Regular Expression Indexing for Log Analysis

LING ZHANG, University of Wisconsin-Madison, United States
SHALEEN DEEP, Microsoft Jim Gray Systems Lab, United States
JIGNESH M. PATEL, Carnegie Mellon University, United States
KARTHIKEYAN SANKARALINGAM, University of Wisconsin-Madison, United States

In this paper, we present the design and architecture of REI, a novel system for indexing log data for regular expression queries. Our main contribution is an *n*-gram-based indexing strategy and an efficient storage mechanism that results in a speedup of up to 14× compared to state-of-the-art regex processing engines that do not use indexing, using only 2.1% of extra space. We perform a detailed study that analyzes the space usage of the index and the improvement in workload execution time, uncovering interesting insights. Specifically, we show that even an optimized implementation of strategies such as inverted indexing, which are widely used in text processing libraries, may lead to suboptimal performance for regex indexing on log analysis tasks. Overall, the REI approach presented in this paper provides a significant boost when evaluating regular expression queries on log data. REI is also modular and can work with existing regular expression packages, making it easy to deploy in a variety of settings. The code of REI is available at https://github.com/mush-zhang/REI-Regular-Expression-Indexing.

CCS Concepts: • **Information systems** → *Query optimization*; **Search engine indexing**; *Semi-structured data.*

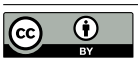
Additional Key Words and Phrases: regular expression; indexing; log analysis; query processing

ACM Reference Format:
Ling Zhang, Shaleen Deep, Jignesh M. Patel, and Karthikeyan Sankaralingam. 2025. Regular Expression Indexing for Log Analysis. *Proc. ACM Manag. Data* 3, 6 (SIGMOD), Article 355 (December 2025), 29 pages. https://doi.org/10.1145/3769820

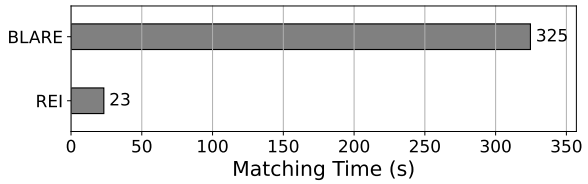
1 Introduction

Text processing for data analytics is a fundamental requirement for several applications such as system diagnostics, performance evaluations, security audits, business intelligence. Within text processing, log data processing is a common task that extracts information from semi-structured log entries from heterogeneous sources. A routine, but computationally expensive, log processing operation is regular expression (regex, for short) processing. In many scenarios (such as debugging production system errors, root cause analysis, etc.), it is important to get responses from the underlying regex engine in a timely manner. This challenge is further exacerbated by the ever-increasing size of the data that needs to be processed, where brute-force regex searching is unsatisfactory. Recent work [111] has introduced new insights on how to improve regex query evaluation for log analysis. BLARE, focusing on regexes in log analysis tasks, made the conscious choice of not using any indexes to ensure the wide applicability of their techniques. Indeed, [111] remarked that since the dataset may be used in an ad-hoc manner without any need for querying repeatedly, the cost

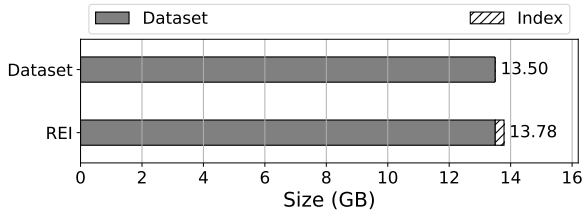
Authors' Contact Information: Ling Zhang, University of Wisconsin-Madison, Madison, WI, United States, ling-zhang@cs.wisc.edu; Shaleen Deep, Microsoft Jim Gray Systems Lab, United States, shaleen.deep@microsoft.com; Jignesh M. Patel, Carnegie Mellon University, Pittsburgh, PA, United States, jignesh@cmu.edu; Karthikeyan Sankaralingam, University of Wisconsin-Madison, Madison, WI, United States, karu@cs.wisc.edu.



ACM 2836-6573/2025/12-ART355
https://doi.org/10.1145/3769820



(a) Workload query matching time comparison.



(b) Size comparison of the dataset and index achieving best performance.

Fig. 1. Compared to the state-of-the-art regex matching framework BLARE, REI improves the performance by 14× on a production workload, with 2.1% extra space for the index.

of building indexes is not justified. However, if one hopes to achieve faster log processing, building indexes over the log data becomes necessary.

Log processing workload characteristics. Log processing workloads have several unique properties compared to other text processing tasks such as code search and document indexing. First, the smallest granular unit for logs is a log line whose length exhibits a skewed distribution. Most of the log lines are very small in length (up to 100 characters in our workloads) with very few lines containing many characters (as large as 2000). Consequently, the number of log lines are of the same order as the size of the logs. In contrast, for document indexing related tasks, each document is the smallest granular unit that needs to be retrieved for processing. The document size can range from a few bytes to several hundred MBs in size. Second, the sheer volume of log data even for simple applications can be huge. As an example, GitHub indexes [21] 15.5B documents across all its code repositories with total data size of 115TB. In contrast, for our log processing datasets, 15B log lines is generated by production applications in only about 5 hours. Third, the log dataset is usually generated by log-generating formatted code from different components of the system. Finally, unlike for search applications where retrieving top-k matches is the common setting (which allows for optimizations such as early termination of processing), the query workloads for log data require fetching all matches correctly.

There are three key parameters that need to be considered when constructing an index for regex log processing: the index construction time, the space used by the index, and the performance gain obtained when running the workload at hand. In particular, the choice of index data structure and entries for indexing has a substantial impact on the performance. Prior works mainly use inverted index suffix trees, and signature files as index data structures and n-grams of characters or words as index entries [5, 15, 31, 59, 74, 85, 103]. Novel variants of suffix trees are used for processing biological data, utilizing the small alphabet size (such as the limited character set for expressing DNA). Yet suffix tree indexes can easily occupy space more than 10× compared to the original dataset itself [82]. Signature files [31] use list or hierarchical lists of special signatures that encodes all words in each document. This method is less popular as inverted index presents a better running

time (but at the cost of using more space)[1]. Postgres uses inverted index of all trigrams for speeding up both exact and approximate text search [20]. Inverted indexes are also a popular choice that is used in frameworks such as Lucene [92] for text search. However, for both Postgres and Lucene, there is no option to control the space usage. As we will demonstrate later, inverted indexes for log processing can lead to significant space overhead ([31] noted a $50-300\%$ space overhead). Index size can be significantly reduced by reducing the number of entries indexed. Cho and Rajagopalan [19] proposed an algorithm called FREE that builds an inverted index over carefully chosen multigrams. This work was subsequently improved upon by Hore et al. [45], who also utilized multigrams but developed a more effective multigram selection strategy that formed the core of BEST.

Why existing solutions do not work for log indexing. Each of the above works have limitations. Both FREE and BEST require both the query workload and the data as a part of the input for selecting the multigrams. While the improved strategy of BEST helped the query performance, finding the near-optimal subset of multigram to index is significantly more expensive. Unlike document search based applications where building inverted and tree-based indexes requires less than 1% of the total input size[2] since the number of documents is much smaller than the total size of all the documents, for logs, the sheer number of log lines make it impractical to build tree-based indexes.

Our Contribution. In this paper, we revisit the problem of regular expression indexing and systematically analyze how classical approaches such as n-grams usage and inverted indexing behave when used with state-of-the-art regex evaluation engines on modern hardware. Our contributions are as follows.

1. *A new framework for regex indexing.* We present a lightweight indexing framework, REI, that is configurable to balance the cost of building indexes with query performance improvement. In contrast with existing approaches, our technique differs in two aspects. First, we use a filtering-based bit-vector index for a carefully chosen subset of strings of length n (called n-grams). Second, rather than storing the index in a separate data structure, we store the bit-vector index along with each log line of our input. The framework leverages the *negative index property*, where the absence of required n-grams in the index can quickly eliminate log lines from consideration without the need for expensive regex evaluation. This approach has several benefits, which we outline in Section 3.2.

2. *Thorough analysis of gram selection strategies.* We compare and contrast our proposal with prior works in the area of regex processing using indexes. We empirically demonstrate that existing approaches fall short and lead to suboptimal performance compared to our approach. Regex queries in log analysis workloads tend to have relatively low selectivity as they mostly contain queries that are finding *needle-in-a-haystack* [30, 101, 102, 110]. We demonstrate that for such workloads, it is sufficient to use the query workload itself to choose the right set of n-grams, rather than finding the optimal set of n-grams.

3. *Adaption to Unknown Queries.* We propose a strategy for constructing an index when the query set is unknown. Existing key selection methods struggle with unknown queries in log processing workloads. Our system adapts by using similar index key selection guidelines, leveraging log processing characteristics. This strategy significantly reduces runtime and is comparable to an index built with a known query set.

[1] [31] noted that signature files method requires space overhead of about $5-30\%$ of initial file sizes but this overhead is still too large for our use-cases. In fact, 5% is the threshold on the space usage based on our discussions with product teams at X.
[2] Using GitHub as an example, a complete binary tree with $\sim 15B$ leaf nodes and each node storing 10 bytes of information is 0.26% of the 115TB input size.

Table 1. Summary of symbols and notation used in this paper.

Symbol	Meaning	Symbol	Meaning
Σ	Finite alphabet	W	Workload consisting of Q, L
R	Regular expression	q	Individual query
L	Log dataset	n	Length of n-grams
Q	Set of regex queries	k	Number of n-grams indexed
I	Index built	g	Individual n-gram
ℓ	Individual log line	`literals`	Literal components

4. *Experimental Evaluation.* To demonstrate REI's efficiency, scalability, configurability, and robustness, we tested it on three real-world workloads. Our analysis shows REI consistently outperforms popular schemes like inverted indexes in both construction overhead and performance. It improves the matching time of BLARE by up to 14× with low space overhead (Figure 1). Additionally, REI shows significant gains even without prior knowledge of regex queries by utilizing the log processing workload characteristics.

Overall, our experiments and analysis demonstrate that REI is a simple yet effective regex indexing framework, and showcases its potential to significantly enhance regex query performance in real-world log analysis tasks.

2 Background

Regular Expression. We define regular expressions over a finite alphabet Σ using syntax such as the empty language ($\emptyset$), empty string (ε), literals (character $c \in \Sigma$), union of two languages ($|$), concatenation of two languages ($\cdot$), and the Kleene Star (zero or more concatenations of the language, denoted by $*$). The syntax allows us to represent a wide variety of string patterns, with the language denoted by a regular expression defined through a function $L : R \rightarrow 2^{\Sigma*}$. The syntax of regex is as follows:

$$R : \emptyset \mid \varepsilon \mid c \mid (R \mid R) \mid (R \cdot R) \mid (R*)$$

Literal Component. It is also important to consider literal components, which is a major source for indexing, in regular expressions. Literal components are sequences of characters from the alphabet that match the input string exactly as they appear, without any interpretation as operators or special characters. Formally, we denote them as $\texttt{literals} \leftarrow \Sigma*$. They are distinct from the regular expression components, which include operators like $*$, +, ?, |, and other defined patterns. Literal components in regexes are especially relevant in log processing workload, as the regexes in such workload often contains literals that are more selective than their regex components. The concept of `literals` has been used in several prior works [100, 111], in which the authors utilize the fact that exact character matching is much cheaper than state transitions in an automaton. Hyperscan [100] evaluates components in a sequential fashion together with other optimizations. For each regex, BLARE [111] selects a specific proportion of the input data and uses machine learning techniques on the picked subset to determine the best regex-literal splitting strategy for each query, and then evaluate the rest of the dataset with the chosen strategy. It usually evaluates all literal components before regex components.

***N*-Grams.** An n-gram (also referred to as a q-gram in the literature) is a subsequence of length n. An n-gram generation from a string s of length c refers to constructing the set of subsequences of length n for each starting position $i \in \{0, \ldots, c - n\}$. Given a string of length c, there are $c - n + 1$ n-grams that can be generated from s. For example, given a string *vmName* and $n = 3$, there are four

n-grams that can be constructed: *vmN*, *mNa*, *Nam*, and *ame*. If $c \leq n$, then the n-gram generation of s is s itself. The most commonly used n-grams in practice are bigrams ($n = 2$) and trigrams ($n = 3$). Mixed n-grams have also been used in prior works.

N-Gram Selection. There have been several optimizations proposed in the literature for improving the space efficiency of an inverted index. For example, FREE [19] shows that there is an inverse trade-off between the increasing value of n for generating the n-grams and the size of the inverted index. BEST [45] reduces the gram selection problem to graph covering problem, and select the multigram set with approximately (with provable guarantees) highest *benefit* (which will be discussed in Section 3.1). LPMS [96] combines strategies of FREE and BEST and uses linear programming to approximate the optimal multigram set. These methods require full knowledge of the dataset and the query before selection and the calculation is very time consuming.

