An Evaluation of N-Gram Selection Strategies for Regular Expression Indexing in Contemporary Text Analysis Tasks

Ling Zhang
University of Wisconsin-Madison
Madison, WI, United States
ling-zhang@cs.wisc.edu

Shaleen Deep
Microsoft Jim Gray Systems Lab
Madison, WI, United States
shaleen.deep@microsoft.com

Jignesh M. Patel
Carnegie Mellon University
Pittsburgh, PA, United States
jignesh@cmu.edu

Karthikeyan Sankaralingam
University of Wisconsin-Madison
Madison, WI, United States
karu@cs.wisc.edu

ABSTRACT

Efficient evaluation of regular expressions (regex, for short) is crucial for text analysis, and n-gram indexes are fundamental to achieving fast regex evaluation performance. However, these indexes face scalability challenges because of the exponential number of possible n-grams that must be indexed. Many existing selection strategies, developed decades ago, have not been rigorously evaluated on contemporary large-scale workloads and lack comprehensive performance comparisons. Therefore, a unified and comprehensive evaluation framework is necessary to compare these methods under the same experimental settings. This paper presents the first systematic evaluation of three representative n-gram selection strategies across five workloads, including real-time production logs and genomic sequence analysis. We examine their trade-offs in terms of index construction time, storage overhead, false positive rates, and end-to-end query performance. Through empirical results, this study provides a modern perspective on existing n-gram based regular expression evaluation methods, extensive observations, valuable discoveries, and an adaptable testing framework to guide future research in this domain. We make our implementations of these methods and our test framework available as open-source at https://github.com/mush-zhang/RegexIndexComparison.

PVLDB Reference Format:

Ling Zhang, Shaleen Deep, Jignesh M. Patel, and Karthikeyan Sankaralingam. An Evaluation of N-Gram Selection Strategies for Regular Expression Indexing in Contemporary Text Analysis Tasks . PVLDB, 18(13): 5703 - 5715, 2025.
doi:10.14778/3773731.3773744

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at https://github.com/mush-zhang/RegexIndexComparison.

Proceedings of the VLDB Endowment, Vol. 18, No. 13 ISSN 2150-8097.
doi:10.14778/3773731.3773744

1 INTRODUCTION

Regular expressions are a foundational tool for text pattern matching, powering critical applications such as real-time log analysis [32], genomic sequence alignment [2], and web information retrieval [13]. However, as datasets grow in their size, the computational cost of brute-force matching becomes prohibitive. To address this issue, n-gram indexing has been widely adopted to accelerate regex processing by pre-filtering candidate text regions using selected n-grams [12, 28]. Despite its widespread adoption, the scalability of this approach hinges on a critical problem: how to select the optimal set of n-grams to index, balancing trade-offs between index size, construction time, and query accuracy.

Legacy methods like frequency-based selection [6], coverage-optimized strategies [17], and heuristics like `LPMS` [35] were introduced decades ago to reduce index size and memory usage [6, 17]. Despite being an active area of research, surprisingly, the performance of these methods on modern hardware is lacking for three reasons. The absence of a comprehensive, unbiased, and systematic understanding of these methods is due to three main reasons.: (1) inconsistent experimental setups hinder fair comparisons, (2) evaluations are narrow in scope and datasets, and (3) a lack of standardized testing framework hindering practical assessment. As a result, recent studies [7, 31] continue to adopt these outdated approaches without re-evaluating their assumptions. These methods struggle with today's large datasets (e.g., genomics, IoT, logs), where selection strategies break down. For instance, `BEST`'s quadratic runtime becomes impractical on even 10GB of text, often forcing downsampling or skipping indexing. Modern systems like Postgres [8] and GitLab [29] bypass selection entirely, indexing all trigrams or fixed n-grams. These limitations motivate our work.

M.1 Lack of Comprehensive Comparison. Prior evaluations focus on specific metrics (e.g., false-positive rate) or evaluate their method using only a single workload, failing to capture the diverse demands of real-world workloads. No study has systematically compared and contrasted the three key methods `FREE`, `BEST`, and `LPMS` across wide-ranging scenarios.

M.2 Outdated Resource Assumptions. Existing methods often optimize assuming limited memory, investing significant efforts to reduce memory overhead and index sizes. Additionally, some approaches focus on scenarios where the index and/or dataset cannot fit in memory and use the IO cost as an optimization parameter [20].

However, in many modern hardware settings, abundant memory is often available, and it is also important to consider this case.

M.3 Fine-grained empirical analysis. Beyond evaluating the existing methods on a comprehensive set of common metrics, we also conduct detailed resource usage measurements for each method, providing deeper performance related insights.

Our Contributions. We systematically study three state-of-the-art methods, FREE, BEST, and LPMS, across various workloads. Specifically, we make the following contributions.

1. Systematic Method Analysis. We formalize and categorize three state-of-the-art n-gram selection strategies, FREE (frequency-based), BEST (coverage-optimized), and LPMS (linear programming approximation), within a unified framework. This enables direct theoretical comparison of their computational complexities. Additionally, we provide a detailed account of their implementation and design decisions, emphasizing both similarities and subtle differences with other methods, addressing **(M.1)**

2. Methods and Benchmark Framework Implementation. In the absence of original code, we implemented FREE, BEST, and LPMS as described in the respective papers with modern C++. While BEST's original design includes parallelism for multi-core CPUs, FREE and LPMS lack scalable implementations. We modernized all three methods by: (1) redesigning FREE with parallelism to leverage modern multi-core CPUs, and (2) implementing LPMS with Gurobi, which inherently supports multi-threading in its LP solver. Our benchmarking framework is modular, easily extensible, and publicly available at the link in the abstract. All experiments were conducted on a multi-core machine with large memory, addressing **(M.2)**.

3. Broad Workload Benchmark. To address the lack of comprehensive empirical comparisons on workloads with different characteristics, we evaluate the methods across eight workloads. This includes using three legacy benchmarks from prior work and two real-world workloads in contemporary regex matching tasks.

4. Empirical Trade-off Analysis and Guidelines. Our experiments quantify performance across various workloads by measuring index construction time, memory footprint, and index precision, among other metrics (addressing **(M.3)**). We correlate these results with workload characteristics, such as regex complexity and dataset size, to derive actionable guidelines. For instance, FREE achieves a 92% reduction in index build time compared to BEST with only a 1.2% increase in query latency, making it more suitable for streaming log analysis. These findings challenge the necessity of computationally intensive methods for large-scale workloads.

2 RELATED WORK

Efficient indexing of regular expressions is crucial to enhance query performance in large-scale database systems, log analysis, and information extraction applications. Traditionally, regular expressions are evaluated using finite automata, which is computationally expensive due to backtracking and state transitions. Since regex queries operate on string datasets, n-gram-based indexing techniques have been widely adopted to accelerate regex evaluation by pre-filtering candidate matches before full regex matching. N-gram selection techniques are critical for regex indexing, as they impact both query filtering accuracy and index storage overhead.

To maximize regex evaluation performance before the introduction of n-gram indexing techniques or when such indexing is not feasible, early research focused on string literal filtering to optimize regex performance. A suffix-based string matching heuristic was presented in [3], laying the foundation for later suffix-tree-based regex indexing methods. Suffix-based indexing was then proposed for efficient pattern matching over large datasets [16]. Literal filtering, or prefix heuristics for approximate string matching, was introduced in [9, 38]. These studies demonstrated that prefix-based early termination can significantly reduce regex search time. Later works proposed filtering with all discriminative string literals in addition to prefixes and suffixes in the regexes [37, 41].

N-gram indexing was introduced to improve regex query performance by precomputing substrings of length n as index keys. Besides pattern matching, n-gram indexing is also common in approximate string searching. Early works compared different n values and decided to index all trigrams. We refer to this method as Trigrams, and will use it for comparison in Section 6. Other works index n-grams with different n values for various workloads. The database community has conducted extensive research on effective indexing strategies for regular expression queries. While existing works primarily focus on improving regex query performance through optimized index structures, relatively fewer studies have explored n-gram key selection strategies for regex indexing. N-gram indexing has also been used for the closely related problem of indexing regular expressions (instead of data) to find out which regexes match a given input string [4]. Similar indexing ideas have also been used to speed up regular path query evaluation [22].

The first known work discussing n-gram selection for regex indexing is FREE [6]. The high-level idea of selecting a covering n-gram set was proposed in a study of Asian language indexing [28]. The concept of a covering set was further developed by Kim et al. [20] for a more efficient n-gram selection method considering I/O cost. The analogy between the set-covering problem and the n-gram selection problem was formalized in BEST [17], where the authors presented a near-optimal algorithm to select variable-length n-grams considering index size constraints. Later work, LPMS [35], combined the solution formulations of FREE and BEST and reformulated the n-gram selection problem into linear programs. In this work, we compare the n-gram selection and indexing methods of FREE, BEST, and LPMS for in-memory workloads and indices. Several works have explored ML-based approaches for regex evaluation in security contexts. On resource-constrained edge devices, compressed on-device ML models are commonly used for high-throughput malicious packet detection. However, these methods rely on extensive preprocessing and assume access to high-quality training data, which doesn't hold for log analytics. For a broader overview, see the survey by Xu et al. [39].

3 PROBLEM DEFINITION

In this section, we discuss the n-gram selection problem and provide formal definitions for some common terms.

Definition 3.1 (N-Gram). For a given finite alphabet Σ, an n-gram g is a sequence of n characters $g = g_1 g_2 \cdots g_n$ where $g_i \in \Sigma$.

