UltraLogLog: A Practical and More Space-Efficient Alternative to HyperLogLog for Approximate Distinct Counting

Otmar Ertl
Dynatrace Research
Linz, Austria
otmar.ertl@dynatrace.com

ABSTRACT

Since its invention HyperLogLog has become the standard algorithm for approximate distinct counting. Due to its space efficiency and suitability for distributed systems, it is widely used and also implemented in numerous databases. This work presents UltraLogLog, which shares the same practical properties as HyperLogLog. It is commutative, idempotent, mergeable, and has a fast guaranteed constant-time insert operation. At the same time, it requires 28% less space to encode the same amount of distinct count information, which can be extracted using the maximum likelihood method. Alternatively, a simpler and faster estimator is proposed, which still achieves a space reduction of 24%, but at an estimation speed comparable to that of HyperLogLog. In a non-distributed setting where martingale estimation can be used, UltraLogLog is able to reduce space by 17%. Moreover, its smaller entropy and its 8-bit registers lead to better compaction when using standard compression algorithms. All this is verified by experimental results that are in perfect agreement with the theoretical analysis which also outlines potential for even more space-efficient data structures. A production-ready Java implementation of UltraLogLog has been released as part of the open-source Hash4j library.

PVLDB Reference Format:
Otmar Ertl. UltraLogLog: A Practical and More Space-Efficient Alternative to HyperLogLog for Approximate Distinct Counting. PVLDB, 17(7): 1655 - 1668, 2024.
doi:10.14778/3654621.3654632

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/dynatrace-research/ultraloglog-paper.

1 INTRODUCTION

Many applications require counting the number of distinct elements in a data set or data stream. It is well-known that exact counting needs linear space [6]. However, the space requirements can be drastically reduced, if approximate results suffice. HyperLogLog (HLL) [23] with improved small-range estimation [20, 27, 39, 43, 51] represents the state-of-the-art approximate distinct-count algorithm. It allows distributed counting of up to the order of $2^{64} \approx 1.8 \cdot 10^{19}$ distinct elements with a relative standard error of $1.04/\sqrt{m}$ using only $6m$ bits [27]. Therefore, it is nowadays offered by a big number of data stores as part of their query language (see e.g. documentation of Timescale, Redis, Oracle Database, Snowflake, Microsoft SQL Server, Google BigQuery, Vertica, Elasticsearch, Aerospike, Amazon Redshift, KeyDB, DuckDB, or Dynatrace Grail). Further applications of HLL include query optimization [25, 34], caching [48], graph analysis [9, 38], attack detection [13, 15], network volume estimation [8], or metagenomics [7, 10, 19, 31].

HLL is actually very simple as exemplified in Algorithm 1. It typically consists of a densely packed array of 6-bit registers $r_0, r_1, \ldots, r_{m-1}$ [27] where the number of registers m is a power of 2, $m = 2^p$. The choice of the precision parameter p allows trading space for better estimation accuracy. Adding an element requires calculating a 64-bit hash value. p bits are used to choose a register for the update. The number of leading zeros (NLZ) of the remaining $64 - p$ bits are interpreted as a geometrically distributed update value with success probability $\frac{1}{2}$ and positive support, that is used to update the selected register, if its current value is smaller. Estimating the distinct count from the register values is more challenging, but can also be implemented using a few lines of code [20, 23].

The popularity of HLL is based on following advantageous features that make it especially useful in distributed systems:

Speed: Element insertion is a fast and allocation-free operation with a constant time complexity independent of the sketch size. In particular, given the hash value of the element, the update requires only a few CPU instructions.
Idempotency: Further insertions of the same element will never change the state. This is actually a natural property every count-distinct algorithm should support to prevent duplicates from changing the result.
Mergeability: Partial results calculated over subsets can be easily merged to a final result. This is important when data is distributed or processed in parallel.
Reproducibility: The result does not depend on the processing order, which often cannot be guaranteed in practice anyway. Reproducibility is achieved by a commutative insert operation and a commutative and associative merge operation.
Reducibility: The state can be reduced to a smaller state corresponding to a smaller precision parameter. The reduced state is identical to that obtained by direct recording with lower precision. This property allows adjusting the precision without affecting the mergeability with older records.
Estimation: A fast and robust estimation algorithm ensures nearly unbiased estimates with a relative standard error bounded by a constant over the full range of practical distinct counts.
Simplicity: The implementation requires only a few lines of code. The entire state can be stored in a single byte array of fixed length

Proceedings of the VLDB Endowment, Vol. 17, No. 7 ISSN 2150-8097.
doi:10.14778/3654621.3654632

Algorithm 1: Inserts an element with 64-bit hash value $\langle h_{63}h_{62}\dots h_0\rangle_2$ into a HyperLogLog consisting of $m = 2^p$ ($p \geq 2$) 6-bit registers $r_0, r_1, \dots, r_{m-1}$ with initial values $r_i = 0$.

$i \leftarrow \langle h_{63}h_{62}\dots h_{64-p}\rangle_2$ ▷ extract register index

$a \leftarrow \langle \underbrace{0\dots0}_{p} h_{63-p}h_{62-p}\dots h_0\rangle_2$ ▷ mask register index bits

$k \leftarrow \texttt{nlz}(a) - p + 1$ ▷ update value $k \in [1, 65-p]$

▷ function nlz returns the number of leading zeros

$r_i \leftarrow \max(r_i, k)$ ▷ update register

which makes serialization very fast and convenient. Furthermore, add and in-place merge operations do not require any additional memory allocations.

To our knowledge HLL is so far the most space-efficient practical data structure having all these desired properties. Space-efficiency can be measured in terms of the memory-variance product (MVP) [37], which is the relative variance of the (unbiased) distinct-count estimate $\hat{n}$ multiplied by the storage size in bits

$$\text{MVP} := \text{Var}(\hat{n}/n) \times (\text{storage size in bits}), \tag{1}$$

where n is the true distinct count. If the MVP is asymptotically (for sufficiently large distinct counts) a constant specific to the data structure, it can be used for comparison as it eliminates the general inverse dependence of the relative estimation error on the root of the storage size. Most HLL implementations use 6-bit registers [27] to support distinct counts beyond the billion range, resulting in a MVP of 6.48 [37]. A recent theoretical work conjectured a general lower bound of 1.98 for the MVP of sketches supporting mergeability and reproducibility [36], which shows the potential for improvement. Many different approaches have been proposed to beat the space efficiency of HLL, but they all sacrificed at least one of the properties listed above.

1.1 Related Work

Lossless compression of HyperLogLog (HLL) can significantly reduce the storage size [18, 29, 40]. Since the compressed state prevents random access to registers as required for insertion, bulking is needed to realize at least amortized constant update times. The required buffer partially cancels out the memory savings. Recent techniques avoid buffering of insertions. The Apache Data Sketches library [2] provides an implementation using 4 bits per register to store the most frequent values relative to a global offset. Out of range values are kept separately in an associative array. Overall, this leads to a smaller MVP but also to a more expensive insert operation. Its runtime is proportional to the memory size in the worst case, because all registers must be updated whenever the global offset is increased. HyperLogLogLog [28] takes this strategy to the extreme with 3-bit registers and achieves a space saving of around 40% at the expense of an insert operation which, except for very large numbers, has been reported to be on average more than an order of magnitude slower compared to HLL [28].

Interestingly, lossless compression of probabilistic counting with stochastic averaging (PCSA) [24], a less-space efficient (when uncompressed) predecessor of HLL also known as FM-sketch, yields a smaller MVP than compression of HLL [29, 40]. The compressed probability counting (CPC) sketch as part of the Apache Data Sketches library [2] uses this finding. The serialized representation of the CPC sketch achieves a MVP of around 2.31 [3] that is already quite close to the conjectured lower bound of 1.98 [36]. However, the need of bulked updates to achieve amortized constant-time insertions more than doubles the memory footprint which also makes serialization significantly slower than for the original HLL. Similar to all compressed variants of HLL, the insert operation of the CPC sketch takes time proportional to the sketch size in the worst case.

Lossy compression of HLL has also been proposed [49, 50]. However, like other approaches such as HyperBitBit/HyperBit [41], they trade idempotency for less space and are therefore risky to use [36]. In contrast, sacrificing mergeability for less space is of greater practical interest [14, 26, 30]. When the data is not distributed and a merge operation is not actually needed, the MVP of HLL can be reduced by 36% down to 4.16 by martingale also known as historic inverse probability (HIP) estimation [16, 42]. The theoretical limit of MVP for non-mergeable sketches is at most 1.63 [37], and is also nearly reached by the serialized representation of the CPC sketch [3]. Data structures with in-memory representations that can be updated in constant time in the worst case are HLL with vectorized counters [11] and the martingale curtain sketch which achieves a MVP of 2.31 [37]. As the insertions are not commutative, all these non-mergeable approaches also do not support reproducibility.

The only mergeable data structure we know of that is more space-efficient than HLL while having essentially the same properties, in particular constant-time worst-case updates, is ExtendedHyperLogLog (EHLL) [33]. It extends the HLL registers from 6 to 7 bits to store not only the maximum update value, but also whether there was an update with a value smaller by one. This additional information can be used to obtain more accurate estimates. In particular, the MVP is reduced by 16% to 5.43. The only thing missing to make it really practical is an estimator for small distinct counts. As with the original HLL [23], it was only proposed to switch to the linear probabilistic count estimator [47]. However, this is problematic because the estimation error in the transition region can be large [20, 27]. Nevertheless, EHLL motivated us to generalize its basic idea by extending HLL registers even further.

Table 1: Notation

Symbol	Comment
n	distinct count
$\hat{n}$	distinct count estimate
b	base, $b > 1$, defines distribution of update values, compare (2)
p	precision parameter
m	number of registers, $m = 2^p$
w	maximum possible update value, cf. Section 2.1
q	number of bits used for storing the maximum occurred update values, $2^q > w$
d	number of additional register bits to indicate updates with smaller values
r_i	value of i-th register , $0 \leq i < m$, $0 \leq r_i < (w+1)2^d$
u_i	maximum occurred update value for i-th register, $u_i = \lfloor r_i/2^d \rfloor$, $0 \leq u_i \leq w$
c_j	number of registers with value j, $c_j := \lvert\{i \lvert r_i = j\}\rvert$
ρ_{reg}	probability mass function (PMF) of register values, see (4)
$\tilde{\rho}_{\text{reg}}$	approximated PMF of register values, see (5)
$\mathcal{L}$	likelihood function, $\mathcal{L} = \mathcal{L}(n \lvert r_0 \dots r_{m-1})$
$\mathcal{I}$	Fisher information, $\mathcal{I} = \mathbb{E}(-\partial^2/\partial n^2 \ln \mathcal{L})$, see (7)
$\mathcal{H}$	Shannon entropy, $\mathcal{H} = \mathbb{E}(-\log_2 \mathcal{L})$, see (9)
τ	free parameter of generalized remaining area (GRA) estimators
z_k	short notation for $\exp(-n(b-1)/(mb^k))$, compare (3)
Γ	gamma function, $\Gamma(x) := \int_0^\infty y^{x-1}e^{-y}dy$
ζ	Hurvitz zeta function, $\zeta(x,y) := \sum_{u=0}^\infty (u+y)^{-x} = \frac{1}{\Gamma(x)} \int_0^\infty \frac{z^{x-1}e^{-yz}}{1-e^{-z}} dz$
μ	state change probability
$\langle\dots\rangle_2$	binary representation, e.g. $\langle 110\rangle_2 = 6$
$\lfloor\dots\rfloor$	floor function, e.g. $\lfloor 3.7 \rfloor = 3$

1.2 Summary of Contributions

We first describe a data structure that generalizes the known data structures HyperLogLog (HLL) [23], ExtendedHyperLogLog (EHLL) [33], and probabilistic counting with stochastic averaging (PCSA) [24]. We derive analytic expressions for the Fisher information and the Shannon entropy as functions of the data structure parameters. Although these expressions reveal even more space-efficient configurations that could be the subject of future research, the focus of this work is on a setting that leads to a very practical data sketch called UltraLogLog (ULL) with a MVP of 4.63 which is 28% below that of HLL. Since the Shannon entropy is also 24% smaller for the same estimation error, ULL is also more compact when using lossless compression. Moreover, our experimental results indicate that standard compression algorithms additionally benefit from the 8-bit register size of ULL.

