Swift Regular Expression Matching

Paper #706

ABSTRACT

Queries involving Regular Expressions (RegEx) have a wide range of applications including document stores, bioinformatics and information retrieval. However, efficiently executing RegEx queries over large datasets remains a challenging task. Data scans do not scale well with input size; however, 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 execution time of the black-box approach can be far from optimal. We then propose Swift, a set of transformations over the original RegEx that allow avoiding the black-box approach for such operators. We implement Swift over several data structures (including suffix trees, suffix arrays, compressed indexes, etc.) and show that Swift achieves significant speedups over the black-box approach and over popular open-source data stores that support RegEx via data scans, sometimes by as much as two orders of magnitude.

1. INTRODUCTION

Regular expressions (RegEx) are a powerful tool for text and data analytics. Traditionally, RegEx have been used in a wide range of applications including XML databases [15–17], bioinformatics [35, 48] and document stores [2, 50]. 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 in the community 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 data cleaning and wrangling [6, 27, 33, 40], information extraction [18, 19, 21, 25, 26, 41, 42], natural language processing [36, 49, 56, 58] and even interactive queries [12, 13, 29, 30].

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 two traditional approaches to executing RegEx queries is *full-data scans*, supported by DFA/NFA [2, 4, 7, 50]. While efficient on small data sets, scanning the entire dataset does not scale very well with input data size, resulting in high query latency as the input size grows to tens or hundreds of gigabytes [22, 61]. Arguing the inefficiency of data scans in software, recent industry systems propose using hardware accelerators to speed up data scans [1, 51].

Full-data scans can be avoided using *m-gram indexes* [22], potentially combined with *partial data scans*. The idea is to preprocess the input and index *tokens* of length m, either for multiple [22, 44, 55] or all values of m [10, 14, 43, 62, 63]. RegEx-friendly indexes are often considered space-inefficient [22]. 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 [9, 38, 53, 54]. 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 `Union, Repeat` and `Wildcard` operators combine the intermediate results in near-optimal time. For RegEx queries that contain these operators only, the black-box approach thus executes in near-optimal time. 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: a simple, yet efficient, query rewriting technique that optimizes RegEx execution for the latter set of queries (§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 then uses a set of transformations, that, 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 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, 55], suffix trees [62, 63], suffix arrays [43, 62], compressed suffix trees [10], and compressed suffix arrays [9, 38, 54], along with support for partial data scans.

One of the by-products of Swift is support for RegEx query execution directly on a compressed representation of the input data. This is in the spirit of the seminal work in columnar stores that enabled executing analytical queries directly on compressed data [8, 59]. Our implementation of Swift on compressed suffix arrays extends it to the case of RegEx queries, enabling query execution directly on compressed data without performing data decompression (§5).

We evaluate Swift over real-world and benchmark datasets for applications from bioinformatics and document stores (§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 [2], MongoDB [50] and ScanProsite [34]. 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 Swift on top of a wide range of data structures including inverted indexes, suffix trees and suffix arrays. Our implementation on top of compressed suffix arrays enables executing RegEx queries directly on a compressed representation of the input data. 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 [3]. 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 [37, 60]. We call this tree an RTree. Each internal node of the RTree represents a RegEx operator, while the leaves represent tokens (see Figure 1).

Note that the problem of constructing an RTree is orthogonal to Swift techniques. For instance, 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. It is an interesting problem to construct an optimized RTree given the cardinality of tokens constituting the input RegEx.

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

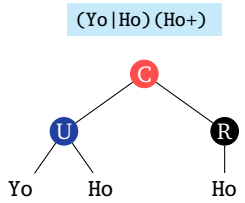


Figure 1: RTree for RegEx `(Yo|Ho)(Ho+)`. Nodes represent `Concat (C)`, `Union (U)` and `Repeat (R)` operators.

Scope. The search results for individual tokens can be computed using a wide variety of techniques [9, 10, 14, 22, 24, 28, 31, 38, 43, 44, 46, 53–55, 62, 63]. 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 [24, 28, 44, 55]; 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 [43, 62, 63], suffix arrays [46], corresponding compressed data structures [9, 10, 38, 53, 54].

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. Later in the paper, 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 [22] 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.