GitHub Code Search [21] reduces the size by using multigrams starting from bigrams for its generality. It limits the number of multigrams with $n > 2$ by assigning weights to every bigram and creates indices for multigrams formed by adjoining bigrams. For these multigrams, the weights of the internal bigrams are strictly lower than those of the bigrams at both ends.

Index Data Structure. An n-gram index is a data structure used to determine if a subsequence identical to a given string s exists in an input ℓ. Google Code Search [25] introduced a common method for constructing an n-gram index for document search, creating an inverted index with trigrams as keys and lists of document identifiers as values. This allows for finding documents containing all trigrams generated from s and supports disjunctions, such as finding documents containing either *abc* or *def*. Baeza-Yates and Gonnet [5] introduced suffix tree indexes for regex evaluation, which store all possible suffix strings in a dataset. The large space requirement makes them only practical in protein datasets with limited alphabets. Bit-vector or bitmap indexing is popular for its fast bitwise operations and performance benefits for highly selective queries [17, 18]. This type of index associates one bit-vector with each record, suitable for general selection queries across various data types. For text data, methods like superimposed signature files [31, 38, 83, 104] and bloom filters [41] encode documents into bit-vectors, providing effective prefiltering but with significant space overhead.

3 Framework and Techniques

In this section, we present the design and techniques used for our indexing framework. The framework consists of two steps. The first step is the extraction of k bigrams from the query set and ordering them based on their frequencies. The second step involves the construction of an n-gram index. Using the bigrams extracted in the first step, we construct a bit-vector filter consisting of k bits for each line in the input log. The i^{th} bit in the filter for a log line ℓ indicates whether the i^{th} bigram is present in ℓ or not. In the querying phase, given a regular expression, we first find the string literals in the regular expression, generate n-grams for each string literal, and use the n-gram index to check if all n-grams for a specific literal exist in the index. We will discuss in details in the following subsections.

3.1 N-Gram Selection

The choice of n-grams for the index is crucial to the effectiveness of the indexing framework when matching regular expressions over large log datasets. It is influenced by four components: queries Q, dataset L, the choice(s) of n, and the number of chosen n-grams (k). Prior works have made mixed choices of n or have chosen a fixed sized n. We made the choice of only using bigrams (i.e. $n = 2$). Bigrams strike an optimal balance between being general enough to appear across diverse queries while maintaining sufficient filtering power. As n increases, n-grams become more specific

to particular queries, reducing their ability to filter across a broad range of regex patterns. We demonstrate it experimentally in in the Appendix C [113] with a detailed comparison between bigrams and trigrams. Our key findings are that as n goes beyond two, the fraction of bigrams that are a subset of n-grams also increases, and thus, with high probability, the bigrams generated from the larger n-gram are as selective as the bigger n-gram itself.

Recent work compared across the existing n-gram selection methods (FREE, BEST, and LPMS) both theoretically and experimentally [112]. It runs the three methods on different types of workloads and summarize the insights into a general guideline to select among the three selection methods according to the workload characteristics. For our workloads where unseen queries may be expected, query set is large with long query literals, FREE performs the best. On the other hand, BEST can generate the theoretically optimal n-gram set for indexing.

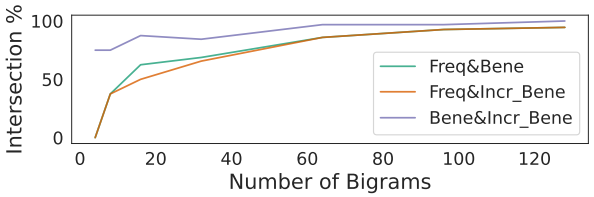
Therefore, we compare against the strategy of choosing n-gram as introduced by the BEST and FREE algorithms. FREE rus with space overhead $O(max_n \cdot |L|)$ and the compute overhead as $O(max_n \cdot |L|)$ where max_n is the configurable maximum length of n-grams. BEST finds k n-grams that theoretically maximizes the effectiveness in time $O(k \cdot |Q| \cdot |L|)$ and uses $O(|Q| \cdot |L|)$ space.

The BEST [45] method introduces the concept of *benefit* and *cover* to measure the effectiveness of n-gram selection. The *cover* of an n-gram g on workload $W = (L, Q)$, $cover(g) = \{(q, l) \in Q \times L \mid g \in q \wedge g \notin l\}$, is the set of log lines that can be filtered out by g among all queries. The *benefit* can be expressed as $bene_W(g) = |cover(g)|$. FREE selects grams by choosing the n-grams with the highest benefit, in order of increasing n (i.e., smaller n is preferred). The *benefit* of index I on a workload denoted as $bene(I) = |\bigcup_{g \in I} cover(g)|$, is the number of log lines that can be filtered out by the index I. Therefore, the incremental benefit given an index is $bene(g|I) = |cover(I \cup \{g\}) - cover(I)|$. This metric captures both the filtering power of the n-gram and its relevance to the query workload, considering other n-grams in the index. BEST reduces the gram selection to a graph covering problem and selects the multigram set with provable guarantees on the benefit.

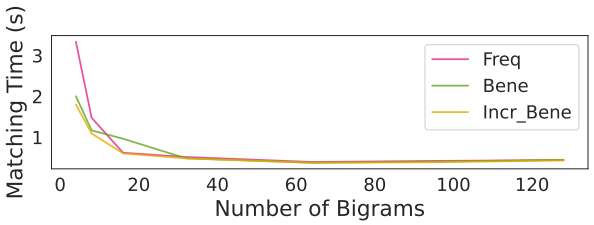
We propose an n-gram selection strategy that runs in time $O(|Q|)$ with negligible space overhead by selecting based on n-gram frequency in the query workloads. This is significantly smaller than both FREE and BEST in that it focuses on query-relevant n-grams rather than dataset-based selectivity measures. The intuition is that frequently occurring n-grams in queries are more likely to be useful for filtering across diverse log analysis tasks. Our n-gram selection strategy works as follows: For each query q in the workload Q, we first preprocess the query to extract literal components, then scan these literals from left to right to identify all bigrams present in the query. For each bigram discovered, we update a global counter dictionary that tracks the number of queries containing that specific bigram. After processing all queries, we rank bigrams by their frequency (number of queries containing them) and select the top-k most frequent bigrams for indexing.

In order to show the cost and effectiveness of n-gram selection strategies, we run a micro-benchmark that builds and uses indexes selected with 3 different strategies with low index overhead. We take the simplified concept of *benefit* from BEST, defined in [112]. Let g be an individual bigram. The three are: (1) the most frequently occurring bigrams (Freq), (2) bigrams with the highest $benefit(g, \emptyset)$ (Bene), and (3) selected bigrams g_i based on $benefit(g_i|\{g_{i-1}\})$, where g_j ranks according to $benefit(g_j, \emptyset)$ (Incr_Bene). We run the three methods on a subset of a real-world workload containing one million log lines and 132 queries.

To develop some intuition about how different the bigrams chosen by each of the strategies are, we look at their overlap in their selected bigrams when selecting the top 4, 8, 16, 32, 64, 96, and 128 bigrams and present the *intersection percentage* in Figure 2a. The *intersection percentage* refers to the percentage of common bigrams selected by different strategies. We use it to measure the overlap between the output sets of different selection methods. With the selection by *Incr_Bene* closer to



(a) Bigram intersection as a percentage from strategy pairs.



(b) Regex Matching Time.

Fig. 2. Varying the number of bigrams, compare the set of bigrams selected by the three methods and the matching time applying their resulting indices.

optimal selection, a higher intersection percentage with *Incr_Bene* shows a higher performance of the selection method. There is a notable similarity in the bigrams selected by Bene and Incr_Bene. The intersection percentage is high at 75% with only 4 and 8 bigrams and steadily increases to 96.9% for 96 bigrams and 100% for 128 bigrams. We observed that conditional on the top-100 selected bigrams, no other remaining bigrams can generate a positive incremental benefit. As the number of chosen bigrams increases from 4 to 64, the intersection percentage of Freq with the other two methods gradually increases to 85.9%. Their rates of intersection plateaus as the number of bigrams selected continues to increase.

In addition, we evaluate the matching performance using the indices derived from the three methods and present the results in Figure 2b. The matching time is quite different when the number of bigrams selected is small, where using the index built with bigrams selected by Freq gives the slowest matching time. The times gradually converge for the indices built with the three methods as the number of bigrams increases. The running time reaches a minimum when indexing with 64 bigrams, where they are 0.395, 0.376, and 0.376 for Freq, Bene, and Incr_Bene respectively.

We also evaluate the time needed to choose bigrams for each approach. The ranking calculations for frequency and benefit are finished in 0.0014 seconds and 21.5 seconds, respectively, for selecting 64 bigrams. In sharp contrast, the Incr_Bene technique requires hours (31,638.2 seconds) when only taking into account the bigrams with top-200 individual benefit. The computational time increases even more when the Incr_Bene approach is applied to the entire dataset, taking well over 24 hours to complete[3]. When choosing bigram selection strategies for indexing, there are trade-offs between filtering power of selection result and computational efficiency.

Based on this microbenchmark, we find that selecting bigrams based on frequency is the most effective strategy for log analysis workloads, offering significantly lower overhead than the proposed method FREE, while yielding results comparable to the most precise method BEST.

3.1.1 Considerations for N-Gram Selection. In this subsection, we outline some of the considerations for *n*-gram selection.

[3]We terminate the experiments at 24 hours.

Frequency Threshold for N-Grams Selection. Prior work has proposed heuristics such as removing n-grams that appear in more than 10% of queries, as these are typically pruned by algorithms when identifying the optimal covering set of bigrams. We chose not to adopt this strategy due to differences in index use cases. Our selected n-grams are intended not only to enhance matching performance for the current set of queries and dataset but also to improve performance for potential future queries on new inputs that may be added to the log.

Number of N-Grams Indexed and Choosing k. The choice of k is non-trivial. Assuming we know the ideal k, our microbenchmark shows that selecting the k most frequent n-grams from the workload is sufficient. This works because regex queries typically exhibit *needle-in-a-haystack* behavior with very low selectivity, largely driven by string literals [30, 101, 102, 110]. Thus, focusing on literals when picking n-grams is effective. Increasing k improves coverage but offers diminishing returns beyond a point.

The size of a bit-vector index is determined by the number of bigrams k we choose to index. This fixed size ensures predictability in storage requirements, making memory allocation and management more straightforward. We can set the value of k to multiples of word size such that no space is wasted due to alignment or padding. When the size of the query subset in the workload is small and expected to not be a perfect representation of all future queries, consider increasing k. This will allow indexing some bigrams that appear less frequently in the query subset at hand, but may appear in future queries.

In practice, users may want to specify size constraints for the index. We propose to automatically suggest optimal values for k and index granularity under given size constraints by using heuristics involving bigram selectivity analysis. We provide a prototype `REI-Tuner` with heuristic estimation of filtering benefit per bit and present initial experimental results in the Appendix D [113].

3.2 Index Construction

Once the bigrams have been selected and ranked according to their frequency, we proceed to scan the dataset. For each log line in the dataset, we get all its bigrams and identify the presence of each of the selected k bigrams. This information is encoded as a k-bit vector where the i^{th} bit indicates if the i^{th} bigram is present in the log line or not.

Algorithm 1 shows the execution steps for building the index. Let $\mathbf{b} = (b_1, b_2, \cdots, b_k)$ be the ordered sequence of the selected bigrams. We use $\mathbf{G}$ as one-to-one mapping from each bigram b_i to its offset i in the bit-vector. First, for each log line ℓ, we use procedure Get-Unique-Bigrams to do the bigram generations for ℓ. Then, given the bigrams, we create the bit-vector in procedure Set-BitVector. The resulting bit-vector for the processed log line is added to a list I that stores the bit-vectors, which is the final index. Observe that both the time requirement and the space requirement of the index are $O(k \cdot |L|)$.

Discussion. Beyond bit-vector based index, signature files based index, and inverted index are alternate methods techniques for building index. In fact, [38] showed that signature files inspired bit-vector based methods can outperform inverted indexes contrary to conventional belief. In Section 4.6, we experimentally demonstrate that for log processing, carefully crafted bit-vector based methods outperform signature files and inverted indexes.

