Exploiting Structure in Regular Expression Queries

LING ZHANG, University of Wisconsin-Madison, United States
SHALEEN DEEP, Microsoft Gray Systems Lab, United States
AVRILIA FLORATOU, Microsoft Gray Systems Lab, United States
ANJA GRUENHEID, Microsoft Gray Systems Lab, United States
JIGNESH M. PATEL, University of Wisconsin-Madison, United States
YIWEN ZHU, Microsoft Gray Systems Lab, United States

Regular expression, or regex, is widely used to extract critical information from a large corpus of formatted text by finding patterns of interest. In tasks like log processing, the speed of regex matching is crucial. Data scientists and developers regularly use regex libraries that implement optimized regular expression matching using modern automata theory. However, computing state transitions in the underlying regex evaluation engine can be inefficient when a regex query contains a multitude of string literals. This inefficiency is further exasperated when analyzing large data volumes. This paper presents **BLARE**, *Blazingly Fast Regular Expression*, a regular expression matching framework that is inspired by the mechanisms that are used in database engines, which use a declarative framework to explore multiple equivalent execution plans, all of which produce the correct final result. Similarly, BLARE decomposes a regex into multiple regex and string components and then creates evaluation strategies in which the components can be evaluated in an order that is not strictly a left-to-right translation of the input regex query. Rather than using a cost-based optimization approach, BLARE uses an adaptive runtime strategy based on a multi-armed bandit approach to find an efficient execution plan. BLARE is also modular and can be built on top of any existing regex library. We implemented BLARE on four commonly used regex libraries, RE2, PCRE2, Boost Regex, and ICU Regex, and evaluated it using two production workloads and one open-source workload. BLARE was 1.6× to 3.7× faster than RE2 and 3.4× to 7.9× faster than Boost Regex. PCRE2 did not finish on one of the workloads, but on the remaining two workloads, BLARE improved the performance of PCRE2 by 3.1× to over 100×. For the open-source dataset, BLARE provided a speed up of 61.7× for ICU Regex. BLARE code is publicly available at https://github.com/mush-zhang/Blare.

CCS Concepts: • **Information systems → Structured text search**; *Database query processing*.

Additional Key Words and Phrases: regular expression, adaptive query processing.

ACM Reference Format:
Ling Zhang, Shaleen Deep, Avrilia Floratou, Anja Gruenheid, Jignesh M. Patel, and Yiwen Zhu. 2023. Exploiting Structure in Regular Expression Queries. *Proc. ACM Manag. Data* 1, 2, Article 152 (June 2023), 28 pages. https://doi.org/10.1145/3589297

1 INTRODUCTION

The ability to collect and analyze large volumes of log data is crucial for large-scale system management to facilitate monitoring, process management, outlier detection, security, and customer support. For example, for large-scale cloud providers, operational logs are a critical source of

Authors' addresses: Ling Zhang, University of Wisconsin-Madison, Madison, Wisconsin, United States, ling-zhang@cs.wisc.edu; Shaleen Deep, Microsoft Gray Systems Lab, United States, shaleen.deep@microsoft.com; Avrilia Floratou, Microsoft Gray Systems Lab, United States, avrilia.floratou@microsoft.com; Anja Gruenheid, Microsoft Gray Systems Lab, United States, anja.gruenheid@microsoft.com; Jignesh M. Patel, University of Wisconsin-Madison, Madison, Wisconsin, United States, jignesh@cs.wisc.edu; Yiwen Zhu, Microsoft Gray Systems Lab, United States, zhu.yiwen@microsoft.com.

2836-6573/2023/6-ART152
https://doi.org/10.1145/3589297

information to help them understand usage characteristics and understand what enhancements may be needed to improve the performance and security of their cloud products.

A key hurdle in effective log analytics is the unstructured nature of logs, which renders them unsuitable for analysis using an RDBMS [45]. Instead, log analysis is routinely performed using a data science approach. Typically, a data scientist obtains operational logs, performs exploratory data analysis in a high-level language (such as Python), followed by querying the logs to extract structured information from the logs. The output is then used as input for downstream processing tasks. We demonstrate two examples, which motivated this work, to illustrate this scenario.

Example 1. *Alice is a data scientist who is analyzing the deployment of virtual machines (VMs) in the cloud. She wants to identify how frequently VMs are being redeployed due to resizing within clusters with names starting with* ***us-east-X****. Her first step is to extract the relevant VM ids using the regex:* `replacing VM (VmId=([a-z0-9\-]+), VmName=us-east-X-([a-z0-9]+)-vm` *. Once the VM ids have been identified, Alice can analyze when the VMs were created and replaced.*

Example 2. *Bob is a data scientist who is responsible for identifying recurring customer issues for a cloud database application. In particular, Bob wants to understand how frequently errors happen because of failed I/Os. He quickly identifies that the I/O object and error messages can be extracted via the regexes* `Read on \"(.+)\" failed: (.+)` *and* `Write on \"(.+)\" failed: (.+)` *respectively. Bob runs these expressions on the logs to find the objects on which I/O requests failed. Following this step, Bob can dive deeper into the analysis of how frequently I/Os fail due to software issues, hardware failures, etc.*

Several prior works have looked at speeding up such analytics pipelines by proposing new architectural frameworks for streaming data [19], proposing novel distributed systems [37], developing auto-tuning knobs in existing systems [57], and building accelerated type systems [74]. Most of these systems use regexes and their extensions as core components. Despite its importance, accelerating regex matching for analytics has received scant attention in the database community. In both the examples above, the first step in identifying the parts of the log that contain the relevant information (VM ids for Example 1 and error messages/objects for Example 2) is a key bottleneck for data scientists. Further, such regex computation is often done with a human-in-the-loop, as at times the data scientist has to experiment with different patterns. Thus, there is a critical need to speed up the evaluation of regexes.

Analytics frameworks and data management solutions routinely use existing regex libraries as pluggable modules under the hood. The two common state-of-the-art libraries that are used in practice for regex are `RE2` and `PCRE2`. `RE2` is an open-source library developed by Google and powers regex matching for spam filtering, code search, and indexed document search in Gmail, Google Sheets, etc. Popular data exploration tools, such as Microsoft's Kusto, also use `RE2` for regex evaluation internally in their platform [65]. `RE2` has a low memory footprint and offers a predictable running time. `PCRE2` is a well-known library that is significantly more expressive than `RE2` and has comparable performance with respect to `RE2`. Both MariaDB and PostgreSQL use `PCRE2` as the default regex evaluation library and `RE2` is available as an alternative library. Essentially, regex evaluation frameworks are present in a multitude of data settings including data exploration and cleaning, intrusion detection, and pattern discovery.

Performance Improvement Challenges and Desiderata. We make four key observations that form the basis for the desiderata for our solution.

R.1 It is important that our approach is engine agnostic, i.e., the benefits of our approach extend to multiple engines. We make the conscious choice of not modifying the code of the underlying regex engine in order to ensure that the benefit of our ideas extend to any engine. Thus, as

improvements are made to existing regex engines, our solution can leverage these benefits immediately.

R.2 We want our solution to be extensible. It should be easy to integrate new optimizations into the framework and have no long term dependency on any specialized hardware or software. (This aspect allows our method to be used in conjunction with methods such as using FPGAs to run an existing regex engine.)

R.3 While we want to speed up the entire regex workload, it is also important to make sure that we do not introduce large regressions for any specific query.

R.4 We should not assume availability of statistics or catalogs about the input logs or the workloads. A significant fraction of log analytics is performed over adhoc logs and queries written by data scientists on a need to know basis. Therefore, it is important to offer good performance for this scenario.

Solution Overview. To address the goals outlined above, we propose a lightweight framework that sits on top of a regex evaluation engine and uses it in a blackbox fashion. We exploit the observation that alternate ways of substring matching in regex can bring significant performance gains. Most regex matching solutions are based on automata theory, where a regex query is compiled into an NFA (non-deterministic finite automata) or a DFA (deterministic finite automata). Log lines are fed to the automata character by character until the automata reaches an accepting state. At each step, a character from the input is consumed by the compiled regex and the internal automaton keeps track of the current state after transition as well as the list of potential next states. However, using automata to perform regex matching also comes at a performance cost. Each literal character in the input regex is converted into a state in the automata. Performing the state transitions and internal bookkeeping in the automata when matching the substring is an added overhead compared to directly performing a string match using standard string matching algorithms. In fact, the overhead can be as high as 3× (we demonstrate this in Section 2.3). This observation suggests that a principled way of shifting the computation from an automata-based evaluation to a string matching paradigm can potentially bring substantial performance gains.

1.1 Our Contribution

In this paper, we present BLARE, a novel fast regex query evaluation framework that carefully exploits the inherent structure within a regex query and identifies the best evaluation strategy. More specifically, we make the following contributions.

1. ***A new framework for regex evaluation.*** BLARE is a novel framework that uses an underlying regex evaluation engine as a blackbox and improves its performance. In particular, we make no assumptions about the underlying engine **(R.1)**, the presence of any specialized hardware **(R.2)**, or statistics **(R.4)**. BLARE complements existing regex evaluation engines by using a regex splitting strategy that partitions the regex and carefully moves the computation out of the underlying engine to improve query performance. To demonstrate the applicability of BLARE with respect to regexes seen in the wild, we analyze regexes found in 14.5M publicly available notebooks on GitHub published between 2017-2020. Our analysis shows that at least 35% of all the regexes used can potentially benefit from BLARE (see Section 2.3). Obtaining a performance improvement requires examining several non-trivial implementation considerations, which we discuss in Section 4.

2. ***Learning on a budget.*** We use lightweight mechanisms to identify whether our new evaluation strategy is better than running the entire regex as-is on the regex engine. This approach contextually picks the right execution strategy, which includes simply executing the regex query unmodified in

the underlying regex engine. Using a learning-based technique is key to achieving our goal of not introducing large regressions for queries in our workload **(R.3)**.

3. *Experimental Evaluation.* To demonstrate that BLARE is engine agnostic and widely applicable to several engines, we implement it alongside four existing regex engines: RE2, PCRE2, ICU Regex (the regex engine used in MySQL [72]), and Boost Regex (the regex engine used in Lucene++ [58] and C++ standard library [78]).

We highlight two key results from our empirical evaluation. First, BLARE significantly improves performance across all considered workloads across all the engines. BLARE provides an average improvement of 4.0× (and up to 7.9×) for the two production workloads over all engines. For the open-source workload, the improvement ranges from 1.6× to 168.3× across the four engines. Second, the overhead of the learning component in BLARE is minimal. For regex queries where our rewriting is slower, BLARE is able to quickly learn on-the-fly and pick an efficient evaluation strategy.

Overall, our experiments show that BLARE is a simple and effective method to speed up the processing of regex queries. Further, to the best of our knowledge, our work is the first step towards realizing a general, cost-based optimizer framework that targets regex evaluation for log analytics.

The remainder of the paper is structured as follows. Section 2 presents the relevant background and existing known optimizations. Section 3 presents the framework and main ideas behind BLARE, and Section 4 describes other design parameters considered in developing BLARE. An extensive empirical evaluation on real-world (production and academic) datasets is presented in Section 5. We discuss the related work in Section 6 and conclude in Section 7.

