Swift Regular Expression Matching

Anurag Khandelwal
anuragk@berkeley.edu

Rachit Agarwal
ragarwal@berkeley.edu

Ion Stoica
istoica@berkeley.edu

ABSTRACT

Queries involving Regular Expressions (RegEx) have a wide range of applications including textual data analytics, bioinformatics, natural language processing, information retrieval and interactive queries on graph data. However, the recent growth in dataset sizes have rendered existing techniques for executing RegEx queries inefficient. Data scans do not scale well with input size, leading to high query latency; and, existing techniques that avoid data scans — referred to as "black-box" approaches — offer little or no benefit over data scans for RegEx. The latter typically execute RegEx queries by decomposing the query along operators, computing intermediate results for individual sub-queries (using indexes and/or partial data scans) and combining the intermediate results along respective operators.

We analyze the black-box approach and identify operators for which the black-box approach can be far from optimal. We then propose Swift, a suite of optimizations that allow avoiding the black-box approach for such operators. The design of Swift is independent of the underlying data structures; we implement Swift over several data structures (suffix trees, compressed indexes, etc.) and show that Swift achieves significant speedups over existing popular open-source systems, sometimes by as much as two orders of magnitude. Swift is completely open-sourced, and is already being used in production clusters.

1. INTRODUCTION

Regular expressions (RegEx) are a powerful tool for text analytics and information extraction. Traditionally, RegEx have been used in applications like textual data analytics [3, 39], information extraction [15, 16, 19, 22, 23, 33, 34] and bioinformatics [28, 37]. Unsurprisingly, efficiently executing queries involving RegEx is a problem that has been studied for decades.

However, over the last few years, RegEx have witnessed a renewed interest due to queries involving RegEx becoming both more important and more challenging. Indeed, the rise of social media and Internet of Things (IoT) has enabled access to a data source that is information rich, but also unstructured and noisy. To extract insights from this data source, increasingly many applications use RegEx across various stages in their data analytics pipeline including natural language processing [29, 38, 45, 47], recommender systems [6, 9] and even interactive queries on graph data [13, 14, 24, 25]. One case in point is Spark [2], a popular open-source framework for distributed data analytics, where users frequently execute complex RegEx queries for text analytics and machine learning pipelines.

Queries involving RegEx have also become more challenging due to *tremendous growth in amount of data*, which in turn has led to new scalability issues. Specifically, one of the traditional approaches to executing RegEx queries is *full-data scans*, supported by DFA/NFA [3, 7, 8, 39]. While efficient on small data sets, scanning the entire dataset does not scale well with input data size, resulting in high query latency as the input size grows to tens or hundreds of gigabytes [20, 49]. Arguing the inefficiency of data scans in software, recent industry systems propose using hardware accelerators to speed up data scans [1, 40].

Full-data scans can be avoided using *m-gram indexes* [20]. The idea is to preprocess the input and index *tokens* of length m, either for multiple [20, 44] or all values of m [12, 35, 50]. RegEx-friendly indexes are often considered space-inefficient [20]. However, recent research has shown that their space requirements can be reduced down to no more than the input size without asymptotic increase in query latency [10, 31]. RegEx queries using such indexes are usually executed by decomposing the query into multiple tokens along RegEx operators, searching for each token individually (using index and/or partial data scans), and combining the intermediate results based on respective operators (§3). We call this the "black-box" approach.

In this paper, we first *analyze the performance of the black-box approach* (§3). We show that, under the standard algorithmic cost model, the black-box approach executes in near-optimal time if the RegEx query comprises of `Union, Repeat` and `Wildcard` operators only! We also show that if the query contains `Concatenation` operator, the execution time of the black-box approach could be far from optimal.

We then present Swift[1]: a simple, yet efficient, suite of transformations that optimize RegEx execution for queries containing `Concatenation` operator (§4). Swift first constructs a RegEx execution tree (referred to as RTree) in a manner that traversing an RTree in a bottom-up fashion is equivalent to executing the black-box approach. Swift transformations, when applied on this RTree, ensure that the black-box approach: (1) is used only for `Union, Wildcard` and `Repeat` operators; and (2) can be avoided altogether for `Concat` operator for most queries.

The design and implementation of Swift optimizations is independent of the underlying data structures used to perform search of individual tokens after decomposing the original RegEx query. We have implemented Swift on top of a variety of data structures, including inverted indexes [44], suffix trees [50], suffix arrays [35], compressed suffix trees [12], and compressed suffix arrays [10, 31, 42, 43]. An implementation of Swift on top of all these data structures is also available in the open-source release.

[1]We have open-sourced our implementation of Swift, including all the datasets and queries necessary to reproduce our results: https://github.com/amplab/swift. Moreover, we have also implemented Swift on top of Spark; this implementation is being used in production and can be easily run on any Spark cluster.

We evaluate Swift over real-world and benchmark datasets for applications from text analytics and bioinformatics (§5). We show that Swift transformations help in a wide range of queries, leading to significant (as much as two orders of magnitude) improvements over the black-box approach. We also compare Swift against popular open-source systems from each application that support RegEx query execution, including ElasticSearch [3], MongoDB [39], ScanProsite [27] and Spark [2]. We find that Swift achieves significant speedups compared to these systems, often as high as three orders of magnitude.

In summary, our contributions are three-fold:

- We analyze the black-box approach to executing RegEx queries. We show that the black-box approach over RegEx queries containing only `Union`, `Wildcard` and `Repeat` operators executes in near-optimal time; however, when the query contains `Concat` operator, the execution time of black-box approach could be far from optimal.
- We present Swift: a simple, yet efficient, set of transformations that ensure that the black-box approach is executed over `Union`, `Wildcard` and `Repeat` operators, and can be avoided for the `Concat` operator for most queries.
- We implement (and provide an open-source implementation of) Swift on top of a wide range of data structures including inverted indexes, suffix trees and compressed indexes, as well as on top of Spark [2], one of the popular distributed computing frameworks used in applications varying from text analytics to machine learning. We evaluate Swift against the black-box approach, and against popular open-source systems that support RegEx queries. The evaluation shows that Swift leads to significant speed up in RegEx query execution across a wide range of underlying data structures, queries, and applications.

2. PRELIMINARIES

We start with the notation used in the paper and the class of data structures for which our results apply.

Notation. Throughout the paper, we use the usual definitions of RegEx operators, as summarized in Table 1. The supported RegEx syntax is the POSIX extended standard [4]. Let Σ denote a totally ordered set of alphabets in the input. The operators are interleaved by *tokens*, that can be either:

- A *character class*, denoted by '`[]`'; for example, `[0-9a-dA-F]` represents any character from `0` through `9`, `a` through `d`, and `A` through `F`;
- An *m-gram*, which is a sequence of m alphabets from Σ.

RTree. A RegEx can equivalently be represented as a binary tree that takes standard precedence constraints between operators into account [30, 48]. We call this tree an RTree. Each internal node of the RTree represents a RegEx operator, while the leaves represent tokens (see Figure 1).

Consider an example RegEx $(RE_1).*\ (RE_2)(RE_3)$. There are at least two ways to construct at RTree for the above RegEx: one with `Wildcard` operator as the root and the other with `Concatenation` as the root. In the former, the sub-RegEx $(RE_2)(RE_3)$ is evaluated first, and then the intermediate results are combined along the `Wildcard` operator; in the latter, the sub-RegEx $(RE_1).*\ (RE_2)$ is evaluated first, and then the intermediate results are combined along the `Concatenation` operator. The performance of the two execution strategies depends on the cardinality of RE_1, RE_2 and RE_3 in the underlying dataset. The problem of constructing an optimized RTree has been explored in a number of previous works [11, 18, 30, 48] and is orthogonal to Swift optimizations and techniques. We consider an optimized RTree as an input to Swift.

Table 1: Supported operator classes.