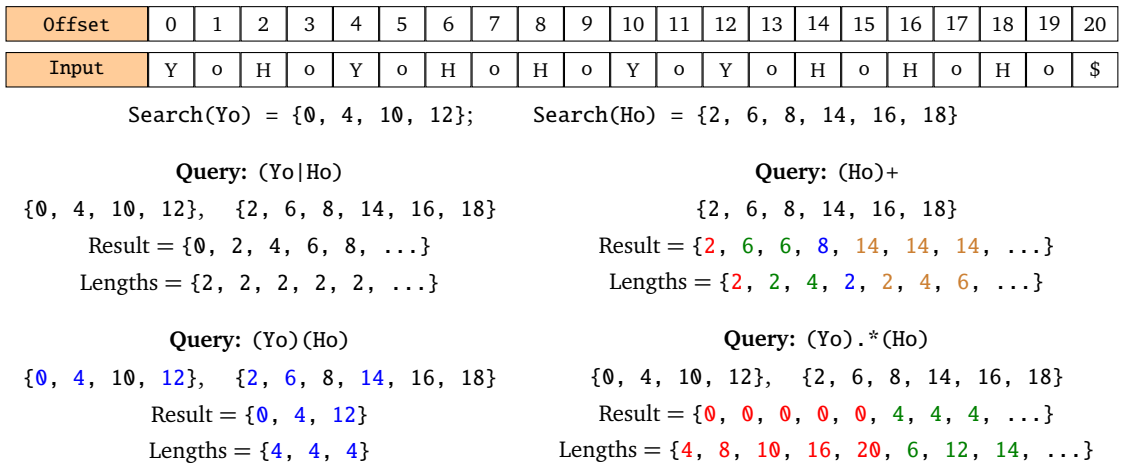


Figure 2: Illustration of the third step in black-box approach from §3 — executing algorithms in Appendix A on an example input file (the top row shows the file offsets for ease of illustration). The intermediate search results (i.e., offsets into the input file) for the 2-grams `Yo` and `Ho` are shown 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[1]
3. Traverse the tree bottom up, generating the results at each operator node using intermediate results for the left and the right subtrees. The algorithms to combine intermediate results for each operator[2] 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 the corresponding RTree (as in 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 [23] [3]. Specifically, we obtain the following result for the individual operator algorithms from Appendix A (proofs in Appendix A):

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 cardinality of the final output.*

It is known that, under the RAM computational model, the time complexity of an algorithm is lower bounded by the output size [23]. 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[4]. 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.

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

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

[3] While a standard for algorithmic analysis, the RAM computation model ignores the 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. Caching can still affect partial data scan performance, but since partial scans are performed at random locations for only a few characters, the effect on end-to-end query performance is minimal.

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

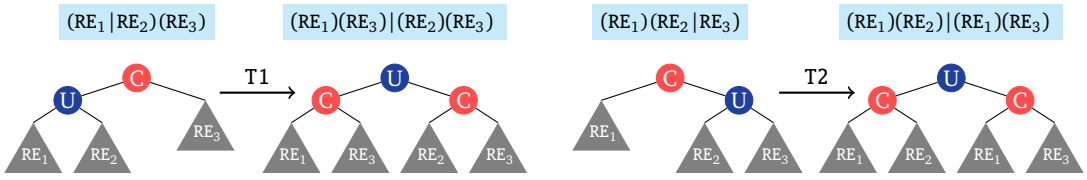


Figure 3: `Pull-Up Union` (§4.1): transformation `T1` is used if the `Union` operator is the left child, and `T2` otherwise.

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 Appendix B 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 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, searching for leaf tokens, and traversing the tree combining the intermediate results at nodes. We show in §5 that the last step is indeed the performance bottleneck (thus making Lemma 1 result more relevant). Intuitively, this is because constructing an RTree and searching for individual tokens is extremely fast when tokens are indexed. The performance of the third step, in turn, requires combining intermediate results across the operators along the RTree, which is significantly more complex than (a few) binary searches to search for tokens in the index.

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.