We use $|g|$ to denote the length of the n-gram g. A literal is a string from the set $\Sigma*$, where $*$ is the standard closure operator. A common operation on a regex query q is to identify the maximal literal components from the regex. For a literal l, we will use $G(l)$ to denote the set of all possible substrings of l. The set of all possible n-grams of a regex q is $G(q) = \bigcup_{\text{maximal } l \in q} G(l)$. We use G to define the set of all candidate n-grams considered in an n-gram selection method. Note that G may be different for different methods, and we will further discuss it in Section 4. G_i to is used to denote subset of candidate n-grams that has length exactly i.

Example 3.2. Regex queries often times look for patterns as a sequence of characters. The regex `<a href=("|').*ZZZ\.pdf ("|')>` from a workload of regex queries over webpages matches a URL pattern. The set of literal components in the example regex are: { `<a href=` , `ZZZ.pdf` , `>` }.

A workload $W = (Q, D)$ consists of the set of regex queries $Q = \{q_1, q_2, \cdots\}$ and dataset $D = \{d_1, d_2, \cdots\}, d_i \in \Sigma*$. The size of the dataset is defined as $|D| = \sum_{d_i \in D} |d_i|$. Each d_i will be referred to as a data record. Next, we define the support of an n-gram.

Definition 3.3 (Support). The support s of a literal g in dataset D is the number of elements in D that contains g. Similarly, its support in query set Q is the number of individual queries which contains g as a literal.

$$s_D(g) = \sum_{d \in D} \mathbb{1}\,[g \in G(d)] \qquad s_Q(g) = \sum_{q \in D} \mathbb{1}\,[g \in G(q)]$$

Using the notion of support, define an n-gram g's selectivity.

Definition 3.4 (Selectivity). The selectivity c of an n-gram g in string set D is the fraction of individual strings in D that contains g, i.e., $c_g = s_D(g)/|D|$.

For n-gram selection methods, selectivity is a crucial parameter in determining whether an n-gram should be included in the index. Consider a regex pattern designed to match URLs of PDF files whose filenames end with the substring `ZZZ`. By indexing the n-gram `ZZZ.pdf`, we can significantly reduce the number of candidate strings that need to be evaluated during exact regex matching. This is because such a specific n-gram is likely to occur infrequently in the dataset, making it highly selective. In contrast, other literals in the regex, such as `<a href=` and `>`, are extremely common in HTML documents. Indexing these low-selectivity n-grams would not meaningfully reduce the candidate set and could instead introduce unnecessary computational overhead. Therefore, prioritizing high-selectivity n-grams like `ZZZ.pdf` can lead to more efficient query processing by minimizing the number of strings that require full regex evaluation.

However, lower selectivity is not always better when considering the entire query set Q. For example, the n-gram `ZZZ.pdf` may have low selectivity and effectively filter out irrelevant data points. Since `ZZZ` is an uncommon character sequence in the English language, it might not appear in other queries. On the other hand, indexing `.pdf`, which has relatively higher selectivity than `ZZZ.pdf`, might not filter as many data records, but it can benefit other queries that look for URLs of PDF files with different names. Thus, there is a trade-off between selecting higher and lower selectivity n-grams.

4 METHODS OVERVIEW

In this section, we present the overview of the three state-of-the-art n-gram selection methods: FREE, BEST, and LPMS. We will describe their selection strategy and provide a complexity analysis. In Table 1, we summarize and compare the three methods in terms of their source of n-gram, selection criteria in each step, index structure accompanied, and other common configurations.

4.1 FREE

4.1.1 Selection Strategy. FREE uses the dataset in the workload as the source of n-gram selection, and selects a prefix-free set of n-grams based on selectivity. Note that FREE does not use the query workload for n-gram selection. The candidate n-gram set G is $G(D) = \bigcup_{d \in D} G(d)$. FREE decides whether an n-gram will be selected by the *Usefulness* criteria as defined below.

Definition 4.1 (FREE Usefulness). By setting a fixed *selectivity threshold* **c**, n-gram g is deemed *useful* if its selectivity is less than the *selectivity threshold*, $c_g < \mathbf{c}$.

FREE makes two important assumptions for n-grams selection.

Assumption FREE-1. FREE assumes that n-grams with high selectivity are less *useful*.

Assumption FREE-2. With a careful selected *selectivity threshold* **c**, FREE assumes that a query without any *useful* n-grams is rare. Overall workload performance will still improve as most of the regexes can benefit from indexing only *useful* n-grams.

FREE selects only n-grams that are *useful*. Since *usefulness* selection criteria is essentially a selectivity upper bound, if we know that an n-gram g is *useful*, then any longer n-gram having g as a substring is also useful. For example, if n-gram `pdf` is *useful* with a selectivity c_g, then n-grams such as $g' =$ `.pdf` and $g'' =$ `pdfg` will have selectivity $c(g') < \mathbf{c}$ and $c(g'') < \mathbf{c}$, therefore also *useful*.

Property FREE: Usefulness. For a *useful* n-gram $g = g_1 g_2 \cdots g_{n_1}$ of size n_1 having selectivity c_g, any other longer n-grams of size n_2

$$g' = p_1 \cdots p_m g_1 g_2 \cdots g_{n_1} s_1 \cdots s_{n_2 - n_1 - m}$$

with a prefix size m and suffix size $n_2 - n_1 - m$, where each character $p_i, s_i \in \Sigma$, we have $0 \leq c_{g'} \leq c_g$, is also *useful*.

Since the set of all *useful* n-grams can still be too large, to further reduce the index size, FREE selects only the minimal n-gram among all n-grams with the same prefix. For example, n-grams `pdf` and `pdfg` have the same prefixes `p`, `pd`, and `pdf`. If the selectivity of the first two n-grams are not *useful* while `pdf` is *useful*, we index only n-gram `pdf`, although `pdfg` is also useful. Thus, the second n-gram selection criterion besides selectivity is *minimality*.

Definition 4.2 (FREE Minimal). An *useful* n-gram g in alphabet Σ of size n is *prefix-minimal* among the set of n-grams that has g as a prefix if no prefix substring of g with size smaller than n is *useful*. An n-gram g is *suffix minimal* if no suffix substring of g with size smaller than n is *useful*. N-gram g is *prefix-suffix minimal*, or *pre-suf minimal* if no substring of g with size smaller than n is *useful*.

To effectively reduce the size of n-grams selected, FREE selects the prefix-minimal useful set of n-grams, deriving from the Apriori

Table 1: Selection methods summary.

Method	Source	Selection Criteria	Selectivity Threshold (c)	N-Gram Constraint	Time Complexity	Space Complexity
FREE [6]	D	Prefix-free Selectivity	0.1	$[2, 10]$	$O(k \cdot \lvert D\rvert)$	$O(\lvert D\rvert)$
BEST [17]	Q, D	Utility	$[0.05, 0.1]$	-	$O(\lvert D\rvert + \lvert Q\rvert)$	$O(\lvert I\rvert \cdot \lvert G(Q)\rvert \cdot \lvert D\rvert \cdot \lvert Q\rvert)$
LPMS [35]	Q, D	Prefix-free Utility optimized	-	-	$O(\sum_{i \in [10]} \{\lvert G_i(Q)\rvert^{2.5} + \lvert G_i(Q)\rvert \cdot (\lvert D\rvert + \lvert Q\rvert)\})$	$O(\max_{i \in [10]} \lvert G_i(Q)\rvert \cdot (\lvert D\rvert + \lvert Q\rvert)$
VGGraph [30]	Q, D	Combines FREE and BEST	-	$[2, \inf)$	$O(n \cdot \lvert D\rvert + \lvert Q\rvert \cdot \lvert G(Q)\rvert \cdot n)$	$O(\lvert D\rvert + \lvert G(Q)\rvert)$

method of finding the maximal frequent sets in data mining literature [1]. Essentially, it generates all candidate n-grams in increasing size order iteratively. For iteration i, it generates the candidate set of n-grams of length i by extending all useless n-grams of length $i-1$ by one character, inserting the useful n-grams in the candidate set into the index, and using the useless ones in the candidate set for iteration $i+1$. This way, it is not necessary to generate all possible n-grams in each iteration. This method also ensures that the set of n-grams in the index is a prefix-minimal set, as the breadth-first search ensures that the shortest prefix of a useful n-gram is visited first. No n-grams in candidate sets of future iterations have the selected n-grams as prefixes, as the useful n-grams are never extended to generate a candidate n-gram.

FREE confines the search space by constraining the length n of the selected n-grams. Briefly, there are two parameters to tune: 1) selectivity threshold c that distinguish useful n-grams from useless n-grams, and 2) n-gram size n that controls the length of the index keys. In the original paper, the authors use $c = 0.1$ and $2 \leq n \leq 10$ in their experiments. It also defined a simple query plan considering the index keys. We include a description of the query plan generator in Appendix A [42] of the full paper and implemented FREE with the query plan and an accompanied matcher in the codebase. However, for a fair comparison of the n-gram selection methods, we do not use it for the experiments. The index data structure used is inverted index with n-grams as keys and posting lists as values.

4.1.2 Complexity Analysis. Let M_i be the candidate set at step i. First, FREE scans the dataset to identify all unique unigrams and compute their selectivities, using a temporary hashmap of size $|M_1| = |\Sigma|$. This base step has time complexity $O(|D| + |\Sigma|)$. The useful unigrams are extended to form bigrams M_2, with space and compute overheads of $O(|M_2|)$ and $O(|D| + |M_2|)$, respectively. At each step i, the runtime space is $O(|M_i|)$ and compute time is $O(|D| + |M_i|)$. Since previous hashmaps are discarded, selecting n-grams up to size k results in total space $O(|M_k|)$ and compute time $O(k \cdot (|D| + |M_k|))$. Given that $\sum_{i=1}^{k} |M_i| \leq |D|$, the overall space is $O(|D|)$ and compute time is $O(k \cdot |D|)$. In practice, k is usually small and can be treated as a constant.