To extract all the information contained in the ULL sketch, we applied the maximum likelihood (ML) method that achieves an estimation error as theoretically predicted by the Cramér-Rao bound [12]. Alternatively, we present a faster approach based on a further generalization of the recently proposed generalized remaining area (GRA) estimator [46]. Even though our theoretical analysis shows a smaller estimation efficiency than the ML estimator, the MVP with a value of 4.94 still corresponds to a 24% space reduction compared to HLL. As the GRA estimator, the basic version of our new estimator works only for distinct counts that are neither too small nor too large. Therefore, we developed two additional estimators specific to those ranges that are also easy to evaluate. Using a novel approach, they are seamlessly combined with the basic estimator to cover the full range of distinct count values. We also analyzed martingale estimation, which can be used for non-distributed data, and found that in this case ULL reduces the MVP by 17% to 3.47 compared to HLL.

All theoretically derived estimators were verified by intensive simulations, which all show perfect agreement with the theoretically predicted estimation errors. In particular, we use a technique that allows verification for distinct counts on the order of 2^{64}, which is not possible using traditional simulations. Finally, we also present the results of speed benchmarks, which show that ULL is similarly fast as HLL.

An implementation of ULL is publicly available as part of the Hash4j open-source Java library at https://github.com/dynatrace-oss/hash4j. Detailed instructions together with the necessary source code to reproduce all presented results and figures can be found at https://github.com/dynatrace-research/ultraloglog-paper. A version of this paper extended by an appendix with mathematical derivations and proofs is also available [22].

2 GENERALIZED DATA STRUCTURE

We start by introducing a data structure for approximate distinct counting that generalizes HyperLogLog (HLL) [23], ExtendedHyperLogLog (EHLL) [33], and probabilistic counting with stochastic averaging (PCSA) [24]. As those, it consists of m registers which are initially set to zero. For every added element a uniformly distributed hash value is computed. This hash value is used to extract a uniform random register index $i \in [0, m)$ and some geometrically distributed integer value with probability mass function (PMF)

$$\rho_{\text{update}}(k) = (b-1)b^{-k} \qquad k \geq 1, b > 1, \tag{2}$$

parameterized by the base parameter b, that is used to update the i-th register. Each register consists of $q + d$ bits. q bits are used to store the maximum update value u_i seen so far. The remaining d bits indicate whether there have been any updates with values $u_i - 1, \ldots, u_i - d$, respectively. Obviously, this update procedure is idempotent meaning that further occurrences of the same elements will never change any register state. As a consequence, the final state of this data structure can be used to estimate the number of inserted distinct elements n.

Every register state can be described by an integer value r_i with $0 \leq r_i < 2^{q+d}$. We assume that the most significant q bits of r_i are used to store u_i, which can therefore be simply obtained by $u_i = \lfloor r_i/2^d \rfloor$. As example for $q = 6$ and $d = 2$, $r_i = \langle 00011010 \rangle_2$ would mean that the largest update value was $u_i = \langle 000110 \rangle_2 = 6$. The right-most d bits indicate that the register was also already updated with a value of 5 but not yet with 4. If the register gets further updated with value 8, the state would become $r_i = \langle 00100001 \rangle_2$ where the first $q = 6$ bits encode $u_i = \langle 001000 \rangle_2 = 8$ and the left-most $d = 2$ bits indicate that there was no update with value 7 but one with 6. Information about smaller update values is lost. If $u_i \leq d$, there are only $u_i - 1$ smaller update values and therefore only $u_i - 1$ of the d extra bits are relevant and some values like $r_i = \langle 00001001 \rangle_2$ for $q = 6$ and $d = 2$ cannot be attained. Enumerating just possible states would lead to a slightly more compact encoding. However, for the sake of simplicity and also to avoid special cases, we refrain from this small improvement.

In practice, the number of registers m is usually some power of 2, $m = 2^p$ with p being the precision parameter. In this way, a uniform random register index can be chosen by just taking p bits from the hash value. Furthermore, the parameter b is often 2 such that the update value k can be easily obtained from the number of leading zeros (NLZ) of the remaining hash bits and therefore usually requires just a single CPU instruction. Obviously, the cases $b = 2$, $d = 0$ and $b = 2$, $d = 1$ correspond to HLL and EHLL, respectively. Furthermore, since PCSA effectively keeps track of any update values, the stored information corresponds to our generalized data structure with $b = 2$, $d \to \infty$. (PCSA typically uses just 64 bits for each register which is sufficient as update values greater than 64 are unlikely for real-world distinct counts.) Another advantage of choosing m as a power of 2 and $b = 2$ is that the data structure can be implemented in such a way that it can later be reduced to a smaller precision parameter [20]. The result will then be identical as if the smaller precision parameter was chosen from the beginning. This is important for migration scenarios where precision needs to be changed in a way that is compatible and mergeable with historical data.

2.1 Statistical Model

For simplification, we use the common Poisson approximation [20, 23, 46] that the number of inserted distinct elements is not fixed, but follows a Poisson distribution with mean n. As a consequence, since updates are evenly distributed over all registers, the number of updates with value k per register is again Poisson distributed

with mean $n\rho_{\text{update}}(k)/m = n(b-1)/(mb^k)$. The probability that a register was updated with value k at least once, denoted by event A_k, is therefore

$$\Pr(A_k) = 1 - z_k \quad \text{with } z_k := \exp(-n(b-1)/(mb^k)). \tag{3}$$

The probability that u was the largest update value, which implies that there were no updates with values greater than u, is given by $\Pr(A_u \wedge \bigwedge_{k=u+1}^{\infty} \overline{A}_k) = (1-z_u)\prod_{k=u+1}^{\infty} z_k = z_u^{\frac{1}{b-1}}(1-z_u)$. The Poisson approximation results in registers that are independent and identically distributed. For the generalized data structure the corresponding probability mass function (PMF) can be written as

$$\rho_{\text{reg}}(r|n) = \Pr(r_i = r) = \tag{4}$$

$$\begin{cases} z_0^{\frac{1}{b-1}} & r{=}0, \\ z_u^{\frac{1}{b-1}}(1-z_u)\prod_{j=1}^{u-1} z_{u-j}^{1-l_j}(1-z_{u-j})^{l_j} & r{=}u2^d{+}\langle l_1 \ldots l_{u-1}\rangle_2 2^{d+1-u},\ 1{\le}u{\le}d, \\ z_u^{\frac{1}{b-1}}(1-z_u)\prod_{j=1}^{d} z_{u-j}^{1-l_j}(1-z_{u-j})^{l_j} & r{=}u2^d{+}\langle l_1 \ldots l_d\rangle_2,\ d{+}1{\le}u{<}w, \\ (1-z_{w-1}^{\frac{1}{b-1}})\prod_{j=1}^{d} z_{w-j}^{1-l_j}(1-z_{w-j})^{l_j} & r{=}w2^d{+}\langle l_1 \ldots l_d\rangle_2, \\ 0 & \text{else.} \end{cases}$$

This formula takes into account that update values are limited to the range $[1, w]$. The upper limit is on the one hand a consequence of the number of register bits reserved for storing the maximum update value. If q bits are used, the update values must be truncated at $2^q - 1$ because higher update values cannot be stored. On the other hand, the way the geometrically distributed integer values are typically determined also leads to an upper limit. For example, the update values in Algorithm 1 do not exceed $65 - p$, which results from extracting the update value and the register index from a single 64-bit hash value.

For our theoretical analysis, we consider a simplified model. We assume that registers are initially set to $-\infty$ and that there are no restrictions on the update values. In particular, there are also updates with non-positive values $k \le 0$ with corresponding (virtual) events A_k occurring with probabilities according to (3). Then the PMF for a register simplifies to

$$\tilde{\rho}_{\text{reg}}(r|n) = \Pr(r_i = r) = z_u^{\frac{1}{b-1}}(1-z_u)\prod_{j=1}^{d} z_{u-j}^{1-l_j}(1-z_{u-j})^{l_j}$$

$$\text{with } r = u2^d + \langle l_1 \ldots l_d\rangle_2. \tag{5}$$

If all register values are in the range $[(d+1)2^d, w2^d)$, which is usually the case for not too small and not too large distinct counts, (4) and (5) are equivalent. As a consequence, the following theoretical results will also hold for (4) when the distinct count is in the intermediate range.

2.2 Theoretical Analysis

Given the register states $r_0, \ldots, r_{m-1}$, the log-likelihood function can be expressed as

$$\ln \mathcal{L} = \ln \mathcal{L}(n|r_0, \ldots, r_{m-1}) = \textstyle\sum_{i=0}^{m-1} \ln \rho_{\text{reg}}(r_i|n). \tag{6}$$

When assuming the simplified PMF (5), the Fisher information can be written as (see extended paper [22])

$$\mathcal{I} = \mathbb{E}\left(-\frac{\partial^2}{\partial n^2}\ln \mathcal{L}\right) \approx \frac{m}{n^2}\frac{1}{\ln b}\zeta\left(2, 1+\frac{b^{-d}}{b-1}\right) \tag{7}$$

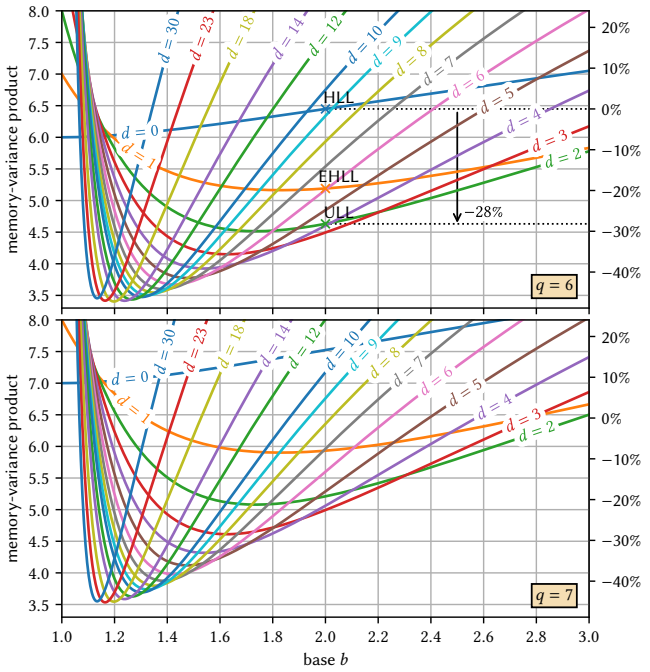

Figure 1: The theoretical asymptotic memory-variance product (MVP) (8) over the base b for $q = 6$ and $q = 7$ and various values of d when assuming a memory footprint of $m(q+d)$ bits. The top chart shows the 28% improvement of UltraLogLog (ULL) over HyperLogLog (HLL).

using the Hurvitz zeta function ζ as defined in Table 1. The approximation is based on the fact that the Fisher information is a periodic function of $\log_b n$ with period 1 and tiny relative amplitude that can be ignored in practice as shown in the extended paper [22]. Formula (7) matches the results reported for the special cases of generalized HLL ($d = 0$) and generalized PCSA ($d \to \infty$) [36].

According to Cramér-Rao [12] the reciprocal of the Fisher information $\mathcal{I}$ is a lower bound for the variance of any unbiased estimator. Our experiments will show that for a sufficiently large number of registers m, this lower bound can be actually reached using maximum likelihood (ML) estimation. The corresponding asymptotic MVP is given by

$$\text{MVP} = m(q+d)\operatorname{Var}\left(\frac{\hat{n}}{n}\right) = \frac{m(q+d)}{\mathcal{I}n^2} \approx \frac{(q+d)\ln b}{\zeta(2, 1+\frac{b^{-d}}{b-1})}. \tag{8}$$

$(q+d)$ is the number of bits used for a single register. q bits are used for storing the maximum update value and d is the number of extra bits as already described before. The MVP (8) is plotted in Figure 1 over the base b for $q = 6$ and $q = 7$ and various values of d and allows comparing the memory-efficiencies of different configurations, when assuming that the state takes overall $m(q+d)$ bits and is not further compressed.