2 BACKGROUND

In this section, we begin with the notation and the class of regex queries that are considered in the paper. Given a finite alphabet Σ, the syntax of regex can be defined by:

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

where $\emptyset$ denotes the empty language, ε is the language containing the empty string $\{\epsilon\}$, $c \in \Sigma$ denotes the language consisting of the single character $\{c\}$, $R \mid R$ is the union of two languages, $R \cdot R$ is the concatenation of two languages, and $R*$ denotes zero or more concatenations of the language R. We denote by $\mathcal{R}$ the set of all possible regular expressions. The language of regular expression is defined by $\mathcal{L} : \mathcal{R} \rightarrow 2^{\Sigma*}$:

$$\begin{aligned}
\mathcal{L}(\emptyset) &= \emptyset \\
\mathcal{L}(\varepsilon) &= \epsilon \\
\mathcal{L}(R_1 \mid R_2) &= \mathcal{L}(R_1) \cup \mathcal{L}(R_2) \\
\mathcal{L}(R_1 \cdot R_2) &= \mathcal{L}(R_1) \cdot \mathcal{L}(R_2) \\
\mathcal{L}(c) &= \{c\} \\
\mathcal{L}(R*) &= \mathcal{L}(R)*
\end{aligned}$$

The concatenation operator (denoted by $\cdot$) over languages is defined as $L_1 \cdot L_2 = \{\omega_1\omega_2 \mid \omega_1 \in L_1, \omega_2 \in L_2\}$. The closure operator (denoted by $*$) is defined as $L* = \bigcup_{i \geq 0} L^i$ where $L^0 = \{\epsilon\}$ and $L^i = L \cdot L^{i-1}$ for any $i > 0$.

We define $\texttt{literals} \leftarrow \Sigma*$. Given a regex R, we can rewrite the regex as,

$$R \leftarrow (\texttt{prefix}\ S\ \texttt{suffix})$$

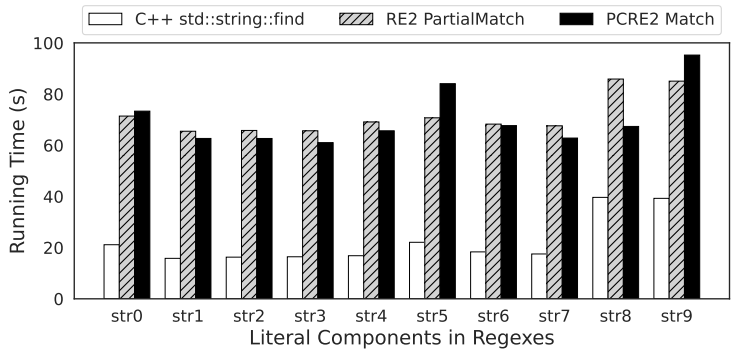


Fig. 1. Matching time comparison on a production dataset. Using C++'s std::string::find(), RE2's PartialMatch(), and PCRE2's Match()

where `prefix` and `suffix` are `literals`[1]. In other words, we can *decompose* the regex by extracting the letters at the beginning and end of the expression. Note that both the `prefix` and the `suffix` can be empty. As an example, the regex `[0-1]*` has no prefix or suffix but the regex from Example 1 has a `prefix` `replacing VM (VmId=` and a `suffix` `-vm` . In practice, there are several standards defined for writing concise regexes. POSIX defines the basic regex (BRE) syntax that uses metacharacters such as `^` and `$` to signify the beginning and the end of a term. The metacharacters `?` and `+` are used to signify that the token preceding them is matched zero or more times and one or more times respectively. The syntax also defines shorthands such as `\d` (to denote `[0-9]`), `\w` (to denote `[a-z]`), etc. We refer the reader to the full standard specification [11] for more details.[2]

Grouping syntax in regexes, which is part of the POSIX standard [42], allows extracting the matched content from the input logs. POSIX standard allows creating groups by placing a regex within the parentheses. Any data that matches the subregex within the parentheses is *captured* and can be returned to the user. For example, the regex in Example 2 `Read on \"(.+)\" failed: (.+)` contains two groups and will only match content delimited by `Read on \"` and `\" failed:` for the first group. The second group will contain matched content delimited by `\" failed:` and the end of the log line. In this paper, unless specified otherwise, we use capture groups in our experiments to extract the matched content and return that to the user. For use cases that only require counting log lines containing a match or finding the longest match, regex libraries provide specialized methods. We show that our approach is also beneficial in this setting (see Section 5.5.1).

2.1 Evaluation Techniques

Regex evaluation can be broadly classified into two classes. The first regex evaluation method uses an **NFA**. An NFA implements a space-efficient automaton. A defining characteristic of an NFA is that it is possible to reach multiple states from any given state. In the worst case, it is possible that every state may be reachable from a given state. Thus, each character requires $O(m)$ memory lookups, where m is the number of states in the automata. The benefit of an NFA is its compactness compared to a DFA. On the downside, an NFA is generally slower than a DFA.

A **DFA** is a special case of NFA where each state and input can transition to exactly one state. This also means that a DFA can be exponentially large compared to an NFA, but regex evaluation is

[1]We refer to these as string literals and literal components interchangeably.
[2]Operators such as look-ahead and look-behind are not a part of the POSIX API.

faster because every transition takes $O(1)$ amount of time. This is the main reason why a DFA is the preferred implementation for regex evaluation unless the space overhead is a concern.

String matching is a special case of a DFA, where the automaton is acyclic when viewed as a graph and there is exactly one start and one accepting state. In practice, string matching is performed using specialized algorithms such as the celebrated KMP [46] and Aho-Corasick [1] algorithm. String matching algorithms run in time linear in the size of the input string and the searched pattern. Most high-performance implementations of string matching make use of hardware features such as SIMD, vectorization, etc.

In order to demonstrate the performance gap between specialized algorithms and DFA to perform string matching, we develop a microbenchmark where the regex contains only string literals. Figure 1 shows the performance when executing the regex on a production cloud service log dataset[3]. The regexes vary in length from 5 (str4) to 27 (str6 and str7) with the mean length being 16.3 characters. They mostly contain letters of the English alphabet and some less frequently occurring characters such as underscore, round brackets, colon, and hyphens. For these regexes, string matching is consistently $2-3\times$ faster compared to the best of `RE2` and `PCRE2`. This experiment illustrates the potential benefit of moving computation out of automata-based matching for string matching.

2.2 Known Optimizations in Existing Systems

The most common optimization used that is relevant to our work is the idea of prefiltering. The basic idea of prefiltering is to eliminate from consideration any input that does not contribute to the result. It is implemented as an initial inexpensive pass before the more expensive computation is performed. Instead of evaluating the regex directly, most systems apply some form of prefiltering-based ideas. Automata-based regex matching is performed only if the input passes the prefilter. Both Snort [22] and Suricata [71], two popular open-source intrusion detection systems, use string-based prefiltering extensively on network packets to find suspicious packets to do more intensive checking.

`RE2` also employs a prefiltering approach. If the regex begins with a sequence of string literals (we call this a prefix), then `RE2` takes the first and the last character of the prefix and searches the input for a candidate starting point that has the same first and last character as the prefix and the same length as the prefix [32], a variant of the classic Boyer-Moore [12] algorithm. This approach is memory efficient and has less overhead for long prefixes.

`PCRE2` also contains string-matching-based optimizations. Whenever a character is detected, `PCRE2` matches it using a character comparison. For example, the regex `[^a]` is processed specially since it is the only character within the group [38]. It makes extensive use of just-in-time compilation and has sophisticated algorithms to avoid backtracking when evaluating regexes [38].

2.3 Input Properties for Log Analysis

In this subsection, we discuss some of the real-life properties of the logs and regex queries that we have observed in our production settings and open-source datasets. To understand how prevalent string literals in regexes are, we analyzed 14.5 million publicly available notebooks on GitHub authored between 2017-2020. We found that at least 35% of the regexes (out of 200,000 unique regexes) contain at least one literal component. We also analyzed ~9000 regexes and their logs across two different applications – analysis of cloud VMs and finding errors in a production cloud database server logs. We make two key observations. First, the query structure of the regex queries is often simple. Most expressions contain long string literal components and the actual regex that requires automata-based execution does not have any nesting or complex unions. Most of these

[3]We anonymize the strings to avoid revealing potentially sensitive information.

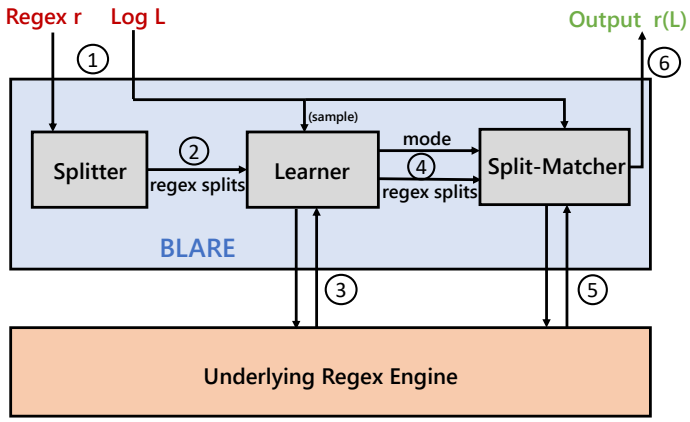


Fig. 2. BLARE architecture diagram. It takes the log and the regex as the input, and returns the query result as output.

regex queries are also written by humans and thus, readable. Second, the generated logs contain messages from multiple system components that are mixed together. Since the input extraction for downstream tasks is usually not repeated on a given log, it is uncommon to group the log messages based on the system component generating it. Doing such a partitioning of the log also makes it harder to perform temporal analysis on understanding the sequence of events logged by other components. Further, log partitioning means that a post hoc log stitching via index lookups on the timestamp may be required to find log lines from other components around the timeframe of interest, which is an expensive operation. As an example, a single general log for a cloud-hosted VM stores information about clients connecting and disconnecting, query statements, network information, system process information, etc. Once the user/analyst finds errors in the log, they may want to see what was happening on the VM a few seconds before the error to gather additional useful clues. The overall implication of having a single log is that for most regex queries, only a small fraction of log lines contain any potential matches.

3 FRAMEWORK AND TECHNIQUES

In this section, we begin with an overview of BLARE. Figure 2 shows the overall architecture of BLARE. It consists of three main parts: namely, a *splitter*, a *split-matcher*, and a *learner*. The first part is the splitter that takes the input regex and decomposes it into multiple components by performing a single scan of the regex and extracting the string literals. The splitter is run exactly once when the regex is submitted and has negligible overhead.

The split-matcher is responsible for executing the decomposed regex(es) generated by the splitter on the log data. For each log line, it first matches the literal components. If the log line contains matches to each literal component, then the regex engine is invoked for a full evaluation. In some cases, directly running the decomposition generated by the split-matcher may introduce an unintentional slowdown. As an example, consider the regex from Example 1 and the log line $\ell =$ `replacing VM \(VmId=t56er3, VmName=us-east-X-az09re4-vm`. The split-matcher checks that the `prefix` and the `suffix` of the regex are present in the log line and then runs the regex on the line using the engine. However, for this log line, string matching is unable to filter the log line, and we need to evaluate the regex on the engine. In the worst case, all the log lines may contain the string literals that we want to prefilter, and thus, using the split-matcher may not be better than using the engine directly. Therefore, we introduce a learning component to identify when

a split-match-based strategy is better than running the regex directly on the engine. Since the learning step is an overhead, we employ a tunable knob to control the time spent in learning the best strategy. We present our technique in more detail in the following sections.