4.2 BEST

Although FREE is effective in selecting n-grams with high filtering power for indexing, it has the limitation of not considering the query set. As a result, it cannot guarantee that the index will be as helpful when the character frequency distribution for the query literals differs from that of the dataset. BEST remedies this problem.

4.2.1 Selection Strategy. BEST utilizes both the dataset and the query set as sources for n-gram selection. In addition to considering the *selectivity* of n-grams within the dataset, it also takes into account the frequency of n-gram occurrences in the query set. This approach helps avoid selecting n-grams that do not benefit any regex query, as illustrated in the example in Section 3.

Assumption BEST-1. It is *beneficial* to index an n-gram that **does not appear** in a data record, so that it can filter out the data earlier; it is *beneficial* to index an n-gram that **appears** in a query, so that it can be used to filter out data records for the query.

Assumption BEST-2. The benefit of filtering out a data record d_i is similar to the benefit of filtering out a data record d_j, where $i \neq j$.

With these assumptions, BEST abstracts the n-gram selection problem into a graph cover problem. Each data record, query, and n-gram is regarded as an individual node in the universe U. If an n-gram g is present in a query q, there is an edge between g and q; if g is absent from an input string d, there is an edge between g and d. In the context of our regex workload, each regex is matched once, so all edges have equal weights due to Assumption BEST-2. Each subset in U represents a set of connected nodes interlinked by a single n-gram. Formally, we have the following construction.

Definition 4.3 (BEST Cover). For a workload $W = (Q, D)$, the cover of an n-gram g is the set of data records $d \in D$ and query $q \in Q$ pairs defined as follows:

$$cover(g) = \{(q, d) \in Q \times D \mid g \in q \wedge g \notin d\}$$

For a set of n-grams P, the cover of the n-gram set is the union of the cover of each n-gram in the set.

$$cover(P) = \bigcup_{g \in P} cover(g)$$

The n-gram selection problem is then transformed to a budgeted maximum set cover problem [19] that aims to find the set that maximize the cover over all sets P that satisfy the budget constraint. Formally, BEST define the value of indexing an n-gram with respective to the entire workload by its *benefit*.

Definition 4.4 (BEST Benefit). For a workload $W = (Q, D)$ and an n-gram set I, the benefit of an n-gram $g \notin I$ is the number of additional query-data pairs covered by g that is not covered by I.

$$benefit(g, I) = |cover(I \cup \{g\}) - cover(I)|$$

To progressively calculate the benefit of each n-gram g of $G(W) = G(D) \bigcup G(Q)$, we need two matrices to store the existence of g in each data record d and each query q. Specifically, the two matrices are of size $|Q| \cdot |G(W)|$ and $|D| \cdot |G(D)|$ respectively.

Definition 4.5 (BEST Cost). For a workload $W = (Q, D)$, the cost to index an n-gram g is the storage overhead of g.

The cost depends on the index structure. In the original BEST paper, B+-tree is used, and the cost of g is the number of leaf pointers corresponding to g, i.e. the number of data records in D that contain g. For the inverted index, the cost of g is the size $s_D(g)$.

Definition 4.6 (BEST Utility). For a workload $W = (Q, D)$ and an n-gram set I, the utility of indexing an additional n-gram g is the ratio of its benefit over cost, i.e., $utility(g) = benefit(g, I)/cost(g)$

BEST selects n-grams based on their utility. The brute-force method would be to iteratively select the n-gram with the highest utility, g_{max}, based on the workload W and the set of selected n-grams I, among all possible n-grams with a positive benefit.

Assumption BEST-3. The average selectivity of candidate n-grams is low for the data records and the literals in the queries.

Assumption BEST-4. The average number of characters in all literals of a regex is much smaller than that of data records. The number of regex queries is much smaller than number of data records in the same workload.

Therefore, according to the sparsity assumption BEST-3 and the definition of cover, instead of considering all n-grams in the query set and dataset, we can trim away the n-grams that only exists in the dataset. Therefore, the candidate n-gram set G of BEST is $G(Q)$. BEST choose to use adjacency lists: *Q-G-list* and *G-D-list* rather than two matrices of sizes $|Q| \cdot |G(W)|$ and $|G(W)| \cdot |D|$ to store the existence of candidate n-grams to reduce space usage. The candidate n-grams, queries, and data records are each assigned an index number. The indices of *Q-G-list* correspond to query numbers, and each element is a list of n-gram numbers in this query. Similarly, the indices of *G-D-list* correspond to n-gram numbers, and each element is a list of data record numbers which the n-gram is in.

4.2.2 Approximation Techniques. The budgeted maximum set cover problem optimization problem is NP-hard [19, 25]. The search space is the product of the number of possible query-data pairs, $|Q| \cdot |D|$ and the number of possible n-grams, $|G|$. When the query size and the dataset size is large, the number of candidate query-data pairs can be prohibitively large to select the n-grams using brute force. BEST employs several techniques to get an approximation.

Pruning. BEST introduces pruning of common n-grams to reduce the candidate set size. With the same assumption as Assumption FREE-1, BEST prune n-grams with selectivity larger than a threshold c.

Parallelism by Clustering. BEST clusters the regex queries into small groups of similar queries that contains largely overlapping n-grams. Within each small group Q_i, the computation will be $O(|Q_i| \cdot |D| \cdot |G(Q_i)|)$. Formally, the distance is calculated as

$$Dist(q_1, q_2) = \frac{|(G(q_1) - G(q_2)) \cup (G(q_2) - G(q_1))|}{|G(q_1) \cap G(q_2)|}$$

Since queries with similar set of n-grams are clustered together, we minimize the number of n-grams for each subproblem, that is, the size of $G(Q_i)$. The clustering allows BEST to divide the search in each iteration into smaller sub-problems that allows for parallel computation. The intermediate cost and benefit results from all sub-problems are aggregated at the end of each iteration to select the n-gram with maximum utility.

Workload Reduction. When the query set size $|Q|$ is large, BEST selects a representative sample $Q' \subseteq Q$ to reduce the computation overhead. To get a representative sample, BEST use the same clustering technique to cluster the literals in Q. When the clusters stabilize, the median query of each group is added to Q'.

4.2.3 Complexity Analysis. In the first step, the workload reduction method described above shrinks the query set from $|Q|$ to $|Q'| = \frac{|Q|}{t}$, with the candidate n-gram set G' also reduced proportionally. Assuming that the average query length is μ and requires T iterations of k-median, this step takes $O(|Q| \cdot |Q'| \cdot \mu \cdot T)$ time. A suffix tree is used to enumerate all n-grams in $O(|G'|)$ time. Next, clustering enables parallelism and, per the original BEST paper, reduces time and space by $\sim 5\times$ even on a single thread. Using adjacency lists (*Q-G-list* and *G-D-list*) instead of matrices cuts space from $O(|G| \cdot (|D| + |Q|))$ to $O(\mu \cdot (|D| + |Q|))$, which is small since $\mu \ll |G|$. However, building these lists still takes $O(|G| \cdot (|D| + |Q|))$ time. By Assumption BEST-4, $\mu \ll |G|$, and therefore, the memory overhead during computation is small. Note that we still need $O(|G| \cdot (|D| + |Q|))$ time to build both data structures. The two adjacency lists contributes to the majority of extra space overhead for BEST. In each iteration, BEST will examine all remaining candidate n-grams and their benefit considering the index I built so far. For each pair $(q, d) \in (Q' \times D)$, it checks if the pair covered by each candidate n-gram g. In each iteration, BEST evaluates all remaining n-grams against $(q, d) \in Q' \times D$, costing $O(|I| \cdot |G(Q)| \cdot |Q| \cdot |D|)$.

4.3 LPMS

Despite several optimizations in the BEST algorithm, its performance is still too slow. LPMS remedies this issue by introducing approximations in the algorithm for BEST via integer programming.

4.3.1 Selection Strategy. Similar to BEST, LPMS also uses both the query set and the dataset as sources of n-gram selection. LPMS also incorporates assumption FREE-1 that an n-gram that eliminates more data records is more *useful*, but it incorporates the impact of queries by adjusting it with the *selectivity* of n-grams in queries and the length of the n-gram. Formally,

Definition 4.7 (LPMS Coverage). The *coverage* of an n-gram g is defined as the ratio of the support of g in the dataset D and the support of g in the query set Q, normalized by the n-gram length: $cv(g) = s_D(g)/|g| \cdot s_Q(g)$

Similar to Assumption BEST-4, we assume $G = G(Q)$. Using the binary variable $x_g = \mathbb{1}\,[g \in I]$, we form the objective function as $\sum_{g \in G} cv(g) x_g$. Note that $x_g \in \{0, 1\}\ \forall g \in G$. To provide good approximation to the optimal solution, the approximate set of n-grams should satisfy the following:

Assumption LPMS-1. The index should filter out at least as many data records compared to any candidate n-gram.

Let g_j be the j-th n-gram in the candidate n-gram set G and q_i be the i-th query in the query set Q. LPMS constructs a matrix A of size $|Q| \times |G|$ and a vector b of size $|Q|$, where $A_{i,j} = s_D(g_j) \cdot \mathbf{1} g_j \in G(q_i)$ and $b_i = \min_{g \in G(q_i)} s_D(g)$. This setup allows us to establish the constraint of the integer program, formalizing the Assumption LPMS-1 into the constraint $Ax \geq b$. However, the search space for all possible n-grams G remains too large when the query set is large. LPMS adopts an iterative approach to select a prefix-minimal n-gram set from FREE. In the i-th iteration, LPMS generates the candidate n-gram set G_i with n-grams of size i from all the useless n-grams from G_{i-1}. After solving the integer program in the i-th iteration, we insert the set of n-grams I_i with $x_g = 1$ for all $g \in I_i$ into the index, and the remaining n-grams $G_i \setminus I_i$ are used to extend and generate G_{i+1}.

