SuSe: Summary Selection for Regular Expression Subsequence Aggregation over Streams

Anonymized Authors

ABSTRACT

Regular expressions (RegEx) are an essential tool for pattern matching over streaming data, e.g., in network and security applications. The evaluation of RegEx queries becomes challenging, though, once subsequences are incorporated, i.e., characters in a sequence may be skipped during matching. Since the number of subsequence matches may grow exponentially in the input length, existing RegEx engines fall short in finding *all* subsequence matches, especially for queries including Kleene closure.

In this paper, we argue that common applications for RegEx queries over streams do not require the enumeration of all *distinct* matches at *any* point in time. Rather, only an aggregate over the matches is typically fetched at specific, yet unknown time points. To cater for these scenarios, we present SuSe, a novel architecture for RegEx evaluation that is based on a query-specific summary of the stream. It employs a novel data structure, coined State-Summary, to capture aggregated information about subsequence matches. This structure is maintained by a summary selector, which aims at choosing the stream projections that minimize the loss in the aggregation result over time. Experiments on real-world and synthetic data demonstrate that SuSe is both effective and efficient, with the aggregates being based on several orders of magnitude more matches compared to baseline techniques.

CCS CONCEPTS

• **Information systems → Data management systems**; **Information systems applications**.

KEYWORDS

regular expression, stream processing, stream summary

ACM Reference Format:
Anonymized Authors . 2018. SuSe: Summary Selection for Regular Expression Subsequence Aggregation over Streams. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The evaluation of regular expressions (RegEx) over streaming data is at the core of applications in various domains, such as network monitoring [19, 24, 27, 28], financial fraud detection [39, 42], or infrastructure security [8, 23]. RegEx have been studied extensively in theoretical computer science [18, 20] and automata-based approaches for their evaluation, most prominently the Thompson construction [40] and its derivates have been around for several decades.

Existing RegEx engines, such as PCRE2 [13], pcregrep [17], Oniguruma [1], Boost.Regex [25], RE2 [35] and TRE [41] are severely limited in their ability to consider subsequences during evaluation. That is, they lack native support for skipping characters in a sequence when constructing matches and do not support the enumeration of *all* matches. We note, though, that this is not a matter of expressiveness since the skipping of characters can be encoded explicitly, e.g., by transforming a regular expression $\gamma = abc$ into $\gamma' = \Sigma^* a \Sigma^* b \Sigma^* c \Sigma^*$ with Σ being the underlying alphabet. Rather, the efficiency of the evaluation of regular expressions is the limiting factor, and such transformed expressions quickly become intractable with the aforementioned engines, due to the exponential growth of the matches in the size of the input. REmatch [36] is a notable exception in that it supports the enumeration of all matches. Yet, there is still no explicit support for subsequences, so the engine suffers from performance issues, as we demonstrate later empirically.

In this paper, we argue that RegEx evaluation with subsequences may be optimized based on the following two observations:

(1) Some application scenarios demand that all matches of a regular expression are incorporated, whereas the output is given in terms of an aggregate (e.g., count or sum) computed over all matches. Such scenarios have been referred to as *event trend aggregation* and dedicated solutions that are based on incremental computation of the aggregate have been developed for it, e.g., GRETA [32] and its successors [29, 31, 33, 37].
(2) The matches of a regular expression, and aggregates over them, are not necessarily needed at any point in time. Rather, many application scenarios involve some external trigger that determines when the result is considered to be relevant. For instance, users may load an analysis report, open a dashboard, or refresh a visualization at certain, generally unknown time points.

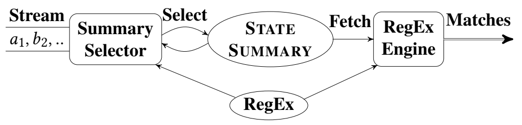


Figure 1: SuSe architecture for RegEx evaluation over streams.

Combining these two observations, we present SuSe, a novel architecture for RegEx evaluation that is based on a query-specific summary of the stream, which is illustrated in Fig. 1. Specifically, SuSe makes the following contributions:

(1) We present a model for RegEx subsequence matching with aggregations that is based on a summary of a stream. Based thereon, we formulate the problem of constructing an optimal summary that minimizes the expected information loss.
(2) We introduce StateSummary, a novel data structure to capture aggregated information about subsequence matches. We elaborate on the operations to maintain this structure.
(3) We present an approach for summary selection, which aims at constructing an optimal summary. It is based on a cost model to assess the potential benefit of stream elements.

Below, we first introduce preliminaries (§2), before giving a formal problem statement (§3). Then, we define the StateSummary (§4), as the basis for summary selection (§5). We present evaluation results (§6), demonstrating that SuSe computes aggregates based on several orders of magnitude more matches than baseline techniques. Finally, we review related work (§7) and conclude the paper (§8).

2 PRELIMINARIES

2.1 Regular Expressions

Syntax. An *alphabet* $\Sigma = \{a_1, \ldots, a_m\}$ is a totally ordered set of *characters* $a_1 < \ldots < a_m$, where $m \in \mathbb{N}^+$. A *valued character* c^v is an extension of a character, annotated with an attribute value $v \in \mathbb{N}$. Most of the techniques we introduce are based on non-valued characters; however, for certain extensions, we will refer to valued characters. A *word* is a sequence of characters $w = \langle c_1, \ldots, c_n \rangle$ of size $n = |w|$ over Σ, and each $c_i \in \Sigma$. The concatenation of words $w = \langle c_1, \ldots, c_n \rangle$ and $w' = \langle c'_1, \ldots, c'_m \rangle$ is $w.w' = \langle c_1, \ldots, c_n, c'_1, \ldots, c'_m \rangle$. A *prefix* of w is any word that starts from the first character of w and has a length of at most n.

A *subsequence* y of a word w (denoted by $y \preceq w$) is obtained by deleting zero or more characters from w without altering the order of the remaining characters. Hence, there are 2^n possible subsequences of w. Formally, a subsequence y can be represented as $y = \langle w_{i_1}, \ldots, w_{i_k} \rangle$, where $k \leq n$ and $1 \leq i_1 \leq \ldots \leq i_k \leq n$.

To establish the relationship between the positions in y and w, a *mapping function* $m : \{1, \ldots, k\} \to \{1, \ldots, n\}$ is defined such that $j \mapsto i_j$. This function maps the k positions of y to the corresponding n positions in the word w, denoted as $y \preceq_m w$. A *partial mapping* of a subsequence y' to w, denoted as $y' \preceq_{pm} w$, involves any prefix y' of y that can be mapped to w using function w.

Example 1. *Consider the alphabet $\Sigma = \{a, b, c, d\}$ and a word $w = \langle a_1, b_2, d_3, c_4, a_5, b_6, c_7, d_8 \rangle$. Here, $y = \langle a_1, b_2, d_3, c_4 \rangle$ is a subsequence with two different mappings, $y \preceq_{m_1} w$ with $m_1 : 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 4$; $y \preceq_{m_2} w$ with $m_2 : 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 7$.*

For a word w, the i-th character is denoted by $w[i]$, where $1 \leq i \leq |w|$. The set of all words of length n over an alphabet Σ is Σ^n. The set of all possible words over Σ is $\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$. A *language* L over Σ is any subset $L \subseteq \Sigma^*$. Special cases include the *empty word*, $\varepsilon = \Sigma^0$, and the *empty language*, $\emptyset$. The set of all prefixes of language L is $L_{\text{pre}} = \{w' \mid \exists\, w \in L \text{ s.t. } w' \text{ is a prefix of } w\}$.

Let A and B be languages defined over an alphabet Σ, we define concatenation, union, and Kleene star for languages, respectively:

- $AB = \{x.y \mid x \in A, y \in B\}$,
- $A \cup B = \{x \in \Sigma^* \mid x \in A \vee x \in B\}$, and
- $A^* = \bigcup_{n \geq 0} A^n$.

Semantics. We inductively define regular expression semantics. The symbols $\emptyset$, ε, and $a \in \Sigma$ are regular expressions, describing:

- the empty language $L(\emptyset) = \emptyset$;
- the language $L(\varepsilon) = \langle \varepsilon \rangle$, and
- for each character $a \in \Sigma$, the language $L(a) = \{\langle a \rangle\}$.

Let α and β be regular expressions, with languages $L(\alpha)$ and $L(\beta)$. Then, semantics of concatenation $\alpha\beta$, union $(\alpha|\beta)$, and Kleene star $(\alpha)^*$ are defined as follows:

- $L(\alpha\beta) = L(\alpha)L(\beta)$,
- $L(\alpha|\beta) = L(\alpha) \cup L(\beta)$, and
- $L((\alpha)^*) = L(\alpha)^*$.

2.2 Regular Expression Queries

Positive natural numbers represent time points, denoted as $(\mathbb{T}, \leq)$ where $\mathbb{T} \subseteq \mathbb{N}^+$. Each character c_i is associated with a timestamp $c_i.t \in \mathbb{T}$. Time progresses as ascending positive natural numbers, so the m-th character c_m will have a timestamp $c_m.t = m$.

