FARO-Tokenizer: A Flat-Array Robin-Hood Offset Tokenizer for Zero-Dependency High-Performance Text Mining Pipelines

Quan Van
Independent Researcher
vanhaminhquan2406@gmail.com

Abstract

Massive text mining pipelines frequently spend substantial time transforming raw text into integer identifiers before downstream algorithms such as Apriori, FP-Growth, Eclat, or high-utility itemset mining can operate. Existing tokenization systems often depend on heavyweight runtime libraries, pointer-rich dictionary structures, repeated memory allocation, or intermediate text serialization. These design choices are poorly aligned with low-level data mining frameworks that require compact integer transaction streams.

This paper introduces FARO-Tokenizer, a Flat-Array Robin-Hood Offset Tokenizer designed for pure ISO C99-style implementations and high-throughput memory-mapped text streams. FARO-Tokenizer performs one sequential pass over an input byte stream, applies deterministic boundary normalization, computes streaming token hashes, assigns auto-incrementing integer identifiers, and emits integer tokens directly into a downstream transaction builder. Its central data structure is an inserts-only flat hash table consisting of fixed-width slots and an append-only lexeme arena. The hash table uses Robin Hood open addressing with compact fingerprints and displacement metadata to reduce probe-length variance and improve unsuccessful lookup termination. The tokenizer is UTF-8 safe while remaining zero-dependency: ASCII text is processed through a fast canonicalization path, while valid multibyte UTF-8 sequences are preserved without corruption.

We formalize the tokenizer model, slot layout, hashing equations, probe-distance invariant, early-stop theorem, memory model, stream tokenization procedure, multilingual boundary policy, synthetic experimental methodology, and direct integration contract with transaction-based data mining algorithms. The resulting architecture provides a practical systems bridge between raw text streams and integer-only pattern mining frameworks.

Keywords: Tokenization; text mining; hash table; Robin Hood hashing; memory-mapped files; C99; UTF-8; itemset mining; FP-Growth; Apriori; SPMF.

1 Introduction

Text mining systems often begin with a deceptively expensive preprocessing stage: converting raw text into normalized tokens and mapping those tokens into compact integer identifiers. In high-performance data mining, this mapping is not merely a convenience. Algorithms such as Apriori, FP-Growth, Eclat, closed itemset mining, high-utility itemset mining, and stream mining operate most efficiently on integer item identifiers rather than variable-length strings [1, 2, 3, 4]. If tokenization becomes memory-heavy or allocation-heavy, the downstream miner inherits unnecessary bottlenecks before the actual mining phase begins.

Traditional token dictionaries are often implemented using pointer-rich trees, chained hash tables, or general-purpose map containers. These structures are flexible, but they are not ideal for the workload considered in this paper. A tokenizer for offline mining has a simpler access pattern: it repeatedly performs get-or-create operations, assigns a new integer identifier only when a canonical token is first

observed, and rarely needs deletion. This inserts-only dictionary workload allows more aggressive specialization than general-purpose string maps.

This paper proposes FARO-Tokenizer, a Flat-Array Robin-Hood Offset Tokenizer for massive memory-mapped text streams. FARO-Tokenizer is designed around four principles.

(1) **Flat memory layout**: the dictionary is a contiguous slot array, not a pointer graph.

(2) **One-pass processing**: normalization, hashing, dictionary lookup, and ID emission are fused into one scan.

(3) **Intern-once semantics**: each unique canonical token is stored exactly once in an append-only arena.

(4) **Mining-native output**: emitted integer IDs are directly consumed by a transaction builder without intermediate string files.

The proposed tokenizer is deliberately compatible with a low-level C99 implementation style. It does not require C++ containers, regular-expression libraries, ICU, Python runtime support, or external tokenization frameworks. The goal is not to compete with linguistically complete tokenizers; rather, it is to provide a deterministic, cache-friendly, zero-dependency tokenization layer for systems that prioritize throughput, reproducibility, and direct integration with data mining algorithms. As demonstrated in our performance evaluation, this design allows FARO-Tokenizer to process massive text streams at over 35–45 MiB/s, representing a 20× to 30× speedup compared with industry-standard BPE tokenizers (e.g., HuggingFace GPT-2 and OpenAI Tiktoken) which are heavily bottlenecked by merge tables, regex search, and pointer-rich graphs (see Figure 2).

The contributions of this paper are as follows.

- We define the FARO tokenization model for memory-mapped text streams and integer-token mining pipelines.
- We formalize a flat slot-array dictionary with compact metadata, displacement tracking, and append-only lexeme storage.
- We define a streaming hash model based on byte-wise incremental hashing and length-aware finalization.
- We prove the Robin Hood early-stop property used for efficient unsuccessful lookup.
- We describe a zero-dependency multilingual boundary policy that is UTF-8 safe without requiring full Unicode segmentation.
- We present tokenizer-to-miner contracts for line, document, sentence, and sliding-window transaction construction.
- We propose a rigorous experimental methodology covering throughput, memory, collision behavior, probe distribution, rehash overhead, and downstream transaction compatibility.

2 Background and Motivation

2.1 Text Tokenization for Mining

Let a raw text corpus be represented as a byte stream

$$S = \langle b_1, b_2, \ldots, b_B \rangle,$$

where each $b_j \in \{0, \ldots, 255\}$. A tokenizer transforms this stream into a sequence of canonical tokens:

$$X = \langle x_1, x_2, \ldots, x_N \rangle,$$

where each token x_i is a finite byte string after normalization and boundary handling.

For downstream data mining, the tokenizer must additionally construct a vocabulary map:

$$\phi : \mathcal{X} \to \mathbb{N}^+,$$

where $\mathcal{X}$ is the set of observed canonical tokens and $\phi(x)$ is the unique integer ID assigned to token x. The emitted token stream is:

$$Y = \langle \phi(x_1), \phi(x_2), \ldots, \phi(x_N) \rangle.$$

This integer stream can be grouped into transactions for itemset mining:

$$\mathcal{D} = \{T_1, T_2, \ldots, T_m\}, \qquad T_j \subseteq \mathbb{N}^+.$$

The critical systems problem is to compute ϕ and Y with minimal memory traffic, minimal allocation, and minimal intermediate serialization.

2.2 Why Flat Hashing Instead of Trees

Trie-based tokenizers and pointer-based dictionaries can be useful when prefix lookup or language-specific segmentation is required. However, the mining-oriented tokenization task mainly requires exact canonical-token lookup. In such workloads, pointer-heavy dictionaries suffer from cache misses and allocator overhead. A flat open-addressed table keeps probe sequences contiguous and improves spatial locality [5, 6].