3.2.1 Index Granularity. Index granularity defines how many log lines each index entry covers. Our framework creates one bit-vector for every m consecutive lines, where each bit marks the presence of a specific bigram in that group (note that Algorithm 1 considers $m = 1$, i.e., each log line has a bitvector). For example, with $m = 4$, a single bit-vector represents four lines; if any line

Algorithm 1: BitVector-Index-Construction

```
Input : Workload W with logs L and regexes R, k
I ← []
G ← Select-Bigram(R, k)
foreach ℓ ∈ L                                              /* processing each log line */
do
    S ← Get-Unique-Gram(ℓ, 2)
    b ← Set-BitVector(S, G)
    I.append(b)
return I
procedure Get-Unique-Gram(s, n)
    S ← hash map
    for i ∈ {0, ..., size(s) − n} do
        g ← s[i : i + n − 1]
        if ¬S.contains(g) and g is a string literal then
            S.put(g, 0)
        S.put(g, S.get(g) + 1)
    return S
procedure Select-Bigram(R, k)
    T ← hash map
    foreach r ∈ R do
        C ← Get-Literal-In-Regex(r)
        T_i ← hash set
        foreach c ∈ C do
            T_i.add (Get-Unique-Gram(c, 2).keys())
        foreach g ∈ T_i do
            if ¬T.contains(g) then
                T.put(g, 0)
            T.get(g) ← T.get(g) + 1
    Sort T.keys() by value in descending order into vector V
    G ← hash map
    for i ∈ {0, ..., k − 1} do
        G.put(V[i], i)
    return G
procedure Set-BitVector(S, G)
    b ←bit vector of length size(G)
    /* Compare sizes of the two hash-maps and decide the probing map                */
    if size(S) < size(G) then
        foreach key ∈ S do
            if G.contains(key) then
                b[G[key]] ← 1
    else
        foreach key, val ∈ G do
            if S.contains(key) then
                b[val] ← 1
    return b
```

Algorithm 2: BitVector-Index-Query

```
Input :Log L, Index I, Regex r, n, hash-map G from n-gram to offset
m ← Get-Bitmask(r, n, G)
allOne ← k bit array with all bits set to 1
for j ← 0 to size(L) do
    if I[j] ∨ m = allOne then
        do Regex-Match(L_j, r)
procedure Get-Bitmask(r, n, G)
    T ← Extract-Literals-In-Regex(r)        /* T is the list of string literals in the regex */
    S ← ∅                                   /* empty set */
    foreach t ∈ T do
        S_t ← Get-Unique-Gram(r, n)
        S.add(S_t)
    m ← Set-BitVector(S, G)
    m ← ¬m                                  /* flipping all the bits */
    return m
```

contains a bigram, its bit is set to 1. During queries, if all required bigrams appear in the bit-vector, the group is scanned line by line using regex.

The choice of granularity involves important trade-offs:

- **Finer granularity** (smaller m): Higher filtering precision but larger index size and higher construction cost.
- **Coarser granularity** (larger m): Lower storage and computation cost but more false positives.

The optimal setting depends on log characteristics, query patterns, and resource constraints. `REI` supports configurable granularity to balance precision and efficiency.

3.3 Regular Expression Querying

Leveraging the created index, when querying over the dataset with a regular expression r, we build a bit-mask for the literal components of r and filter out part of the logs that do not match. We present the steps for querying a regex on the dataset with a bit-vector index in Algorithm 2. When a regular expression query is received, it is first parsed to generate the subset of bigrams that are used in the index. Then, we construct a bit-mask of size k, placing a bit value of 1 in the i^{th} index to denote that a bigram is *not* present in the regex. In particular, if a bigram g that is used in indexing but is absent in the regex, we set $G[g]^{\text{th}}$ bit of the mask to 1, and 0 otherwise.

We scan through the array of bit-vectors and do a bitwise OR operation for each entry with the bitmask of the regex. The log line passes the index's initial filter if the outcome of the OR operation is a k-bit vector of all 1's. Here, we leverage the negative index property: a set bit in the bitmask indicates absence of a bigram and thus, its presence or absence in the log line is inconsequential. The OR operation effectively verifies this condition. Observe that for bigrams that are not used in indexing, we always need to read the log line. The index does not produce false-negatives, and we include the proof in in the Appendix A [113] of the full paper. Our implementation use the C++ `all()` function to check if all bits are set. This is done to implicitly capture the negative index property.

An example of the querying process is illustrated in Figure 3. We use four bigrams in the index (g_0 to g_3). Suppose the query q has the bigrams g_0, g_1, and g_3. Then, the bitmask generated will be 0010. Only the bit-vector of log lines four and five lead to a vector of all 1's when a bitwise OR

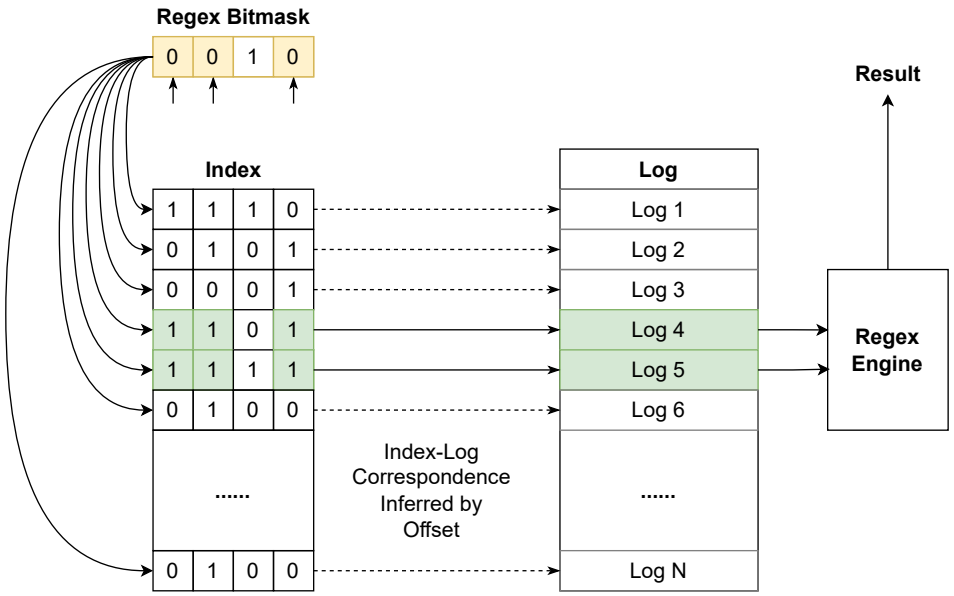


Fig. 3. Query overview using a bit-vector index with $k = 4$.

operator is performed with 0100. In Figure 3, the log lines corresponding to index entries with all relevant bits set to 1 (colored green) pass the filter.

Only the log lines whose bit-vector representations match the derived bigrams are considered for further processing. This significantly reduces the number of log lines that need to be processed, ensuring efficiency. For the log lines that pass this initial filtering, a full regex matching process is initiated (line 5). We employ the Google-RE2 [39] regex library for this purpose, given its robustness and efficiency in handling complex regular expressions.

3.3.1 Unknown Workload And Future Queries. In the previous section, we discussed the effectiveness of `REI` given a fixed known workload. However, `REI` can also help with a new set of queries on the same set of log data or an unknown workload. This is particularly relevant for ad-hoc querying scenarios where analysts may need to search logs with previously unseen regex patterns.

`REI` addresses this challenge by leveraging a key characteristic of log analysis workloads: both logs and queries contain substantial amounts of human-readable text. Log analysis queries typically aim to extract variable values (e.g., using `\d+` to extract numeric vmID) while filtering based on literal English text components (such as `vmID=`). This suggests that bigrams in log analysis workloads have distributions similar to English text, and that bigrams selected from English literature have a high probability of appearing in future log analysis queries.

Our strategy for unknown workloads involves using the most frequent bigrams from English literature as index keys. This approach significantly reduces the computational overhead of index construction while maintaining effectiveness comparable to indexes built with known query sets. The limitation of this strategy arises when the distribution of literal components in the query set significantly diverges from that of natural English language. This is especially true when the unknown query set exhibits domain-specific vocabulary. Therefore, `REI`'s performance on unknown workloads is most reliable when they retain sufficient overlap with general human-readable English patterns. We demonstrate the effectiveness and limitations of this strategy in Section 4.4.

Table 2. Statistical information of bigrams in the literal component of regular expression queries in the workload.

Workload	# Total Bigrams	% Occurrence in Regexes				
		min	Q1	mid	Q3	max
DB-X	762	0.8	0.8	3.0	11.4	89.4
Sys-Y	138	5.9	5.9	11.8	17.6	41.2
US-Acc.	18	25.0	25.0	25.0	50.0	75.0

4 Evaluation

We implemented REI using C++ and used the state-of-the-art regular expression library, Google's RE2, for any exact regex matching after passing the index checking. We also implemented an inverted index and signature files under the same setting. For comparison purposes, we implement posting lists and inverted indexes in-memory using hashmaps, where each n-gram serves as a key and maps to a list (or set) of identifiers indicating which log lines or groups contain that n-gram. During query processing, the relevant posting lists are retrieved and intersected to find candidate log lines that contain all required n-grams. We use these terms interchangeably throughout the section. We want to answer the following questions:

- **Q.1** How does the chosen value of n for n-grams of the index affect the overhead of constructing the index and the improvement of performance (cf. Section 4.2)?
- **Q.2** How does the chosen value of k (the number of n-grams used in the index) affect the overhead of constructing the index and the improvement of performance (cf. Section 4.3)?
- **Q.3** How does the index impact query performance when we have an unknown log analysis workload (cf. Section 4.4)?
- **Q.4** How does the granularity of the index affect the overhead of constructing the index and the improvement of performance (cf. Section 4.5)?
- **Q.5** How does REI compare with other commonly used indexing schema on the overhead of constructing the index and the improvement of performance (cf. Section 4.6)?

4.1 Experiment Setup

In our experiments, we construct indices using bigrams, trigrams, and 4-grams. The index is physically represented by a k-bit vector using a C++ bitset for each log line. The bit-vector indicates the presence of the top k n-grams. For regex matching, the regular expressions are first parsed to derive all n-grams intersecting with the n-grams used in the index. The n-grams in the intersection are encoded in a length k bit mask, facilitating the initial filtering of log lines using the index. Only the log lines that pass the filtering are used as input for the full regex matching process using RE2 [39] which is widely used in the industry. We use BLARE [111] with RE2 as the backend supporting regex engine, as the performance baseline , which is faster than RE2 without indexing. For simplicity, all experiments are single-threaded and in the main memory setting.

4.1.1 Workloads. Our experiments employ three real-world datasets and their respective query workloads. Two datasets include logs produced by production database systems, whereas the third is text-oriented. For each workload, detailed statistical information about the literal elements in regexes is shown in Table 2.

Database X Workload. A 13.5 GB dataset with 8,941 regexes and 101,876,733 log lines from DB-X was obtained from a well-known cloud provider. These data were utilized in log analysis tasks by various data scientists and engineers. When examining all the regex queries in the original DB-X workload, the majority of the queries had no matches on the dataset, which represents logs from a

specific time duration. Thus, for a realistic representation of the workload, we consulted an expert data scientist working in this domain and sampled regexes favoring those with matches on the dataset. For completeness, we report the experimental results for the entire set of 8,941 regexes in Section 4.7. We picked 132 regexes from their workloads and evaluated the regexes on the dataset to construct `Database X` (**DB-X**). In this workload, a log line has an average of 138.5 characters. The regexes used in this task are characterized by an average of 105 literal characters per regex and 39.9 characters per literal component. From the literal components of the regexes, 762 unique bigrams were extracted, with the majority appearing in just one regex.

`System Y` Workload. This dataset is 100 GB in size and consists of 890,623,051 log lines generated by System-Y, a production data exploration tool. The associated workload includes 17 regexes used by data scientists for analysis tasks. The log lines have an average length of 116.2 characters. We also use the shorthand **Sys-Y** to denote this workload. Compared to **DB-X**, the regexes in this workload often have fewer literal components per regex and shorter literal components overall. We get 138 unique bigrams from the literal components of the regex queries. Similarly, most of the bigrams are unique and appear only in one regex.

`US-Accident` Workload. The dataset, sourced publicly and collected by Moosavi et al. [69], has 1.08 GB of data and includes 2,845,343 records of traffic accidents that occurred in the United States between February 2016 and March 2019. For our objectives, we used the 4 regexes described in their paper along with the *accident description* strings from the dataset. There are 66.6 characters on average per line. We extracted 18 unique bigrams from the query set, and among them, slightly more than half occurs in only on regex, and one of the bigrams occurs in 3 regexes.

4.1.2 Methodology. We compare a number of metrics, such as index building time, index size, and regex matching execution time, across various index configurations. The time required to extract and select the n-grams and then build the index is used to compute the index construction time. We measure the bit-vectors' overall size, including any padding, as the size of the index. The cumulative time required for each regex in the workload to match against all log lines in the dataset is measured for determining the workload running time, using indices when applicable. The runtimes are calculated using a trimmed mean of 10 runs, which leaves out the greatest and lowest runtimes.

Indices were created using the bigram, trigram, or 4-gram that appeared the most frequently in the literal components of the workload queries. Depending on the workload and experiment design n-grams with n varying from 4 to 512 may be used to build the index. Additionally, indices were created with index granularity of 1, 8, 64, 192, 256, or 512 lines. Detailed methodologies for individual experiments will be described in the respective subsections.