A *regular expression query* $q = (\gamma, \tau, \mathscr{A})$ is a triple, comprising a regular expression γ (also denoted as q_γ), a time window $\tau \in \mathbb{T}$ (also q_τ), and an aggregate function $\mathscr{A}$ (also $q_{\mathscr{A}}$). The time window q_τ specifies the maximum timestamp difference between the earliest $w[1].t$ and the latest $w[n].t$ elements in a matched word w of size n, and must satisfy $q_\tau \geq w[n].t - w[1].t$.

A mapping of a subsequence $y \in L(\gamma)$ to a word $w \in \Sigma^*$ is a *match* of a regular expression query q, if y fully aligns with w and meets the time window constraint q_τ. The timestamps of its characters determine the mapping of a matched subsequence. For instance, if $q_\gamma = ab$ and $w = \langle \ldots, a_{102}, b_{103}, \ldots \rangle$, then the mapping $1 \mapsto 102$ and $2 \mapsto 103$ represents a match.

Example 2. *For $\Sigma = \{a, b, c\}$, consider a non-valued character stream $s = \langle a_1, b_2, c_3, c_4 \rangle$ and a time window $q_\tau = 2$. For $q_\gamma = abc$, two possible matches exist: $m_1 : 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3$ and $m_2 : 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 4$. The time window constraint invalidates m_2.*

Conversely, a *partial match* occurs when a subsequence $y \in L_{\text{pre}}(\gamma)$ maps to a word. Such a match is a candidate for becoming a complete match as more characters of w are examined. Notably, a complete match can also serve as a partial match. For instance, for $\gamma = a(b|c)d^*$, any match of the word $\langle a, b, d \rangle \in L(\gamma)$ also serves as a partial match of $\langle a, b, d, d \rangle \in L(\gamma)$.

Our work utilizes a streaming model of computation. Initially, the regular expression query $q = (\gamma, \tau, \mathscr{A})$ is received and preprocessed. After preprocessing, the aggregate function $\mathscr{A}$ dictates whether we receive a stream of non-valued characters $s = \langle c_1, c_2, \ldots \rangle$ or valued characters $s^v = \langle c_1^{v_1}, c_2^{v_2}, \ldots \rangle$, where $c_i \in \Sigma$ and $v_i \in \mathbb{N}$.

We focus on the aggregate functions Count, Sum, and Avg, i.e., $\mathscr{A} \in \{\text{Count}, \text{Sum}, \text{Avg}\}$. These functions are defined over the matches of a regular expression query q. For Count, the aggregate calculates the number of matches. Sum and Avg apply only to a stream of valued characters; the former aggregates the payload values of all matches, and the latter is the ratio of Sum to Count.

2.3 Regular Expression Query Matching

Automata-Based Matching. Traditionally, to match a regular expression q_γ against a stream (word), an NFA is obtained by the *Thompson construction* [40], which accepts all words in $L(\gamma)$.

Table 1: Overview of notations for regular expressions.

Notation	Explanation
$\Sigma = \{a_1, \ldots, a_m\}$	Alphabet with $a_1 < \ldots < a_m$ characters.
c	A *non valued* character $c \in \Sigma$.
c^v	A *valued* character $c \in \Sigma$ with value $v \in \mathbb{N}$.
$w = \langle c_1, \ldots, c_n \rangle$	Word (sequence of characters) of size $n = \|w\|$.
$w[i], w[i].t$	The i-the character of w and its timestamp t
$y \preceq_m x$	A subsequence y with a mapping m onto the string x.
$q = (\gamma, \tau, \mathscr{A})$	Regular expression query q, composed of regular expression γ, a time window size τ, and an aggregate function $\mathscr{A}$.
$s = \langle c_1, c_2, \ldots \rangle$	A *stream* of *non valued* characters
$s^v = \langle c_1^{v_1}, c_2^{v_2}, \ldots \rangle$	A *stream* of *valued* characters
$L(\gamma), L_{\text{pre}}(\gamma)$	The language of *all words*, the language of *all prefixes* of γ.
PM_w^q, CM_w^q	The sets of partial and complete matches respectively, resulting from evaluating q over $w \in \Sigma^*$.

An NFA is a tuple $\mathcal{N} = (Z, \Sigma, \delta, z_0, F)$, with Z as a finite set of states, Σ as the alphabet, $z_0 \in Z$ as the initial state, $\delta : Z \times \Sigma \to 2^Z$ as the transition function, and $F \subseteq Z$ as the set of final states.

Our objective goes beyond the traditional REGULAR EXPRESSION MEMBERSHIP problem [11], where the task is to decide if a word $w \in \Sigma^*$ is accepted, i.e., $w \in L(\mathcal{N})$. We aim to detect *all possible (partial) matches of subsequences from $L(\gamma)$ to positions in w*. To this end, a simple automata-based method works as follows: Each incoming character is checked against the current set of automata. If the character enables a transition for an automaton, it is duplicated, with one copy disregarding the transition and the other one taking it. Such a strategy yields an exponential growth of the number of automata in the number of processed characters.

Regular Expression Subsequence Matching. To formalize the evaluation of a regular expression query $q = (\gamma, \tau, \mathscr{A})$, only q_γ and q_τ are required, as the aggregate function $\mathscr{A}$ is applied to the resulting matches. We capture the matches resulting from evaluating q over a word $w \in \Sigma^*$ using two sets: partial matches PM_w^q and complete matches CM_w^q. These sets are based on the language $L(q_\gamma)$ and its prefix language $L_{\text{pre}}(q_\gamma)$. PM_w^q includes a partial match $y \preceq_{\text{pm}} w$, if it maps a subsequence $y \in L_{\text{pre}}(q_\gamma)$ to positions in w. CM_w^q includes a match $y \preceq_m w$, if it maps a subsequence $y \in L(q_\gamma)$ to positions in w. Either way, the mapping must satisfy the time window q_τ.

Both sets are constructed recursively, providing a framework for evaluating q over w. We illustrate the construction for CM_w^q based on $L(q_\gamma)$, while PM_w^q is defined analogously based on $L_{\text{pre}}(q_\gamma)$.
Base Case: For a character $c \in \Sigma$ and a query $q = (\gamma, \tau, \mathscr{A})$:

$$CM_w^{q_\gamma} = \{c \mid c \preceq_m w \text{ exists } \wedge c \in L(q_\gamma) \wedge q_\tau > 0\}.$$

Concatenation: For two regular expressions q_{γ_1} and q_{γ_2}, and two words $w_1 \in CM_w^{q_{\gamma_1}}$, $w_2 \in CM_w^{q_{\gamma_2}}$:

$$CM_w^{(q_{\gamma_1} q_{\gamma_2})} = \{w_1.w_2 \mid w_1.w_2 \preceq_m w \text{ exists} \wedge w_1.w_2 \in L(q_{\gamma_1} q_{\gamma_2}) \wedge q_\tau \geq w_2[n].t - w_1[1].t\}.$$

Union: For two regular expressions q_{γ_1} and q_{γ_2}:

$$CM_w^{(q_{\gamma_1} | q_{\gamma_2})} = \{w' \mid w' \in CM_w^{q_{\gamma_1}} \vee w' \in CM_w^{q_{\gamma_2}}\}.$$

Kleene Star: For q_{γ^*} and words $w_1, \ldots, w_{k-1} \in CM_w^{q_{\gamma^*}}$:

$$CM_w^{q_{\gamma^*}} = \{w_1. \ \ldots \ .w_k \mid k \geq 0, w_k \in CM_w^{q_\gamma}, w_1. \ \ldots \ .w_k \preceq_m w \text{ exists} \wedge \ w_1. \ \ldots \ .w_k \in L(q_{\gamma^*}) \wedge q_\tau \geq w_k[n].t - w_1[1].t\}.$$

EXAMPLE 3. *Consider an alphabet $\Sigma = \{a, b, c\}$ and a regular expression query $q = (\gamma, \tau, \mathscr{A})$ with $q_\gamma = abc$ and $q_\tau = 5$. The language generated by q_γ is $L(q_\gamma) = \{\langle a, b, c \rangle\}$, and its prefix language is $L_{pre}(q_\gamma) = \{\langle a \rangle, \langle ab \rangle\}$. For the word $w = \langle a_1, c_2, b_3, a_4, b_5, c_6, c_7 \rangle$, six potential matches exist: $\{\langle a_1, b_3, c_6 \rangle, \langle a_1, b_3, c_7 \rangle, \langle a_1, b_5, c_6 \rangle, \langle a_1, b_5, c_7 \rangle, \langle a_4, b_5, c_6 \rangle, \langle a_4, b_5, c_7 \rangle\}$. Incorporating the time window, we arrive at $CM_w^q = \{\langle a_1, b_3, c_6 \rangle, \langle a_1, b_5, c_6 \rangle, \langle a_4, b_5, c_6 \rangle, \langle a_4, b_5, c_7 \rangle\}$.*

Finally, using the set of complete matches as CM_w^q, we integrate the aggregate function $\mathscr{A}$. With $q_\mathscr{A}$ = COUNT, the result is $|CM_w^q|$, i.e., the total number of complete matches. If $q_\mathscr{A}$ = SUM, the result is $\sum_{m \in CM_w^q} \sum_{a_i^v \in m} v$, summing all attribute values of complete matches. For $q_\mathscr{A}$ = AVG, the result is $\text{SUM}(CM_w^q)/\text{COUNT}(CM_w^q)$.