Solving the integer program is challenging, as the search space is $|G_i|^{O(|G_i|)}$ [21]. LPMS approximates the problem using linear programming with relaxation, as follows:

$$\text{minimize} \sum_{g \in G} cv(g) x_g \text{ subject to } Ax \geq b; 0 \leq x_g \leq 1 \quad \forall g \in G$$

4.3.2 Complexity Analysis. To calculate the space overhead, let's examine the sizes of each component. Both *coverage* and the output of the linear program are vectors of size $|G_i|$. As previously discussed, A has a size of $|Q| \cdot |G_i|$ and b has a size of $|Q|$. Summing these, the space overhead for LPMS n-gram selection is $O(|Q| \cdot |G_i|)$. During the algorithm runtime, *coverage* is calculated using $s_D(g)$ and $s_Q(g)$. By utilizing *coverage* for each n-gram rather than *cover*, LPMS reduces the time complexity for constructing the *coverage* vector to $O(|G_i| \cdot (|D| + |Q|))$ for each iteration. The linear program runs in polynomial time $O(|G_i|^{2.5})$ [36]. In practice, the size of the index key typically does not exceed 10, a number also used as the upper bound for n-gram size in FREE. Therefore, we can consider the small number of iterations as a constant, making the overall computational complexity of LPMS $O(\sum_{i \in [10]} \{|G_i|^{2.5} + |G_i| \cdot (|D| + |Q|)\})$.

4.4 VGGraph

The n-gram selection strategy from VGGraph [30] combines the insights from FREE and BEST to select the n-grams.

4.4.1 Selection Strategy. The idea is to select the most cost-effective n-gram from a set S to cover characters in an uncovered set E. Each n-gram is evaluated based on a cost-efficiency ratio (a notion similar to *cover* from Theorem 4.3), where the cost is size of the inverted index for the n-gram and the efficiency is the number of characters it helps cover in E. VGGraph approximate a perfect set cover with efficiency by adopting FREE's idea of *Usefulness*, and only extend the useful n-grams. By maintaining the uncovered set E that is continuously updated as the n-grams are picked, VGGraph ensures that prefix-free n-grams are selected preferentially.

4.4.2 Complexity Analysis. VGGraph's index key selection consists of two steps: (1) recursive extension of grams exceeding the frequency threshold c, and (2) greedy n-gram selection via the set-cover approximation. Given a dataset D, the complexity of recursively extending grams is $O(|D| \cdot q_{\min})$, as each character position is indexed exactly once and each gram is extended at most once. The greedy set-cover step selects, for each regex, a minimal subset of grams from the candidate set $G(Q)$. For a regex with literal size n, the greedy selection has a complexity of $O(|G(Q)| \cdot n)$ per query. The total runtime complexity for key selection, combining both steps, is thus $O(|D| \cdot q_{\min} + |Q| \cdot |G(Q)| \cdot n)$. The corresponding runtime space complexity during key selection is $O(|D| + |G(Q)|)$, dominated by storing candidate grams and their counts.

5 EXPERIMENT SETUP

Due to the absence of source code from the original works, we implement the three n-gram selection methods in our codebase. For a fair comparison of the n-gram selection methods, we use the same index structure (inverted index) and the same basic query matcher for all experiments. We include the implementations of other index structures and query plans as per the original papers in our codebase for reference.

5.1 Benchmark Framework

The benchmark framework is designed to facilitate the comparison of n-gram selection techniques (FREE, BEST, and LPMS) for regex indexing, with a focus on modularity, extensibility, and reproducibility across a range of workloads. We summarize its detailed architecture and workflow in Figure 1.

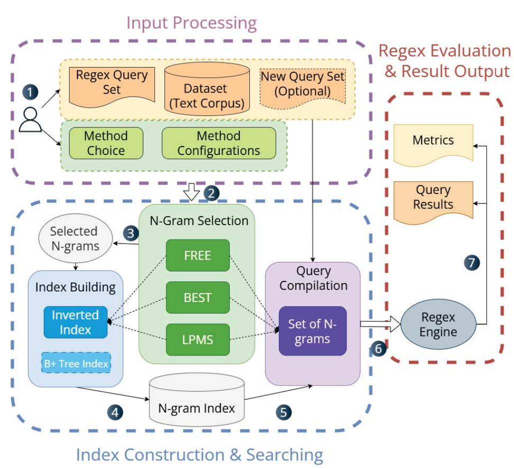


Figure 1: Benchmarking framework overview.

The end-to-end process follows a seven-step sequence, as illustrated in the framework diagram Figure 1. It begins with user-specified inputs (dataset, regex queries, methods, and configurations), which are processed by the framework. The selected n-gram strategy analyzes the workload to identify optimal n-grams for index construction. These n-grams are then used to build an index (e.g., inverted index or B+-tree). Next, the index search plan extracts n-grams from regex literals to guide candidate filtering. The regex engine verifies these candidates, discards false positives, and computes performance metrics. Finally, results are aggregated and exported. The pipeline is organized into three phases—input processing, index construction, and regex evaluation—with components grouped accordingly in Figure 1.

Input Processing. The framework begins by accepting a text file containing the string dataset and a text file with the regex query set. Users also specify n-gram selection methods and their parameters, including n-gram length, selectivity thresholds, maximum number of n-grams, thread counts, and more for index building. Optionally, a new query set can be provided at runtime, enabling dynamic evaluation of the index against unseen query workloads. These inputs are standardized to a unified format across experiments. During this phase, all data in the workload is loaded into memory.

Index Construction and Searching. In the index construction phase, one of the three n-gram selection strategies is applied to the workload. The selected n-grams are used to build an index. This phase leverages multi-threaded execution if specified by the user. After the index is built, the index search plan is compiled if necessary. An inverted index with data string IDs is built with the selected n-grams. In the simple query plan, posting lists corresponding to the set of n-grams that exists in the regex query literals are unioned together later for full regex evaluation.

Regex Evaluation and Result Output. The final phase processes all possible data points after index lookup and validates them using a regex engine to perform exact matches and eliminate false positives. Metrics such as index construction time, runtime memory consumption, workload processing time, and false-positive rates are measured during index construction or after regex evaluation.

Table 2: Workload $W = (Q, D, \Sigma)$ statistics. $|\bar{d}|$ denotes the average data record size (in bytes) and $\bar{TP}$ denotes the average number of actual matches of all queries. Robust has two query sets – the first set of 500 queries that is used to build the index, and a second set of 100 queries that is used to test the impact of queries that are not indexed.

Workload	$\|Q\|$	$\|D\|$	$\|\Sigma\|$	$\|\bar{d}\|$	$\bar{TP}$	Dataset Size
Webpages	10	695,565	255	68,650	86,524	47 GB
DBLP	1000	305,798	122	38	914	32 MB
Prosite	101	111,788	22	416	722	45 MB
US-Acc	4	2,845,343	99	405	92,042	1.1 GB
SQL-Srvr	132	101,876,733	114	139	50,356	14 GB
Enron	107	517,401	120	2719	54,242	2.6 GB
ADX	18	890,623,051	118	101	4,653,057	100 GB
Synthetic	vary	50,000	16	200	vary	9.6 MB

5.2 Workloads

We use several real-world text datasets and queries with varying numbers of queries, data strings, alphabet sizes, average string lengths, and average matches per query for a comprehensive analysis. We summarize the workload characteristics in Table 2. Among the workloads, three (**Webpages**, **DBLP**, and **Prosite**) are those used by the original FREE, BEST, and LPMS papers to evaluate their methods. The other four workloads (**US-Acc**, **SQL-Srvr**, **Enron**, **ADX**) are more recent and feature larger datasets.

Webpages. This workload was used by FREE. In the original paper, the authors utilized 700,000 random web page HTML files downloaded in 1999, along with 10 regex queries suggested by researchers at IBM Almaden [6]. While the exact query set is included in the paper, the dataset is not. Since we could not locate the original dataset, we constructed a similar dataset using web pages from 2013 stored in Common Crawl [11]. We chose 2013 data because it is relatively close to 1999, ensuring that most of the regexes constructed for the 1999 dataset would still have matches in the 2013 dataset. We selected the web pages to ensure a relatively balanced number of matches for each regex query.

DBLP. This workload was used by BEST. The authors collected 305,798 (Author-Name, Title-of-Publication) tuples as the dataset. We used the DBLP-Citation-network dataset [34] and selected the same number of entries uniformly at random. The query set is constructed by choosing author last names from the pool of author names uniformly at random to obtain 1000 queries. We then constructed the regex query for each last name by appending `.+` and a space in front of the last name.

Prosite. This workload was used by LPMS. Following the paper's description, we selected 100,000 protein sequences from the PFAM-A database [26] and chose 100 Prosite signatures [33], transforming them into 100 regular expressions.

US-Acc. This is an open-source dataset containing descriptions of traffic accidents in the United States from February 2016 to March 2019 [27]. The dataset has 2,845,343 strings. The authors included four regex queries on the dataset in the original table.

SQL-Srvr. This is a production workload consisting of in total 101,876,733 log messages generated by Microsoft SQL Server and 132 regex queries used for data analysis tasks on this dataset.

Enron. This is a production workload consists of more than 50 thousand of real emails and 107 regex queries. The dataset is the May 7, 2015 Version of Enron dataset [10], and the queries are from Avatar search engine [18] collected by Chen et al. [5].

ADX. This is a real production workload consists of 890 millions of logs from Azure Data Explorer (ADX) and 18 regexes for an log analysis task on this dataset by an data scientist.

Synthetic. This workload is based on the method described in the paper proposing LPMS [35]. The workload consists of 50,000 data records with a fixed alphabet size of 16. We varied the query set size across [500, 1000, 2000, 5000] and explored different levels of query selectivity ranging from 0.001 to 0.5.

