of leading zeros (after the initial p bits that are used to determine the substream) is measured independently. These numbers are kept in an array of registers M, where $M[i]$ stores the maximum number of leading zeros plus one for substream with index i. That is,

$$M[i] = \max_{x \in S_i} \varrho(x)$$

where $\varrho(x)$ denotes the number of leading zeros in the binary representation of x plus one. Note that by convention $\max_{x\in\emptyset} \varrho(x) = -\infty$. Given these registers, the algorithm then computes the cardinality estimate as the normalized bias corrected harmonic mean of the estimations on the substreams as

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

where

$$\alpha_m := \left(m \int_0^\infty \left(\log_2 \left(\frac{2+u}{1+u} \right) \right)^m du \right)^{-1}$$

Full details on the algorithm, as well as an analysis of its properties can be found in [7]. In a practical setting, however, this algorithm has a series of problems, which Flajolet et. al. address by presenting a practical variant of the algorithm. This second algorithm uses 32 bit hash values with the precision argument p in the range [4..16]. The following modifications are applied to the algorithm. For a full explanation of these changes, see [7].

1. *Initialization of registers.* The registers are initialized to 0 instead of $-\infty$ to avoid the result 0 for $n \ll m \log m$ where n is the cardinality of the data stream (i.e., the value we are trying to estimate).

2. *Small range correction.* Simulations by Flajolet et. al. show that for $n < \frac{5}{2}m$ nonlinear distortions appear that need to be corrected. Thus, for this range LinearCounting [16] is used.

3. *Large range corrections.* When n starts to approach $2^{32} \approx 4 \cdot 10^9$, hash collisions become more and more likely (due to the 32 bit hash function). To account for this, a correction is used.

The full practical algorithm is shown in Figure 1. In the remainder of this paper, we refer to it as Hll$_{\text{Orig}}$.

```
Require: Let $h : \mathcal{D} \to \{0,1\}^{32}$ hash data from domain $\mathcal{D}$.
    Let $m = 2^p$ with $p \in [4..16]$.
Phase 0: Initialization.
 1: Define $\alpha_{16} = 0.673$, $\alpha_{32} = 0.697$, $\alpha_{64} = 0.709$,
 2:     $\alpha_m = 0.7213/(1 + 1.079/m)$ for $m \geq 128$.
 3: Initialize $m$ registers $M[0]$ to $M[m-1]$ to 0.
Phase 1: Aggregation.
 4: for all $v \in S$ do
 5:     $x := h(v)$
 6:     $idx := \langle x_{31}, \dots, x_{32-p} \rangle_2$        { First $p$ bits of $x$ }
 7:     $w := \langle x_{31-p}, \dots, x_0 \rangle_2$
 8:     $M[idx] := \max\{M[idx], \varrho(w)\}$
 9: end for
Phase 2: Result computation.
10: $E := \alpha_m m^2 \cdot \left( \sum_{j=0}^{m-1} 2^{-M[j]} \right)^{-1}$        { The "raw" estimate }
11: if $E \leq \frac{5}{2}m$ then
12:     Let $V$ be the number of registers equal to 0.
13:     if $V \neq 0$ then
14:         $E^* :=$ LinearCounting$(m, V)$
15:     else
16:         $E^* := E$
17:     end if
18: else if $E \leq \frac{1}{30}2^{32}$ then
19:     $E^* := E$
20: else
21:     $E^* := -2^{32} \log(1 - E/2^{32})$
22: end if
23: return $E^*$

Define LinearCounting$(m, V)$
    Returns the LinearCounting cardinality estimate.
24: return $m \log(m/V)$
```

Figure 1: The practical variant of the HyperLogLog algorithm as presented in [7]. We use *LSB 0* bit numbering.

5. IMPROVEMENTS TO HYPERLOGLOG

In this section we propose a number of improvements to the HyperLogLog algorithm. The improvements are presented as a series of individual changes, and we assume for every step that all previously presented improvements are kept. We call the final algorithm HyperLogLog++ and show its pseudo-code in Figure 6.

5.1 Using a 64 Bit Hash Function

An algorithm that only uses the hash code of the input values is limited by the number of bits of the hash codes when it comes to accurately estimating large cardinalities. In particular, a hash function of L bits can at most distinguish 2^L different values, and as the cardinality n approaches 2^L, hash collisions become more and more likely and accurate estimation gets impossible.

A useful property of the HyperLogLog algorithm is that the memory requirement does not grow linearly with L, unlike other algorithms such as MinCount or LinearCounting. Instead, the memory requirement is determined by the number of registers and the maximum size of $\varrho(w)$ (which is stored in the registers). For a hash function of L bits and a precision p, this maximum value is $L + 1 - p$. Thus, the memory required for the registers is $\lceil \log_2(L + 1 - p) \rceil \cdot 2^p$ bits. The algorithm Hll$_{\text{Orig}}$ uses 32 bit hash codes, which requires $5 \cdot 2^p$ bits.