It is also interesting to study the case where the state is ideally compressed, which means that the number of bits needed to store the state is given by its Shannon entropy. Considering the resulting MVP, also called Fisher-Shannon (FISH) number [36], again allows

a better comparison. Using the simplified PMF (5), the Shannon entropy can be approximated by (see extended paper [22])

$$\mathcal{H} = \mathbb{E}(-\log_2 \mathcal{L}) \approx \frac{m}{(\ln 2)(\ln b)} \left(\left(1 + \frac{b^{-d}}{b-1}\right)^{-1} + \int_0^1 z^{\frac{b^{-d}}{b-1}} \frac{(1-z)\ln(1-z)}{z \ln z} dz \right). \quad (9)$$

Combining (7) and (9) gives for the MVP under the assumption of an efficient estimator and optimal compression

$$\mathrm{MVP} = \mathcal{H} \operatorname{Var}\left(\frac{\hat{n}}{n}\right) = \frac{\mathcal{H}}{\mathcal{I} n^2} \approx \frac{(1+\frac{b^{-d}}{b-1})^{-1} + \int_0^1 z^{\frac{b^{-d}}{b-1}} \frac{(1-z)\ln(1-z)}{z\ln z} dz}{\zeta(2, 1+\frac{b^{-d}}{b-1}) \ln 2}. \quad (10)$$

This function is plotted in Figure 2 over the base b for various values of d. The results agree with the conjecture that the MVP of sketches supporting mergeability and reproducibility is fundamentally bounded by [36]

$$\lim_{\frac{b^{-d}}{b-1} \to 0} \frac{\mathcal{H}}{\mathcal{I} n^2} \approx \frac{1 + \int_0^1 \frac{(1-z)\ln(1-z)}{z \ln z} dz}{\zeta(2,1) \ln 2} \approx 1.98.$$

2.3 Choice of Parameters

The data structure proposed in Section 2 has four parameters, the base b, the number of registers $m = 2^p$, the number of register bits q reserved for storing the maximum update value, and the number of additional bits d to indicate the occurrences of the d next smaller update values relative to the maximum. The previous theoretical results allow us to find parameters that lead to a small MVP. Here we can essentially leave m aside since it can be used to define the accuracy/space tradeoff, but has essentially no effect on the MVP according to (8) and (10).

The parameters b and q define the operating range of the data structure and must be chosen such that it is very unlikely that all registers get saturated. In other words, the fraction of registers with $r_i \geq w2^d$ must be small. This requires according to (4) that $\Pr(r_i < w2^d) = z_{w-1}^{\frac{1}{b-1}} = \exp(-\frac{n}{mb^{w-1}}) \approx 1$ or $n \ll mb^{w-1}$. Hence, the maximum supported distinct count $n_{\max}$ can be roughly estimated by $n_{\max} \approx mb^{w-1}$. If w is limited by the register size ($w = 2^q - 1$, cf. Section 2.1), we get $n_{\max} \approx mb^{2^q-2}$. For HLL with $m = 256$ registers, $b = 2$, and a registers size of $q = 5$ bits, as originally proposed, we have $n_{\max} \approx 275$ billions which might not be sufficient in all situations. Therefore, most HLL implementations use nowadays $q = 6$ [27], which is definitely sufficient for any realistic counts. Some implementations even use a whole byte per register ($q = 8$), but mainly to have more convenient register access in memory [5].

Bearing in mind the operating range defined by m, b, and q, we explore the theoretical MVP (8) for different configurations as shown in Figure 1. First we consider the case $b = 2$ which also covers HLL ($b = 2$, $d = 0$) and EHLL ($b = 2$, $d = 1$) and is of particular practical interest due to the very fast mapping of hash values to update values as discussed in Section 2. For $b = 2$, we need to choose $q = 6$, if we want to make sure that any realistic distinct counts can be handled. According to Figure 1 the optimal choice for $b = 2$ would be $d = 3$ resulting in a theoretical MVP of 4.4940. Despite its slightly larger theoretical MVP of 4.6313, the case $d = 2$ is very attractive from a practical point of view since a register takes $q + d = 8$ bits and fits perfectly into a single byte. This enables fast

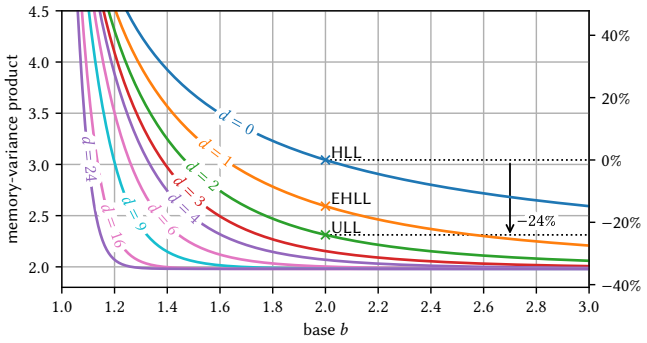

Figure 2: The theoretical asymptotic memory-variance product (MVP) (10) over the base b for various values of d under the assumption of optimal lossless compression. The MVP of UltraLogLog (ULL) is 24% smaller than that of HyperLogLog (HLL).

updates as registers can be directly accessed and modified when stored in a byte array. This is why we picked $b = 2$, $d = 2$, $q = 6$ as configuration for our new data structure called UltraLogLog (ULL). Its theoretical MVP is 28% smaller than that of HLL with a register size of $q = 6$ and a MVP of 6.4485. This even corresponds to a 46% improvement, when compared to HLL implementations that use $q = 8$ bits per register [5] leading to MVP ≈ 8.5981.

Figure 1 shows that the MVP could be further reduced by choosing bases b other than 2. For $q = 6$, a minimum MVP of 3.4030 is achieved for $b \approx 1.1976$ and $d = 18$. However, a small b leads to a small operating range. For example, $b = 1.1976$ yields $n_{\max} \approx 18$ millions for $m = 256$ which is too small for many applications. However, the working range can be extended again by increasing q. For $q = 7$ the minimum MVP ≈ 3.5338 is obtained for $d = 23$ and $b \approx 1.1642$ leading to $n_{\max} \approx 53$ billions for $m = 256$. From a practical point of view the configuration $q = 7$, $d = 9$, $b = \sqrt{2}$ with MVP ≈ 3.9025 could be an interesting choice for future research. The operating range would be similar to ULL or HLL, registers would take exactly two bytes ($q + d = 16$), and the mapping of hash values to update values for $b = \sqrt{2}$ according to (2) could be accomplished without expensive logarithm evaluations or table lookups. The update value can be obtained by doubling the number of leading zeros (NLZ) and adding 0 or 1 depending on the remaining hash bits, whose value range is divided into two parts in the ratio $1 : \sqrt{2}$. However, a single 64-bit hash value might not be sufficient in this case.

Figure 2 shows the MVP under the assumption of optimal lossless compression as given by (10). The ULL configuration with $d = 2$ leads to MVP ≈ 2.3122 which is 24% less than for HLL with $d = 0$ and MVP ≈ 3.0437. Therefore, ULL is potentially superior to any variant of HLL that uses lossless compression techniques [2, 28], if similar techniques are also applied to ULL.

3 DISTINCT COUNT ESTIMATION

The theoretical results show clearly that UltraLogLog (ULL) as a special case of the presented generalized data structure with $b = 2$, $d = 2$ and $q = 6$ encodes distinct count information, uncompressed

and compressed, more efficiently than HLL with $b = 2$, $d = 0$ and $q = 6$. However, an open question is whether this information can also be readily accessed using a simple and robust estimation procedure, which will be the focus of the following sections.

3.1 Maximum-Likelihood Estimator

Since we know the probability mass function (PMF) (4) when using the Poisson approximation, we can simply use the maximum likelihood (ML) method. For $b = 2$, the log-likelihood function (6) is shaped like

$$\ln \mathcal{L} = -\frac{n}{m}\alpha + \sum_{u=1}^{w-1} \beta_u \ln(1 - e^{-\frac{n}{m2^u}}),$$

where for ULL with $d = 2$ the coefficients α and β_u are given by

$$\alpha = c_0 + \frac{c_4}{2} + \frac{3c_8+c_{10}}{4} + \left(\sum_{u=3}^{w-1} \frac{7c_{4u}+3c_{4u+1}+5c_{4u+2}+c_{4u+3}}{2^u}\right) + \frac{3c_{4w}+c_{4w+1}+2c_{4w+2}}{2^{w-1}},$$

$$\beta_1 = c_4+c_{10}+c_{13}+c_{15}, \quad \beta_2 = c_8+c_{10}+c_{14}+c_{15}+c_{17}+c_{19},$$

$$\beta_u = c_{4u}+c_{4u+1}+c_{4u+2}+c_{4u+3}+c_{4u+6}+c_{4u+7}+c_{4u+9}+c_{4u+11} \quad \text{for } 3 \le u \le w-2,$$

$$\beta_{w-1} = c_{4w-4}+c_{4w-3}+c_{4w-2}+c_{4w-1}+c_{4w}+c_{4w+1}+2c_{4w+2}+2c_{4w+3}.$$

$c_j := |\{i|r_i = j\}|$ is the number of registers having value j.

As the corresponding ML equation has the same shape as that for HLL, we can reuse the numerically robust solver we developed based on the secant method [20]. This algorithm avoids the evaluation of expensive mathematical functions, but is still somewhat costly as the solution needs to be found iteratively. The ML estimate $\hat{n}_{\text{ML}}$ can be further improved by correcting for the first-order bias [17]. Applying the correction factor as derived in the extended paper [22] under the assumption of the simplified PMF (5) gives for ULL with $b = 2$ and $d = 2$

$$\hat{n} = \hat{n}_{\text{ML}} \left(1 + \frac{1}{m} \frac{3(\ln 2)\zeta(3, \frac{5}{4})}{2(\zeta(2, \frac{5}{4}))^2}\right)^{-1} \approx \hat{n}_{\text{ML}} \left(1 + \frac{0.48147}{m}\right)^{-1} \tag{11}$$

where ζ denotes again the Hurvitz zeta function.

The ML method is known to be asymptotically efficient as $m \to \infty$. The experimental results presented later in Section 5.1 show that the ML estimate really matches the theoretically predicted MVP (8). For HLL, estimators have been found that are easier to compute than the ML estimator while giving almost equal estimates over the whole value range [20]. This was our motivation to look for simpler estimators for ULL.

3.2 GRA Estimator

Recently, the generalized remaining area (GRA) estimator was proposed, which can be easily computed and is more efficient than existing estimators for probabilistic counting with stochastic averaging (PCSA) and HyperLogLog (HLL) [46]. Therefore, we investigated if this estimation approach is also suitable for our generalized data structure and in particular for ULL. The basic idea is to sum up $b^{-\tau k}$ with some constant $\tau > 0$ for all update values k that we know with certainty, based on the current register value $r = u2^d + \langle l_1 \dots l_d \rangle_2$, could not have occurred previously. The corresponding statistic for our generalized data structure can be expressed as $\sum_{k=u-d}^{u-1}(1 - l_{u-k})b^{-\tau k} + \sum_{k=u+1}^{\infty} b^{-\tau k} = b^{-\tau u}(\frac{1}{b^\tau - 1} + \sum_{s=1}^{d}(1 - l_s)b^{\tau s})$. The analysis of the first two moments of this statistic under the assumption of the simplified PMF (5), together with the delta method (see extended paper [22]) yields

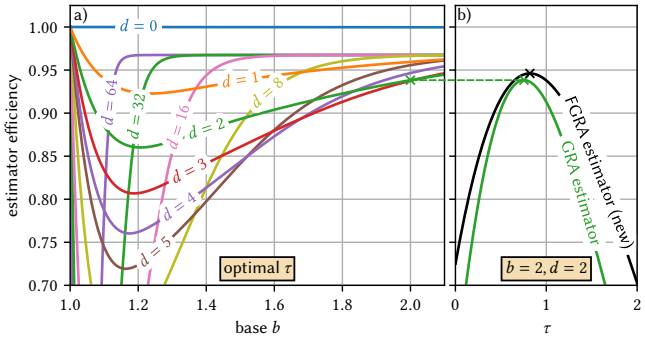

Figure 3: a) The asymptotic GRA estimator efficiency over the base b for $q = 6$ and various values of d. b) The efficiencies of the GRA estimator and our proposed FGRA estimator as a function of τ for $b = 2$ and $d = 2$. Crosses indicate optimal choices of τ.

the distinct count estimator

$$\hat{n} = m^{1+\frac{1}{\tau}} \cdot \left(\sum_{i=0}^{m-1} g(r_i)\right)^{-\frac{1}{\tau}} \cdot \left(1 + \frac{1+\tau}{2}\frac{v}{m}\right)^{-1} \tag{12}$$

where the individual register contributions are defined as

$$g(r) := \frac{(b-1+b^{-\tau})^\tau \ln b}{\Gamma(\tau)} b^{-\tau u} \left(\frac{1}{b^\tau - 1} + \sum_{s=1}^{d}(1 - l_s)b^{s\tau}\right) \tag{13}$$

with $r = u2^d + \langle l_1 \dots l_d \rangle_2$ and

$$v = \frac{1}{\tau^2}\left(\frac{\Gamma(2\tau)\ln b}{(\Gamma(\tau))^2}\left(1 + \frac{2b^{-\tau d}}{b^\tau - 1} + \sum_{s=1}^{d} \frac{2b^{-\tau s}}{\left(1 + \frac{(b-1)b^{-s}}{b-1+b^{-d}}\right)^{2\tau}}\right) - 1\right).$$

The last factor of the estimator (12) comes from applying the second-order delta method [12] and corrects some bias similar to the last factor in (11).

