Draft version June 9, 2025
Typeset using LaTeX **modern** style in AASTeX7.0.1

Implementing HyperLogLog on Fuzzy Counting

吳述宇 (B12902036),[1] 吳竣凱 (B12201033),[1] 張耕綸 (B10902120),[1] and
謝博仲 (B10505001)[1]

[1] *Advanced Data Structures Final Project*

ABSTRACT

This work explores the design and evaluation of probabilistic data structures for fuzzy counting in streaming environments under memory and privacy constraints. We present a three-phase baseline approach combining Bloom filters and Cuckoo filters to support approximate string matching. By encoding input strings into q-grams and inserting them into randomized, user-side filters, our method enables efficient aggregation and decoding with tunable similarity thresholds. Furthermore, we extend our system with a novel adaptation of the HyperLogLog (HLL) sketch to perform autocorrection via q-gram frequency estimation. Our work demonstrates that combining fuzziness with efficiency is feasible through careful probabilistic design and provides a foundation for future LDP-compatible autocorrection frameworks.

1. INTRODUCTION

As the volume of data generated in real-time continues to expand rapidly, effectively handling streaming data has become an increasingly critical challenge in various applications such as network monitoring, natural language processing, and online services. Traditional exact counting methods are often impractical due to high memory usage and computational costs. Consequently, approximate counting techniques have emerged as essential tools to manage these constraints, enabling efficient and scalable data processing.

One of the most promising approximate counting techniques is fuzzy counting, which allows approximate matching between data items, thereby accommodating typos, misspellings, or semantic variations. However, achieving fuzzy counting under constraints of limited memory and low latency, while maintaining accuracy and privacy, remains a significant challenge.

In this work, we reproduce a robust baseline solution utilizing probabilistic data structures specifically, Bloom filters and Cuckoo filters for fuzzy counting in streaming settings. The solution encodes input strings into segments (q-grams), enabling rapid fuzzy matching.

Furthermore, we explore an innovative extension leveraging HyperLogLog (HLL), a highly space-efficient data sketching technique initially designed for distinct counting.

By rethinking fuzzy counting from a frequency estimation perspective, we introduce adaptations that allow HLL to be employed effectively for fuzzy autocorrection tasks, significantly improving computational speed and memory efficiency.

We conduct comprehensive experimental analyses to rigorously evaluate our proposed methodologies. These experiments involve varying parameters and metrics demonstrating the relative strengths and trade-offs of our baseline and HLL-based approaches. Additionally, we present a user-friendly autocorrection library built upon these techniques to facilitate easy integration into practical systems.

Finally, we outline potential directions for future research, including refining our HLL adaptation for improved fuzzy counting, enhancing privacy-preserving mechanisms, and integrating advanced hybrid structures to optimize overall performance. Our work aims to lay a solid foundation for combining fuzzy matching, efficiency, and privacy, thus addressing critical challenges in modern data stream processing.

2. RELATED WORK

In this section, we review existing privacy-preserving and approximate counting techniques, focusing on methods relevant to real-time typo correction, fuzzy string matching, and stream processing. We begin with two cornerstone LDP systems and then move on to recent fuzzy-counting approaches.

Probabilistic data structures have become popular for privacy-preserving analytics thanks to their compactness and speed. Úlfar Erlingsson et al. (2014) introduced RAPPOR, which encodes each client's string into a Bloom filter and applies randomized response bit-flipping for ϵ-LDP. RAPPOR supports exact-value frequency estimation but does not handle approximate or fuzzy queries. G. Cormode & S. Muthukrishnan (2004) proposed the Count-Min Sketch (CMS), which achieves linear-time updates and offers deterministic additive error bounds, yet has no built-in privacy and likewise only supports exact matching.

More recently, D. Vatsalan et al. (2022) presented a hybrid, LDP-compatible fuzzy counting system. They segment each input into q-grams, encode these into Bloom filters, and insert the segments into an adaptive Cuckoo filter. Their two-stage randomized response ensures ϵ-LDP while preserving semantic similarity, achieving low false-positive rates even as the stream grows. Experiments show substantial gains in both exact and fuzzy count accuracy versus RAPPOR and CMS. However, their approach incurs high communication overhead and complex client-side preprocessing, which may limit deployment in lightweight auto-correction scenarios. Thus, there remains a need for a novel approach that simultaneously supports fuzzy matching, high efficiency, and strong privacy guarantees.

3. PRELIMINARIES

3.1. *Differential Privacy*

Differential Privacy (DP), which was introduced in C. Dwork (2006), is a mathematical framework for quantifying privacy guarantees in data analysis. It 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, 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. Differential privacy offers several desirable resilience to auxiliary information. The privacy guarantee holds regardless of the adversary's prior knowledge. Also, if multiple DP mechanisms are applied, their total privacy loss adds up. For instance, composing two mechanisms with ε_1- and ε_2-DP results in $(\varepsilon_1 + \varepsilon_2)$-DP. Last but not least, any data-independent function applied to the output of a DP mechanism does not weaken its privacy guarantee.

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

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

In this work, we focus on privacy-preserving data structures and mechanisms that operate under the local model.

3.2. *Fuzzy Counting*

Fuzzy counting refers to the task of estimating the number of occurrences of values that are approximately similar to a given query, rather than exactly equal. This concept is particularly useful in scenarios involving typographical errors, noisy inputs, or semantically similar strings (e.g., in search engines, spell checkers, or anonymized databases).