4.1.3 Hardware. We run all experiments on an Azure Standard_E32-16ds_v5 machine with Intel(R) Xeon(R) Platinum 8370C CPU @ 2.80GHz with 16 vCPUs, 256 GB memory, and 1 TB hard disk. Our experiment code is written in C++17 and compiled with the -O3 flag. We used RE2 (release version 2022-06-01) and the version of BLARE (state-of-the-art system for regex processing over log data) retrieved on 09-01-2023.

4.2 Type of N-Gram

In this section, we address **Q.1** by evaluating our strategy through a comparison of different types of n-gram used for index building. Specifically, we compare the query performance improvement and index construction overhead in terms of space and time. The top 64 most frequent n-grams, ranging from bigrams to 4-grams, were extracted from the regular expression queries of `Database`

Fig. 4. Comparing the impact of different types of n-grams on index construction time and matching time on indexes with the top 64 most frequent n-grams in workload queries.

Table 3. Comparison of mean percentage of log lines processed by regex matching engine after filtering with bit-vector index for different types of n-grams used in the index.

n	2	3	4
Log % filtered through the index	0.63	0.58	4.99

`X` workload. For each log line, a bit-vector of size 64 was constructed to denote the presence (or absence) of these n-grams.

4.2.1 Index Construction Overhead. The bit-vector index size remains constant across different types of n-grams, as each n-gram is represented in an index entry with one bit regardless of the value of n. Specifically, the size depends solely on the number of n-grams, k, ($k = 64$ in this case) and the total number of log lines. Consequently, for a fixed k, the space overhead of bit-vector index is directly proportional to the number of log lines.

In Figure 4a, we present a summary of the index construction time for the `Database X` workload, comparing bigrams, trigrams, and 4-grams. For bigrams, the index took 511.7 seconds to build. Bigrams are short and simple, and their extraction and subsequent indexing are fairly quick. Building the trigram index took slightly longer, at 589.6 seconds. The 4-grams, being the longest among the three types of n-grams, took the longest time to construct the index at 612.7 seconds. This shows that the length of the n-grams subtly impacts the overall index construction time. The subtle difference in time for different n can be traced back to the number of unique n-character sequences extracted from the log and the hashing strategy of character tuples. As the length of the n-grams increases, the number of unique n-grams also increases. This translates to an increase in the number of hashmap probing and higher probability of hash collision (recall that our index contruction algorithm uses hashmaps on line 10 and line 18 of Algorithm 1), and consequently, an increase in the time required for index construction.

4.2.2 Matching Time with Index. To compare the query efficiency of our indexing approach with varying n-gram sizes, we benchmark the performance of regular expression matching on `Database X` workload using indices constructed with bigrams, trigrams, and 4-grams. We report the time taken to match all 132 regexes against the log dataset using each n-gram index type. From Figure 4b, we found notable differences in matching times. Using the bigram index, the full set of workload queries runs in 34.5 seconds. The trigram index results in a slightly slower matching time of 35.5 seconds. The matching process takes significantly longer with the 4-gram index, at 92.4 seconds,

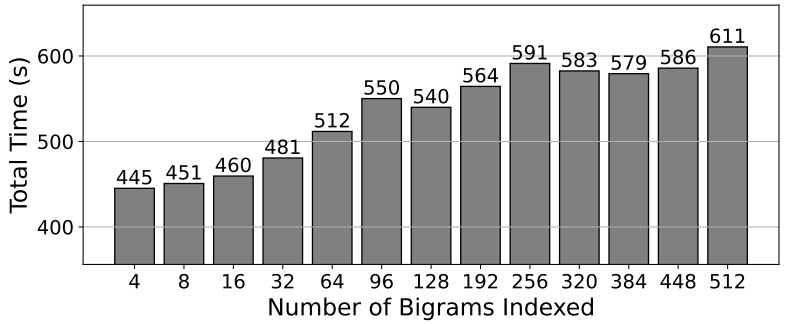


Fig. 5. Comparing the impact of different numbers of n-grams on index construction time of the indices. Uses top-k most frequent n-grams in workload queries.

Table 4. Comparison of the size of the bit-vector index for different k values.

k	4 - 64	96 - 128	192	256	320	384	448	512
Size (GB)	0.8	1.5	2.3	3.0	3.8	4.6	5.3	6.1

compared to the bigram and trigram indices. Despite the increase in time with 4-grams, all indexed methods were substantially faster than the baseline of 324.6 seconds, which is the time taken to match using the state-of-the-art regular expression matching framework without using an index.

After leveraging the indices to filter potential matches, we summarize the mean percentage of log lines that remain for each regex in Table 3. The bigram and trigram indices leave 0.63% and 0.58% of log lines respectively, showing similar filtering efficacy. The percentages suggest that trigram index has slightly higher actual *benefit*, but both bigrams and trigrams are effective in the task. This is reflected in their relatively similar matching times of 34.5 and 35.5 seconds. However, the 4-gram index retains a larger subset of log lines at 4.99%, signifying a much lower actual *benefit*, despite its longer construction time. This larger data pool for regex matching results in a longer matching time of 92.4 seconds, indicating a direct correlation between the volume of data sent to the regex engine and total matching time. The choice of n-gram length can significantly impacts the construction overhead and the performance of the subsequent regex matching process.

4.3 Number of N-Gram

In this section, we answer **Q.2** by comparing the number of n-grams used for index building. Specifically, we compare the performance gain and index construction overhead across using the most frequent k n-grams in `Database X` workload, where choices of k are 4, 8, 16, 32, 64, 96, 192, 256, and 512. We use bigrams, trigrams, and 4-grams for all k. Similar to the previous experimental setting, one bit-vector of size k in the index corresponds to one log line.

4.3.1 Index Construction Overhead. In this subsection, we evaluate the impact of the number of n-grams, k, on the index size and the construction time. The number of bigrams selected determines the length of each bit-vector representation for a log line, thereby directly affecting the size and construction time of the index.

Table 4 presents the comparison result of index sizes for different numbers of bigrams used. Due to word-size and padding in C++ bitset, the index size stays constant when indexing with 4 to 64 bigrams. Packing multiple index entries together can further reduce the index size for storage and loading. The index size grows from 0.8 GB for $k = 64$ to 1.5 GB for $k = 512$. This trend shows that

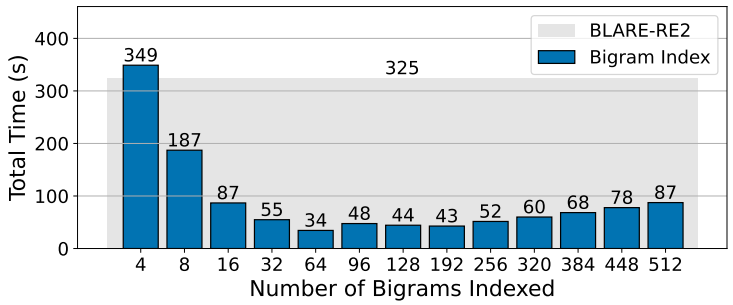


Fig. 6. Comparing the impact on matching time for different numbers of n-grams indexed. Uses top-k most frequent n-grams in workload queries.

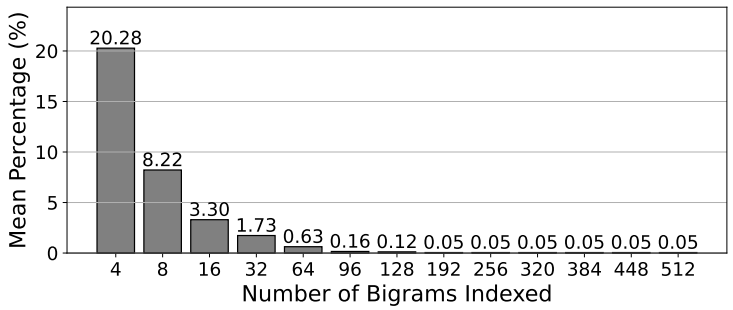


Fig. 7. Comparing the mean percentage of log lines processed by regex matching engine after filtering with bit-vector index. Uses top-k most frequent n-grams in queries.

integrating more bigrams into the index encapsulates more information, increasing the filtering power by eliminating more unmatched log lines, at a cost of more space. We also analyze the index construction time for various numbers of bigrams, as depicted in Figure 5. Construction time marginally increases from 445.2 seconds with 4 bigrams to 610.6 seconds with 512 bigrams. Thus, a 100× increase in the number of bigrams results in just a 30% increase in index construction time. The slight variation in index construction time, despite the increase in bigrams, highlights the low cost of the bit-setting operation and REI's ability to balance the filtering power with index construction overhead.

4.3.2 Matching Time with Index. Now we compare the overall matching time for the Database X workload across different numbers of bigrams used to construct the bit-vector index. Figure 6 shows the regex matching time corresponding to each k, the number of bigrams used.

The baseline, BLARE-RE2, which operates without any indexing, has a matching time of 324.6 seconds for the given workload. An index constructed with 4 bigrams results in a matching time of 348.9 seconds. The matching time drops to 187.1 seconds with an index built with 8 bigrams and further decreases to a minimum of 34.5 seconds when using 64 bigrams in the index. Beyond 64 bigrams, the matching time gradually increases, reaching 87.4 seconds for the index constructed with 512 bigrams. Figure 7 provides a visual representation of the percentage of log lines remaining after index lookup for subsequent regex matching.

From 4 to 64 bigrams, the matching time drops significantly as the proportion of log lines passing the index filter decreases from 20.28% to 0.63%. Consequently, fewer log lines are subjected to the

expensive regex matching. A more moderate decline of the percentage is observed as k increases from 64 to 128. It drops to 0.12% when there is a rise in matching time in Figure 6. The gains from reduced regex matching provided by the decreased number of log lines after index filtering are offset by the incremental overhead introduced by the increasing complexity and size of the index. While the index lookup remains a low-cost operation, the increasing number of bigrams introduces additional overhead for bit operations across multiple bytes, affecting the overall performance of the matching process. Therefore, as the percentage stabilizes at approximately 0.05% for k from 192 to 512, the matching time increases instead.

In the `System Y` workload, the blue bars in Figure 8b demonstrate that the index created using bigrams from queries outperforms the baseline after indexing with 16 or more bigrams. The overall query runtime decrease sharply as the k increase from 4 to 16. The performance gradually plateaus as more bigrams are indexed beyond 16. When $k = 64$, it achieves more than 7× query performance improvement compared to the baseline. Figure 8c shows that the n-gram index provide performance gain for `US-Accident` workload for $k >= 2$, and the gain increases as more bigrams are indexed, achieving a near 5× speedup when $k = 16$. With only 4 queries containing short literals, each indexed bigram appears in at least 25% of the queries, having similar filtering power for the log lines.

We observe that although increasing the number of bigrams indexed improves performance by reducing the number of log lines subjected to regex matching, it also requires careful consideration on the runtime overhead introduced by the additional bigrams. Balancing these factors is crucial for achieving optimal matching performance in regex log analysis.

4.4 Unknown Workload

In this section, we answer **Q.3** by showing if `REI` can improve the log analysis workload performance without knowing the workload in advance. Due to the incapability of existing n-gram selection methods in handling unknown log processing workloads, we compare our index performance to the baseline where the queries are assumed to be and usable for index construction. Specifically, we conduct a direct comparison of the matching performance achieved with indices built from the most frequent bigrams in the queries (i.e. having prior information about string literal distribution in the queries), QueryIndex, to those created using the most common bigrams in an English corpus [54] (i.e. the query workload is not known), EnglishIndex , across three real-world workloads: `Database X`, `System Y`, and `US-Accident`.

Given that the indices are constructed using identical data structures and an equivalent number of bigrams, the sizes of the indices remain the same. This similarity in construction time can be attributed to the same index structure and bigram count.

4.4.1 Matching Time with Index. We explore the performance differences between indices created using the most frequent bigrams from the queries, QueryIndex, and those from an English corpus, EnglishIndex. Examining Figure 8, we observe similar performance in workload queries for indices built with the two sets of bigrams. Also, Figure 8 illustrates the impact of using different k, the numbers of bigrams for indexing, across the three workloads. Both indices show a consistent decrease in overall runtimes as the k indexed increases, particularly for workloads `System Y` and `US-Accident`. For `Database X` workload, runtimes using indices from the two sources follow a similar trend that decreases sharply as the k increases, reaches a minimum at 64 bigrams for indexing, and then gradually increases as the number of bigrams indexed increases.