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
```

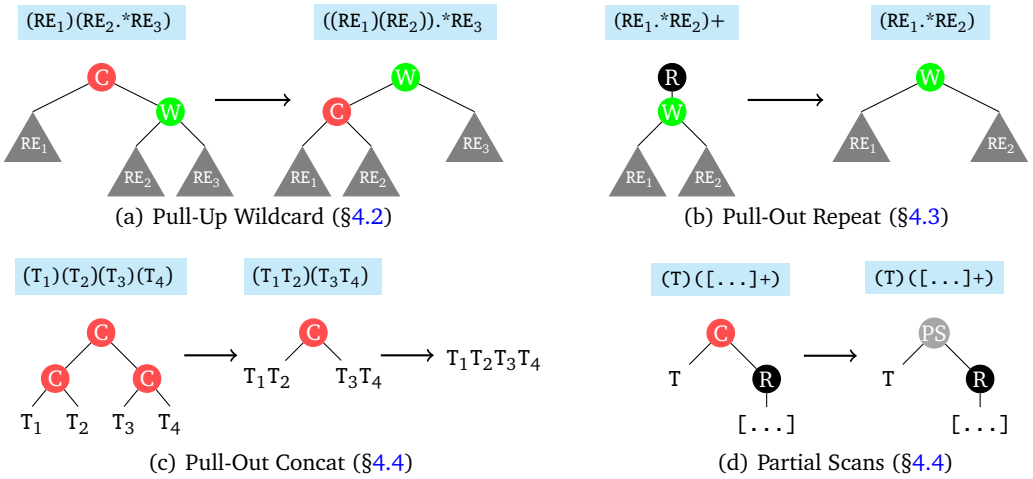
4.2 Pull-Up Wildcard

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.

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))).



(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

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 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 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 $(R_1)(R_2+)$. 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 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.

Limitations of Swift. Note that there are two cases where Swift may still execute the `Concat` operator on intermediate results. These cases are when one of the children of the `Concat` operator is a: (i) `PS` operator; and, (ii) `Repeat` operator with `Union` and/or `Concat` operators as its children.

5. EVALUATION

We now evaluate the performance of Swift against the black-box approach and 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 two applications that commonly use RegEx: bioinformatics [35, 48] and document stores [2, 50].

For the bioinformatics application, we use the standard Pfam-A Protein dataset [32], 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 [57] database (see Table 2).

For the document store application, we use a collection of 4.8 million English Wikipedia articles, constituting roughly 10GB of data. Unfortunately, there is no standard workload for RegEx

Table 2: Protein Signature RegEx queries taken from the Prosite Database [57]

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: Document analysis RegEx queries taken from [22]; \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.

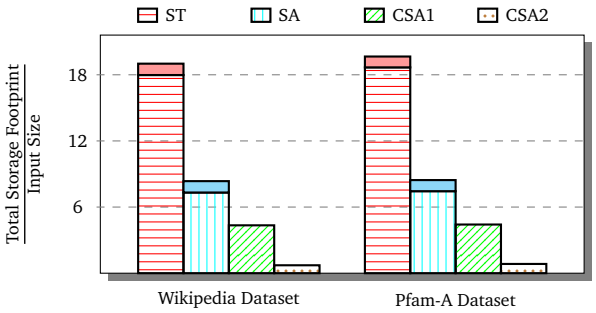


Figure 5: Storage footprint for different data structures for the Wikipedia and Pfam-A datasets. Note that ST and SA require storing the original input as well (shown as solid fill), while CSA provides similar functionality without storing the input.

queries on document stores; to that end, we ran all the queries from [22], and present results for queries that output non-zero results for Wikipedia dataset (see Table 3).

Data structures and Systems. 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) [63], Suffix Arrays with LCP (SA) [43], `k-gram` indexes, compressed suffix trees (CST), and compressed suffix arrays (CSA) [9], 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 (termed CSA1 and CSA2)[5] since these achieve strictly better space-latency tradeoff than other data structures. Figure 5 shows the storage footprint for these data structures.

Note that CSA2 acheives a storage footprint *smaller than the input itself*; consequently, our following results show that Swift enables execution of RegEx queries *directly on compressed data*.

[5]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.

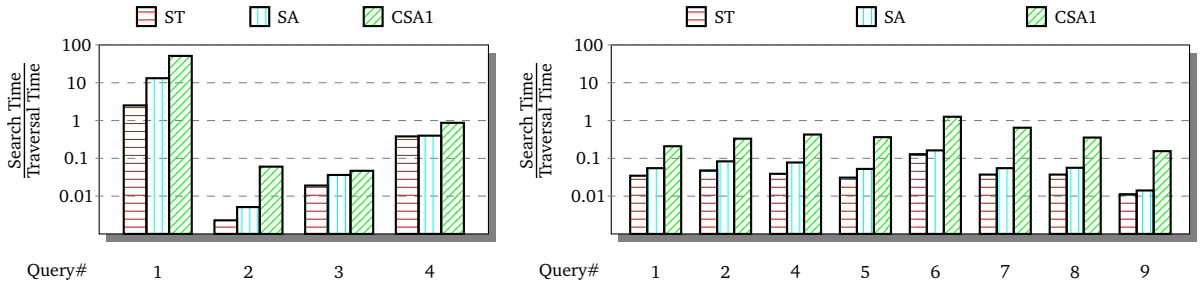
We then compare the performance of Swift against popular open-source systems that support RegEx queries — ElasticSearch [2] and MongoDB [50] for the document store application, and ScanProsite [34] for the bioinformatics application. ElasticSearch uses Lucene [45] as its underlying searching and indexing engine, and executes RegEx queries using an automaton-based approach. MongoDB indexes are not supported for documents larger than 1KB (which is the case for some of the Wikipedia articles); thus, MongoDB executes RegEx queries using full-data scans. Finally, ScanProsite is a publicly available tool for executing RegEx on protein sequences using main memory data scans.

In terms of storage overhead, ElasticSearch and MongoDB have storage footprint of roughly 1.4× the input size while ScanProsite uses storage exactly 1× the input size. The rest of the paper focuses on latency of executing RegEx, over an Amazon EC2 r3.8xlarge instance with 244GB RAM, large enough to fit each of the data structures completely in memory (for all the systems).

5.2 Comparison against Black-box

We now evaluate the performance benefits of Swift against the black-box approach. Our key observations are:

- **Figure 7 and Figure 8:** When a query comprises of `Union`, `Repeat` and `Wildcard` operators only (that execute in near-optimal time as shown in Lemma 1), Swift performance is identical to 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.