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 $f_q(\cdot)$ returns the number of elements $w_i \in \mathcal{D}$ such that

$$\text{sim}(q, w_i) \geq \theta \ ,$$

where $\text{sim}\colon \Sigma^* \times \Sigma^* \to [0, 1]$ is a similarity function such as normalized edit distance, Jaccard similarity, or cosine similarity, and $\theta \in (0, 1]$ is a predefined similarity threshold.

Unlike exact counting, fuzzy counting requires efficient approximation techniques to handle the high cost of comparing each query against a potentially large corpus of candidate strings. To reduce computational and memory overhead, probabilistic data structures such as Bloom filters, Count-Min Sketches, and Cuckoo filters are often employed. When privacy is also a concern, the challenge increases: mechanisms must both preserve differential privacy and support fuzzy matching, typically via randomized perturbation of the similarity encoding.

In our first work, we aim to support fuzzy counting compact, segment-based data encodings to reproduce the setting in D. Vatsalan et al. (2022).

3.3. *Bloom Filter*

A Bloom filter is a compact, probabilistic data structure designed to efficiently test whether an element is a member of a set, allowing false positives but no false negatives. It supports fast insert and query operations using bitwise manipulation, making it suitable for memory-constrained or streaming environments.

A Bloom filter consists of a bit array $B \in \{0,1\}^m$ of length m, initially all set to 0, and k independent hash functions $h_1, h_2, \ldots, h_k$:$\mathcal{U} \to [m]$, where $\mathcal{U}$ is the universe of possible elements.

Insertion—To insert an element $x \in \mathcal{U}$, the filter sets the bits at positions $h_1(x), h_2(x), \ldots, h_k(x)$ to 1:

$$\forall i \in [k], \quad B[h_i(x)] \leftarrow 1 \ .$$

To check if an element y is in the set, the filter checks whether all k corresponding bits are set:

$$\text{Return “possibly in set” if } \bigwedge_{i=1}^{k} B[h_i(y)] = 1; \quad \text{otherwise “definitely not”.}$$

False Positive Rate—The main trade-off of Bloom filters is the possibility of false positives. After inserting n elements, the probability that a query for a non-inserted element yields a false positive is approximately:

$$\left(1 - e^{-kn/m}\right)^k \ .$$

This can be minimized by choosing $k = \frac{m}{n} \ln 2$, yielding an optimal false positive rate of approximately $(0.6185)^{m/n}$.

Applications—Bloom filters are widely used in distributed systems, networking, databases, and approximate membership testing. In our context, Bloom filters are used to encode q-gram segments of strings, enabling approximate matching and fast pruning of irrelevant candidates in fuzzy queries.

Limitations—While space-efficient, Bloom filters do not support deletion (without extensions such as counting Bloom filters), and collisions may accumulate, leading to degraded precision over time.

3.4. *Cuckoo Filter*

A Cuckoo filter is a space-efficient, probabilistic data structure for approximate set membership that supports insertion, lookup, and deletion with low false positive rate. It builds on the principles of cuckoo hashing by storing compact fingerprints

instead of full keys. Itconsists of an array of m buckets, each bucket holding up to b entries. Each entry stores an f-bit fingerprint of an element. Two hash functions $h_1, h_2 : \mathcal{U} \to [m]$ map an element x to two candidate bucket indices:

$$i_1 = h_1(x), \qquad i_2 = i_1 \oplus (\text{hash\_fp}(x))$$

where hash_fp(x) is a hash of x reduced to f bits, and $\oplus$ denotes bitwise XOR.

Insertion—To insert x:

1. Compute fingerprint $f_x = \text{fingerprint}(x)$.
2. Compute candidate buckets i_1, i_2.
3. If either bucket has an empty slot, place f_x there and return success.
4. Otherwise, randomly pick one of the two buckets, evict an existing fingerprint f_y, insert f_x, and then attempt to insert f_y into its alternate bucket. Repeat up to a maximum of T relocations; if no slot is found, the filter is "full" and insertion fails.

Lookup—To query membership of x, compute f_x, i_1, and i_2 as above. Return "possibly in set" if f_x appears in any slot of bucket i_1 or i_2; otherwise return "definitely not."

Deletion—To delete x, compute its fingerprint and buckets. If f_x is found in either bucket, clear that slot.

Load Factor and Performance—Cuckoo filters achieve high occupancy; typical maximum load factor α is around 95% for $b = 4$. Lookup and deletion are $O(1)$ in the worst case. Insertion is $O(1)$ expected but may degrade to $O(T)$ during relocations.

Applications—Cuckoo filters are used in networking (e.g., packet routing, join operations), storage systems (e.g., approximate key-value caches), and databases. Their support for deletion makes them attractive for workloads with dynamic sets.

Limitations—Insertion failures occur when the filter is nearly full or relocations exceed T, requiring resizing. Also, it requires careful choice of fingerprint size f and bucket size b to balance space, false positive rate, and insertion success probability.

3.5. *Similarity Threshold (st)*

A similarity threshold defines the minimum required similarity between two strings (or objects) for them to be considered a match. In fuzzy matching systems, this threshold $\tau \in [0, 1]$ is typically applied to a normalized similarity measure. A candidate string y is considered a match to query string x if $\text{sim}(x, y) \geq \tau$. Choosing a higher τ increases precision but may reduce recall, whereas a lower τ allows more flexible matches at the cost of potential false positives.