The `Database X` workload exhibits similarity with the English language in terms of bigram distribution, as it has the largest number of queries, each of which contain long literal components of human-readable text. From Table 2, we can see that 89% of the log entries contain the most

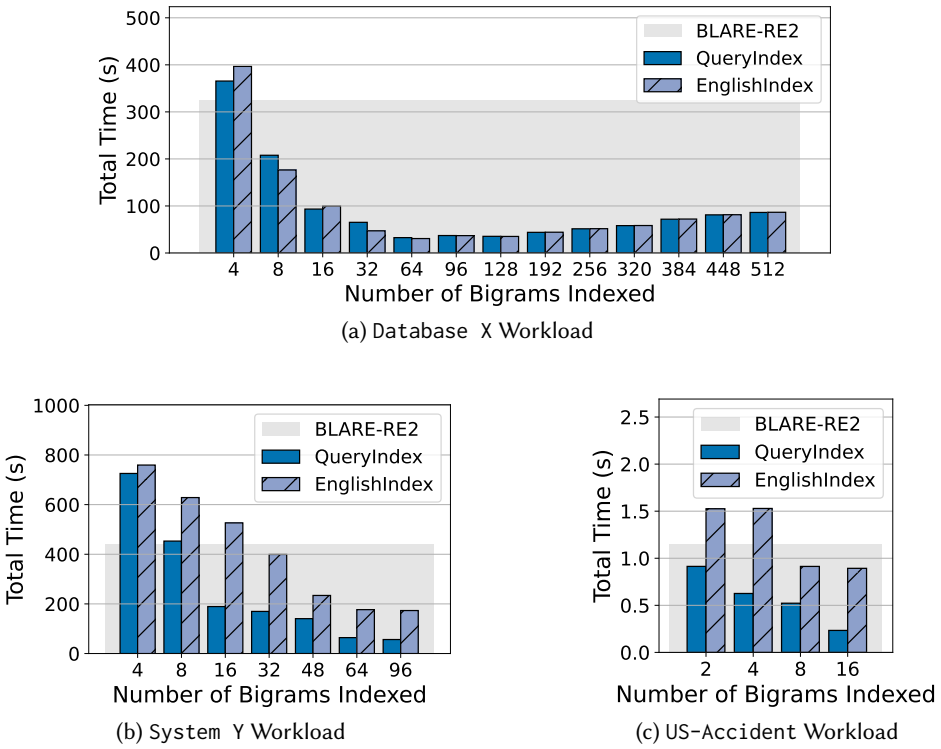


Fig. 8. Comparing the impact on matching time for different numbers of bigrams indexed. Uses top-k most frequent bigrams appeared in the workload query and top-k most frequent bigrams in English literature.

frequently occurring bigram, and 25% of all 762 bigrams have frequency percentages above 11%. Figure 8a presents the performance of the two indices built with a varying number of bigrams. When indexing 8 or more bigrams, both indices outperform the baseline performance by as much as 10× with 64 bigrams, indicating that performance improvement can be achieved with a relatively small number of bigrams. We notice a minor difference in performance between the two indices when using bigrams from 4 to 64. This discrepancy in performance becomes negligible as the number of indexed bigrams continues to rise.

The System Y workload consists of fewer queries, and there is noticeable dissimilarity across the majority of them. From Table 2, we can see that half of the 138 bigrams appear in more than 11.8% of the queries, and the most frequently occurring bigram is found in 41.2% of the queries. In Figure 8b, EnglishIndex needs at least 32 bigrams to surpass the baseline runtime. In contrast, QueryIndex requires only 16 bigrams and achieves close to 8× speedup when k reaches 96. The performance results of System Y workload exhibits a consistent trend in which QueryIndex outperforms EnglishIndex. While the numbers are similar when indexing with 4 bigrams, the performance gap widens as the k increases to 8 and 16. Despite that EnglishIndex has lower filtering power than QueryIndex, it still achieves a good speedup of 2.6× when indexing with 96 bigrams.

The US-Accident workload has only 4 queries with short literal components. With only 18 bigrams extracted, each bigram is substantially important as even the most infrequent one appears in a quarter of all the queries as shown in Table 2. QueryIndex continuously beats the baseline. Conversely, the top-6 most frequent bigrams in English are absent from the queries, causing EnglishIndex to only surpass the baseline after indexing 8 or more bigrams. QueryIndex outperforms

EnglishIndex across all values of k. For QueryIndex, the runtime gradually decreases as the number of bigrams increases. EnglishIndex also experiences a reduction in workload runtime, although meaningful speed improvements are only shown when we include one or more new bigrams that also exist in the queries of the workload.

With the result, we show that REI is resilient to unknown workload, taking advantage of the fact that logs and literal components in log analysis tasks are mostly human-readable English text. This is most significant when the workload has a large query set with a large portion of text literals. The performance gain for using bigrams according to their frequency in the English corpus is less significant when the literal components in the query set have bigram distribution dissimilar to that of the English language. Even so, REI can still achieve satisfactory performance gain by increasing the number of bigrams used for indexing.

4.5 Index Granularity

Now we answer **Q.4** by analysing the impact of index granularity on the index construction overhead and performance gain. In the experimental setup, we use index granularity of 8, 64, 192, 256, and 512 as number of log lines in a group. We maintain uniformity with the same set of bigrams for result reporting. Specifically, we plot results for the top-128 and top-64 bigrams derived from the queries of the Database X workload, and run the full Database X workload to compare the results under different levels of granularity.

4.5.1 Index Construction Overhead. Figure 9a and Figure 9b show as granularity levels become coarser, there is a drastic decrease in index size. In Figure 9a and Figure 9b, we observe that construction time decreases as the number of log lines within each group increases. When log lines are amalgamated into larger groups, for bit-vector index, it has smaller number of bit-vectors. Additionally, fewer bigram existence checks are made on average for each log line as the number of log lines corresponding to one index entry increases. This correlation between granularity and index overhead suggests opportunity for a coarser granularity to achieve lower space overhead and higher performance improvement.

4.5.2 Matching Time with Index. Looking at Figure 9e, when we index with the top-64 bigrams, we can see that as the number of log lines represented by one index entry increases, the total runtime also increases. However, for the case with top-128 bigrams in Figure 9f, having a granularity of 8 actually achieves better performance than a granularity of 1. Although having a finer granularity of 1 records more information about the dataset compared to a granularity level of 8, the gain in indexing the top 64 to 128 bigrams cannot compensate for the increased time in index lookup. For the Database X workload, the fastest running time is achieved with a granularity level of 8 and by indexing 128 bigrams. Selecting index granularity involves balancing index overhead and matching time. Finer granularity, which results in a larger number of index entries due to the more detailed division of the dataset, increases overhead but typically reduces matching time due to more selective filtering. Conversely, coarser granularity decreases overhead by aggregating information but may result in longer matching time due to a less selective filtering process, resulting in more log lines being considered during matching. The choice of granularity thus depends on optimizing computational efficiency within the constraints and requirements of the specific application.

4.6 Other Indexing Methods

Other popular solutions to encode the existence of n-grams are signature files and inverted index. In this section, we answer **Q.5** with a comparative analysis of our bit-vector index, signature files, and the inverted index, all constructed using an identical set of bigrams. We use the hashing schema, MurmurHash [1] for bigram signatures in BitFunnel [38]. We used line level bit array sizes of 64

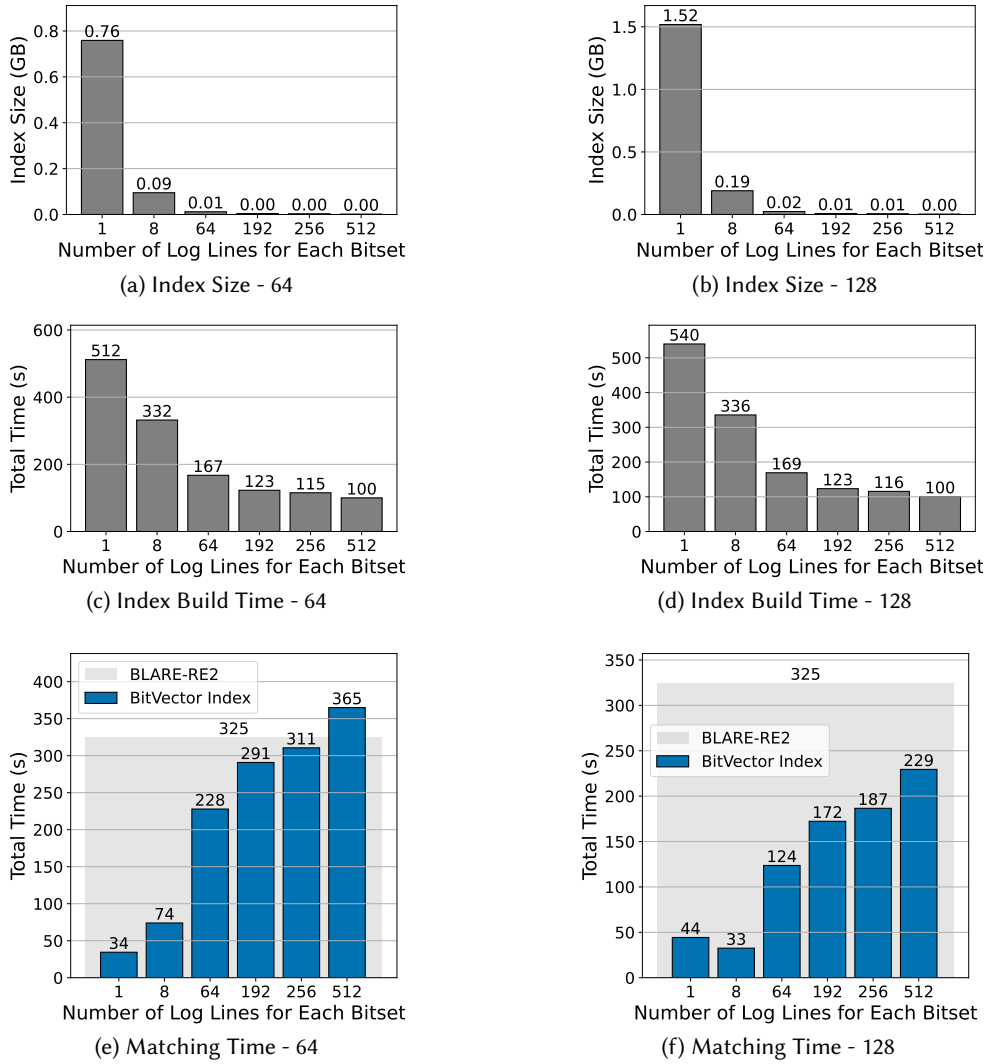


Fig. 9. Comparing the impact of the bit-vector index under different granularity levels. Uses top-128 and the top-64 most frequent bigrams in workload queries.

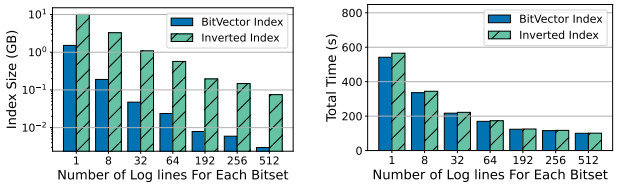
and 128 for both signature files and bivector index. For each size k, we build the bit-vector index with top k bigrams, and signature files with top-k, upper half of the top-k, and lower half of the top-k bigrams. We will explain the setting during evaluation.

Since an inverted index is often used to index documents of a size larger than the average size of a log line, we compare the impact of the indexing strategies on varying levels of index granularity. We group a fixed number of log lines together and assign a unique group id for indexing purposes. For example, at a granularity of 64, consecutive log lines were amalgamated into groups of 64, resulting in indices that are constructed based on whether specific bigrams are present in any of the log lines within each group. We build the inverted index using a hash map, with the bigrams being the key, and a set of log group IDs where the corresponding bigram exists being the value. Each bit in the bit-vector index now represent the existence of the bigrams in the corresponding

(a) Percentage After Filter (b) Matching Time

Fig. 10. Comparing the performance of REI index and signature files with the same number of bits per log line in the dataset. Uses top-64 most frequent bigrams in workload queries to construct the REI (BitVec 64). Uses top-64 (SigF 64), top-32 (SigF U32), and 32th-to-64th (SigF L32) most frequent bigrams for signature files.

log group. We run the experiments using the same configurations of index granularity and number of bigrams as in Section 4.5.



(a) Index Size (b) Index Construction Time

Fig. 11. Comparing the time and space overhead of index construction of REI index and inverted index. Uses top-128 most frequent bigrams in workload queries for both indices.

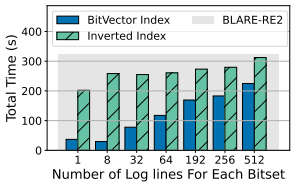


Fig. 12. Comparing matching time of the REI index and inverted index. Uses top-128 most frequent n-grams in workload queries for both indices.

4.6.1 Index Construction Overhead. Since the index structures and construction steps of signature files and REI index are very similar, where the only difference lies in how to encode the bigrams for each line, their index construction overheads are almost identical. However, this is not the case for inverted index. Figure 11a shows the detailed index size in log scale of both indices for different levels of granularity. In our analysis, we consider only the minimum required size of the inverted index, counting only the space used to hold actual index data. This is important to note, as the space reserved for a common hash map data structure is usually much larger. Upon observing the bar plot, we can see a notable difference in the trends of sizes between bit-vector index and the inverted index. For both index types, the index size decreases as granularity becomes coarser. The inverted index demands less space to denote the presence of specific bigram, and the bit-vector index has smaller number of index entries when the size of each entry stays constant. Under identical conditions, the bit-vector index has a significantly smaller size compared to the inverted index. This difference in size highlights the space efficiency of the bit-vector index in maintaining a compact representation while managing the same set of bigrams. In Figure 11b, we present the comparison of index construction time between the two types of index under varying granularity levels. Notably, the construction time of the two indices aligns closely under comparable conditions, as both indices are constructed by scanning through the dataset once, extracting all

unique bigrams from each log line, looking up the existence of the same number of bigrams in hash-map type data structure, and then performing an update on the index.

