Executing RegEx Queries on Compressed Data

Anurag Khandelwal
UC Berkeley
anuragk@berkeley.edu

Rachit Agarwal
Cornell University
ragarwal@cs.cornell.edu

Ion Stoica
UC Berkeley
istoica@berkeley.edu

ABSTRACT

Queries involving Regular Expressions (RegEx) have a wide range of applications including textual data analytics, natural language processing, information retrieval, bioinformatics and interactive graph queries. However, recent growth in dataset sizes have led to new challenges in RegEx query execution. On the one hand, traditional techniques for executing RegEx queries (*e.g.*, full data scans, or word-based indexes supported by partial data scans) are memory-efficient allowing data to be stored in main memory; however, data scan latency does not scale well with data sizes. On the other hand, more powerful indexes (*e.g.*, *m*-gram indexes) have query latency that grows sub-linearly in data sizes; however, these indexes suffer from significantly higher storage overheads.

This paper builds upon recent advances in compressed data structures to achieve the best of the two worlds — memory-efficiency of scan-based techniques, and latency-efficiency of powerful in-memory indexes — using Swift, a query execution engine that executes RegEx queries directly on compressed data (without requiring data decompression). Evaluation of Swift against four open-source systems shows that Swift achieves significant speedups, sometimes by as much as two orders of magnitude. Swift is open-sourced, and is already being used in several 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, 41], information extraction [15, 16, 19, 22, 23, 34, 36] and bioinformatics [28, 39]. Unsurprisingly, efficiently executing queries involving RegEx is a problem that has been studied for decades.

However, RegEx have recently witnessed a renewed interest due to queries involving RegEx becoming both more important and more challenging. Increasingly many applications use RegEx across various stages in their data analytics pipeline including natural language processing [29,40,47,49], recommender systems [6,9] and even interactive queries on graph data [13,14,24,25]. One case in point is Apache 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.

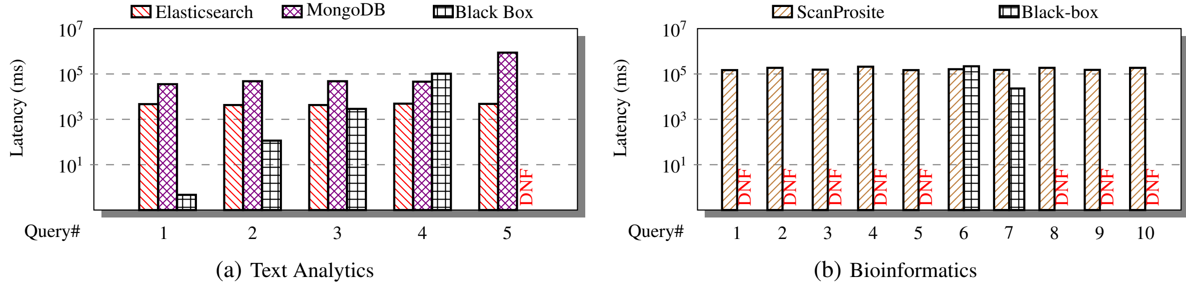
Queries involving RegEx have also become more challenging due to increasingly large data sizes in above applications. Traditional techniques for executing RegEx queries (*e.g.*, full-data scans [7, 8] and word-based indexes supported by partial data scans [3, 41, 46]) are memory-efficient, allowing the data to be stored and scanned in main memory. However, these techniques suffer from new scalability issues — data scans do not scale well with input data size, resulting in high query latency as the input size grows to tens or hundreds of gigabytes [1, 20, 42, 51]. On the other hand, powerful indexes like suffix trees and tries [12, 20, 37, 52] have significantly better query latency. However, these indexes often have high storage overheads [32, 35, 37]; for large datasets, these indexes suffer from degraded performance when the index size grows larger than the available memory [10, 20].

This paper builds upon recent advances in compressed data structures [10, 31, 44, 45] to achieve the best of the two worlds — memory-efficiency of scan-based techniques and performance of powerful indexes. These compressed data structures support exact match of strings of arbitrary length in the input data as well as random access of the input data. Our main contribution is Swift, a query execution engine that extends the functionality from exact string match to RegEx queries directly on these compressed data structures (that is, without requiring decompression). By storing and querying a compressed representation of powerful indexes, Swift not only avoids data scans but also avoids the performance degradation due to indexes not fitting in main memory.

Swift uses two key insights. The first insight is regarding the main challenge in efficiently executing RegEx queries on compressed data. Consider the following "black-box" approach (§3) — decompose the RegEx into *tokens*[1], search for individual tokens using compressed indexes (that support search of arbitrary substrings in input file), and combine the intermediate results along the RegEx operators. Figure 1 shows that naïvely executing the black-box approach can actually lead to performance even worse than scan-based techniques. The result of Figure 1 is not merely an experimental artifact; our key insight here is a simple, yet surprising, analytical result supporting the result of Figure 1 (§3) — under the standard algorithmic cost model, if the RegEx query contains `Concatenation` operator, the execution time of the black-box approach could be arbitrarily far from optimal. Perhaps more suprisingly, we show that the black-box approach executes in near-optimal time if the RegEx query comprises of `Union, Repeat` and `Wildcard` operators only.