EXAMPLE 4. *Consider for Example 3 a valued stream $s^v = \langle a_1^2, c_2^3, b_3^5, a_4^7, b_5^{11}, c_6^{13}, c_7^{17} \rangle$, resulting in $CM_w^q = \{\langle a_1^2, b_3^5, c_6^{13} \rangle, \langle a_1^2, b_5^{11}, c_6^{13} \rangle, \langle a_4^7, b_5^{11}, c_6^{13} \rangle, \langle a_4^7, b_5^{11}, c_7^{17} \rangle\}$. For* COUNT, *we get 4 and for* SUM, *we get $(2+5+13)+(2+11+13)+(7+11+13)+(7+11+17) = 112$.*

3 PROBLEM STATEMENT

Using the above model, we formulate the problem addressed in our work. We consider scenarios, in which the query evaluation is relevant at a random time instance (e.g., when a user refreshes a report). Let $U : \Omega \to \mathbb{T}$ be a random variable representing this trigger, with Ω as the sample space encompassing all potential trigger instances. The probability distribution of U is denoted by $P_U(t)$, specifying the probability that U equals t. Let $\mathcal{E} \subseteq \mathbb{T}$ be the set of *evaluation timestamps* representing evaluation triggers, drawn according to the probability distribution $P_U(t)$ over $\mathbb{T}$.

Given a stream of valued characters $s^v = \langle c_1^{v_1}, c_2^{v_2}, \ldots \rangle$ over Σ, we define a summary representation function $r_\mathcal{S} : \mathbb{T} \to \cup_{i=0}^n \Sigma^i$. For the stream (defined as a word over Σ^*) and a specific time instance t, this function provides a projection (subset) of the stream that consists of up to n characters out of the t characters that appeared. This projection represents the summary as a word over all possible subsets up to size n of the t processed characters.

Using our definitions for computing complete matches and aggregates, we aim to assess how the projected characters in a summary affect a loss function $l_\mathscr{A} : \mathbb{N} \times \mathbb{N} \to \mathbb{R}$ upon an evaluation trigger. The loss function l quantifies the discrepancy between the value of the aggregate function $q_\mathscr{A}$ applied to matches derived from the query q_γ and time window q_τ from evaluating the stream $s(t)$ (i.e., x_{opt}) and its projection $r_\mathcal{S}(t)$ (i.e., x) at that specific time t.

For $l_{\text{COUNT}}(x, x_{\text{opt}})$ and $l_{\text{SUM}}(x, x_{\text{opt}})$, we employ the loss function $l(x, x_{\text{opt}}) := |x - x_{\text{opt}}|$. For AVG, we define the loss as the relative error $l_{\text{AVG}}(x, x_{\text{opt}}) := |\frac{x - x_{\text{opt}}}{x_{\text{opt}}}|$. For each evaluation timestamp $t \in \mathcal{E} \cap \mathbb{T}$, we aim to minimize the loss function $l_\mathscr{A}$.

PROBLEM 1. *Let $\Sigma = \{c_1, \ldots, c_m\}$ be an alphabet and $s = \langle c_1^{v_1}, c_2^{v_2}, \ldots \rangle$ a stream of valued characters. Also, let $q = (\gamma, \tau, \mathscr{A})$ be a regular expression query, $r_\mathcal{S}(t)$ be a summary representation function, $\mathcal{E} \subseteq \mathbb{T}$ a finite set of evaluation timestamps drawn from a probability distribution P_U. Our objective is to find, for each time point $t \in \mathcal{E}$, the stream projection $r_\mathcal{S}(t)$ that minimizes the loss:*

$$\min_{r_\mathcal{S}(t)} (l_\mathscr{A}(q_\mathscr{A}(CM_{s(t)}^q), q_\mathscr{A}(CM_{r_\mathcal{S}(t)}^q)))$$

$$\textit{subject to: } |r_\mathcal{S}(t)| \leq n, \ \forall \ t \in \mathcal{E}$$

4 STATE SUMMARY DATA STRUCTURE

The traditional continuous evaluation of regular expression queries over streams poses disadvantages for the problem introduced in §3. The number of automata that need to be maintained may grow exponentially in the number of processed characters, yielding an exponential evaluation algorithm runtime. Yet, many matches are materialized only to be discarded before the occurrence of an evaluation trigger, which wastes computational resources.

As part of the SuSe framework, we approach the evaluation of regular expression queries with a stream summary, as illustrated already in Fig. 1. That is, the summary captures aggregated information about the stream, which, upon the occurrence of an evaluation trigger, enables the (approximate) computation of the aggregate over the complete matches of the query.

The core of the SuSe framework is a stream summary and the StateSummary data structure. Let $s^v = \langle c_1^{v_1}, c_2^{v_2}, \dots \rangle$ be a valued (or non-valued) stream. Then, the summary $\mathcal{S} = \langle c_k^{v_k}, c_l^{v_l}, \dots, c_m^{v_m} \rangle$ is a subsequence of the stream, i.e., an order-preserving projection. Given a regular expression query q, the StateSummary stores (i) aggregated information about partial and complete matches of q for the characters in $\mathcal{S}$; and (ii) information on the contribution of each character in $\mathcal{S}$ to these matches. Moreover, this information is kept for an *active time window* that denotes a temporal context.

To simplify the presentation, we first introduce the StateSummary data structure for queries involving Count as the aggregation function. To this end, we introduce its construction (§4.1), before turning to the operations to maintain it (§4.2). Then, we discuss how it is adapted for Sum and Avg aggregate functions (§4.3).

4.1 StateSummary Compilation

Given a regular expression query $q = (\gamma, \tau, \mathscr{A})$, the StateSummary is constructed based on the NFA obtained by the Thompson construction [40]. The NFA is further adapted by removing epsilon transitions, subsequent removal of unreachable states, and merging of states (which are final or non-final) with identical transitions. The resulting NFA, denoted as $\mathcal{N} = (Z, \Sigma, \delta, q_0, F)$, is used to compile our StateSummary for the selected characters $\mathcal{S}$. The data structure includes state counters (summarized in Table 2) that are defined and derived as follows.

Global State Counter. For each state $z_i \in Z$, where $1 \le i \le |Z|$, the *global state counter* $\#z_i \in \mathbb{N}$ keeps track of the number of (partial) matches in that state z_i, considering all selected characters $\mathcal{S}$. These counters are initialized to zero and can also be seen as a vector $\#v \in \mathbb{N}^{|Z|}$ with $\#v_i = 0$ for $1 \le i \le |Z|$. Let $\mathcal{S}_{\#Z} = \{\#z_0, \dots, \#z_{|Z|}\}$ denote the set of *global state counters* of $\mathcal{S}$, resulting from compiling $\mathcal{N}$ and let $\mu : \mathcal{S}_{\#Z} \rightarrow Z$ denote a bijection. The set of *partial match counters* is defined as $\#Z_{PM} = \{\#z_i \in \mathcal{S}_{\#Z} \mid \mu(\#z_i) \in Z \setminus F\}$. It contains counters where the associated state z_i in the automaton is not a final state. Conversely, the *complete match counters* $\#Z_{CM} = \{\#z_i \in \mathcal{S}_{\#Z} \mid \mu(\#z_i) \in F\}$ are capturing the counters of final states.

Count Rules. The rules specify the number of (partial) matches in a given state, from evaluating the query q over the selected characters $\mathcal{S}$. We derive them from the transitions of $\mathcal{N}$ by checking, for each state $z_i \in Z$ and each character $c \in \Sigma$, if there is a transition leading to any state in Z. As an illustration, consider the NFA in Fig. 2, where $\delta(z_1, C) = \{z_1\}$, while $\delta(z_1, A) = \emptyset$.

Table 2: Overview of different counter concepts.

Notation	Explanation
$\mathcal{S}_{\#Z}$	**Global state counters**: counting *all* (partial) matches in $\mathcal{S}$.
$c_{\#Z}$	**Local state counters**: counting (partial) matches c is involved.
$\mathscr{A}_{\#Z}$	**Active global state counters**: counting *all* (partial) matches satisfying q_τ, thus having the potential to lead to future matches.
$\mathscr{A}_{c_{\#Z}}$	**Active local state counters**: counting (partial) matches c is involved satisfying q_τ, thus having the potential to lead to future matches.

Upon encountering a streamed character c, any (partial) match in a state z_i satisfying $\delta(z_i, c) \neq \emptyset$ can transition using c. Through matching, additional (partial) matches in the resulting state(s) are created, as each partial match in z_i transitions using c, causing the initial count of partial matches in the subsequent state(s), given by $\delta(z_i, c)$, to increment by the current count z_i. Transitions from the initial state $z_0 \in Z$ to a state $z_k \neq z_0$ are special, as they represent the creation of a new (partial) match, adding one to z_k.