3.1 Regex splitting

The splitting phase decomposes a regex R into an equivalent expression of the form: `prefix` S `suffix`. We refer to this form as the 3-way split when the decomposition has three parts (two string literals and a regex component in the middle). The decomposition is done in a way so as to maximize the length of the string literals. There are other possible decompositions of interest here. For instance, one can decompose R into `prefix` T where $T \leftarrow S$ `suffix`. This is a 2-way split. It may also be possible to decompose S recursively into S_1 `middle` S_2 to generate a 5-way split of the form `prefix` S_1 `middle` S_2 `suffix`. The 5-way split is called so because there are 5 components in the decomposition, three string literals (`prefix`, `middle`, and `suffix`) and two regex components (S_1 and S_2) . As we will see later, a 3-way split is usually the preferred splitting method. In order to understand why a 3-way split is often the method of choice, we need to understand the costs and the associated benefits. For each string literal component in the decomposition, we need to scan the log line to identify if the string literal is present. If the log line contains all the string literal components, then the engine runs its regex matching on a substring extracted from the log line. Assuming the fraction of log lines left after filtering through the string literals is small, the benefit of doing containment checks is the fact that we could avoid running the regex matching on the engine for a large fraction of the log lines. We translate this intuition into a cost model in Section 3.2.1.

In some cases, a 3-way split may have an empty prefix and/or suffix . In that case, we may need to recursively decompose until we find more string literals. Therefore, we generate a full recursive decomposition of the regex that extracts all string literals in the query. We call this the multi-way split of the regex.

Example 3. *Consider the regex shown in Example 1. A 3-way split of the regex will set the* `prefix` *as* `replacing VM \(VmId=(` *and the suffix as* `)-vm` *. The middle component S is* `[a-z0-9\-]+) VmName=us-east-X-([a-z0-9]+` *. In a 5-way split, S is further split into* S_1 *is* `[a-z0-9\-]+` *,* S_2 *is* `[a-z0-9]+` *, and* `middle` *is* `) VmName=us-east-X-(` *. Note that* `middle` *can no longer be split any further. Therefore, a 5-way split is also the full recursive decomposition for the regex.*

3.2 Split-Matcher

The split-matcher takes as input the 3-way split, the multi-way split decomposition generated by the splitter, and the regex that the user is trying to match. It runs in three modes: *direct*, *3-way-split*, and *multi-way-split*. In the direct mode, the split-matcher takes the regex given by the user and evaluates it on the underlying engine. Algorithm 1 shows the execution steps of the split-matcher. In the 3-way-split mode, for each log line, we first verify if the `prefix` and the `suffix` are both present in the log line. If this is true, we extract the substring from the log line between the `prefix` and the `suffix`. Then, we execute the regex s, which is the regex component in the 3-way split of r, on the engine with the substring as the input. Similarly, in the multi-way-split mode, a generalization of 3-way-split mode, we verify if all string literals are present in the log line and then execute the regex by extracting the substrings. Next, we create a cost model to characterize the behavior of these split modes compared to the direct mode.

3.2.1 Cost Model. In this section, we show that the 3-way split is the best strategy for decomposing a regex in most cases. Let $|\ell| + f \cdot |s|$ denote the cost of running a specialized string matching algorithm on a given log line ℓ for a string s, and f is the multiplicative constant that captures the overhead of processing string s by a string matching algorithm. Let $\Theta(r)$ denote a constant

Algorithm 1: SPLIT-MATCHER

```
Input : Log L, 3-way split of r, multi-way split of r,
        mode ∈ {direct, 3-way-split, multi-way-split}.
foreach l ∈ L do
    if mode = direct then
        execute r on l on the engine
        return
    if mode = 3-way-split then
        parameterized-split-match(k = 2)
    if mode = multi-way-split then
        parameterized-split-match(k = # string literals in multi-way split)
procedure parameterized-split-match(k)
    /* str_i is i^th string literal in the split used. Assume prefix/suffix is non-empty. */
    c_list ← empty array , i ← 0, c_{-1} ← −1
    do
        while i < k do
            if c_i ← l.find(str_i, pos = c_{i−1} + 1) then
                c_list[i] ← c_i
                if i = 0 then c_{-1} ← c_0
                i ← i + 1
            else
                if i = 0 then return
                i ← i − 1
        foreach c_j ∈ c_list[0 : −1] do
            /* s_j is j^th regex in the split of r. */
            match s_j on substring l.substr(c_j + len(str_j), c_{j+1}) using engine
            if not matched i ← j + 1 and break
    while input not exhausted
```

such that $\Theta(r) \cdot |\ell|$ is the cost of executing a given regex r using the engine. In Section 2.1, the gap between running a string matching algorithm versus using a DFA to match is about 3×. Based on this empirical observation, we are informally going to assume for the rest of the paper that $1 \leq \Theta(r) \leq 3$. Let c denote the average number of characters matched in a string literal and let lsize be the average string literal size. Let $\sigma_1, \sigma_2, \ldots, \sigma_k$ denote the selectivity of the k string literals in the decomposition of r over the input log and σ is the overall selectivity. Note that $k = 1$ for 2-way split and $k = 2$ for 3-way split. The cost of running the split-matcher on a log line ℓ is:

$$\underbrace{\texttt{SMCost}(r,k) = |\ell| + \sum_{i=1}^{k} f \cdot \underbrace{(c \cdot (1-\sigma_i) \cdot \prod_{j=1}^{i-1} \sigma_j}_{\text{part one}} + \underbrace{\prod_{j=1}^{i} \sigma_j \cdot \text{lsize})}_{\text{part two}}}_{\text{string matching cost}} + \underbrace{\underbrace{2 \cdot \sigma \cdot \sum_{i=1}^{k-1} |\ell_i'|}_{\text{substring extraction cost}} + \underbrace{\Theta(r) \cdot \sum_{i=1}^{k-1} |\ell_i'| \cdot \sigma}_{\text{running on engine}}}_{\text{total regex evaluation cost}}$$

Here, ℓ_i' denotes the substring between some occurrence of the i^{th} and the $(i+1)^{th}$ string literal. The string matching cost is divided into two parts. In the first part, for each string literal, c characters of the literal are matched in the log line on average. However, for the i^{th} string literal,

Table 1. Performance comparison of Split-Matcher implemented with RE2 on a production dataset.

Regexes	Run Time (s)				
	$k = 1$	$k = 2$	$k = 4$	$k = 6$	$k > 6$
A	2.03	1.99	2.27	2.21	2.22
B	2.03	1.99	2.27	2.21	2.22
C	2.10	2.04	2.30	2.24	2.25
D	2.00	1.94	2.22	2.16	2.20
E	1.84	1.80	2.05	2.00	2.00
F	2.09	2.03	2.32	2.24	2.26
G	2.08	2.02	2.31	2.24	2.26
H	2.40	1.99	2.18	2.08	1.99

we only perform matching if all previous $i-1$ literals also matched. This is accounted for by the product of selectivities of the $i-1$ string literals. In the second part, since the previous $i-1$ literals matched, it means we performed $(i-1)\cdot$ lsize letter matches too. The i^{th} string literal is also present in the log line with probability σ_i and thus contributes an additional lsize character matches. The cost of extracting the substrings for executing on the engine is set as twice the sum of their lengths since we need to read the substrings and create a copy. This is done only when the log line has all the string literals we are looking for. The cost of running r directly on the underlying engine is: $\texttt{EngineCost}(r) = \Theta(r)\cdot|\ell|$. Therefore, we would obtain a performance advantage only when $\texttt{SMCost}(r,k) \leq \texttt{EngineCost}(r)$. Based on the analysis of our datasets, we fix lsize $= |\ell|/10$, $c = |\ell|/100$, $\sigma_1 = \sigma_2 = 0.3$, and $\sum_i |l_i'| = |l|/2$. We set $f = 3$ since every character match of string s in ℓ requires two more operations (checking the character in s with the log line and advancing the pointer) besides reading. For $k = 1$, $\texttt{SMCost}(r, k=1) \approx 1.69|\ell|$ and $\texttt{EngineCost}(r) = 2|\ell|$. However, $\texttt{SMCost}(r, k=2) \approx 1.51|\ell|$, which means that a 3-way split is better than a 2-way split.

Next, consider a multi-way split. In other words, the case when $k \geq 3$. In this case, the overall selectivity σ does not change for most queries in our workloads and thus, the regex evaluation cost term remains the same. We also observed that the substring extraction cost also remains close to $2\cdot\sigma\cdot|\ell|/2$ as there are few lines that contain repeated string literals. However, the string match cost is strictly larger; thus, $\texttt{SMCost}(r, k \geq 3) > \texttt{SMCost}(r, k=2)$. We note that there is one scenario where a multi-way split is better than a 3-way split. If the 3-way split has a `prefix-suffix` combination with low selectivity or either one of them is empty, then the split-matcher has a lower cost for a multi-way split. Having string literal(s) with low selectivity means that the overall selectivity is not high enough and a recursive decomposition is beneficial.

To validate our cost model, we use a production dataset and select 8 regexes that can be split into more than 7 components. We compare their performance (trimmed average by removing the slowest and fastest runs over 10 experimental runs) for varying values of k, and show these results in Table 1. The 3-way split ($k = 2$) approach has the best performance for *all* regexes. Both $k = 1$ and $k \geq 3$ are more expensive, in line with the prediction from the cost model, by as much as 20%. We provide further evidence for the validity of our informal cost model in Section 5.4.

3.3 Learner

Since we do not assume any statistics are available apriori, it is not possible to know which of the direct mode or split-match mode is the better strategy. The goal of the learner is to identify an efficient strategy *on-the-fly*. Given a regex r in the workload W, let $\text{opt}(r)$ denote the optimal mode

for running it and let s be a fixed decomposition of r. Our goal is to minimize the overall regret, *Regret*, in the online setting, by choosing a strategy (denoted by chosen(r)) that is close to optimal. Formally, we can define *Regret* as follows:

$$Regret = \sum_{r \in W} \Big(\text{Cost}\big(\text{Split-Matcher}(L, s, \text{chosen}(r))\big) - \text{Cost}\big(\text{Split-Matcher}(L, s, \text{opt}(r))\big) \Big)^2$$

Multi-armed bandits (MABs). The regret minimization problem in the above equation is a multi-armed bandit [102] problem. An agent must maximize their reward (i.e., minimize regret) by repeatedly selecting from a fixed number of arms. The agent first receives some information, and must then select an arm. Each time an arm is selected, the agent receives a payout. The payout of each arm is assumed to be independent given the contextual information. After receiving the payout, the agent receives a new context and must select another arm. Each trial is considered independent. In the case of BLARE, the *arm* is one of the three modes described in Section 3.2, and the chosen arm in a particular iteration informs our choice about the next iteration. Over time, the learning agent improves its selection and gets closer to choosing optimally (i.e., minimizing regret). Doing so involves balancing exploration and exploitation: the agent must not always select a mode randomly (as this would not help to improve the performance), nor must the agent blindly use the mode that gives good performance initially, without exploring the other choices.

Thompson Sampling. One solution to the MAB regret minimization problem is Thompson sampling [76]. Intuitively, Thompson sampling works by building up experience (i.e., past observations of split-matcher mode choices and their performance). It models the reward for each arm choice with a distribution over its possible values informed by the uncertainty in its estimate, as compared to treating it as a point estimate. In other words, if one of the arms has a high reward but has not been tried many times (i.e. the uncertainty is high), the distribution will flatten to account for this fact. This technique promotes the exploration of other arms, rather than choosing the arm with the highest reward all the time.

To model the rewards for each arm, we use a beta distribution $B(\alpha, \beta)$. At the beginning of the learning phase, $\alpha = \beta = 1$ which leads to the priors being uniform distribution (modeling the fact that we have no prior so any strategy is equally likely to be the best). After every iteration, α or β is incremented depending on whether the chosen arm has been the arm with the maximum rewards so far. Note that the learning procedure runs for a fixed number of iterations. At the end of the learning phase, we pick the arm with the highest reward as the mode in which the split-matcher must be run for the regex under consideration.