Our second insight is that RegEx queries containing `Concatenation` can be efficiently handled via query re-writing. Intuitively, given an input RegEx query, we can perform a series of transformations to eliminate the `Concatenation` operator (by concatenating multiple smaller tokens into a longer token); this results in a new equivalent RegEx query that contains only `Union, Repeat` and `Wildcard` operators along with (potentially longer) tokens. Since the compressed indexes support exact match of arbitrary strings, we could then execute the black-box approach on this new equivalent query. We present the Swift transformations for such RegEx query re-writing in §4.

[1] Tokens are parts of RegEx that do not contain operators. For instance, a RegEx `(Yo|Ho)(Ho+)` has two tokens `Yo` and `Ho`.



(a) Text Analytics (b) Bioinformatics

Figure 1: The black-box approach for RegEx execution can be just as slow as, or even slower than, existing scan-based approaches for many RegEx queries (see §5 for details on queries and experimental setup). Queries marked **DNF** did not finish within 10 minutes of execution time.

We present evaluation of Swift[2] over real-world and benchmark datasets in §5. We compare Swift against four popular open-source systems that support RegEx query execution, including ElasticSearch [3], MongoDB [41], ScanProsite [27] and Apache Spark [2]. We find that Swift achieves significant speedups compared to these systems, often as high as two orders of magnitude.

Interestingly, many Swift techniques turn out to have more general applicability and lead to performance improvements even for uncompressed data structures. We have implemented Swift on top of a variety of data structures, including inverted indexes [46], suffix trees [52], suffix arrays [37], compressed suffix trees [12], and compressed suffix arrays [10, 31, 44, 45][3]. We present evaluation result for these data structures in [33].

In summary, this paper makes three contributions:

- We analyze the black-box approach to executing RegEx queries on compressed data. 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, RegEx query engine that enables execution of RegEx queries directly on compressed data. We evaluate Swift against four popular open-source systems that support RegEx queries. The evaluation shows that Swift leads to significant speed up in RegEx query execution latency, sometimes by as much as two orders of magnitude.
- We show that Swift techniques are applicable to several uncompressed data structures as well. In addition, we 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 Apache Spark [2].

2. PRELIMINARIES

We start with a description of the notation used in the paper.

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*, which 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`; or (b) *m-gram*, which is a sequence of *m* alphabets from Σ.

[2]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 Apache Spark; this implementation is being used in production and can be easily run on any Apache Spark cluster.

[3]Implementation also available in the open-source release.

Table 1: Supported operator classes.

Operator	Contents	Explanation
`Concat`	$(RE_1)(RE_2)$	RE_2 immediately follows RE_1
`Union`	$RE_1 \| RE_2$	Either RE_1 or RE_2

`Repeat`	
	`RE?`
	`RE*`
	`RE+`
		`Concat` of `RE` with `RE`
		Zero or one (?)
		Zero or more (*)
		One or more (+)

`Wildcard`	(RE_1).* (RE_2)	RE_2 occurs anywhere after RE_1

RTree. A RegEx can equivalently be represented as a binary tree that takes standard precedence constraints between operators into account [30, 50]. We call this tree an RTree. Each internal node of the RTree represents a RegEx operator, while the leaves represent tokens (see Figure 2). The problem of constructing an optimized RTree has been explored in a number of previous works [11, 18, 30, 50] and is orthogonal to Swift techniques. We use an optimized RTree from [18] as an input to Swift.

Compressed data structures. Compressed Suffix Arrays (CSA) [10, 31] store a compressed representation of the input file. CSAs support exact matches of arbitrary strings within the input, as well as random access to the input file. The description of these data structures is not needed to keep the paper self-contained; we refer the readers to [10, 31].

3. NEED FOR SWIFT

In this section, we outline the need for Swift using a naïve black-box approach to executing RegEx queries on compressed data.

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.
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 are outlined in §3.1 and are illustrated in Figure 3.

Example. Consider a query `(Yo|Ho)(Ho+)` over the input file of Figure 3. The black-box approach first constructs an RTree (Figure 2) 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 3, and then combined along the `Concat` operator to get the final

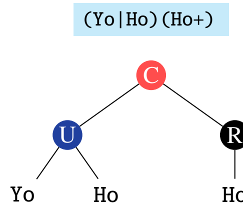


Figure 2: RTree for RegEx `(Yo|Ho)(Ho+)`. Nodes represent `Concat (C)`, `Union (U)` and `Repeat (R)` operators.

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 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[4]. We assume the input to be a 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. A discussion on extending these algorithms to support RegEx on semi-structured data is provided in [33].

Union. The trivial algorithm for the `Union` operator outputs the set union of left (L) and right (R) subtree results. Trivially, 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 $O(s_o)$.

Algorithm 1 `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
```

Concat. Algorithm 1 for the `Concat` operator scans L and R, and outputs all offsets `L[i].offset` in L for which there exists an offset `R[j].offset` in R such that `R[j].offset = L[i].offset + 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 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 $O(|L|+|R|)$.

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

Algorithm 2 `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 3 `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
```

Repeat. Algorithm 2 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 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, $|L|+|\mathcal{O}| < 2|\mathcal{O}| = 2s_o$, since $L \subseteq \mathcal{O}$.