Operator	Contents	Explanation
`Concat`	$(RE_1)(RE_2)$	RE_2 immediately follows RE_1
`Union`	$RE_1 \mid RE_2$	Either RE_1 or RE_2
`Repeat`	RE?	`Concat` of RE with RE
	RE*	Zero or one (?)
	RE+	Zero or more (*)
		One or more (+)
`Wildcard`	$(RE_1).*\ (RE_2)$	RE_2 occurs anywhere after RE_1

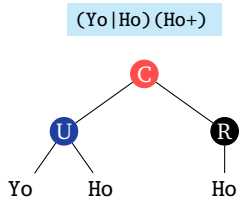


Figure 1: RTree for RegEx `(Yo|Ho)(Ho+)`. Nodes represent `Concat (C)`, `Union (U)` and `Repeat (R)` operators.

Scope. The search results for individual tokens can be computed using a wide variety of techniques [10, 12, 20, 31, 35, 44, 50]. In this paper, we focus on techniques that avoid full-data scans, including:

- A `k-gram` *index*, that supports search of tokens of some fixed length k [36, 44]; tokens of length different than k can be searched either via partial scans over index and/or input.
- An *arbitrary* `m-gram` *index*, that supports search of tokens of arbitrary length m. Examples include suffix trees [50], suffix arrays [35], corresponding compressed data structures [10, 31].

For the case of semi-structured data, we assume that the indexes above map each token to a `(documentID, offset)` pair, where the latter is the `offset` into the `documentID` where the token occurs. For ease of description, we drop the `documentID` in the better half of the paper and assume the input file to be a flat unstructured file. In Appendix A.1, we adapt all the algorithms and techniques for flat unstructured files to semi-structured data without any change in the asymptotic complexity. Irrespective of the indexing technique, we also assume access to the input file to support partial data scans.

One way to use indexes is to filter the documents that contain tokens in the query, and to execute RegEx via full scans on filtered set of documents. This works well when tokens have high selectivity, but may require full data scans for many queries (see [20] for detailed discussion). For instance, as discussed in §5, each and every query in the bioinformatics application will be executed using full data scans when using indexes for filtering only. The black-box approach outlined in §3 uses indexes more aggressively, performing partial data scans very infrequently.

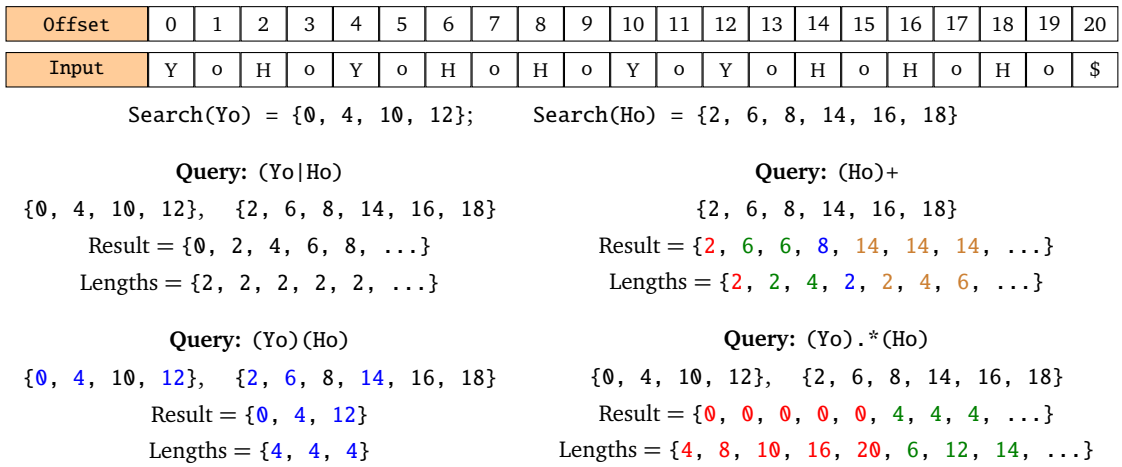


Figure 2: Illustration of the third step in black-box approach from §3 — executing algorithms in Appendix A on an example input file (the top row shows the file offsets for ease of illustration). The intermediate search results (i.e., offsets into the input file) for the 2-grams `Yo` and `Ho` are shown next. (top left) The `Union` operator outputs the set union of the offsets for the two operands. (bottom left) The `Concat` operator outputs all left operand offsets for which there exists a right operand offset satisfying $\texttt{offset}_{\texttt{right}} = \texttt{offset}_{\texttt{left}} + \texttt{length}_{\texttt{left}}$. (top right) The `Repeat` operator is similar to the `Concat` operator except for `length` admits values depending on last result. (bottom right) The `Wildcard` operator outputs all left operand offsets for which there exists a right operand offset satisfying $\texttt{offset}_{\texttt{right}} \geq \texttt{offset}_{\texttt{left}} + \texttt{length}_{\texttt{left}}$.

3. BLACK-BOX REGEX

In this section, we outline and analyze the black-box approach to executing RegEx queries.

Black-box RegEx. The "black-box" approach can be summarized in three steps (see example below):

1. Construct an RTree;
2. Compute search results (offsets into the input file) for each leaf of the tree (token) individually[2]
3. Traverse the tree bottom up, generating the results at each operator node using intermediate results for left and right subtrees. Algorithms to combine intermediate results for each operator[3] are in Appendix A and are illustrated in Figure 2.

Example. Consider a query `(Yo|Ho)(Ho+)` over the input file of Figure 2. The black-box approach first constructs an RTree (Figure 1) and computes the offsets for individual tokens (`{Yo, Ho}`). The RTree is then traversed bottom-up — token results are first used to compute the result for `(Yo|Ho)` and for `(Ho)+`, as in Figure 2, and then combined along the `Concat` operator to get the final result `{4, 12, 14}`. Note that to combine the results across multiple operators, the length for corresponding intermediate results (*e.g.*, 2 for `(Yo|Ho)`) also needs to be tracked.

3.1 Analysis of Black-box RegEx

We now analyze the black-box approach under the standard RAM computational model [21] [4]. Specifically, we obtain the following result for the individual operator algorithms from Appendix A (proofs in Appendix A):

[2]Any of the techniques from §2 may be used to compute intermediate search results; thus the "black-box" approach.

[3]We believe these algorithms to be standard, but outline them for sake of completeness of our analysis results.

[4]While a standard for algorithmic analysis, the RAM computation model ignores effects of data caching. Nevertheless, it provides a rough estimate of the efficiency of the individual operators in the black-box approach. Our evaluation (§5) takes this limitation into account by ensuring that all data fits in memory.

Lemma 1 *Given the intermediate results for the left and the right subtree as sorted arrays of size m and $n \geq m$, there exist algorithms for* `Union, Repeat, Wildcard` *and* `Concat` *operators that combine the intermediate results in time $O(s_o), O(s_o), O(s_o \log n)$ and $O(m+n)$, respectively, where s_o is the final output cardinality.*

It is known that, under the RAM computational model, the time complexity of an algorithm is lower bounded by the output size [21]. Since the output cardinality s_o is dependent on the input file and is unknown a priori, the above lemma shows that *independent of the cardinality of the results for the left and the right subtree*, the `Union, Repeat` and `Wildcard` operators combine these results in *almost optimal* time for any fixed RTree[5]. However, such is not the case for the `Concat` operator — the output cardinality for the `Concat` operator ($O(1)$ in the worst-case) can be arbitrarily smaller than the cardinality of results for the left or the right subtree. Thus, the `Concat` operator when operating on intermediate results of the left and the right subtree may end up performing significantly more operations than ideal — linear in the output size — making the black-box approach inefficient.

Lemma 1, thus, suggests two possible ways to improve the execution latency for any given RegEx. First, if possible, avoid executing `Concat` operator on intermediate results of the left and the right subtree. Second, transform the RTree so that the `Union, Repeat` and `Wildcard` operators are pushed up the RTree; compared to the original RTree, the output cardinality s_o for these operators is smaller up the RTree, making these operators more efficient. In §4, we present a set of transformations on the RTree that achieve these goals.

Concatenation of Repeats of Character Class tokens. Consider a RegEx $(R_1)(R_2+)$ with two tokens R_1, R_2, where R_1 is an `m-gram` and R_2 is a character class. Indeed, one way to avoid the black-box approach for this particular case of the `Concat` operator is to search for the offsets of R_1, and then perform a *partial scan* around these offsets to check if the following characters belong to R_2. We show in [32, Appendix B] that, in this case, partial

[5]The `Wildcard` operator requires an extra logarithmic factor in terms of the cardinality of the intermediate results.