Based on $\mathcal{N}$'s transitions, we define Count rules to quantify the number of (partial) matches within a state resulting from matching an element against the current state. For each state $z_i \in Z$, global state counter $\#z_i \in \mathcal{S}_{\#Z}$, and character $c \in \Sigma$:

(1) If z_i has a self-loop with c:

$$\text{if } \delta(z_i, c) = \{z_i\} \text{ then } \#z_i \longleftarrow \#z_i + \#z_i$$

(2) If transitions from states $\{z_{i_1}, z_{i_2}, \dots, z_{i_l}\}$ lead to z_k upon processing character c (excluding the self-loop case):

$$\text{if } \delta(z_{i_j}, c) = \{z_k\} \text{ for } j \in \{1, 2, ..., l\} \text{ then } \#z_k \longleftarrow \#z_k + \textstyle\sum_{j=1}^{l} \#z_{i_j}$$

(3) If the transition goes from the initial state z_0 to $z_k \neq z_0$:

$$\text{if } \delta(z_0, c) = \{z_k\} \text{ then } \#z_k \longleftarrow \#z_k + 1$$

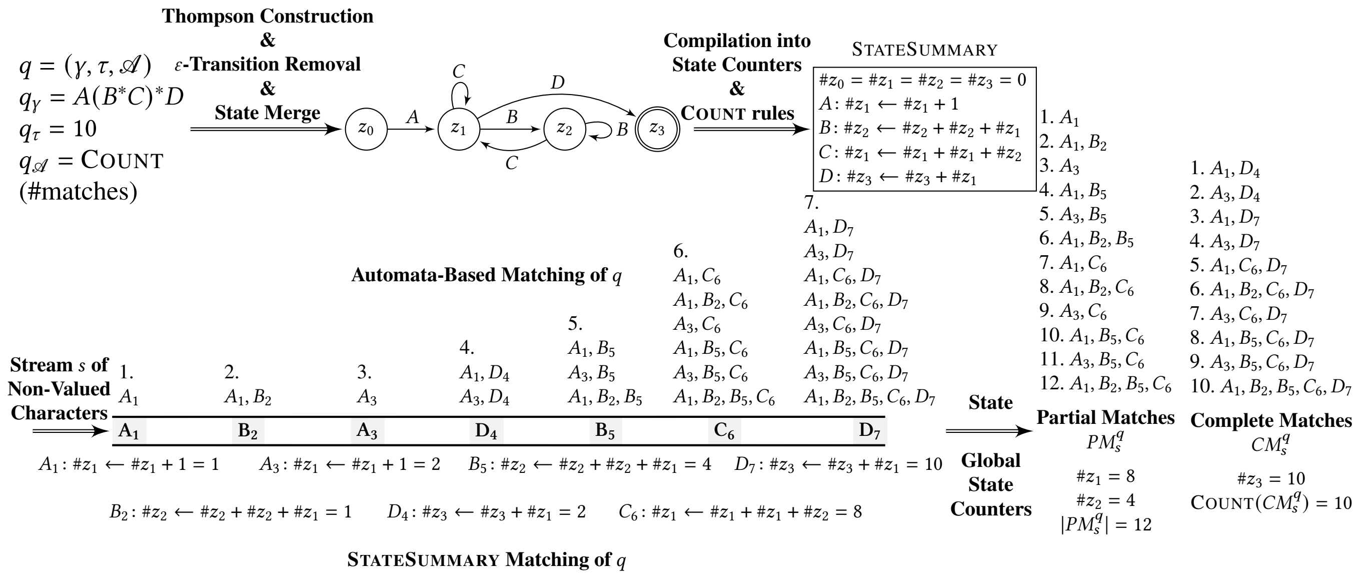
Note that for counters impacted by both, a self-loop and a transition from a different state, the number must first be doubled by the self-loop. This way, we ensure that no (partial) match undergoes two transitions, maintaining the correctness of the counter.

Since δ yields a subset of 2^Z, a character c may update multiple state counters, with $O(|\mathcal{S}_{\#Z}|) \in O(|Z|)$ as an upper bound.

Example 5. *Consider the stream* $s = \langle A_1, B_2, A_3, D_4, B_5, C_6, D_7 \rangle$ *in Fig. 2, which is a word* $w \in \Sigma^*$. *The expression* $\gamma = A(B^*C)^*D$ *yields global state counters* $\mathcal{S}_{\#Z} = \{\#z_0, \#z_1, \#z_2, \#z_3\}$, *each initialized to* 0. *The* Count *rules are depicted in the box in Fig. 2.*

Processing s character-by-character, A_1 *increments* $\#z_1$ *by one. Then,* B_2 *increases* $\#z_2$ *by the sum of* $\#z_1$ *and* $\#z_2$*;* A_3 *adds one to* $\#z_1$*; and* D_4 *augments* $\#z_3$ *to* 2. *In summary, after processing s, we have accumulated* 12 *partial matches and* 10 *complete matches.*

Local State Counter. We introduce the concept of *local state counters* to enrich the information we gather per state. For each character c considered in the summary, these counters, denoted as $c_{\#Z}$, represent the number of partial and complete matches in which c participates. Those counters are constructed only for states, for which the respective matches may include the character c. The principle behind local state counters mirrors the one behind global state counters. When the character c arrives, the local state counters reflect the (partial) matches that c contributes to the global state counters. The counters $c_{\#Z}$ are updated using the same Count rules as introduced above for global state counters (but applied to $c_{\#Z}$).


$q = (\gamma, \tau, \mathscr{A})$
$q_\gamma = A(B^*C)^*D$
$q_\tau = 10$
$q_\mathscr{A} = $ COUNT (#matches)
Thompson Construction & ε-Transition Removal & State Merge
Compilation into State Counters & COUNT rules
STATESUMMARY
$\#z_0 = \#z_1 = \#z_2 = \#z_3 = 0$
$A: \#z_1 \leftarrow \#z_1 + 1$
$B: \#z_2 \leftarrow \#z_2 + \#z_2 + \#z_1$
$C: \#z_1 \leftarrow \#z_1 + \#z_1 + \#z_2$
$D: \#z_3 \leftarrow \#z_3 + \#z_1$
Automata-Based Matching of q
Stream s of Non-Valued Characters
A_1 B_2 A_3 D_4 B_5 C_6 D_7
State
Partial Matches PM_s^q
Complete Matches CM_s^q
Global State Counters
$\#z_1 = 8$
$\#z_2 = 4$
$|PM_s^q| = 12$
$\#z_3 = 10$
COUNT$(CM_s^q) = 10$
$A_1: \#z_1 \leftarrow \#z_1 + 1 = 1$
$B_2: \#z_2 \leftarrow \#z_2 + \#z_2 + \#z_1 = 1$
$A_3: \#z_1 \leftarrow \#z_1 + 1 = 2$
$D_4: \#z_3 \leftarrow \#z_3 + \#z_1 = 2$
$B_5: \#z_2 \leftarrow \#z_2 + \#z_2 + \#z_1 = 4$
$C_6: \#z_1 \leftarrow \#z_1 + \#z_1 + \#z_2 = 8$
$D_7: \#z_3 \leftarrow \#z_3 + \#z_1 = 10$
STATESUMMARY Matching of q

1.	2.	3.	4.	5.	6.	7.
A_1	A_1, B_2	A_3	A_1, D_4	A_1, B_5	A_1, C_6	A_1, D_7
			A_3, D_4	A_3, B_5	A_1, B_2, C_6	A_3, D_7
				A_1, B_2, B_5	A_3, C_6	A_1, C_6, D_7
					A_1, B_5, C_6	A_1, B_2, C_6, D_7
					A_3, B_5, C_6	A_3, C_6, D_7
					A_1, B_2, B_5, C_6	A_1, B_5, C_6, D_7
						A_3, B_5, C_6, D_7
						A_1, B_2, B_5, C_6, D_7

Partial Matches PM_s^q	Complete Matches CM_s^q
1. A_1	1. A_1, D_4
2. A_1, B_2	2. A_3, D_4
3. A_3	3. A_1, D_7
4. A_1, B_5	4. A_3, D_7
5. A_3, B_5	5. A_1, C_6, D_7
6. A_1, B_2, B_5	6. A_1, B_2, C_6, D_7
7. A_1, C_6	7. A_3, C_6, D_7
8. A_1, B_2, C_6	8. A_1, B_5, C_6, D_7
9. A_3, C_6	9. A_3, B_5, C_6, D_7
10. A_1, B_5, C_6	10. A_1, B_2, B_5, C_6, D_7
11. A_3, B_5, C_6	
12. A_1, B_2, B_5, C_6	

Figure 2: A non-valued character stream and the NFA for the regular expression $q_\gamma = A(B^*C)^*D$. The counters of the STATESUMMARY are contrasted with the partial matches constructed by traditional automata-based matching.