To fulfill the requirement of being able to estimate multisets of cardinalities beyond 1 billion, we use a 64 bit hash function. This increases the size of a single register by only a single bit, leading to a total memory requirement of $6 \cdot 2^p$. Only if the cardinality approaches $2^{64} \approx 1.8 \cdot 10^{19}$, hash collisions become a problem; we have not needed to estimate inputs with a size close to this value so far.

With this change, the large range correction for cardinalities close to 2^{32} used in HLLORIG is no longer needed. It would be possible to introduce a similar correction if the cardinality approaches 2^{64}, but it seems unlikely that such cardinalities are encountered in practice. If such cardinalities occur, however, it might make more sense to increase the number of bits for the hash function further, especially given the low additional cost in memory.

5.2 Estimating Small Cardinalities

The raw estimate of HLLORIG (cf. Figure 1, line 10) has a large error for small cardinalities. For instance, for $n = 0$ the algorithm always returns roughly $0.7m$ [7]. To achieve better estimates for small cardinalities, HLLORIG uses LINEARCOUNTING [16] below a threshold of $\frac{5}{2}m$ and the raw estimate above that threshold

In simulations, we noticed that most of the error of the raw estimate is due to *bias*; the algorithm overestimates the real cardinality for small sets. The bias is larger for smaller n, e.g., for $n = 0$ we already mentioned that the bias is about $0.7m$. The statistical variability of the estimate, however, is small compared to the bias. Therefore, if we can correct for the bias, we can hope to get a better estimate, in particular for small cardinalities.

Experimental Setup. To measure the bias, we ran a version of HLL64BIT that does not use LINEARCOUNTING and measured the estimate for a range of different cardinalities. The HYPERLOGLOG algorithm uses a hash function to randomize the input data, and will thus, for a fixed hash function and input, return the same results. To get reliable data we ran each experiment for a fixed cardinality and precision on 5000 different randomly generated data sets of that cardinality. Intuitively, the distribution of the input set should be irrelevant as long as the hash function ensures an appropriate randomization. We were able to convince ourselves of this by considering various data generation strategies that produced differently distributed data and ensuring that our results were comparable. We use this approach of computing results on randomly generated datasets of the given cardinality for all experiments in this paper.

Note that the experiments need to be repeated for every possible precision. For brevity, and since the results are qualitatively similar, we illustrate the behavior of the algorithm by considering only precision 14 here and in the remainder of the paper.

We use the same proprietary 64 bit hash function for all experiments. We have tested the algorithm with a variety of hash functions including MD5, SHA1, SHA256, MURMUR3, as well as several proprietary hash functions. However, in our experiments we were not able to find any evidence that any of these hash functions performed significantly better than others.

Empirical Bias Correction. To determine the bias, we calculate the mean of all raw estimates for a given cardinality minus that cardinality. In Figure 2 we show the average raw estimate with 1% and 99% quantiles. We also

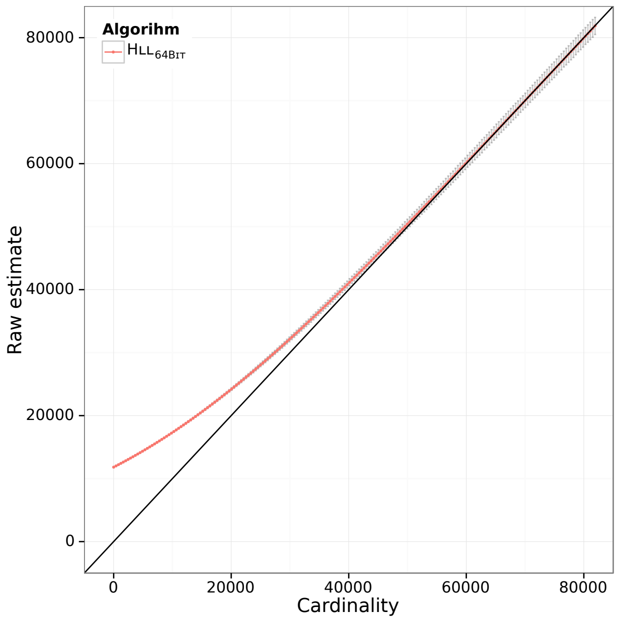

Figure 2: The average raw estimate of the HLL64BIT algorithm to illustrate the bias of this estimator for $p = 14$, as well as the 1% and 99% quantiles on 5000 randomly generated data sets per cardinality. Note that the quantiles and the median almost coincide for small cardinalities; the bias clearly dominates the variability in this range.

show the $x = y$ line, which would be the expected value for an unbiased estimator. Note that only if the bias accounts for a significant fraction of the overall error can we expect a reduced error by correcting for the bias. Our experiments show that at the latest for $n > 5m$ the correction does no longer reduce the error significantly.

With this data, for any given cardinality we can compute the observed bias and use it to correct the raw estimate. As the algorithm does not know the cardinality, we record for every cardinality the raw estimate as well as the bias so that the algorithm can use the raw estimate to look up the corresponding bias correction. To make this practical, we choose 200 cardinalities as interpolation points, for which we record the average raw estimate and bias. We use k-nearest neighbor interpolation to get the bias for a given raw estimate (for $k = 6$)[1]. In the pseudo-code in Figure 6 we use the procedure ESTIMATEBIAS that performs the k-nearest neighbor interpolation.

