Frequency-Quantized HyperLogLog Autocorrection

Author: Shun (@shun4midx)

1 Introduction

1.1 What is HyperLogLog?

"HyperLogLog", also stylized as HYPERLOGLOG and abbreviated as HLL, is a very space-efficient probabilistic data sketch originally designed for estimating cardinality (number of distinct elements) of a multiset with sublinear memory, introduced in the paper [Fla+07]. Instead of storing all observed elements, HLL maintains an array of small registers, each storing the index of the leftmost `1`-bit in the hashed value of elements assigned to that register. From the distribution of these values, it estimates with high probability the cardinality with minimal memory overhead.

HyperLogLog Properties

The key properties of HLL that are important in most algorithms incorporating it are:

- **Memory efficiency:** The structure requires only $O(m)$ space for m registers, typically a few kilobytes even for large datasets.
- **Probabilistic accuracy:** The relative error is approximately $\frac{1.04}{\sqrt{m}}$ [Fla+07], which can be tuned by adjusting the number of registers.
- **Mergeability:** Two HLL sketches built on different datasets can be combined in $O(m)$ time to produce a sketch for the union of both datasets.

These properties make HLL a strong candidate for large-scale approximate matching problems, which is related to autocorrection. In this algorithm, memory efficiency and probabilistic accuracy are the most important parts of HLL. Mergeability is a theoretical property of HLL that we do not currently use, but will be useful for future development.

1.2 Inspiration for HyperLogLog Autocorrection

With increasing need for efficient and scalable data processing in autocorrection, traditional linear memory counting models are often impractical due to their computational overhead. Hence, approximate counting techniques must overcome these constraints. One of the most promising approximate counting techniques useful to autocorrection is fuzzy counting, which allows for approximate matching of data items.

Definition for Fuzzy Counting

Formally, let $\mathcal{D} = \{w_1, w_2, \ldots, w_n\}$ be a multiset of strings from a finite alphabet Σ, and let $Q \in \Sigma^*$ be a query string. A fuzzy count function $f(Q)$ returns the number of elements $w_i \in \mathcal{D}$ such that $\boxed{\text{sim}(Q, w_i) \geq \theta}$, where $\text{sim} : \Sigma^* \times \Sigma^* \to [0, 1]$ is a similarity function such as normalized edit distance or Jaccard similarity, and $\theta \in (0, 1]$ is a predefined similarity threshold.

Of course this accommodates typos, but fuzzy counting with limited memory and low latency while maintaining accuracy, is a significant challenge, unless we use the efficient probabilistic data structure HLL.

1.3 From Cardinality to Autocorrection

However, traditionally, HLL is not used for such natural language processing problems, and is only a cardinality estimator. There does not seem to be an evident link between HLL and autocorrection on the surface.

Hence, this work thinks of autocorrection in a different light. "Autocorrection" alone as a term is quite vague, but an angle to tackle it is viewing it as a problem of finding the optimal $w_i \in \mathcal{D}$, such that it has most fuzzy matching q-grams (substrings of length q) with the query $Q \in \Sigma^*$. Here, we define the fuzzy matching function $f(Q)$ to check for only Levenshtein edit distance of at most 1, **allowing replacing or swapping, but no deletion** — a choice that mirrors how dyslexic people process new words as an overall impression rather than letter by letter in order. In other words, it's similar to a dyslexic person scanning Q and determining if it visually "roughly" matches another word.

While conventional autocorrection algorithms may incidentally accommodate certain transpositions or shape-based similarities (such as Damerau-Levenshtein, q-gram overlap), they typically do not prioritize this matching pattern. Our approach deliberately models it as the primary similarity measure, motivated by the author's own reading patterns with dyslexia.

Key Insight

If we do so and we treat each fuzzy q-gram as a separate "signal", HLL can estimate which "signals" are used by the most $w_i \in \mathcal{D}$. With its low memory and time efficiency with our reinterpretation of what autocorrection is, HLL seems to be able to be a great candidate for an autocorrection algorithm.

2 Weighting: Frequency-Quantization and Tie-Breakers

Of course, alone as HLL, it does not immediately accurately approximate language, let alone with natural language processing tasks such as autocorrection. To have a more accurate representation of language, we would need to account for word frequencies, for example, the typo "applw" is very likely "apple" rather than "appl" even if both are words. However, how would we do so when we do not know exactly the frequency of each word, like in real world scenarios? Especially due to memory and time concerns, we cannot simply rely on repeated inserts (with slight variations each time, because HLL is a distinct object counter) into a sketch, how else can we approximate frequency?