As τ is a free parameter, it is ideally chosen to minimize the variance approximated by (see extended paper [22])

$$\text{Var}(\hat{n}/n) \approx v/m + O(m^{-2}). \tag{14}$$

For ULL with $b = 2$ and $d = 2$, numerical minimization gives $v \approx 0.616990$ for $\tau \approx 0.755097$. In this case (13) can be written as

$$g(r) := 2^{-\tau\lfloor r/4 \rfloor} \eta_{r \bmod 4} \tag{15}$$

with coefficients $\eta_0 \approx 4.841356$, $\eta_1 \approx 2.539198$, $\eta_2 \approx 3.477312$, $\eta_3 \approx 1.175153$. The corresponding MVP is $8v = 4.935917$ which is slightly greater than the theoretical MVP (8) with a value of 4.631289. Hence, the efficiency of the GRA estimator is 93.8%.

Figure 3a shows the asymptotic GRA estimator efficiency as $m \to \infty$ relative to (8) over the base b for $q = 6$ and for various values of d. The GRA estimator is very efficient for $d = 0$ and therefore for HLL. It can be clearly seen that its efficiency is lower for other configurations with $d \ge 1$. Therefore, we investigated if we can find a more efficient estimator for ULL, that is as simple and cheap to evaluate as the GRA estimator.

3.3 New FGRA Estimator

Our idea is to further generalize the GRA estimator, by choosing not only τ but also the coefficients η_0, η_1, η_2, and η_3 in (15) such that

the variance is minimized. This is why we call the new estimator further generalized remaining area (FGRA) estimator. The optimal values for η_0, η_1, η_2, and η_3 for fixed τ are given by [22]

$$\eta_j = \frac{\ln b}{\Gamma(\tau)} \frac{\omega_j(\tau)}{\omega_j(2\tau)} \left(\frac{\omega_0^2(\tau)}{\omega_0(2\tau)} + \frac{\omega_1^2(\tau)}{\omega_1(2\tau)} + \frac{\omega_2^2(\tau)}{\omega_2(2\tau)} + \frac{\omega_3^2(\tau)}{\omega_3(2\tau)} \right)^{-1} \tag{16}$$

where the functions ω_0, ω_1, ω_2, and ω_3 are defined as

$$\omega_0(\tau) := \tfrac{1}{(b^3-b+1)^\tau} - \tfrac{1}{b^{3\tau}},$$
$$\omega_1(\tau) := \tfrac{1}{(b^2-b+1)^\tau} - \tfrac{1}{b^{2\tau}} - \tfrac{1}{(b^3-b+1)^\tau} + \tfrac{1}{b^{3\tau}},$$
$$\omega_2(\tau) := \tfrac{1}{(b^3-b^2+1)^\tau} - \tfrac{1}{(b^3-b^2+b)^\tau} - \tfrac{1}{(b^3-b+1)^\tau} + \tfrac{1}{b^{3\tau}},$$
$$\omega_3(\tau) := \tfrac{1}{(b^3-b+1)^\tau} - \tfrac{1}{(b^3-b^2+1)^\tau} + \tfrac{1}{(b^3-b^2+b)^\tau} - \tfrac{1}{b^{3\tau}} - \tfrac{1}{(b^2-b+1)^\tau} + \tfrac{1}{b^{2\tau}} + 1 - \tfrac{1}{b^\tau}.$$

The minimum variance is then given by (14) with

$$v = \frac{1}{\tau^2} \left(\frac{\Gamma(2\tau)\ln b}{(\Gamma(\tau))^2} \left(\frac{\omega_0^2(\tau)}{\omega_0(2\tau)} + \frac{\omega_1^2(\tau)}{\omega_1(2\tau)} + \frac{\omega_2^2(\tau)}{\omega_2(2\tau)} + \frac{\omega_3^2(\tau)}{\omega_3(2\tau)} \right)^{-1} - 1 \right).$$

Numerical minimization of this expression with regard to τ for the ULL case with $b = 2$ and $d = 2$ yields $\tau \approx 0.819491$ for which $v \approx 0.611893$ and the corresponding coefficients following (16) are

$$\eta_0 \approx 4.663135, \eta_1 \approx 2.137850, \eta_2 \approx 2.781145, \eta_3 \approx 0.982408.$$

The resulting MVP $= 8v \approx 4.895145$ corresponds to an efficiency of 94.6%, which is a small improvement over the GRA estimator as shown in Figure 3b.

An interesting but not optimal choice would be $\tau = 1$, since this would make exponentiations in (12) and (15) particularly cheap. The corresponding coefficients would be $\eta_0 \approx 6.037409$, $\eta_1 \approx 2.415940$, $\eta_2 \approx 3.364340$, $\eta_3 \approx 0.934924$, which result in $v \approx 0.617163$ and a MVP of $8v \approx 4.937304$ corresponding to an efficiency of 93.8%. In contrast, the efficiency of the GRA estimator would already drop to 91.3% if $\tau = 1$ is chosen.

3.4 Corrections for GRA/FGRA Estimators

Both the GRA as well as our newly proposed FGRA estimator assume that register values are distributed according to (5) rather than (4). However, as mentioned in Section 2.1, this assumption is not valid, if a significant number of register values lies outside of $[(d+1)2^d, w2^d)$. For small (large) distinct counts, there are register values below (above) this range, and the presented plain GRA/FGRA estimators would no longer work without the corrections introduced below.

Previous work has often not properly addressed this problem. For example, the estimation error of the original HLL estimator [23] exceeded the predicted asymptotic error significantly for certain distinct counts. To be useful in practice, the distinct count estimate should be within the same error bounds over the entire range. While the estimator for HLL was patched in the meantime [20, 27, 39, 43, 51], recent algorithms such as EHLL [33] or HyperLogLogLog [28] still suggest, like the original HLL algorithm [23], to switch between different estimators. However, the corresponding transitions are not seamless which results in discontinuous estimation errors [20]. In the following we present a technique, which generalizes ideas presented in our earlier works [20, 21], to compose estimators of kind (12) with specific estimators for small and large distinct counts such that the resulting estimator works seamlessly over the whole value range.

A register state $r = u2^d + \langle l_1 \dots l_d \rangle_2$ allows to draw conclusions about the occurrence of certain events A_k as introduced in Section 2.1. For example, if $u < w$, we know for sure that A_u has occurred while events A_k with $k > u$ have not occurred. Bit l_s indicates the occurrence of A_{u-s} as long as $u > s$. However, we know nothing about events A_k with $k < u - d$ or $k \le 0$.

We can derive the PMF $\tilde{\rho}_{\text{reg}}(\tilde{r}|r,n)$, conditioned on the secured information about event occurrences we know from r, for register values $\tilde{r}$ following the simple model (5). For the case $d+1 \le u < w$ corresponding to $(d+1)2^d \le r < w2^d$, $\tilde{r}$ and r will always be equal, which yields $\tilde{\rho}_{\text{reg}}(\tilde{r}|r,n) = [\tilde{r} = r]$ when using the Iverson bracket notation. A complete derivation of $\tilde{\rho}_{\text{reg}}(\tilde{r}|r,n)$ for ULL with $d = 2$ and $b = 2$ is given in the extended paper [22].

Assuming that the distinct count is equal to the estimate $\hat{n}_{\text{alt}}$ obtained by an alternative distinct count estimator, we define the corrected register contribution $g_{\text{corr}}(r)$, to be used in (12) as replacement for $g(r)$, as the expected register contribution of $g(\tilde{r})$ where $\tilde{r}$ follows the conditional PMF $\tilde{\rho}_{\text{reg}}(\tilde{r}|r, n = \hat{n}_{\text{alt}})$

$$g_{\text{corr}}(r) = \textstyle\sum_{\tilde{r}=-\infty}^{\infty} \tilde{\rho}_{\text{reg}}(\tilde{r}|r, n = \hat{n}_{\text{alt}}) g(\tilde{r}). \tag{17}$$

This approach leads to $g_{\text{corr}}(r) = g(r)$ for the case $(d+1)2^d \le r < w2^d$. Hence, if all registers are in this intermediate range, the estimator (12) remains unchanged. This is expected as the PMFs (4) and (5) are equivalent in this case. However, the corrected contributions $g_{\text{corr}}(r)$ of very small and very large register values will differ significantly from $g(r)$. For ULL with $d = 2$ and $b = 2$ where $g(r)$ follows (15), the corrected register contributions can be written as

$$g_{\text{corr}}(r) = \begin{cases} \sigma(\hat{z}_0) & r{=}0, \\ 2^{-\tau}\psi(\hat{z}_0) & r{=}4, \\ 4^{-\tau}(\hat{z}_0(\eta_0-\eta_1)+\eta_1) & r{=}8, \\ 4^{-\tau}(\hat{z}_0(\eta_2-\eta_3)+\eta_3) & r{=}10, \\ 2^{-\tau\lfloor r/4 \rfloor}\eta_{r \bmod 4} & r{\in}[12,4w), \\ \frac{\hat{z}_w(1+\sqrt{\hat{z}_w})\eta_0 + 2^{-\tau}\sqrt{\hat{z}_w}(\hat{z}_w(\eta_0-\eta_2)+\eta_2)+\varphi(\sqrt{\hat{z}_w})}{2^{\tau w}(1+\sqrt{\hat{z}_w})(1+\hat{z}_w)} & r{=}4w, \\ \frac{\hat{z}_w(1+\sqrt{\hat{z}_w})\eta_1 + 2^{-\tau}\sqrt{\hat{z}_w}(\hat{z}_w(\eta_0-\eta_2)+\eta_2)+\varphi(\sqrt{\hat{z}_w})}{2^{\tau w}(1+\sqrt{\hat{z}_w})(1+\hat{z}_w)} & r{=}4w{+}1, \\ \frac{\hat{z}_w(1+\sqrt{\hat{z}_w})\eta_2 + 2^{-\tau}\sqrt{\hat{z}_w}(\hat{z}_w(\eta_1-\eta_3)+\eta_3)+\varphi(\sqrt{\hat{z}_w})}{2^{\tau w}(1+\sqrt{\hat{z}_w})(1+\hat{z}_w)} & r{=}4w{+}2, \\ \frac{\hat{z}_w(1+\sqrt{\hat{z}_w})\eta_3 + 2^{-\tau}\sqrt{\hat{z}_w}(\hat{z}_w(\eta_1-\eta_3)+\eta_3)+\varphi(\sqrt{\hat{z}_w})}{2^{\tau w}(1+\sqrt{\hat{z}_w})(1+\hat{z}_w)} & r{=}4w{+}3 \end{cases} \tag{18}$$

(see extended paper [22]). The functions ψ, σ, and φ are given by

$$\psi(z) := z(z(z(\eta_0-\eta_1-\eta_2+\eta_3)+(\eta_2-\eta_3))+(\eta_1-\eta_3))+\eta_3, \tag{19}$$

$$\sigma(z) := \tfrac{1}{z}\textstyle\sum_{u=0}^{\infty} 2^{\tau u}(z^{2^u} - z^{2^{u+1}})\psi(z^{2^{u+1}}), \tag{20}$$

$$\varphi(z) := \tfrac{4^{-\tau}}{1-z}\textstyle\sum_{u=0}^{\infty} 2^{-\tau u}(z^{2^{-u-1}} - z^{2^{-u}})\psi(z^{2^{-u}}) \tag{21}$$
$$= \tfrac{4^{-\tau}}{2-2^{-\tau}}\left(\frac{2\psi(z)\sqrt{z}}{1+\sqrt{z}} + \textstyle\sum_{u=1}^{\infty} \frac{z^{2^{-u-1}}\left(2\psi(z^{2^{-u}}) - (z^{2^{-u-1}} + z^{2^{-u}})\psi(z^{2^{-u+1}})\right)}{2^{\tau u}\prod_{j=1}^{u+1}(1+z^{2^{-j}})} \right).$$

Furthermore, $\hat{z}_0$ and $\hat{z}_w$ are defined as $\hat{z}_0 := e^{-\frac{\hat{n}_{\text{alt}}}{m}}$ and $\hat{z}_w := e^{-\frac{\hat{n}_{\text{alt}}}{m2^w}}$ and are therefore estimates of $z_0 = e^{-\frac{n}{m}}$ and $z_w = e^{-\frac{n}{m2^w}}$, respectively. According to (18), $\hat{z}_0$ is only needed for register values from $\{0, 4, 8, 10\}$. Therefore, the estimator $\hat{z}_0$ must work well only for small distinct counts. Similarly, the estimator $\hat{z}_w$ must work only for large distinct counts, when there are registers greater than or equal to $4w$. Corresponding estimators for both cases are presented in the next sections.