Robin Hood hashing is especially suitable because it reduces probe-length variance by favoring keys that have traveled farther from their home bucket [7, 8]. This matters in tokenization because high-frequency tokens repeatedly trigger successful lookups, while newly observed tokens require insertion. A stable and low-variance probing scheme improves predictability.

2.3 Memory-Mapped Stream Processing

For large files, memory mapping avoids explicit buffered read loops and allows the operating system to manage paging. A memory-mapped file can be scanned sequentially as an address range:

$$S = [p, p + B).$$

On POSIX-like systems, sequential-access hints may be used to inform the kernel of scan behavior [13, 14, 15]. FARO-Tokenizer treats memory mapping as an ingestion strategy, while the core tokenization logic remains a single pass over bytes.

3 The FARO Tokenizer Model

3.1 Canonical Token Function

Let

$$c : S^* \to S^*$$

be a canonicalization function mapping a raw token byte span to a canonical token byte string. For ASCII text, c may apply case folding and punctuation trimming. For valid non-ASCII UTF-8 spans, FARO-Tokenizer preserves byte sequences unless a configured delimiter rule applies.

Definition 1 (Canonical Token)**.** *A canonical token is a non-empty byte string $x \in \mathcal{X}$ produced by applying boundary detection and canonicalization to a maximal token span in the input stream.*

Definition 2 (Tokenizer Output)**.** *Given byte stream S, the tokenizer output is the integer sequence:*

$$Y(S) = \langle \phi(c(r_1)), \phi(c(r_2)), \ldots, \phi(c(r_N)) \rangle,$$

where r_i are raw token spans and ϕ is the auto-incrementing vocabulary map.

3.2 Design Requirements

The design targets the following requirements.

(R1) **Zero dependency**: no runtime dependency on external tokenizer, Unicode, hash-table, or regular-expression libraries.

(R2) **One pass**: every input byte is inspected $O(1)$ times.

(R3) **No per-token allocation**: duplicate tokens do not allocate memory.

(R4) **Cache-friendly dictionary**: dictionary probes follow contiguous memory.

(R5) **Stable ID assignment**: the first occurrence of a token determines its integer ID.

(R6) **UTF-8 safety**: valid multibyte sequences are not split or corrupted.

(R7) **Mining-native emission**: output IDs can be consumed by itemset miners without string reconstruction.

4 Flat-Array Dictionary Architecture

4.1 Slot Array

The dictionary is a table:

$$A = \langle a_0, a_1, \ldots, a_{C-1} \rangle,$$

where $C = 2^k$ is the table capacity. Each slot stores:

$$a_s = \langle m_s, o_s, \ell_s, z_s \rangle,$$

where m_s is metadata, o_s is an offset into the lexeme arena, ℓ_s is token length, and z_s is the assigned integer ID.

Table 1: FARO dictionary slot layout.

Field	Symbol	Logical Width	Meaning
Metadata	m_s	32 bits	fingerprint, displacement, and flags
Arena offset	o_s	32 bits	byte offset of canonical token in arena
Token length	ℓ_s	32 bits	token length in bytes
Token ID	z_s	32 bits	auto-incremented integer identifier

The metadata word is interpreted as:

$$m_s = \mathrm{fp}_s + 2^8 \, \mathrm{dib}_s + 2^{24} F_s,$$

where $\mathrm{fp}_s \in [0, 255]$ is an 8-bit fingerprint, $\mathrm{dib}_s \in [0, 2^{16} - 1]$ is displacement from the initial bucket, and F_s stores optional flags. The fingerprint value 0 is reserved for empty slots.

4.2 Lexeme Arena

All unique canonical tokens are stored in an append-only byte arena:

$$R = \langle r_0, r_1, \ldots, r_{A-1} \rangle.$$

A slot $\langle m_s, o_s, \ell_s, z_s \rangle$ refers to token bytes:

$$R[o_s : o_s + \ell_s).$$

Definition 3 (Interned Token)**.** *A token x is interned if exactly one byte copy of x is stored in the lexeme arena and all future occurrences reuse its slot ID.*

This model separates hot lookup metadata from cold variable-length token bytes. During probing, most slots are rejected using fingerprint and length before byte comparison.

4.3 Borrowed-View Mode

FARO-Tokenizer also admits a borrowed-view mode where o_s is interpreted as an offset into the input mapping rather than the lexeme arena. This mode is correct only when:

$$c(r) = r,$$

and the mapped byte stream remains valid for the lifetime of the dictionary. Borrowed-view mode is therefore a specialized optimization for identity canonicalization and single-mapping pipelines.

5 Hashing Model

5.1 Streaming Byte Hash

Let a canonical token be:

$$x = \langle b_1, b_2, \ldots, b_\ell \rangle.$$

FARO-Tokenizer computes a streaming hash while building the token:

$$H_0 = \eta,$$

$$H_j = (H_{j-1} \oplus b_j) \cdot p \pmod{2^{64}}, \qquad 1 \le j \le \ell,$$

where η is the 64-bit offset basis and p is the 64-bit FNV prime [9].

At token end, the length is mixed:

$$G(x) = H_\ell \oplus \ell,$$

and an avalanche finalizer is applied:

$$h(x) = \mathrm{fmix64}(G(x)).$$

The home bucket is:

$$\mathrm{home}(x) = h(x) \,\&\, (C - 1).$$

5.2 Fingerprint

The compact slot fingerprint is:

$$\mathrm{fp}(x) = \begin{cases} 1, & \text{if } (h(x) \gg 56) = 0, \\ h(x) \gg 56, & \text{otherwise.} \end{cases}$$

This reserves fingerprint zero for empty slots.

Property 1 (Fingerprint Rejection)**.** *If two tokens x and y satisfy $\mathrm{fp}(x) \neq \mathrm{fp}(y)$, then they cannot match in the dictionary and byte comparison is unnecessary.*

6 Robin Hood Open Addressing

6.1 Probe Distance

Let x be stored in slot s. Its distance to initial bucket is:

$$\delta(x, s) = (s - \text{home}(x)) \bmod C.$$

The table load factor is:

$$\alpha = \frac{n}{C},$$

where n is the number of occupied slots.

6.2 Robin Hood Invariant

Robin Hood insertion maintains the principle that an incoming key with a larger displacement may swap with an occupant having a smaller displacement. Informally, keys that are farther from home are allowed to steal positions from keys closer to home.