2.1 Zipfian Distribution

Usually, even without knowing the exact frequencies of words, users can roughly arrange the relative frequencies of words, for example, it is clear "the" is more common than the word "algorithm". Zipf's law suggests the frequency usage of words in any language would naturally follow a Zipfian distribution, which is when a list of words is ranked in order from most to least frequent, **word position is inversely proportional to frequency**.

Zipf's Law

Formally, if the most frequent word has an estimated frequency f_1, then the n-th ranked word would have frequency approximately $\frac{f_1}{n}$.

Zipf's Law Application in our Algorithm

In practice, this means we can simply allow a user to input a list of words from most to least frequent, and estimate the relative frequencies of each word. To avoid making frequency adjustments too unbelievable, we divide the list of words into different buckets, and use Zipf's law according to bucket rank rather than word rank, so each word in the same bucket has the same estimated frequency.

This property is especially useful for FQ-HLL autocorrection, since it allows frequency quantization without having to explicitly store every individual count. With the rank-based frequency estimate, the algorithm can assign weights to candidate words in a way that closely mimics natural language usage, while still keeping the memory footprint extremely small. In our implementation, we set the number of buckets to be approximately the square root of the total number of words. This heuristic balances the granularity of frequency ranking (more buckets give finer distinction) with the extra memory required for storing more sketches.

2.2 Frequency-Quantized HyperLogLog

Traditionally, HLL stores, for each register, the maximum ρ value observed, where $\rho(w)$ follows a geometric distribution: the probability that the first `1` occurs at position k is $p = 2^{-k}$. Interpreting this as repeated Bernoulli trials with "success" = "bit is 1" gives $\rho(w) \approx \lfloor \log_2 p \rfloor$. In cardinality estimation, these large ρ values are rare and therefore signal the presence of many distinct elements.

Frequency-Quantized HLL (FQ-HLL) uses this to its advantage by introducing a small shift to the stored ρ value based on the frequency of the q-gram. Since ρ already acts like a $\log_2$-scaled frequency indicator, we can bias the register upward for items that are common via shifting ρ when inserting values.

FQ-HLL Key Idea

Formally, instead of storing

$$R_j \leftarrow \max(R_j,\, \rho(w)),$$

we store

$$R_j \leftarrow \max(R_j,\, \rho(w) + s(w)),$$

where $s(w) \in \mathbb{Z}$ is a small shift based on the q-gram's estimated frequency rank. More common q-grams receive larger $s(w)$ values, so their ρ is boosted more, giving them greater influence during autocorrection scoring. The specific way $s(w)$ is calculated can vary by implementation but it is always chosen to be small (bounded by the register's maximum value) so the extra memory cost is negligible.

Since the implied weight of a q-gram in HLL is proportional to $2^{\rho(w)}$, adding a constant shift $s(w)$ gives: $2^{\rho(w)+s(w)} = 2^{\rho(w)} \cdot 2^{s(w)}$. Thus, the shift simply multiplies the implied weight by the fixed factor $2^{s(w)}$, making it both interpretable and computationally cheap. This preserves HLL's sublinear memory and $O(1)$ update time while biasing the sketch to give **frequent** q-grams proportionally more weight during candidate ranking.

2.3 Jaccard Index and Bitvector Encoding

However, most importantly, we require a standardized way to determine the level of "fuzzy matching" between a query Q and a word w_i. Thus, on top of using the fuzzy similarity introduced earlier, we could use the Jaccard similarity index.

Jaccard Index Definition

The Jaccard Index is a measure of set similarity defined as the ratio of the intersection size to the union size of two sets, where higher Jaccard Index corresponds to higher similarity:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}.$$

In this algorithm, A will be the set of q-grams (including fuzzy matches) extracted from the query, and B will be the set of q-grams (including fuzzy matches) for a candidate word.

Bitvector Encoding in this Algorithm

If we rely on **bitvector encoding** on sets of q-grams (encoding bits with `1` if the corresponding q-gram is present and `0` otherwise), we could compute the **intersection** size simply via a **bitwise AND**, and determine its size by counting the number of `1` bits, also known as `popcount`.

Moreover, by the inclusion-exclusion principle, their union size equals $\boxed{|A| + |B| - |A \cap B|}$. This representation enables Jaccard similarity to be computed in $O(1)$ with respect to the number of registers, avoiding expensive string comparisons while still supporting fuzzy matching with low time overhead.