EXAMPLE 6. *Consider again Fig. 2 and character B_5. Upon its arrival, B_5 is incorporated by all partial matches in states z_1 and z_2, producing partial matches A_1, B_5, A_3, B_5, and A_1, B_2, B_5. Hence, we have $B_{5\#z_2} \leftarrow \#z_2 + \#z_1 = 3$. Note that only when B_5 is added, we apply the global state counters $\#z_i \in \mathcal{S}_{\#Z}$ to the local state counters. Afterwards, the* COUNT *rules update B_5's local state counters by factoring in the corresponding local state counter values. Therefore, when C_6 is processed next, the local state counter for B_5 gets updated: $B_{5\#z_1} \leftarrow B_{5\#z_1} + B_{5_{\#z_1}} + B_{5\#z_2} = 3$. Finally, with the arrival of D_7, B_5 is part of* 6 *partial matches and* 3 *complete matches.*

If each state is reachable from any other state, for each character c, we must monitor all $|Z|$ counters, which yields a space complexity for both types of counters of $O(|Z| + |\mathcal{S}| \cdot |Z|)$, which simplifies to $O(|\mathcal{S}| \cdot |Z|)$, where typically $|Z| \ll |\mathcal{S}|$.

Sliding Time Window. The time window q_τ of a regular expression query constrains matches based on the timestamps of characters. If a character c is added to the summary, it is possible that not all its elements fall within the same time window as c. Hence, the global state counters $\mathcal{S}_{\#Z}$ may not represent the correct values for initializing the local state counters $c_{\#Z}$ for character c. We now detail the extensions to STATESUMMARY for such temporal context.

We define the *active time window* $\mathcal{A}$ as the subsequence of the selected characters $\mathcal{S}$ that, upon adding a character c, satisfy the time window q_τ with c. Based thereon, we maintain separate *active* global and local state counters, denoted by $\mathcal{A}_{\#Z}$ and $\mathcal{A}_{c_{\#Z}}$, respectively. They reflect the match counts per character in $\mathcal{A}$ and are updates based on the same COUNT rules as before. Within $\mathcal{A}$, the term $\mathcal{A}_{init}$ denotes the oldest initiator element, i.e., the oldest character selected from the stream that aligns with a transition in the automaton starting in the initial state z_0. Note that there is no need to keep characters before this element that do not align with transitions from the initial state since they cannot contribute to new (partial) matches.

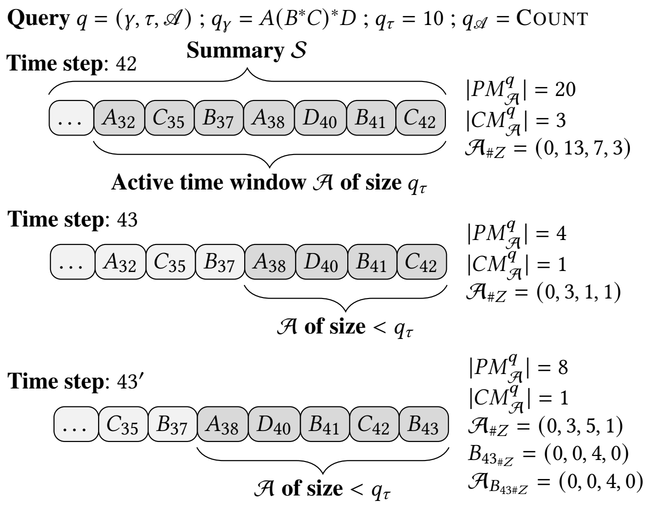

Query $q = (\gamma, \tau, \mathscr{A})$; $q_\gamma = A(B^*C)^*D$; $q_\tau = 10$; $q_\mathscr{A} = $ COUNT
Time step: 42
Summary $\mathcal{S}$
$\dots$ A_{32} C_{35} B_{37} A_{38} D_{40} B_{41} C_{42}
$|PM_\mathcal{A}^q| = 20$
$|CM_\mathcal{A}^q| = 3$
$\mathcal{A}_{\#Z} = (0, 13, 7, 3)$
Active time window $\mathcal{A}$ of size q_τ
Time step: 43
$\dots$ A_{32} C_{35} B_{37} A_{38} D_{40} B_{41} C_{42}
$|PM_\mathcal{A}^q| = 4$
$|CM_\mathcal{A}^q| = 1$
$\mathcal{A}_{\#Z} = (0, 3, 1, 1)$
$\mathcal{A}$ of size $< q_\tau$
Time step: 43′
$\dots$ C_{35} B_{37} A_{38} D_{40} B_{41} C_{42} B_{43}
$|PM_\mathcal{A}^q| = 8$
$|CM_\mathcal{A}^q| = 1$
$\mathcal{A}_{\#Z} = (0, 3, 5, 1)$
$B_{43_{\#Z}} = (0, 0, 4, 0)$
$\mathcal{A}_{B_{43\#Z}} = (0, 0, 4, 0)$
$\mathcal{A}$ of size $< q_\tau$

Figure 3: Evolution of a summary for the given regular expression query; shaded elements are in the active time window $\mathcal{A}$.

EXAMPLE 7. *Consider the summary at time* 42 *in Fig. 3 and the query q from Fig. 2. The summary considers the selected events $\mathcal{S}$, with C_{42} being the most recent selected character. Owing to q_τ, A_{32} remains the earliest element that still matches C_{42}. Character $\mathcal{A}_{init} = A_{32}$ remains in $\mathcal{A}$, as it satisfies the time window. Yet, at time* 43*, it is no longer valid and hence is removed from $\mathcal{A}$, but it remains in $\mathcal{S}$. Both C_{35} and B_{37} fit within the time window. Since they are no initiators, they are discarded from $\mathcal{A}$ and we set $\mathcal{A}_{init} = A_{38}$.*

As we add B_{43}, its local state counters $B_{43\#Z}$ are initialized from the active time window state counters $\mathcal{A}_{\#Z} = (1, 3, 1, 1)$, so that we arrive at $B_{43\#Z} = (0, 0, \mathcal{A}_{\#z_1} + \mathcal{A}_{\#z_2}, 0) = (0, 0, 4, 0)$. These counts represent the current number of (partial) matches. Then, we update the global state counter values by adding the contributions from B_{43}: $\mathcal{S}_{\#Z} \leftarrow \mathcal{S}_{\#Z} + B_{43\#Z}$, i.e., we add the respective state counter vector.

In addition, we maintain separate *active local state counters* $\mathcal{A}_{c_{\#Z}}$ for characters c in $\mathcal{A}$. These counters track the number of matches each element participates in within $\mathcal{A}$ (whereas the local state counters $c_{\#Z}$ of c also consider characters no longer in $\mathcal{A}$). This nuanced tracking is crucial to later estimate the benefit of an character in $\mathcal{A}$ to yield future matches.

As soon as a character c drops out of $\mathcal{A}$, its local state counters $c_{\#Z}$ can no longer increase, but only decrease. The reason being that c can no longer lead to (partial) matches.

EXAMPLE 8. *Refer to Fig. 4 at time* 43. *Even after the removal of* A_{32} *from* $\mathcal{A}$*, matches with elements like* B_{41} *remain. Yet, future matches between* A_{32} *and* B_{41} *are no longer possible. Hence, the local state counters of* B_{41} *may still include counts involving* A_{32}*, whereas this is not true for its active local state counters.*

Maintaining the active global and local state counters results in additional space requirements of size $O(|\mathcal{A}| \cdot |Z|)$. Since $|\mathcal{A}|$ depends on q_τ and $|\mathcal{S}|$, we arrive at $O(min\{q_\tau, |\mathcal{S}|\} \cdot |Z|)$.

4.2 Operations

In this section, we detail the INITIALIZE, INSERT, and REMOVE operations that manipulate the summary $\mathcal{S}$.

INITIALIZE. The STATESUMMARY is initialized for a regular expression query $q = (\gamma, \tau, \mathscr{A})$, the NFA $\mathcal{N} = (Z, \Sigma, \delta, z_0, E)$ derived for it, and a user-defined *summary size*. We create vectors $\mathcal{S}_{\#Z}, \mathcal{A}_{\#Z} \in \mathbb{N}^{|Z|}$ to capture the global state counters and the active global state counters, respectively. Note that the active time window $\mathcal{A}$ is a subsequence of $\mathcal{S}$, so that we do not store it explicitly.

Additionally, we allocate $O(size \cdot |Z|)$ memory for all future characters and their local state counters. Consequently, the runtime of the INITIALIZE operation is dominated by the initialization of the local state counter vectors for all future elements, which requires $O(|\mathcal{S}| \cdot |Z|)$ time and space.

We use an array to hold the characters when implementing the STATESUMMARY. While this choice induces an overhead for the REMOVE operation, it facilitates efficient linear scans, which are required by our INSERT and REMOVE operations.

INSERT. The INSERT(c) operation (Alg. 1) appends an element c from the stream to the summary $\mathcal{S}$. To incorporate c, it is necessary to adjust the global state counters $\mathcal{S}_{\#Z}$, the active global state counters $\mathcal{A}_{\#Z}$, and for each element $e \in \mathcal{A}$ its active local state counters $\mathcal{A}_{e_{\#Z}}$ and local state counter $e_{\#Z}$. Finally, we have to initialize the active local state counter $\mathcal{A}_{c_{\#Z}}$ of c and its local state counters $c_{\#Z}$.

To comprehend the influence of character c on the underlying NFA $\mathcal{N}$, we assess its impact on ongoing (partial) matches within the active time window $\mathcal{A}$. Specifically, we identify the states affected by c. This is determined by the COMPUTESTATECOUNTERCHANGE function (line 11), which first initializes a new state counter vector *newCounters* of size $|Z|$ with zeros. Then, for each source state $from\_z \in Z$, we determine each target state $to\_z \in Z$, resulting from a transition using c, to increase the state counter $newCounters_{\#to\_z}$ by the count of $counters_{\#from\_z}$ (line 15). The resulting update vector is saved to *globCounterChange* (line 1).