4.6.2 Matching Time with Index. Now we compared the workload runtime of the REI, signature files, and inverted indices.

Looking at signature files first, we run an experiment with bit-vector size of 64 to index the top-64 bigrams for both signature files and REI index. Signature for each log line filtered out only 11.7% of the dataset, left with 88.3% log lines going through the regex engine, whereas REI index filtered out 99.4% of the dataset, as shown in the leftmost and rightmost bars in Figure 10a. Looking at the bit-vector signatures for each log line and the bitmask for each regex, we note that they are over saturated; bit-vectors are often all ones for log lines and all zeros for regexes.

With the same bit-vector size 64, we reduce the number of bigrams indexed to 32. By selecting the top-32 frequent bigrams, the percentage of log lines filtered increase to 31.9%; by selecting the other half of the top-64 frequent bigrams, only about half (51.2%) of the dataset need to be matched with regex engine after the index filtering. Both percentages of signature files method are still significantly higher than REI index, as shown in Figure 10a. Looking at Figure 10b, the filtering difference translates to more higher querying overhead for signature files compared to REI index, resulting in a 62×, 48×, and 38× slower overall matching time for signature files using 64 bigrams, top-32 bigrams, and 32-64th bigrams respectively compared to REI index of the same size. In fact, the matching time for signature files with 64 bigrams is very close to the total workload time of using RE2 directly without any index as the signature of each log line is usually too dense.

Shifting our focus to inverted index, in Figure 12 there is a general trend that the REI index consistently outperforms the inverted index in terms of matching time, especially at finer granularities. This performance disparity becomes less pronounced as granularity become coarser. Also, as we transition to coarser granularities, the inverted index demonstrates a more gradual increase in the runtime compared to the REI index, which experiences a sharper increase. The cost of index look-up to find all candidate log lines is higher for the inverted index than for the bit-vector index. When the granularity gets coarser, the number of lines remaining for full regex matching increases, and the cost of candidate search is amortized. Therefore, we can see from Figure 12 that as the granularity becomes coarser, the performance gap between the bit-vector index and the inverted index becomes smaller. This aligns with conclusions of prior works that inverted index is suitable for indexing documents much larger in sizes than the log lines and the log groups in our experiments.

4.7 Case Study: Impact of Query Set Size

To analyze the scalability of REI as the query workload size increases, we conducted an additional experiment comparing the sampled `Database X` workload (132 queries) with the complete `Database X` workload (8,941 queries). As the query set expanded, we did not see a significant change in index construction time. This is because the query set size impacts only the bigram selection process, which is very lightweight: around 0.05 seconds in the entire ∼ 500 seconds of index construction time. For query performance, the sampled workload runs 35.7s using REI indexing with 64 bigrams, achieving a runtime speedup of more than 9×, whereas the larger workload runs 49.9s with the same configuration, achieving a significant speedup of 379× over the baseline time of 18928s. These results confirm that REI efficiently scales to larger query sets.

5 Related Work

Due to the complexity of regex patterns and the variety of datasets, efficient regular expression matching and indexing are vital in several areas such as bioinformatics [4, 40, 44, 71, 78], event

stream processing [3, 23, 43, 88], network intrusion detection system [6, 33, 53, 55, 62, 64, 66, 68, 80, 94, 97], text processing [13, 91] and data mining [28, 34, 35, 93].

Traditional regex matching algorithms. Without the aid of indexing, traditional pattern-matching approaches, such as DFA and NFA, have established the foundations. Adapting ideas from KMP [60], Aho-Corasick algorithms, and Boyer-Moore [11], there has been work optimizing automata matching [26, 46, 89, 95]. DFA, despite its reputation for rapid matching, frequently encounters state explosion issues, and works had been done optimizing its space usage [8–10, 12, 32, 47–49, 61–63, 73, 79, 98, 105, 108, 109]. NFA provides a more space-efficient option, at the cost of computational blow-up during matching, and there are works on optimizing its performance [50, 57, 72, 75, 80, 106]. Automaton variations that employed one or more of the above techniques have been proposed, including suffix automaton [27, 75], XFA [87], D^2FA [63], and $\delta^N FA$ [32]. Recent work has also investigated how to obtain speed up those methods using modern hardwares [7, 12, 33, 48, 53, 55, 64, 68, 77, 84, 86, 94, 97, 99, 100, 107] Those solutions targeted general regex matching task without considering the characteristics of a specific workload.

Theoretical analysis of index building. When we can preprocess the dataset, building an index for faster regex queries becomes an option. [36] conducted a theoretical analysis to understand hardness constraints and trade-offs of the index built for regex matching. Different data preparation methods and indexing strategies are proposed considering the trade-offs. Some works focus on a specific type of automata (e.g. Wheeler automata), analyzing the performance bounds [24] .

Bit-Vector Indexes. In the related realm of information retrieval, research has been done on signature file, generating bit signatures for documents [31], and multikey attributes [83] for fast lookup. The signatures are generated by hashing each word into bit-vector of the same size, and then superimposing the bit-vector [31]. Though with various attempts of optimization [51, 58, 65, 83], evaluation [114] demonstrated that there is a trade-off wrt. inverted indexes, as it requires multiple hash computation for each word and a signature size as large as $10K$ bits. The difference in bit-vector generation step makes `REI` inherently free of the above disadvantages. Recent work on Multi-Dimensional Data Layouts (MDDL) [29] has also explored bitmap-based indexing for prefiltering in database systems. The key idea in MDDL is to maintain a bitmap per row that encodes whether the row satisfies a fixed set of query predicates. While MDDL orders predicates based on estimated selectivity to minimize overall filtering cost, our approach orders n-grams by their frequency in the query workload to ensure broader applicability across diverse regex patterns. Additionally, MDDL supports dynamic updates and maintenance of bitmap indexes as workloads evolve, which our current design does not support—an exciting direction for future work. However, unlike MDDL, which benefits from sorted bitmap indexes, applying similar sorting strategies to log data for regex evaluation is significantly more expensive due to the sheer data volume. Moreover, such sorting could degrade regex evaluation performance, especially when logs are already sorted by another dimension (e.g., time). Consequently, data-partitioning-based techniques require deeper investigation, presenting an exciting future work direction.Feed-forward Bloom filter is also used in [16, 70] as a lightweight indexing structure on the patterns for multi-pattern matching together with hardware optimization techniques.

Inverted indexes. A very large collection of works uses inverted index or similar data structures to index multigrams of the dataset. Inverted index with positional information works well for database when each entry is large. The inverted index uses n-gram as the search key and returns a list of positional identifiers [19, 22, 59, 74, 85, 103]. There are works that aim to reduce the space consumption [22] or build the index under space constraint [45].

Tree indexes. General tries and suffix trees are also commonly used for indexing data for pattern matching [5, 14, 15, 37, 56, 67]. However, they have a heavy space requirement. Without compression, these types of indexes on a natural language dataset can easily reach a size more than 10 times of the original dataset [82]. A two-level index is also employed using a hash table and a trie on encoded signature of DNA data [15]. This method is popular in genome database where the alphabet size is small. Suffix trees are also commonly used for multi-pattern matching where indexes are constructed on the queries.

***N*-grams for indexing and filtering.** Besides regex matching, the n-gram indexing or pre-filtering is broadly used in string matching for predicate matching and joins [42]. Trigrams are commonly used in existing indexing solutions [2, 20, 25] where all trigrams are indexed, usually using inverted index. Works has been done comparing the impact of n of n-gram for inverted index [22, 25, 59]. Some [22, 52] build an index for all n-grams of $n \leq k$. Solutions in [19, 45, 76, 79] instead use variable length multigrams. Github Code Search [21] assigns weights to each bigram, and selectively indexes on multigrams that are composed of consecutive bigrams according to their weights .

6 Conclusion and Future Work

We have revisited the challenging problem of regular expression indexing in this paper. We conducted a systematic analysis of traditional methods like inverted indexes with n-grams. We examined how well they performed when combined with modern hardware and state-of-the-art regex evaluation engines. We introduced a lightweight indexing framework, `REI`, that is configurable. This allows for a balance between the cost of building indexes and the improvement in query performance. Our technique stands out from existing approaches in two key aspects. First, we employ a bit filtering-based index for a carefully chosen subset of n-grams. Second, we store the bit index directly alongside each log line in our input, rather than in a separate data structure. This integration not only simplifies the indexing process with a much lower cost than existing methods but also results in a more streamlined and efficient framework. It is also important to emphasize that `REI` remains independent of the regex engine, which guarantees flexibility and wide applicability on different platforms and use cases.

We envision our lightweight indexing framework developing further in the future to leverage distributed computing and parallel processing to manage ever-increasing volumes of data with ease. To further increase the adaptability and robustness of our framework, another area worth noting is the development of a dynamic index updating strategy. The strategy is expected to include the ability to add, remove, or update bigrams in the index. Future extensions could leverage incremental statistics from query execution to adaptively refine the index, enabling dynamic reconfiguration for settings where the data and queries are continuously changing. This setting is an important consideration to remedy the limitation of our work's assumption that the distribution of literals is close to that of the English language. The dynamic setting is also common in streaming and security-focused scenarios such as intrusion detection systems (for example, Snort [81] and Suricata [90]). We also plan to explore bitmap compression techniques like RLE, Roaring Bitmaps, and WAH, which can reduce storage overhead without affecting precision. These methods are especially promising for integration with columnar storage formats, further improving space efficiency. Furthermore, a thorough investigation of storage strategies on efficient data formats like Parquet can enhance index storage and retrieval procedures.

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] aappleby. [n. d.]. *SMHasher*. https://github.com/aappleby/smhasher

[2] Elizabeth S. Adams and Arnold C. Meltzer. 1993. Trigrams as Index Element in Full Text Retrieval: Observations and Experimental Results. In *Proceedings of the 1993 ACM Conf. on Computer Science* (Indianapolis, Indiana, USA) *(CSC '93)*. Association for Computing Machinery, New York, NY, USA, 433–439. doi:10.1145/170791.170891

[3] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. 2008. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD Int'l conference on Management of data*. ACM. doi:10.1145/1376616.1376634

[4] Abdullah N. Arslan and Dan He. 2006. An improved algorithm for the regular expression constrained multiple sequence alignment problem. In *Sixth IEEE Symp. on BioInformatics and BioEngineering (BIBE'06)*. IEEE. doi:10.1109/bibe.2006.253324

[5] Ricardo A. Baeza-Yates and Gaston H. Gonnet. 1996. Fast text searching for regular expressions or automaton searching on tries. *J. ACM* 43, 6 (Nov. 1996), 915–936. doi:10.1145/235809.235810

[6] Zachary Baker, Hong jip Jung, and Viktor Prasanna. 2006. Regular Expression Software Deceleration for Intrusion Detection Systems. In *2006 Int'l Conf. on Field Programmable Logic and Applications*. IEEE. doi:10.1109/fpl.2006.311246

[7] Zachary K. Baker and Viktor K. Prasanna. 2004. Time and area efficient pattern matching on FPGAs. In *Proceedings of the 2004 ACM/SIGDA 12th Int'l Symp. on Field programmable gate arrays*. ACM. doi:10.1145/968280.968312

[8] M. Becchi and S. Cadambi. 2007. Memory-Efficient Regular Expression Search Using State Merging. In *IEEE INFOCOM 2007 - 26th IEEE Int'l Conf. on Computer Communications*. IEEE. doi:10.1109/infcom.2007.128

[9] Michela Becchi and Patrick Crowley. 2007. An improved algorithm to accelerate regular expression evaluation. In *Proceedings of the 3rd ACM/IEEE Symp. on Architecture for networking and communications systems*. ACM. doi:10.1145/1323548.1323573

[10] Michela Becchi and Patrick Crowley. 2013. A-DFA: A Time- and Space-Efficient DFA Compression Algorithm for Fast Regular Expression Evaluation. *ACM Transactions on Architecture and Code Optimization* 10, 1 (apr 2013), 1–26. doi:10.1145/2445572.2445576

[11] Robert S. Boyer and J. Strother Moore. 1977. A Fast String Searching Algorithm. *Commun. ACM* 20, 10 (oct 1977), 762–772. doi:10.1145/359842.359859

[12] B.C. Brodie, D.E. Taylor, and R.K. Cytron. 2006. A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching. In *33rd Int'l Symp. on Computer Architecture (ISCA'06)*. 191–202. doi:10.1109/ISCA.2006.7