Deciding Which Algorithm To Use. The procedure described so far gives rise to a new estimator for the cardinality, namely the *bias-corrected raw estimate*. This procedure corrects for the bias using the empirically determined data for cardinalities smaller than $5m$ and uses the unmodified raw estimate otherwise. To evaluate how well the bias correction

[1]The choice of $k = 6$ is rather arbitrary. The best value of k could be determined experimentally, but we found that the choice has only a minuscule influence.

works, and to decide if this algorithm should be used in favor of LinearCounting, we perform another experiment, using the bias-corrected raw estimate, the raw estimate as well as LinearCounting. We ran the three algorithms for different cardinalities and compare the distribution of the error. Note that we use a different dataset for this second experiment to avoid overfitting.

As shown in Figure 3, for cardinalities that up to about 61000, the bias-corrected raw estimate has a smaller error than the raw estimate. For larger cardinalities, the error of the two estimators converges to the same level (since the bias gets smaller in this range), until the two error distributions coincide for cardinalities above $5m$.

For small cardinalities, LinearCounting is still better than the bias-corrected raw estimate[2]. Therefore, we determine the intersection of the error curves of the bias-corrected raw estimate and LinearCounting to be at 11500 for precision 14 and use LinearCounting to the left, and the bias-corrected raw estimate to the right of that threshold.

As with the bias correction, the algorithm does not have access to the true cardinality to decide on which side of the threshold the cardinality lies, and thus which algorithm should be used. However, again we can use one of the estimates to make the decision. Since the threshold is in a range where LinearCounting has a fairly small error, we use its estimate and compare it with the threshold. We call the resulting algorithm that combines the LinearCounting and the bias-corrected raw estimate $\text{Hll}_{\text{NoBias}}$.

Advantages of Bias Correction. Using the bias-corrected raw estimate in combination with LinearCounting has a series of advantages compared to combining the raw estimate and LinearCounting:

- The error for an important range of cardinalities is smaller than the error of $\text{Hll}_{\text{64Bit}}$. For precision 14, this range is roughly between 18000 and 61000 (cf. Figure 3).
- The resulting algorithm does not have a significant bias. This is not true for $\text{Hll}_{\text{64Bit}}$ (or Hll_{Orig}), which uses the raw estimate for cardinalities above the threshold of $5/2m$. However, at that point, the raw estimate is still significantly biased, as illustrated in Figure 4.
- Both algorithms use an empirically determined threshold to decide which of the two sub-algorithms to use. However, the two relevant error curves for $\text{Hll}_{\text{NoBias}}$ are less steep at the threshold compared to $\text{Hll}_{\text{64Bit}}$ (cf. Figure 3). This has the advantage that a small error in the threshold has smaller consequences for the accuracy of the resulting algorithm.

[2]This is not entirely true, for very small cardinalities it seems that the bias-corrected raw estimate has again a smaller error, but a higher variability. Since LinearCounting also has low error, and depends less on empirical data, we decided to use it for all cardinalities below the threshold.

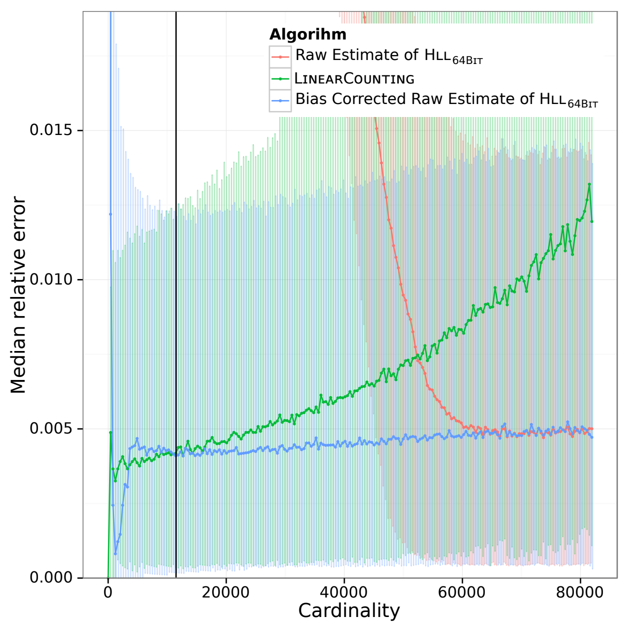

Figure 3: The median error of the raw estimate, the bias-corrected raw estimate, as well as LinearCounting for $p = 14$. Also shown are the 5% and 95% quantiles of the error. The measurements are based on 5000 data points per cardinality.

5.3 Sparse Representation

$\text{Hll}_{\text{NoBias}}$ requires a constant amount of memory throughout the execution of $6m$ bits, regardless of n, violating our memory efficiency requirement. If $n \ll m$, then most of the registers are never used and thus do not have to be represented in memory. Instead, we can use a *sparse representation* that stores pairs $(idx, \varrho(w))$. If the list of such pairs would require more memory than the dense representation of the registers (i.e., $6m$ bits), the list can be converted to the dense representation.