Ensemble Method. We measure the cost of a mode running a regex r on a specific log line ℓ with the actual running time. However, the measured data can be noisy, as the running time can be impacted by factors like other running processes and system glitches. To avoid being affected by noisy data and choosing the wrong mode, we employ ensemble learning, a well-known technique to increase the robustness of the learner [99]. We begin by taking a random sample of the input log. The learning component of BLARE then splits the measurement sample into 10 folds and generates 10 separate MABs. The majority vote of choices of arms among the 10 MABs is selected. We discuss the accuracy of our ensembled MAB learner in Section 5.4.

4 DESIGN CONSIDERATIONS

There are several considerations that must be taken into account when designing an effective solution for regex query evaluation. Continuing our discussion from Section 1, below we outline some of these considerations that informed our design choices.

4.1 Extensibility and Simplicity

One of the key choices we made was not to make changes to the code of underlying engine as making changes to widely used libraries has the potential to introduce bugs and regressions. Implementing BLARE as a standalone software artifact that uses the engine as a blackbox also increases the chances of adoption since it is lightweight, easy to use, and modular. Further, this choice also allows multiple engines to benefit instantly from our optimizations. The simplicity of our techniques and a small codebase (< 1000 LOC) makes it easy to understand and aids explainability.

4.2 Minimize Learning Overhead

Keeping the learning overhead low is also important. The learning overhead increases linearly in the number of different modes that we can choose from. Therefore, we deliberately keep the number of modes in BLARE to be small (three). Note that BLARE is easily extensible by adding other evaluation strategies. One possible improvement is to choose the optimal order of `prefix` and `suffix` matching on the log line. We revisit this point in Section 5.5.2. We also minimize the learning overhead by early stopping the MAB. In our setting, we want to find out which *arm* is generally better rather than finding the actual distribution of lines that runs faster with the three *arms*. MAB in BLARE will stop as soon as one *arm* achieves at least half of the total iteration number of wins. This mechanism allows BLARE to drop out of the learning stage and proceed to use the chosen strategy earlier.

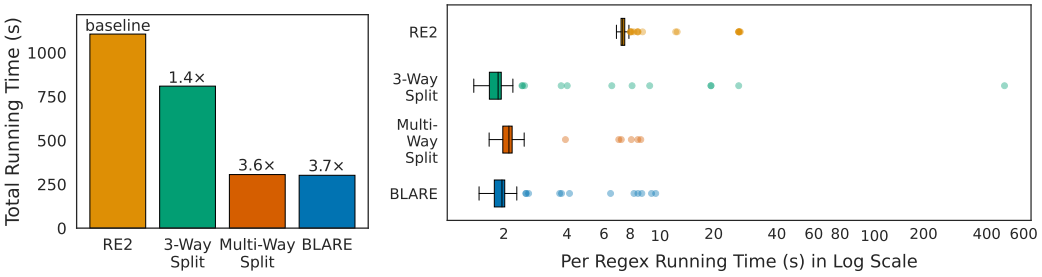
4.3 Prefix and Suffix Sizes

The selectivities of the prefix and the suffix strings play a critical role in determining which mode of execution is chosen by the learner. It is not uncommon to have a single character suffix in regex queries (such as ending with a dot). Such suffixes have low selectivity and the split-match mode does not offer any benefit in this case. However, highly selective characters (and words starting with them) such as the character z filter out a large fraction of the log. Therefore, string literal sizes are not always a reliable indicator of selectivity information. Instead of estimating the selectivity of string literal(s) (a hard problem), we simply let the learner component determine the best strategy by comparing the three different execution modes. For regexes that do not contain a prefix and/or suffix, multi-way splitting performs much better.

5 EVALUATION

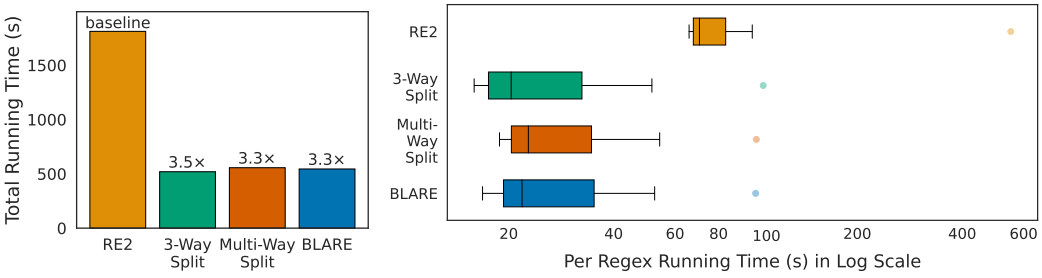
We implemented BLARE on top of four state-of-art regular expression libraries that are used in log analytics: Google's RE2, PCRE2, `Boost Regex`, and `ICU Regex`. `ICU Regex` (Regex component of International Components for Unicode) is the regex engine used in MySQL [72] and `Boost Regex` is the underlying engine used for regex evaluation in Lucene++ [58], a C++ migration of Java-based search engine Lucene [84]. `Boost Regex` has also been part of C++ standard library since C++11 [78]. We then used these implementations to evaluate the impact of BLARE. In particular, we seek to answer the following questions:

- **Q.1** What is the performance improvement with BLARE (cf. Section 5.2)?
- **Q.2** What is the overhead of using the multi-armed bandit (MAB) approach (cf. Section 5.3)?
- **Q.3** How does the performance of the 3-way split compare with the multi-way split strategy (cf. Section 5.4)?
- **Q.4** Does BLARE offer any performance improvement for other regex queries (such as counting matches and finding the longest match) and is BLARE extensible to incorporate more fine-grained splitting strategies (cf. Section 5.5)?



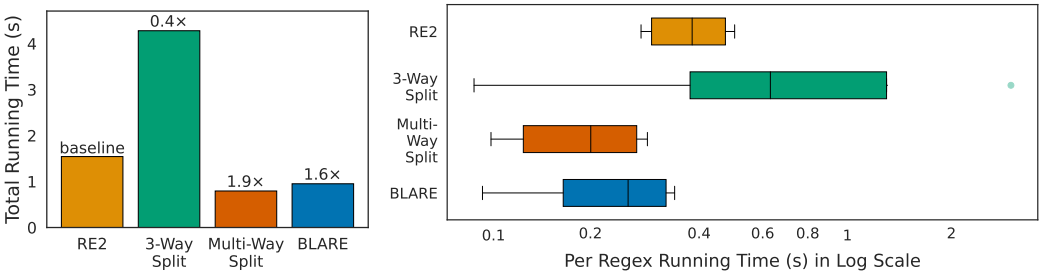
(a) Overall running time (b) Distribution of per regex running time

Fig. 3. BLARE vs. Multi-way split vs. 3-way split vs. RE2 on DB-X Workload



(a) Overall running time (b) Distribution of per regex running time

Fig. 4. BLARE vs. Multi-way split vs. 3-way split vs. RE2 on System-Y Workload



(a) Overall running time (b) Distribution of per regex running time

Fig. 5. BLARE vs. Multi-way split vs. 3-way split vs. RE2 on US-Accident Workload

5.1 Experiment Setup

For our evaluation we use 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 use a C++ wrapper, JPCRE2 [35] version 10.32.01, for PCRE2 version 10.40, with JIT compilation on and the latest (09-20-2022) version of RE2. We used the official C/C++ version of ICU library, ICU4C version 72.1 and Boost library version 1.65.1.0ubuntu1.

Table 2. Statistical information of literal components of regexes in **DB-X**, **System-Y**, and **US-Accident**

workload		DB-X	System-Y	US-Accident
# literal per regex	mean	3.2	1.4	1.8
	median	3	1	2
mean # char of	prefix	33.1	14.5	4.7
	suffix	72.8	3.9	0
	literal	39.9	12.2	5.1
# regex with	only prefix	3	11	3
	only suffix	5	0	0
	both	123	7	0
	none	1	0	1

For simplicity, experiments are run in a single-threaded mode. In the experiments, we first load the full set of regexes and dataset into main memory as an array of strings, then match each regex query in the workload on the full array. Unless stated otherwise, we extract the first match of each regex query on each log line throughout Section 5.2 to Section 5.4. We present experimental results of other types of queries in Section 5.5. The captured substring in each log line is stored in a local variable. For each regex, we timed the duration of the matching from the first log line to the last log line for the workload under consideration. We run the matching process 10 times, and record the trimmed mean by removing the maximum and minimum running time for each method. For each workload, we report the running time of a specific method by summing over the trimmed mean running time of all the regexes. In the figures that follow we use this trimmed mean sum to report the average workload times in bar charts with speedup numbers. To better understand the running time of the individual queries in each workload, we show the distribution of the individual query run times (by using the trimmed mean of each query) as a box plot. For the learning component in BLARE, we fix the sample size of each run to be 0.001% of the log. We ensure that the sample always has at least 200 lines for learning.

5.1.1 Workloads. We use the following 3 real-world datasets and their corresponding query workloads for our experiments. Two of the datasets contain logs generated by production database systems while the third dataset is text based. The detailed statistical information of literal components in regexes in each workload can be found in Table 2.

DB-X. We obtained a production dataset from a large cloud provider containing 101,876,733 log lines generated by DB-X with a file size of 13.5 GB and 8, 941 regexes used in log analyzing tasks by data scientists, engineers, customer support, etc. We randomly sampled 132 regexes and evaluate those on the logs to create this workload, which we call **DB-X**. The average log line length in this workload is 138.5 characters. Regexes in this workload have large number of literal characters, with an average of 105 characters per regex and 39.9 characters per literal component. The majority of the regex queries have both a `prefix` and a `suffix`.

System-Y. This is a production dataset containing 890, 623, 051 log lines generated by System-Y, a popular data exploration tool used in the industry. The size of this dataset is 100 GB, and the workload has 18 regexes used for analysis tasks by data scientists. The average log line length is 116.2 characters. This workload contains regexes with smaller-sized literal components and fewer literal components per regex compared to **DB-X**. 11 of the regexes have at least 2 literal components and 7 have only prefixes.

US-Accident. This is an open-source dataset collected by Moosavi et al. [68]. This dataset contains 2, 845, 343 records of traffic accidents in the United States from February 2016 to March 2019 in a file of size 1.08 GB. We used the *accident description* strings in the dataset and the 4 regexes presented in their paper. The average line length is 66.6 characters. Regular expressions in this workload have the least number of literal characters across all workloads; three of them contain only prefixes, and one has no prefix or suffix.

5.1.2 Methodology. We compare the running time of BLARE with the three modes of Split-Matcher individually; i.e., we hard code the execution mode to a fixed value (direct, 3-way-split, multi-way-split) for every query in the workload. We record the overall execution time of the workload and collect execution metrics to construct the box plots. We also record the execution time for each regex to observe changes in running time for different strategies.

5.2 Performance of BLARE

In this section, we answer **Q.1** by evaluating BLARE on each of the four engines.

RE2 Performance. Figures 3a, 4a, and 5a show that for each of the three workloads described above, BLARE on top of RE2 improves the performance compared to direct execution on RE2. BLARE significantly reduces the running time of log analysis for the DB-X and the System-Y workloads, with a 3.7× reduction for the DB-X workload and a 3.3× reduction for the System-Y workload. For the US-Accident workload, which benefits from structural decomposition in BLARE but has generally shorter literal components, our framework still reduces the overall running time by 1.6×. For the DB-X and the US-Accident workloads, 3-way split is significantly slower than multi-way split. There is one regex in each of the two workloads without a `prefix` and a `suffix`, so the 3-way split falls back to using direct RE2 execution for this query. Two regex in each workload also have a string literal with low selectivity, which means the 3-way split is ineffective in reducing the number of log lines that need to be examined. However, each of these regexes happens to have a highly selective literal component in the middle and thus evaluation is faster using the multi-way split mode. For the System-Y workload, the 3-way split approach is only marginally faster than multi-way split, as a majority of the queries have a highly selective `prefix` and/or `suffix`. There exists one regex whose `prefix` and `suffix` have low selectivity, but it also has no string literal anywhere else, which renders the multi-way split mode ineffective. For all three workloads, BLARE is either the best strategy or close to the best, thanks to its learning component.