4. BASELINE : BLOOM CUCKOO FILTER

4.1. *Overview*

The primary objective of our baseline method is to effectively perform fuzzy counting in streaming environments while ensuring both accuracy and privacy under memory constraints.

To achieve this, the solution integrates two powerful probabilistic data structures — Bloom Filters and Cuckoo Filters. Bloom filters first encode input strings by segmenting them into fixed-length q-grams and transforming these segments into compact bitwise representations. This encoding preserves similarity among input strings, enabling efficient approximate matching.

Subsequently, these encoded q-gram segments are stored in a Cuckoo Filter, a probabilistic data structure renowned for its high space efficiency, dynamic resizing capability, and low false positive rates. The Cuckoo filter facilitates rapid insertion, querying, and deletion operations, crucial for handling real-time streaming data effectively.

Moreover, to protect user privacy, the paper incorporates Local Differential Privacy (LDP) by introducing noise at the client-side through randomized responses, which generate self-constructed Bloom filters that obscure the actual input. This privacy-preserving mechanism ensures that no sensitive user data can be reliably inferred from the stored records. Experimental evaluations demonstrate the strength of our baseline approach. It significantly outperforms existing methods such as RAPPOR and Count-Min Sketch (CMS) by achieving substantially lower false positive rates, reduced query latency, and superior accuracy for both exact and fuzzy counting tasks. These results confirm that our proposed baseline is a robust, privacy-preserving, and efficient solution for fuzzy counting in streaming data contexts.

4.2. *Insertion*

Algorithm 1: Insertion (Client-side and server-side)

Input:
- x: A data/element
- m: Number of Bloom filter segments
- $bf(\cdot)$: Bloom filter encoding function
- $h(\cdot)$: Hash function
- $fp(\cdot)$: Fingerprint function
- max_num_kicks: Maximum number of kicks to relocate items

Output:
- **C**: Updated Cuckoo filter

Client-side:

$x' = bf(x)$ // Bloom filter encoding
$bf\_segs = x'.segment(m)$ // Split into m segments
$send\_to\_server(bf\_segs)$ // Send to server

Server-side:

for $bf\_seg \in bf\_segs$ **do**: // Iterate segments
  $f = fp(bf\_seg)$ // fingerprint
  $i1 = h(bf\_seg)$ // First bucket index
  $i2 = i1 \oplus h(f)$ // Second bucket index
  if $\mathbf{C}.bucket[i1]$ $not\ full$ **then**
    $\mathbf{C}.bucket[i1].add(f)$ // Add to first bucket
  else if $\mathbf{C}.bucket[i2]$ $not\ full$ **then**
    $\mathbf{C}.bucket[i2].add(f)$ // Add to second bucket
  else
    $i = random(i1, i2)$
    for $n = 0; n \leq max\_num\_kicks$ **do** // Relocate items
      $f' = random(\mathbf{C}.bucket[i])$
      $f, f' = f', f$ // Swap values
      $i = i \oplus h(f)$
      if $\mathbf{C}.bucket[i]$ $not\ full$ **then**
        $\mathbf{C}.bucket[i].add(f)$ // Add f to bucket i
        $break$ // Exit loop
      $insertion = fail$ // Insertion failed
  if $insertion == fail$ **then**
    $\mathbf{C}.bucket\_size+ = \mathbf{C}.bucket\_size$ // If full, adaptive filter
    $\mathbf{C}.bucket[i1].add(f)$ // Add to bucket i1
return **C** // Output **C**

Figure 1. Algorithm 1: Insertion

Each incoming data element x is first encoded on the client side into a Bloom filter bit vector $x' = bf(x)$, which is then split into m segments to preserve inter-string similarities. These segments are transmitted to the server, where each segment undergoes the following insertion procedure. For a given segment bf_seg, the server computes a compact fingerprint $f = fp(bf\_seg)$ and derives two candidate bucket indices:

$$i_1 = h(bf\_seg), \quad i_2 = i_1 \oplus h(f).$$

The algorithm attempts to insert f into the first non-full bucket among i_1 and i_2. If both buckets are already saturated, one bucket is chosen at random, and one of its existing fingerprints is evicted to make room for f. The evicted fingerprint then

repeats this same insertion logic into its alternate bucket. This relocation (or "kick") process continues for up to *max_num_kicks* iterations. If all relocation attempts fail—indicating that the filter is approaching capacity—the bucket size is automatically increased, and f is then placed directly into its original bucket. This adaptive resizing ensures that the filter maintains low false-positive rates and high throughput even under heavy streaming loads, supporting stable, real-time operation.