Note that pairs with the same index can be merged by keeping the one with the highest $\varrho(w)$ value. Various strategies can be used to make insertions of new pairs into the sparse representation as well as merging elements with the same index efficient. In our implementation we represent $(idx, \varrho(w))$ pairs as a single integer by concatenating the bit patterns for idx and $\varrho(w)$ (storing idx in the higher-order bits of the integer).

Our implementation then maintains a sorted list of such integers. Furthermore, to enable quick insertion, a separate set is kept where new elements can be added quickly without keeping them sorted. Periodically, this temporary set is sorted and merged with the list (e.g., if it reaches 25% of the maximum size of the sparse representation), removing any pairs where another pair with the same index and a higher $\varrho(w)$ value exists.

Because the index is stored in the high-order bits of the integer, the sorting ensures that pairs with the same index occur consecutively in the sorted sequence, allowing the merge to

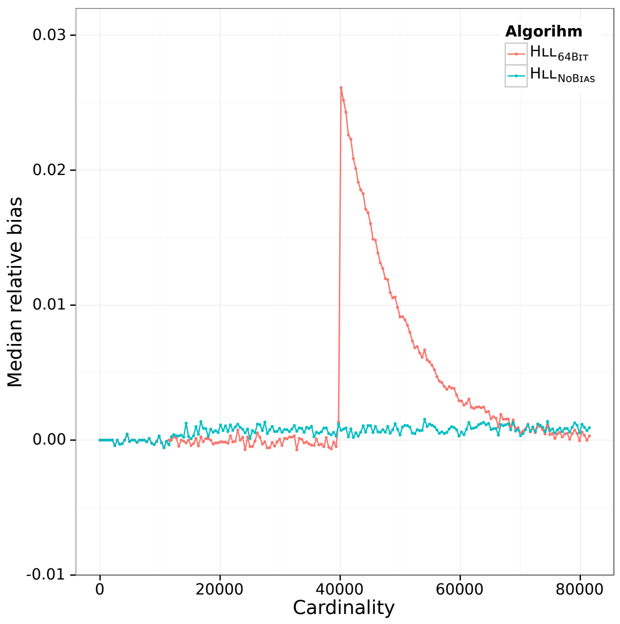

Figure 4: The median bias of HLL$_{\text{Orig}}$ and HLL$_{\text{NoBias}}$. The measurements are again based on 5000 data points per cardinality.

happen in a single linear pass over the sorted set and the list. In the pseudo-code of Figure 6, this merging happens in the subroutine MERGE.

The computation of the overall result given a sparse representation in phase 2 of the algorithm can be done in a straight forward manner by iterating over all entries in the list (after the temporary set has been merged with the list), and assuming that any index not present in the list has a register value of 0. As we will explain in the next section, this is not necessary in our final algorithm and thus is not part of the pseudo-code in Figure 6.

The sparse representation reduces the memory consumption for cases where the cardinality n is small, and only adds a small runtime overhead by amortizing the cost of searching and merging through the use of the temporary set.

5.3.1 Higher Precision for the Sparse Representation

Every item in the sparse representation requires $p + 6$ bits, namely to store the index (p bits) and the value of that registers (6 bits). In the sparse representation we can choose to perform all operations with a different precision argument $p' > p$. This allows us to increase the accuracy in cases where only the sparse representation is used (and it is not necessary to convert to the normal representation). If the sparse representation gets too large and reaches the user-specified memory threshold of $6m$ bits, it is possible to fall back to precision p and switch to the dense representation. Note that falling back from p' to the lower precision p is always possible: Given a pair $(idx', \varrho(w'))$ that has been determined with precision p', one can determine the corresponding pair $(idx, \varrho(w))$ for the smaller precision p as follows. Let $h(v)$ be the hash value for the underlying data element v.

1. idx' consists of the p' most significant bits of $h(v)$, and since $p < p'$, we can determine idx by taking the p most significant bits of idx'.
2. For $\varrho(w)$ we need the number of leading zeros of the bits of $h(v)$ after the index bits, i.e., of bits $63 - p$ to 0. The bits $63 - p$ to $64 - p'$ are known by looking at idx'. If at least one of these bits is one, then $\varrho(w)$ can be computed using only those bits. Otherwise, bits $63 - p$ to $64 - p'$ are all zero, and using $\varrho(w')$ we know the number of leading zeros of the remaining bits. Therefore, in this case we have $\varrho(w) = \varrho(w') + (p' - p)$.

This computation is done in DECODEHASH of Figure 6. It is possible to compute at a different, potentially much higher accuracy p' in the sparse representation, without exceeding the memory limit indicated by the user through the precision parameter p. Note that choosing a suitable value for p' is a trade-off. The higher p' is, the smaller the error for cases where only the sparse representation is used. However, at the same time as p' gets larger, every pair requires more memory which means the user-specified memory threshold is reached sooner in the sparse representation and the algorithm needs to switch to the dense representation earlier.

Also note that one can increase p' up to 64, at which points the full hash code is kept.

We use the name HLL$_{\text{Sparse1}}$ to refer to this algorithm. To illustrate the increased accuracy, Figure 5 shows the error distribution with and without the sparse representation.