The numerical evaluation of σ is very cheap, because only basic mathematical operations are involved and the infinite series converges quickly. The convergence is slowest for arguments $\hat{z}_0$ close to 1. The largest arguments smaller than 1 occur for distinct counts equal to 1, which implies $\hat{z}_0 \approx e^{-\frac{1}{m}}$. However, even in this extreme case, empirical analysis showed numerical convergence after the first $p+7$ terms for any precision $p \in [3, 26]$ when using double-precision floating-point arithmetic. For not too small values of p, the estimation costs are dominated by the iteration over all $m = 2^p$ registers to sum up the individual register contributions.

When using 64-bit hash values, the maximum update value is given by $w = 65 - p$ (see Section 2.1). In this case, registers with values $\geq 4w$ are very unlikely in practice and the evaluation of φ is rarely needed. Nevertheless, it also can be computed efficiently with the second (more complex appearing) expression for φ, which converges numerically after at most 22 terms for any $p \in [3, 26]$.

3.5 Estimator for Small Distinct Counts

The original HLL algorithm [23] switches, in case of small distinct counts with many registers equal to zero, to the estimator

$$\hat{n}_{\text{low}} = m \ln(m/c_0), \tag{22}$$

known from probabilistic linear counting [47]. $c_0 := |\{i|r_i = 0\}|$ denotes the number of registers that have never been updated. This estimator corresponds to the ML estimator, when considering just the number of registers with $r_i = 0$ and using $\Pr(r_i = 0) = z_0 = e^{-\frac{n}{m}}$ which follows from (4) for $b = 2$. For HLL, the combination of this estimator with (17) leads to the same correction terms as we have previously derived in a different way [20]. The resulting so-called corrected raw (CR) estimator was shown to be nearly as efficient as the ML estimator.

This finding provides confidence that the same approach also works for ULL to correct the GRA/FGRA estimator. Even though (22) could be used again to estimate small distinct counts, we found a simple estimator that is able to exploit more information. We consider the four smallest possible register states $r_i \in \{0, 4, 8, 10\}$ with corresponding probabilities following (4) and apply the ML method to estimate $z_0 = e^{-\frac{n}{m}}$. As shown in the extended paper [22] the resulting ML estimator can be written as

$$\hat{z}_0 = ((\sqrt{\beta^2 + 4\alpha\gamma} - \beta)/(2\alpha))^4. \tag{23}$$

Here α, β, γ are defined as

$$\alpha := m+3(c_0+c_4+c_8+c_{10}), \quad \beta := m-c_0-c_4, \quad \gamma := 4c_0+2c_4+3c_8+c_{10}$$

and $c_j := |\{i|r_i = j\}|$ is again the number of registers with value j.

3.6 Estimator for Large Distinct Counts

An estimator for large distinct counts can be found in a similar way through ML estimation when considering the largest 4 register states $r_i \in \{4w, 4w+1, 4w+2, 4w+3\}$. The ML estimator for $z_w = e^{-\frac{n}{m2^w}}$ can be written as [22]

$$\hat{z}_w = \sqrt{(\sqrt{\beta^2 + 4\alpha\gamma} - \beta)/(2\alpha)}. \tag{24}$$

Here α, β, γ are defined as

$$\alpha := m+3(c_{4w}+c_{4w+1}+c_{4w+2}+c_{4w+3}), \quad \beta := c_{4w}+c_{4w+1}+2c_{4w+2}+2c_{4w+3},$$
$$\gamma := m+2c_{4w}+c_{4w+2}-c_{4w+3}.$$

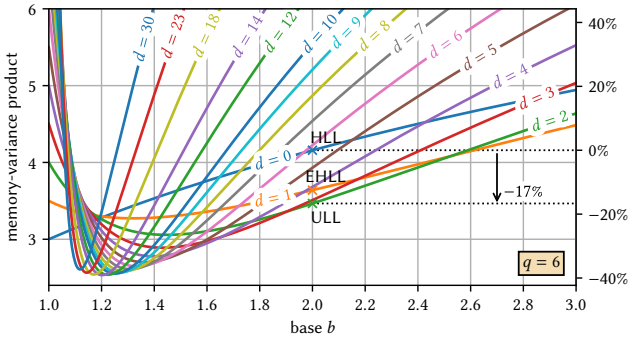

Figure 4: The memory-variance product (MVP) (26) as a function of the base b for $q = 6$ and various values of d when using martingale estimation. The MVP of UltraLogLog (ULL) is 17% smaller compared to HyperLogLog (HLL).

3.7 Martingale Estimator

If the data is not distributed and merging of sketches is not needed, the distinct count can be estimated in an online fashion by incremental updates, which is known as martingale or historic inverse probability (HIP) estimation [16, 42] and which even leads to smaller estimation errors. This estimation approach can also be used for ULL. It keeps track of the state change probability μ. Initially, $\mu = 1$ as the first update will certainly change the state. Whenever a register is changed, the probability of state changes for further elements decreases. The probability, that a new unseen element changes the ULL state is given by

$$\mu(r_0, \ldots, r_{m-1}) = \sum_{i=0}^{m-1} h(r_i).$$

$h(r_i)$ returns the probability that register r_i is changed with the next new element. Obviously, we have $h(0) = \frac{1}{m}$ and $h(4w+3) = 0$ for the smallest and largest possible states, respectively. In the general case we have (see extended paper [22])

$$\begin{aligned} &h(0) = \tfrac{1}{m}, \quad h(4) = \tfrac{1}{2m}, \quad h(8) = \tfrac{3}{4m}, \quad h(10) = \tfrac{1}{4m}, \\ &h(r) = \tfrac{7-2l_1-4l_2}{2^u m} \quad \text{for } r = 4u + \langle l_1 l_2 \rangle_2 \text{ and } 3 \leq u < w, \\ &h(r) = \tfrac{3-l_1-2l_2}{2^{w-1} m} \quad \text{for } r = 4w + \langle l_1 l_2 \rangle_2. \end{aligned} \tag{25}$$

The martingale estimator is incremented with every state change by $\frac{1}{\mu}$ prior the update as demonstrated by Algorithm 2. μ itself can also be incrementally adjusted, such that the whole update takes constant time. The values of h can be stored in a lookup table. The martingale estimator is unbiased and optimal [37], but cannot be used if the data is distributed and mergeability is required. For our generalized data structure introduced in Section 2, the asymptotic MVP can be derived as (see extended paper [22])

$$\text{MVP} \approx (q+d)\tfrac{1}{2}\ln(b)(1 + \tfrac{b-d}{b-1}). \tag{26}$$

This expression is consistent with previous specific results reported for HLL [37] and EHLL [33] and is plotted in Figure 4 for $q = 6$ and various values of b and d. For $b = 2$, the ULL configuration with $d = 2$ is even optimal. ULL achieves MVP ≈ 3.4657 which is 17% less than for HLL with MVP ≈ 4.1589.

4 PRACTICAL IMPLEMENTATION

As registers of ULL have $q+d=8$ bits, they can be stored in a plain byte array. Algorithm 3 shows the update procedure for inserting an element. The actual input is the hash value of the element from which the register index i and an update value k are obtained. As for other probabilistic data structures it is essential to use a high-quality hash function such as WyHash [52], Komihash [45], or PolymurHash [35] whose outputs are in practice indistinguishable from true random integers. If 64-bit hash values are used, the maximum update value is limited to $w = 65 - p$ leading to a maximum distinct count of roughly $n_{\max} \approx m2^{w-1} = 2^{64}$ (cf. Section 2.3) which is more than sufficient for any realistic application.

Algorithm 3 updates the i-th register by first unpacking its value to a 64-bit value $x = \langle x_{63}x_{62}\dots x_1x_0\rangle_2$ (see Algorithm 5) where bit x_{j+1} indicates the occurrence of update value j. After setting the corresponding bit for the new update value k, x is finally packed again into a register value (see Algorithm 4), preserving just the information about the maximum update value u and whether update values $u-1$ and $u-2$ have already occurred.

Although the update algorithm for HLL (cf. Algorithm 1) looks simpler, in practice the implementation complexity is similar if one also includes the bit twiddling required to pack 6-bit registers densely into a byte array. ULL insertions can be implemented entirely branch-free, as exemplified by the implementation in our Hash4j Java library (see https://github.com/dynatrace-oss/hash4j), which however uses the transformation $r \to r + 4p - 8$ for non-zero registers, so that the largest possible register state is always $(4w+3) + 4p - 8 = 255$, the maximum value of a byte.

Algorithm 6 summarizes FGRA estimation including corrections for small and large distinct counts as described in Section 3.4. For intermediate counts, if registers are all in the range $[12, 4w)$, the algorithm just sums up the individual register contributions $g(r_i)$ given by (15). The values of g can be pre-computed and stored in a lookup table for all possible register values. Following (12), the final estimate is obtained by exponentiating the sum of all register contributions by $-\frac{1}{\tau}$ and multiplying by the precomputed factor λ_p. For small distinct counts, when there are any registers smaller than 12, thus $c_0 + c_4 + c_8 + c_{10} > 0$, the small range correction branch is executed, which requires computing the estimator described in Section 3.5. Similarly, for large distinct counts with registers greater than or equal to $4w$, equivalent to $c_{4w}+c_{4w+1}+c_{4w+2}+c_{4w+3} > 0$, the large range correction is applied based on the estimator presented in Section 3.6.

4.1 Merging and Downsizing

ULL sketches can be merged which is important if the data is distributed over space and/or time and partial results must be combined. The final result will only depend on the set of added elements and not on the insertion or merge order. It is even possible to merge ULL sketches with different precisions. For that, the sketch with higher precision must be reduced first to the lower precision of the other sketch.

Downsizing from precision p' to $p < p'$ is realized by merging the information of batches of $2^{p'-p}$ consecutive registers. The resulting register value depends not only on the individual register values, but also on the least significant $p' - p$ bits of the register indices. Only the value of the first register of a batch, whose index has $p' - p$ trailing zeros, is relevant. All other registers need to be treated differently, as the $p' - p$ trailing index bits would have affected the calculation of the number of leading zeros (NLZ) computation in Algorithm 3 when recorded with lower precision p.

Algorithm 7 shows in detail how an ULL sketch with precision p' can be merged in-place into another one with precision $p \le p'$. The algorithm simplifies a lot when the precisions are equal, because $p' = p$ implies $2^{p'-p} = 1$ and consequently the inner loop can be skipped entirely. In this case, and also due to the byte-sized registers, the ULL merge operation is very well suited for single-instruction multiple-data (SIMD) processing.