4.3. *Noise Addition for Differential Privacy*

Algorithm 2: Noise addition for local differential privacy (Client-side)

Input:
- x: A data/element
- $bf(\cdot)$: Bloom filter (BF) encoding function
- $h(\cdot)$: Hash function
- ϵ: Privacy budget
- s_t: Minimum similarity threshold
- p: Probability to flip bits $p = \frac{1}{1+s.e^{\epsilon}}$ where $s = \frac{2^{l.(1-s_t)}.B}{2^l}$ (Thm 3.1)
- F': Corresponding fingerprints for each bucket

Output:
- V': Perturbed vector containing fingerprints

$x' = bf(x)$ //BF of x
$i1 = h(x')$ //First bucket
$V = \{-1 \in \mathbb{R}^B\}$ //Initialize a vector
$V[i1] = +1$
$N \in \{-1, +1\}^B$, $Pr[n \in N = +1] = 1 - p$//Noise vector
$V = [v_1 n_1, \cdots, v_B N_B]$ //Flipping bits
for $v \in V$ **do**
  if $v == i1$ **then**
    $\tilde{x'} = generate\_similar\_bf(x', s_t)$ //similar BF with s_t
    $V'.add(\tilde{x'})$ //Add similar BF $\tilde{x'}$
  else
    $x" = randomly\_choose\_from(F'[r])$ //Corresponding BF
    $V'.add(x")$ //Add artificial BFs
return V' //Send V' to server

Figure 2. Algorithm 2: Noise Addition

To protect each user's input under ϵ-local differential privacy, the client perturbs the Bloom filter encoding of its element rather than flipping individual bits directly. Given an element x, the client first computes its Bloom filter representation $x' = bf(x)$ and determines the primary bucket index $i_1 = h(x')$. A length-B one-hot vector V is initialized with -1 in every position except $+1$ at index i_1. Randomized response noise is then applied: each entry of V is independently flipped with probability

$$p = \frac{1}{1 + s \cdot e^{\epsilon}} \quad \text{where} \quad s = \frac{2^l \, (1 - st) \, B}{2^l},$$

ensuring the resulting noisy vector still satisfies ϵ-LDP.

Next, for each position j in the perturbed vector, the client constructs an output Bloom filter: if the noisy bit $V[j]$ remains $+1$, a "similar" Bloom filter $\tilde{x}'$ is generated

by randomly flipping a fraction of bits to preserve at least the similarity threshold st; otherwise, an artificial Bloom filter is drawn uniformly at random from a precomputed set $F'[j]$ corresponding to bucket j. The collection of these B Bloom filters constitutes the perturbed vector V', which the client then transmits to the server. By sending entire Bloom filters rather than single bits, this mechanism maintains high utility while rigorously bounding privacy loss.

4.4. *Count querying with fuzzy matching*

Algorithm 3: Count querying with fuzzy matching (server-side)

Input:
- x: Query data/element
- m: Number of Bloom filter segments
- $bf(\cdot)$: Bloom filter encoding function
- $h(\cdot)$: Hash function
- $fp(\cdot)$: Fingerprint function
- s_t: Minimum similarity threshold

Output:
- c: Estimated count of x

$x' = bf(x)$ //Bloom filter
$bf\_segs = x'.segment(m)$ //Segments
$C = [\,]$
for $bf\_seg \in bf\_segs$ **do** //Iterate segments
  $f = fp(bf\_seg)$ //Fingerprint
  $i1 = h(bf\_seg)$ //First bucket
  $i2 = i1 \oplus h(f)$ //Second bucket
  if $\mathbf{C}.bucket[i1]$ *or* $\mathbf{C}.bucket[i2]$ *has* f **then**
    $C.add(count(f))$ //Add f's count to C
  else
    $C.add(0)$ //Add 0 count to C
  0-count_segs = $[j \in C \; if \; j == 0]$ // 0-count segments
  $sim_{max} = (m - |\text{0-count\_segs}|)/m$ //Max similarity
  if $sim_{max} \geq s_t$ **then**
    $c = min([j \; for \; j \in C \; if \; j > 0])$
  else
    $c = 0$
  return c

Figure 3. Algorithm 3: Fuzzy Matching

When a query for an element x arrives at the server, it is first encoded into its Bloom filter representation $x' = bf(x)$ and divided into m segments to form the list bf_segs. For each segment, a fingerprint $f = fp(bf\_seg)$ is computed, and the two candidate bucket indices

$$i_1 = h(bf\_seg), \quad i_2 = i_1 \oplus h(f)$$

are derived. The server then inspects both buckets for the presence of f: if found in either, the count $count(f)$ from that bucket is added to an accumulator list C; otherwise, zero is appended. After processing all m segments, the algorithm determines

how many segments yielded zero counts, computes the maximum achieved similarity as

$$sim_{max} = \frac{m - |\{j : C[j] = 0\}|}{m},$$

and compares it to the threshold st. If $sim_{max} \geq st$, the estimated count of x is taken as the minimum nonzero entry in C, representing the tightest lower bound across matching segments; otherwise, the count is reported as zero. This procedure ensures that only items matching at least st fraction of segments contribute to the final count, enabling robust fuzzy counting in one pass over the data structure.

4.5. *Performance*

Our experimental results demonstrate that the Bloom–Cuckoo baseline achieves competitive performance across multiple metrics. Insertion time remains slightly higher than state-of-the-art RAPPOR and CMS, yet within less than one order of magnitude, making it acceptable for streaming applications where insertions can be batched or parallelized. Query latency exhibits superior efficiency, typically under 1-1.5 orders of magnitude in seconds, enabling real-time responsiveness.

False positive rates remain near zero even after one million insertions, whereas CMS and RAPPOR escalate to full saturation (false positive rate of 1) at approximately 10,000 and 500,000 records, respectively. In both exact and fuzzy counting scenarios, the variance of count estimates under our Cuckoo filter approach outperforms RAPPOR and CMS by two to three orders of magnitude, confirming both high accuracy and stability.