5.3.2 Compressing the Sparse Representation

So far, we presented the sparse representation to use a temporary set and a list which is kept sorted. Since the temporary set is used for quickly adding new elements and merged with the list before it gets large, using a simple implementation with some built-in integer type works well, even if some bits per entry are wasted (due to the fact that built-in integer types may be too wide). For the list, however, we can exploit two facts to store the elements more compactly. First of all, there is an upper limit on the number of bits used per integer, namely $p' + 6$. Using an integer of fixed width (e.g., `int` or `long` as offered in many programming languages) might be wasteful. Furthermore, the list is guaranteed to be sorted, which can be exploited as well.

We use a *variable length encoding* for integers that uses variable number of bytes to represent integers, depending on their absolute value. Furthermore, we use a *difference encoding*, where we store the difference between successive elements in the list. That is, for a sorted list $a_1, a_2, a_3, \ldots$ we would store $a_1, a_2 - a_1, a_3 - a_2, \ldots$. The values in such a list of differences have smaller absolute values, which makes the variable length encoding even more efficient. Note that when sequentially going through the list, the original items can easily be recovered.

We use the name HLL$_{\text{Sparse2}}$ if only the variable length encoding is used, and HLL$_{\text{Sparse3}}$ if additionally the difference

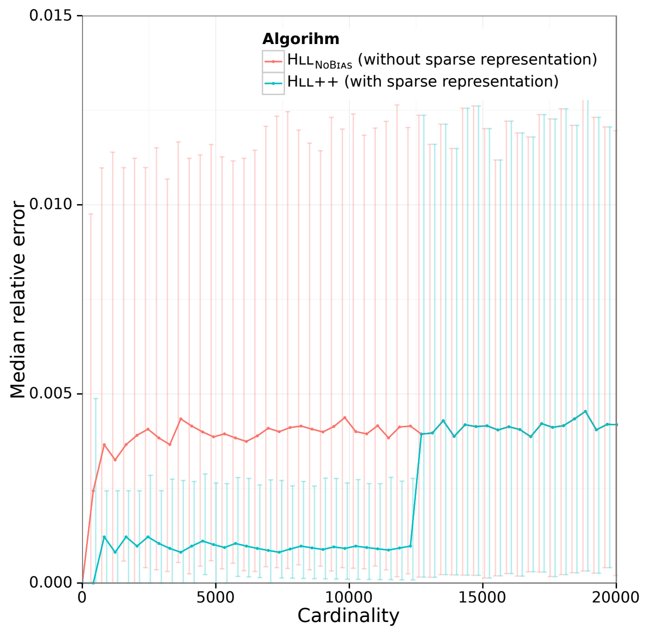

Figure 5: Comparison of HLL$_{\text{NoBias}}$ and HLL++ to illustrate the increased accuracy, here for $p' = 25$, with 5% and 95% quantiles. The measurements are on 5000 data points per cardinality.

encoding is used.

Note that introducing these two compression steps is possible in an efficient way, as the sorted list of encoded hash values is only updated in batches (when merging the entries in the set with the list). Adding a single new value to the compressed list would be expensive, as one has to potentially read the whole list in order to even find the correct insertion point (due to the difference encoding). However, when the list is merged with the temporary set, then the list is traversed sequentially in any case. The pseudo-code in Figure 7 uses the not further specified subroutine DecodeSparse to decompress the variable length and difference encoding in a straight forward way.

5.3.3 *Encoding Hash Values*

It is possible to further improve the storage efficiency with the following observations. If the sparse representation is used for the complete aggregation phase, then in the result computation HLL$_{\text{NoBias}}$ will always use LinearCounting to determine the result. This is because the maximum number of hash values that can be stored in the sparse representation is small compared to the cardinality threshold of where to switch from LinearCounting to the bias-corrected raw estimate. Since LinearCounting only requires the number of distinct indices (and m), there is no need for $\varrho(w')$ from the pair. The value $\varrho(w')$ is only used when switching from the sparse to the normal representation, and even then only if the bits $\langle x_{63-p}, \dots, x_{64-p'}\rangle$ are all 0. For a good hash function with uniform hash values, the value $\varrho(w')$ only needs to be stored with probability $2^{p-p'}$.

This idea can be realized by only storing $\varrho(w')$ if necessary,

p	m	HLL$_{\text{Sparse1}}$	HLL$_{\text{Sparse2}}$	HLL$_{\text{Sparse3}}$	HLL++
10	1024	192.00	316.45	420.73	534.27
12	4096	768.00	1261.82	1962.18	2407.73
14	16384	3072.00	5043.91	8366.45	12107.00
16	65536	12288.00	20174.64	35616.73	51452.64

Table 1: The maximum number of pairs with distinct index that can be stored before the representation reaches the size of the dense representation, i.e., $6m$ bits. All measurements have been repeated for different inputs, for $p' = 25$.

and using one bit (e.g., the least significant bit) to indicate whether it is present or not. We use the following encoding: If the bits $\langle x_{63-p}, \dots, x_{64-p'}\rangle$ are all 0, then the resulting integer is

$$\langle x_{63}, \dots, x_{64-p'}\rangle \parallel \langle \varrho(w')\rangle \parallel \langle 1\rangle$$

(where we use $\parallel$ to concatenate bits). Otherwise, the pair is encoded as

$$\langle x_{63}, \dots, x_{64-p'}\rangle \parallel \langle 0\rangle$$

The least significant bit allows to easily decode the integer again. Procedures EncodeHash and DecodeHash of Figure 7 implement this encoding.