5.3 Metrics

We compare the three methods on the following aspects:

N-Gram Index Construction Time (T_I). We measure the time required to select all n-grams for indexing and building the index, denoted by T_I, after loading the necessary data into memory. This aspect is crucial, as we deal with much larger datasets than when these n-gram selection algorithms were originally presented. The selection time may become prohibitively long.

Precision. We use micro-average precision to compare the filtering power of an index. This metric is commonly used in information retrieval [24, 40]. It provides a balanced measure across all queries with different numbers of matches, without giving disproportionate weight to any individual query. It aggregates actual data records that match the regex query (true positives, denoted by TP) and data records that pass the index filtering but do not match the regex query (false positives, denoted by FP). For a workload $W = (Q, D)$, the overall precision on index I is: $Prec_W = \frac{\sum_{q \in Q} \#TP_q}{\sum_{q \in Q} \#TP_q + \sum_{q \in Q} \#FP_q}$

Query Time (T_Q). We use the time to run the workload, denoted by T_Q, to demonstrate the runtime gains with the index.

Runtime Space Usage (S_Q). This metric is the peak space used when running a experiment.

Index Size (S_I). We measure the size of the index constructed, which is the total size of the n-gram keys, the posting lists, and necessary index data structure (such as the inverted index).

5.4 Hardware and Implementation Details

We implemented n-gram selection methods `FREE`, `BEST`, `LPMS`, `VGGraph`, and `Trigrams` with the index construction and matching methods mentioned in the benchmarking framework. We implemented `FREE`' query parser and `BEST` with a B+-tree in the codebase as well. We used Gurobi Optimizer [15] as the linear program solver. All experiments were conducted on an Azure Standard_E32-16ds_v5 machine with an Intel(R) Xeon(R) Platinum 8370C CPU @ 2.80GHz, 16 vCPUs, 256 GB of memory, and 1 TB of disk storage. Our benchmarking framework is written in C++17 and compiled with the -O3 flag. Google's RE2 (release version 2022-06-01) [14] as the regex engine for regex evaluation.

Each method has its own configurable parameters. For `BEST` and `FREE`, we vary the selectivity threshold c between 0.01 and 0.7. `FREE` also varies the maximum n-gram length ($\max_n \in \{2, 4, 6, 8, 10\}$) and whether the selected n-grams are prefix-minimal or pre-suf-minimal. All experiments are capped at 3 hours; if `BEST` exceeds this, we reduce the workload to 0.1–0.85%. `LPMS` supports both deterministic (`LPMS`-D) and randomized (`LPMS`-R) relaxations. We also cap the number of n-grams for `LPMS` if runtime is excessive. All methods run with 16 threads. To ensure fair comparison, we match index sizes across methods. Since `FREE` and `LPMS` don't natively support key limits, we implement early stopping via a max-key parameter for all three methods. By providing an optional parameter of max number of keys, the n-gram selection methods will stop once the limit is reached. Each configuration of a selection method generates a specific set of n-grams of varying sizes. To ensure a fair comparison, we compare selected sets of similar sizes, as indexing all possible n-grams would yield the highest precision for the workload. Since each methods has different configurations, and those configuration values interfere with each other in a non-linear way, isolating one configuration and study its behavior as value changes is not meaningful. For each n-gram selection method under a specific key number constraint K $s.t.$ $|I| \leq K$, we select the configuration with the highest precision. We choose the values of max number of keys, K, on a case-by-case basis for each workload, as specified in Section 6.

6 EXPERIMENTS

In this section, we run the aforementioned workloads on the indexes build from different n-gram selection methods with varying parameters. In particular, we aim to answer the following questions:

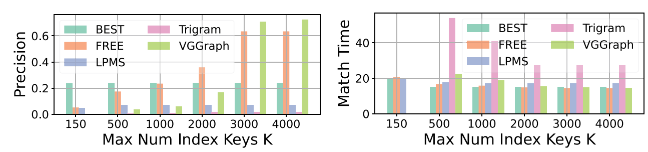
- **Q.1** What is the usefulness of each n-gram selection method? We evaluate this through precision and T_Q.
- **Q.2** What is the index construction overhead for each method? We evaluate this through T_I, S_Q, and S_I.
- **Q.3** How do the characteristics of each workload affect the usefulness of n-gram selection methods?

We answer question Q.1 in Section 6.1 and question Q.2 in Section 6.2. To answer question Q.3, we discuss the impact of characteristics of different real-world workloads on index construction overhead in Section 6.2 and on index effectiveness in Section 6.1. We also use present a more thorough ablation study on impact of workload parameters such as the query set size $|Q|$, dataset size $|D|$, query selectivity, etc. in Section 6.3.

6.1 Query Performance

To address question Q.1, we compare the performance gains from using indices built with n-grams selected by the presented methods, focusing on filtering precision. Higher precision indicates a more effective n-gram set. We also measure the regex query runtime T_Q, which reflects precision since the index structure and matching strategy remain constant.

Generally, indexing more n-grams improves precision. However, each method behaves differently. `BEST` often starts with the highest precision when the key budget K is small, but its precision gains plateau due to time constraints (e.g., 3-hour index build). In contrast, `FREE` begins with lower precision—especially for small query sets—but improves significantly as K increases, often achieving the highest precision overall. Precision of `VGGraph` is low when number of keys is small, but quickly rises as number of keys gets larger. `LPMS` has similar precision as `BEST` when the numbers of key is small and usually grows insignificantly as the number of n-grams selected increases. `Trigrams` typically has very low precision when the number of index keys fixed is low. We now investigate the methods' precision for different workloads with varying characteristics.



(a) Precision **(b) Query Time T_Q (s)**

Figure 2: DBLP.

6.1.1 *Workload* ***DBLP***.

We evaluate `BEST`, `FREE`, `LPMS`, and `VGGraph` across various input configurations, and applying the same key limit to `Trigrams`. Performance is compared based on index size, focusing on configurations with at most 5000 n-grams.

As shown in Figure 2a, `BEST`($c = 0.5$) achieves a precision of 0.235 for $K \leq 150$, outperforming `FREE`($\max_n = 2$, $c = 0.5$) and `LPMS`-D, which yield much lower precision (0.051 and 0.048, respectively) at a similar index size. At $K \leq 500$, the 363 n-grams selected by `BEST`($c = 0.5$) achieves significantly better precision compared to `LPMS`-D. With similar index construction overhead, `FREE`($\max_n = 2, c = 0.5$) achieves higher precision than `LPMS`-D. `LPMS` tends to select infrequent n-grams, which may not benefit overall workload filtering. Although `BEST` improves slightly with more keys (e.g., +0.007 precision and +4.5s query performance from $K = 150$ to $K = 500$), gains are modest. Meanwhile, `VGGraph`($\max_n = 2, c = 0.5$) builds an index with 215 n-grams but with a low precision of 0.035.

When relaxing the index size constraint to 1000 n-grams, `FREE` ($\max_n = 4, c = 0.5$) generated an index of size 810 that achieved a precision of 0.219, slightly lower than BEST($c = 0.5$). VGGraph($max_n = 3$, $c = 0.5$)'s precision increase is insignificant. As the number of n-grams allowed in the index increases, starting from 2000, more indices from different configurations of FREE fall into this range. The best configurations of FREE($\max_n = 2$,$c = 0.2$) achieve higher precision than BEST. The precision increases from 0.321 at 2000 keys to 0.524 at 4000 keys upper limit by FREE($\max_n = 4$,$c = 0.15$). Precision of VGGraph increases significantly from 0.167 to 0.725 as number of keys increases from 1014 to 3681.

DBLP runs much faster with all index configurations than the baseline (i.e. using no indexes at all) of 99 seconds. Even when Trigrams indexes are built using only the first few trigrams—yielding low precision between 0.006 and 0.017, they still achieves performance improvement of 1.8× to 3.6×. Looking at Figure 2b, however, the improvement in workload performance is significant compared to the change in precision. The precision of FREE increases around 10 times as K increases from 150 to 3000; its query time decreases by 41%. When $K = 150$, the precision of BEST is significantly larger than FREE and LPMS in Figure 2a, the performance difference for BEST and LPMS is not significantly different. In the **DBLP** workload, each string in the dataset is short, resulting in very quick regex evaluation, and therefore the query time reduction brought by precision increase is less significant. Trigrams's precision increases as number of keys increases. When indexing all possible 28,275 trigrams, we can get a high precision of 0.97. However, the query time is 14.7s, similar to BEST with less than 500 keys. This suggests that indexing all trigrams is not cost-efficient in terms of storage.

Insights. FREE performs best on the **DBLP** workload. Generally, if the workload has a large number of queries that are not skewed (i.e., all very similar and covering a small subset of the dataset) and the strings in the dataset are not long, it is best to choose FREE. VGGraph significantly improve its precision only when the number of keys increase. When the query set is large, BEST and LPMS require significant computation time and memory, owing to their higher time and space complexities. Additionally, for a large-sized and balanced query set, frequent n-grams in the datasets are likely to be covered in the queries as well.

Table 3: Index cost and query performance on Webpages.

K	Method	T_Q s	T_I s	S_Q GB	S_I MB	*Prec*
5	BEST	422	8587	150	0.1	0.138
	FREE	440	382	101	0.1	0.123
1.7×10^5	BEST	422	8587	150	0.1	0.138
	FREE	353	1184	159	357.9	0.163
3.6×10^5	BEST	422	8587	150	0.1	0.138
	FREE	240	5965	240	2007.4	0.301

6.1.2 *Workload* ***Webpages***.

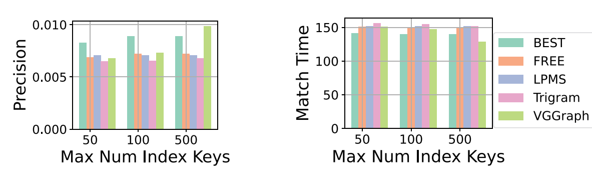
We evaluate **Webpages**, a workload with significantly fewer queries than **DBLP**, using HTML files from **Webpages**, which has the longest average string length among all workloads. Results are summarized in Table 3, with a baseline (no index) query time of 437s.