These results underscore that our baseline successfully balances throughput, accuracy, and memory efficiency. Having established a robust foundation, we now turn to an alternative perspective based on HyperLogLog sketches to further improve latency and space usage while preserving fuzzy matching capabilities.

5. THE HYPERLOGLOG ALGORITHM

5.1. *Motivation*

Although D. Vatsalan et al. (2022) amongst many others seemed to try and address the problem of excessive memory usage when handling streaming data, by relying on various methods of fuzzy counting, the truth is that is only one perspective. Suggested by D. Vatsalan et al. (2022) in their future work, probabilistic data structures, especially HLL, were said to have potential to be used to decrease local storage and increase memory efficiency.

First suggested by P. Flajolet et al. (2007), HLL is a probabilistic data sketching algorithm which estimates the cardinality (i.e. the number of distinct elements) of a multiset using sublinear memory. Typically, an HLL sketch consists of a small fixed number of m small registers, creating a constant-size data structure with respect to the input size, requiring $O(m)$ space. In addition, HLL supports efficient element insertion, cardinality estimation, and merging of sketches in $O(m)$ time, since given

two HLL sketches M_1 and M_2, their merged sketch M is simply the element-wise maximum: $M[i] = \max(M_1[i], M_2[i])$ for $i = 1, \ldots, m$.

Additionally, due to the nature of the data sketching, HLL does not store full inputs, but rather naturally introduces obfuscation, which leaves less footprints, suggesting a perhaps possible easier step towards reaching privacy.

While HLL was not designed for language modeling, in the context of the streaming data problem, HLL could be repurposed to estimate the number of distinct users who submit a particular 'signal' (e.g., a q-gram), enabling us to identify which signals are most commonly seen in the stream. In the context of autocorrection, this may be beneficial information to know, which gives HLL a promising applicability in this problem, as it offers both practical feasibility and low computational overhead in terms of space and time complexity.

5.2. *Challenges*

Recall, the Bloom-Cuckoo algorithm approaches autocorrection through fuzzy matching, using approximate encodings to tolerate variation in user input. In contrast, HLL does not support approximate matching, as it only considers exact occurrences of items during insertion. To adapt HLL to the autocorrection setting, a reframing of the underlying task is required.

Rather than treating autocorrection as a search for the closest match to a given noisy input, we instead interpret it as a problem of q-gram popularity scoring. That is, each input string is decomposed into overlapping substrings (q-grams), and we attempt to identify which q-grams are most frequently observed across users. The assumption is that highly frequent q-grams are more likely to correspond to correct or intended inputs. Under this formulation, HLL remains applicable, since exactness is no longer a strict constraint, and fuzzy similarity is replaced by frequency-based inference.

However, a significant challenge arises in the decoding step: Bloom-Cuckoo stores uncompressed data and thus allows direct reconstruction of candidate words via matched fingerprints. In contrast, HLL is a lossy, non-reversible sketch, and does not retain any association with original inputs. This raises the question of how to suggest correction candidates solely based on q-gram frequency estimates.

To address this, we propose maintaining an auxiliary q-gram-to-candidate map, denoted by $\varphi : \mathcal{G} \to 2^{\mathcal{C}}$, where $\mathcal{G}$ is the set of all observed q-grams and $\mathcal{C}$ is the set of candidate words. This map can be implemented using a standard dictionary in Python, where each q-gram points to the set of words it appears in. When a query is made, we retrieve all candidate words associated with the query's q-grams, rank them by their cumulative estimated frequencies (obtained via HLL), and return the top-ranked word(s).

While this approach sacrifices some of HLL's memory efficiency, it retains its favorable time complexity. In practice, the memory usage remains significantly lower than

that of Bloom-Cuckoo, while providing a practical mechanism for recovering likely word suggestions from compressed frequency information.

5.3. *Algorithms (No Privacy Version)*

Despite the data structure used being different, in essence, the HLL algorithm we propose for autocorrection is similar in theory to the Bloom-Cuckoo counterpart on a high-level perspective, since even if the algorithm is different, fundamentally the concept is similar, with the difference being we consider q-gram popularity scoring instead of fuzzy similarity.

In this section, we present the non-LDP (non-private) version of the HLL algorithm for autocorrection. Although privacy mechanisms such as LDP are not applied here, the structure allows for straightforward extension to a privacy-preserving setting, which we discuss in future work.

On a high level, the non-private Bloom-Cuckoo and HLL pipelines proceed as outlined in **Table 1**. Following the table, we provide the detailed pseudocode for the HLL autocorrection algorithm, separated into Phase 1 and Phase 3 as outlined in Table 1 to match the Bloom-Cuckoo algorithm's structure. The following pseudocode assumes a HLL library and a maintained φ q-grams-to-candidate mapping.