In line 2 and line 3, we update the global state counters $\mathcal{S}_{\#Z}$ and active global state counters $\mathcal{A}_{\#Z}$ using *globCounterChange* (by vector addition, line 17). From line 4 to line 7, for each character e

Algorithm 1: INSERT.

```
input :Summary $\mathcal{S}$; character $c$ to insert; NFA $\mathcal{N} = (Z, \Sigma, \delta, z_0, E)$; active
       time window $\mathcal{A}$
1  globCounterChange ← computeStateCounterChange($\mathcal{A}_{\#Z}, \mathcal{N}, c$);
2  $\mathcal{S}_{\#Z}$ ← addStateCounters($\mathcal{S}_{\#Z}$, globCounterChange, $\mathcal{N}$);
3  $\mathcal{A}_{\#Z}$ ← addStateCounters($\mathcal{A}_{\#Z}$, globCounterChange, $\mathcal{N}$);
4  for $e$ in $\mathcal{A}$ do
5      localChange ← computeStateCounterChange($\mathcal{A}_{e_{\#Z}}, \mathcal{N}, c$);
6      $\mathcal{A}_{e_{\#Z}}$ ← addStateCounters($\mathcal{A}_{e_{\#Z}}$, localChange, $\mathcal{N}$);
7      $e_{\#Z}$ ← addStateCounters($e_{\#Z}$, localChange, $\mathcal{N}$);
8  $\mathcal{S}$ ← $\mathcal{S}.c$;
9  $\mathcal{A}_{c_{\#Z}}$ ← globCounterChange;
10 $c_{\#Z}$ ← globCounterChange;
11 function computeStateCounterChange(counters, $\mathcal{N}, c$)
12     newCounters ← $0^{|Z|}$;
13     for $from\_z \in Z$ do
14         for $to\_z \in \delta(from\_z, c)$ do
15             newCounters$_{\#to\_z}$ ← newCounters$_{\#to\_z}$ + counters$_{\#from\_z}$;
16     return newCounters;
17 function addStateCounters(stateCounters1, stateCounters2, $\mathcal{N}$)
18     summedCounters ← $0^{|Z|}$;
19     for $z \in Z$ do
20         summedCounters$_{\#z}$ ← stateCounters1$_{\#z}$ + stateCounters2$_{\#z}$;
21     return summedCounters;
```

in the active time window $\mathcal{A}$, we determine the impact of c on the active local state counters of e, denoted by *localChange* (line 5), to update its active local state counters $\mathcal{A}_{e_{\#Z}}$ and local state counters $e_{\#Z}$ by *localChange*. Finally, after appending the new character c to $\mathcal{S}$ (line 8), we initialize its active local state counters $\mathcal{A}_{c_{\#Z}}$ (line 9) and local state counters (line 10).

Complexity. The time complexity of INSERT is dominated by the updates of the (active) local state counters of characters in $\mathcal{A}$. The size of $\mathcal{A}$ is dictated by q_τ and, if $q_\tau > |\mathcal{S}|$, a single insert can affect *all* characters in $\mathcal{S}$. Since COMPUTESTATECOUNTERCHANGE requires $O(|Z|^2)$ time, it follows that the operation requires $O(min\{q_\tau, |\mathcal{S}|\} \cdot |Z_{\mathcal{N}}|^2)$ time using additional space $O(|Z|)$.

REMOVE. The REMOVE(c) operation (Alg. 2) removes element c from $\mathcal{S}$, which updates all counters. From global state counters $\mathcal{S}_{\#Z}$, we subtract the local state counters $c_{\#Z}$ (line 1), representing all (partial) matches involving c. Similarly, the active global state counters $\mathcal{A}_{\#Z}$ are reduced by the active local state counters $\mathcal{A}_{c_{\#Z}}$ (line 2), which signify the active (partial) matches involving c. When removing c, related (active) local state counters for remaining elements may also require updating.

The STATESUMMARY provides the count of (partial) matches for each element and state but lacks information regarding the joint matches between two characters c_1, c_2 from $\mathcal{S}$. The removal of c_1 may necessitate adjustments to the (active) local state counters of other elements in $\mathcal{S}$ if they had (partial) matches with c_1. Consequently, determining the proportion of joint (partial) matches in the respective counters $c_{1\#Z}$ and $c_{2\#Z}$ is required for updating $c_{2\#Z}$.

The immediate question that follows is if we can discern the mutual (partial) matches based on the values in $c_{1\#Z}$ and $c_{2\#Z}$. Unfortunately, this is impossible. The reason is that c_1 and c_2 can match many elements that are exclusively matched with either c_1 or c_2. Even if two counters correspond to the same state but originate from distinct elements, their values can diverge heavily, making them unsuitable for determining the number of joint (partial) matches. Thus, we will now present a procedure that determines the joint counts.

Let c_i represent the element we intend to remove from $\mathcal{S}$. Due to the time window q_τ, element c_i can only match elements within the interval $[i - q_\tau, i + q_\tau]$. Consequently, only the (active) local state counters of elements within the interval of size $2 \cdot q_\tau$ need to be updated. Thus, we execute two replays over the range $[i - q_\tau, i + q_\tau]$, initiating from $i - q_\tau$ and finishing at $i + q_\tau$ (line 6). These replays instantiate additional local state counters for each element within the interval, initializing them to zero. For updating the local state counters in the affected area, the initial replay incorporates c_i, while the latter omits c_i. Through this approach, we obtain two sets: $\mathcal{R}$, representing the local state counter vectors within the specified range post the first replay (c_i included) (line 7), and $\mathcal{R}'$, which stands for the local state counter vectors in the same range following post the second replay (c_i excluded) (line 9).

Next, we aim to determine the set of update vectors, $\mathcal{U}$, for the affected elements. For each update vector $\#u \in \mathcal{U}$, the values within $\#u$ indicate the number of (partial) matches the corresponding element has in common with c_i for a given state z in Z. For an element c_j $(j \neq i)$ in the affected range, we determine the corresponding replay vectors $\#v_j \in \mathcal{R}$ and $\#v_{j'} \in \mathcal{R}'$ such that the difference between $\#v_j - \#v_{j'}$ represents the shared match counts c_j and c_i have in common for each state. For instance, in Fig. 4 for the element D_{17} the respective replay vectors are $\#Z_{17} \in \mathcal{R}$ and $\#Z_{17'} \in \mathcal{R}'$.

The update vector is formulated as $\#u_j = \#v_j - \#v_{j'}$ and $\#u_j$ is then added to $\mathcal{U}$. Once $\mathcal{U}$ is computed, the local state counters, $c_{j\#Z}$, are adjusted for each affected c_j using the respective update vector $\#u_j \in \mathcal{U}$ with the update operation $c_{j\#Z} \longleftarrow c_{j\#Z} - \#u_j$ (line 11).

EXAMPLE 9 (REMOVE OPERATION). *Consider Fig. 4, where we aim to remove B_{21} and subsequently update the local state counters of affected elements. Given the time window constraint $q_\tau = 4$, B_{21} can be involved in matches spanning up to $2 \cdot q_\tau$, as depicted. Consequently, only the local state counters of elements within this range that are impacted by the removal of B_{21} need adjustment. Therefore, we start two replays over the interval $[17, 25]$, where for $\mathcal{R}$, we include B_{21} and for $\mathcal{R}'$, we exclude B_{21}. For each replay local state counter in $\mathcal{R}$, we subtract the corresponding one in $\mathcal{R}'$ to obtain the update vectors $\mathcal{U}$, denoting for each element in $[17, 25]$ the joint counts with B_{21}. In the last step, we subtract these update vectors from the corresponding local state counters to obtain the updated local state counters.*

Additionally, it is necessary to verify if the removed element falls within the active time window $\mathcal{A}$ (line 12). If so, *all* elements in $\mathcal{A}$ are potentially affected, making it necessary to update their active local state counters by replaying $\mathcal{A}$ separately once (line 13).

We want to highlight two points: (1) To update the local state counters, more than a single replay of the affected area is required. Elements within the area can match those outside, creating shared counts that must be kept for correctness. To achieve this, we subtract the update vectors from the (active) local state counters. (2) The replay produces a zero vector for elements in the area without matches to the removed element, as both replays yield identical vectors. For instance, this occurs for D_{17} in Fig. 4 where $\#Z_{17} = \#Z'_{17}$.