At $K = 5$, BEST($c = 0.1$) selects just four n-grams and achieves 0.138 precision. A manually early-stopped FREE($\max_n = 2$, $c = 0.02$) with five n-grams yields a similar 0.123 precision. To match that precision without early stopping, FREE($\max_n = 2$, $c = 0.5$) requires 1.7×10^5 keys. This difference stems from selection strategies: BEST chooses n-grams from the query–dataset intersection, limiting candidates when queries are few. In contrast, FREE selects from the dataset alone, resulting in a much larger candidate pool. Despite requiring more keys, FREE achieves 0.025 higher precision than BEST and delivers a 1.2× speedup in query time—reducing it from 421.6s to 353.0s as shown in Table 3. When the number of keys allowed K increases from 1.7×10^5 to 3.6×10^5 , the precision of the index by FREE($\max_n = 2$,$c = 0.7$) increases significantly from 0.163 to 0.301. At $K = 3.6 \times 10^5$, FREE delivers a query time that is 1.47× faster than its own earlier version and 1.75× faster than BEST. Compared to the overall precision and query time trend for **DBLP** workload in Figure 2, the precision increase of FREE in **Webpages** corresponds to a more significant workload runtime decrease from 353 seconds to 240 seconds. Since each string in **Webpages** is a long HTML page, filtering more strings upfront yields greater performance benefits. While FREE's larger index may not always justify its size relative to BEST, it offers better robustness to unseen queries on the same dataset. Due to lack of space, we defer the reader to Appendix B [42] of the full paper for detailed experiments. VGGraph and LPMS did not finish for this dataset within the set time frame due to the large average length per string of the dataset.

Insights. BEST is suitable for a workload like **Webpages** where the query set is small. BEST selects the set of n-grams that achieves near-optimal precision, with the optimality resulting from the long computation time. With a small query set, the index construction time is reasonable. However, for a workload where each document entry is large, the precision and robustness of the index become more important, especially if the dataset is likely to be queried with different regex queries.

6.1.3 *Workload* ***Prosite***.

Prosite has the largest query size to number of records in the dataset ratio among the real-world workloads. The strings in the dataset have a mean length of 416 characters. Two distinguishing characteristics of **Prosite** are: 1) it has the smallest alphabet size of 22, and 2) it has very short literal components in its the regex queries. The short literal components gives little opportunity for n-gram index improvement, and all methods created indices with low precision and performance improvement. In fact, the baseline workload runtime is 154 seconds.



(a) Precision **(b) Query Time T_Q (s)**

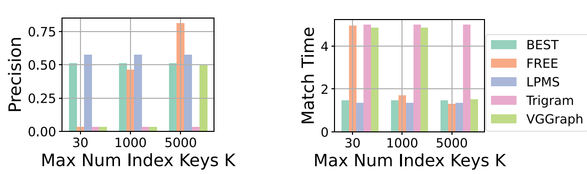
Figure 3: Prosite.

BEST is the most performant for this workload, when K is small, generating index with highest precision while using only 364 keys. The small alphabet size and small literal size per query make the number of possible n-grams considered in each calculation iteration small, and thus drastically reduce the index construction runtime overhead, which is the biggest advantage of BEST in other workloads. Similarly, VGGraph which adapts a similar cover strategy, achieves the best precision as K gets larger.

When indexing with 50 keys, FREE($\max_n = 2, c = 0.15$) generates an index with a higher precision of 0.00651, while taking the least computational overhead and smallest index size. LPMS-D, taking slightly longer to select the n-grams and generate the index, achieves a precision of 0.00708 within 14 seconds. BEST($c = 0.7$) achieves the highest precision, 0.00826, among the three methods. Its query time is also more than 10 seconds lower than the other two methods. Overall index precisions are low for **Prosite** workload, as the average length of literals is very short in the regexes.

When $K = 100$, FREE, LPMS, and Trigrams does not lead to much improvement. Index by BEST($c = 0.7$) has a higher precision of 0.0089, resulting in a more than 10 seconds workload performance improvement over other methods. VGGraph ($max_n = 6$, $c = 0.7$) also increases slightly to 0.0732. The precision increments for all methods does not corresponds to significant performance improvement as shown in the bars in Figure 3b. As the key size constraint relaxes to $K = 500$, selected n-grams of VGGraph ($max_n = 2$, $c = 0.7$) achieves the highest precision of 0.0987, and the significant precision improvement from the previous K led to a query time reduction to 128.5 seconds, 1.13× query time reduction from the baseline. Since the literal lengths are small in this workload, the effectiveness of each n-gram is much more important for effective filtering. Therefore, n-gram sets selected by dataset-based methods, FREE and Trigrams, are not effective in filtering.

Insights. For workloads where query literals are short and/or the alphabet size is small, BEST can select an n-gram set with high filtering precision while incurring reasonable computational and storage overhead. This advantage arises from the smaller n-gram candidate set. VGGraph outperform other methods in terms of precision as number of n-grams scales up. FREE does not benefit from this advantage since it only looks at n-grams in the dataset. LPMS benefits less from the characteristics of the workload due to the fixed overhead for constructing the integer program.



(a) Precision **(b) Query Time T_Q (s)**

Figure 4: US-Acc.

6.1.4 *Workload **US-Acc**.*

This workload contains the smallest number of queries. Compared to the **Webpages** workload, which also has a small query set, the datasets and queries are generated by a limited number of templates. The data records in the dataset consist of the location description (e.g., "At I-270, Between OH-48/Exit 29 and Dayton Intl Airport Rd/Exit 32") and a brief description of the accident. Each string in the **US-Acc** dataset is much shorter.

Figure 4 shows the results. The methods that uses both query set and dataset as n-gram sources generate better query performance than the baseline query time of 4.8 seconds. LPMS-D performs the best for the **US-Acc** workload, achieving a precision of 0.572 using only 12 keys. When K increases to 1000, FREE generates an index with 837 keys and a precision of 0.46, which is much lower than BEST and LPMS with fewer than 30 keys. This is because FREE selects n-grams based solely on their selectivity in the dataset, omitting those n-grams that are frequent in both the regex query and the dataset. When K increases to 5000, VGGraph ($max_n = 3$, $c = 0.7$) generates a index with 4247 keys and achieves an query time of 1.51 seconds that is comparable to other faster methods.

Insights. For workloads where the query literals are very short and common in the dataset, LPMS can quickly locate a small set of n-grams with high precision. VGGraph, BEST, and FREE remove n-grams that are frequent in the dataset, as they both adhere to Assumption FREE-1 that does not hold for the dataset.

6.1.5 *Workload **SQL-Srvr**.*

The **SQL-Srvr** workload, like **US-Acc**, consists of structured strings—specifically, system log reports—but with a much larger dataset and query set. BEST was unable to complete within the time limit. Results are summarized in Table 4.LPMS outperforms FREE due to its query-aware design. With just 50 n-grams, LPMS-D achieves a precision of 9.9×10^{-3} which is 10× higher than FREE($\max_n = 2$, $c = 0.7$) under the same key limit. FREE's query-agnostic approach often selects shor n-grams from variable fields (e.g., VM IDs), which offer little benefit for filtering. Consequently, LPMS-D delivers a 2.43× faster query time. Although constructing LPMS incurs 3.4× and 1.9× more space and time overhead, the index built by LPMS uses 1.48× less space. Since both methods use the same inverted index and number of keys, the difference stems from posting list lengths. LPMS selects more selective n-grams with shorter posting lists. This, combined with higher precision and faster queries, makes LPMS more effective overall. The results remain consistent until the upper limit of the number of n-grams grows to 5000. At this point, FREE($\max_n = 2, c = 0.03$) selects an n-gram set of size 4643, increasing the precision to 2.68×10^{-3}. However, it still lags behind LPMS (even when using as low as 50 n-grams) in both precision and query time.

Insights. When query literals are common, the n-grams that have high *benefit* may also have high *selectivity*, which contradicts Assumption FREE-1. For workloads where query literals are common in the dataset, even when the query literals are long, LPMS selects the n-gram set that improves query matching time the most. This is due to LPMS's design choice of not discarding any n-grams based on selectivity threshold.

6.1.6 *Workload **Enron** .*

The **Enron** workload contains long data strings in the dataset like **Webpages**, yet it has relatively small dataset size and larger query size. The lengths of literal components among queries are generally short. Although each data string is long, the small dataset size and the short literal components make

Table 4: Index cost and query performance on SQL-Srvr.

K	Method	T_Q s	T_I s	S_QGB	S_IMB	*Prec*
50	FREE	3226.7	937	76.7	6643.1	0.001480
	LPMS	1421.1	4569	147.5	4474.2	0.009855
	Trigrams	1552.3	70	49.9	5529.5	0.003016
100	FREE	3226.7	937	74.0	6643.1	0.001480
	LPMS	1421.1	4569	147.5	4474.2	0.009855
	Trigrams	1552.3	70	49.9	5529.5	0.003016

Table 5: Index cost and query performance on Enron.