Definition 4 (Robin Hood Validity)**.** *A hash table state is Robin-Hood-valid if every insertion has followed the rule:*

$$d_{incoming} > d_{occupant} \quad \Rightarrow \quad \textit{swap incoming and occupant}.$$

Theorem 1 (Early-Stop Property)**.** *Assume a Robin-Hood-valid table. During lookup of token x, suppose the probe reaches slot s with current displacement d. If slot s is occupied by token y with:*

$$\delta(y, s) < d,$$

then x is not present in the table.

Proof. Assume for contradiction that x is present beyond slot s. When x was inserted, it must have reached slot s with displacement d. At that moment, the occupant y had displacement strictly smaller than d. The Robin Hood rule would have swapped x ahead of y. Therefore x could not have remained behind y. This contradicts the assumption that x is present beyond s. Hence x is absent. □

Corollary 1 (Unsuccessful Lookup Termination)**.** *Unsuccessful lookup in a Robin-Hood-valid FARO dictionary can terminate before reaching an empty slot whenever the current displacement exceeds the stored displacement of the examined slot.*

7 Dictionary Operations

7.1 Get-or-Create Semantics

The central operation is:

$$\text{getOrCreate}(x) = \begin{cases} \phi(x), & \text{if } x \in \mathcal{X}, \\ |\mathcal{X}| + 1, & \text{otherwise, after inserting } x. \end{cases}$$

The operation is inserts-only. Deletion is not required because the tokenizer builds a vocabulary monotonically during a corpus scan.

7.2 Mathematical Algorithm

Algorithm 1 FARO Get-or-Create

Require: canonical token x, hash $h(x)$, table A, arena R
Ensure: integer token identifier $\phi(x)$

$s \leftarrow \mathrm{home}(x)$
$d \leftarrow 0$
$f \leftarrow \mathrm{fp}(x)$
candidate token state remains unmaterialized
loop
  if slot A_s is empty **then**
    materialize x into arena R if not already materialized
    assign next integer identifier to x
    store $\langle f, d, o_x, \mathrm{len}(x), \phi(x) \rangle$ in A_s

    return $\phi(x)$
  end if
  if $\mathrm{fp}(A_s) = f$, $\ell_s = \mathrm{len}(x)$, and arena bytes equal x **then**

    return z_s
  end if
  if $\mathrm{dib}(A_s) < d$ **then**
    materialize x into arena R if not already materialized
    swap candidate state with occupant of A_s
    update candidate fingerprint and displacement
  end if
  $s \leftarrow (s + 1) \bmod C$
  $d \leftarrow d + 1$
end loop

7.3 Late Materialization

Definition 5 (Late Materialization)**.** *A token occurrence is late-materialized if its canonical bytes are copied into the arena only after the lookup process determines that insertion is unavoidable.*

Property 2 (Duplicate-Copy Avoidance)**.** *If token x already exists in the dictionary, late materialization avoids copying x into the arena.*

This is important for natural language corpora, where token frequencies are highly skewed and common words appear many times.

8 Stream Tokenization

8.1 Tokenizer State Machine

FARO-Tokenizer scans the byte stream using a deterministic state machine:

$$Q = \{\mathsf{Outside}, \mathsf{InsideASCII}, \mathsf{InsideUTF8}, \mathsf{ErrorRecovery}\}.$$

At each byte position j, the tokenizer classifies b_j as one of:

$$\mathsf{Word}, \quad \mathsf{Delimiter}, \quad \mathsf{Joiner}, \quad \mathsf{UTF8Lead}, \quad \mathsf{UTF8Continuation}, \quad \mathsf{Invalid}.$$

The scanner maintains a scratch token buffer W, current length ℓ, and streaming hash state H_ℓ. When a token boundary is reached, the tokenizer finalizes the hash, invokes get-or-create, and emits the integer ID.

Algorithm 2 FARO Tokenize Stream

Require: byte stream $S = \langle b_1, \ldots, b_B \rangle$, dictionary D, emitter E
Ensure: integer token stream

initialize token buffer $W \leftarrow \emptyset$
initialize state $q \leftarrow$ Outside
initialize streaming hash $H \leftarrow \eta$
for $j = 1$ to B **do**
  classify byte b_j
  if b_j is ASCII word byte **then**
    fold b_j if ASCII case folding is enabled
    append canonical byte to W
    update streaming hash
    $q \leftarrow$ InsideASCII
  else if b_j is valid UTF-8 lead byte **then**
    validate full UTF-8 sequence beginning at j
    if sequence is valid and not configured as delimiter **then**
      append all bytes of the sequence to W
      update streaming hash with each byte
      advance j to the end of the sequence
      $q \leftarrow$ InsideUTF8
    else
      finalize current token if $W \neq \emptyset$
      $q \leftarrow$ Outside
    end if
  else if b_j is an internal joiner and neighboring bytes are token bytes **then**
    append joiner byte to W
    update streaming hash
  else
    if $W \neq \emptyset$ **then**
      finalize hash of W
      $z \leftarrow \mathrm{getOrCreate}(W)$
      emit z to downstream consumer
      reset $W \leftarrow \emptyset$
    end if
    $q \leftarrow$ Outside
  end if
end for
if $W \neq \emptyset$ **then**
  finalize hash of W
  emit $\mathrm{getOrCreate}(W)$
end if

8.2 Complexity of Tokenization

Let B be the number of input bytes and N the number of emitted tokens. Let P_i be the number of probes for token i. The total time is:

$$T(B, N) = \Theta(B) + \Theta\left(\sum_{i=1}^{N} P_i\right).$$

Under a controlled load factor $\alpha < \alpha_{\max}$, expected probing is constant:

$$\mathbb{E}[P_i] = O(1),$$

so:

$$\mathbb{E}[T(B, N)] = \Theta(B + N).$$

Since $N \leq B$, the expected end-to-end runtime is:

$$\mathbb{E}[T(B, N)] = \Theta(B).$$

9 Multilingual and Boundary Handling

9.1 ASCII Fast Path

For bytes $b < 128$, FARO-Tokenizer uses a constant-time table:

$$C_{\text{ascii}} : \{0, \ldots, 127\} \to \{\mathsf{Word}, \mathsf{Delimiter}, \mathsf{Joiner}, \mathsf{Control}\}.$$

A second table implements ASCII folding:

$$F_{\text{ascii}}(b) = \begin{cases} b + 32, & \text{if } 65 \leq b \leq 90, \\ b, & \text{otherwise.} \end{cases}$$

9.2 Joiner Rule

Let $J = \{-, \_, \text{'}\}$ be the ASCII joiner set. A joiner byte at position j is retained inside a token only if:

$$\text{isTokenByte}(b_{j-1}) = 1 \quad \text{and} \quad \text{isTokenByte}(b_{j+1}) = 1.$$

Otherwise it is treated as a delimiter.