- **Figure 9:** The choice of data structure (ST, SA, CSA) has a significant impact on absolute RegEx query execution latency, both for the black-box and the Swift approach. Interestingly, the higher storage footprint often comes with the benefit of super-linear improvements in latency.
- Irrespective of the underlying data structure, character classes (and repeats of character classes) often make RegEx query execution complex and time consuming (for both the black-box and the Swift approach). Intuitively, character classes often lead to a large number of intermediate results (and hence, higher latency in combining intermediate results) or a large fraction of file scanned during partial data scans.



(a) Wikipedia Dataset (b) Pfam-A Dataset

Figure 6: Understanding Latency Bottlenecks. Ratio of search and traversal time for queries from Table 2 and Table 3. The black-box approach does not finish within 10 minutes of execution time for CSA2, for Wikipedia `Query#5` and for Pfam-A `Query#3, #10`. Besides a few exceptions (§5.2), traversal is the latency bottleneck, taking 10× more time than search — precisely the problem Swift aims to solve.

We discuss the results in depth below.

Understanding latency bottlenecks in the black-box approach. Recall from §3, the black-box approach for RegEx comprises of (i) search, which includes constructing the RTree and searching for leaves (i.e., Steps 1 and 2) and (ii) traversal, combining intermediate results while traversing the RTree (Step 3). Figure 6 shows the ratio of the search time and the traversal time for the evaluated queries. We observe that, for most queries, the traversal requires at least 10× more time than search (with two exceptions, as discussed below). The queries are, thus, bottlenecked by combining the intermediate results while traversing the RTree — precisely the problem that Swift aims to solve.

There are two exceptions when the traversal time is comparable or smaller than search time, though. First, in comparison to ST and SA, search in CSA1 requires some extra computations that are linear in cardinality of search results. It, thus, makes traversal taking 2–4× more time than search for most queries (rather than 10×). Second, for some queries (`Query#1, #4` of the Wikipedia dataset), the intermediate results after initial search (`Query#1`) or after first partial scan (`Query#4`) are very small making traversals extremely fast. Nevertheless, for most queries, the traversal takes 4–10× more time than search.

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 7 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 7 and Figure 8), 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 (more in-depth discussion on Swift performance below).

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 9 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 7 and Figure 8, 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×, 4.3× and 26.2× higher storage than SA, CSA1 and CSA2, and offers 4.7×, 10× and 13.3× 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.

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 10 shows that, for the Pfam-A dataset, 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

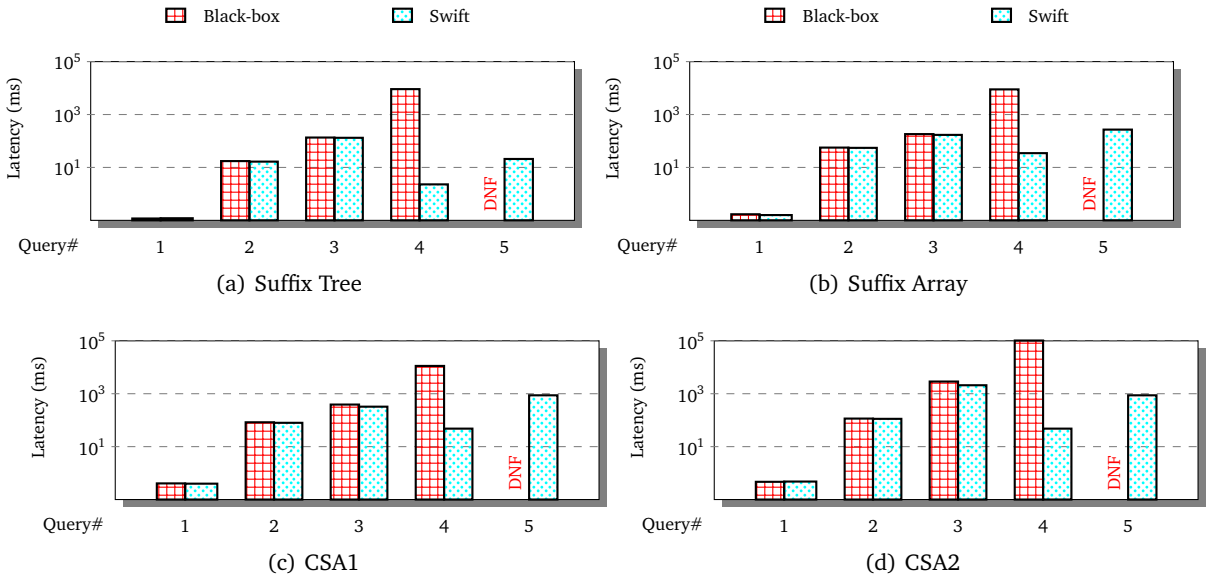


Figure 7: Black-box vs. Swift across different data structures for the Wikipedia dataset. Swift achieves significant speedups for queries where Swift transformations are applicable (`Query#4-5`); queries where the transformations are not applicable or require partial scans see performance similar to the black box approach (`Query#1-3`). Queries marked with **DNF** did not finish within 10 minutes of execution time.

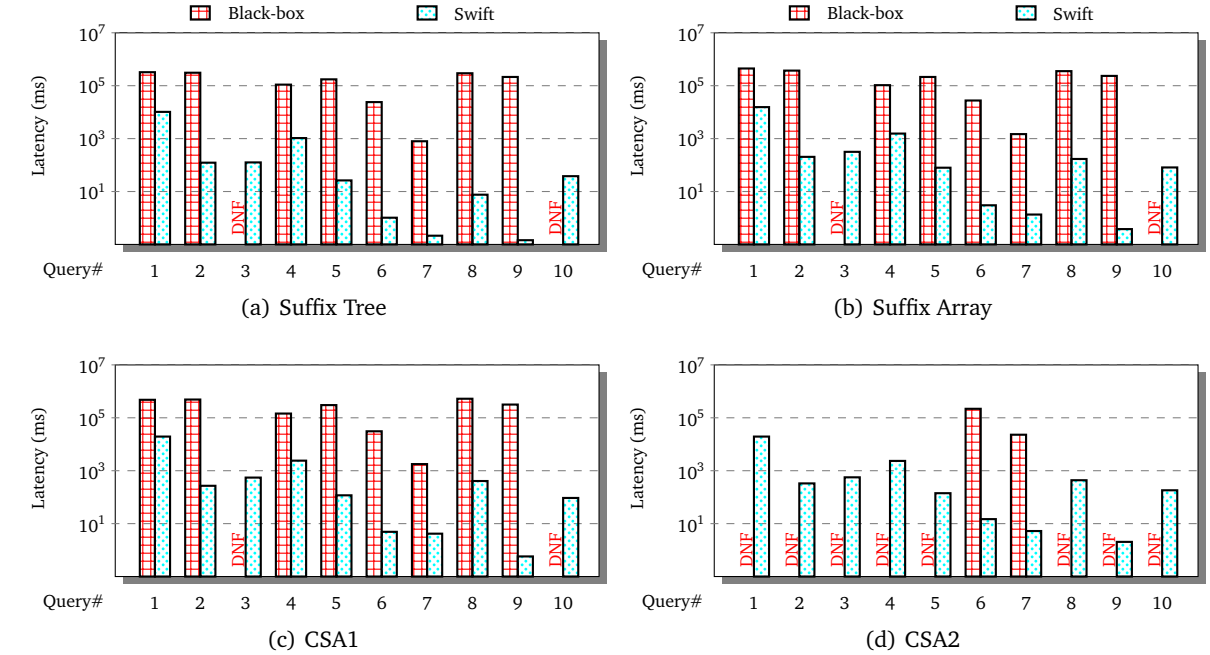
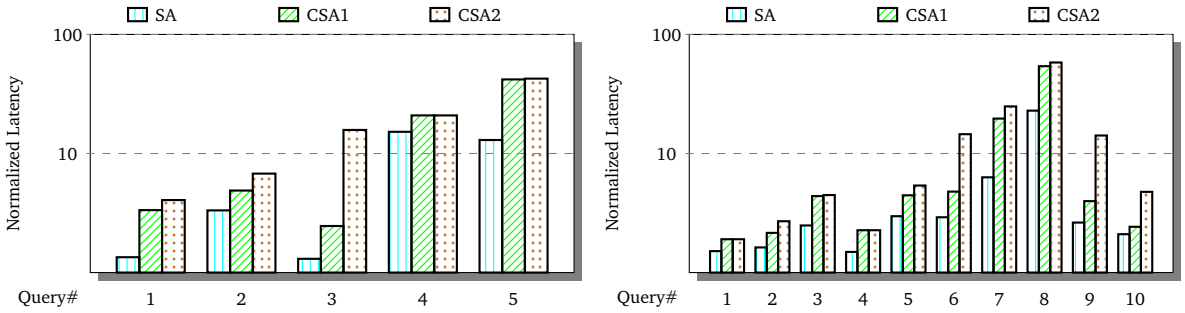


Figure 8: Black-box vs. Swift across different data structures for the Pfam-A dataset. Since Swift transformations are applicable for all queries, Swift offers significantly lower latency compared to the black box approach. Queries marked with **DNF** did not finish within 10 minutes of execution time.



(a) Wikipedia Dataset (b) Pfam-A Dataset

Figure 9: 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.

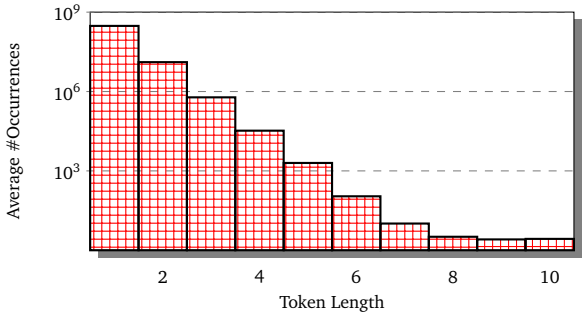


Figure 10: Why Swift works. Variation of token frequency with token length for the Pfam-A dataset — the average number of occurrences of the tokens decrease as their length is increased.

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 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.