[13] D. D. A. Bui and Q. Zeng-Treitler. 2014. Learning regular expressions for clinical text classification. *Journal of the American Medical Informatics Association* 21, 5 (sep 2014), 850–857. doi:10.1136/amiajnl-2013-002411

[14] Stefan Burkhardt, Andreas Crauser, Paolo Ferragina, Hans-Peter Lenhof, Eric Rivals, and Martin Vingron. 1999. q-gram based database searching using a suffix array (QUASAR). In *Proceedings of the third annual Int'l conference on Computational molecular biology*. ACM. doi:10.1145/299432.299460

[15] Xia Cao, Shuai Cheng Li, and Anthony K. H. Tung. 2005. Indexing DNA Sequences Using q-Grams. In *Database Systems for Advanced Applications*. Springer Berlin Heidelberg, 4–16. doi:10.1007/11408079_4

[16] Sang Kil Cha, Iulian Moraru, Jiyong Jang, John Truelove, David Brumley, and David G. Andersen. 2011. SplitScreen: Enabling Efficient, Distributed Malware Detection. *Journal of Communications and Networks* 13, 2 (apr 2011), 187–200. doi:10.1109/jcn.2011.6157418

[17] Chee-Yong Chan and Yannis E. Ioannidis. 1998. Bitmap index design and evaluation. *ACM SIGMOD Record* 27, 2 (June 1998), 355–366. doi:10.1145/276305.276336

[18] Chee-Yong Chan and Yannis E. Ioannidis. 1999. An efficient bitmap encoding scheme for selection queries. *ACM SIGMOD Record* 28, 2 (June 1999), 215–226. doi:10.1145/304181.304201

[19] Junghoo Cho and S. Rajagopalan. 2002. A fast regular expression indexing engine. In *Proceedings 18th Int'l Conf. on Data Engineering*. 419–430. doi:10.1109/ICDE.2002.994755

[20] 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

[21] Timothy Clem. 2023. The technology behind GitHub's new code search. https://github.blog/2023-02-06-the-technology-behind-githubs-new-code-search/

[22] Derrick Coetzee. 2008. TinyLex: Static n-Gram Index Pruning with Perfect Recall. In *Proceedings of the 17th ACM Conf. on Information and Knowledge Management* (Napa Valley, California, USA) *(CIKM '08)*. Association for Computing Machinery, New York, NY, USA, 409–418. doi:10.1145/1458082.1458138

[23] Norman H. Cohen and Karl Trygve Kalleberg. 2008. EventScript. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*. ACM. doi:10.1145/1375657.1375673

[24] Nicola Cotumaccio and Nicola Prezza. 2021. On Indexing and Compressing Finite Automata. In *Proceedings of the 2021 ACM-SIAM Symp. on Discrete Algorithms (SODA)*. Society for Industrial and Applied Mathematics, 2585–2599.

doi:10.1137/1.9781611976465.153

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

[26] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. 1994. Speeding up two string-matching algorithms. *Algorithmica* 12, 4-5 (nov 1994), 247–267. doi:10.1007/bf01185427

[27] Maxime Crochemore, A. Czumaj, L. Gasieniec, T. Lecroq, W. Plandowski, and W. Rytter. 1999. Fast practical multi-pattern matching. *Inform. Process. Lett.* 71, 3-4 (aug 1999), 107–113. doi:10.1016/s0020-0190(99)00092-7

[28] Sandra de Amo and Daniel A. Furtado. 2007. First-order temporal pattern mining with regular expression constraints. *Data and Knowledge Engineering* 62, 3 (sep 2007), 401–420. doi:10.1016/j.datak.2006.08.009

[29] Jialin Ding, Matt Abrams, Sanghita Bandyopadhyay, Luciano Di Palma, Yanzhu Ji, Davide Pagano, Gopal Paliwal, Panos Parchas, Pascal Pfeil, Orestis Polychroniou, Gaurav Saxena, Aamer Shah, Amina Voloder, Sherry Xiao, Davis Zhang, and Tim Kraska. 2024. Automated Multidimensional Data Layouts in Amazon Redshift. In *Companion of the 2024 Int'l Conf. on Management of Data (SIGMOD/PODS '24)*. ACM, 55–67. doi:10.1145/3626246.3653379

[30] Jason Ellis, Achille Fokoue, Oktie Hassanzadeh, Anastasios Kementsietsidis, Kavitha Srinivas, and Michael J. Ward. 2015. Exploring Big Data with Helix. *ACM SIGMOD Record* 43, 4 (feb 2015), 43–54. doi:10.1145/2737817.2737829

[31] Chris Faloutsos and Stavros Christodoulakis. 1984. Signature files: an access method for documents and its analytical performance evaluation. *ACM Transactions on Information Systems* 2, 4 (Oct. 1984), 267–288. doi:10.1145/2275.357411

[32] Domenico Ficara, Andrea Di Pietro, Stefano Giordano, Gregorio Procissi, Fabio Vitucci, and Gianni Antichi. 2011. Differential Encoding of DFAs for Fast Regular Expression Matching. *IEEE/ACM Transactions on Networking* 19, 3 (jun 2011), 683–694. doi:10.1109/tnet.2010.2089639

[33] Ming Gao, Kenong Zhang, and Jiahua Lu. 2006. Efficient packet matching for gigabit network intrusion detection using TCAMs. In *20th Int'l Conf. on Advanced Information Networking and Applications - Vol. 1 (AINA'06)*. IEEE. doi:10.1109/aina.2006.164

[34] M. Garofalakis, R. Rastogi, and K. Shim. 2002. Mining sequential patterns with regular expression constraints. *IEEE Transactions on Knowledge and Data Engineering* 14, 3 (may 2002), 530–552. doi:10.1109/tkde.2002.1000341

[35] Minos N. Garofalakis, Rajeev Rastogi, and Kyuseok Shim. 1999. SPIRIT: Sequential Pattern Mining with Regular Expression Constraints. In *Proceedings of the 25th Int'l Conf. on Very Large Data Bases (VLDB '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 223–234.

[36] Daniel Gibney and Sharma V. Thankachan. 2021. Text Indexing for Regular Expression Matching. *Algorithms* 14, 5 (apr 2021), 133. doi:10.3390/a14050133

[37] Eldar Giladi, Michael G. Walker, James Z. Wang, and Wayne Volkmuth. 2002. SST: An Algorithm for Finding Near-Exact Sequence Matches in Time Proportional to The Logarithm of The Database Size. *Bioinformatics* 18, 6 (jun 2002), 873–877. doi:10.1093/bioinformatics/18.6.873

[38] Bob Goodwin, Michael Hopcroft, Dan Luu, Alex Clemmer, Mihaela Curmei, Sameh Elnikety, and Yuxiong He. 2017. BitFunnel: Revisiting Signatures for Search. In *Proceedings of the 40th Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval (SIGIR '17)*. ACM, 605–614. doi:10.1145/3077136.3080789

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

[40] Philippe Gouret, Julie D Thompson, and Pierre Pontarotti. 2009. PhyloPattern: regular expressions to identify complex patterns in phylogenetic trees. *BMC Bioinformatics* 10, 1 (sep 2009). doi:10.1186/1471-2105-10-298

[41] Szymon Grabowski, Robert Susik, and Marcin Raniszewski. 2016. A Bloom filter based semi-index on q-grams. *Software: Practice and Experience* 47, 6 (Aug. 2016), 799–811. doi:10.1002/spe.2431

[42] Luis Gravano, Panos Ipeirotis, H. Jagadish, Nick Koudas, Senthilmurugan Muthukrishnan, Lauri Pietarinen, and Divesh Srivastava. 2001. Using q-grams in a DBMS for Approximate String Processing. *IEEE Data Eng. Bull.* 24 (2001), 28–34.

[43] Sylvain Halle and Simon Varvaressos. 2014. A Formalization of Complex Event Stream Processing. In *2014 IEEE 18th Int'l Enterprise Distributed Object Computing Conf.* IEEE. doi:10.1109/edoc.2014.12

[44] Laurie Hammel and Jignesh M. Patel. 2002. Searching on the Secondary Structure of Protein Sequences. In *VLDB '02: Proceedings of the 28th Int'l Conf. on Very Large Databases*. Elsevier, 634–645. doi:10.1016/b978-155860869-6/50062-7

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

[46] R. Nigel Horspool. 1980. Practical fast searching in strings. *Software: Practice and Experience* 10, 6 (jun 1980), 501–506. doi:10.1002/spe.4380100608

[47] N. Hua, H. Song, and T. V. Lakshman. 2009. Variable-Stride Multi-Pattern Matching For Scalable Deep Packet Inspection. In *IEEE INFOCOM 2009*. 415–423. doi:10.1109/INFCOM.2009.5061946

[48] Kun Huang, Linxuan Ding, Gaogang Xie, Dafang Zhang, Alex X. Liu, and Kave Salamatian. 2013. Scalable TCAM-based regular expression matching with compressed finite automata. In *Architectures for Networking and Communications Systems*. IEEE. doi:10.1109/ancs.2013.6665178

[49] Sheng Huo, Dafang Zhang, and Yanbiao Li. 2015. Fast and Scalable Regular Expressions Matching with Multi-Stride Index NFA. In *Algorithms and Architectures for Parallel Processing*. Springer Int'l Publishing, 597–610. doi:10.1007/978-3-319-27137-8_43

[50] Heikki Hyyrö and Gonzalo Navarro. 2002. Faster Bit-Parallel Approximate String Matching. In *Combinatorial Pattern Matching*, Alberto Apostolico and Masayuki Takeda (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 203–224. doi:10.1007/3-540-45452-7_18

[51] Yoshiharu Ishikawa, Hiroyuki Kitagawa, and Nobuo Ohbo. 1993. Evaluation of signature files as set access facilities in OODBs. *ACM SIGMOD Record* 22, 2 (June 1993), 247–256. doi:10.1145/170036.170076

[52] Milos Jakubícek and Pavel Rychlý. 2014. Optimization of Regular Expression Evaluation within the Manatee Corpus Management System. In *The 8th Workshop on Recent Advances in Slavonic Natural Languages Processing, RASLAN 2014, Karlova Studanka, Czech Republic, December 5-7, 2014*, Ales Horák and Pavel Rychlý (Eds.). Tribun EU, 37–48. http://nlp.fi.muni.cz/raslan/2014/1.pdf

[53] Muhammad Asim Jamshed, Jihyung Lee, Sangwoo Moon, Insu Yun, Deokjin Kim, Sungryoul Lee, Yung Yi, and KyoungSoo Park. 2012. Karguss: a highly-scalable software-based intrusion detection system. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. doi:10.1145/2382196.2382232

[54] Michael N. Jones and D. J. K. Mewhort. 2004. Case-sensitive letter and bigram frequency counts from large-scale English corpora. *Behavior Research Methods, Instruments, Computers* 36, 3 (aug 2004), 388–396. doi:10.3758/bf03195586

[55] Hong-Jip Jung, Z.K. Baker, and V.K. Prasanna. 2006. Performance of FPGA implementation of bit-split architecture for intrusion detection systems. In *Proceedings 20th IEEE Int'l Parallel and Distributed Processing Symp.* IEEE. doi:10.1109/ipdps.2006.1639434

[56] Ramakrishnan Kandhan, Nikhil Teletia, and Jignesh M. Patel. 2010. SigMatch: Fast And Scalable Multi-Pattern Matching. *Proceedings of the VLDB Endowment* 3, 1-2 (sep 2010), 1173–1184. doi:10.14778/1920841.1920987

[57] Yusaku Kaneta, Shin-Ichi Minato, and Hiroki Arimura. 2010. Fast Bit-Parallel Matching for Network and Regular Expressions. In *Proceedings of the 17th Int'l Conf. on String Processing and Information Retrieval* (Los Cabos, Mexico) *(SPIRE'10)*. Springer-Verlag, Berlin, Heidelberg, 372–384. doi:10.1007/978-3-642-16321-0_39

[58] A. Kent, R. Sacks-Davis, and K. Ramamohanarao. 1990. A signature file scheme based on multiple organizations for indexing very large text databases. *Journal of the American Society for Information Science* 41, 7 (Oct. 1990), 508–534. doi:10.1002/(sici)1097-4571(199010)41:7<508::aid-asi5>3.0.co;2-j

[59] 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 Int'l Conf. on Data Engineering (ICDE 2010)*. 721–732. doi:10.1109/ICDE.2010.5447866

[60] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. 1977. Fast Pattern Matching in Strings. *SIAM J. Comput.* 6, 2 (1977), 323–350. arXiv:https://doi.org/10.1137/0206024 doi:10.1137/0206024

[61] Shijin Kong, Randy Smith, and Cristian Estan. 2008. Efficient signature matching with multiple alphabet compression tables. In *Proceedings of the 4th Int'l conference on Security and privacy in communication netowrks*. ACM. doi:10.1145/1460877.1460879