In our implementation[3] we fix $p' = 25$, as this provides very high accuracy for the range of cardinalities where the sparse representation is used. Furthermore, the 25 bits for the index, 6 bits for $\varrho(w')$ and one indicator bit fit nicely into a 32 bit integer, which is useful from a practical standpoint.

We call this algorithm HyperLogLog++ (or HLL++ for short), which is shown in Figure 6 and includes all improvements from this paper.

5.3.4 *Space Efficiency*

In Table 1 we show the effects of the different encoding strategies on the space efficiency of the sparse encoding for a selection of precision parameters p. The less memory a single pair requires on average, the longer can the algorithm use the sparse representation without switching to the dense representation. This directly translates to a high precision for a larger range of cardinalities.

For instance, for precision 14, storing every element in the sparse representation as an integer would require 32 bits. The variable length encoding reduces this to an average of 19.49 bits per element. Additionally introducing a difference encoding requires 11.75 bits per element and using the improved encoding of hash values further decreases this value to 8.12 bits on average.

5.4 Evaluation of All Improvements

To evaluate the effect of all improvements, we ran HLL$_{\text{Orig}}$ as presented in [7] and HLL++. The error distribution clearly illustrates the positive effects of our changes on the accuracy of the estimate. Again, a fresh dataset has been used

[3]This holds for all backends except for our own column store, where we use both $p' = 20$ and $p' = 25$, also see Section 6.

```
Input: The input data set $S$, the precision $p$, the precision
    $p'$ used in the sparse representation where $p \in [4..p']$ and
    $p' \leq 64$. Let $h : \mathcal{D} \to \{0,1\}^{64}$ hash data from domain $\mathcal{D}$
    to 64 bit values.
Phase 0: Initialization.
 1: $m := 2^p$; $m' := 2^{p'}$
 2: $\alpha_{16} := 0.673$; $\alpha_{32} := 0.697$; $\alpha_{64} := 0.709$
 3: $\alpha_m := 0.7213/(1 + 1.079/m)$ for $m \geq 128$
 4: format := SPARSE
 5: tmp_set := $\emptyset$
 6: sparse_list := []
Phase 1: Aggregation.
 7: for all $v \in S$ do
 8:     $x := h(v)$
 9:     switch format do
10:         case NORMAL
11:             idx := $\langle x_{63}, \dots, x_{64-p} \rangle_2$
12:             $w := \langle x_{63-p}, \dots, x_0 \rangle_2$
13:             $M[idx] := \max\{M[idx], \varrho(w)\}$
14:         end case
15:         case SPARSE
16:             $k :=$ EncodeHash$(x, p, p')$
17:             tmp_set := tmp_set $\cup \{k\}$
18:             if tmp_set is too large then
19:                 sparse_list :=
20:                     Merge(sparse_list, Sort(tmp_set))
21:                 tmp_set := $\emptyset$
22:                 if |sparse_list| $> m \cdot 6$ bits then
23:                     format := NORMAL
24:                     $M :=$ ToNormal(sparse_list)
25:                 end if
26:             end if
27:         end case
28:     end switch
29: end for
Phase 2: Result computation.
30: switch format do
31:     case SPARSE
32:         sparse_list := Merge(sparse_list, Sort(tmp_set))
33:         return LinearCounting($m'$, $m'$ − |sparse_list|)
34:     end case
35:     case NORMAL
36:         $E := \alpha_m m^2 \cdot \left( \sum_{j=0}^{m-1} 2^{-M[j]} \right)^{-1}$
37:         $E' := (E \leq 5m)$ ? $(E -$ EstimateBias$(E, p))$ : $E$
38:         Let $V$ be the number of registers equal to 0.
39:         if $V \neq 0$ then
40:             $H :=$ LinearCounting$(m, V)$
41:         else
42:             $H := E'$
43:         end if
44:         if $H \leq$ Threshold$(p)$ then
45:             return $H$
46:         else
47:             return $E'$
48:         end if
49:     end case
50: end switch
```