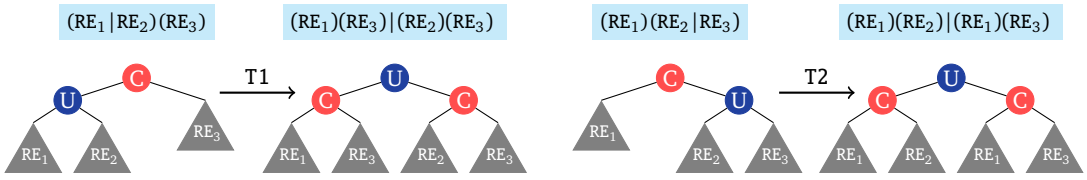


Figure 3: `Pull-Up Union` (§4.1): transformation `T1` is used if the `Union` operator is the left child, and `T2` otherwise.

scans perform better than combining individual results for `R1` and `R2` under the above cost model (independent of the input file). Intuitively, this follows from the result of Lemma 1, which shows that the `Concat` operator may become increasingly inefficient as the cardinality of intermediate results increases. This is especially the case when either of R_1, R_2 is a repeat of character class, since in general, the cardinality for repeats of character class is usually very large.

End-to-end performance. The end-to-end performance of the black-box approach depends on the time taken to construct the RTree, to search for leaf tokens, and to traverse up the tree combining intermediate results at nodes. In our experiments, we found that the last step is indeed the performance bottleneck (thus making Lemma 1 result more relevant). Intuitively, this is because constructing an RTree (scanning the RegEx once) and searching for individual tokens in index (binary search) is extremely fast. The performance of the third step, in turn, requires combining intermediate results across the operators along the RTree, which is significantly more complex.

4. Swift

We now describe Swift, a set of transformations for the RTree that improves upon the black-box approach using two ideas. First, it transforms a naïvely built RTree into one where most `Union, Wildcard` and `Repeat` operators are not the children of a `Concat` operator (§4.1, §4.2, §4.3). These operators are, thus, pushed up the tree and operate in a near-optimal manner as shown in Lemma 1. Second, it avoids the black-box approach for the `Concat` operator for most RegEx queries (§4.4). We finally show how to combine these two ideas to construct an efficient end-to-end RegEx execution engine (§4.5).

4.1 Pull-Up Union

The `Pull-Up Union` transformation attempts to transform a given RTree into one where `Union` operator is not a child of a `Concat` operator. The transformation is formally described in Algorithm 1, and is illustrated in Figure 3. The transformation uses a simple observation that a RegEx of the form $(RE_1|RE_2)(RE_3)$ is equivalent to $(RE_1)(RE_3)|(RE_2)(RE_3)$, for arbitrary RegEx RE_1, RE_2, RE_3. However, the ordering of the `Union` and the `Concat` operands needs to be handled carefully (see Figure 3). Note that if both children of the `Concat` operator are `Union` operators, the transformation needs to be applied recursively (as in Algorithm 1) since the transformation introduces new `Concat` nodes in the RTree.

4.2 Pull-Up Wildcard

The `Pull-Up Wildcard` transformation attempts that the resulting RTree does not have a `Wildcard` operator as a child of a `Concat` operator. The transformation builds upon another simple observation that a RegEx of the form $(RE_1)(RE_2.*RE_3)$ is equivalent to $(RE_1)(RE_2).*RE_3$. Figure 4(a) illustrates this transformation on a RTree containing `Wildcard` as a child of the `Concat` operator. Note that no new nodes are introduced, and thus, the transformation does not need to be applied recursively.

Algorithm 1 `PullUpUnion`

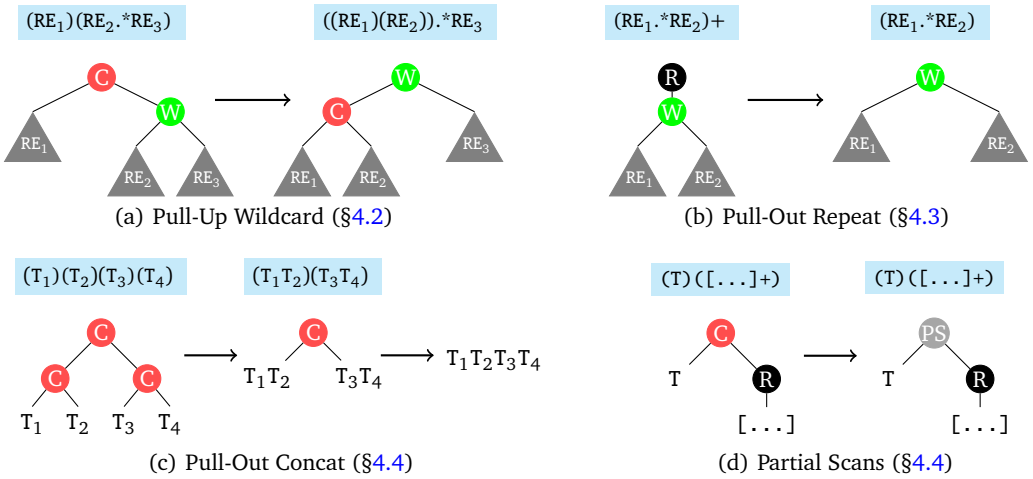
```
1:  procedure Pull-Up-Union(node: RTree)
        /* Base case: terminate if leaf node is a token. */
2:      if node type is Token then
3:          return
4:      end if
        /* Pull up unions in left and right sub-tree. */
5:      pullUpUnion(node.left)
6:      pullUpUnion(node.right)
7:      if node type is Concat then
            /*Apply transformations (recursively)*/
8:          if node.left type is Union then
9:              apply transformation T1 to node (Figure 3)
10:         else if node.right type is Union then
11:             apply transformation T2 to node (Figure 3)
12:         end if
13:         pullUpUnion(node.left)
14:         pullUpUnion(node.right)
15:     end if
16:     return
17: end procedure
```

4.3 Pull-Out Repeat

Unlike `Union` and `Wildcard` operators, ensuring that a `Repeat` operator is not a child of a `Concat` operator is more challenging. Swift only partially handles this case — when the child of the `Repeat` operator is either a `Wildcard` operator or an `m-gram` token, the transformation *pulls out* the `Repeat` operator from the RTree. Otherwise, the subtree rooted at the `Repeat` operator (denoted by `RE+` below) is left as is.

`RE` with `Wildcard`. Note that if `RE` contains a `Wildcard` operator, the child of the `Repeat` operator is the `Wildcard` operator (due to standard precedence order). If $RE \equiv RE_1.*RE_2$, then it is easy to see that results for `RE+` are same as that of `RE`, by definition of the `Wildcard` operator. Therefore, if the (only) child of the `Repeat` operator is a `Wildcard` operator, we simply remove the corresponding `Repeat` node from the RTree (see Figure 4(b)).

`RE` with `m-gram` token. Now consider the case when `RE` does not contain a `Wildcard` operator; since Swift does not transform the RTree when `RE` contains either of `Union` or `Concat` operators, `RE` must be a token. If `RE` is an `m-gram`, the transformation exploits the observation that a `Repeat` operator can equivalently be represented as a `Union` of `Concatenations`. Specifically, let RE^i represent exactly i self-concatenations of `RE`; that is, $RE^1 = RE$,



(a) Pull-Up Wildcard (§4.2)

(b) Pull-Out Repeat (§4.3)

(c) Pull-Out Concat (§4.4)

(d) Partial Scans (§4.4)

Figure 4: Swift Transformations

$\texttt{RE}^2 = \texttt{(RE)(RE)}$, and so on. Then, the expression RE+ can be written as $\texttt{RE+} = (\texttt{RE}^1|\texttt{RE}^2|\texttt{RE}^3|...|\texttt{RE}^n)$, where n is the number of characters in the input file. The transformation, thus, replaces the repeat operator by a subtree composed of `Union` and `Concat` operators corresponding to the above expression.