Algorithm 7 can also be applied to just downsize a ULL sketch from precision p' to p, if the sketch is simply added to an empty ULL sketch with precision p and all registers equal to zero, $r_i = 0$. In this case the first unpack operation can be skipped as it would always return zero.

4.2 Compatibility to HyperLogLog

When using the same hash function for elements, ULL can be implemented in a way that is compatible with an existing HLL implementation meaning that an ULL sketch can be mapped to a HLL sketch of same precision by just dropping the last 2 register bits corresponding to the transformation $\lfloor r_i^{(\mathrm{ULL})}/4 \rfloor \to r_i^{(\mathrm{HLL})}$. This allows to migrate to ULL, even if there is historical data that was recorded using HLL and still needs to be combined with newer

Algorithm 2: Incrementally updates the martingale estimate $\hat{n}_{\mathrm{martingale}}$ and the state change probability μ whenever a register is altered from r to r' ($r < r'$). Initially, $\hat{n}_{\mathrm{martingale}} = 0$ and $\mu = 1$.

$\hat{n}_{\mathrm{martingale}} \leftarrow \hat{n}_{\mathrm{martingale}} + \frac{1}{\mu}$ ▷ update estimate
$\mu \leftarrow \mu - h(r) + h(r')$ ▷ update state change probability, see (25)

Algorithm 3: Inserts an element with 64-bit hash value $\langle h_{63}h_{62}\dots h_0\rangle_2$ into an UltraLogLog with registers $r_0, r_1, \dots, r_{m-1}$, initial values $r_i = 0$, and $m = 2^p$ ($p \ge 3$).

$i \leftarrow \langle h_{63}h_{62}\dots h_{64-p}\rangle_2$ ▷ extract register index
$a \leftarrow \langle \underbrace{0\dots0}_{p} h_{63-p}h_{62-p}\dots h_0\rangle_2$ ▷ mask register index bits
$k \leftarrow \mathtt{nlz}(a) - p + 1$ ▷ update value $k \in [1, 65-p]$
$x \leftarrow \mathtt{unpack}(r_i)$ ▷ see Algorithm 5 for unpack
$x \leftarrow x$ **or** 2^{k+1} ▷ bitwise or-operation
$r_i \leftarrow \mathtt{pack}(x)$ ▷ see Algorithm 4 for pack

Algorithm 4: Packs a 64-bit value $x \ge 4$ into an 8-bit register. $x = \langle x_{63}x_{62}\dots x_1x_0\rangle_2$ is interpreted as bitset where bit x_{k+1} indicates the occurrence of update value k.

function $\mathtt{pack}(x)$
  $u \leftarrow 62 - \mathtt{nlz}(x)$ ▷ extract maximum update value
  return $4u + \langle x_u x_{u-1}\rangle_2$ ▷ return 8-bit register value, is always ≥ 4

Algorithm 5: Unpacks an 8-bit register $r = \langle q_7q_6\dots q_0\rangle_2$ into a 64-bit integer indicating the occurrence of update values if interpreted as bitset.

function $\mathtt{unpack}(r)$
  if $r < 4$ **then return** 0 ▷ special case for initial state when $r = 0$
  $u \leftarrow \langle q_7q_6\dots q_2\rangle_2$ ▷ get maximum update value, same as $\lfloor r/4 \rfloor$
  return $\langle \underbrace{0\dots0}_{62-u} 1 q_1 q_0 \underbrace{0\dots0}_{u-1}\rangle_2$ ▷ return 64-bit value, is always ≥ 4

Algorithm 6: Estimates the number of distinct elements from a given UltraLogLog with registers $r_0, r_1, \ldots, r_{m-1}$ and $m = 2^p$ ($p \geq 3$) using the FGRA estimator. The constants are defined as $w := 65 - p$, $\tau := 0.8194911375910897$, $\upsilon := 0.6118931496978437$, $\eta_0 := 4.663135422063788$, $\eta_1 := 2.1378502137958524$, $\eta_2 := 2.781144650979996$, $\eta_3 := 0.9824082545153715$, and $\lambda_p := m^{1+\frac{1}{\tau}}/(1+\frac{1+\tau}{2}\frac{\upsilon}{m})$.

$s \leftarrow 0$ ▷ used to sum up $\sum_{i=0}^{m-1} g_{\text{corr}}(r_i)$, see (18)
$(c_0, c_4, c_8, c_{10}, c_{4w}, c_{4w+1}, c_{4w+2}, c_{4w+3}) \leftarrow (0,0,0,0,0,0,0,0)$
for $i \leftarrow 0$ **to** $m-1$ **do** ▷ iterate over all m registers
  if $r_i < 12$ **then**
    if $r_i = 0$ **then** $c_0 \leftarrow c_0 + 1$
    if $r_i = 4$ **then** $c_4 \leftarrow c_4 + 1$
    if $r_i = 8$ **then** $c_8 \leftarrow c_8 + 1$
    if $r_i = 10$ **then** $c_{10} \leftarrow c_{10} + 1$
  else if $r_i < 4w$ **then**
    $s \leftarrow s + g(r_i)$ ▷ see (15), $g(r_i)$ can be precomputed
  else
    if $r_i = 4w$ **then** $c_{4w} \leftarrow c_{4w} + 1$
    if $r_i = 4w+1$ **then** $c_{4w+1} \leftarrow c_{4w+1} + 1$
    if $r_i = 4w+2$ **then** $c_{4w+2} \leftarrow c_{4w+2} + 1$
    if $r_i = 4w+3$ **then** $c_{4w+3} \leftarrow c_{4w+3} + 1$
if $c_0 + c_4 + c_8 + c_{10} > 0$ **then** ▷ small range correction
  $\alpha \leftarrow m + 3(c_0 + c_4 + c_8 + c_{10})$
  $\beta \leftarrow m - c_0 - c_4$
  $\gamma \leftarrow 4c_0 + 2c_4 + 3c_8 + c_{10}$
  $z \leftarrow ((\sqrt{\beta^2 + 4\alpha\gamma} - \beta)/(2\alpha))^4$ ▷ see (23)
  if $c_0 > 0$ **then** $s \leftarrow s + c_0 \cdot \sigma(z)$ ▷ for σ see (20)
  if $c_4 > 0$ **then** $s \leftarrow s + c_4 \cdot 2^{-\tau} \cdot \psi(z)$ ▷ for ψ see (19)
  if $c_8 > 0$ **then** $s \leftarrow s + c_8 \cdot (4^{-\tau} \cdot (z \cdot (\eta_0 - \eta_1) + \eta_1))$
  if $c_{10} > 0$ **then** $s \leftarrow s + c_{10} \cdot (4^{-\tau} \cdot (z \cdot (\eta_2 - \eta_3) + \eta_3))$
if $c_{4w} + c_{4w+1} + c_{4w+2} + c_{4w+3} > 0$ **then** ▷ large range correction
  $\alpha \leftarrow m + 3(c_{4w} + c_{4w+1} + c_{4w+2} + c_{4w+3})$
  $\beta \leftarrow c_{4w} + c_{4w+1} + 2c_{4w+2} + 2c_{4w+3}$
  $\gamma \leftarrow m + 2c_{4w} + c_{4w+2} - c_{4w+3}$
  $z \leftarrow \sqrt{(\sqrt{\beta^2 + 4\alpha\gamma} - \beta)/(2\alpha)}$ ▷ see (24)
  $z' \leftarrow \sqrt{z}$
  $s' \leftarrow z \cdot (1 + z') \cdot (\eta_0 \cdot c_{4w} + \eta_1 \cdot c_{4w+1} + \eta_2 \cdot c_{4w+2} + \eta_3 \cdot c_{4w+3})$
  $s' \leftarrow s' + 2^{-\tau} \cdot z' \cdot (z \cdot (\eta_0 - \eta_2) + \eta_2) \cdot (c_{4w} + c_{4w+1})$
  $s' \leftarrow s' + 2^{-\tau} \cdot z' \cdot (z \cdot (\eta_1 - \eta_3) + \eta_3) \cdot (c_{4w+2} + c_{4w+3})$
  $s' \leftarrow s' + \varphi(z') \cdot (c_{4w} + c_{4w+1} + c_{4w+2} + c_{4w+3})$ ▷ for φ see (21)
  $s \leftarrow s + s'/(2^{\tau w} \cdot (1 + z') \cdot (1 + z))$
return $\lambda_p \cdot s^{-1/\tau}$ ▷ return distinct count estimate, see (12)

Algorithm 7: Merges an UltraLogLog with registers $r'_0, r'_1, \ldots, r'_{m'-1}$ and $m' = 2^{p'}$ into another UltraLogLog with registers $r_0, r_1, \ldots, r_{m-1}$ and $m = 2^p$ where $p \leq p'$. This algorithm also allows to downsize an existing UltraLogLog by merging it into an empty UltraLogLog ($r_i = 0$) with smaller precision parameter $p < p'$.

$j \leftarrow 0$
for $i \leftarrow 0$ **to** $m-1$ **do**
  $x \leftarrow \texttt{unpack}(r_i)$ **or** $(\texttt{unpack}(r'_j) \cdot 2^{p'-p})$
    ▷ bitwise or-operation, see Algorithm 5 for unpack
  $j \leftarrow j + 1$
  for $l \leftarrow 1$ **to** $2^{p'-p} - 1$ **do**
    if $r'_j \neq 0$ **then**
      $k \leftarrow \texttt{nlz}(l) + p' - p - 63$ ▷ l is assumed to have 64 bits
      $x \leftarrow x$ **or** 2^{k+1} ▷ bitwise or-operation
    $j \leftarrow j + 1$
  if $x \neq 0$ **then** $r_i \leftarrow \texttt{pack}(x)$ ▷ see Algorithm 4 for pack

data. Our Hash4j library also contains a HLL implementation that is compatible with its ULL implementation.

5 EXPERIMENTS

Various experiments have been conducted to confirm the theoretical results and to demonstrate the practicality of ULL. For better reproduction, instructions and source code for all experiments including figure generation have been published at https://github.com/dynatrace-research/ultraloglog-paper. The simulations used the ULL and HLL implementations available in our open-source Hash4j library (v0.17.0) and were executed on an Amazon EC2 c5.metal instance running Ubuntu Server 22.04 LTS.

Extensive empirical tests [44] have shown that the output of modern hash functions such as WyHash [52], Komihash [45], or PolymurHash [35] can be considered like uniform random values. Otherwise, field-tested probabilistic data structures like HLL would not work. This fact allows us to simplify the experiments and perform them without real or artificially generated data. Insertion of a new element can be simulated by simply generating a 64-bit random value to be used directly as the hash value of the inserted element in Algorithm 3.

5.1 Estimation Error

To simulate the estimation error for a predefined distinct count value, the estimate is computed after updating the ULL sketch using Algorithm 3 with a corresponding number of random values and finally compared against the true distinct count. By repeating this process with many different random sequences, in our experiments 100 000, the bias and the root-mean-square error (RMSE) can be empirically determined. However, this approach becomes computationally infeasible for distinct counts beyond 1 million and we need to switch to a different strategy.

After the first million of insertions, for which a random value was generated each time, we just generate the waiting time (the number of distinct count increments) until a register is processed with a certain update value the next time. For each insertion, the probability that a register is updated with any possible value $k \in [1, 65-p]$ is given by $1/(m2^{\min(k,64-p)})$. Therefore, the number of distinct count increments until a register is updated with a specific value k the next time is geometrically distributed with corresponding success probability. In this way, we determine the next update time for each register and for each possible update value. Since the same update value can only modify a register once, we do not need to consider further updates which might occur with the same value for the same register. Knowing these $m \times (65-p)$ distinct count values in advance, where the state may change, enables us to make large distinct count increments, resulting in a huge speedup. This eventually allowed us to simulate the estimation error for distinct counts up to values of 10^{21} and also to test the presented estimators over the entire operating range.