3 Main Algorithms

3.1 Overall Picture

From Section 2, we already have a general idea of how our FQ-HLL algorithm functions. With Zipf's Law and FQ-HLL, we can insert words in a dictionary into different sets of HLL sketches per each q-gram of the dictionary. Then, using Jaccard similarity via bitvector encoding, we can determine the degree of fuzzy matching of each w_i with respect to Q. Finally, we select the highest score candidate, based on Jaccard index and frequency. In fact, at a high level, FQ-HLL autocorrection functions as follows:

FQ-HLL Autocorrection's Overall Picture

1. **Extract normalized q-grams** from the query and each dictionary word, allowing fuzzy matches (edit distance ≤ 1) that align with the dyslexia-oriented similarity model. *This preserves the overall structure of the word even when minor typos are present.*
2. **Insert q-grams into an FQ-HLL sketch**, applying a small bias $s(w)$ based on the word's Zipf-ordered frequency rank relative to bucket rank. *This frequency quantization gives more common words proportionally higher influence without extra memory.*
3. **Encode each q-gram set as a fixed-length bitvector**. *This compact representation enables constant-time set operations using bitwise logic.*
4. **Compute Jaccard similarity** between the query bitvector and each candidate's bitvector. *This measures the proportion of shared q-grams while normalizing for different word lengths.*
5. **Rank and select top candidates**, breaking ties with frequency rank and lexicographical order. *This ensures stable, predictable results while favoring likely corrections.*

In the following section, each of the five steps of this algorithm would be detailed in pseudocode.

3.2 Actual Algorithms

For all intents and purposes, q is defaulted to the value 2 for the following algorithms, since it allows for the most flexibility when autocorrecting, yet also allows the most fuzziness. However, $q > 2$ is still a theoretically possible algorithm, yet would not be something that has been thoroughly tested for practical accuracy. Here, L is the register index length in bits, so 2^L is the number of possible positions of a block. Also, α is a parameter that determines how much the weighting of frequency ranking should factor into final score determination.

Q-gram Extraction and Normalization

function EXTRACTQGRAMS($w, q, fuzzier$)
  if $|w| < q$ **then**
    return $\emptyset$
  $Q \leftarrow [\,]$
  for $i \leftarrow 0$ to $|w| - q$ **do**
    $g \leftarrow w[i : i + q]$
    Q.APPEND(g)
    if $q = 2$ **then**
      Q.APPEND(g) ▷ Duplicate exact bigrams (extra weight)
      Q.APPEND($g[0] +$ " ") ▷ Left padded
      Q.APPEND(" " $+ g[1]$) ▷ Right padded
      if $fuzzier$ **then**
        Q.APPEND($g[1]g[0]$) ▷ Adjacent swap
  return Q

Sketch Construction (FQ-HLL Build)

function BUILDSKETCHES($\mathcal{D}, b$) ▷ $\mathcal{D}$: lowercased dict words
  $q \leftarrow 2$
  Initialize HLL config with precision b
  $W \leftarrow |\mathcal{D}|$
  $NUM\_BUCKETS \leftarrow 2^{\lceil \log_2(\sqrt{W}) \rceil}$ ▷ Set bucket sizes ≈ square root of total word count
  $BUCKET\_SIZE \leftarrow \lceil W / NUM\_BUCKETS \rceil$
  $M \leftarrow$ empty map ($g \mapsto$ HLL sketch)
  for $i \leftarrow 0$ to $W - 1$ **do**
    $w \leftarrow \mathcal{D}[i]$
    $bucket \leftarrow \min(NUM\_BUCKETS,\ \lfloor (i+1)/BUCKET\_SIZE \rfloor + 1)$
    $shift \leftarrow \min(4 \cdot \lfloor \log_2(NUM\_BUCKETS/bucket) \rfloor,\ 2^L)$
    for all $g \in$ EXTRACTQGRAMS($w, q,$ **false**) **do**
      if $g \notin M$ **then**
        $M[g] \leftarrow$ new HLL config with precision b
      $key \leftarrow g +$ "_" $+ w$
      $M[g]$.SHIFTEDINSERT($key, shift$) ▷ Simply insert key with $\rho \leftarrow \rho + shift$
  return $(M,\ NUM\_BUCKETS,\ BUCKET\_SIZE)$

Bitvector Encoding

function BUILDBITVECTORS($\mathcal{D}, R$)
  $U \leftarrow$ sorted list of all q-grams (keys of M) ▷ Can be sorted in any replicable order
  $ID \leftarrow$ map (q-gram $\mapsto$ index in $[0, |U| - 1]$)
  $blocks \leftarrow \lceil |U|/2^L \rceil$; $B \leftarrow [\,]$ ▷ Bitvectors parallel to $\mathcal{D}$
  for all $w \in \mathcal{D}$ **do**
    $bv \leftarrow$ array of $blocks$ zeros
    for all $g \in$ EXTRACTQGRAMS($w, 2,$ **false**) **do**
      if $g \in ID$ **then**
        $bit \leftarrow ID[g]$
        $blk \leftarrow bit >> L$ ▷ Bits right-shift by L
        $off \leftarrow bit \,\&\, (2^L - 1)$ ▷ Bitwise AND
        $bv[blk] \leftarrow bv[blk] \mid (1 << off)$ ▷ Bits left-shift by off, bitwise OR
    append bv to B
  return (U, ID, B)