However, naïvely doing this transformation will result in RTree having very large depth (due to expanding RE+ for length n, the number of characters in the input file). Indeed, in practice, there exists a small k such that $\texttt{RE}^k$ has non-zero number of occurrences while $\texttt{RE}^{k+1}$ has zero occurrences. It is therefore sufficient to expand the `Repeat` operator for only k terms. Furthermore, since RE is an `m-gram`, it suffices to perform a binary search for k — each step in the binary search looks up the index to check whether $\texttt{RE}^i$ has non-zero occurrences. This requires $\log(n)$ index lookups but is still faster than the black-box approach. The subtree rooted at the `Repeat` operator is thus replaced by a combination of `Union` and `Concat` operators. We then apply the transformations from §4.1 and §4.2 to ensure that `Concat` is not a parent of the `Union` or `Wildcard` operators.

4.4 Pull-Out Concat

Finally, we introduce a simple `Pull-Out Concat` transformation, which is executed when either of the two conditions are met. First, if both the children of a `Concat` operator are tokens (say, `T` and `T'`), the transformation *pulls out* the `Concat` operator and replaces the subtree rooted at the `Concat` operator with a new token `TT'`, a longer string that is a *string concatenation* of the two children tokens (Figure 4(c)). Second, if the child of the `Concat` operator is a `Repeat` operator with character class token as its child, the sub-RegEx must be of the form $(\texttt{R}_1)(\texttt{R}_2+)$. As discussed in §3, Swift executes this sub-expression using *partial scans*. The transformation thus pulls out the `Concat` operator and replaces it with a partial scan (`PS`) operator (Figure 4(d)).

4.5 Putting it all together

We finally connect all the pieces together, and show how Swift executes a given RegEx query. Given the query, we construct a RTree; we then traverse the RTree in a `bottom-up` fashion, applying the transformations from §4.1, §4.2 and §4.3 to transform the original RTree into one with the property that most of the `Concat` operators only have tokens or other `Concat` operators as its children. Given this new RTree, we again traverse the tree *bottom-up*, applying `Pull-Out Concat` transformation. Finally, we execute search for the tokens (corresponding to the leaves of the new RTree), and traverse the RTree bottom-up combining the intermediate results across the operators. Once the root of the tree is reached, the final query results are returned.

5. EVALUATION

We now evaluate the performance of Swift against popular open-source systems that support RegEx query execution, across a range of applications, datasets, and queries.

5.1 Experimental Setup

Datasets and Queries. Our datasets and queries are drawn from three applications: bioinformatics [28,37], text analytics [3,39], and distributed computing framework pipelines [2].

For the bioinformatics application, we use the standard Pfam-A Protein dataset [26], which is 8GB in size and consists of 46 million protein sequences, each composed of 20 distinct amino-acids represented by the standard IUPAC one letter codes [5]. Typical RegEx queries on these sequences search for *protein signatures*, that are certain important regions within the sequence. We present results for 10 randomly selected protein signature RegEx queries from the Prosite [46] database (see Table 2).

For the text analytics application, we use a collection of 4.8 million English Wikipedia articles, constituting roughly 10GB of data for our single machine experiments, and a collection of 19.2 million Wikipedia articles ($\sim$ 10GB of data) for our distributed experiments. Unfortunately, there is no standard workload for RegEx queries in text analytics; to that end, we ran all the queries from [20], and present results for queries that output non-zero results for Wikipedia dataset (see Table 3). For Spark [2], we use the same dataset and queries as text analytics application, but increase the dataset size by 4× and the cluster size by 4×. We provide details on the cluster used in our experiments below.

Data structures and Systems. We have implemented the Swift algorithms on a variety of data structures, including, Suffix Trees (ST) [50], Suffix Arrays with LCP (SA) [35], `k-gram` indexes [44], compressed suffix trees (CST), and compressed suffix arrays (CSA) [10]. Each of these data structures achieves a unique tradeoff between the storage footprint and the search la-

Table 2: Protein Signature RegEx queries taken from the Prosite Database [46]

Query ID	Query	Protein Family
Query#1	[DE][SN]L[SAN][ACDFHKMLNQPSRTWVY][ACDGFIHKMNQPSRWVY][DE].EL	GRANINS_1
Query#2	[LIVMF][LIMN]E[LIVMCA]N[PATLIVM][KR][LIVMSTAC]	CPSASE_2
Query#3	[KRG][KR].[GSAC][KRQVA][LIVMK][WY][LIVM][KRN][LIVM][LFY][APK]	RIBOSOMAL_L16_1
Query#4	[DE]GSW.[GE].W[GA][LIVM].[FY].Y[GA]	TERPENE_SYNTHASES
Query#5	Q[LIV]HH[SA]..DG[FY]H	CAT
Query#6	[AC]GL.FPV	HISTONE_H2A
Query#7	CKPCLK.TC	CLUSTERIN_1
Query#8	Y..[HP]W[FYH][APS][DE].P.KG.[GA][FY]RC[IV][RH][IV]	BTG_1
Query#9	G[MV]ALFCGCGH	MYELIN_PLP_1
Query#10	[FYW]P[GS]N[LIVM]R[EQ]L.[NHAT]	SIGMA54_INTERACT_3

Table 3: Text analysis RegEx queries taken from [20]; \d and \. refer to any digit (i.e. [0-9]) and to the dot ('.') character, respectively.

Query ID	Query	Description
Query#1	<script>.*</script>	HTML Scripts
Query#2	Motorola.*(XPC|MPC)([0-9])+([0-9a-z])*	Motorola PowerPC chip numbers
Query#3	William [A-Z]([a-z])+ Clinton	President Clinton's middle name
Query#4	1-\d\d\d-\d\d\d-\d\d\d\d	US Phone Numbers
Query#5	([a-z0-9_\.])+(([a-z0-9])+\.)*stanford\.edu	Stanford domain URLs.

tency for m-gram tokens. Figure 5 shows the storage footprint for these data structures. Note that CSA achieves a storage footprint *smaller than the input itself* since it is a compressed index that enables access to original data using the index itself. We present results for ST, SA, and CSA since these achieve strictly better space-latency tradeoff than other data structures.

We compare the performance of Swift against several open-source systems that support RegEx— ElasticSearch [3] and MongoDB [39] for the text analytics application, Apache Spark [2] for text analytics on a distributed computing platform, and ScanProsite [27] for the bioinformatics application (§5.2).

ElasticSearch uses Lucene [36] as its underlying searching and indexing engine, and executes RegEx queries using an automaton-based approach. MongoDB indexes are not supported for text documents larger than 1KB (which is the case for some of the Wikipedia articles); thus, MongoDB executes RegEx queries using full-data scans. Spark is a compute engine that can support arbitrary operations; prior to Swift, Spark used Scala's full-scan based RegEx engine to execute queries in a distributed manner. Finally, ScanProsite is a publicly available tool for executing RegEx on protein sequences using in-memory data scans. In terms of storage overhead, ElasticSearch and MongoDB have storage footprint of roughly 1.4× the input size, while Spark and ScanProsite use storage exactly 1× the input size.

We then evaluate Swift's performance across different data structures, and provide insights into the performance gains of Swift optimizations on top of the black box approach (§5.3). The rest of the paper focuses on latency of executing RegEx, over an Amazon EC2 r3.8xlarge instance with 244GB RAM (for bioinformatics and text analytics applications), and a cluster of 4 c3.4xlarge instances with 30GB RAM each (for distributed computing framework application). In both settings, the available RAM is large enough to fit each of the data structures completely in memory (for all systems).

5.2 Comparison against Existing Systems

We start by discussing the performance of Swift against existing systems that support RegEx query execution. We use CSA as our underlying data structure for Swift, since it has the worst performance across all the data structures (lower storage usually leads to higher latency). Thus, Swift with other data structures will only lead to relatively better performance.