K	Method	T_Q s	T_I s	S_QGB	S_IMB	*Prec*
10	BEST	103	1781	3	5.7	0.148
	FREE	126	30	3	0.0	0.105
	LPMS	124	79	3	16.3	0.107
	Trigrams	127	4	3	0.9	0.105
100	BEST	69	2286	3	54.2	0.321
	FREE	123	35	3	5.1	0.109
	LPMS	101	2104	13	256.2	0.149
	Trigrams	127	4	3	0.9	0.105
	VGGraph	125	45	3	0.4	0.105

the computation of majority of the methods within our set time limit. We run BEST, FREE, LPMS, VGGraph, and Trigrams on this dataset and summarize the results in Table 5. Without indexing, **Enron** runs in 126 seconds as a baseline.BEST ($c = 0.7$) achieves highest precision for $K = 10$. With a precision of 0.148, around 1.4× higher than other methods, BEST uses 1.2× less time for the workload querying. As K increases to 100, precision of BEST increases by 2.2× to 0.321, and the query time decreases by 1.5×. LPMS-D, having similar strategy as BEST, also achieves precision incremental as K increases from 10 to 100 with significantly reduced precision. **Enron** has one of the highest regex match rate, where without filtering, the true positive rate per regex is already around 0.105, suggesting that VGGraph and Trigrams does not perform effective filtering. When number of keys selected for the index continue to increase, the query performance and precision fails to increase proportionally. This is due to the fact that literal components in the queries are fairly common in the data strings, getting filtered out due to the selectivity threshold earlier on.

Insights. BEST demonstrates exceptional performance in **Enron**, efficiently leveraging its optimization-based approach with smaller query sets and longer literals. VGGraph shows substantial performance enhancement at moderate index sizes. LPMS offers balanced precision and performance, while FREE and Trigrams indexing lag behind due to their less targeted selection methods.

*6.1.7 Workload **ADX**.* The **ADX** workload has the largest dataset and a very small query set. Trigrams method would not generate useful index if tight constraint is posed on the number of n-grams selected, and will generate an index that is too large to fit into memory if no constraint imposed. VGGraph did not finish for all configuration on this workload. This workload runs in 1663 seconds without indexing. We run FREE, BEST, and LPMS on **ADX** workload and summarize the result in Table 6.

Table 6: Index cost and query performance on ADX.

K	Method	T_Q s	T_I s	S_QGB	S_IMB	*Prec*
10	BEST	311	24810	217	351.5	0.143
	FREE	1201	1324	43	0.3	0.023
	LPMS	373	2184	104	2799.0	0.113
25	BEST	254	28912	219	650.6	0.216
	FREE	1201	1324	43	0.3	0.023
	LPMS	221	2174	105	3360.4	0.328
150	BEST	254	28912	219	650.6	0.216
	FREE	332	947	88	30360.0	0.189
	LPMS	221	2174	105	3360.4	0.328

Due to the small query set size, and large and diverse data strings in the dataset, methods considering both query and dataset, BEST and LPMS achieves better precision and performance improvement for all key constraints K. Overall, LPMS-D achieves best precision of 0.328 and query performance of 221 seconds, 7.52× faster than the baseline. BEST ($c = 0.1$) also achieves 5.3× to 6.5× performance improvement with 10 to 25 keys.

Insights. For **ADX**, a workload with simple patterns and very large dataset size, LPMS stands out by effectively balancing precision and computational overhead. BEST becomes computationally prohibitive due to its complexity, and Trigrams indexing is impractical at this scale.

6.2 Index Construction Overhead

In this section, we aim to answer question Q.2 with a detailed analysis of the index construction overhead for FREE, BEST, LPMS, VGGraph, and Trigrams, based on their time and space usage observed during index construction.

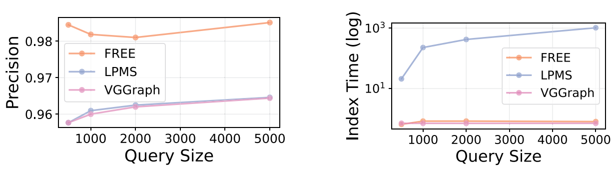
FREE demonstrates a significantly lower index construction time across various workloads due to its straightforward selectivity-based strategy and iterative prefix-free minimal n-gram selection approach. Specifically, for the **Webpages** workload, as shown in Table 3, FREE constructed an index with 1.7×10^5 keys in merely 1184 seconds compared to BEST's significantly longer 8587 seconds. FREE's runtime space overhead remains relatively constant and low across workloads, generally proportional to the dataset size due to the prefix-free minimal set construction method.

BEST has a high index construction overhead because it performs an exhaustive search to approximate optimal n-gram set through a utility-based set-cover strategy. Specifically, in the **DBLP** workload, BEST required 8762 seconds to construct an index of 363 keys, dramatically more than the sub-second construction time achieved by FREE for a comparable number of keys. The high overhead results from the auxiliary data structures mentioned in Section 4.2.2, required to maintain for time-efficient computation. In workloads with larger query sets or longer literals, such as **Webpages** in Table 3, BEST's computation overhead increases and exceeds practical time limits.LPMS has moderate to high index construction times, often higher than FREE but lower than BEST. For instance, in the **SQL-Srvr** workload results in Table 4, LPMS required 4569 seconds for constructing an index with 50 keys, significantly exceeding FREE's 1342 seconds for a similar number of keys but less than

BEST. LPMS's overhead results from its formulation and solving of the linear program.VGGraph has an index construction overhead generally lower than BEST but higher than FREE. The overhead is attributed to its iterative evaluation of cost-efficiency ratios. For the **SQL-Srvr** workload in Table 4, VGGraph constructed indices in 45 seconds, considerably lower than BEST and LPMS that runs more than 2000 seconds but slightly higher than FREE (30s). Its overhead scales as the dataset and query complexity increase. Trigrams has minimal index construction overhead when limited to a small number of keys, but this overhead grows rapidly as more trigrams are indexed—especially without key constraints. Its cost is primarily driven by dataset size rather than query complexity or optimization.

6.3 Case Study: Workload Characteristics

In this section, we perform an ablation study with **Synthetic** workload to evaluate how different regex indexing methods behave under varying query set sizes and query selectivity. For each configuration, we constrain the maximum number of n-grams (K) to 5000 and measured both precision and query runtime (T_Q). BEST failed to finish within the time limit. However, in the experimental result, varying regex selectivity does not result in a common trend on the indices built with different methods in workload performance or index construction. We will focus on the analysis of varying query size with fixed dataset size and regex query selectivity.



(a) Precision **(b) Index Time T_I (in seconds)**

Figure 5: Synthetic workload with $|D| = 50,000$ and average query selectivity 0.02.

Figure 5a illustrates that as the query set size increases from 1000 to 5000, FREE consistently maintained high precision. This can be attributed to the small alphabet size of the workload. LPMS and VGGraph also showed stable but lower precisions, with LPMS ranging around 0.95 and VGGraph at approximately 0.96. All three methods achieves similarly high precision. The precision of all methods slightly improve as query set size increased, as a larger query set provides more opportunities for n-gram filtering. Regarding index construction time (T_I), in Figure 5b, FREE and VGGraph exhibited significantly lower overhead, maintaining fast index build times which is under 1 second across all query set sizes. LPMS required considerably more time as the query set size increased. LPMS is sensitive to query set growth, requires over 90 seconds at larger query sets. This matches our observation from Section 6.2, where LPMS and BEST incurred much larger index construction overhead.

7 LEARNINGS AND FUTURE WORK

The evaluation shows that the optimal choice of n-gram selection strategies among FREE, BEST, and LPMS is strongly influenced by workload characteristics (such as the query set size, query literal sizes, alphabet size, data patterns, etc.). Since no single strategy suits all workloads, this analysis emphasizes the importance of selecting methods based on workload characteristics and provides a general guideline. We summarize the insights in the decision flow chart in Figure 6 as a guideline for practitioners. This guideline recommends 1) choosing FREE for large and diverse workloads or when unseen queries are expected, 2) choosing BEST when precision is crucial for repeated queries and the number of candidate n-grams is small, 3) choosing LPMS for formatted datasets or when query literals are common in the dataset, and 4) choosing VGGraph when precision requirement is high with low index space constraint.

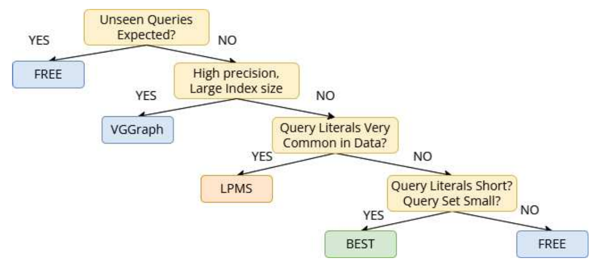


Figure 6: N-gram selection and regex index method decision tree by workload characteristics.

Based on this study, we also propose the following potential directions for future work in this area.

(1) **Unified solution.** Our results demonstrate that while each method performs well on certain workloads, there is no solution that performs well across the board. Designing an algorithm that can combine the best aspects of each indexing method or prove the non-existence of such an algorithm would be beneficial.

(2) **Better indexing formats.** Research in relation query processing has shown the benefits of using bit-based indexing formats (e.g. vector-based formats, BitWeaving [23], etc.). It would be interesting to explore the benefits of those formats for regex indexing.

(3) **Indexing on-the-fly.** All methods for regex indexing require an expensive preprocessing step. To the best of our knowledge, there are no existing indexing methods that can perform indexing on-the-fly as the queries in the workload are executed and as the dataset and query distribution change dynamically. Thus, adaptive query execution in this setting is another interesting direction.

8 CONCLUSION

In this paper, we present an investigation of common regex indexing methods. Our comprehensive evaluation spans a diverse array of scenarios and datasets, establishing a benchmark for assessing the performance of these techniques. Through meticulous analysis, we have identified the inherent strengths and limitations of various methods. Based on our findings, we create a recommendation for practitioners on what method to choose based on the input characteristics.

ACKNOWLEDGMENTS

This research was supported in part by a grant from the Microsoft Jim Gray Systems Lab (GSL) and by the National Science Foundation (NSF) under grant CCF-2407690.

REFERENCES

[1] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, and A. Inkeri Verkamo. 1996. *Fast discovery of association rules.* American Association for Artificial Intelligence, USA, 307–328.