The results of the per-regex running time distribution are presented in Figures 3b, 4b, and 5b. For the DB-X workload, RE2 takes around 7 seconds for most queries, and BLARE can reduce the execution time by $2 - 3\times$. The regexes benefiting the most have long literal prefixes and/or literal suffixes and can take advantage of the structural decomposition in BLARE. Every query in the workload benefits from BLARE, and the RE2 direct execution is never chosen as the strategy by the learner. This is not surprising since performing string matching leads to a drastic reduction in the number of log lines that need to be examined. At this point, the reader may wonder about what mode is chosen by BLARE for the Split-Matcher for each query and whether there is any deeper insight into when it chooses one mode over the other. We defer the discussion of the choice between the 3-way-split and the multi-way-split mode for different regex queries to Section 5.4.

Figure 4b shows the distribution of execution time for different strategies. Once again, the learner does not choose direct mode as the choice for any of the regexes. Every query in the workload benefits from BLARE. In this workload, we found a regex query that takes ∼100 seconds using BLARE and ∼600 seconds using RE2. This regex has very short string literal components and a large number of matches in the dataset, and therefore takes more time to process compared to other

regexes. However, even with short string literals, BLARE effectively reduces the number of log lines that need to be examined further, and thus increases the performance by 6×.

Finally, Figure 5b shows the distribution for the US-Accident workload. Every query in the workload benefits from BLARE.

PCRE2 Performance. We found that in some cases PCRE2 is significantly slower compared to RE2. Specifically, it took upwards of over 90,000 seconds to execute the full DB-X workload (which is over 80× slower compared to the time it took for vanilla RE2 to run this workload). To keep the experiments manageable, and to study the impact of BLARE on PCRE2, we generated a smaller version of this production workload. We derived a new workload, called **DB-X-Small**, that contains 13 regexes from the DB-X workload. The selection of the queries was done manually to include regexes with varying lengths, selectivities, and literal components. In particular, the min and max regex length was 41 and 274. Each regex contained at least one recursive component (as defined in [21]) and the maximum number of recursive components in a regex was 5. The other two workloads, namely System-Y and the US-Accident, ran in a reasonable time with PCRE2 and we use them as is.

Tables 3, 4, and 5 show that for all the workloads, BLARE on PCRE2 improves the performance compared to PCRE2 alone. BLARE reduces the running time of DB-X-Small by 3.2×. Table 3 shows the distribution of the queries for the DB-X-Small workload. Similar to RE2, BLARE never chooses

Table 3. Performance comparison of BLARE vs. PCRE2 on the DB-X-Small Workload.

Method	Per Regex Running Time (s)					Total Time (s)
	min	Q1	median	Q3	max	
PCRE2	6.6	7.0	7.2	7.3	10.9	96.8
BLARE	2.1	2.1	2.2	2.3	3.8	30.3
BLARE Improvement: 3.2×						

Table 4. Performance comparison of BLARE vs. PCRE2 on the System-Y Workload.

Method	Per Regex Running Time (s)					Total Time (s)
	min	Q1	median	Q3	max	
PCRE2	58.8	62.5	65.6	73.6	289.2	1582.2
BLARE	16.0	18.6	20.8	36.0	59.4	503.9
BLARE Improvement: 3.1×						

Table 5. Performance comparison of BLARE vs. PCRE2 on the US-Accident Workload.

Method	Per Regex Running Time (s)					Total Time (s)
	min	Q1	median	Q3	max	
PCRE2	0.3	0.8	2.0	54.2	207.4	211.8
BLARE	0.1	0.2	0.3	0.4	0.5	1.3
BLARE Improvement: 168.3×						

Table 6. Performance comparison of BLARE vs. Boost Regex on the DB-X Workload.

Method	Per Regex Running Time (s)					Total Time (s)
	min	Q1	median	Q3	max	
Boost	13.1	14.2	14.8	15.2	58.3	2164.0
BLARE	1.4	1.7	1.9	1.9	10.1	273.9
BLARE Improvement: 7.9×						

Table 7. Performance comparison of BLARE vs. Boost Regex on the System-Y Workload.

Method	Per Regex Running Time (s)					Total Time (s)
	min	Q1	median	Q3	max	
Boost	109.0	111.0	118.5	123.3	191.5	2290.7
BLARE	15.6	17.9	19.6	33.6	51.3	465.7
BLARE Improvement: 4.9×						

Table 8. Performance comparison of BLARE vs. Boost Regex on the US-Accident Workload.

Method	Per Regex Running Time (s)					Total Time (s)
	min	Q1	median	Q3	max	
Boost	0.3	0.4	0.8	1.4	2.0	4.0
BLARE	0.1	0.2	0.3	0.4	0.5	1.2
BLARE Improvement: 3.4×						

direct execution on PCRE2 as a winning strategy and every query in the workload experiences a performance improvement when using BLARE. For almost all the queries, a 3-way split approach is the best strategy and BLARE is able to predict it correctly. The performance for System-Y is improved by 3.1×, and every query is faster compared to direct PCRE2.

Table 5 shows the distribution for the US-Accident workload. Every query in the workload benefits from BLARE. One of the regex has no `prefix` or `suffix`, and it takes 200 seconds on PCRE2 but runs in 0.3 seconds using BLARE. This regex begins with `(.+)` and we observed that PCRE2 performance is slow for other queries starting with this expression in other workloads as well. Similar to the case for RE2, regexes in US-Accident workload cannot take full advantage of BLARE as they have short literal components. However, they still benefit from structural decomposition in BLARE, and our framework reduces the overall running time by ∼ 168×.

Boost Regex Performance. Direct execution of Boost Regex has similar performance on the DB-X workload compared to RE2 but is much slower than RE2 for the System-Y and US-Accident workloads, as shown in Tables 6, 7, and 8. BLARE reduces the running time of the DB-X workload by 7.9×, making it faster than BLARE-RE2 on the same workload. It reduces the running time of the System-Y and US-Accident workloads by 4.9× and 3.4× respectively. Once again, BLARE never chooses direct execution on Boost Regex as a strategy and every query in the workload experiences a performance improvement when using BLARE.

ICU Regex Performance. ICU Regex has a large running time for some regexes in the DB-X workloads. Therefore, we run DB-X-Small workload on ICU Regex and BLARE. From Table 9, we can see that direct execution of ICU Regex and BLARE-ICU Regex have longer running time on DB-X-Small workload, compared to PCRE2. BLARE manages to shorten the running time by 1.6×. Every query in this workload gets a performance improvement with BLARE. Table 10 shows the distribution for the US-Accident workload. Every query in the workload benefits from BLARE. One regex takes 167 seconds to run with ICU Regex, but runs in 0.4 seconds using BLARE. The US-Accident workload benefits from the structural decomposition component in BLARE and our framework reduces the overall running time by ∼ 62×. We note that ICU Regex accepts ICU Unicode strings which are UTF-16 encoded and have a higher space requirement. For this reason, the System-Y dataset cannot fit in memory. Interestingly, for the DB-X-Small workload, we found that the direct execution on ICU Regex for 4 regexes out of the 13 was the faster strategy (by up to 46% for one regex). In each case, BLARE identified the best strategy correctly.

Table 9. Performance comparison of BLARE vs. ICU Regex on the DB-X-Small Workload.

Method	Per Regex Running Time (s)					Total Time (s)
	min	Q1	median	Q3	max	
ICU	4.0	6.3	8.2	12.7	20.1	127.7
BLARE	4.1	4.9	6.4	7.8	8.1	81.5
BLARE Improvement: 1.6×						

Table 10. Performance comparison of BLARE vs. ICU Regex on the US-Accident Workload.

Method	Per Regex Running Time (s)					Total Time (s)
	min	Q1	median	Q3	max	
ICU	0.3	0.7	1.7	43.9	167.4	171.2
BLARE	0.2	0.2	0.3	0.6	1.5	2.3
BLARE Improvement: 61.7×						

Table 11. Mean percentage of time spent on the learner of BLARE for RE2, PCRE2, Boost Regex, and ICU Regex. × denotes that the underlying engine could not finish running in reasonable time or the test machine could not fit the dataset in memory.

	BLARE-RE2	BLARE-PCRE2	BLARE-Boost Regex	BLARE-ICU Regex
DB-X-Small	4.2%	2.8%	5.6%	8.1%
DB-X	5.1%	×	6.1%	×
System-Y	6.7%	9.1%	10.7%	×
US-Accident	16.5%	23.8%	27.4%	28.1%

5.3 Multi-Armed Bandit Overhead

This section is dedicated to answering **Q.2** by examining the overhead of using the multi-armed bandit method. Table 11 shows the average overhead that the learning component introduces in the total running time across all queries in each workload over the strategy chosen by BLARE.

For BLARE-RE2, the average overhead is below 7% for DB-X, DB-X-Small, and System-Y, and 2.8% (9.1%) for BLARE-PCRE2 on the DB-X-Small (System-Y) workload. BLARE-Boost Regex has a higher overhead percentage of 10.7% on the System-Y workload, and its average overhead on the DB-X and the DB-X-Small workload is below 7%. Considering the average overhead of 8.1% for BLARE-ICU Regex on the DB-X-Small workload, the average overhead of BLARE learner on the two production datasets is less than 11%. We observed that for most queries, the overhead is close to the mean.

For the US-Accident workload, the overhead is higher. Recall that by default, we use 0.001% of the dataset as a random sample for learning. However, for the US-Accident dataset, this fraction yields a sample size of only 28 lines which is too small for learning. Since we ensure that the sample should have at least 200 lines, we generate a sample of size 200 (a 10× increase in the sample size compared to the sample size used in other datasets). This leads to a proportionally higher cost of learning. Nevertheless, the improvement from BLARE is more than enough to compensate for the relatively higher learning cost compared to running it directly with the base regex libraries.

5.4 Comparing 3-way and Multi-way Split

In this section, we answer **Q.3** by comparing the performance of the 3-way split strategy with the multi-way split strategy. Recall that for multi-way split, we decompose the regex into as many components as possible.