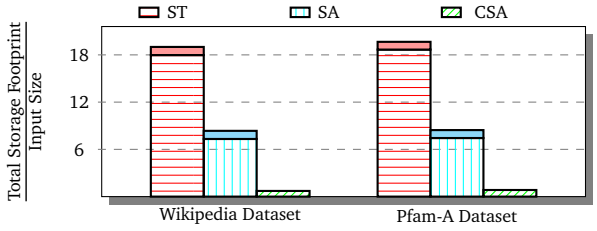
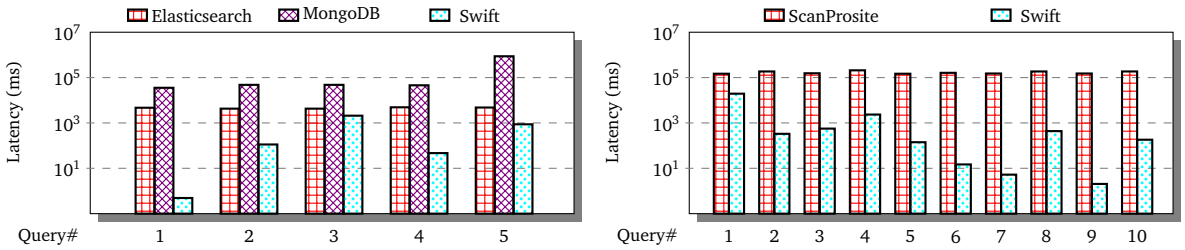


Figure 5: Storage footprint for different data structures for the Wikipedia and Pfam-A datasets. Note that ST and SA require storing the original input as well (shown as solid fill), while CSA provides similar functionality without storing the input.

Text Analytics. Figure 6(a) summarizes the query latency results for the text analytics application. MongoDB scans through all of the documents to find matches to the RegEx, while ElasticSearch scans through all the index entries. Swift, however, transforms the RTree to efficiently search for component m-grams within the RegEx, avoiding data scans as much as possible. This enables Swift to achieve much lower query latency compared to existing systems, with benefits varying from 1–3 orders of magnitude across the evaluated queries.

Bioinformatics. The query latencies for Swift and ScanProsite are summarized in Figure 6(b). Swift significantly outperforms ScanProsite, often as much as by four orders of magnitude. This is primarily because ScanProsite scans the entire data for each query (leading to similar latency across queries). Swift, on the other hand, avoids scans and can efficiently lookup the RegEx tokens from the underlying data structure (CSA, in this case), allowing it to find matches for the protein signatures much faster.

Distributed Computing Framework. Figure 7 compares the RegEx query latency for Spark, with and without Swift; the figure also shows the performance of Swift (outside Spark) for relative comparison with Figure 6(a) results. We observe that Swift significantly speeds up Spark (often by ∼ 1–2 orders of magnitude) due to avoiding Spark's full-scan based approach. For Query#3, however, Swift's implementation on Spark suffers from Java's GC overheads (since the intermediate results contain a large number of small objects) and Spark's task startup time overheads. Swift's standalone implementation, on the other hand, observes



(a) Text Analytics (b) Bioinformatics

Figure 6: Swift executes RegEx significantly faster than popular open-source systems across various application domains.

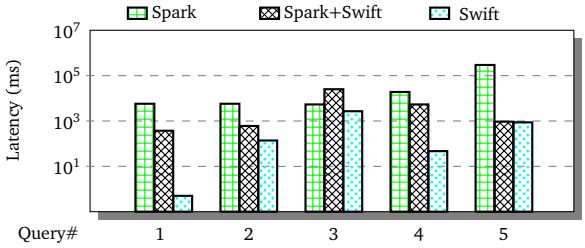


Figure 7: Swift optimizations significantly speed up analytics pipelines involving RegEx queries on distributed frameworks like Spark.

consistently low latency.

5.3 Benefits of Swift Optimizations

We now evaluate the benefits of Swift optimizations on top of the black-box approach. Our key observations are:

- **Figures 8 & 9:**[6] When a query comprises of `Union, Repeat` and `Wildcard` operators only (that execute in near-optimal time as shown in Lemma 1), Swift optimization do not provide benefits over the black-box approach. However, most queries (12 out of 15 in our evaluation) can benefit significantly using Swift, sometimes by as much as two orders of magnitude.
- **Figures 8 & 9:** The choice of data structure (ST, SA, CSA) has a significant impact on absolute RegEx execution latency, both for the black-box and the Swift approach (Figure 8 and Figure 9). Interestingly, the higher storage footprint often comes with the benefit of super-linear improvements in latency.

We discuss the results in depth below.

Queries for which Swift is unnecessary. We start the discussion with queries where Swift transformations are unnecessary (3 out of 15 queries in our evaluation). These queries either: (1) do not contain sub-optimal operators for the black-box approach (*e.g.*, `Query#1` for Wikipedia); or (2) contain character classes where both the black-box and the Swift approaches perform partial scans (*e.g.*, `Query#2, #3` for Wikipedia). Figure 8 shows that Swift has performance similar to the black-box approach for these queries.

Benefits of Swift. For most of the queries (12 out of 15 queries in our evaluation; see Figure 8 and Figure 9), Swift approach yields significant speedup over the black-box approach. These queries have three peculiar properties that make the black-box approach inefficient. First, some of these queries (*e.g.*, `Query #1-#5, #8, #10` in Pfam) contain a large number of `Concat` operators, making the black-box approach inefficient due to Lemma 1. Second, queries that contain fewer `Concat` operators (*e.g.*, `Query #6, #7, #9` in Pfam) often have large number of occurrences for individual tokens; Lemma 1 shows that as the cardinality of results for the left and the right subtree increases, the black-box approach may get worse for the `Concat` operator. Finally, all Pfam queries as well as some Wikipedia queries (*e.g.*, `Query #4, #5`) have character classes around frequently occurring tokens, making partial data scans inefficient since a large fraction of the input needs to be scanned. Swift overcomes these inefficiencies of the black-box approach using its transformations, leading to one to two orders of magnitude faster query execution than the black-box approach.

On choice of data structure. While Swift offers performance benefits across all the evaluated data structures, the absolute performance varies across data structures. Interestingly, our experiments (Figure 8 and Figure 9) show that the higher storage footprint of ST often offers super-linear latency benefits when the system is not memory-constrained — ST requires 2.2× and 26.2× higher storage than SA and CSA, and offers 4.7× and 13.3× lower latency on an average, respectively. Indeed, the tradeoff may be different for memory-constrained systems; we leave a thorough evaluation of this case for future work.

6. RELATED WORK

We compare and contrast Swift against the two traditional approaches to executing RegEx queries.

Index-based approaches. There are a multitude of techniques both for indexing and for using indexes. On the indexing front, note that tokens in RegEx by nature are not linguistically meaningful, making traditional indexing techniques (*e.g.*, inverted indexes) that use English words or other linguistic constructs [44] as keys less useful. As a result, specialized indexes for RegEx have been designed — m-gram indexes [20, 41], full-text indexes [36], and tree-based indexes [12, 17, 50], among others.

How these indexes are used to execute RegEx typically depends on the underlying indexing technique. However, at a high-level, there are two possible approaches. First, using indexes as a mechanism to filter the documents to be scanned [20]; and second, executing the entire RegEx using indexes (the black-box approach from §3). The first approach is extremely fast when the selectivity of indexed tokens is high, that is, filtering results in very few documents to be scanned. However, such is

[6] These figures show results for ST and CSA only; the performance *trends* for SA are similar to those for ST [32]. We discuss them later in the subsection.

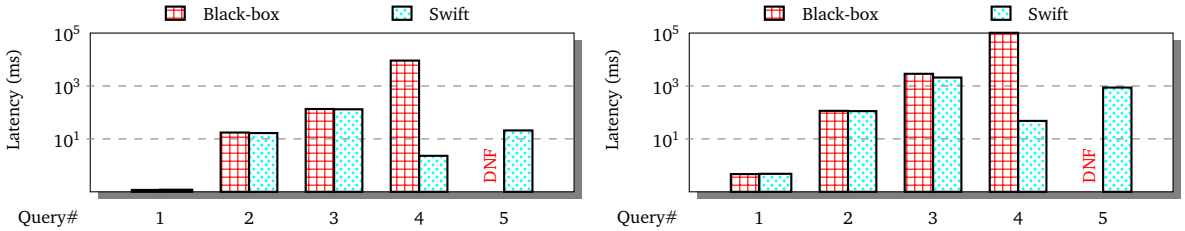


Figure 8: Performance gains for Swift optimizations over Black-box approach across different data structures for the Wikipedia dataset. Swift achieves significant speedups for queries where Swift transformations are applicable (`Query#4-5`); queries where the transformations are not applicable or require partial scans see performance similar to the black box approach (`Query#1-3`). Queries marked **DNF** did not finish within 10 minutes of execution time.