Scoring & Candidate Selection (Top-1)

function SUGGEST($query, \mathcal{D}, (U, ID, B), BUCKET\_SIZE, \alpha$)
  $Q \leftarrow$ set(EXTRACTQGRAMS($query, 2,$ **true**))
  $qb \leftarrow$ zero bitvector of length $|U|$
  for all $g \in Q$ **do**
    if $g \in ID$ **then**
      set bit $ID[g]$ in qb
  $qb\_count \leftarrow \sum_{blk} \text{popcount}(qb[blk])$
  $J \leftarrow$ empty map; $R \leftarrow$ empty map
  for $i \leftarrow 0$ to $|\mathcal{D}| - 1$ **do**
    if $\mathcal{D}[i]$ is removed **then continue**
    $inter \leftarrow \sum_{b} \text{popcount}(B[i][b] \;\&\; qb[b])$ ▷ Bitwise AND
    if $inter = 0$ **then continue**
    $ones \leftarrow \sum_{b} \text{popcount}(B[i][b])$
    $uni \leftarrow qb\_count + ones - inter$
    $J[i] \leftarrow \begin{cases} inter/uni, & uni > 0 \\ 0, & \text{else} \end{cases}$
    $R[i] \leftarrow 1/\big(\lfloor (i+1)/BUCKET\_SIZE \rfloor + 1\big)$ ▷ Bucket-ranked Zipf frequency
  $best \leftarrow (-\infty, -1, -1)$
  for all $\tau \in \{0.8, 0.7, 0.6, 0.5, 0.4\}$ **do** ▷ In practice, this range of checking is optimal
    for all i with $J[i] \geq \tau$ **do**
      $lenPenalty \leftarrow 1 - \big(\frac{||\mathcal{D}[i]| - |query||}{|query|}\big)^2$
      $score \leftarrow (J[i] + \alpha R[i]) \cdot lenPenalty + \mathbf{1}[\mathcal{D}[i] = query]$
      if $score > best.score$ **then**
        $best \leftarrow (score, i, \tau)$
  if $best.i = -1$ **then**
    $best.i \leftarrow \arg\max_i J[i]$; $best.\tau \leftarrow 0.4$
  return $\mathcal{D}[best.i]$

Scoring & Candidate Selection (Top-3)

function TOPTHREE($query, \mathcal{D}, (U, ID, B), BUCKET\_SIZE, \alpha$)
  compute $J[i]$ and $R[i]$ as in SUGGEST
  $base[i] \leftarrow (J[i] + \alpha R[i]) \cdot \left(1 - \left(\frac{||\mathcal{D}[i]| - |query||}{|query|}\right)^2\right)$
  $shortlist \leftarrow$ indices of top 30 by $base[i]$ (desc)
  for all i in $shortlist$ **do**
    $final[i] \leftarrow base[i] + \mathbf{1}[\mathcal{D}[i] = query]$
  sort $final$ desc
  emit first three **distinct** display strings (dedupe via display map), pad with "" if fewer
  return list of up to 3 suggestions

3.3 Remark on Keyboard Layouts

As the purpose of this FQ-HLL algorithm is to demonstrate autocorrection's success via FQ-HLL without the need of English or keyboard knowledge, the algorithms above do not account for the keyboard layout. However, when implementing it as an algorithm used in real life, perhaps this serves as useful information.

In this case, we introduce parameter β, which adjusts scores by adding $\boxed{\frac{\beta}{1 + \text{WORDDIST}(query, w_i)}}$, where WORDDIST is computed based on Levenshtein distance via DP as follows:

Edit Distance Function

function WORDDIST(a, b)
  $m \leftarrow |a|; n \leftarrow |b|$
  Create matrix $dp[0 \ldots m][0 \ldots n]$
  for $i \leftarrow 0$ to m **do**
    $dp[i][0] \leftarrow i$
  for $j \leftarrow 0$ to n **do**
    $dp[0][j] \leftarrow j$
  for $i \leftarrow 1$ to m **do**
    for $j \leftarrow 1$ to n **do**
      if $a[i-1] = b[j-1]$ **then**
        $cost \leftarrow 0$
      else
        $cost \leftarrow$ CHARDIST$(a[i-1], b[j-1])$ ▷ CHARDIST is Manhattan distance
      $dp[i][j] \leftarrow \min\{dp[i-1][j] + 1,\ dp[i][j-1] + 1,\ dp[i-1][j-1] + cost\}$
  return $dp[m][n]$

4 Experimental Results

We evaluate FQ-HLL autocorrection using the `typo_file.txt` dataset derived from Peter Norvig's spelling corrector corpus[1], testing against two dictionary configurations:

- **`database.txt`** — Contains all intended correction words in original order (no frequency information).
- **`20k_shun4midx.txt`** — The 20,000 most common English words[2] placed before the words from `database.txt`, forcing intended corrections to be ranked lower by frequency bias, and thus disadvantaging the results.