Wildcard. Algorithm 3 for the `Wildcard` operator takes L and R and outputs all pairs of elements `(ℓ, r)` such that `r` occurs after ℓ.

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: $(x_1 + x_2 + \dots) \cdot \max(\log(|L|), \log(|R|)) = s_0 \cdot \log(|L|+|R|)$, which is linear in the output size except for the logarithmic terms.

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Input	Y	o	H	o	Y	o	H	o	H	o	Y	o	Y	o	H	o	H	o	H	o	$

`Search(Yo) = {0, 4, 10, 12};` `Search(Ho) = {2, 6, 8, 14, 16, 18}`

Query: `(Yo|Ho)`
`{0, 4, 10, 12}, {2, 6, 8, 14, 16, 18}`
Result = `{0, 2, 4, 6, 8, ...}`
Lengths = `{2, 2, 2, 2, 2, ...}`

Query: `(Ho)+`
`{2, 6, 8, 14, 16, 18}`
Result = `{2, 6, 6, 8, 14, 14, 14, ...}`
Lengths = `{2, 2, 4, 2, 2, 4, 6, ...}`

Query: `(Yo)(Ho)`
`{0, 4, 10, 12}, {2, 6, 8, 14, 16, 18}`
Result = `{0, 4, 12}`
Lengths = `{4, 4, 4}`

Query: `(Yo).*(Ho)`
`{0, 4, 10, 12}, {2, 6, 8, 14, 16, 18}`
Result = `{0, 0, 0, 0, 0, 4, 4, 4, ...}`
Lengths = `{4, 8, 10, 16, 20, 6, 12, 14, ...}`

Figure 3: Illustration of the third step in black-box approach — executing algorithms in §3.1 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 $\texttt{offset}_{\texttt{right}} \geq \texttt{offset}_{\texttt{left}} + \texttt{length}_{\texttt{left}}$.

3.2 Analysis of Black-box RegEx

We now analyze the black-box approach under the standard RAM computational model [21] [5]. Specifically, we obtain the following result for the individual operator algorithms:

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[6]. 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.

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.

Need for Swift. Lemma 1 outlines the central problem in devising a technique for executing RegEx queries on compressed data. As shown in Figure 1, the performance for queries containing `Concat` operator can be arbitrarily far from optimal, and requires careful handling for efficient execution. In the following section, we outline a query re-writing technique that enables the efficient execution of queries containing `Concat` operator through simple transformations of the query RTree.

[5] 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.

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

4. Swift

We now describe Swift, a query re-writing technique 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 (§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 4, and is illustrated in Figure 4. The transformation uses a simple observation that a RegEx of the form $(\texttt{RE}_1|\texttt{RE}_2)(\texttt{RE}_3)$ is equivalent to $(\texttt{RE}_1)(\texttt{RE}_3)|(\texttt{RE}_2)(\texttt{RE}_3)$, for arbitrary RegEx $\texttt{RE}_1, \texttt{RE}_2, \texttt{RE}_3$. However, the ordering of the `Union` and the `Concat` operands needs to be handled carefully (see Figure 4). Note that if both children of the `Concat` operator are `Union` operators, the transformation needs to be applied recursively (as in Algorithm 4) since the transformation introduces new `Concat` nodes in the RTree.

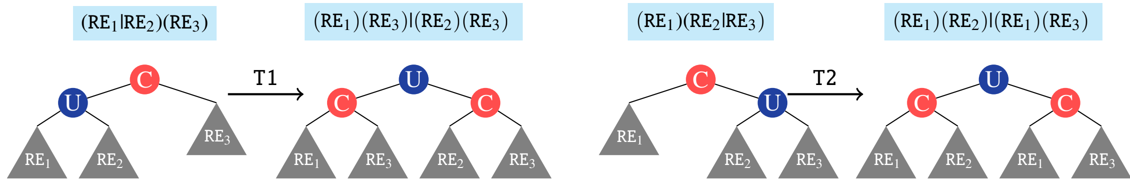
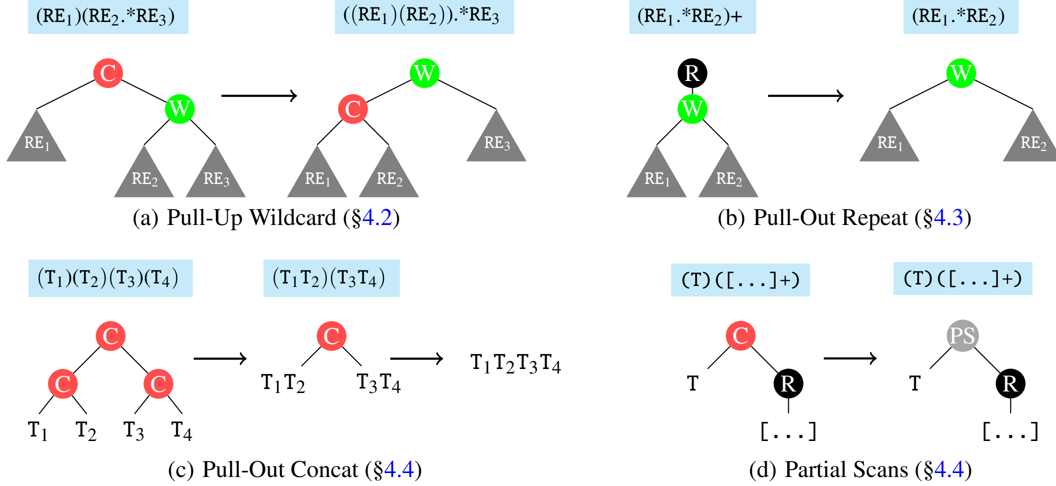


Figure 4: `Pull-Up Union` (§4.1): transformation `T1` is used if the `Union` operator is the left child, and `T2` otherwise.