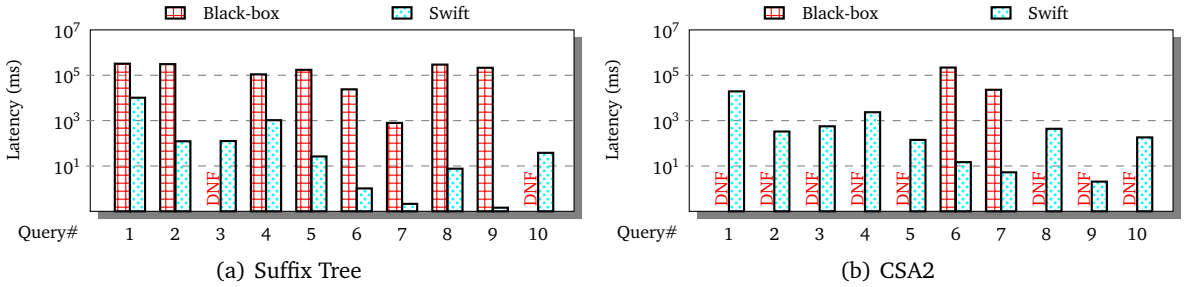


Figure 9: Performance gains for Swift optimizations over Black-box approach across different data structures for the Pfam-A dataset. Since Swift transformations are applicable for all queries, Swift offers significantly lower latency compared to the black box approach. Queries marked **DNF** did not finish within 10 minutes of execution time.

often not the case (*e.g.*, all Pfam-A queries), leading to full data scans. Swift improves the state-of-the-art for both approaches, by avoiding full-data scans as well as using optimizations to speed up the black-box approach.

Scan-based approaches, and why are index-based approaches not used in practice? Most popular open-source data stores that support RegEx queries [3, 39] resort to data scans rather than using index based techniques. We believe this is for two reasons: (i) the storage overhead of indexes specialized for RegEx queries [20]; and (ii) index-based techniques do not offer latency gains over data scans (even in our evaluation from §5, compare results for black-box approach in Figure 8 and Figure 9 with results for scan-based approaches in Figure 6). Indexes thus use more storage while providing little or no latency benefits.

However, recent research has shown that the storage overhead of indexes can be reduced down to no more than the input size without asymptotic increase in query latency [10, 31], thus motivating us to revisit index-based approaches. Moreover, Swift transformations lead to orders of magnitude speed up over the scan-based approaches for most of the evaluated queries. Swift, when operating on CSA, resolves both the above issues with index-based approaches making them an interesting choice for executing RegEx queries.

7. CONCLUSION

Motivated by new applications and by new scalability challenges due to growth in data sizes, this paper revisits the problem of efficient RegEx query execution — a powerful primitive for applications ranging from text analytics to natural language processing to graph queries to distributed data analytics pipelines in machine learning. By fundamentally identifying the sources of inefficiencies in existing approaches, this paper proposes Swift—a suite of optimizations for speeding up index-based RegEx query execution. Evaluation of Swift against the black-box approach and against popular open-source data stores shows that Swift leads to significant speed ups in RegEx query execution, sometimes by two to three orders of magnitude.

8. REFERENCES

[1] Accelerating Text Analytics Queries on Reconfigurable Platforms. http://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-atasu.pdf.

[2] Apache Spark. https://databricks.com/spark/about.

[3] ElasticSearch. http://www.elasticsearch.org/.

[4] Extended Regular Expressions. http://pubs.opengroup.org/onlinepubs/9699919799/.

[5] IUPAC One letter codes for Amino Acids. http://www.bioinformatics.org/sms/iupac.html.

[6] Recommender System with Mahout and Elasticsearch. https://www.mapr.com/products/mapr-sandbox-hadoop/tutorials/recommender-tutorial.

[7] Regular Expressions in MySQL. https://dev.mysql.com/doc/refman/5.7/en/regexp.html.

[8] Regular Expressions in Oracle. https://docs.oracle.com/cd/B19306_01/appdev.102/b14251/adfns_regexp.htm.

[9] Scalable Recommender Systems: Where Machine Learning Meets Search! https://goo.gl/g6eFf7.

[10] R. Agarwal, A. Khandelwal, and I.Stoica. Succinct: Enabling Queries on Compressed Data. In *USENIX Symposium on Network System Design and Implementation (NSDI)*, 2015.

[11] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques*. Addison Wesley, 1986.

[12] Aoe, Jun-ichi and Morimoto, Katsushi and Sato, Takashi. An

Efficient Implementation of Trie Structures. *Software: Practice and Experience*, 1992.

[13] P. Barceló Baeza, M. Romero, and M. Y. Vardi. Semantic Acyclicity on Graph Databases. In *ACM Symposium on Principles of Database Systems (PODS)*, 2013.

[14] Barceló, Pablo and Libkin, Leonid and Reutter, Juan L. Querying Graph Patterns. In *ACM Symposium on Principles of Database Systems (PODS)*, 2011.

[15] P. Bohannon, N. Dalvi, Y. Filmus, N. Jacoby, S. Keerthi, and A. Kirpal. Automatic Web-scale Information Extraction. In *ACM International Conference on Management of Data (SIGMOD)*, 2012.

[16] F. Brauer, R. Rieger, A. Mocan, and W. M. Barczynski. Enabling Information Extraction by Inference of Regular Expressions from Sample Entities. In *ACM International Conference on Information and Knowledge Management (CIKM)*, 2011.

[17] C.-Y. Chan, M. Garofalakis, and R. Rastogi. RE-tree: An Efficient Index Structure for Regular Expressions. *Proceedings of the VLDB Endowment*, 2003.

[18] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. In *ACM Symposium on Principles of Database Systems (PODS)*, 1998.

[19] L. Chiticariu, V. Chu, S. Dasgupta, T. W. Goetz, H. Ho, R. Krishnamurthy, A. Lang, Y. Li, B. Liu, S. Raghavan, F. R. Reiss, S. Vaithyanathan, and H. Zhu. The SystemT IDE: An Integrated Development Environment for Information Extraction Rules. In *ACM International Conference on Management of Data (SIGMOD)*, 2011.

[20] J. Cho and S. Rajagopalan. A Fast Regular Expression Indexing Engine. In *IEEE International Conference on Data Engineering (ICDE)*, 2001.

[21] T. H. Cormen. *Introduction to Algorithms*. 2009.

[22] N. Dalvi, R. Kumar, and M. Soliman. Automatic Wrappers for Large Scale Web Extraction. *Proceedings of the VLDB Endowment*, 2011.

[23] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Spanners: A Formal Framework for Information Extraction. In *ACM Symposium on Principles of Database Systems (PODS)*, 2013.

[24] W. Fan. Graph Pattern Matching Revised for Social Network Analysis. In *ACM International Conference on Database Theory (ICDT)*, 2012.

[25] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Adding regular expressions to graph reachability and pattern queries. In *IEEE International Conference on Data Engineering (ICDE)*, 2011.

[26] R. D. Finn, A. Bateman, J. Clements, P. Coggill, R. Y. Eberhardt, S. R. Eddy, A. Heger, K. Hetherington, L. Holm, J. Mistry, et al. Pfam: The protein families database. *Nucleic Acids Research*, 2013.

[27] A. Gattiker, E. Gasteiger, and A. M. Bairoch. ScanProsite: a reference implementation of a PROSITE scanning tool. *Applied Bioinformatics*, 2002.