Figure 6: The Hll++ algorithm that includes all the improvements presented in this paper. Some auxiliary procedures are given in Figure 7.

```
Define LinearCounting$(m, V)$
    Returns the LinearCounting cardinality estimate.
 1: return $m \log(m/V)$

Define Threshold$(p)$
    Returns empirically determined threshold (we provide
    the values from our implementation at http://goo.gl/
    iU8Ig).

Define EstimateBias$(E, p)$
    Returns the estimated bias, based on the interpolating
    with the empirically determined values.

Define EncodeHash$(x, p, p')$
    Encodes the hash code $x$ as an integer.
 2: if $\langle x_{63-p}, \dots, x_{64-p'} \rangle = 0$ then
 3:     return $\langle x_{63}, \dots, x_{64-p'} \rangle \parallel \langle \varrho(\langle x_{63-p'}, \dots, x_0 \rangle) \rangle \parallel \langle 1 \rangle$
 4: else
 5:     return $\langle x_{63}, \dots, x_{64-p'} \rangle \parallel \langle 0 \rangle$
 6: end if

Define GetIndex$(k, p)$
    Returns the index with precision $p$ stored in $k$
 7: if $\langle k_0 \rangle = 1$ then
 8:     return $\langle k_{p+6}, \dots, k_6 \rangle$
 9: else
10:     return $\langle k_{p+1}, \dots, k_1 \rangle$
11: end if

Define DecodeHash$(k, p, p')$
    Returns the index and $\varrho(w)$ with precision $p$ stored in $k$
12: if $\langle k_0 \rangle = 1$ then
13:     $r := \langle k_6, \dots, k_1 \rangle + (p' - p)$
14: else
15:     $r := \varrho(\langle k_{p'-p-1}, \dots, k_1 \rangle)$
16: end if
17: return (GetIndex$(k, p), r)$

Define ToNormal(sparse_list, $p, p'$)
    Converts the sparse representation to the normal one
18: $M :=$ NewArray$(m)$
19: for all $k \in$ DecodeSparse(sparse_list) do
20:     (idx, $r$) := DecodeHash$(k, p, p')$
21:     $M[idx] := \max\{M[idx], \varrho(w)\}$
22: end for
23: return $M$

Define Merge$(a, b)$
    Expects two sorted lists $a$ and $b$, where the first is com-
    pressed using a variable length and difference encoding.
    Returns a list that is sorted and compressed in the same
    way as $a$, and contains all elements from $a$ and $b$, except
    for entries where another element with the same index,
    but higher $\varrho(w)$ value exists. This can be implemented
    in a single linear pass over both lists.

Define DecodeSparse$(a)$
    Expects a sorted list that is compressed using a variable
    length and difference encoding, and returns the elements
    from that list after it has been decompressed.
```

Figure 7: Auxiliary procedures for the Hll++ algorithm.

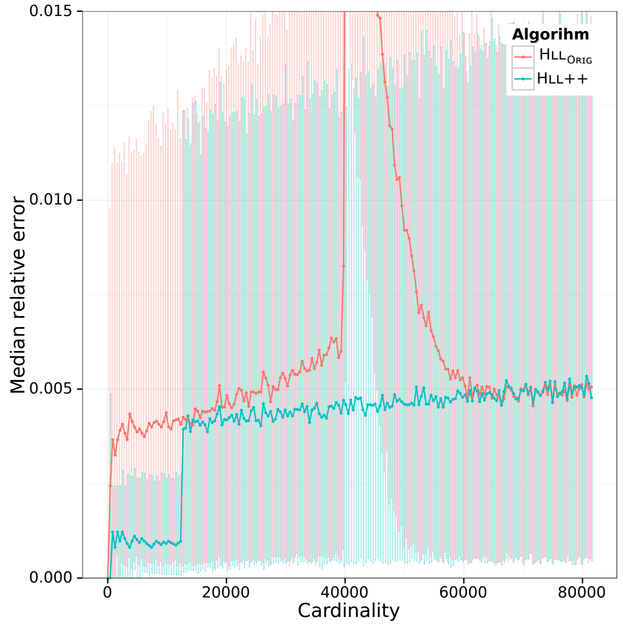

Figure 8: Comparison of HLL_{ORIG} with HLL++. The mean relative error as well as the 5% and 95% quantiles are shown. The measurements are on 5000 data points per cardinality.

for this experiment, and the results of the comparison for precision 14 are shown in Figure 8.

First of all, there is a spike in the error of HLL_{ORIG}, almost exactly at $n = 5/2m = 40960$. The reason for this is that the ideal threshold of when to switch from LINEARCOUNTING to the raw estimate in HLL_{ORIG} is not precisely at $5/2m$. As explained in Section 5.2, a relatively small error in this threshold leads to a rather large error in the overall error, because the error curve of the raw estimate is fairly steep. Furthermore, even if the threshold was determined more precisely for HLL_{ORIG}, its error would still be larger than that of HLL++ in this range of cardinalities. Our HYPERLOGLOG++ algorithm does not exhibit any such behavior.

The advantage of the sparse representation is clearly visible; for cardinalities smaller than about 12000, the error of our final algorithm HLL++ is significantly smaller than for HLL_{ORIG} without a sparse representation.

6. IMPLICATIONS FOR DICTIONARY ENCODINGS OF COLUMN STORES

In this section we focus on a property of HYPERLOGLOG (and HLL++ in particular) that we were able to exploit in our implementation for the column-store presented in [9].

Given the expression materialization strategy of that column store, any cardinality estimation algorithm that computes the hash value of an expression will have to add a data column of these values to the store. The advantage of this approach is that the hash values for any given expression only need to be computed once. Any subsequent computation can use the precomputed data column.