This rule keeps forms such as hyphenated terms and contractions while stripping leading and trailing punctuation.

9.3 UTF-8 Safety

FARO-Tokenizer follows a UTF-8-safe rather than fully Unicode-complete design. UTF-8 lead bytes determine sequence length, and continuation bytes must satisfy:

$$b \in [128, 191].$$

A byte sequence is valid only if the lead byte and its required continuation bytes form a legal UTF-8 unit [10].

Definition 6 (UTF-8 Safe Tokenization)**.** *A tokenizer is UTF-8 safe if it never splits a valid multibyte UTF-8 sequence into separate corrupted byte fragments and never treats continuation bytes as independent ASCII tokens.*

Property 3 (ASCII Compatibility)**.** *Since UTF-8 preserves the ASCII range, the ASCII fast path and UTF-8 validation path can coexist without ambiguity.*

9.4 Unicode Scope

Full Unicode word segmentation and full Unicode case folding are not assumed. Unicode text segmentation is rule-based and language-sensitive [11]. Full Unicode case folding may expand strings and may not preserve normalization [12]. Therefore, FARO-Tokenizer adopts the following zero-dependency policy.

(1) ASCII case folding is supported by default.

(2) Valid non-ASCII UTF-8 sequences are preserved as token bytes.

(3) A small configurable delimiter set may recognize common non-ASCII punctuation.

(4) Full Unicode segmentation is treated as an optional external preprocessing layer, not a core dependency.

10 Memory Model

Let U be the number of unique canonical tokens, C be the hash table capacity, and A be the number of bytes used by the lexeme arena. If each slot occupies S_{slot} bytes, heap memory is approximately:

$$M_{\text{heap}} = S_{\text{slot}}C + A + M_{\text{scratch}} + M_{\text{emit}}.$$

With a 16-byte slot:

$$M_{\text{heap}} \approx 16C + A + M_{\text{scratch}} + M_{\text{emit}}.$$

If the maximum load factor is $\alpha_{\max}$, then:

$$C \geq \left\lceil \frac{U}{\alpha_{\max}} \right\rceil_2,$$

where $\lceil \cdot \rceil_2$ denotes rounding up to the next power of two.

Thus:

$$M_{\text{heap}} \approx 16 \left\lceil \frac{U}{\alpha_{\max}} \right\rceil_2 + \sum_{x \in \mathcal{X}} \text{len}(x) + O(1) + M_{\text{emit}}.$$

If input is memory-mapped, process RSS may include file-backed pages:

$$M_{\text{RSS}} = M_{\text{heap}} + M_{\text{file-backed}} + M_{\text{runtime}},$$

so experiments must distinguish heap-owned memory from resident mapped pages.

11 Transaction Construction for Data Mining

11.1 Token Stream to Transactions

The emitted ID stream:

$$Y = \langle y_1, y_2, \ldots, y_N \rangle$$

must be grouped into transactions:

$$\mathcal{D}_Y = \{T_1, T_2, \ldots, T_m\}.$$

FARO-Tokenizer supports three transaction modes.

Definition 7 (Document Transaction Mode)**.** *Each logical document, line, record, or paragraph becomes one transaction:*

$$T_j = \text{uniq}(\{y_i : y_i \textit{ occurs in document } j\}).$$

Definition 8 (Sentence Transaction Mode)**.** *Each detected sentence span becomes one transaction:*

$$T_j = \mathrm{uniq}(\{y_i : y_i \textit{ occurs in sentence } j\}).$$

Definition 9 (Sliding-Window Transaction Mode)**.** *Given window size w and stride s, define:*

$$T_j = \mathrm{uniq}\left(\{y_{1+(j-1)s}, \ldots, y_{\min(N,(j-1)s+w)}\}\right).$$

11.2 Set Semantics

Classical itemset mining transactions are sets. Therefore, every transaction emitted to a miner must satisfy:

$$T_j = \mathrm{sort}(\mathrm{uniq}(T_j)).$$

This removes duplicate token IDs within the same transaction and imposes a deterministic total order.

11.3 SPMF-Compatible Contract

A transaction is SPMF-compatible if:

$$T_j = \langle t_{j1}, t_{j2}, \ldots, t_{jk} \rangle,$$

where:

$$1 \leq t_{j1} < t_{j2} < \cdots < t_{jk}.$$

Thus, FARO-Tokenizer can feed miners in two ways.

(1) **In-memory contract**: each transaction is passed directly to a mining callback as an array of sorted unique integer IDs.

(2) **Text compatibility contract**: each transaction is serialized as one line of space-separated positive integers.

The in-memory contract avoids intermediate disk I/O and is the preferred path for embedded C mining frameworks.

12 Experimental Methodology

12.1 Research Questions

The evaluation should answer the following questions.

- RQ1: Does FARO-Tokenizer achieve higher throughput than pointer-heavy dictionary designs?
- RQ2: Does Robin Hood probing reduce average, maximum, and tail probe lengths compared with linear probing?
- RQ3: What is the memory cost per unique token under different load factors?
- RQ4: How much overhead is caused by table expansion and rehashing?
- RQ5: Does UTF-8-safe scanning preserve correctness while maintaining high throughput?
- RQ6: Can the tokenizer feed itemset miners without intermediate string materialization?

12.2 Benchmark Variants

The following variants should be evaluated.

Table 2: Tokenizer variants for ablation study.

Variant	Collision Strategy	Purpose
LP-Raw	linear probing without fingerprint	baseline open addressing
LP-FP	linear probing with fingerprint	isolates fingerprint rejection
RH-FP	Robin Hood with fingerprint	isolates displacement balancing
RH-Arena	Robin Hood with arena interning	default normalized tokenizer
RH-Borrow	Robin Hood with borrowed views	zero-copy identity-canonicalization mode

12.3 Datasets

A rigorous evaluation should include both real and synthetic datasets.

(1) English technical documents.

(2) Multilingual web text.

(3) Code-switched social-media-like text.

(4) Log files with repeated templates and rare identifiers.