[28] Gattiker, Alexandre and Gasteiger, Elisabeth and Bairoch, Amos Marc. ScanProsite: a reference implementation of a PROSITE scanning tool. *Applied Bioinformatics*, 2002.

[29] D. Gianfelice, L. Lesmo, M. Palmirani, D. Perlo, and D. P. Radicioni. Modificatory Provisions Detection: A Hybrid NLP Approach. In *ACM International Conference on Artificial Intelligence and Law (ICAIL)*, 2013.

[30] R. R. Goldberg. Finite state automata from regular expression trees. *The Computer Journal*, 1993.

[31] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 2005.

[32] A. Khandelwal, R. Agarwal, and I. Stoica. Swift Regular Expression Matching. Technical Report, UC Berkeley, Available at: http://www.cs.berkeley.edu/~anuragk/swift.pdf.

[33] R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, S. Vaithyanathan, and H. Zhu. SystemT: A System for Declarative Information Extraction. *ACM SIGMOD Record*, 2009.

[34] Y. Li, E. Kim, M. A. Touchette, R. Venkatachalam, and H. Wang. VINERy: A Visual IDE for Information Extraction. *Proceedings of the VLDB Endowment*.

[35] U. Manber and G. Myers. Suffix Arrays: A New Method for On-line String Searches. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1990.

[36] M. McCandless, E. Hatcher, and O. Gospodnetic. *Lucene in Action, Second Edition: Covers Apache Lucene 3.0*. 2010.

[37] Mulder, Michael and Nezlek, GS. Creating Protein Sequence Patterns Using Efficient Regular Expressions in Bioinformatics Research. In *IEEE International Conference on Information Technology Interfaces (ITI)*, 2006.

[38] Y. Ogawa, S. Inagaki, and K. Toyama. Automatic Consolidation of Japanese Statutes Based on Formalization of Amendment Sentences. In *Conference on New Frontiers in Artificial Intelligence (JSAI)*, 2008.

[39] E. Plugge, T. Hawkins, and P. Membrey. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. 2010.

[40] R. Polig, K. Atasu, H. Giefers, and L. Chiticariu. Compiling text analytics queries to FPGAs. In *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, 2014.

[41] D. Robenek, J. Platos, and V. Snasel. Efficient In-memory Data Structures for n-grams Indexing. In *DATESO*, 2013.

[42] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Algorithms and Computation*. 2000.

[43] K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2002.

[44] G. Salton. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. 1989.

[45] M. Shahbaz, P. McMinn, and M. Stevenson. Automated Discovery of Valid Test Strings from the Web Using Dynamic Regular Expressions Collation and Natural Language Processing. In *IEEE International Conference on Quality Software (QSIC)*, 2012.

[46] C. J. Sigrist, E. De Castro, L. Cerutti, B. A. Cuche, N. Hulo, A. Bridge, L. Bougueleret, and I. Xenarios. New and continuing developments at PROSITE. *Nucleic Acids Research*, 2012.

[47] P. Spinosa, G. Giardiello, M. Cherubini, S. Marchi, G. Venturi, and S. Montemagni. NLP-based Metadata Extraction for Legal Text Consolidation. In *ACM International Conference on Artificial Intelligence and Law (ICAIL)*, 2009.

[48] J. W. Thatcher. Tree Automata: An Informal Survey. 1973.

[49] D. Tsang and S. Chawla. A Robust Index for Regular Expression Queries. In *ACM Conference on Information and Knowledge Management (CIKM)*, 2011.

[50] P. Weiner. Linear Pattern Matching Algorithms, 1973.

APPENDIX

A. BLACK BOX ALGORITHMS

We describe the algorithms for combining the intermediate results (corresponding to the left and right subtree) for individual operators using the black-box approach. We begin with the case of flat unstructured file, where a `ResultSet` is a collection of `(offset, length)` pairs, corresponding to the offsets and the match length for the sub-RegEx rooted at a node in the RTree. We then extend these algorithms to support RegEx on semi-structured data.

Union. The trivial algorithm for the `Union` operator outputs the set union of left (`L`) and right (`R`) subtree results. The algorithm accesses each element in `L` and `R` exactly once; thus the complexity of the algorithm is $O(|L|+|R|)$. Since the output cardinality is also $s_o = |L|+|R|$, the complexity of the algorithm is $O(s_o)$.

Concat. The algorithm for the `Concat` operator scans `L` and `R`, and outputs all offsets `L[i].off` in `L` for which there exists an offset `R[j].off` in `R` such that `R[j].off = L[i].off + L[i].length` indicating that the sub-RegEx corresponding to results in `R` immediately follows the one in `L`.

The algorithm maintains two pointers (each initialized to the

Algorithm 2 `Concat`

```
1:  procedure Concat(L : ResultSet, R : ResultSet) ▷ L, sorted by (offset + length), R sorted by offset
2:      i ← 0, j ← 0;        O ← ∅
3:      while i < L.size and j < R.size do
4:          if (L[i].offset + L[i].length = R[j].offset) then
5:              Put (L[i].offset, L[i].length + R[j].length) in O
6:              i ← i+1, j ← j+1
7:          else if (L[i].offset + L[i].length < R[j].offset) then
8:              i ← i+1
9:          else
10:             j ← j+1
11:         end if
12:     end while
13:     return O
14: end procedure
```

first index of the two sets). Whenever the above condition is satisfied, the pointers are advanced to the next index for both the sets; else the pointer corresponding to the smaller offset is advanced. The algorithm terminates when one of the sets is completely scanned. Since the algorithm accesses each element in L and R at most once, the complexity is $O(|L|+|R|)$.

Repeat. The algorithm for `Repeat` is similar to that of `Concat`; the main difference is that the `length` variable (denoted by ℓ) now depends on the number of valid repetitions.

The algorithm again maintains two pointers (on the same set) and checks, in each step, whether the offset for the first pointer summed up with the current length matches the offset for the second pointer. If the condition matches, a single result is output, the length value is updated to reflect another repetition and the second pointer is advanced to check for further repetitions; otherwise, the first pointer is advanced, the length is re-initialized to zero and the second pointer is brought back to the position of the first pointer. Note that each input value corresponds to at least one output value (for single repetitions). Moreover, note that the first pointer access each element in L once; the second pointer may access any element more than once but outputs at least one output for each access. The complexity of the algorithm is, thus, $|L|+|O| < 2|O| = 2s_o$, since $L \subseteq O$.

Wildcard. The algorithm for the `Wildcard` operator takes L and R and outputs all pairs of elements `(`ℓ`, r)` such that r occurs after ℓ (the length of ℓ is taken into account accordingly).

The algorithm has two main ideas. First, to avoid unnecessary operations, the algorithm first picks the element in R that occurs after than the first element in L into the file — this ensures that there exists at least one element in L corresponds to the `Wildcard` results. Second, to find the smaller element in L, the algorithm performs a binary search rather than a scan. The binary search takes time $\log(|L|+|R|)$, and outputs, say x_1 results (the first idea ensures that $x_1 \neq 0$). The complexity of each step is, thus, $x_1 + \log(|L|+|R|) <= x_1 \cdot \log(|L|+|R|)$. The end-to-end complexity of the algorithm is: $(x_1 + x_2 + \ldots) \cdot \max(\log(|L|), \log(|R|)) = s_0 \cdot \log(|L|+|R|)$, which is linear in the output size except for the logarithmic terms.

A.1 Semi-Structured Data

We now discuss extensions to black-box algorithms for semi-structured data. We assume that indexes map tokens to a pair `(documentID, offset)`, where `offset` is the starting offset of the document into a flat file containing all documents. The pairs `(documentId, offset)` are sorted by `offsets`; given an offset, the corresponding `documentID` can be found via binary search.