(a) Pull-Up Wildcard (§4.2)

(b) Pull-Out Repeat (§4.3)

(c) Pull-Out Concat (§4.4)

(d) Partial Scans (§4.4)

Figure 5: Swift Transformations

Algorithm 4 `Pull-Up-Union` (**node**: `RTree`)

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

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 5(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.

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 5(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, RE^2 = `(RE)(RE)`, and so on. Then, the expression RE+ can be written as RE+ = (RE^1|RE^2|RE^3|...|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 RE^k has non-zero number of occurrences

Table 2: Protein Signature RegEx queries taken from the Prosite Database [48]

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.

while 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 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 5(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 $(R_1)(R_2+)$. 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 5(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.

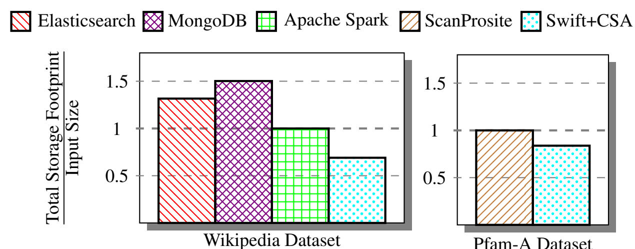


Figure 6: Storage footprint for different systems for the Wikipedia and Pfam-A datasets.

5.1 Experimental Setup

Datasets and Queries. Our datasets and queries are drawn from three applications: bioinformatics [28, 39], text analytics [3, 41], 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 [48] 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 (∼ 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 Apache Spark [2], we use the same dataset and queries as text analytics application, but increase both the dataset size and cluster size by 4×. We provide details on the cluster used in our experiments below.

Compared Systems. We compare the performance of Swift against several open-source systems that support RegEx— Elasticsearch [3] and MongoDB [41] for the text analytics application, Apache Spark [2] for text analytics on a distributed computing plat-

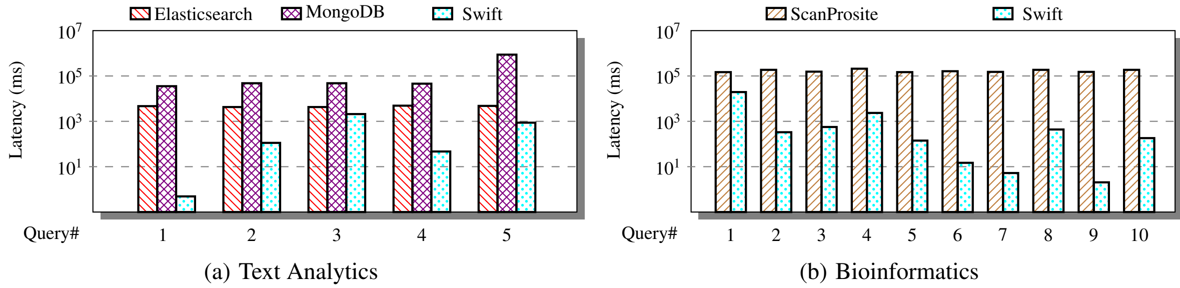


Figure 7: Swift executes RegEx significantly faster than popular open-source systems across various application domains.

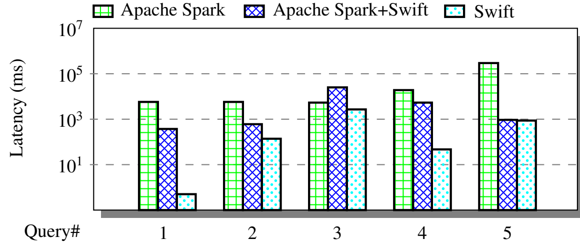


Figure 8: Swift optimizations significantly speed up analytics pipelines involving RegEx queries on distributed frameworks like Apache Spark.

form, and ScanProsite [27] for the bioinformatics application.

Elasticsearch uses Lucene [38] 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. Apache Spark is a compute engine that can support arbitrary operations; prior to Swift, Apache 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.

Finally, Swift executes RegEx queries directly on Compressed Suffix Arrays [31] as outlined in §3 and §4. Figure 6 compares the storage overhead for the different systems. Elasticsearch and MongoDB have storage footprint of $1.3-1.5\times$ the input size, while Apache Spark and ScanProsite use storage exactly $1\times$ the input size. Finally, Swift on CSA has the lowest storage footprint of $0.6-0.8\times$ the input size for different application domains, i.e., it operates on compressed data.

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.