5.3 Comparison against Existing Systems

We now evaluate the performance of Swift against popular open-source systems that support RegEx query execution. We use CSA2 as our underlying data structure for the Swift algorithms since Swift has the worst performance for this data structure among all the evaluated ones.

Document Stores. Figure 11(a) summarizes the query latency results for the compared systems. Swift executes RegEx significantly faster than other systems, with latency benefits varying from 1–3 orders of magnitude across all the evaluated queries. MongoDB scans through all of the documents to find matches to the regular expressions, 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 above systems.

Bioinformatics. The query latencies for Swift and ScanProsite are summarized in Figure 11(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 roughly same 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), enabling it to find matches for the protein signatures much faster.

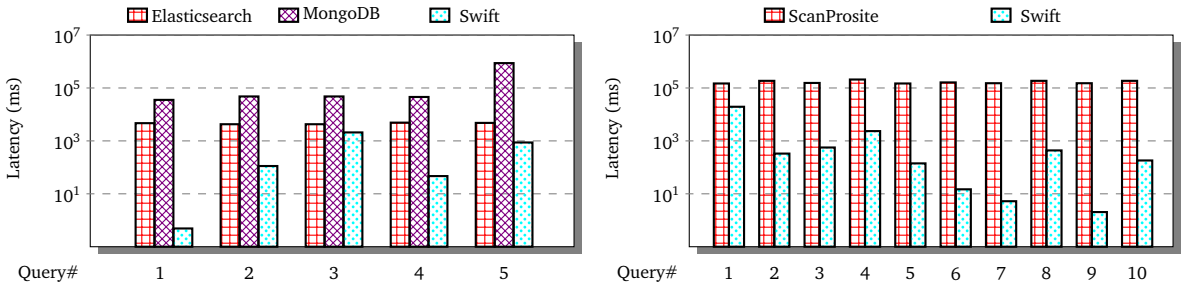
6. RELATED WORK

There are two traditional approaches to executing RegEx queries. We compare and contrast Swift against these approaches.

Index-based approaches. There are a multitude of techniques both for indexing and using indexes. On the indexing front, we note that RegEx by nature contain strings that are not linguistically meaningful, making traditional indexing techniques (*e.g.*, inverted indexes) that use English words or other linguistic constructs [24, 28, 44, 55] as keys less useful. As a result, several specialized indexes for RegEx have been designed — m-gram indexes [22, 52], full-text indexes [45, 50], and tree-based indexes [10, 11, 14, 20, 31, 39, 43, 46, 47, 62, 63], 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 [22]; or, executing the entire RegEx using indexes, potentially supported by partial data scans (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, when such is not the case (*e.g.*, all Pfam-A queries), this results in full data scans. This paper focused on the second approach, and identified the sources of inefficiencies in this approach via analyzing the performance of individual RegEx operators. The proposed Swift technique overcomes these inefficiencies using a set of transformations on the underlying RTree, achieving as much as two orders of magnitude improvements over the black-box approach.

Scan-based approaches, and why are index-based approaches not used in practice?. Most of the popular open-source data stores that support RegEx queries [2, 50] 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 [22]; and (ii) index-based techniques do not offer latency gains over data scans (even in our evaluation



(a) Document Stores (b) Bioinformatics

Figure 11: Swift executes RegEx significantly faster than popular open-source systems across various application domains.

from §5, compare results for black-box approach in Figure 7 and Figure 8 with results for scan-based approaches in Figure 11). 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 [9,38,53,54], 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 CSA1 and CSA2, resolves both the above issues with index-based approaches making them an interesting choice for executing RegEx queries.

7. CONCLUSION

In this paper, we revisit the index-based techniques to executing RegEx queries. We first analyze the performance of individual operators in the black-box approach, and show that while `Union, Repeat` and `Wildcard` operators are individually efficient, index-based techniques are particularly inefficient when the RegEx query contains `Concatenation` operator. We then proposed Swift, a set of transformations on the original RegEx query that ensures that the black-box approach can be avoided for the `Concat` operator to whatever extent possible. 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] Elasticsearch. http://www.elasticsearch.org/.

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

[4] Introducing oracle regular expressions. http://www.oracle.com/technetwork/database/focus-areas/application-development/twp-regular-expressions-133133.pdf.

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

[6] Openrefine. http://openrefine.org.

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

[8] D. J. Abadi, S. R. Madden, and M. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *ACM International Conference on Management of Data (SIGMOD)*, 2006.