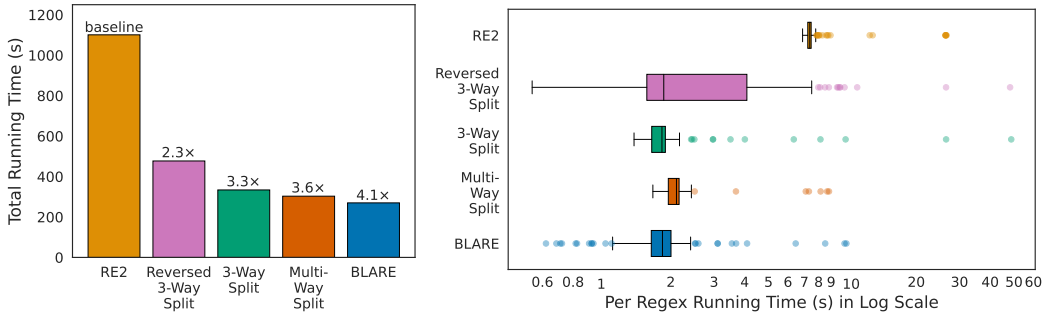
RE2 Performance. We first turn our attention to RE2 on the DB-X workload. Figure 3a shows that BLARE obtains the best performance, followed by the multi-way split strategy, and then the 3-way split strategy. We dived deeper to understand which queries ran using multi-way split in BLARE and which queries chose 3-way split. We found that out of the 132 queries, only 10 queries got a performance improvement using the multi-way split approach, and BLARE's learner correctly picks the multi-way split mode for all 10 queries. Three of the regexes have non-existent `prefix`. Two of these regexes have a dot as a `suffix` and one regex has no `suffix`. Three regexes have long `prefix` and a dot (which has very low selectivity) as a `suffix` and the remaining five have either a `prefix` or a `suffix`, but not both. Each of these cases likely leads to a low overall selectivity and performing multi-way split aids in increasing the selectivity. The query with the largest improvement (over 15×) was: `(.+) is not supported on (.+)\.` . Observe that this query has an empty `prefix` and the `suffix` is a dot, a character with low selectivity. Performing a multi-way split allows for generating at least one more literal component that increases the overall selectivity. This is in line with our cost model presented in Section 3.2.1, which tells us that the overall selectivity of the string literals must be high in order to get a low Split-Match algorithm cost. For the remaining 126 queries in the workload, a 3-way split was faster by 15% (on average), as can be seen from Figure 3b. The cluster of queries for multi-way split has a larger running time (most queries have a running time between 7 to 10 seconds) than the cluster of queries for BLARE and the 3-way split (most queries have a run time between 2 to 7 seconds).

For the System-Y workload, Figure 4a shows that multi-way split is marginally slower than 3-way split. A closer examination of the running time for each regex shows that nearly all of the queries experience a slowdown. This is expected since most of the regex in the System-Y workload have a `prefix` and/or `suffix` already. Further decomposition of the regex to generate additional string literal components leads to a higher string-matching cost. Figure 5a shows that in the US-Accident

workload, the multi-way split strategy obtains the best performance, followed by BLARE, and then the 3-way split strategy. In the US-Accident workload, one of the regexes has a non-existent `prefix` and `suffix`, and two of three regexes each contains a `prefix` with low selectivity. However, all three of these regexes have literal components in the middle that can further reduce the number of invocations of the underlying engine, and thus evaluation is faster when using the multi-way-split.

PCRE2 Performance. For the DB-X-Small workload, we observed that the multi-way split method had an average slowdown of about 8% compared to the 3-way split method and the largest slowdown across all queries was 11%. Multi-way split achieves comparable performance on only one query in the workload against 3-way split and is slower than the 3-way split for all queries in System-Y for PCRE2. The behavior of the US-Accident workload using PCRE2 was similar to RE2.

Boost Regex Performance. We observed that 5 out of the 132 queries in the DB-X workload see better performance when running in multi-way split mode than the 3-way split mode. Among them, two queries have close running times (difference less than 3%) using the two methods. Two other queries, one with no `prefix` or `suffix` components and one with no `prefix` and dot as the `suffix`, experience a performance improvement of 10× and 12× respectively using the multi-way split. For the remainder of the 128 queries, a 3-way split approach is faster than the multi-way split approach by 3% to 50%, with an average improvement of 14%. The multi-way split approach is 12% slower on average than the 3-way split in System-Y. The behavior of the US-Accident workload using `Boost Regex` was similar to RE2.



(a) Overall running time

(b) Distribution of per regex running time

Fig. 6. BLARE vs. multi-way-split vs. 3-way-split vs. Reversed 3-way-split vs. RE2 on the DB-X Workload.

ICU Regex Performance. There are 3 queries out of the total 13 queries in the DB-X-Small workload that benefit from the multi-way split. Two of them have a negligible performance difference between the two methods (1.6% and 3.6%). One of the queries has `prefix` as `Database` and `.` as a `suffix`. Even though the selectivity of `.` is low, a more selective middle component improves its performance by 7% using the multi-way split. On average, a 3-way split is faster than a multi-way split by 6%. The behavior of the US-Accident workload using `ICU Regex` was similar to RE2.

Overall, we observe that all findings are consistent with the cost model introduced in Section 3.2.1 which recommends using a 3-way split if `prefix` and the `suffix` is non-empty.

5.5 Discussion

In this section, we answer both questions raised in **Q.4**. Recall in this question we want to understand if BLARE offers any performance improvement for other regex queries (such as count and longest

Table 12. Performance comparison of Longest_Match BLARE vs. RE2 on the DB-X Workload.

Method	Per Regex Running Time (s)					Total Time (s)
	min	Q1	median	Q3	max	
RE2	7.0	7.4	7.6	7.7	26.8	1128.5
BLARE	1.6	1.9	2.1	2.1	11.1	306.1
BLARE Improvement: 3.7×						

Table 13. Performance comparison of Count_All BLARE vs. RE2 on the DB-X Workload.

Method	Per Regex Running Time (s)					Total Time (s)
	min	Q1	median	Q3	max	
RE2	7.2	7.6	7.7	7.8	27.0	1148.7
BLARE	1.5	1.8	1.9	2.0	10.6	299.8
BLARE Improvement: 3.8×						

match) and whether BLARE is extensible. We begin by examining other regex queries of interest, followed by examining the extensibility aspect.

5.5.1 Other Regex Queries. So far, all of our experiments are performed in the setting where the regex queries extract the matched input (via capturing groups) from the dataset. In this section, we report the experimental results for BLARE on other regex queries of interest. The first query we consider is the *longest_match*. A longest match query finds all possible matches of a regex and then returns the longest one of them, a useful operation used in regex disambiguation [29, 88]. Table 12 shows the results for the longest match query on RE2. We observe that BLARE is effective in this setting as well. BLARE delivers a 3.7× improvement over the direct execution of the workload on RE2 for DB-X. The second query of interest, *count_all*, counts the total number of matches in the input, an operation that is frequently used in practice [70]. Table 13 shows the results for this query, and shows that for BLARE using RE2 on the DB-X workload, we get a 3.8× improvement in running time over direct execution.

5.5.2 Extensibility. Our final evaluation in this section demonstrates the benefit of our framework with respect to extensibility (required by the desiderata **(R.2)**). Specifically, we modify the learning component in BLARE to include another strategy to understand how easy it is to make the change and how BLARE responds to the new strategy. By default, the `prefix` and `suffix` in the 3-way split are searched in the log line in that specific order. However, one can also do the string match in the opposite order without affecting the correctness. If the `suffix` happens to be more selective than the `prefix`, we expect the second strategy to win. Therefore, we extend the multi-armed bandit component in the framework to include another arm. After this change, the split-matcher has to choose from four modes instead of three. Figure 6a shows the result of this modified BLARE approach on the DB-X workload (the distribution of the runtimes is shown in Figure 6b). We observe that this new strategy is the best among all the choices for 52 out of the 132 queries in the workload and the largest performance gain for a regex was 3.5× over the previous best when BLARE had only three modes to choose from. Note that using the reversed 3-way split always was

the next worst strategy after direct execution on RE2. Fortunately for BLARE, we are able to get the best of all the arms owing to its learning component.

6 RELATED WORK

Regular expression query evaluation has been intensively studied, both theoretically and practically, over five decades across many communities, including Systems, PL, Security, Data Mining, and Networking. It has widespread applications ranging from pattern matching [44], intrusion detection [50, 2], data validation and repair [52], data discovery [75], code search [23, 77], etc. (see [94] for a detailed survey). Architectures for regex evaluation at line rate are based on either DFAs or NFAs. For both of these solutions, there is a wide body of work that has looked at state-minimization algorithms [50, 9], alphabet reduction [14, 9, 48, 8], and the so-called increased stride where multiple characters are consumed per state transition [97, 39, 69, 14]. A different line of work has explored improving regex performance by using modern hardware such as programmable switches [91], FPGA [101, 5, 66], GPUs [43, 89], and network processors [54, 41], and optimized pipelined infrastructure [31, 59, 10, 98].

Since most practical systems are DFA-based, a large number of studies have been proposed in recent years to optimize DFA-based pattern matching. The core goal is to minimize the memory footprint while guaranteeing high performance. Some of these works [86, 7, 90] leverage bitmap indexes to eliminate certain transitions and minimize memory usage. Other works have focused on rewriting the regex to make it similar by replacing chain of transitions into a single transition [50, 3], or use default transitions to reduce redundancy [4, 53]. There is also considerable work on efficient pattern matching techniques such as partitioning the regex rules [55, 87], and variants of automata tailored to specific scenarios [49, 95]. Recent work [73] has also made progress in developing domain-specific programmable engines that define a novel instruction set and compiler framework to translate regex into a program that is amenable to exploiting parallel hardware.

Within the database community, regex evaluation on FPGA and modern architectures [67, 80, 81], building indexes for regex evaluation [18, 30], and its deep connections to XML data management solutions [61, 56, 15] have been intensively studied. The related problem of regex evaluation over labeled graphs has a rich history within the database community [26, 16, 28, 27, 93]. We also refer the reader to recent work by Martens et al. [62, 63, 60] that describes the theoretical aspects and real-world behavior of evaluating regular path queries in graphs. Regex and its augmented versions have also found applications in searching patterns in visualizations [79]. Prior work [36, 82, 83] has also studied the related problem of declarative querying of biological datasets. These solutions develop novel algebraic frameworks that are geared towards pattern extraction and exploit the special properties of biological datasets (such as genomic data, protein data, etc.). Similar to our approach, prior work [47] has also made the case for using string-matching based ideas for faster XML parsing. Prior work has also proposed using DFAs [33] and NFAs [24, 25] in conjunction with multi-query optimization techniques for XPath matching over XML documents and streams. Several works have also looked at the related but orthogonal problem of using indexes for speeding up regular expression matching. To this end, [85] proposed identifying strings that can be used as keys to index records. Li et al. [51] proposed a novel indexing structure to answer long regex path queries over XML data. Creating indexes over multigrams has also shown performance improvements [20]. Indexing-based approaches are also heavily used when querying over biological datasets [40, 96, 13, 100, 64]. The indexes support not only exact matches but also approximate matches and similarity based on distance metrics. However, most of the works related to indexing assume knowledge of the workload (both [85] and [20] explicitly note this). Most of these approaches advocate for a preprocessing phase that allows for workload-aware index creation whereas in this paper, we do

not make any specific assumptions about the workload and the data. We leave the study of building indexes (either in the preprocessing phase or on-the-fly) for BLARE as a problem for future work.

Besides RE2, PCRE, and its successor PCRE2, Boost Regex, and ICU Regex, several real-world systems boast of many innovative performance improvements. Snort [22] and Suricata [71], two state-of-the-art intrusion detection systems, use pattern matching as a lightweight prefiltering mechanism. SplitScreen [17], a malware detection system, makes extensive use of Bloom filters to perform a signature-based detection of a regex match. Intel's Hyperscan [92] extends the prefiltering technique from RE2 (described in Section 2) and applies the idea to the entire query. Specifically, all string literal components in the query are matched outside the finite automata and the regex part that occurs just after the string is executed only if the preceding string component matches. This process is done in a left-to-right linear fashion. Hyperscan focuses extensively on using graph-based techniques to identify the dominating patterns in the query and uses SIMD-accelerators for fast pattern matching. Our setting departs from the use cases of Hyperscan in several aspects. First, regexes used in log analysis are often simpler and do not contain nested sub-expressions. This eliminates the need for doing Hyperscan's NFA analysis of the query to find repeated patterns of string components. Second, Hyperscan executes the automata next to a string literal immediately if the string literal matches. In contrast, our prefiltering first scans the input for all string literals and only then evaluates the regex. Thirdly, it is important to make sure that no query in the workload suffers from performance degradation. Therefore, we need to explore some form of learning-based strategies as well to ensure we don't introduce regressions. Finally, our techniques are more broadly applicable and do not make any assumptions on the available hardware. Prefiltering-based ideas have also been applied to other parts of the regex. One such example is using range hashes [6], where the range of characters (e.g., presence of `[a-z]`, `\d`, etc. is checked by using FPGAs).