As explained in [9], most column stores use a dictionary encoding for the data columns that maps values to identifiers. If there is a large number of distinct elements, the size of the dictionary can easily dominate the memory needed for a given count distinct query.

HYPERLOGLOG has the useful property that not the full hash code is required for the computation. Instead, it suffices to know the first p bits (or p' if the sparse representation with higher accuracy from Section 5.3 is used) as well as the number of leading zeros of the remaining bits.

This leads to a smaller maximum number of distinct values for the data column. While there are 2^{64} possible distinct hash values if the full hash values were to be stored, there are only $2^p \cdot (64 - p' - 1) + 2^p \cdot (2^{p'-p} - 1)$ different values that our integer encoding from Section 5.3.3 can take[4]. This reduces the maximum possible size of the dictionary by a major factor if $p' \ll 64$

For example, our column store already bounds the size of the dictionary to 10 million (and thus there will never be 2^{64} different hash values). Nonetheless, using the default parameters $p' = 20$ and $p = 14$, there can still be at most 1.74 million values, which directly translates to a memory saving of more than 82% for the dictionary if all input values are distinct.

7. CONCLUSIONS

In this paper we presented a series of improvements to the HYPERLOGLOG algorithm. Most of these changes are orthogonal to each other and can thus be applied independently to fit the needs of a particular application.

The resulting algorithm HYPERLOGLOG++ fulfills the requirements listed in Section 3. Compared to the practical variant of HYPERLOGLOG from [7], the *accuracy* is significantly better for large range of cardinalities and equally good on the rest. For precision 14 (and $p' = 25$), the sparse representation allows the average error for cardinalities up to roughly 12000 to be smaller by a factor of 4. For cardinalities between 12000 and 61000, the bias correction allows for a lower error and avoids a spike in the error when switching between sub-algorithms due to less steep error curves. The sparse representation also allows for a more adaptive use of memory; if the cardinality n is much smaller than m, then HYPERLOGLOG++ requires significantly less memory. This is of particular importance in PowerDrill where often many count distinct computations are carried out in parallel for a single count distinct query. Finally, the use of 64 bit hash codes allows the algorithm to estimate cardinalities well beyond 1 billion.

All of these changes can be implemented in a straight-forward way and we have done so for the PowerDrill system. We provide a complete list of our empirically determined parameters at `http://goo.gl/iU8Ig` to allow easier reproduction of our results.

[4]This can be seen as follows: There are $2^p(64-p'-1)$ many encoded values that store $\varrho(w')$, and similarly $2^p(2^{p'-p}-1)$ many without it.

8. REFERENCES

[1] K. Aouiche and D. Lemire. A comparison of five probabilistic view-size estimation techniques in OLAP. In *Workshop on Data Warehousing and OLAP (DOLAP)*, pages 17–24, 2007.

[2] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *Workshop on Randomization and Approximation Techniques (RANDOM)*, pages 1–10, London, UK, UK, 2002. Springer-Verlag.

[3] P. Clifford and I. A. Cosma. A statistical analysis of probabilistic counting algorithms. *Scandinavian Journal of Statistics*, pages 1–14, 2011.

[4] M. Durand and P. Flajolet. Loglog counting of large cardinalities. In G. D. Battista and U. Zwick, editors, *European Symposium on Algorithms (ESA)*, volume 2832, pages 605–617, 2003.

[5] C. Estan, G. Varghese, and M. Fisk. Bitmap algorithms for counting active flows on high-speed links. *IEEE/ACM Transactions on Networking*, pages 925–937, 2006.

[6] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[7] P. Flajolet, Éric Fusy, O. Gandouet, and F. Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *Analysis of Algorithms (AOFA)*, pages 127–146, 2007.

[8] F. Giroire. Order statistics and estimating cardinalities of massive data sets. *Discrete Applied Mathematics*, 157(2):406–427, 2009.

[9] A. Hall, O. Bachmann, R. Büssow, S. Gănceanu, and M. Nunkesser. Processing a trillion cells per mouse click. In *Very Large Databases (VLDB)*, 2012.

[10] P. Indyk. Tight lower bounds for the distinct elements problem. In *Foundations of Computer Science (FOCS)*, pages 283–288, 2003.

[11] D. M. Kane, J. Nelson, and D. P. Woodruff. An optimal algorithm for the distinct elements problem. In *Principles of database systems (PODS)*, pages 41–52. ACM, 2010.

[12] J. Lumbroso. An optimal cardinality estimation algorithm based on order statistics and its full analysis. In *Analysis of Algorithms (AOFA)*, pages 489–504, 2010.

[13] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, T. Vassilakis, and G. Inc. Dremel: Interactive analysis of web-scale datasets. In *Very Large Databases (VLDB)*, pages 330–339, 2010.

[14] A. Metwally, D. Agrawal, and A. E. Abbadi. Why go logarithmic if we can go linear? Towards effective distinct counting of search traffic. In *Extending database technology (EDBT)*, pages 618–629, 2008.

[15] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data, parallel analysis with Sawzall. *Journal on Scientific Programming*, pages 277–298, 2005.

[16] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems*, 15:208–229, 1990.