Text Analytics. Figure 7(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 7(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 8 compares the RegEx query latency for Apache Spark, with and without Swift; the figure also shows the performance of Swift (outside Apache Spark) for relative comparison with Figure 7(a) results. We observe that Swift significantly speeds up Apache Spark (often by $\sim$ 1–2 orders of magnitude) by avoiding Apache Spark's full-scan based approach. For `Query#3`, however, Swift's implementation on Apache Spark suffers from Java's GC overheads (since the intermediate results contain a large number of small objects) and Apache Spark's task startup time overheads. Swift's standalone implementation, on the other hand, observes 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 observation is that when a query comprises of `Union, Repeat` and `Wildcard` operators only (that execute in near-optimal time as shown in Lemma 1), Swift optimizations 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. 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 9 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 9), Swift approach yields significant speedup over the black-box approach. These queries have three peculiar properties that make the black-box approach ineffi-

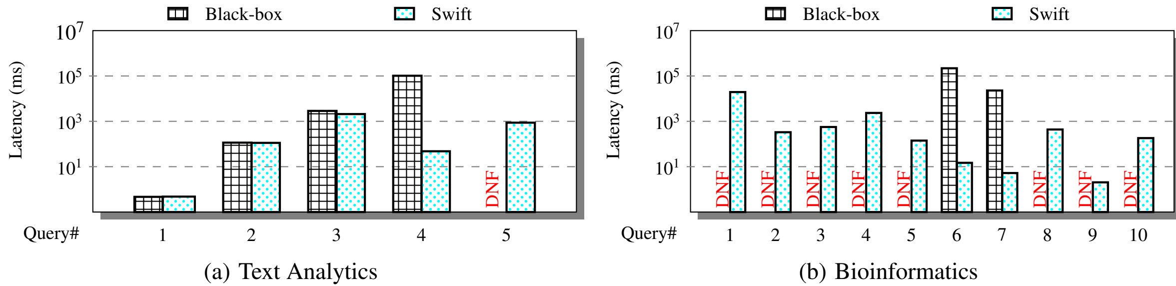


Figure 9: Performance gains for Swift optimizations over Black-box approach across different application domains. Swift achieves significant speedups for queries where Swift transformations are applicable (`Query#4-5` for Text Analytics, all queries for Bioinformatics); queries where the transformations are not applicable or require partial scans see performance similar to the black box approach (`Query#1-3` for Text Analytics). Queries marked **DNF** did not finish within 10 minutes of execution time.

cient. 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.

5.4 Generality of Swift

Although our discussions so far have been restricted to flat unstructured inputs encoded as Compressed Suffix Arrays (CSA), Swift algorithms can be adapted to more general data representations, and even several uncompressed index structures.

Index structures. Recall from §3 and §4 that the Swift query execution relies on CSA for arbitrary token searches and random access to the input. Interestingly, Swift leads to performance improvements even for uncompressed index structures that provide functionality similar to CSA [10, 12, 20, 31, 37, 46, 52]. Although not originally our goal, we have implemented Swift on top of a variety of data structures, including inverted indexes [46], suffix trees [52], suffix arrays [37], compressed suffix trees [12], and compressed suffix arrays [10, 31, 44, 45]. We provide a comparison of the storage requirements and Swift performance for the different data structures in [33].

Semi-structured Data. For 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 document where the token occurs. This allows us to adapt Swift algorithms for flat unstructured files to semi-structured data without any change in the asymptotic complexity. A detailed discussion can be found in [33].

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 [46] as keys less useful. As a result, specialized indexes for RegEx have been designed — *m*-gram indexes [20, 43], full-text indexes [38], and tree-based indexes [12, 17, 52], 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 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, 41] 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, compare results for black-box approach with scan-based approaches in Figure 1). 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 leads 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 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. We present Swift— a query execution engine that builds upon recent advances in compressed data structures to enable RegEx query execution directly over compressed data. Evaluation of Swift 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] W.-K. Hon, T. W. Lam, W.-K. Sung, W.-L. Tse, C.-K. Wong, and S.-M. Yiu. Practical aspects of compressed suffix arrays and fm-index in searching dna sequences. In *ALENEX/ANALC*, pages 31–38, 2004.

[33] 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.

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

[35] S. Kurtz. Reducing the space requirement of suffix trees. *Software Practice and Experience*, 29(13):1149–1171, 1999.

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

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

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

[39] 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.

[40] Y. Ogawa, S. Inagaki, and K. Toyama. Automatic Consolidation of Japanese Statutes Based on Formalization of Amendment Sentences. In *JSAI Conference*, 2008.

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

[42] 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.

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

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

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

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

[47] 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.

[48] 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.

[49] 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.

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

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

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

APPENDIX

A. SWIFT REGEX EXECUTION WITH OTHER DATA STRUCTURES

Intuitively, the performance benefits of Swift over the black-box approach depend on the query as well as the underlying data structure used to search `m-gram` tokens. We have implemented the black-box and the Swift approaches on a variety of data structures, including Suffix Trees (ST) [52], Suffix Arrays with LCP (SA) [37], `k-gram` indexes, compressed suffix trees (CST), and compressed suffix arrays (CSA) [10], along with support for partial scans. Each of these data structures achieves a unique tradeoff between the storage footprint and the search latency for `m-gram` tokens. We present results for ST, SA, and CSA, since these achieve strictly better space-latency tradeoff than other data structures. CSA can achieve multiple operating points on the storage-latency tradeoff space depending on the desired compression factor; we present the results for the two extremes (termed CSA1 and CSA2).