Algorithm 2: REMOVE.

```
input :Summary $\mathcal{S}$; character $c$ to remove; query $q = (\gamma, \tau, \mathscr{A})$ NFA
       $\mathcal{N} = (Z, \Sigma, \delta, z_0, E)$; active time window $\mathcal{A}$
1  $\mathcal{S}_{\#Z} \leftarrow \mathcal{S}_{\#Z} - c_{\#Z}$;
2  $\mathcal{A}_{\#Z} \leftarrow \mathcal{A}_{\#Z} - \mathcal{A}_{c_{\#Z}}$;
3  $\mathcal{S}_{\text{included}}$.INITIALIZE$(q, 2 \cdot q_\tau)$;
4  $\mathcal{S}_{\text{excluded}}$.INITIALIZE$(q, 2 \cdot q_\tau - 1)$;
5  affectedElements $\leftarrow \{e \text{ in } \mathcal{S} \mid |e.t - c.t| \leq q_\tau\}$;
6  for $e$ in affectedElements do
7      $\mathcal{S}_{\text{included}}$.INSERT$(e)$;
8      if $e \neq c$ then
9          $\mathcal{S}_{\text{excluded}}$.INSERT$(e)$;
10 for $e$ in affectedElements do
11     $e_{\#Z} \leftarrow e_{\#Z} - (\mathcal{S}_{\text{included}\, e_{\#Z}} - \mathcal{S}_{\text{excluded}\, e_{\#Z}})$;
12 if $|c.t - currentTimestamp| \leq q_\tau$ then
13     replayActiveTimeWindow();
14 function replayActiveTimeWindow()
15     $\mathcal{S}_{\text{temp}}$.INITIALIZE$(q, q_\tau)$;
16     for $e$ in $\mathcal{A}$ do
17         $\mathcal{S}_{\text{temp}}$.INSERT$(e)$;
18     for $e$ in $\mathcal{A}$ do
19         $\mathcal{A}_{e\#Z} = \mathcal{S}_{\text{temp}\, e\#Z}$;
```

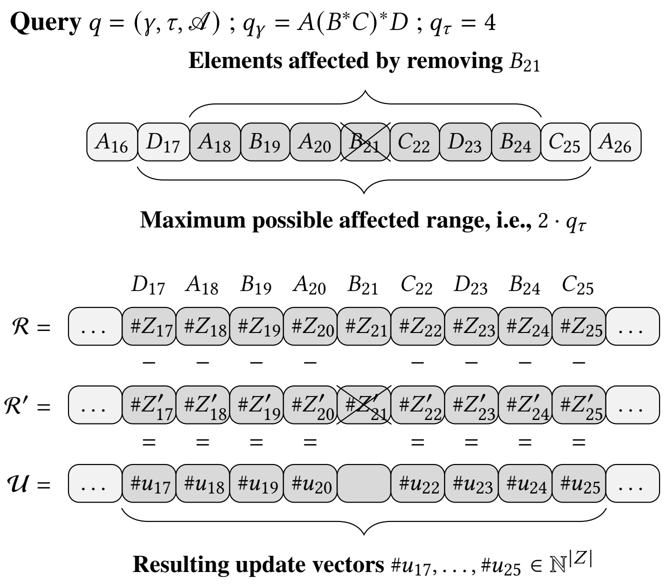

Query $q = (\gamma, \tau, \mathscr{A})$; $q_\gamma = A(B^*C)^*D$; $q_\tau = 4$
Elements affected by removing B_{21}
A_{16} D_{17} A_{18} B_{19} A_{20} B_{21} C_{22} D_{23} B_{24} C_{25} A_{26}
Maximum possible affected range, i.e., $2 \cdot q_\tau$
D_{17} A_{18} B_{19} A_{20} B_{21} C_{22} D_{23} B_{24} C_{25}
$\mathcal{R} =$ ... $\#Z_{17}$ $\#Z_{18}$ $\#Z_{19}$ $\#Z_{20}$ $\#Z_{21}$ $\#Z_{22}$ $\#Z_{23}$ $\#Z_{24}$ $\#Z_{25}$ ...
$\mathcal{R}' =$ ... $\#Z'_{17}$ $\#Z'_{18}$ $\#Z'_{19}$ $\#Z'_{20}$ $\#Z'_{21}$ $\#Z'_{22}$ $\#Z'_{23}$ $\#Z'_{24}$ $\#Z'_{25}$ ...
$\mathcal{U} =$ ... $\#u_{17}$ $\#u_{18}$ $\#u_{19}$ $\#u_{20}$ $\#u_{22}$ $\#u_{23}$ $\#u_{24}$ $\#u_{25}$ ...
Resulting update vectors $\#u_{17}, \ldots, \#u_{25} \in \mathbb{N}^{|Z|}$

Figure 4: Evaluation of a summary when removing character B_{21}; for shaded elements, the local state counters are updated.

Complexity. As for the insertion, removing an element c_i could affect the local state counters of *all* elements in $\mathcal{S}$, if $q_\tau > |\mathcal{S}|$. The cumulative effect of replaying the summary $\mathcal{S}$ leads to $\sum_{i=1}^{|\mathcal{S}|} i = \frac{|\mathcal{S}|^2 + |\mathcal{S}|}{2} \in O(|\mathcal{S}|^2)$ potential local state counter updates. In the worst case, each update for $c_{\#Z}$ requires time $|Z|^2$ and $\mathcal{A}$ may also be replayed, resulting in a total replay time of $O(min\{q_\tau, |\mathcal{S}|\}^2 \cdot |Z|^2 + |\mathcal{A}|^2) \in O(min\{q_\tau, |\mathcal{S}|\}^2 \cdot |Z|^2)$. Moreover, for initializing the replays, we require space of size $O(min\{q_\tau, |\mathcal{S}|\} \cdot |Z|)$.

An overview of the operations is given in Table 3.

Table 3: Run time and space complexity of summary operations.

Operation	Time complexity	Space complexity
INITIALIZE	$O(\lvert\mathcal{S}\rvert \cdot \lvert Z\rvert)$	$O(\lvert\mathcal{S}\rvert \cdot \lvert Z\rvert)$
INSERT	$O(min\{q_\tau, \lvert\mathcal{S}\rvert\} \cdot \lvert Z\rvert^2)$	$O(\lvert Z\rvert)$
REMOVE	$O(min\{q_\tau, \lvert\mathcal{S}\rvert\}^2 \cdot \lvert Z\rvert^2)$	$O(min\{q_\tau, \lvert\mathcal{S}\rvert\} \cdot \lvert Z\rvert)$

4.3 Other Aggregate Functions

Above, we described the STATESUMMARY for queries involving COUNT as the aggregation function. We now turn to the changes needed to support SUM, which then also facilitates AVG.

As before, when a (valued) character c is added to $\mathcal{S}$, transitions of the automata are triggered, creating new (partial) matches in subsequent states. For all affected (partial) matches, character c and its payload value c^{v_i} need to be incorporated. We define the SUM counters and rules based on those introduced for COUNT.

For each state $z_i \in Z$, we create a SUM counter $\mathcal{S}_{\#z_i} \in \mathbb{N}$, representing the current total sum over all payload values of (partial) matches in z_i. This yields a set of *global sum counters* $\mathcal{S}_{\#Z} = \{\mathcal{S}_{\#z_0}, \dots, \mathcal{S}_{\#z_{|Z|}}\}$. As for the COUNT rules, there are three cases we have to handle for defining the SUM rules: (1) self-loops, (2) transitions from other states, and (3) transitions originating from z_0.

Before defining the respective SUM rules, we want to explain the idea behind handling each of the three cases. When c^v triggers a transition from z_0 to z_k with $z_0 \neq z_k$, a new partial match is created; hence, we add value v to the current value of $\mathcal{S}_{\#z_i}$. If c triggers a self-loop in z_k, all (partial) matches in z_k are doubled, i.e., we add to $\mathcal{S}_{\#z_k}$ the number of (partial) matches present in z_k multiplied by v. If c triggers transitions originating from states $z_i \neq z_k$, all partial matches in z_i use c for transition. Consequently, we add to the current value of $\mathcal{S}_{\#z_k}$ the SUM counter $\mathcal{S}_{\#z_i}$ plus the number of (partial) matches in z_i multiplied by v, i.e., $\#z_i \cdot v$. Based thereon, we formalize the SUM rules.

For each $z_i \in Z$, each $\mathcal{S}_{\#z_i} \in \mathcal{S}_{\#Z}$, and a valued character c^v:

(1) If z_i has a self-loop with c^v:

$$\text{if } \delta(z_i, c^v) = \{z_i\} \text{ then } \mathcal{S}_{\#z_i} \longleftarrow \mathcal{S}_{\#z_i} + \#z_i \cdot v$$

(2) If transitions from states $\{z_{i_1}, z_{i_2}, \dots, z_{i_l}\}$ lead to z_k upon processing character c^v (excluding the self-loop case):

$$\text{if } \delta(z_{i_j}, c^v) = \{z_k\} \text{ for } j \in \{1, 2, ..., l\} \text{ then}$$
$$\mathcal{S}_{\#z_k} \longleftarrow \mathcal{S}_{\#z_k} + \textstyle\sum_{j=1}^{l} \mathcal{S}_{\#z_{i_j}} + \#z_{i_j} \cdot v$$

(3) If the transition goes from the initial state z_0 to $z_k \neq z_0$:

$$\text{if } \delta(z_0, c^v) = \{z_k\} \text{ then } \mathcal{S}_{\#z_k} \longleftarrow \mathcal{S}_{\#z_k} + 1 \cdot v$$

The (active) local sum counters are defined analogously to their COUNT-based counterparts, but rely on the SUM rules.