[62] Pawan Kumar and Virendra Singh. 2012. Efficient regular expression pattern matching for network intrusion detection systems using modified word-based automata. In *Proceedings of the Fifth Int'l Conf. on Security of Information and Networks*. ACM. doi:10.1145/2388576.2388590

[63] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. 2006. Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. In *Proceedings of the 2006 Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Pisa, Italy) *(SIGCOMM '06)*. Association for Computing Machinery, New York, NY, USA, 339–350. doi:10.1145/1159913.1159952

[64] Chun-Liang Lee, Chung-Yuan Huang, Kai-Ping Lu, and Jhao-Han Chen. 2015. A Fast and Scalable Multi-Pattern Matching Algorithm for Intrusion Detection Systems. In *The Proceedings of the 2nd Int'l Conf. on Industrial Application Engineering 2015*. The Institute of Industrial Applications Engineers. doi:10.12792/iciae2015.057

[65] Z. Lin and C. Faloutsos. 1992. Frame-sliced signature files. *IEEE Transactions on Knowledge and Data Engineering* 4, 3 (June 1992), 281–289. doi:10.1109/69.142018

[66] Rong-Tai Liu, Nen-Fu Huang, Chih-Hao Chen, and Chia-Nan Kao. 2004. A fast string-matching algorithm for network processor-based intrusion detection system. *ACM Transactions on Embedded Computing Systems* 3, 3 (aug 2004), 614–633. doi:10.1145/1015047.1015055

[67] Colin Meek, Jignesh M. Patel, and Shruti Kasetty. 2003. OASIS: An Online and Accurate Technique for Local-alignment Searches on Biological Sequences. In *Proceedings 2003 VLDB Conf.* Elsevier, 910–921. doi:10.1016/b978-012722442-8/50085-9

[68] Chad R. Meiners, Jignesh Patel, Eric Norige, Eric Torng, and Alex X. Liu. 2010. Fast Regular Expression Matching Using Small TCAMs for Network Intrusion Detection and Prevention Systems. In *Proceedings of the 19th USENIX Conf. on Security* (Washington, DC) *(USENIX Security'10)*. USENIX Association, USA, 8.

[69] 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 Int'l Conf. on Advances in Geographic Information Systems* (Chicago, IL, USA) *(SIGSPATIAL '19)*. Association for Computing Machinery, New York, NY, USA, 33–42. doi:10.1145/3347146.3359078

[70] Iulian Moraru and David G. Andersen. 2012. Exact Pattern Matching with Feed-Forward Bloom Filters. *ACM J. Exp. Algorithmics* 17, Article 3.4 (sep 2012), 18 pages. doi:10.1145/2133803.2330085

[71] M. Mulder and G.S. Nezlek. 2006. Creating protein sequence patterns using efficient regular expressions in bioinformatics research. In *28th Int'l Conf. on Information Technology Interfaces, 2006.* IEEE. doi:10.1109/iti.2006.1708479

[72] Gene Myers. 1999. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM* 46, 3 (may 1999), 395–415. doi:10.1145/316542.316550

[73] Maleeha Najam, Usman Younis, and Raihan Ur Rasool. 2014. Multi-byte Pattern Matching Using Stride-K DFA for High Speed Deep Packet Inspection. In *2014 IEEE 17th Int'l Conf. on Computational Science and Engineering*. IEEE. doi:10.1109/cse.2014.125

[74] Gonzalo Navarro and Ricardo Baeza-Yates. 2018. A Practical q -Gram Index for Text Retrieval Allowing Errors. *CLEI Electronic Journal* 1, 2 (sep 2018). doi:10.19153/cleiej.1.2.3

[75] Gonzalo Navarro and Mathieu Raffinot. 2000. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics* 5 (dec 2000), 4. doi:10.1145/351827.384246

[76] Gonzalo Navarro and Leena Salmela. 2009. Indexing Variable Length Substrings for Exact and Approximate Matching. In *String Processing and Information Retrieval*. Springer Berlin Heidelberg, 214–221. doi:10.1007/978-3-642-03784-9_21

[77] Kunyang Peng, Siyuan Tang, Min Chen, and Qunfeng Dong. 2011. Chain-Based DFA Deflation for Fast and Scalable Regular Expression Matching Using TCAM. In *2011 ACM/IEEE Seventh Symp. on Architectures for Networking and Communications Systems*. IEEE. doi:10.1109/ancs.2011.13

[78] Gorka Prieto, Asier Fullaondo, and Jose A. Rodriguez. 2014. Prediction of nuclear export signals using weighted regular expressions (Wregex). *Bioinformatics* 30, 9 (jan 2014), 1220–1227. doi:10.1093/bioinformatics/btu016

[79] 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 (mar 2022), 1133–1148. doi:10.1109/tkde.2020.2992295

[80] Huang-chun Roan, Wen-jyi Hwang, and Chia-tien Dan Lo. 2006. Shift-Or Circuit for Efficient Network Intrusion Detection Pattern Matching. In *2006 Int'l Conf. on Field Programmable Logic and Applications*. 1–6. doi:10.1109/FPL.2006.311314

[81] Martin Roesch et al. 1999. Snort: Lightweight intrusion detection for networks.. In *Lisa*, Vol. 99. 229–238.

[82] Darío Ruano, Norma Herrera, Jésica Cornejo, and Paola Azar. 2021. *Sequential Representation of Suffix Trie: An Empirical Evaluation*. Springer Int'l Publishing, 182–196. doi:10.1007/978-3-030-75836-3_13

[83] R. Sacks-Davis, A. Kent, and K. Ramamohanarao. 1987. Multikey access methods based on superimposed coding techniques. *ACM Transactions on Database Systems* 12, 4 (Nov. 1987), 655–696. doi:10.1145/32204.32222

[84] Daniele Paolo Scarpazza and Gregory F. Russell. 2009. High-performance regular expression scanning on the Cell/B.E. processor. In *Proceedings of the 23rd Int'l conference on Supercomputing*. ACM. doi:10.1145/1542275.1542284

[85] Seon-Ho SHIN, HyunBong KIM, and MyungKeun YOON. 2018. Regular Expression Filtering on Multiple q-Grams. *IEICE Transactions on Information and Systems* E101.D, 1 (2018), 253–256. doi:10.1587/transinf.2017edl8180

[86] Reetinder Sidhu and Viktor K. Prasanna. 2001. Fast Regular Expression Matching Using FPGAs. In *Proceedings of the the 9th Annual IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM '01)*. IEEE Computer Society, USA, 227–238.

[87] Randy Smith, Cristian Estan, and Somesh Jha. 2008. XFA: Faster Signature Matching with Extended Automata. In *2008 IEEE Symp. on Security and Privacy (sp 2008)*. IEEE. doi:10.1109/sp.2008.14

[88] Kento Sugiura and Yoshiharu Ishikawa. 2020. Multiple Regular Expression Pattern Monitoring over Probabilistic Event Streams. *IEICE Transactions on Information and Systems* E103.D, 5 (may 2020), 982–991. doi:10.1587/transinf.2019dap0009

[89] Daniel M. Sunday. 1990. A very fast substring search algorithm. *Commun. ACM* 33, 8 (aug 1990), 132–142. doi:10.1145/79173.79184

[90] Home Suricata. 2009. Home-Suricata. *The Open Information Security Foundation* (2009).

[91] Naoshi Tabuchi, Eijiro Sumii, and Akinori Yonezawa. 2003. Regular Expression Types for Strings in a Text Processing Language. *Electronic Notes in Theoretical Computer Science* 75 (feb 2003), 95–113. doi:10.1016/s1571-0661(04)80781-3

[92] The Apache Software Foundation. 2022. *Apache Lucene 9.4.2 Documentation*. Retrieved January 17, 2023 from https://lucene.apache.org/core/9_4_2/index.html

[93] Roberto Trasarti, Francesco Bonchi, and Bart Goethals. 2008. Sequence Mining Automata: A New Technique for Mining Frequent Sequences under Regular Expressions. In *2008 Eighth IEEE Int'l Conf. on Data Mining*. IEEE. doi:10.1109/icdm.2008.111

[94] Gerald Tripp. 2007. Regular expression matching with input compression: a hardware design for use within network intrusion detection systems. *Journal in Computer Virology* 3, 2 (apr 2007), 125–134. doi:10.1007/s11416-007-0047-z

[95] Uday Trivedi. 2020. An Optimized Aho-Corasick Multi-Pattern Matching Algorithm for Fast Pattern Matching. In *2020 IEEE 17th India Council Int'l Conf. (INDICON)*. 1–5. doi:10.1109/INDICON49873.2020.9342041

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

[97] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. 2011. MIDeA: A Multi-Parallel Intrusion Detection Architecture. In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM. doi:10.1145/2046707.2046741

[98] Kai Wang, Zhe Fu, Xiaohe Hu, and Jun Li. 2014. Practical regular expression matching free of scalability and performance barriers. *Computer Communications* 54 (dec 2014), 97–119. doi:10.1016/j.comcom.2014.08.005

[99] Shicheng Wang, Menghao Zhang, Guanyu Li, Chang Liu, Zhiliang Wang, Ying Liu, and Mingwei Xu. 2023. Bolt: Scalable and Cost-Efficient Multistring Pattern Matching With Programmable Switches. *IEEE/ACM Transactions on Networking* 31, 2 (apr 2023), 846–861. doi:10.1109/tnet.2022.3202523

[100] 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 Symp. on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 631–648. https://www.usenix.org/conference/nsdi19/presentation/wang-xiang

[101] Stefan Weigert, Matti A. Hiltunen, and Christof Fetzer. 2014. Finding the Needle in the Haystack: Identifying Business Communities in Internet Traffic. In *2014 IEEE/WIC/ACM Int'l Joint Conf.s on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*. IEEE. doi:10.1109/wi-iat.2014.31

[102] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. 2004. Configuration Debugging as Search: Finding the Needle in the Haystack. In *Proceedings of the 6th Conf. on Symp. on Operating Systems Design & Implementation - Vol. 6* (San Francisco, CA) *(OSDI'04)*. USENIX Association, USA, 6.

[103] H.E. Williams and J. Zobel. 2002. Indexing and retrieval for genomic databases. *IEEE Transactions on Knowledge and Data Engineering* 14, 1 (2002), 63–78. doi:10.1109/69.979973

[104] Harry K. T. Wong, Hsiu-Fen Liu, Frank Olken, Doron Rotem, and Linda Wong. 1985. Bit Transposed Files. In *Proceedings of the 11th Int'l Conf. on Very Large Data Bases - Vol. 11* (Stockholm, Sweden) *(VLDB '85)*. VLDB Endowment, 448–457.

[105] Jiajia Yang, Lei Jiang, Qiu Tang, Qiong Dai, and Jianlong Tan. 2016. PiDFA: A Practical Multi-Stride Regular Expression Matching Engine Based on FPGA. In *2016 IEEE Int'l Conf. on Communications (ICC)*. IEEE. doi:10.1109/icc.2016.7511199

[106] Liu Yang, Rezwana Karim, Vinod Ganapathy, and Randy Smith. 2011. Fast, memory-efficient regular expression matching with NFA-OBDDs. *Computer Networks* 55, 15 (oct 2011), 3376–3393. doi:10.1016/j.comnet.2011.07.002

[107] Yi-Hua E. Yang, Weirong Jiang, and Viktor K. Prasanna. 2008. Compact architecture for high-throughput regular expression matching on FPGA. In *Proceedings of the 4th ACM/IEEE Symp. on Architectures for Networking and Communications Systems*. ACM. doi:10.1145/1477942.1477948

[108] Yi-Hua E. Yang and Viktor K. Prasanna. 2011. Space-time tradeoff in regular expression matching with semi-deterministic finite automata. In *2011 Proceedings IEEE INFOCOM*. IEEE. doi:10.1109/infcom.2011.5934986

[109] Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. 2006. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE Symp. on Architecture for networking and communications systems*. ACM. doi:10.1145/1185347.1185360

[110] Han Yu, Aiping Li, and Rong Jiang. 2019. Needle in a Haystack: Attack Detection from Large-Scale System Audit. In *2019 IEEE 19th Int'l Conf. on Communication Technology (ICCT)*. IEEE. doi:10.1109/icct46805.2019.8947201

[111] 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 (jun 2023), 1–28. doi:10.1145/3589297

[112] 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. arXiv:2504.12251 [cs.DB] https://arxiv.org/abs/2504.12251

[113] Ling Zhang, Shaleen Deep, Jignesh M. Patel, and Karthikeyan Sankaralingam. 2025. Regular Expression Indexing for Log Analysis. Extended Version. arXiv:2510.10348 [cs.DB] https://arxiv.org/abs/2510.10348

[114] Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao. 1998. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems* 23, 4 (Dec. 1998), 453–490. doi:10.1145/296854.277632

Received April 2025; revised July 2025; accepted August 2025