Figure 5 shows the empirically determined relative bias and RMSE as well as the theoretical RMSE given by $\sqrt{\text{MVP}/(8m)}$ for the FGRA, ML, and the martingale estimator for precisions $p \in \{8, 12, 16\}$. For intermediate distinct counts, for which the assumptions of our theoretical analysis hold, perfect agreement with theory is observed. For small distinct counts, the difference in efficiency between ML and FGRA is more significant, but not particularly relevant in practice, as the estimation errors in both cases are well below the theoretically predicted errors. Interestingly, the

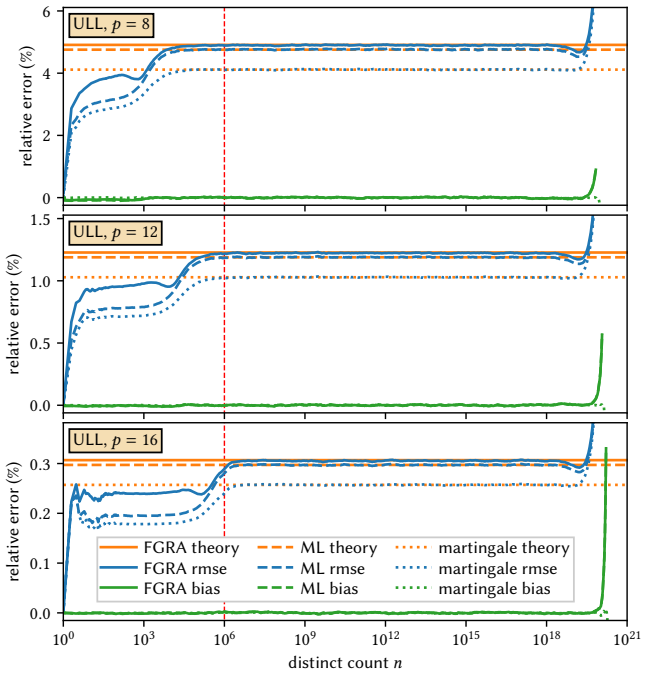

Figure 5: The relative bias and the RMSE for the FGRA, ML, and the martingale estimator for precisions $p \in \{8, 12, 16\}$ obtained from 100 000 simulation runs. The theoretically predicted errors perfectly match the experimental results. Individual insertions were simulated up to a distinct count of 10^6 before switching to the fast simulation strategy.

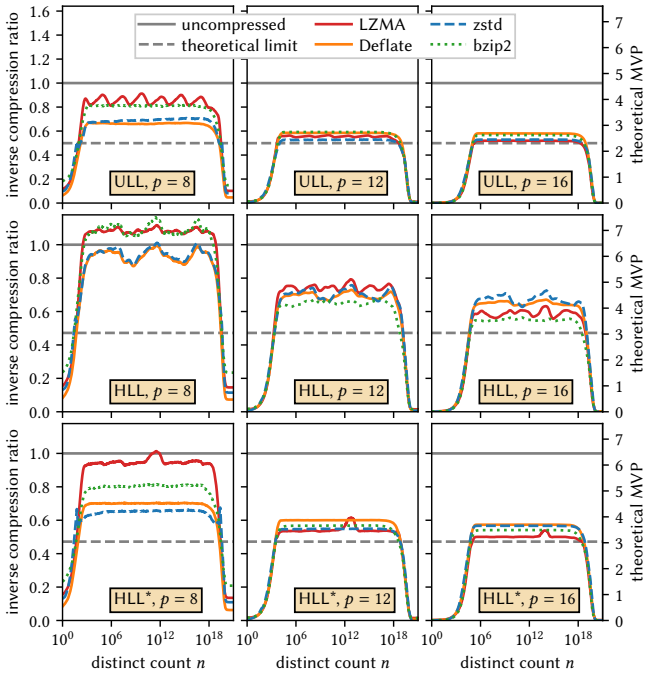

Figure 6: The average inverse compression ratio for HLL and ULL over the distinct count for various standard compression algorithms. For HLL*, the 6-bit registers are mapped to individual bytes before compression. The gray dashed line indicates the theoretical lower bound for intermediate distinct counts given by (10).

estimation error also decreases slightly near the end of the operating range, which is as predicted on the order of $2^{64} \approx 1.8 \cdot 10^{19}$. The estimators are essentially unbiased. The tiny bias which appears for small precisions for the ML and FGRA estimators can be ignored in practice as it is much smaller than the RMSE.

Previous experiments have already confirmed that the proposed estimators for HLL [20, 37] and EHLL [33] do not undercut and at best reach the corresponding theoretical MVPs given by (8) and (26). Therefore, the perfect agreement with the theory for ULL observed in our experiments finally proves the claimed and theoretically predicted improvements in storage efficiencies over the state of the art as shown in Figures 1 and 4.

5.2 Compression

To analyze the compressibility of the state, we generated 100 random sketches for predefined distinct counts and applied various standard compression algorithms (LZMA, Deflate, zstd, and bzip2) from the Apache Compress Java library [1]. The corresponding average inverse compression ratios over the distinct count are shown in Figure 6 for HLL and ULL. The results indicate that the algorithms generally work better for ULL than HLL, since the compressed size is closer to the theoretical limit given by the Shannon entropy (9). Interestingly, the compression for HLL improves, if its 6-bit registers are first represented as individual bytes (cf. HLL* in Figure 6). Even though this leads to compression ratios that are sometimes better than for ULL, ULL is still more memory-efficient overall due to the significant lower MVP when uncompressed.

The theoretical lower bound applies only to intermediate distinct counts under the validity of the simple model (5). The observed high compressibility at small and large distinct counts is due to the large number of initial and saturated registers, respectively. Therefore, many HLL implementations support a sparse mode that encodes only non-zero registers [27]. We expect that this and other lossless compression techniques developed for HLL [2, 28] can also be applied to ULL, but obviously at the cost of slower updates and memory reallocations.

5.3 Performance

Since low processing costs are also critical for practical use, we measured the average time for inserting a given number of distinct elements into HLL and ULL sketches configured with precisions $p \in \{8, 10, 12, 14, 16\}$. To reduce the impact of variable processor frequencies, Turbo Boost was disabled on the used Amazon EC2 c5.metal instance by setting the processor P-state to 1 [4]. The benchmarks were executed using OpenJDK 21.0.2.

The measurements shown in Figure 7 also include sketch initialization as well as the generation of random numbers which were used instead of hash values as described before. The initialization costs dominate for small distinct counts, which are proportional to $m = 2^p$ due to the allocated register array. For large distinct counts

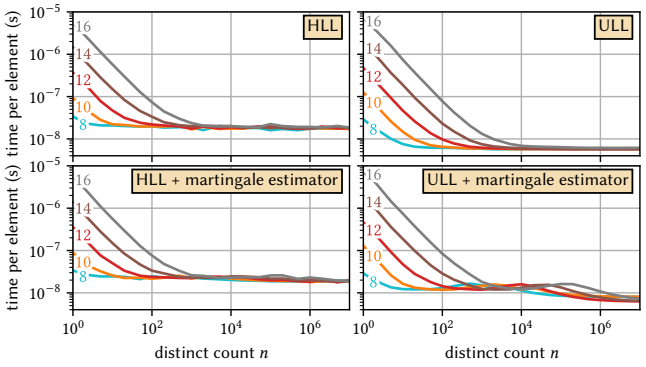

Figure 7: Average insertion time per element when initializing a HLL or ULL sketch with $p \in \{8, 10, 12, 14, 16\}$ and adding the given number of distinct elements.

the initialization costs can be neglected, and the average time per insertion converges to a value that is essentially independent of p. There, it becomes apparent that HLL insertions are significantly slower, mainly caused by the overhead of accessing the 6-bit registers packed in a byte array. If the insertions are accompanied by martingale estimator updates following Algorithm 2, the difference is less clear. Only for large distinct counts, where changes of register values become less frequent, ULL is significantly faster again.

We also investigated the estimation costs for precisions $p \in [8, 16]$ and $n \in \{1, 2, 5, 10, 20, 50, \ldots, 10^7\}$ as shown in Figure 8. We considered the ML estimator for both, the new FGRA estimator for ULL, and the corrected raw (CR) estimator for HLL [20]. The latter corresponds to the GRA estimator with $\tau = 1$ which is for HLL almost as efficient as the ML and optimal GRA estimators [46]. ML estimation is more expensive for ULL than for HLL for larger distinct counts. The CR and the FGRA estimator are most of the time significantly faster than their ML counterparts. The costs of the CR estimator are roughly constant, while the FGRA estimator peaks briefly before it falls back on the same level as the CR estimator for equal p. Our investigations showed that this peak is related to the more difficult branch prediction in Algorithm 6, when significant portions of registers have values from $\{0, 4, 8, 10\}$ and also values greater than or equal to 12.

A fair comparison must take into account that an ULL sketch with same precision generally leads to smaller errors. According to (1) the theoretical relative errors for HLL and ULL are given by $\sqrt{\mathrm{MVP}^{(\mathrm{HLL})}/(6 \cdot 2^{p^{(\mathrm{HLL})}})}$ and $\sqrt{\mathrm{MVP}^{(\mathrm{ULL})}/(8 \cdot 2^{p^{(\mathrm{ULL})}})}$. They are approximately equal if $p^{(\mathrm{HLL})} \approx p^{(\mathrm{ULL})} + 0.8$, which means that an ULL with $p = 8$ is rather compared to a HLL with $p = 9$. Figure 9 shows the average estimation time for $n = 10^6$ and averaged over the cases $n \in \{1, 2, 5, 10, 20, 50, \ldots, 10^7\}$, which shows that FGRA estimation from an ULL is often faster (except for the peaks) than CR estimation from an HLL of equivalent precision.

6 FUTURE WORK

Although the ML estimator has been shown to be efficient and achieves the Cramér-Rao bound, it is slower than the FGRA estimator, which in turn has a worse efficiency of 94.6%. Improving

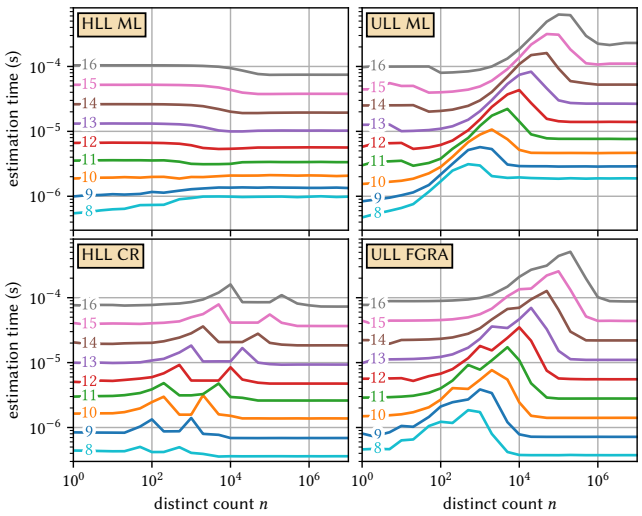

Figure 8: Average estimation time over the true distinct count for HLL and ULL for various precisions $p \in [8, 16]$.

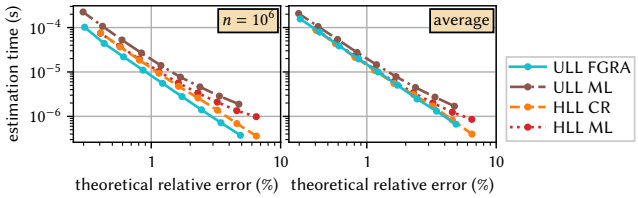

Figure 9: Estimation time for $n = 10^6$ (left) and averaged over the cases $n \in \{1, 2, 5, 10, 20, 50, \ldots, 10^7\}$ (right) versus the theoretical relative error when varying $p \in [8, 16]$.

the ML solver could be a way to achieve better estimation performance. Possibly, an alternative estimation method may also lead to a faster and even more efficient estimator. Another interesting question is whether parameter choices with smaller MVPs, as discussed in Section 2.3, can be turned into practical data structures with efficient estimation algorithms. Finally, since HLL has also been successfully applied for set similarity estimation [20, 21, 32], what is the space efficiency of ULL in this context, and can a fast and robust estimation algorithm be found for that as well?

7 CONCLUSION

We derived the ULL sketch as a special case of a generalized data structure unifying HLL, EHLL, and PCSA. The theoretically predicted space savings of 28%, 24%, and 17% over HLL when using the ML, FGRA, and martingale estimator, respectively, were perfectly matched by our experiments. The byte-sized registers lead to faster recording speed as well as better compressibility. Since ULL also has the same properties as HLL (constant-time insertions, idempotency, mergeability, reproducibility, reducibility), efficient and robust estimation with similar execution speed is possible, and even compatibility with HLL can be achieved, we believe that ULL has the potential to become the new standard algorithm for approximate distinct counting.