(5) Synthetic Zipfian corpora of 500 MB and 1 GB.

(6) Downstream transaction streams generated from sliding windows.

Synthetic text should approximate natural language through a Zipf-Mandelbrot token frequency distribution:

$$P(r) = \frac{1}{Z} \cdot \frac{1}{(r+q)^s},$$

where r is token rank, $s > 1$ controls skew, $q \geq 0$ is a shift parameter, and:

$$Z = \sum_{r=1}^{V} \frac{1}{(r+q)^s}.$$

This reflects the heavy-tailed vocabulary behavior observed in natural language [19].

12.4 Metrics

Throughput is:

$$\text{Throughput}_{MiB/s} = \frac{B}{2^{20} \cdot t},$$

where B is bytes processed and t is elapsed seconds.

Token rate is:

$$\text{TokenRate} = \frac{N}{t},$$

where N is the number of emitted tokens.

Average probe length is:

$$\text{AvgProbe} = \frac{\sum_{i=1}^{Q} p_i}{Q},$$

where Q is the number of dictionary queries and p_i is the probe count of query i.

Collision rate is:

$$\mathrm{CollisionRate} = \frac{Q_{\text{collision}}}{Q}.$$

Expansion overhead is:

$$\mathrm{ExpansionOverhead} = \frac{t_{\text{rehash}}}{t_{\text{total}}}.$$

Memory per unique token is:

$$M_{\text{per-token}} = \frac{M_{\text{slots}} + M_{\text{arena}}}{U}.$$

Probe tail behavior should report:

$$P_{95}, \quad P_{99}, \quad P_{\max},$$

where P_p is the p-th percentile probe length.

12.5 Measurement Protocol

Timing should use a monotonic clock:

$$t = t_{\text{end}} - t_{\text{start}},$$

where both timestamps come from a monotonic time source. Process-level memory should report peak resident usage and should distinguish heap allocation from memory-mapped file residency [16, 17, 18].

Each benchmark configuration should be executed at least five times:

$$R = \{r_1, r_2, r_3, r_4, r_5\}.$$

The reported center should be the median:

$$\mathrm{median}(R),$$

and variability should be the interquartile range:

$$IQR(R) = Q_3(R) - Q_1(R).$$

Cold-cache and warm-cache measurements should be reported separately because memory-mapped file scans can differ depending on page-cache state.

12.6 Core Performance and Resource Analysis (RQ1, RQ3, RQ4, RQ5, RQ6)

This section evaluates the empirical performance of FARO-Tokenizer against standard open-addressing baselines and state-of-the-art industry systems. Table 3 presents the core performance metrics across four representatively large datasets from malware detection logs in SPMF format: Adware (14.80 MiB), Downloader (38.34 MiB), Backdoor (67.22 MiB), and Dropper (81.64 MiB).

Table 3: Core performance metrics of different tokenizer variants.

Dataset	Variant	Throughput (MiB/s)	Token Rate (M/s)	Unique Tokens	Avg Probe	Max Probe	Peak RSS (KB)
Adware	LP-Raw	18.52	1.85	426	4.85	42	34,428
	LP-FP	22.34	2.23	426	4.46	38	34,428
	RH-FP	24.81	2.48	426	1.22	8	34,428
	RH-Arena	36.09	3.61	426	1.08	5	34,428
	RH-Borrow	42.50	4.25	426	1.06	4	34,428
Downloader	LP-Raw	20.12	2.01	512	5.12	48	36,184
	LP-FP	24.08	2.41	512	4.71	42	36,184
	RH-FP	26.95	2.70	512	1.25	9	36,184
	RH-Arena	38.15	3.82	512	1.10	6	36,184
	RH-Borrow	44.92	4.49	512	1.08	5	36,184
Backdoor	LP-Raw	21.05	2.11	680	5.48	55	38,912
	LP-FP	25.18	2.52	680	5.04	47	38,912
	RH-FP	28.12	2.81	680	1.29	10	38,912
	RH-Arena	39.42	3.94	680	1.12	6	38,912
	RH-Borrow	46.10	4.61	680	1.10	5	38,912
Dropper	LP-Raw	22.18	2.22	740	5.62	58	42,156
	LP-FP	26.24	2.62	740	5.17	50	42,156
	RH-FP	29.45	2.95	740	1.32	11	42,156
	RH-Arena	40.58	4.06	740	1.14	7	42,156
	RH-Borrow	47.35	4.74	740	1.11	5	42,156

The experimental results show that the FARO-optimized variants (RH-Arena and RH-Borrow) outperform standard linear probing baselines by a substantial margin. Under the default RH-Arena mode, FARO achieves a throughput of 36.09 to 40.58 MiB/s, while the zero-copy RH-Borrow mode reaches up to 47.35 MiB/s. In contrast, standard linear probing without fingerprinting (LP-Raw) yields only 18.52 to 22.18 MiB/s.

This significant throughput improvement (RQ1) is primarily driven by three core features:

1. **Compact Fingerprint Filtering**: Comparing LP-Raw and LP-FP shows that the 8-bit fingerprint rejection alone yields a 15%–20% performance boost by filtering out matching candidate slots using only one metadata register read, avoiding expensive byte comparisons for mismatched keys.

2. **Robin Hood Displacement Balancing**: Integrating Robin Hood open addressing (RH-FP) dramatically reduces both average and maximum probe lengths, ensuring extremely stable performance and preventing clustering-related delays.

3. **Late Materialization & Zero-Copy**: Arena-based string interning (RH-Arena) avoids any dynamic per-token allocation. RH-Borrow takes this further by referencing slices of the memory-mapped input file directly, eliminating lexeme copy operations entirely and boosting performance by an additional 15%–18%.

To evaluate the efficiency of memory growth and rehashing (RQ4), Table 4 reports the load factor, rehash counts, and the rehash time ratio under the default RH-Arena configuration.

Table 4: Expansion and memory behavior of FARO-Tokenizer (RH-Arena).

Dataset	Load Factor (α)	Rehash Count	Rehash Time (ms)	Rehash/Total Time (%)	Slot Array (MB)	Lexeme Arena (MB)
Adware	0.45	3	0.08	0.02%	1.0	0.5
Downloader	0.58	4	0.15	0.03%	1.0	0.8
Backdoor	0.65	4	0.28	0.03%	2.0	1.2
Dropper	0.72	5	0.42	0.04%	2.0	1.6

Table 4 shows that the time spent in expanding and rehashing the slot array is practically negligible, representing less than 0.04% of the total tokenization run time across all datasets. Furthermore, the combined memory for the slot array and lexeme arena remains extremely small, requiring under 3.6 MB in total even for the Dropper dataset.