Algorithm 3 `Repeat`

```
1:  procedure Repeat(L : ResultSet)        ▷ L, sorted by (offset + length)
2:      for i ← 0 to L.size do
3:          j ← i;        ℓ ← 0
4:          while (L[i].offset + ℓ = L[j].offset) do
5:              ℓ += L[j].length
6:              Put (L[i].offset, ℓ) in O
7:              j ← j+1
8:          end while
9:      end for
10:     return O
11: end procedure
```

Algorithm 4 `Wildcard`

```
1:  procedure Wildcard(L : ResultSet, R : ResultSet) ▷ L, sorted by (offset + length), R sorted by offset
2:      O ← ∅
3:      Binary search to find smallest index idx2 into R such that,
            L[0].offset + L[0].length <= R[idx2].offset
4:      for i ← idx2 to R.size do
5:          Binary search to find largest index idx1 into L such that,
                L[idx1].offset + L[idx1].length <= R[i].offset
6:          for j ← 0 to idx1 do
7:              ℓ ← (R[i].offset − L[j].offset) + R[i].length
8:              Put (L[j].offset, ℓ) in O
9:          end for
10:     end for
11:     return O
12: end procedure
```

Union. No modifications required, since each `(documentID, offset)` pair already corresponds to a valid result.

Concat. Line 4 in Algorithm 2 is modified to additionally check if both `L[i].offset` and `R[j].offset` have the same `documentID`. This ensures that two offsets are concatenated only if they belong to the same `documentID`.

Repeat. As above, Line 4 in Algorithm 3 is modified to additionally check if both `L[i].offset` and `L[j].offset` have the same `documentID`.

Wildcard. Line 8 in Algorithm 4 is modified to insert only those results into R for which `L[j]` and `R[i]` have the same `documentID`. For each `R[i]`, we determine the start and end offset for the corresponding document by consulting the `(documentId, offset)` pairs; while inserting corresponding `L[j]` entries in `ROut`, we check if `L[j].offset` lies between the begin and end offsets for `R[i]`'s document.

Since we perform an additional binary search on the list of documents for each `R[i]`, this adds an additional `log(#documents)` term to the complexity, bringing the overall complexity to $s_0 \cdot (\log(|L|+|R|) + \log(\#\text{documents}))$.

B. CHARACTER CLASSES

Character classes can be viewed as unions of single character tokens, *e.g.*, `[0-9]` can be viewed as a `Union` of character tokens `0`, `1`, `2`, ..., `9`. They can, therefore, be replaced by equivalent `Union` operators in the RegEx query. Another approach to computing character classes is by performing *partial scans* on the original input. To see how, consider the expression

$$(\mathtt{T})(\mathtt{R}_1)(\mathtt{R}_2)(\mathtt{R}_3)...(\mathtt{R}_k)$$

where `T` is a token, and each $\mathtt{R}_i$ is a character class composed of $|\mathtt{R}_i|$ characters. In order to search for such an expression, we search for token `T`, which returns, say, f_0 offsets into the input, and scan starting at each of these offsets for k characters to find all matches of the expression above.

Intuitively, if the number of occurrences of the token T is small, then it would be require fewer operations to compute the results for the expression using partial scans of the input, as opposed to computing them using the Black Box or Swift approach. We analytically determine a strategy which minimizes the number of operations required to compute such an expression. In all of our following analysis, we consider the *worst case execution time* for each of the approaches.

Partial scans. To evaluate the expression using partial scans, we scan through each of the offsets corresponding to the occurrences of `T`, and scan the input starting at those offsets for k characters. Thus, the time taken for partial scans is

$$T_s = f_0 + kf_0$$

Black Box approach. To compute the results using the Black box approach, we search for each of the characters in the character ranges, combine them using the `Union` operator, and finally combine the occurrences of `T` with the occurrences of character class tokens using the `Concat` operator. If F_i be the number of occurrences of character range $\mathtt{R}_i$, then the time taken for the black box approach is:

$$T_b = f_0 + \sum_{i=1}^{k} F_i$$

Swift approach. With the Swift approach, we perform `Pull-Up Union` followed by `Pull-Out Concat` transformations across each of the character classes (see §4) to get a transformed RTree composed of `Union`s of tokens. The time taken by the Swift approach would be depend on the number of leaves in the transformed RTree, and the time taken to perform a union of the results of the `Union` operator. It is clear to see that the maximum number of leaves in the transformed RTree is $\prod_{i=1}^{k} |\mathtt{R}_i|$.[7] The time taken to perform the final `Union` would be equal to the size of the final output (say s_0). Therefore, the time to taken by the Swift approach is given by

$$T_p = \prod_{i=1}^{k} |\mathtt{R}_i| + s_0$$

[7] In practice, however, we can prune the leaves that have zero occurrences while applying the `Pull-Out Concat` transformation. The expression shown is therefore an *overestimate* of the number of leaves in the RTree.

Execution Strategy for Black Box. For the Black Box approach to incur fewer operations, we must have

$$\begin{aligned} & T_s > T_b \\ \Rightarrow\ & kf_0 > \sum_{i=1}^{k} F_i \end{aligned} \tag{1}$$

Since the number of occurrences of a token is typically much less than that for a character class, we have,

$$F_i > f_0, \forall i \tag{2}$$

and therefore,

$$\sum_{i=1}^{k} F_i > kf_0$$

This implies that Equation 1 would never hold, and partial scans would always incur fewer operations compared to the Black box approach.

Execution Strategy for Swift. As with the Black box approach, we must have

$$\begin{aligned} & T_s > T_p \\ \Rightarrow\ & f_0 + kf_0 > \prod_{i=1}^{k} |\mathtt{R}_i| + s_0 \\ \Rightarrow\ & f_0 > \frac{\prod_{i=1}^{k} |\mathtt{R}_i|}{(1+k)} \end{aligned}$$

as $s_0 > 0$.

Since we know the values of f_0, k and $|\mathtt{R}_i|$ while executing the query, we can determine whether Swift approach requires fewer operations than a partial scan during query execution by evaluating Equation B, and pick the optimal strategy on the fly.

Repeat of Character Classes. Consider the expression

$$(\mathtt{T})(\mathtt{R}+)$$

where `T` is a token with f_0 occurrences, and `R` is a character class composed of $|\mathtt{R}|$ character tokens. In order to analyze the time taken for this scenario, we assume k to be the maximum number of repetitions, beyond which the `Repeat` operator yields no results for the expression above.

Partial scans. For partial scans, the time taken to evaluate the expression would be similar to the earlier scenario, i.e.,

$$T_s = f_0 + kf_0$$

Black Box approach. If the size of the results for the character range `R` be F, then in the worst case, the size of the output for the expression `R+` would be kF. We know from Appendix A that executing the `Repeat` operator would take $F + kF$ time. Additionally, performing the `Concat` of token `T` with the expression `R+` would take an additional $(f_0 + kF)$ time. Therefore, the total time taken for the Black box approach would be

$$T_b = f_0 + (2k+1)F$$

Swift approach. The total number of leaf nodes in the transformed RTree for the Swift approach would be given by

$$|R| + |R|^2 + |R|^3 + ... + |R|^k$$

where the ith term in the expression corresponds to performing the repeat for the character class i times. Therefore, the total time taken by the Swift approach is bound by

$$T_p = \frac{|R|(|R|^k - 1)}{|R| - 1} + s_0$$

Execution Strategy for Black Box approach. For the Black Box approach to incur fewer operations, we must have

$$\begin{aligned} & T_s > T_b \\ \Rightarrow\ & kf_0 > (2k+1)F \end{aligned} \tag{3}$$

Since the number of occurrences of a token is typically much less than that for a character class, we have,

$$F > f_0$$

and therefore,

$$(2k+1)F > kf_0$$

This implies that Equation 3 would never hold, i.e., partial scans would always incur fewer operations than the Black box approach.

Execution Strategy for Swift approach. As before, we must have

$$\begin{aligned} & T_s > T_b \\ \Rightarrow\ & f_0 + kf_0 > \frac{|R|(|R|^k - 1)}{|R| - 1} + s_0 \\ \Rightarrow\ & f_0 > \frac{|R|(|R|^k - 1)}{(k+1)(|R| - 1)} \end{aligned} \tag{4}$$

as $s_0 > 0$.

Since we know the values of f_0, and $|R|$ while executing the query, we can determine the value of k beyond which partial scans would incur fewer operations than the Swift approach using Equation 4 on the fly.