Table 1. Comparison of Bloom-Cuckoo and HyperLogLog Algorithms (No Privacy Version)

Bloom-Cuckoo	**HyperLogLog**
Phase 1: Fuzzy Encoding - Divide every input string into q-grams - Use locality-sensitive hashing for approximate matching	**Phase 1: Extract q-gram Encoding** - Split every string into q-grams - Insert each (q-gram, user) into HLL sketch (insert "{q-gram}_{user}", e.g. "ap_user1", if user1 types "apple")
Phase 3: Aggregation & Decoding - Server collects noisy sketches from multiple users	**Phase 3: Candidate Scoring & Output** - Maintain a $\varphi : \mathcal{G} \to 2^{\mathcal{C}}$ mapping - For each query: · Tally shared q-grams with each candidate · Score = Estimated overlap count by HLL frequency

Algorithm 1 Extract q-gram Psseudocode

Require: User inputs $\{(u_i, w_i)\}$, q-gram size q, sketch config parameter b
Ensure: q-gram sketches `SketchMap`, q-gram to candidate words map φ
1: **function** ExtractQGrams(w, q)
2: **if** $|w| < q$ **then return** $\emptyset$
3: **else return** $\{w_i, w_{i+1}, \ldots, w_{i+q-1} \mid 0 \leq i \leq |w| - q\}$

Algorithm 2 Phase 1 Pseudocode — q-gram Extraction and Sketching

Require: User inputs $\{(u_i, w_i)\}$, q-gram size q, sketch config parameter b
Ensure: q-gram sketches `SketchMap`, q-gram to candidate words map φ

Initialize empty map `SketchMap`: $g \mapsto$ HLL sketch
Initialize empty map φ: $g \mapsto 2^{\mathcal{C}}$
for all user-word pairs (u, w) **do**
    $Q \leftarrow$ ExtractQGrams(w, q)
    for all $g \in Q$ **do**
        $\varphi[g] \leftarrow \varphi[g] \cup \{w\}$
        if $g \notin$ `SketchMap` **then**
            Initialize new HLL sketch with config b and assign to `SketchMap`$[g]$
        Insert unique string "g_u" into `SketchMap`$[g]$

Algorithm 3 Phase 3 Pseudocode — Query Processing and Correction Suggestion

Require: Query set Q, SketchMap, mapping $\varphi : \mathcal{G} \to 2^{\mathcal{C}}$
Ensure: Suggested correction per query

for all $q \in Q$ **do**
    $\tilde{Q} \leftarrow$ ExtractQGrams(q, q-gram size)
    Initialize score map $S \leftarrow \emptyset$
    Initialize merged sketch M_q for user estimation
    Initialize set of matched grams $G_q \leftarrow \emptyset$
    for all $g \in \tilde{Q}$ **do**
        if $g \in$ SketchMap **then**
            $e \leftarrow$ Estimate from HLL sketch of g
            Add g to G_q, merge HLL of g into M_q
            for all candidate $c \in \varphi[g]$ **do**
                $S[c] \leftarrow S[c] + e$
    if $|G_q| \geq 3$ **then**
        $u \leftarrow$ Estimate(M_q)
        $c^\star \leftarrow \arg\max_c S[c]$
        Output $c^\star$ as suggested correction and u as estimated users
    else
        Output "Not enough matching segments"

6. EXPERIMENT

6.1. *Dataset*

In this section, we discuss how we tested our design. We evaluated our program using two datasets. The first dataset is from https://github.com/first20hours/google-10000-english/blob/master/google-10000-english-no-swears.txt, which provides 9,894 English vocabulary words. The second dataset includes common spelling errors, obtained from https://www.kaggle.com/datasets/bittlingmayer/spelling and https://en.wikipedia.org/wiki/Wikipedia:Lists_of_common_misspellings/For_machines.

Using these datasets, we tested various metrics including accuracy, execution time, and memory usage. More details are discussed in the following paragraphs.

6.2. *Design*

We evaluated our design differently from the baseline paper. The baseline paper's definition of accuracy tolerates a single-letter difference between the output and the correct word. However, we argue that autocorrection systems should not be evaluated this way, as even a single incorrect letter can significantly affect usability, and especially because the focus of our project is on autocorrection rather than LDP like in the baseline paper. Therefore, our accuracy metric only accepts perfectly spelled words as correct.

The parameters used in our approach were selected through trial and error. To address overfitting, we evaluated our model on an additional typo dataset to ensure generalization.

6.3. *Accuracy*

We tested our system under two scenarios:

- **Scenario 1:** Database of 528 words, Input of 528 words
- **Scenario 2:** Database of 9,894 + 528 words, Input of 528 words

The results were as follows:

- **Scenario 1:** Baseline: 76.52% HLL: 72.16%
- **Scenario 2:** Baseline: 66.66% HLL: 72.16%

6.4. *Execution Time*

We measured execution time using the combined dataset (9,894 + 528 words), resulting in the following times:

- Baseline: 5.472 seconds HLL: 0.562 seconds

6.5. *Memory Usage*

We measured memory usage using `/usr/bin/time -v` while running the system with the combined dataset (9,894 + 528 words). The HLL implementation used 7.89% less memory compared to the baseline.

7. PYTHON LIBRARY

We create a python library for text autocorrection using our method. User can import our package using

```
# sample usage
!pip install hll_py
from hll_py import autocorrect
history_input = [("u1","apple"),
("u2", "banana"),
("u3", "grape"),
("u4", "orange"),
("u5", "banana")
```

```
]
queries = ["applle", "banana", "banan", "orenge", "grap", "pineapple"]

results = autocorrect(history_input, queries)

# sample output
  -> Suggested correction: apple
------------------------------

  -> Suggested correction: banana
------------------------------

  -> Suggested correction: banana
------------------------------

  -> Suggested correction: orange
------------------------------

  -> Suggested correction: grape
------------------------------

  -> Suggested correction: apple
```

8. EVALUATION

8.1. *Was HyperLogLog Given a Fair Chance?—Why Did it Perform So Well?*

Analyzing the results as shown in the experiment, we notice the accuracy of Bloom-Cuckoo and HLL algorithms are almost the same in **Scenario 1**. Yet, when an additional 9,894 words were added to the database in **Scenario 2**, the accuracy of the Bloom-Cuckoo algorithm dropped by around 10 percentage points, whereas the accuracy of the HLL algorithm remained the same.