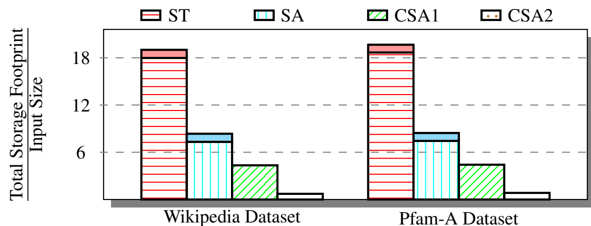


Figure 10: Storage footprint for different data structures for the Wikipedia and Pfam datasets. Note that for ST and SA, we store the original input as well (shown as solid fill), while CSA implicitly encodes the input.

On choice of data structure. While Swift offers performance benefits across all the evaluated data structures, the absolute performance depends on the underlying data structure. Figure 13 shows the performance of SA, and the two versions of CSA relative to the ST data structure; these are the same results as in Figure 11 and Figure 12, just focusing on Swift performance and scaled by the ST latency. Interestingly, the higher storage footprint of ST often offers super-linear latency benefits when the system is not memory-constrained — ST requires $2.2\times, 4.3\times$ and $26.2\times$ higher storage than SA, CSA1 and CSA2, and offers $4.7\times, 10\times$ and $13.3\times$ lower latency on an average, respectively. Indeed, the tradeoff may be different for memory-constrained systems; we leave a through evaluation of this case for future work.

B. SEMI-STRUCTURED DATA

We 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.

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

Concat. Line 4 in Algorithm 1 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 2 is modified to additionally check if both `L[i].offset` and `L[j].offset` have the same `documentID`.

Wildcard. Line 8 in Algorithm 3 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 $\log(\#\texttt{documents})$ term to the complexity, bringing the overall complexity to $s_0 \cdot (\log(|\texttt{L}|+|\texttt{R}|) + \log(\#\texttt{documents}))$.

C. CHARACTER CLASSES

Consider a RegEx $(\texttt{R}_1)(\texttt{R}_2+)$ with two tokens $\texttt{R}_1, \texttt{R}_2$, where $\texttt{R}_1$ is an `m-gram` and $\texttt{R}_2$ 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 $\texttt{R}_1$, and then perform a *partial scan* around these offsets to check if the following characters belong to $\texttt{R}_2$. In this section we show that, in this case, partial 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 $\texttt{R}_1, \texttt{R}_2$ is a repeat of character class, since in general, the cardinality for character classes is usually very large.

C.1 Analysis

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

$$(\texttt{T})(\texttt{R}_1)(\texttt{R}_2)(\texttt{R}_3)\ldots(\texttt{R}_k)$$

where `T` is a token, and each $\texttt{R}_i$ is a character class composed of $|\texttt{R}_i|$ 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

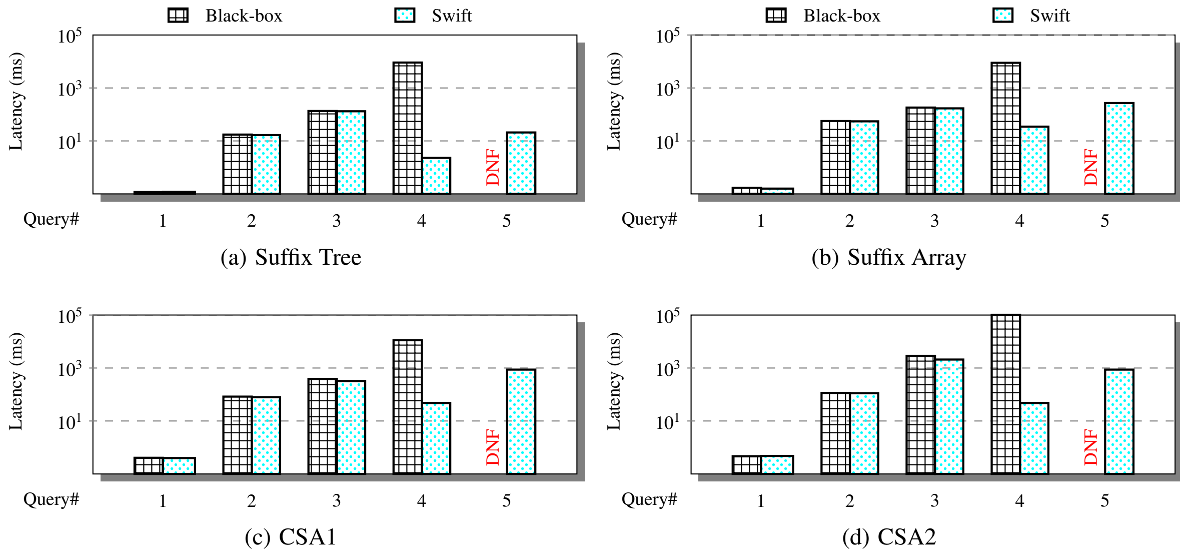


Figure 11: Performance gains for Swift optimizations over Black-box approach across different data structures for the Wikipedia dataset. Swift achieves significant speedups for queries where Swift transformations are applicable (`Query#4-5`); queries where the transformations are not applicable or require partial scans see performance similar to black box approach (`Query#1-3`). Queries marked **DNF** did not finish within 10 minutes of execution time.