[2] A.N. Arslan. 2005. Multiple Sequence Alignment Containing a Sequence of Regular Expressions. In *2005 IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology*. IEEE, 1–7. https://doi.org/10.1109/cibcb.2005.1594922

[3] Robert S. Boyer and J. Strother Moore. 1977. A fast string searching algorithm. *Commun. ACM* 20, 10 (Oct. 1977), 762–772. https://doi.org/10.1145/359842.359859

[4] Chee-Yong Chan, Minos Garofalakis, and Rajeev Rastogi. 2003. Re-tree: an efficient index structure for regular expressions. *The VLDB Journal* 12 (2003), 102–119.

[5] Ting Chen. 2012. *Indexing text documents for fast evaluation of regular expressions.* Ph.D. Dissertation. USA. Advisor(s) Doan, Anhai. AAI3524345.

[6] Junghoo Cho and S. Rajagopalan. 2002. A fast regular expression indexing engine. In *Proceedings 18th International Conference on Data Engineering*. 419–430. https://doi.org/10.1109/ICDE.2002.994755

[7] Supawit Chockchowwat, Chaitanya Sood, and Yongjoo Park. 2022. Airphant: Cloud-oriented Document Indexing. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 1368–1381. https://doi.org/10.1109/icde53745.2022.00107

[8] Teodor Sigaev Christopher Kings-Lynne, Oleg Bartunov and Alexander Korotkov. [n.d.]. *F.35. pg_trgm — support for similarity of text using trigram matching.* Retrieved October 12, 2023 from https://www.postgresql.org/docs/current/pgtrgm.html

[9] Charles L. A. Clarke. 1996. *An algebra for structured text search.* Ph.D. Dissertation. CAN. Advisor(s) Cormack, Gordon V. AAINN15297.

[10] William W. Cohen. 2015. Enron Email Dataset. https://www.cs.cmu.edu/~enron

[11] Common Crawl Foundation. 2013. Common Crawl Winter 2013 Crawl Archive (CC-MAIN-2013-48). https://data.commoncrawl.org/crawl-data/CC-MAIN-2013-48/index.html

[12] Russ Cox. 2012. Regular Expression Matching with a Trigram Index or How Google Code Search Worked. https://swtch.com/%7Ersc/regexp/regexp4.html

[13] P.M.E. De Bra and R.D.J. Post. 1994. Information retrieval in the World-Wide Web: Making client-based searching feasible. *Computer Networks and ISDN Systems* 27, 2 (Nov. 1994), 183–192. https://doi.org/10.1016/0169-7552(94)90132-5

[14] Google. [n.d.]. *Google-RE2.* https://github.com/google/re2

[15] Gurobi Optimization, LLC. 2024. Gurobi Optimizer Reference Manual, version 12.0.0. https://www.gurobi.com

[16] Dan Gusfield. 1997. Introduction to Suffix Trees. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology* 89 (1997), 93.

[17] Bijit Hore, Hakan Hacigumus, Bala Iyer, and Sharad Mehrotra. 2004. Indexing text data under space constraints. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*. 198–207.

[18] T. S. Jayram, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and Huaiyu Zhu. 2006. Avatar Information Extraction System. *IEEE Data Eng. Bull.* 29, 1 (2006), 40–48. http://sites.computer.org/debull/A06mar/jayram.ps

[19] Samir Khuller, Anna Moss, and Joseph (Seffi) Naor. 1999. The budgeted maximum coverage problem. *Inform. Process. Lett.* 70, 1 (April 1999), 39–45. https://doi.org/10.1016/s0020-0190(99)00031-9

[20] Younghoon Kim, Kyoung-Gu Woo, Hyoungmin Park, and Kyuseok Shim. 2010. Efficient processing of substring match queries with inverted q-gram indexes. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE, 721–732. https://doi.org/10.1109/icde.2010.5447866

[21] H. W. Lenstra. 1983. Integer Programming with a Fixed Number of Variables. *Mathematics of Operations Research* 8, 4 (Nov. 1983), 538–548. https://doi.org/10.1287/moor.8.4.538

[22] Quanzhong Li, Bongki Moon, et al. 2001. Indexing and querying XML data for regular path expressions. In *VLDB*, Vol. 1. 361–370.

[23] Yinan Li and Jignesh M. Patel. 2013. BitWeaving: fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD/PODS'13)*. ACM, 289–300. https://doi.org/10.1145/2463676.2465322

[24] Yang Liu, Qianqian Xu, Peisong Wen, Siran Dai, and Qingming Huang. 2024. Not All Pairs are Equal: Hierarchical Learning for Average-Precision-Oriented Video Retrieval. In *Proceedings of the 32nd ACM International Conference on Multimedia (MM '24)*. ACM, 3828–3837. https://doi.org/10.1145/3664647.3681110

[25] Carsten Lund and Mihalis Yannakakis. 1994. On the hardness of approximating minimization problems. *J. ACM* 41, 5 (Sept. 1994), 960–981. https://doi.org/10.1145/185675.306789

[26] Jaina Mistry, Sara Chuguransky, Lowri Williams, Matloob Qureshi, Gustavo A Salazar, Erik L L Sonnhammer, Silvio C E Tosatto, Lisanna Paladin, Shriya Raj, Lorna J Richardson, Robert D Finn, and Alex Bateman. 2020. Pfam: The protein families database in 2021. *Nucleic Acids Research* 49, D1 (Oct. 2020), D412–D419. https://doi.org/10.1093/nar/gkaa913

[27] Sobhan Moosavi, Mohammad Hossein Samavatian, Srinivasan Parthasarathy, Radu Teodorescu, and Rajiv Ramnath. 2019. Accident Risk Prediction Based on Heterogeneous Sparse Data: New Dataset and Insights. In *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (Chicago, IL, USA) *(SIGSPATIAL '19)*. Association for Computing Machinery, New York, NY, USA, 33–42. https://doi.org/10.1145/3347146.3359078

[28] Yasushi Ogawa and Toru Matsuda. 1998. Optimizing query evaluation in n-gram indexing. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR98)*. ACM, 367–368. https://doi.org/10.1145/290941.291057

[29] Yorick Peterse. 2016. Fast Search Using PostgreSQL Trigram Text Indexes. https://about.gitlab.com/blog/2016/03/18/fast-search-using-postgresql-trigram-indexes/

[30] Tao Qiu, Zheng Gong, Mengxiang Wang, Chuanyu Zong, Rui Zhu, and Xiaochun Yang. 2025. *An Efficient Algorithm for Regular Expression Matching Using Variable-length-gram Inverted Index.* Springer Nature Singapore, 325–340. https://doi.org/10.1007/978-981-97-5779-4_22

[31] Tao Qiu, Xiaochun Yang, Bin Wang, and Wei Wang. 2022. Efficient Regular Expression Matching Based on Positional Inverted Index. *IEEE Transactions on Knowledge and Data Engineering* 34, 3 (March 2022), 1133–1148. https://doi.org/10.1109/tkde.2020.2992295

[32] John P. Rouillard. 2004. Real-time Log File Analysis Using the Simple Event Correlator (SEC). In *18th Large Installation System Administration Conference (LISA 04)*. USENIX Association, Atlanta, GA. https://www.usenix.org/conference/lisa-04/real-time-log-file-analysis-using-simple-event-correlator-sec

[33] Christian J. A. Sigrist, Edouard de Castro, Lorenzo Cerutti, Béatrice A. Cuche, Nicolas Hulo, Alan Bridge, Lydie Bougueleret, and Ioannis Xenarios. 2012. New and continuing developments at PROSITE. *Nucleic Acids Research* 41, D1 (Nov. 2012), D344–D347. https://doi.org/10.1093/nar/gks1067

[34] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. 2008. ArnetMiner: extraction and mining of academic social networks. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD08)*. ACM. https://doi.org/10.1145/1401890.1402008

[35] Dominic Tsang and Sanjay Chawla. 2011. A robust index for regular expression queries. In *Proceedings of the 20th ACM international conference on Information and knowledge management*. ACM. https://doi.org/10.1145/2063576.2063968

[36] P.M. Vaidya. 1989. Speeding-up linear programming using fast matrix multiplication. In *30th Annual Symposium on Foundations of Computer Science*. IEEE, 332–337. https://doi.org/10.1109/sfcs.1989.63499

[37] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 631–648. https://www.usenix.org/conference/nsdi19/presentation/wang-xiang

[38] Sun Wu and Udi Manber. 1992. Fast text searching: allowing errors. *Commun. ACM* 35, 10 (Oct. 1992), 83–91. https://doi.org/10.1145/135239.135244

[39] Chengcheng Xu, Shuhui Chen, Jinshu Su, Siu-Ming Yiu, and Lucas CK Hui. 2016. A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms. *IEEE Communications Surveys & Tutorials* 18, 4 (2016), 2991–3029.

[40] Chengxiang Zhai and John Lafferty. 2001. Model-based feedback in the language modeling approach to information retrieval. In *Proceedings of the tenth international conference on Information and knowledge management (CIKM01)*. ACM, 403–410. https://doi.org/10.1145/502585.502654

[41] Ling Zhang, Shaleen Deep, Avrilia Floratou, Anja Gruenheid, Jignesh M. Patel, and Yiwen Zhu. 2023. Exploiting Structure in Regular Expression Queries. *Proceedings of the ACM on Management of Data* 1, 2 (June 2023), 1–28. https://doi.org/10.1145/3589297

[42] Ling Zhang, Shaleen Deep, Jignesh M. Patel, and Karthikeyan Sankaralingam. 2025. An Evaluation of N-Gram Selection Strategies for Regular Expression Indexing in Contemporary Text Analysis Tasks. Extended Version. arXiv:2504.12251 [cs.DB] https://arxiv.org/abs/2504.12251