7 CONCLUSIONS AND FUTURE WORK

In this paper, we have presented BLARE, a novel, lightweight, extensible framework that accelerates regex query evaluation for log analytics. BLARE improves performance by considering a regex as a query composed of fixed strings and core regular expression components. It then uses an execution approach that is inspired by query processing/optimization strategies in database systems. Specifically, we proposed a multi-armed bandit method to selectively move some computation out of the automata machinery to a faster string-matching component. BLARE runs on top of any regex engine since it uses the engine in a blackbox fashion. We also presented experimental results that demonstrate significant performance improvements across all workloads that we tested on, and for four popular regex libraries, namely RE2, PCRE2, Boost Regex, and ICU Regex.

We view this paper as a starting point in a line of work that brings the benefits of decades of database research to log analytics. In particular, we envision building a regex evaluation engine that features more operators such as specialized string matching algorithms, a non-trivial query plan space, specialized indexes, and using a catalog that collects statistics about the queries and the input data. Multi-query optimization is another area that can bring significant improvements, as sometimes log queries are presented in a batch that can be processed concurrently. It would also be interesting to examine how our work can help in speeding up JSON schema containment [34] where regexes with many literals are a bottleneck.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation (NSF) under grant OAC-1835446.

REFERENCES

[1] Alfred V. Aho and Margaret J. Corasick. 1975. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18, 6, (June 1975), 333–340. DOI: 10.1145/360825.360855.

[2] Monther Aldwairi and Duaa Alansari. 2011. Exscind: fast pattern matching for intrusion detection using exclusion and inclusion filters. In *2011 7th International Conference on Next Generation Web Services Practices*. IEEE. (Oct. 2011), 24–30. DOI: 10.1109/nwesp.2011.6088148.

[3] Rafael Antonello, Stenio Fernandes, Djamel Sadok, Judith Kelner, and Géza Szabo. 2012. Deterministic finite automaton for scalable traffic identification: the power of compressing by range. In *2012 IEEE Network Operations and Management Symposium*. IEEE. (Apr. 2012), 155–162. DOI: 10.1109/noms.2012.6211894.

[4] Rafael Antonello, Stenio Fernandes, Djamel Sadok, Judith Kelner, and Géza Szabó. 2015. Design and optimizations for efficient regular expression matching in dpi systems. *Computer Communications*, 61, (May 2015), 103–120. DOI: 10.1016/j.comcom.2014.12.011.

[5] Zachary K Baker and Viktor K Prasanna. 2004. Time and area efficient pattern matching on fpgas. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*. ACM, (Feb. 2004), 223–232. DOI: 10.1145/968280.968312.

[6] Masanori Bando, N Sertac Artan, Rihua Wei, Xiangyi Guo, and H Jonathan Chao. 2010. Range hash for regular expression pre-filtering. In *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. (Oct. 2010), 1–12. DOI: 10.1145/1872007.1872032.

[7] Michela Becchi and Srihari Cadambi. 2007. Memory-efficient regular expression search using state merging. In *IEEE INFOCOM 2007-26th IEEE International Conference on Computer Communications*. IEEE, 1064–1072. DOI: 10.1109/infcom.2007.128.

[8] 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 (TACO)*, 10, 1, (Apr. 2013), 1–26. DOI: 10.1145/2445572.2445576.

[9] Michela Becchi and Patrick Crowley. 2007. An improved algorithm to accelerate regular expression evaluation. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, 145–154. DOI: 10.1145/1323548.1323573.

[10] Michela Becchi, Charlie Wiseman, and Patrick Crowley. 2009. Evaluating regular expression matching engines on network and general purpose processors. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. (Oct. 2009), 30–39. DOI: 10.1145/1882486.1882495.

[11] 2022. 8.5.7 'posix-basic' regular expression syntax. Retrieved October 7, 2022 from https://www.gnu.org/software/findutils/manual/html_node/find_html/posix_002dbasic-regular-expression-syntax.html.

[12] Robert S Boyer and J Strother Moore. 1977. A fast string searching algorithm. *Communications of the ACM*, 20, 10, (Oct. 1977), 762–772. DOI: 10.1145/359842.359859.

[13] Tolga Bozkaya and Meral Ozsoyoglu. 1997. Distance-based indexing for high-dimensional metric spaces. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, 357–368. DOI: 10.1145/253260.253345.

[14] Benjamin C Brodie, David E Taylor, and Ron K Cytron. 2006. A scalable architecture for high-throughput regular-expression pattern matching. *ACM SIGARCH computer architecture news*, 34, 2, 191–202. DOI: 10.1109/isca.2006.7.

[15] Anne Brüggemann-Klein and Derick Wood. 1998. One-unambiguous regular languages. *Information and computation*, 140, 2, 229–253. DOI: 10.1006/inco.1997.2688.

[16] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y Vardi. 1999. Rewriting of regular expressions and regular path queries. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 194–204. DOI: 10.1145/303976.303996.

[17] 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.

[18] Chee-Yong Chan, Minos Garofalakis, and Rajeev Rastogi. 2003. Re-tree: an efficient index structure for regular expressions. *The VLDB Journal*, 12, 2, 102–119. DOI: 10.1007/s00778-003-0094-0.

[19] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. 2014. Trill: a high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment*, 8, 4, 401–412. DOI: 10.14778/2735496.2735503.

[20] Junghoo Cho and Sridhar Rajagopalan. 2002. A fast regular expression indexing engine. In *Proceedings 18th International Conference on Data Engineering*. IEEE, 419–430. DOI: 10.1109/ICDE.2002.994755.

[21] Byron Choi. 2002. What are real dtds like? In *International Workshop on the Web and Databases*.

[22] [SW] Cisco, Snort - Network Intrusion Detection and Prevention System. URL: https://www.snort.org/snort3, VCS: https://github.com/snort3/snort3.

[23] Russ Cox. 2012. Regular expression matching with a trigram index or how google code search worked. (2012).

[24] Yanlei Diao, Mehmet Altinel, Michael J Franklin, Hao Zhang, and Peter Fischer. 2003. Path sharing and predicate evaluation for high-performance xml filtering. *ACM Transactions on Database Systems (TODS)*, 28, 4, 467–516. DOI: 10.1145/958942.958947.

[25] Yanlei Diao, Peter Fischer, Michael J Franklin, and Raymond To. 2002. Yfilter: efficient and scalable filtering of xml documents. In *Proceedings 18th International Conference on Data Engineering*. IEEE, 341–342. DOI: 10.1109/icde.2002.994748.

[26] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Yinghui Wu. 2012. Adding regular expressions to graph reachability and pattern queries. *Frontiers of Computer Science*, 6, 3, (June 2012), 313–338. DOI: 10.1007/s11704-012-1312-y.

[27] Daniela Florescu, Alon Levy, and Dan Suciu. 1998. Query containment for conjunctive queries with regular expressions. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, 139–148. DOI: 10.1145/275487.275503.

[28] Dominik D Freydenberger, Benny Kimelfeld, and Liat Peterfreund. 2018. Joining extractions of regular expressions. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 137–149. DOI: 10.1145/3196959.3196967.

[29] Alain Frisch and Luca Cardelli. 2004. Greedy regular expression matching. In *International Colloquium on Automata, Languages, and Programming*. Springer, 618–629. DOI: 10.1007/978-3-540-27836-8_53.

[30] Daniel Gibney and Sharma V Thankachan. 2021. Text indexing for regular expression matching. *Algorithms*, 14, 5, (Apr. 2021), 133. DOI: 10.3390/a14050133.

[31] Vaibhav Gogte, Aasheesh Kolli, Michael J Cafarella, Loris D'Antoni, and Thomas F Wenisch. 2016. Hare: hardware accelerator for regular expressions. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. (Oct. 2016), 1–12. DOI: 10.1109/micro.2016.7783747.

[32] [SW] Google, Google-RE2 version release 2022-06-01. URL: https://github.com/google/re2, VCS: https://github.com/google/re2.

[33] Todd J Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. 2003. Processing xml streams with deterministic automata. In *International Conference on Database Theory*. Springer, 173–189. DOI: 10.1145/1042046.1042051.

[34] Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. 2021. Finding data compatibility bugs with json subschema checking. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 620–632. DOI: 10.1145/3460319.3464796.

[35] [SW] Md. Jahidul Hamid, JPCRE2 version 10.32.01. URL: https://docs.neuzunix.com/jpcre2/, VCS: https://github.com/jpcre2/jpcre2.

[36] Laurie Hammel and Jignesh M Patel. 2002. Searching on the secondary structure of protein sequences. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 634–645. DOI: 10.1016/b978-1558608690-6/50062-7.

[37] Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. 2016. Logmine: fast pattern recognition for log analytics. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, 1573–1582. DOI: 10.1145/2983323.2983358.

[38] Philip Hazel. 2019. Technical notes about pcre2. (July 2019). Retrieved October 11, 2022 from https://github.com/PCRE/pcre2/blob/6c941e3f778970b3012093da7bf888c875dd1739/HACKING%5C#L454.

[39] 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, 83–93. DOI: 10.1109/ancs.2013.6665178.

[40] Ela Hunt, Malcolm P Atkinson, and Robert W Irving. 2002. Database indexing for large dna and protein sequence collections. *The VLDB Journal*, 11, 3, 256–271. DOI: 10.1007/s007780200064.

[41] Joel Hypolite, John Sonchack, Shlomo Hershkop, Nathan Dautenhahn, André DeHon, and Jonathan M Smith. 2020. Deepmatch: practical deep packet inspection in the data plane using network processors. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, 336–350. DOI: 10.1145/3386367.3431290.

[42] 2018. Ieee standard for information technology–portable operating system interface (posix(tm)) base specifications, issue 7. *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, 1–3951. DOI: 10.1109/IEEESTD.2018.8277153.

[43] Muhammad Asim Jamshed, Jihyung Lee, Sangwoo Moon, Insu Yun, Deokjin Kim, Sungryoul Lee, Yung Yi, and KyoungSoo Park. 2012. Kargus: a highly-scalable software-based intrusion detection system. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, (Oct. 2012), 317–328. DOI: 10.1145/2382196.2382232.

[44] Ramakrishnan Kandhan, Nikhil Teletia, and Jignesh M Patel. 2010. Sigmatch: fast and scalable multi-pattern matching. *Proceedings of the VLDB Endowment*, 3, 1-2, 1173–1184. DOI: 10.14778/1920841.1920987.

[45] Seongyoung Kang, Jiyoung An, Jinpyo Kim, and Sang-Woo Jun. 2021. Mithrilog: near-storage accelerator for high-performance log analytics. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 434–448. DOI: 10.1145/3466752.3480108.

[46] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. 1977. Fast pattern matching in strings. *SIAM Journal on Computing*, 6, 2, 323–350. eprint: https://doi.org/10.1137/0206024. DOI: 10.1137/0206024.

[47] Christoph Koch, Stefanie Scherzinger, and Michael Schmidt. 2008. Xml prefiltering as a string matching problem. In *2008 IEEE 24th International Conference on Data Engineering*. IEEE. (Apr. 2008), 626–635. DOI: 10.1109/icde.2008.4497471.

[48] Shijin Kong, Randy Smith, and Cristian Estan. 2008. Efficient signature matching with multiple alphabet compression tables. In *Proceedings of the 4th international conference on Security and privacy in communication netowrks*, 1–10. DOI: 10.1145/1460877.1460879.