Accuracy is defined strictly: a prediction is correct only if it exactly matches the intended target word. For example, for the typo "mant", even if our algorithm suggests "many", if the intended answer was "want", we still count it as wrong. We also measure **Top-3 accuracy**, where a prediction is counted correct if the intended word appears in the first three suggestions. We also defaulted α to 0.2, and b (for HLL) to 10.

Python Implementation On `database.txt`, FQ-HLL achieved **87-88%** Top-1 accuracy and **93-94%** Top-3 accuracy, with total runtime $0.223\,\text{s}$. On `20k_shun4midx.txt`, accuracy dropped to **59-60%** (Top-1) and **75-76%** (Top-3) with runtime $9.955\,\text{s}$.

For comparison, a BK-tree with Levenshtein edit distance ≤ 2 scored **75-76%** (Top-1) and **43-44%** (Top-3) but was over $5\times$ slower and had a lower accuracy. Even at edit distance ≤ 3, BK-tree accuracy rose modestly to **89-90%** (Top-1) but remained slower than FQ-HLL. SymSpell (edit distance ≤ 5) achieved similar accuracy to BK-tree but required separate build and query phases. The SymSpell times are displayed assign Build + Query time.

Method	`database.txt`	`20k_shun4midx.txt`
FQ-HLL	0.223 s	9.955 s
BK-tree (ED ≤ 2)	2.126 s	46.028 s
BK-tree (ED ≤ 3)	3.816 s	92.812 s
SymSpell	0.515 s + 0.996 s	16.994 s + 29.355 s

C++ Implementation C++ produced identical accuracy to Python but with significantly faster runtimes: $0.071\,\text{s}$ (`database.txt`) and $2.649\,\text{s}$ (`20k_shun4midx.txt`) for Top-1; $0.084\,\text{s}$ and $3.716\,\text{s}$ for Top-3.

Remark on Keyboard-Aware Mode We optionally incorporate keyboard layout distance into the scoring function, with β defaulted to 0.35. On `20k_shun4midx.txt`, this increases accuracy to **64-65%** (Top-1) and **80-81%** (Top-3) with only $\sim$1 s additional runtime. However, the core results above are layout-agnostic, demonstrating FQ-HLL's strength without keyboard-specific information.

5 Workarounds — Amortized $O(1)$ Addition and Deletion

Although it is technically possible to create a new sketch every time we add or remove an element in the dictionary, this is not so convenient, and we require a low time and memory overhead method

[1] https://www.kaggle.com/datasets/bittlingmayer/spelling/data
[2] https://github.com/first20hours/google-10000-english/blob/master/20k.txt

to do so to maintain the advantage of FQ-HLL. Upon careful inspection of the algorithms, we realize that the addition of words can be simply done by adding onto certain maps, if the register exponent $\lfloor \log_2(NUM\_BUCKETS) \rfloor$ remains unchanged. In the case where it differs, then we can rebuild everything. Similarly, this logic can be applied to deletion. We may not be able to fully delete from a sketch due to limitations by HLL, but by blocking words to be unable to appear as suggestions, it functionally is almost the same as deletion. If we rebuild often enough, it would not be too big of a difference in sketch values.

Thus, we can use the following algorithms to maintain addition and deletion without needing costly rebuilds of sketches each time. In fact, they are amortized $O(1)$, because addition would occur $o(n)$ times, yet rebuild takes maximum $O(n)$, and otherwise, the operations are done in $O(1)$. For subtraction, if we set a certain threshold $t \in [0, 1)$, and rebuild after $t\times$ the original size of words are removed (by keeping track of the words removed), that means we compute an $O(n)$ operation for every $t \cdot n$ words removed.