[9] R. Agarwal, A. Khandelwal, and I.Stoica. Succinct: Enabling Queries on Compressed Data. In *USENIX Symposium on Network System Design and Implementation (NSDI)*, pages 337–350, 2015.

[10] Aoe, Jun-ichi and Morimoto, Katsushi and Sato, Takashi. An Efficient Implementation of Trie Structures. *Software: Practice and Experience*, pages 695–721, 1992.

[11] N. Askitis and R. Sinha. HAT-trie: A Cache-conscious Trie-based Data Structure for Strings. In *Australasian Conference on Computer Science (ACSC)*, pages 97–105. Australian Computer Society, Inc., 2007.

[12] P. Barceló, L. Libkin, and J. L. Reutter. Querying Graph Patterns. In *Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '11, pages 199–210, New York, NY, USA, 2011. ACM.

[13] P. Barceló Baeza, M. Romero, and M. Y. Vardi. Semantic acyclicity on graph databases. In *Proceedings of the 32Nd Symposium on Principles of Database Systems*, PODS '13, pages 237–248, New York, NY, USA, 2013. ACM.

[14] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indices. In *ACM-SIGMOD Workshop on Data Description, Access and Control*, pages 107–141, 1970.

[15] G. J. Bex, W. Gelade, W. Martens, and F. Neven. Simplifying XML Schema: Effortless Handling of Nondeterministic Regular Expressions. In *ACM International Conference on Management of Data (SIGMOD)*, pages 731–744, 2009.

[16] G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls. Inference of Concise DTDs from XML Data. In *International Conference on Very Large Data Bases (VLDB)*, pages 115–126, 2006.

[17] G. J. Bex, F. Neven, and S. Vansummeren. Inferring XML Schema Definitions from XML Data. In *International Conference on Very Large Data Bases (VLDB)*, pages 998–1009, 2007.

[18] P. Bohannon, N. Dalvi, Y. Filmus, N. Jacoby, S. Keerthi, and A. Kirpal. Automatic Web-scale Information Extraction. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 609–612, New York, NY, USA, 2012. ACM.

[19] F. Brauer, R. Rieger, A. Mocan, and W. M. Barczynski. Enabling Information Extraction by Inference of Regular

Expressions from Sample Entities. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 1285–1294, New York, NY, USA, 2011. ACM.

[20] C.-Y. Chan, M. Garofalakis, and R. Rastogi. RE-tree: An Efficient Index Structure for Regular Expressions. *Proceedings of the VLDB Endowment*, pages 102–119, 2003.

[21] 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 *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 1291–1294, New York, NY, USA, 2011. ACM.

[22] J. Cho and S. Rajagopalan. A Fast Regular Expression Indexing Engine. In *IEEE International Conference on Data Engineering (ICDE)*, page 419, 2001.

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

[24] D. Cutting and J. Pedersen. Optimization for Dynamic Inverted Index Maintenance. In *ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 405–411, 1990.

[25] N. Dalvi, R. Kumar, and M. Soliman. Automatic Wrappers for Large Scale Web Extraction. *Proc. VLDB Endow.*, 4(4):219–230, Jan. 2011.

[26] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Spanners: A formal framework for information extraction. In *Proceedings of the 32Nd Symposium on Principles of Database Systems*, PODS '13, pages 37–48, New York, NY, USA, 2013. ACM.

[27] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Cleaning inconsistencies in information extraction via prioritized repairs. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '14, pages 164–175, New York, NY, USA, 2014. ACM.

[28] C. Faloutsos. Access Methods for Text. *ACM Computing Surveys (CSUR)*, pages 49–74, 1985.

[29] W. Fan. Graph Pattern Matching Revised for Social Network Analysis. In *Proceedings of the 15th International Conference on Database Theory*, ICDT '12, pages 8–21, New York, NY, USA, 2012. ACM.

[30] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Adding regular expressions to graph reachability and pattern queries. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 39–50. IEEE, 2011.

[31] P. Ferragina and R. Grossi. The String B-tree: A New Data Structure for String Search in External Memory and Its Applications. *Journal of the ACM (JACM)*, pages 236–280, 1999.

[32] 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*, page gkt1223, 2013.

[33] V. Ganti and A. D. Sarma. Data cleaning: A practical perspective. *Synthesis Lectures on Data Management*, 5(3):1–85, 2013.