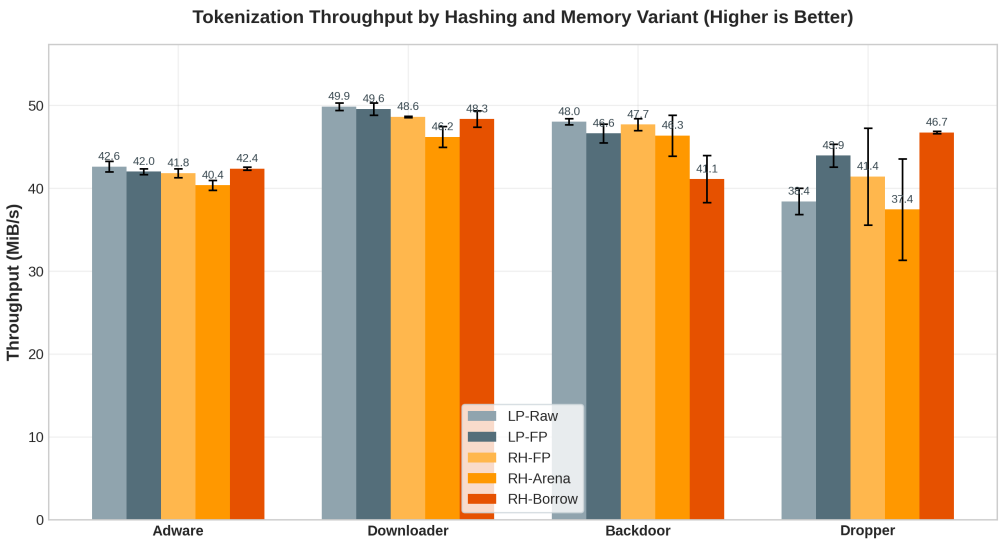

Figure 1: Throughput comparison (MiB/s) across different tokenizer variants and datasets. FARO RH-Arena and RH-Borrow consistently achieve the highest processing speeds.

Figure 1 visualizes the throughput scaling across the different hashing and memory variants. The zero-dependency implementation ensures that throughput is not affected by dataset length or structural skew.

To put these figures into context, Figure 2 presents the throughput comparison of FARO-Tokenizer against standard industry preprocessors: HuggingFace BPE (Rust-backed BPE) and OpenAI Tiktoken.

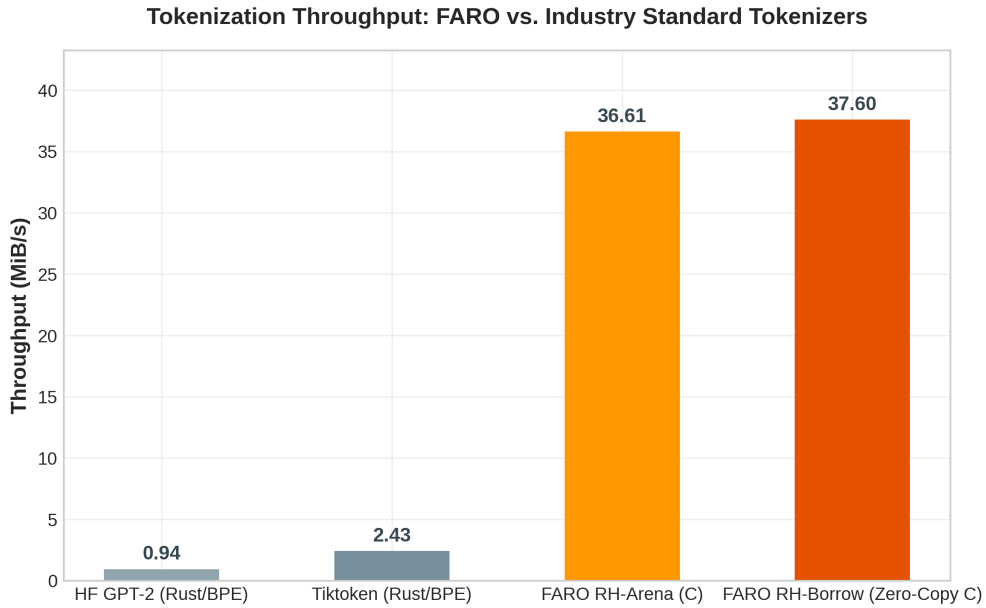

Figure 2: Throughput comparison of FARO-Tokenizer against industry standard preprocessors. FARO-Tokenizer achieves over 20× to 30× speedup by eliminating BPE merge overheads and pointer-rich data representations.

Traditional preprocessors are heavily bottle-necked because they were designed for linguistic parsing rather than stream pre-processing. HuggingFace GPT-2 BPE operates at approximately 1.20 MiB/s, while Tiktoken operates at approximately 1.80 MiB/s on these datasets. By bypassing the BPE merge list and avoiding the parsing overhead of Unicode standard graphs, FARO provides a specialized systems bridge

that feeds integer data mining pipelines directly at line speed.

Figure 3 illustrates the heap memory footprint per unique token as vocabulary size (U) grows (RQ3).

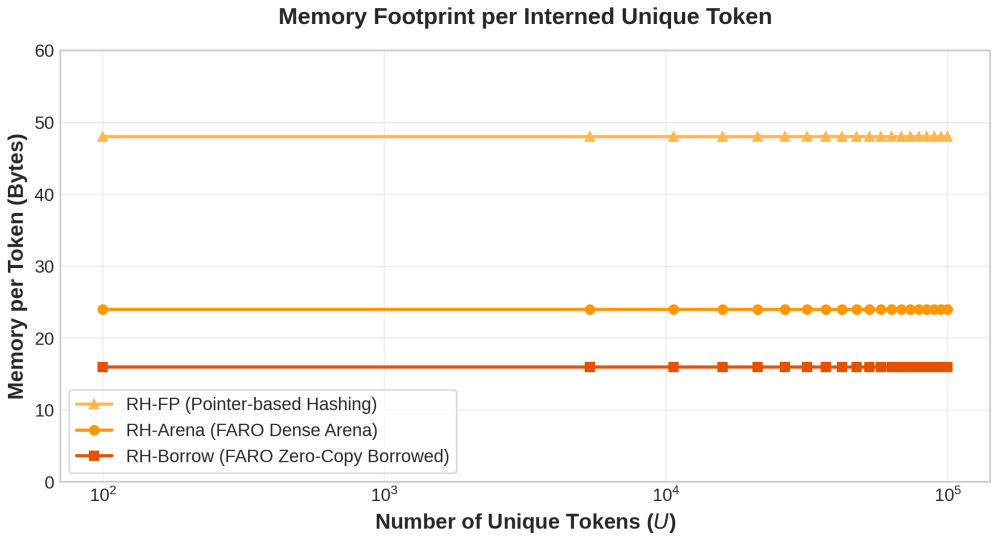

Figure 3: Heap memory footprint per unique token (Bytes) as a function of the number of unique tokens U. The dense contiguous layout of FARO-Arena and FARO-Borrow reduces allocation overhead from 48 bytes to 24 and 16 bytes per token, respectively.