[49] Sailesh Kumar, Balakrishnan Chandrasekaran, Jonathan Turner, and George Varghese. 2007. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, 155–164. DOI: 10.1145/1323548.1323574.

[50] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. 2006. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. *ACM SIGCOMM computer communication review*, 36, 4, 339–350. DOI: 10.1145/1159913.1159952.

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

[52] Zeyu Li, Hongzhi Wang, Wei Shao, Jianzhong Li, and Hong Gao. 2016. Repairing data through regular expressions. *Proceedings of the VLDB Endowment*, 9, 5, 432–443. DOI: 10.14778/2876473.2876478.

[53] Alex X Liu and Eric Torng. 2014. An overlay automata approach to regular expression matching. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*. IEEE, 952–960. DOI: 10.1109/INFOCOM.2014.6848024.

[54] 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 (TECS)*, 3, 3, 614–633. DOI: 10.1145/1015047.1015055.

[55] Tingwen Liu, Alex X Liu, Jinqiao Shi, Yong Sun, and Li Guo. 2014. Towards fast and optimal grouping of regular expressions via dfa size estimation. *IEEE Journal on Selected Areas in Communications*, 32, 10, 1797–1809. DOI: 10.1109/JSAC.2014.2358839.

[56] Katja Losemann. 2012. Foundations of regular expressions in xml schema languages and sparql. In *Proceedings of the on SIGMOD/PODS 2012 PhD Symposium*, 39–44. DOI: 10.1145/2213598.2213609.

[57] Jiaheng Lu, Yuxing Chen, Herodotos Herodotou, and Shivnath Babu. 2019. Speedup your analytics: automatic parameter tuning for databases and big data systems. *Proceedings of the VLDB Endowment*. DOI: 10.14778/3352063.3352112.

[58] 2020. Luceneplusplus build instructions. (Aug. 2020). Retrieved January 08, 2023 from https://github.com/luceneplusplus/LucenePlusPlus/blob/9eb5e63e336546bc927e622fdda49919c4143e8e/doc/BUILDING.md.

[59] Jan Van Lunteren, Christoph Hagleitner, Timothy Heil, Giora Biran, Uzi Shvadron, and Kubilay Atasu. 2012. Designing a programmable wire-speed regular-expression matching accelerator. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO-45). IEEE Computer Society, Vancouver, B.C., CANADA, 461–472. ISBN: 9780769549248. DOI: 10.1109/MICRO.2012.49.

[60] Wim Martens. 2022. Towards theory for real-world data. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 261–276. DOI: 10.1145/3517804.3526066.

[61] Wim Martens, Frank Neven, and Thomas Schwentick. 2007. Simple off the shelf abstractions for xml schema. *ACM SIGMOD Record*, 36, 3, 15–22. DOI: 10.1145/1324185.1324188.

[62] Wim Martens and Tina Trautner. 2019. Bridging theory and practice with query log analysis. *ACM SIGMOD Record*, 48, 1, 6–13. DOI: 10.1145/3371316.3371319.

[63] Wim Martens and Tina Trautner. 2018. Evaluation and enumeration problems for regular path queries. In *21st International Conference on Database Theory (ICDT 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. DOI: 10.4230/LIPIcs.ICDT.2018.19.

[64] 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 Conference*. Morgan Kaufmann, 910–921. ISBN: 978-0-12-722442-8. DOI: 10.1016/B978-012722442-8/50085-9.

[65] Microsoft. 2022. Reference materials for kusto query language - re2 syntax. (Nov. 2022). Retrieved October 11, 2022 from https://learn.microsoft.com/en-us/azure/data-explorer/kusto/query/re2.

[66] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. 2007. Compiling pcre to fpga for accelerating snort ids. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*. ACM, (Dec. 2007), 127–136. DOI: 10.1145/1323548.1323571.

[67] Kento Miura, Toshiyuki Amagasa, Hiroyuki Kitagawa, R Bordawekar, and T Lahiri. 2019. Accelerating regular path queries using fpga. In *ADMS@ VLDB*, 47–54.

[68] Sobhan Moosavi, Mohammad Hossein Samavatian, Srinivasan Parthasarathy, Radu Teodorescu, and Rajiv Ramnath. 2019. Accident risk prediction based on heterogeneous sparse data: new dataset and insights. In *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (SIGSPATIAL '19). Association for Computing Machinery, Chicago, IL, USA, 33–42. ISBN: 9781450369091. DOI: 10.1145/3347146.3359078.

[69] 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 International Conference on Computational Science and Engineering*. IEEE, 547–553. DOI: 10.1109/CSE.2014.125.

[70] Pierre Nicodeme. 2003. Regexpcount, a symbolic package for counting problems on regular expressions and words. *Fundamenta Informaticae*, 56, 1-2, 71–88.

[71] [SW] Open Information Security Foundation, Suricata. URL: https://suricata.io/, VCS: https://github.com/OISF/suricata.

[72] Oracle. [n. d.] Mysql 8.0 reference manual: 12.8.2 regular expressions. Retrieved January 08, 2023 from https://dev.mysql.com/doc/refman/8.0/en/regexp.html.

[73] Daniele Parravicini, Davide Conficconi, Emanuele Del Sozzo, Christian Pilato, and Marco D Santambrogio. 2021. Cicero: a domain-specific architecture for efficient regular expression matching. *ACM Transactions on Embedded Computing Systems (TECS)*, 20, 5s, 1–24. DOI: 10.1145/3476982.

[74] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: meta's unified execution engine. *Proceedings of the VLDB Endowment*, 15, 12, 3372–3384. DOI: 10.14778/3554821.3554829.

[75] El Kindi Rezig, Anshul Bhandari, Anna Fariha, Benjamin Price, Allan Vanterpool, Vijay Gadepally, and Michael Stonebraker. 2021. Dice: data discovery by example. *Proceedings of the VLDB Endowment*, 14, 12, 2819–2822. DOI: 10.14778/3476311.3476353.

[76] Daniel J Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, Zheng Wen, et al. 2018. A tutorial on thompson sampling. *Foundations and Trends® in Machine Learning*, 11, 1, 1–96. DOI: 10.1561/2200000070.

[77] Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. 2015. How developers search for code: a case study. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 191–201. DOI: 10.1145/2786805.2786855.

[78] Boris Schäling. 2014. The boost c++ libraries, chapter 8. boost.regex. (Sept. 2014). Retrieved January 08, 2023 from https://theboostcpplibraries.com/boost.regex.

[79] Tarique Siddiqui, Paul Luh, Zesheng Wang, Karrie Karahalios, and Aditya G Parameswaran. 2021. From sketching to natural language: expressive visual querying for accelerating insight. *ACM SIGMOD Record*, 50, 1, 51–58. DOI: 10.1145/3471485.3471498.

[80] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. 2017. Accelerating pattern matching queries in hybrid cpu-fpga architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*, 403–415. DOI: 10.1145/3035918.3035954.

[81] Evangelia Sitaridi, Orestis Polychroniou, and Kenneth A Ross. 2016. Simd-accelerated regular expression matching. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, 1–7. DOI: 10.1145/2933349.2933357.

[82] Sandeep Tata and Jignesh M Patel. 2003. Piqa: an algebra for querying protein data sets. In *15th International Conference on Scientific and Statistical Database Management, 2003.* IEEE, 141–150. DOI: 10.1109/SSDM.2003.1214975.

[83] Sandeep Tata, Jignesh M. Patel, James S Friedman, and Anand Swaroop. 2006. Declarative querying for biological sequences. In *22nd International Conference on Data Engineering (ICDE'06).* IEEE, 87–87. DOI: 10.1109/ICDE.2006.47.

[84] The Apache Software Foundation. 2022. Apache lucene 9.4.2 documentation. (Nov. 2022). Retrieved January 17, 2023 from https://lucene.apache.org/core/9_4_2/index.html.

[85] Dominic Tsang and Sanjay Chawla. 2011. A robust index for regular expression queries. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, 2365–2368. DOI: 10.1145/2063576.2063968.

[86] Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. 2004. Deterministic memory-efficient string matching algorithms for intrusion detection. In *IEEE INFOCOM 2004.* Vol. 4. IEEE, 2628–2639. DOI: 10.1109/INFCOM.2004.1354682.

[87] Jan Van Lunteren. 2006. High-performance pattern-matching for intrusion detection. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications.* Citeseer, 1–13. DOI: 10.1109/INFOCOM.2006.204.

[88] Stijn Vansummeren. 2003. Unique pattern matching in strings. *arXiv preprint cs/0302004*.

[89] 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* (CCS '11).

Association for Computing Machinery, Chicago, Illinois, USA, 297–308. ISBN: 9781450309486. DOI: 10.1145/2046707.2046741.

[90] Kai Wang, Yaxuan Qi, Yibo Xue, and Jun Li. 2011. Reorganized and compact dfa for efficient regular expression matching. In *2011 IEEE International Conference on Communications (ICC)*. IEEE, 1–5. DOI: 10.1109/icc.2011.5963291.

[91] Shicheng Wang, Menghao Zhang, Guanyu Li, Chang Liu, Zhiliang Wang, Ying Liu, and Mingwei Xu. 2022. Scalable and cost-efficient multistring pattern matching with programmable switches. *IEEE/ACM Transactions on Networking*. DOI: 10.1109/TNET.2022.3202523.

[92] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: a fast multi-pattern regex matcher for modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 631–648. ISBN: 978-1-931971-49-2. https://www.usenix.org/conference/nsdi19/presentation/wang-xiang.

[93] Peter T. Wood. 2012. Query languages for graph databases. *SIGMOD Rec.*, 41, 1, (Apr. 2012), 50–60. DOI: 10.1145/2206869.2206879.

[94] Chengcheng Xu, Shuhui Chen, Jinshu Su, Siu-Ming Yiu, and Lucas CK Hui. 2016. A survey on regular expression matching for deep packet inspection: applications, algorithms, and hardware platforms. *IEEE Communications Surveys & Tutorials*, 18, 4, 2991–3029. DOI: 10.1109/COMST.2016.2566669.

[95] Yang Xu, Junchen Jiang, Rihua Wei, Yang Song, and H Jonathan Chao. 2014. Tfa: a tunable finite automaton for pattern matching in network intrusion detection systems. *IEEE Journal on Selected Areas in Communications*, 32, 10, 1810–1821. DOI: 10.1109/JSAC.2014.2358856.

[96] Xifeng Yan, Philip S Yu, and Jiawei Han. 2004. Graph indexing: a frequent structure-based approach. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, 335–346. DOI: 10.1145/1007568.1007607.

[97] 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 International Conference on Communications (ICC)*. IEEE, 1–7. DOI: 10.1109/ICC.2016.7511199.

[98] Fang Yu, Zhifeng Chen, Yanlei Diao, Tamil 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 symposium on Architecture for networking and communications systems*, 93–102. DOI: 10.1145/1185347.1185360.

[99] Cha Zhang and Yunqian Ma. 2012. *Ensemble Machine Learning: Methods and Applications*. Springer Publishing Company, Incorporated. ISBN: 1441993258.

[100] Zhenjie Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, and Divesh Srivastava. 2010. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 915–926. DOI: 10.1145/1807167.1807266.

[101] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C Hoe, Vyas Sekar, and Justine Sherry. 2020. Achieving 100gbps intrusion prevention on a single server. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 1083–1100.

[102] Li Zhou. 2015. A survey on contextual multi-armed bandits. *arXiv preprint arXiv:1508.03326*.

Received October 2022; revised January 2023; accepted February 2023