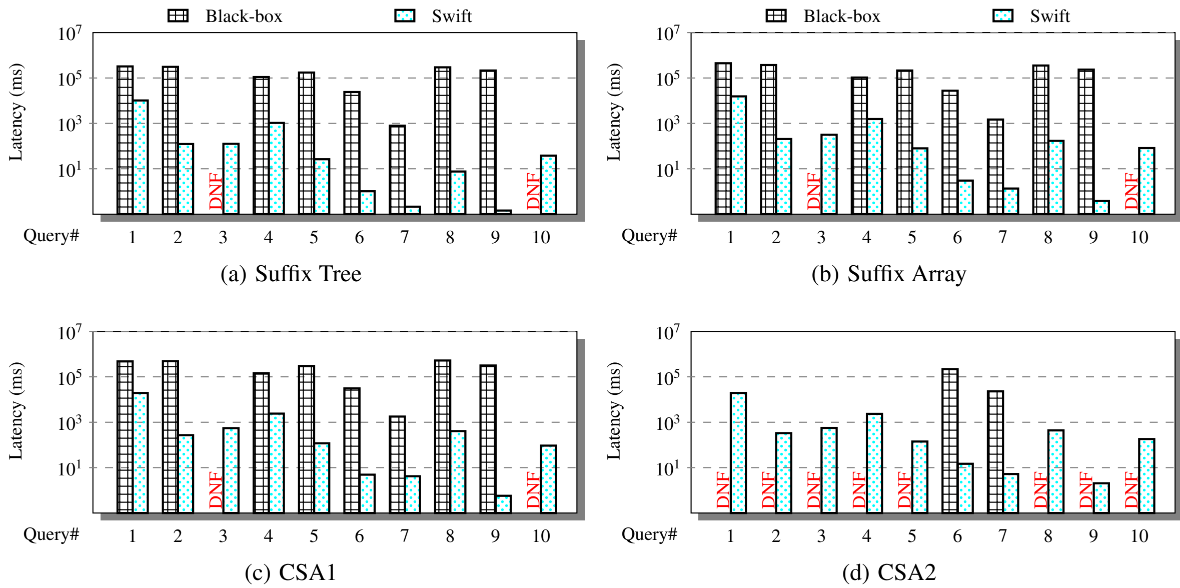


Figure 12: Performance gains for Swift optimizations over Black-box approach across different data structures for the Pfam-A dataset. Since Swift transformations are applicable for all queries, Swift offers significantly lower latency compared to the black box approach. Queries marked **DNF** did not finish within 10 minutes of execution time.

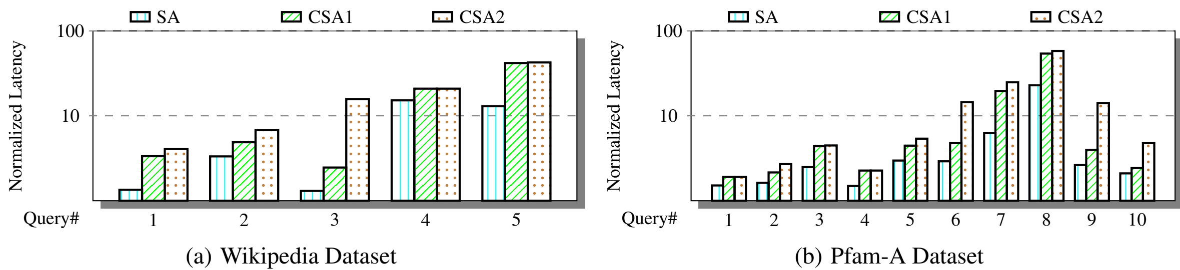


Figure 13: Comparison of Swift latency across different data-structures. Query latency results are normalized against Suffix Tree latency. Note that the higher storage footprint of Suffix Tree offers super-linear gains over Suffix Array and Compressed Suffix Arrays.

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 $\texttt{R}_i$, 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 $\prod_{i=1}^{k} |\texttt{R}_i$.[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

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

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

$$T_s > T_b$$
$$\Rightarrow k f_0 > \sum_{i=1}^{k} F_i \quad (1)$$

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

$$F_i > f_0, \forall i \quad (2)$$

and therefore,

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

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

$$T_s > T_p$$
$$\Rightarrow f_0 + k f_0 > \prod_{i=1}^{k} |\texttt{R}_i| + s_0$$
$$\Rightarrow f_0 > \frac{\prod_{i=1}^{k} |\texttt{R}_i|}{(1+k)}$$

as $s_0 > 0$.

[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.

Since we know the values of f_0, k and $|\texttt{R}_i|$ while executing the query, we can determine whether Swift approach requires fewer operations than a partial scan during query execution by evaluating Equation C.1, and pick the optimal strategy on the fly.