[34] A. Gattiker, E. Gasteiger, and A. M. Bairoch. ScanProsite: a reference implementation of a PROSITE scanning tool. *Applied Bioinformatics*, pages 107–8, 2002.

[35] Gattiker, Alexandre and Gasteiger, Elisabeth and Bairoch, Amos Marc. Scanprosite: a reference implementation of a prosite scanning tool. *Applied Bioinformatics*, pages 107–8, 2002.

[36] D. Gianfelice, L. Lesmo, M. Palmirani, D. Perlo, and D. P. Radicioni. Modificatory provisions detection: A hybrid nlp approach. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Law*, ICAIL '13, pages 43–52, New York, NY, USA, 2013. ACM.

[37] R. R. Goldberg. Finite state automata from regular expression trees. *The Computer Journal*, pages 623–630, 1993.

[38] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, pages 378–407, 2005.

[39] S. Heinz, J. Zobel, and H. E. Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Transactions on Information Systems (TOIS)*, pages 192–223, 2002.

[40] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3363–3372. ACM, 2011.

[41] R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, S. Vaithyanathan, and H. Zhu. Systemt: a system for declarative information extraction. *ACM SIGMOD Record*, 37(4):7–13, 2009.

[42] Y. Li, E. Kim, M. A. Touchette, R. Venkatachalam, and H. Wang. Vinery: A visual ide for information extraction. *Proc. VLDB Endow.*, 8(12):1948–1951, Aug. 2015.

[43] U. Manber and G. Myers. Suffix Arrays: A New Method for On-line String Searches. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 319–327, 1990.

[44] U. Manber and S. Wu. GLIMPSE: A Tool to Search Through Entire File Systems. In *USENIX Winter Technical Conference*, pages 4–4, 1994.

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

[46] E. M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM (JACM)*, pages 262–272, 1976.

[47] D. R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM (JACM)*, pages 514–534, 1968.

[48] 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)*, pages 207–212, 2006.

[49] Y. Ogawa, S. Inagaki, and K. Toyama. Automatic Consolidation of Japanese Statutes Based on Formalization of Amendment Sentences. In *Proceedings of the 2007 Conference on New Frontiers in Artificial Intelligence*, JSAI'07, pages 363–376, Berlin, Heidelberg, 2008. Springer-Verlag.

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

[51] R. Polig, K. Atasu, H. Giefers, and L. Chiticariu. Compiling text analytics queries to fpgas. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–6. IEEE, 2014.

[52] D. Robenek, J. Platos, and V. Snasel. Efficient In-memory Data Structures for n-grams Indexing. In *DATESO*, pages 48–58, 2013.

[53] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Algorithms and Computation*, pages 410–421. 2000.

[54] K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 225–232, 2002.

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

[56] 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 *Proceedings of the 2012 12th International Conference on Quality Software*, QSIC '12, pages 79–88, Washington, DC, USA, 2012. IEEE Computer Society.

[57] 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*, page gks1067, 2012.

[58] P. Spinosa, G. Giardiello, M. Cherubini, S. Marchi, G. Venturi, and S. Montemagni. Nlp-based metadata extraction for legal text consolidation. In *Proceedings of the 12th International Conference on Artificial Intelligence and Law*, ICAIL '09, pages 40–49, New York, NY, USA, 2009. ACM.

[59] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-Oriented DBMS. In *International Conference on Very Large Data Bases (VLDB)*, 2005.

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

[61] D. Tsang and S. Chawla. A Robust Index for Regular Expression Queries. In *ACM Conference on Information and Knowledge Management (CIKM)*, pages 2365–2368, 2011.

[62] E. Ukkonen. On-Line Construction of Suffix Trees. *Algorithmica*, pages 249–260, 1995.

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

APPENDIX

A. BLACK BOX ALGORITHMS

We start by describing the algorithms for combining the intermediate results (corresponding to the left and right subtree) for individual operators using the black-box approach. We first describe algorithms for 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. We then extend these algorithms to support RegEx on semi-structured data.

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

Algorithm 2 `Union`