REFERENCES

[1] [n.d.]. *Apache Commons Compress.* Retrieved March 17, 2024 from https://commons.apache.org/proper/commons-compress/

[2] [n.d.]. *Apache Data Sketches: A software library of stochastic streaming algorithms.* Retrieved March 17, 2024 from https://datasketches.apache.org/

[3] [n.d.]. *Apache Data Sketches: Features Matrix for Distinct Count Sketches.* Retrieved March 17, 2024 from https://datasketches.apache.org/docs/DistinctCountFeaturesMatrix.html

[4] [n.d.]. *Processor state control for your EC2 instance.* Retrieved March 17, 2024 from https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/processor_state_control.html

[5] [n.d.]. *Snowflake Documentation: Estimating the Number of Distinct Values.* Retrieved March 17, 2024 from https://docs.snowflake.com/en/user-guide/querying-approximate-cardinality

[6] N. Alon, Y. Matias, and M. Szegedy. 1999. The Space Complexity of Approximating the Frequency Moments. *J. Comput. System Sci.* 58, 1 (1999), 137–147. https://doi.org/10.1006/jcss.1997.1545

[7] D. N. Baker and B. Langmead. 2019. Dashing: fast and accurate genomic distances with HyperLogLog. *Genome Biology* 20, 265 (2019). https://doi.org/10.1186/s13059-019-1875-0

[8] R. B. Basat, G. Einziger, S. L. Feibish, J. Moraney, and D. Raz. 2018. Network-wide routing-oblivious heavy hitters. In *Proceedings of the 16th Symposium on Architectures for Networking and Communications Systems (ANCS)*. 66–73. https://doi.org/10.1145/3230718.3230729

[9] P. Boldi, M. Rosa, and S. Vigna. 2011. HyperANF: Approximating the neighbourhood function of very large graphs on a budget. In *Proceedings of the 20th International Conference on World Wide Web (WWW)*. 625–634. https://doi.org/10.1145/1963405.1963493

[10] F. P. Breitwieser, D. N. Baker, and S. L. Salzberg. 2018. KrakenUniq: confident and fast metagenomics classification using unique k-mer counts. *Genome biology* 19, 1 (2018), 1–10. https://doi.org/10.1186/s13059-018-1568-0

[11] V. Bruschi, P. Reviriego, S. Pontarelli, D. Ting, and G. Bianchi. 2021. More Accurate Streaming Cardinality Estimation With Vectorized Counters. *IEEE Networking Letters* 3, 2 (2021), 75–79. https://doi.org/10.1109/LNET.2021.3076048

[12] G. Casella and R. L. Berger. 2002. *Statistical Inference* (2nd ed.). Duxbury, Pacific Grove, CA.

[13] Y. Chabchoub, R. Chiky, and B. Dogan. 2014. How can sliding HyperLogLog and EWMA detect port scan attacks in IP traffic? *EURASIP Journal on Information Security* 2014, 5 (2014). https://doi.org/10.1186/1687-417X-2014-5

[14] A. Chen, J. Cao, L. Shepp, and T. Nguyen. 2011. Distinct Counting With a Self-Learning Bitmap. *J. Amer. Statist. Assoc.* 106, 495 (2011), 879–890. https://doi.org/10.1198/jasa.2011.ap10217

[15] V. Clemens, L.-C. Schulz, M. Gartner, and D. Hausheer. 2023. DDoS Detection in P4 Using HYPERLOGLOG and COUNTMIN Sketches. In *Network Operations and Management Symposium (NOMS)*. 1–6. https://doi.org/10.1109/NOMS56928.2023.10154315

[16] E. Cohen. 2015. All-Distances Sketches, Revisited: HIP Estimators for Massive Graphs Analysis. *IEEE Transactions on Knowledge and Data Engineering* 27, 9 (2015), 2320–2334. https://doi.org/10.1109/TKDE.2015.2411606

[17] D. R. Cox and E. J. Snell. 1968. A General Definition of Residuals. *Journal of the Royal Statistical Society. Series B (Methodological)* 30, 2 (1968), 248–275. http://www.jstor.org/stable/2984505

[18] M. Durand. 2004. *Combinatoire analytique et algorithmique des ensembles de données.* Ph.D. Dissertation. École Polytechnique, Palaiseau, France. https://pastel.hal.science/pastel-00000810

[19] R. A. L. Elworth, Q. Wang, P. K. Kota, C. J. Barberan, B. Coleman, A. Balaji, G. Gupta, R. G. Baraniuk, A. Shrivastava, and T. J. Treangen. 2020. To Petabytes and beyond: recent advances in probabilistic and signal processing algorithms and their application to metagenomics. *Nucleic Acids Research* 48, 10 (2020), 5217–5234. https://doi.org/10.1093/nar/gkaa265

[20] O. Ertl. 2017. New cardinality estimation algorithms for HyperLogLog sketches. (2017). arXiv:1702.01284 [cs.DS]

[21] O. Ertl. 2021. SetSketch: Filling the Gap between MinHash and HyperLogLog (extended version). (2021). arXiv:2101.00314 [cs.DS]

[22] O. Ertl. 2023. UltraLogLog: A Practical and More Space-Efficient Alternative to HyperLogLog for Approximate Distinct Counting (extended version). (2023). arXiv:2308.16862 [cs.DS]

[23] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. 2007. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In *Proceedings of the International Conference on the Analysis of Algorithms (AofA)*. 127–146. https://doi.org/10.46298/dmtcs.3545

[24] P. Flajolet and G. N. Martin. 1985. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences* 31, 2 (1985), 182–209. https://doi.org/10.1016/0022-0000(85)90041-8

[25] M. J. Freitag and T. Neumann. 2019. Every Row Counts: Combining Sketches and Sampling for Accurate Group-By Result Estimates. In *Proceedings of the 9th Conference on Innovative Data Systems Research (CIDR)*.

[26] A. Helmi, J. Lumbroso, C. Martínez, and A. Viola. 2012. Data Streams as Random Permutations: the Distinct Element Problem. In *Proceedings of the 23rd International Meeting on Probabilistic, Combinatorial, and Asymptotic Methods in the Analysis of Algorithms (AofA)*. 323–338. https://doi.org/10.46298/dmtcs.3002

[27] S. Heule, M. Nunkesser, and A. Hall. 2013. HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT)*. 683–692. https://doi.org/10.1145/2452376.2452456

[28] M. Karppa and R. Pagh. 2022. HyperLogLogLog: Cardinality Estimation With One Log More. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*. 753––761. https://doi.org/10.1145/3534678.3539246

[29] K. J. Lang. 2017. Back to the Future: an Even More Nearly Optimal Cardinality Estimation Algorithm. (2017). arXiv:1708.06839 [cs.DS]

[30] J. Lu, H. Chen, J. Zhang, T. Hu, P. Sun, and Z. Zhang. 2023. Virtual self-adaptive bitmap for online cardinality estimation. *Information Systems* 114, 102160 (2023). https://doi.org/10.1016/j.is.2022.102160

[31] G. Marçais, B. Solomon, R. Patro, and C. Kingsford. 2019. Sketching and Sublinear Data Structures in Genomics. *Annual Review of Biomedical Data Science* 2, 1 (2019), 93–118. https://doi.org/10.1146/annurev-biodatasci-072018-021156

[32] A. Nazi, B. Ding, V. Narasayya, and S. Chaudhuri. 2018. Efficient Estimation of Inclusion Coefficient Using Hyperloglog Sketches. In *Proceedings of the 44th International Conference on Very Large Data Bases (VLDB)*. 1097–1109.

[33] T. Ohayon. 2021. ExtendedHyperLogLog: Analysis of a new Cardinality Estimator. (2021). arXiv:2106.06525 [cs.DS]

[34] C. Pavlopoulou, M. J. Carey, and V. J. Tsotras. 2022. Revisiting Runtime Dynamic Optimization for Join Queries in Big Data Management Systems. In *Proceedings of the 25th International Conference on Extending Database Technology (EDBT)*. https://doi.org/10.5441/002/edbt.2022.01

[35] O. Peters. [n.d.]. *PolymurHash.* Retrieved March 17, 2024 from https://github.com/orlp/polymur-hash

[36] S. Pettie and D. Wang. 2021. Information Theoretic Limits of Cardinality Estimation: Fisher Meets Shannon. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing (STOC)*. 556–569. https://doi.org/10.1145/3406325.3451032

[37] S. Pettie, D. Wang, and L. Yin. 2021. Non-Mergeable Sketching for Cardinality Estimation. In *48th International Colloquium on Automata, Languages, and Programming (ICALP)*, Vol. 198. 104:1–104:20. https://doi.org/10.4230/LIPIcs.ICALP.2021.104

[38] B. W. Priest, R. Pearce, and G. Sanders. 2018. Estimating Edge-Local Triangle Count Heavy Hitters in Edge-Linear Time and Almost-Vertex-Linear Space. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC)*. https://doi.org/10.1109/HPEC.2018.8547721

[39] J. Qin, D. Kim, and Y. Tung. 2016. LogLog-Beta and More: A New Algorithm for Cardinality Estimation Based on LogLog Counting. (2016). arXiv:1612.02284 [cs.DS]

[40] B. Scheuermann and M. Mauve. 2007. Near-optimal compression of probabilistic counting sketches for networking applications. In *Proceedings of the 4th ACM International Workshop on Foundations of Mobile Computing (FOMC)*.

[41] R. Sedgewick. 2022. HyperBit: A Memory-Efficient Alternative to HyperLogLog. (2022). https://www.birs.ca/workshops/2022/22w5004/files/BobSedgewick/HyperBit.pdf Analytic and Probabilistic Combinatorics Workshop at the Banff International Research Station (BIRS) for Mathematical Innovation and Discovery.

[42] D. Ting. 2014. Streamed Approximate Counting of Distinct Elements: Beating Optimal Batch Methods. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 442–451. https://doi.org/10.1145/2623330.2623669

[43] D. Ting. 2019. Approximate distinct counts for billions of datasets. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 69–86. https://doi.org/10.1145/3299869.3319897

[44] R. Urban. [n.d.]. *SMhasher: Hash function quality and speed tests.* Retrieved March 17, 2024 from https://github.com/rurban/smhasher

[45] A. Vaneev. [n.d.]. *Komihash.* Retrieved March 17, 2024 from https://github.com/avaneev/komihash

[46] D. Wang and S. Pettie. 2023. Better Cardinality Estimators for HyperLogLog, PCSA, and Beyond. In *Proceedings of the 42nd ACM Symposium on Principles of Database Systems (PODS)*. 317—-327. https://doi.org/10.1145/3584372.3588680

[47] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. 1990. A Linear-Time Probabilistic Counting Algorithm for Database Applications. *ACM Transactions on Database Systems* 15, 2 (1990), 208—-229. https://doi.org/10.1145/78922.78925

[48] J. Wires, S. Ingram, Z. Drudi, N. J. A. Harvey, and A. Warfield. 2014. Characterizing storage workloads with counter stacks. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 335–349. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/wires

[49] Q. Xiao, S. Chen, Y. Zhou, and J. Luo. 2020. Estimating Cardinality for Arbitrarily Large Data Stream With Improved Memory Efficiency. *IEEE/ACM Transactions on Networking* 28, 2 (2020), 433–446. https://doi.org/10.1109/TNET.2020.2970860

[50] Q. Xiao, Y. Zhou, and S. Chen. 2017. Better with fewer bits: Improving the performance of cardinality estimation of large data streams. In *IEEE Conference on Computer Communications (IEEE INFOCOM)*. 1–9. https://doi.org/10.1109/INFOCOM.2017.8057088

[51] Y. Zhao, S. Guo, and Y. Yang. 2016. Hermes: An Optimization of HyperLogLog Counting in real-time data processing. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*. 1890–1895. https://doi.org/10.1109/IJCNN.2016.7727430

[52] W. Yi. [n.d.]. *Wyhash*. Retrieved March 17, 2024 from https://github.com/wangyi-fudan/wyhash