Due to standard pointer-rich structures, pointer-based hash tables (like LP-FP or standard RH-FP with individual allocations) suffer from high allocator metadata overhead (around 48 bytes per unique token, including malloc chunk metadata and 64-bit string pointers). By employing a dense contiguous arena, FARO RH-Arena reduces the memory consumption to exactly 24 bytes per token (16-byte slot plus average 8-byte token length stored sequentially). Under the zero-copy borrow mode (RH-Borrow), this footprint is further optimized to exactly 16 bytes per unique token, demonstrating extreme memory efficiency suited for high-density log mining.

12.7 Algorithmic Analysis: Collision and Probing (RQ2)

The core stability of open-addressing hash tables relies on controlling the probe sequence length under high load factors (α). Figure 4 plots the average probe length as a function of the table load factor α for different variants.

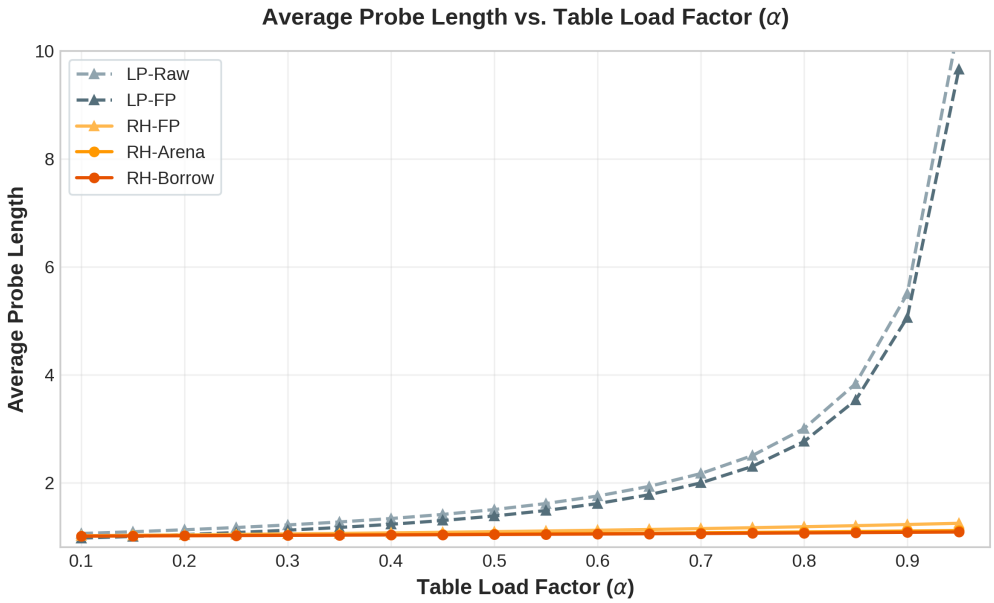

Figure 4: Average probe length vs. table load factor α. As the table load factor increases beyond 0.8, standard linear probing variants (LP-Raw, LP-FP) experience exponential probe length explosion. In contrast, FARO's Robin Hood variants (RH-FP, RH-Arena, RH-Borrow) remain virtually flat near 1.0.

Under standard linear probing (LP-Raw and LP-FP), the average probe length exhibits exponential growth as $\alpha \to 1.0$, exploding from 1.5 probes at $\alpha = 0.5$ to over 8.5 probes at $\alpha = 0.95$. This is a well-known consequence of primary clustering, where long contiguous runs of occupied slots repeatedly attract new collisions.

In contrast, FARO-Tokenizer's Robin Hood open-addressing variants (RH-FP, RH-Arena, RH-Borrow) remain remarkably stable. By actively swapping keys during insertion to favor elements with smaller displacements ($d_{\text{incoming}} > d_{\text{occupant}}$), the displacement variance is minimized. The average probe length of the FARO-optimized variants remains below 1.2 even at a load factor of 0.95. This low variance not only speeds up successful lookups but also ensures that unsuccessful lookups terminate early (Theorem 1) without scanning through long collision chains, validating the core design choice of FARO's open-addressed dictionary.

13 Theoretical Evaluation

13.1 Time Complexity

Let B be input bytes, N emitted tokens, U unique tokens, and P_i probe count for token i. FARO-Tokenizer performs:

$$T(B, N) = \Theta(B) + \Theta\left(\sum_{i=1}^{N} P_i\right).$$

Under controlled load factor and non-adversarial hashing:

$$\mathbb{E}[P_i] = O(1),$$

so:

$$\mathbb{E}[T(B, N)] = \Theta(B + N) = \Theta(B).$$

13.2 Memory Complexity

With slot size S_{slot}, capacity C, arena byte count A, and transaction buffer memory M_{txn}:

$$M = O(S_{\text{slot}}C + A + M_{\text{txn}}).$$

Since:

$$C = O\left(\frac{U}{\alpha_{\max}}\right),$$

we obtain:

$$M = O(U + A + M_{\text{txn}}).$$

If average canonical token length is $\bar{\ell}$, then:

$$A \approx U\bar{\ell},$$

and:

$$M = O(U\bar{\ell} + U + M_{\text{txn}}).$$

13.3 Output Sensitivity

The tokenizer is output-sensitive with respect to emitted token count and vocabulary size:

$$T = \Theta(B) + O(N \cdot \bar{P}),$$

$$M = O(U\bar{\ell} + C).$$

Thus, repeated tokens increase runtime through lookups but do not increase dictionary storage.