```
1: procedure Union(L : ResultSet, R : ResultSet)
2:     return set union of L and R.
3: end procedure
```

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 sub-RegEx corresponding to results 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. Clearly. the algorithm accesses each element in L and R at most once; thus the complexity of the algorithm is $O(|L|+|R|)$.

Algorithm 3 `Concat`

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

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 $L \subseteq O$.

Algorithm 4 `Repeat`

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

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 how the above black-box algorithms are adapted to semi-structured data. We assume that indexes map tokens to `(documentID, offset)` pair, where `offset` is the starting offset of the document into a flat file containing all documents. The `(documentId, offset)` pairs 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 5 in Algorithm 3 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`.

Algorithm 5 `Wildcard`

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

Repeat. As above, Line 5 in Algorithm 4 is modified to additionally check if both `L[i].offset` and `L[j].offset` have the same `documentID`.

Wildcard. Line 10 in Algorithm 5 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 $s_0 \cdot (\log(|L|+|R|)+\log(\#documents))$.

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

$$(T)(R_1)(R_2)(R_3)...(R_k)$$

where `T` is a token, and each R_i is a character class composed of $|R_i|$ 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 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 `Unions` 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 $\prod_{i=1}^{k} |R_i|$.[6] 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

$$T_p = \prod_{i=1}^{k} |R_i| + s_0$$

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

$$T_s > T_b$$
$$\Rightarrow kf_0 > \sum_{i=1}^{k} F_i \quad (1)$$

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,

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

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

[6] 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.

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

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 |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 i[th] 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

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

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

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

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.