Upon further inspection of the outputs by the HLL algorithm, we notice an interesting pattern with a select few of its wrong autocorrection suggestions. When considering the typo 'titalate' using the database in **Scenario 1**, the Bloom-Cuckoo algorithm was able to correctly suggest the correction 'titillate', whereas the HLL algorithm suggested 'international', as shown in **Figure 4**.

```
  titalate -> matched segments: ti(114.85866666666668) it(84.8187076923077) ta(18.610412956890
894) al(1378.304) la(18.610412956890894) at(153.1448888888889) te(344.576)
  -> Estimated users: 11026.432
  -> Suggested correction: international
```

Figure 4. HLL autocorrection algorithm's suggested correction to 'titalate' is 'international' in **Scenario 1**

This test case highlights the biggest structural difference between the Bloom-Cuckoo and HLL algorithms. The Bloom-Cuckoo algorithm utilizes fuzzy matching via segment overlap of fingerprints, and does not rely directly on frequency. Thus, picks a candidate with many shared segment encodings even if it is not exact. By contrast, HLL is a probabilistic counter, and in the HLL algorithm, it estimates how many users typed each exact q-gram without fuzziness. More importantly, the algorithm

fundamentally relies on segment-level frequency with respect to the database to suggest results, meaning it would suggest words in the database with high segment-level frequency, even if they are unrelated, as shown in the example in **Figure 4**.

There were other instances, where the Bloom-Cuckoo algorithm also suggested wrong corrections, where the typo was clearly of a very common word in real life, but the suggested correction suggested a very obscure word. In a deprecated attempt of doing "top-3 suggestions" like what is typically done on mobile devices today, the HLL algorithm suggested obscure corrections to the query 'applw' when fed a 370,099 word dictionary, as shown in **Figure 5**.

```
Query: applw -> Suggestions: ['aplopappus', 'appl', 'applanate']
  -> Estimated users: 2822766.592
```

Figure 5. HLL autocorrection algorithm's top-3 suggested correction to 'applw'

These examples highlight a critical limitation: the HLL-based method is highly sensitive to how words are weighted in the dataset. If all words are treated uniformly, the algorithm tends to overfit to high-frequency q-grams that appear in many unrelated words. Therefore, HLL can only be fairly evaluated in scenarios where the word frequency distribution reflects real-world usage (e.g., Zipfian weighting).

However, interestingly, when using **Scenario 2** to test the typo 'titalate' with the HLL autocorrection algorithm, it was able to suggest to correct the result correctly to 'titillate'. Perhaps, it is because the larger dataset was able to capture the distributions of q-grams more accurately to real-world data, because even if it did not use Zipfian weighting, the added vocabulary in fact helps HLL to simulate a more real-world usage weighted distribution of q-grams.

In **Scenario 2**, where no such weighting was applied and in fact unrelated words were significantly more heavily weighted than they should in real life, HLL still achieved over 70% accuracy by our strict correctness metric, and has the same accuracy as in **Scenario 1**. The fact that even queries such as 'titalate' resulted in correct suggestions in **Scenario 2** and not in **Scenario 1** by the HLL autocorrection algorithm, further suggests the importance of a fairly weighted q-gram and hence word distribution.

This strongly suggests that, if provided with a realistically weighted vocabulary, HLL may perform even better. Its ability to not drop in accuracy when changing between both databases as opposed to Bloom-Cuckoo, with its substantial gains in speed and memory efficiency, may justify its use, especially in distributed or resource-constrained environments.

8.2. *Remark on Implementation*

In our previous attempts of testing for HLL autocorrection, even if we believed to have followed the HLL algorithms perfectly, it is very easy to make seemingly minor

mistakes which change the result of everything. In our case, there was a small error with the leading zeros counter.

This bug resulted in unchanged accuracy for **Scenario 1**, but a 20 percentage point lower accuracy in **Scenario 2**. At first, we assumed the accuracy drop was due to HLL struggling to generalize to larger datasets. However, after correcting our bug, we found that HLL was actually far more robust to such changes than expected, maintaining stable performance even when Bloom-Cuckoo degraded. Thus, in real-world implementations, perhaps it is better to call an already established correct HLL library to avoid such errors.

9. FUTURE WORK

9.1. *A Sketch of HyperLogLog LDP*

Although HLL appears to offer significantly more benefits than drawbacks—especially compared to the Bloom-Cuckoo autocorrection algorithm, there is still very strong argument to justify not using HLL, which is its lack of LDP privacy. Although its existing obfuscation is better than none at all, an adversary may still be able to deduce what words a user inputs by comparing the similarity in output. However, based on the HLL algorithm's logic, any LDP method could still be used, as long as it can preserve the same merging system, and still maintain LDP after each merge.