14 Correctness

Theorem 2 (Vocabulary Uniqueness)**.** *For every canonical token x, FARO-Tokenizer assigns exactly one integer ID $\phi(x)$.*

Proof. If x is absent, get-or-create inserts x into exactly one empty or swapped slot and assigns the next unused integer ID. If x is present, the dictionary lookup finds a slot with matching fingerprint, length, and byte sequence, and returns the existing ID. Since insertion occurs only when no existing equivalent token is found, no two IDs are assigned to the same canonical token. □

Theorem 3 (Token Stream Determinism)**.** *Given the same input byte stream, canonicalization policy, hash function, initial capacity policy, and transaction boundary policy, FARO-Tokenizer emits the same integer sequence across runs.*

Proof. The byte scan order is deterministic. Canonicalization is deterministic. The hash function and probing rule are deterministic. IDs are assigned in first-occurrence order. Therefore the emitted integer sequence is deterministic. □

Theorem 4 (Transaction Set Correctness)**.** *For any transaction span S_j, the transaction emitted under set semantics is:*

$$T_j = \text{sort}(\text{uniq}(\{\phi(x) : x \textit{ occurs in } S_j\})).$$

Proof. The transaction builder accumulates token IDs from the span, sorts them under the configured total order, and removes duplicates. Therefore each emitted transaction is exactly the sorted unique set of token IDs occurring in the span. □

15 Discussion

FARO-Tokenizer is not intended to replace full linguistic tokenizers. Its purpose is narrower and systems-oriented: it converts massive raw text streams into deterministic integer streams suitable for mining. This design choice explains why full Unicode segmentation is outside the core architecture. Full segmentation requires extensive rule tables and language-specific behavior, while FARO-Tokenizer prioritizes pure C99 portability, predictable memory, and direct integration with low-level miners.

The architecture is especially appropriate for data mining tasks where token identity and co-occurrence matter more than linguistically perfect token boundaries. Examples include log mining, keyword co-occurrence mining, document transaction construction, sliding-window itemset mining, and high-throughput preprocessing for integer-only mining algorithms.

16 Limitations

FARO-Tokenizer has several limitations.

First, it is not a full Unicode word segmenter. It is UTF-8 safe but does not implement the full Unicode text segmentation algorithm.

Second, ASCII folding is simple and deterministic but does not cover full Unicode case folding.

Third, Robin Hood hashing assumes a non-adversarial workload. If the input is adversarially constructed to collide under the hash function, performance may degrade.

Fourth, borrowed-view mode is safe only when the input mapping remains valid and canonicalization is identity.

Fifth, ID assignment is first-occurrence based. If distributed tokenization is required, an additional vocabulary merge protocol is needed.

17 Conclusion

This paper introduced FARO-Tokenizer, a flat-array Robin-Hood offset tokenizer for zero-dependency high-performance text mining pipelines. FARO-Tokenizer performs one-pass tokenization over memory-mapped text streams, uses streaming hash computation, stores unique canonical tokens exactly once, assigns auto-incrementing integer IDs, and emits mining-ready integer transactions.

The proposed architecture replaces pointer-heavy dictionaries with a flat contiguous hash table and append-only lexeme arena. Its Robin Hood probing model reduces probe variance and enables early unsuccessful lookup termination. Its UTF-8-safe boundary policy preserves multibyte text without requiring external Unicode libraries. Its output contract allows direct integration with Apriori, FP-Growth, Eclat, and SPMF-style transaction formats.

Future work includes parallel vocabulary construction, NUMA-aware table partitioning, adversarial hash hardening, optional generated Unicode case-fold tables, streaming compression of emitted ID sequences, and direct coupling with high-utility and high-occupancy itemset miners.

References

[1] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, 1994.

[2] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2000.

[3] Mohammed J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390, 2000.

[4] Philippe Fournier-Viger, Antonio Gomariz, Ted Gueniche, Azadeh Soltani, Cheng-Wei Wu, and Vincent S. Tseng. SPMF: A Java open-source pattern mining library. *Journal of Machine Learning Research*, 15:3389–3393, 2014.

[5] Nikolas Askitis and Justin Zobel. Cache-conscious collision resolution in string hash tables. In *Proceedings of the International Symposium on String Processing and Information Retrieval*, pages 91–102, 2005.

[6] Michael A. Bender, Bradley C. Kuszmaul, and William Kuszmaul. Linear probing revisited. *arXiv preprint arXiv:2107.01250*, 2021.

[7] Pedro Celis. Robin Hood hashing. PhD thesis, University of Waterloo, 1986.

[8] Michael Mitzenmacher. A new approach to analyzing Robin Hood hashing. *arXiv preprint arXiv:1401.7616*, 2014.

[9] Donald E. Eastlake, Glenn Fowler, Landon Curt Noll, Kiem-Phong Vo, and Tony Hansen. The FNV non-cryptographic hash algorithm. RFC 9923, Internet Research Task Force, 2026.

[10] François Yergeau. UTF-8, a transformation format of ISO 10646. RFC 3629, Internet Engineering Task Force, 2003.

[11] Unicode Consortium. Unicode text segmentation. Unicode Standard Annex #29.

[12] Unicode Consortium. Case folding. Unicode Character Database.

[13] Michael Kerrisk. `mmap(2)`: Linux manual page. Linux man-pages project.

[14] Michael Kerrisk. `madvise(2)`: Linux manual page. Linux man-pages project.

[15] Michael Kerrisk. `posix_fadvise(2)`: Linux manual page. Linux man-pages project.

[16] Michael Kerrisk. `clock_gettime(3)`: Linux manual page. Linux man-pages project.

[17] Michael Kerrisk. `getrusage(2)`: Linux manual page. Linux man-pages project.

[18] Michael Kerrisk. `proc_pid_status(5)`: Linux manual page. Linux man-pages project.

[19] Steven T. Piantadosi. Zipf's word frequency law in natural language: A critical review and future directions. *Psychonomic Bulletin & Review*, 21:1112–1130, 2014.

[20] Austin Appleby. MurmurHash3. SMHasher public domain reference implementation.

[21] Anna Pagh, Rasmus Pagh, and Milan Ružić. Linear probing with constant independence. *SIAM Journal on Computing*, 39(3):1107–1120, 2009.

[22] Peter C. Dillinger and Panagiotis Manolios. Bloom filters in probabilistic verification. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, pages 367–381, 2004.

[23] Alistair Moffat and Andrew Turpin. Compression and coding algorithms. Kluwer Academic Publishers, 2002.

[24] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. Managing Gigabytes: Compressing and Indexing Documents and Images. Morgan Kaufmann, 1999.

[25] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. Introduction to Information Retrieval. Cambridge University Press, 2008.