Addition to Dictionary

function ADDDICTIONARY($\mathcal{D}, W_{\text{new}}, b,\ M, U, ID, B,\ NUM\_BUCKETS,\ BUCKET\_SIZE$)
  $A \leftarrow$ words in W_{new} not already in $\mathcal{D}$ (after validation/lowercasing)
  if $|A| = 0$ **then**
    return *unchanged*
  $old\_exp \leftarrow \log_2(NUM\_BUCKETS)$
  $new\_exp \leftarrow \left\lceil \dfrac{\log_2(|\mathcal{D}| + |A|)}{2} \right\rceil$
  if $new\_exp \neq old\_exp$ **then**
    append A to $\mathcal{D}$ (and update display/sets)
    return REBUILDDICTIONARY($\mathcal{D}, \varnothing, b$)
  $NUM\_BUCKETS \leftarrow \left\lceil \dfrac{|\mathcal{D}| + |A|}{BUCKET\_SIZE} \right\rceil$ ▷ keep $BUCKET\_SIZE$ fixed
  $base \leftarrow |\mathcal{D}|$
  for $i \leftarrow 0$ to $|A| - 1$ **do**
    $w \leftarrow A[i]$; append w to $\mathcal{D}$ (and update display/sets)
    $bucket \leftarrow \min\Big(NUM\_BUCKETS,\ \left\lfloor \dfrac{base + i + 1}{BUCKET\_SIZE} \right\rfloor + 1\Big)$
    $shift \leftarrow \min\Big(4 \cdot \lfloor \log_2(\frac{NUM\_BUCKETS}{bucket}) \rfloor,\ \text{SHIFTCAP}\Big)$
    for all $g \in$ EXTRACTQGRAMS($w, 2,$ **false**) **do**
      if $g \notin M$ **then**
        $M[g] \leftarrow$ new HLL(b)
      $key \leftarrow (g + \text{“\_”} + w)$
      $M[g]$.SHIFTEDINSERT($key,\ shift$)
  $newU \leftarrow$ sorted keys of M
  if $|newU| > |U|$ **then** ▷ new q-grams appeared
    extend every bitvector in B with zero blocks to length $|newU|$
    $U \leftarrow newU$; recompute ID (q-gram $\mapsto$ index)
  for each newly added w **do**
    build bitvector by setting $ID[g]$ for $g \in$ EXTRACTQGRAMS($w, 2,$ **false**); append to B
  return *updated structures*

Deletion from Dictionary

function REMOVEDICTIONARY($\mathcal{D}, W_{\text{del}}, \mathcal{R}, t,\ b,\ M,\ U,\ ID,\ B$)
  for all $w \in W_{\text{del}}$ **do**
    if $w \in \mathcal{D}$ **then**
      add w to $\mathcal{R}$ ▷ Logical delete
  if $|\mathcal{R}| \geq t \cdot |\mathcal{D}|$ **then**
    return REBUILDDICTIONARY($\mathcal{D}, \mathcal{R}, b$)
  else
    return (*updated* $\mathcal{R}$) ▷ Other structures unchanged

Rebuild Dictionary (Shared with both Addition and Deletion)

function REBUILDDICTIONARY($\mathcal{D}, \mathcal{R}, b$)
  $\mathcal{D} \leftarrow \mathcal{D} \setminus \mathcal{R}$ ▷ physically remove blocked words
  reset all sketches/maps
  $(M,\ NUM\_BUCKETS,\ BUCKET\_SIZE) \leftarrow$ BUILDSKETCHES($\mathcal{D}, b$)
  $(U,\ ID,\ B) \leftarrow$ BUILDBITVECTORS($\mathcal{D}, M$) ▷ q-gram universe, index map, bitvectors
  $\mathcal{R} \leftarrow \varnothing$
  return $(\mathcal{D}, \mathcal{R}, M, U, ID, B,\ NUM\_BUCKETS,\ BUCKET\_SIZE)$

6 Future Developments with Local Differential Privacy

A huge part of the inspiration for this work was based on the paper [VBK22], which detailed an algorithm that contained fuzziness, efficiency, and (local differential) privacy. This was because the mainstream methods of autocorrection such as Apple's CMS (Count-Min Sketch) only had efficiency and fuzziness, whereas Google's RAPPOR (Randomized Aggregatable Privacy-Preserving Ordinal Response) only had efficiency and privacy, so they wanted to create an algorithm that contained all three.

This work's ultimate goal for FQ-HLL autocorrection is to also contain all three, which was what raised the interest with how to incorporate Local Differential Privacy (LDP) in this algorithm.

6.1 Definition

Definition of DP

Introduced in the paper [Dwo06], differential privacy ensures that the inclusion or exclusion of a single individual's data does not significantly affect the output of a randomized algorithm.