Repeat of Character Classes. Consider the expression

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

where `T` is a token with f_0 occurrences, and `R` is a character class composed of $|\texttt{R}|$ 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.,

$$T_s = f_0 + k f_0$$

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 §3.1 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

$$|R| + |R|^2 + |R|^3 + \ldots + |R|^k$$

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

$$T_s > T_b$$
$$\Rightarrow k f_0 > (2k+1)F \quad (3)$$

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,

$$(2k+1)F > k f_0$$

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

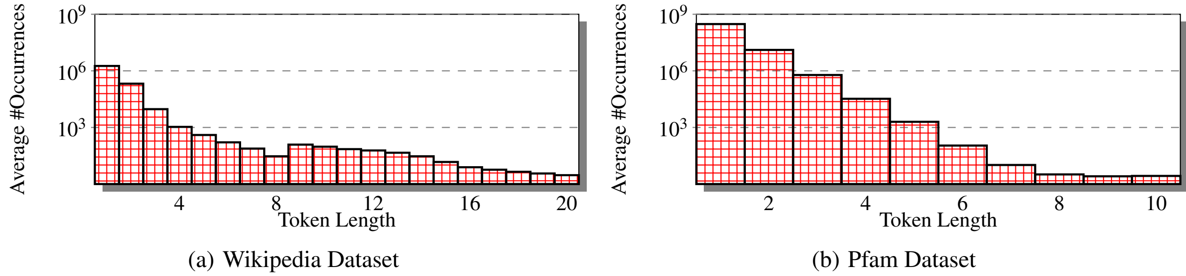


Figure 14: Variation of token frequency with token length for the Wikipedia and Pfam-A datasets. The token frequency decreases as the length of the tokens is increased for both the datasets.

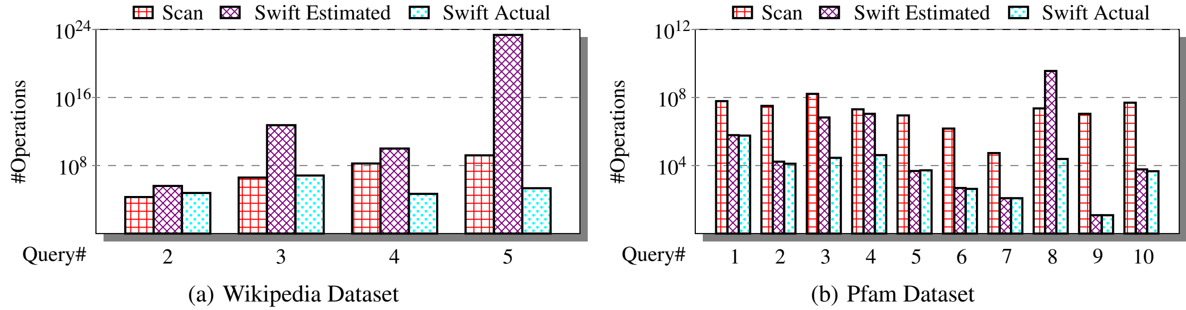


Figure 15: Comparison of the number of estimated operations for partial scans, Swift and the actual number of operations for Swift across the Wikipedia and Pfam-A datasets.

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

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.

D. OTHER EVALUATION RESULTS

Digging deeper into Swift performance: when and why it works. Irrespective of the underlying data structure, Swift achieves its performance benefits by avoiding the `Concat` operator over the intermediate results altogether. This is, for instance, the case for all queries in the bioinformatics application. Besides avoiding the suboptimal `Concat` operator, Swift achieves performance benefits due to another interesting reason. Intuitively, after the transformations are applied on the RTree, the leaves of the resulting RTree has tokens that are of length longer than the tokens in the original query. Figure 14 shows that, for both Wikipedia and Pfam-A datasets, the number of occurrences (and hence, the cardinality of intermediate results) decreases exponentially as the length of the tokens increase; we see a similar trend for the Wikipedia dataset. The operators up the RTree, hence, operate on smaller cardinality sets leading to further improvements in the query latency.

Finally, we observe that Swift performance varies significantly across queries. Interestingly, there is a particular parameter that allows us to explain this performance difference. It turns out that Swift performance is proportional to the number of leaves with non-zero occurrences in the *transformed* RTree. Of course, it is hard to find the number of leaves with non-zero occurrences apriori since it depends on the input file. We can, however, estimate this by assuming that each leaf in the RTree has non-zero number of occurrences. The number of leaves are then given by the cartesian product of the sets corresponding to each token in the original RTree. Our evaluation (see below) suggests that in most cases (except for one query, `Query#8`), the total number of leaves computed using the cartesian product provides a good estimate for the number of leaves in the transformed RTree. Intuitively, this is because most of the tokens have at least a few occurrences in large datasets.

Number of Leaves in the transformed RTree. The performance variation across different RegEx queries for Swift can be explained on the basis of the number of leaves in the transformed RTree. In particular, the latency of execution of RegEx queries in the Swift approach is proportional to the number of leaves in the transformed RTree; Figure 15 demonstrates this. Following our analysis in Appendix C, we estimate the number of operations required for Swift and partial-scan approaches for RegEx queries containing character classes; we see that our analysis provides a reasonable estimate of the actual number of operations for the queries in our evaluation. Queries where the number of leaves in the transformed RTree is high tend to require higher number of Swift operations; in such situations, our analysis provides a means to determine whether the RegEx execution should adopt partial-scans or the Swift approach.