An idea may be to reference how the Bloom-Cuckoo algorithm achieved LDP by flipping bits. In the case of HLL, we could create noise on HLL register bits. A technical detail is HLL requires the maximum number of leading zeros as a huge factor in determining estimated cardinality, thus a plausible approach may be to flip high registers with low probability and low registers with high probability. That way the noise is still unpredictable but its affect on HLL may be minimal enough, unlike if we for example only flip low register bits which offer a certain degree of knowledge to the adversary of what could have been the original queries.

Two more plausible ideas may be to use local hashing of user identity to preserve anonymity but still get per-user uniqueness, or if each user only reports a random subset of their q-grams. This way, the fundamental HLL structure does not change unlike with the idea above, which may be riskier. However, whether this can achieve true LDP, or whether this would still achieve accurate results is open for future research. Perhaps, a plausible research direction if the former idea of local hashing of user identity were to be investigated, would be to see how to ensure good enough hashing, or else there may be identity collisions, which would be problematic if they occur too often.

In short, we must keep in mind of the privacy-utility tradeoff: too much noise breaks accuracy, too little fails LDP.

9.2. *A Sketch of fusing HyperLogLog with Bloom-Cuckoo*

Alternatively, if LDP on pure HLL appears to be too difficult, the fusion of Bloom-Cuckoo and HLL may be a way to preserve LDP yet preserve the benefits HLL has

with time and space efficiency, since D. Vatsalan et al. (2022) has proven the Bloom-Cuckoo algorithm achieves LDP.

The Bloom-Cuckoo algorithm itself relies on fingerprint counting of each bucket, which takes a lot of memory when the exact values are stored. Perhaps, HLL can be used instead within each bucket, to decrease memory usage and perhaps time, yet still maintain results that are accurate with high probability.

However, it may be hard to implement, since there is no reverse mapping with HLL, meaning we cannot tell what input led to a change in bits. This means we cannot 'undo' changes words had to the data — We can blacklist words from being suggested, but we cannot fully remove the affects of those words on our data. Additionally, if future progress were to be made with suggesting top-k words instead of the first most likely word with autocorrection, HLL's precision may decrease drastically since it uses approximate counting. It would be a matter of finding a balance between accuracy and cost, to have the decrease in cost be significant enough to justify the possible decrease in accuracy.

9.3. *A Sketch of Optimizing Q-gram Candidate Mappings for Large Vocabularies*

Finally, pure HLL itself is a very space-efficient algorithm, but the HLL autocorrection algorithm developed in this project required to maintain a space-taxing $\varphi : \mathcal{G} \to 2^{\mathcal{C}}$ q-gram-to-candidate mapping, where every q-gram maps to an explicit set of candidate words. Although it is still fine with smaller amounts of vocabulary, if there are a million or more like in real life scenarios, it may consume a lot of memory. One open problem is optimizing the q-gram-to-candidate mapping φ for large-scale vocabularies. Future work could explore more compact or approximate representations of φ, such as trie-based structures that compress shared prefixes, or clustering-based methods that map q-grams to sets of representative candidates rather than all matches. Additionally, sketching or hashing techniques such as MinHash or SimHash may be used to reduce redundancy and accelerate lookup without a significant loss in accuracy.

10. CONCLUSION AND TAKEAWAYS

In conclusion, HyperLogLog is a space-efficient, fast, and easily mergeable data structure perfect for cardinality estimations. In this project, we investigated how HLL can be a promising direction towards autocorrection in a new light, with an acceptable accuracy deficit for a significant runtime decrease. Surprisingly, it may even be more robust than other currently widely-accepted algorithms and approaches to autocorrection which focus on fuzzy counting, with its new angle of 'q-gram popularity scoring'. This suggests a new possible usage of HLL with language processing, and how data sketching is a plausible angle to handle some NLP-adjacent problems.

In addition, future work suggests a promising possibility that LDP can be done on HLL natively, or fused with Bloom-Cuckoo, and the memory usage could be even more optimized based on how our q-gram-to-candidate mapping is implemented. Perhaps in

today's world where privacy is increasingly-valued, a Zipfian-weighted database could be used to implement HLL-based autocorrection in place of current autocorrection algorithms. Such an approach may eliminate the need to store egregious amounts of data in a central system, yet still maintain privacy, which suggests a very promising privacy-preserving autocorrection suitable in distributed and embedded systems with high speed.

REFERENCES

Cormode, G., & Muthukrishnan, S. 2004, in Lecture Notes in Computer Science, Vol. 2976, Proceedings of the Latin American Theoretical Informatics Symposium (LATIN), ed. R. Baeza-Yates (Buenos Aires, Argentina: Springer), 29–38, doi: 10.1007/978-3-540-24698-5_3

Dwork, C. 2006, in Lecture Notes in Computer Science, Vol. 4052, Proceedings of the 33rd International Colloquium on Automata, Languages and Programming (ICALP) (Springer), 1–12, doi: 10.1007/11787006_1

Flajolet, P., Fusy, É., Gandouet, O., & Meunier, F. 2007, in DMTCS Proceedings, Vol. DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07), DMTCS Proceedings (Discrete Mathematics and Theoretical Computer Science), 137–156, doi: 10.46298/dmtcs.3545

Vatsalan, D., Bhaskar, R., & Kaafar, M. A. 2022, arXiv preprint arXiv:2208.05264. https://arxiv.org/abs/2208.05264

Úlfar Erlingsson, Pihur, V., & Korolova, A. 2014, in Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS) (Scottsdale, Arizona, USA: ACM), 1054–1067, doi: 10.1145/2660267.2660348