Formally, for any two neighboring datasets D and D' differing in at most one record with $D \subseteq D'$, and for any subset of outputs $S \subseteq \text{Range}(\mathcal{A})$, a randomized algorithm $\mathcal{A}$ satisfies ε-differential privacy if:

$$\Pr[\mathcal{A}(D) \in S] \leq e^{\varepsilon} \cdot \Pr[\mathcal{A}(D') \in S]$$

where $\varepsilon > 0$ is the privacy budget. A smaller ε implies stronger privacy, as it limits the algorithm's sensitivity to individual records

Definition of LDP

In decentralized settings, LDP provides per-user privacy by requiring the randomization to be applied before data leaves the user's device. A mechanism $\mathcal{A} : \mathcal{X} \to \mathcal{Y}$ satisfies ε-LDP if for all $x, x' \in \mathcal{X}$ and $y \in \mathcal{Y}$,

$$\Pr[\mathcal{A}(x) = y] \leq e^{\varepsilon} \cdot \Pr[\mathcal{A}(x') = y]$$

6.2 LDP within FQ-HLL Autocorrection — Temporary Progress

Despite promising results from amortized constant-time addition and deletion, in practice the suggested results of typos seem to be at a slightly lower accuracy, albeit still acceptable. This is likely due to an implementation bug rather than a fundamental issue with the algorithm. As such, even with two plausible ideas for LDP in FQ-HLL, I have not yet pushed a working implementation. In this section, I detail the intuition and proof for the idea of replacing certain reported q-grams with random q-grams to achieve LDP.

Mechanism (Unary Randomized Response on q-gram presence)

Formally, fix an ordered q-gram universe $U = \{g_1, \ldots, g_m\}$. Given a client set $S \subseteq U$, encode $v \in \{0,1\}^m$ with $v_j = \mathbf{1}[g_j \in S]$. Independently for each coordinate j, output $\tilde{v}_j \in \{0,1\}$ as

$$\Pr[\tilde{v}_j = 1 \mid v_j = 1] = p, \quad \Pr[\tilde{v}_j = 1 \mid v_j = 0] = q, \quad \Pr[\tilde{v}_j = 0 \mid v_j = \cdot] = 1 - \Pr[\tilde{v}_j = 1 \mid v_j = \cdot],$$

with the standard ε-LDP choice

$$p = \frac{e^{\varepsilon}}{e^{\varepsilon} + 1}, \qquad q = \frac{1}{e^{\varepsilon} + 1}.$$

(Optionally cap $|S| \leq K$ or sample a subset of coordinates to control composition.)

Theorem (Per-bit ε-LDP)

For each coordinate j, the above mechanism satisfies $(\varepsilon, 0)$-local differential privacy.

Proof. Fix j and any two inputs $v_j, v_j' \in \{0,1\}$ and any output $y \in \{0,1\}$. We must show $\frac{\Pr[\tilde{v}_j = y \mid v_j]}{\Pr[\tilde{v}_j = y \mid v_j']} \leq e^{\varepsilon}$.
If $y = 1$:

$$\frac{\Pr[\tilde{v}_j = 1 \mid v_j = 1]}{\Pr[\tilde{v}_j = 1 \mid v_j = 0]} = \frac{p}{q} = \frac{\frac{e^{\varepsilon}}{e^{\varepsilon}+1}}{\frac{1}{e^{\varepsilon}+1}} = e^{\varepsilon}.$$

If $y = 0$:

$$\frac{\Pr[\tilde{v}_j = 0 \mid v_j = 1]}{\Pr[\tilde{v}_j = 0 \mid v_j = 0]} = \frac{1-p}{1-q} = \frac{\frac{1}{e^{\varepsilon}+1}}{\frac{e^{\varepsilon}}{e^{\varepsilon}+1}} = e^{-\varepsilon} \leq e^{\varepsilon}.$$

Thus, the likelihood ratio is at most e^{ε} in all cases, proving $(\varepsilon, 0)$-LDP. □

Lemma (Unbiased frequency estimation)

Let N be the number of reports and, for a fixed j, let

$$\tilde{c}_j = \sum_{i=1}^{N} \tilde{v}_j^{(i)}$$

be the *observed privatized count* of 1's after applying the randomized response mechanism. Let

$$c_j = \sum_{i=1}^{N} v_j^{(i)}$$

be the (unobserved) true count, and $f(g_j) = \frac{c_j}{N}$ its true frequency. Then the unbiased estimators are

$$\widehat{c}_j = \frac{\tilde{c}_j - Nq}{p - q}, \qquad \widehat{f}(g_j) = \frac{\widehat{c}_j}{N} = \frac{\frac{\tilde{c}_j}{N} - q}{p - q}.$$

Proof. Let $\theta_j = \Pr[v_j = 1]$. For a single report,

$$\mathbb{E}[\tilde{v}_j] = p \cdot \theta_j + q \cdot (1 - \theta_j) = q + (p - q)\theta_j.$$

Summing over N independent reports gives $\mathbb{E}[\tilde{c}_j] = N\big(q + (p - q)\theta_j\big)$. Solving for θ_j yields the stated estimator with $\mathbb{E}[\widehat{f}(g_j)] = \theta_j = f(g_j)$. □

Variance

Let $\theta_j = f(g_j)$. For one report,

$$\mathrm{Var}(\tilde{v}_j) = \underbrace{\theta_j\, p(1-p) + (1-\theta_j)\, q(1-q)}_{\mathbb{E}[\mathrm{Var}(\tilde{v}_j|v_j)]} + \underbrace{\theta_j(1-\theta_j)(p-q)^2}_{\mathrm{Var}(\mathbb{E}[\tilde{v}_j|v_j])}.$$

Hence

$$\mathrm{Var}\left(\widehat{f}(g_j)\right) = \frac{1}{N} \cdot \frac{\mathrm{Var}(\tilde{v}_j)}{(p-q)^2}.$$

As N grows, variance shrinks at rate $\frac{1}{N}$; for fixed ε, larger $(p - q)$ improves accuracy.

Composition and post-processing

Reporting multiple coordinates composes additively in ε under independence; in practice we cap $|S|$ or subsample coordinates per report to bound total privacy loss. Any server-side computation (debiasing, bucketization, updating $s(\cdot)$) is a post-processing of the privatized data and cannot weaken privacy.

7 Future Scalability

With its low memory footprint and $O(1)$ update/query time, FQ-HLL autocorrection is well suited for many real-life scenarios, especially in both embedded and distributed systems.

In a distributed setting, FQ-HLL can directly exploit the **mergeability** property of HLL: each device can maintain its own set of per q-gram sketches (with frequency shifts applied as usual), and these can be

merged in $O(m)$ time, where m is the number of registers. This allows work to be split across devices and recombined without loss of accuracy.

Such merging is also valuable for cloud-backed applications: a user's device can periodically upload its local sketches, which the server can merge into a global model. This avoids the need to rebuild or reinsert the entire dictionary, enabling incremental, bandwidth-efficient updates. Most importantly, it still has natural obfuscation from HLL only counting q-grams, and could even maintain LDP if algorithms from Section 6 are implemented, which is a great way to maintain privacy.

8 Conclusion and Takeaways

In conclusion, HyperLogLog is a space-efficient, fast, and easily mergeable data structure perfect for cardinality estimations. In this work, we investigated how HLL can be a promising direction towards seeing autocorrection in a new light, allowing for a significant runtime decrease with acceptable accuracy.

We investigated how frequency quantization can be added natively within HLL, which is originally thought to be only a cardinality estimator, to create an equally memory and time efficient FQ-HLL. Notably, by viewing "autocorrection" in a new light similar to how dyslexic people process words, we were able to utilize HLL for language processing. This suggests a new possible usage of data sketching to handle NLP-adjacent problems.

Furthermore, future work suggests a promising possibility that LDP can be done on HLL natively. Perhaps, in today's world where privacy is increasingly-valued, FQ-HLL autocorrection could be used in place of current autocorrection algorithms. Such an approach may eliminate the need to store large amounts of data in a central system, yet maintain privacy, suggesting a very promising privacy-preserving autocorrection suitable in distributed and embedded systems with high speed.

9 References

[Dwo06] Cynthia Dwork. "Differential Privacy". In: *Proceedings of the 33rd International Colloquium on Automata, Languages and Programming (ICALP)*. Vol. 4052. Lecture Notes in Computer Science. Springer, 2006, pp. 1–12. DOI: 10.1007/11787006_1.

[Fla+07] Philippe Flajolet et al. "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm". In: *DMTCS Proceedings*. Ed. by Philippe Jacquet. Vol. DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07). DMTCS Proceedings. Juan les Pins, France: Discrete Mathematics and Theoretical Computer Science, June 2007, pp. 137–156. DOI: 10.46298/dmtcs.3545. URL: https://inria.hal.science/hal-00406166.

[VBK22] Dinusha Vatsalan, Raghav Bhaskar, and Mohamed Ali Kaafar. "Local Differentially Private Fuzzy Counting in Stream Data Using Probabilistic Data Structures". In: *IEEE Transactions on Knowledge and Data Engineering* 35.8 (2022), pp. 8185–8198. DOI: 10.1109/TKDE.2022.3198478.

Acknowledgements

Some definitions in Section 1, 2, and 6 of this algorithm description are adapted from the final project for Advanced Data Structures (2025), which I co-authored with my group members. I would like to thank my then team member for their contribution to those definitions.