Next, we will illustrate the SUM counter update process based on the SUM rules.

EXAMPLE 10 (STATESUMMARY MATCHING FOR SUM). *Consider Fig. 2, where we will now set $q_{\mathscr{A}}$ = SUM. Due to the underlying NFA, we have the* SUM *counters $\mathcal{S}_{\#Z} = \{\mathcal{S}_{\#z_1}, \mathcal{S}_{\#z_2}, \mathcal{S}_{\#z_3}\}$. The resulting* SUM *rules are:*
$A^v : \mathcal{S}_{\#z_1} \longleftarrow \mathcal{S}_{\#z_1} + 1 \cdot v,$
$B^v : \mathcal{S}_{\#z_2} \longleftarrow \mathcal{S}_{\#z_2} + \mathcal{S}_{\#z_2} + \#z_2 \cdot v + \mathcal{S}_{\#z_1} + \#z_1 \cdot v,$
$C^v : \mathcal{S}_{\#z_1} \longleftarrow \mathcal{S}_{\#z_1} + \mathcal{S}_{\#z_1} + \#z_1 \cdot v + \mathcal{S}_{\#z_2} + \#z_2 \cdot v$, *and*
$D^v : \mathcal{S}_{\#z_3} \longleftarrow \mathcal{S}_{\#z_3} + \mathcal{S}_{\#z_1} + \#z_1 \cdot v.$

We modify the stream s to $s^v = \langle A^5_1, B^3_2, A^2_3, D^1_4, B^7_5, C^3_5, D^8_7\rangle$. After inserting A^5_1, we update $\mathcal{S}_{\#z_1} \longleftarrow 0 + 1 \cdot 5 = 5$. When B^3_2 occurs, $\mathcal{S}_{\#z_2} \longleftarrow 0 + 0 \cdot 3 + 5 + 1 \cdot 3 = 8$ and A^2_3 causes $\mathcal{S}_{\#z_1} \longleftarrow 5 + 1 \cdot 2 = 7$. Due to B^5_4, we update $\mathcal{S}_{\#z_2} \longleftarrow 8 + 1 \cdot 4 + 7 + 2 \cdot 4 = 27$. Encountering C^7_5, we update $\mathcal{S}_{\#z_1} \longleftarrow 7 + 2 \cdot 5 + 27 + 4 \cdot 5 = 64$. Finally, we receive D^8_7, therefore, $\mathcal{S}(\#z_3) \longleftarrow 0 + 64 + 8 \cdot 8 = 128$.

Maintaining the STATESUMMARY for SUM and COUNT, support for AVG is obtained by using the ratio $\mathcal{S}_{\#z_i}/\#z_i$ for each state $z_i \in Z$.

5 SUMMARY SELECTION

To address Problem 1, we shall maintain a summary using the STATESUMMARY that aims at minimizing the loss in query evaluation. To this end, we first introduce a benefit function to quantify the potential of a character to contribute to a substantial amount of complete matches (§5.1), which then guides our selection strategy (§5.2). Again, we introduce the concepts for COUNT aggregations, and later discuss the changes required to support SUM and AVG (§5.3).

5.1 Benefit Function

In order to build a summary, i.e., a stream projection as introduced in §3, we define a benefit function $\mathcal{B} : \mathcal{S} \longrightarrow \mathbb{R}$ that scores a character based on the *present benefit* for current aggregation results, and the *expected benefit* for future results.

Present Benefit. The *present benefit* $\mathcal{B}_{\text{pres}} : \mathcal{S} \longrightarrow \mathbb{N}$ (for COUNT) of character $c \in \mathcal{S}$ is defined by the number of complete matches:

$$\mathcal{B}_{\text{pres}}(c) = \textstyle\sum_{c_{\#z_i} \in c_{\text{CM}}} c_{\#z_i} \qquad (1)$$

We write $\mathcal{B}_{\mathcal{A}_{\text{pres}}}(c)$ to denote all complete matches of c within $\mathcal{A}$.

Expected Benefit. The *expected benefit* quantifies the expected count of complete matches with c using its active local state counter $\mathcal{A}_{c_{\#Z}}$ (i.e., partial matches with c that may lead to further matches) and the remaining time span Δ_τ, in which c can participate in matches.

We assume that the probability distribution over Σ, coined *alphabet probability* and denoted as $P_\Sigma : \Sigma \longrightarrow [0, 1]$, is known. That is, $P_\Sigma(c)$ represents the probability that character c is the subsequent stream element. The time window q_τ determines the maximum time an element can lead to (partial) matches. As such, to compute the expected benefit, we induce for each timestamp $t \in [0, q_\tau]$ all possible characters $c \in \Sigma$ and measure their influence on the active state counters weighted by their occurrence probability $P_\Sigma(c)$. Based on the active local state counters $c_{\#z_i} \in \mathcal{A}_{c_{\#Z}}$, we compute how each $c_{\#z_i}$ is affected for each time step. For each $c_{\#z_i}$, the resulting count represents the averaged count over all possible words Σ^t weighted by $P_\Sigma(c)$, i.e., the expected count of matches after t time steps.

Consider Fig. 5, with the regular expression $q_\gamma = A(B^*C)^*D$ and the illustrated COUNT rules. Each rule models how an active state counter is *influenced*. For each active state counter $\#z_i$, we determine the *total influence* of all characters on $\#z_i$ for a single time step. For instance, A and C influence $\#z_1$, resulting in the total influence $1+\#z_1+\#z_2$. Each influence in a count rule is weighted by probability $P_\Sigma(c)$ of character c influencing the state counter, e.g., for $\#z_1$, we obtain $1 \cdot P(A) + (\#z_1 + \#z_2) \cdot P(C)$, the *total weighted influence*.

Let $\mathcal{I} = \{I_1, \dots, I_{|Z|}\}$ be the set of all influences from COUNT rules. An *influence function*, $I_k : \mathbb{T} \longrightarrow \mathbb{R}$, indicates a weighted COUNT rule at time t. It and outputs the character's influence at

Query $q = (\gamma, \tau)$
$q_\gamma = A(B^*C)^*D$
$q_\tau = \tau$

Alph. prob.:
$P(A) = 0.5,$
$P(B) = 0.2,$
$P(C) = 0.15,$
$P(D) = 0.15$

StateSummary

$$\boxed{\begin{array}{l} \#z_0 = \#z_1 = \#z_2 = \#z_3 = 0 \\ A\colon \#z_1 \leftarrow \#z_1 + 1 \\ B\colon \#z_2 \leftarrow \#z_2 + \#z_2 + \#z_1 \\ C\colon \#z_1 \leftarrow \#z_1 + \#z_1 + \#z_2 \\ D\colon \#z_3 \leftarrow \#z_3 + \#z_1 \end{array}}$$

$\boxed{t = 0}$

$$\begin{aligned} \#z_1(0) &= \#z_1 \\ \#z_2(0) &= \#z_2 \\ \#z_3(0) &= \#z_3 \end{aligned}$$

$\boxed{t = 1}$

$$\begin{aligned} \#z_1(1) &= \#z_1(0) + \overbrace{1 \cdot P(A)}^{A's\ \text{influence}} + \overbrace{(\#z_1(0) + \#z_2(0)) \cdot P(C)}^{C's\ \text{influence}} \\ \#z_1(1) &= \#z_1(0) + 0.5 + \#z_1(0) \cdot 0.15 + z_2(0) \cdot 0.15 \\ \#z_1(1) &= 1.15 \cdot \#z_1(0) + 0.15 \cdot \#z_2(0) + 0.5 \\ \#z_2(1) &= \#z_2(0) + (\#z_2(0) + 1) \cdot P(B) \\ \#z_2(1) &= \#z_2(0) + \#z_2(0) \cdot 0.2 + 0.2 \\ \#z_2(1) &= 1.2 \cdot \#z_2(0) + 0.2 \\ \#z_3(1) &= \#z_3(0) + \#z_1(0) \cdot P(D) \\ \#z_3(1) &= \#z_3(0) + 0.15 \cdot \#z_1(0) \end{aligned}$$

$\boxed{t = 2}$

$$\begin{aligned} \#z_1(2) &= \#z_1(1) + \overbrace{1 \cdot P(A) + (\#z_1(1) + \#z_2(1)) \cdot P(C)}^{\text{Total weighted influence of } A \text{ and } C \text{ at } t\text{=}1} \\ \#z_1(2) &= \#z_1 \cdot 1.15 + \#z_2(1) \cdot 0.15 + 0.5 + P(A) + \\ &\quad (\#z_1 \cdot 1.15 + \#z_2(1) \cdot 0.15 + 0.5 + \#z_2(1) \cdot 1.2 + 0.2) \cdot P(C) \\ \#z_1(2) &= 1.15 \cdot (1.15 \cdot z_1(1) + 0.15 \cdot z_2(1) + 0.5) + 0.15 \\ &\quad \cdot (1.2 \cdot \#z_2(1) + 0.2) + 0.5 \\ \#z_1(2) &= \underbrace{1.3225 \cdot \#z_1(0) + 0.3525 \cdot \#z_2(0) + 1.105}_{\text{Linear combination}} \\ \#z_2(2) &= \#z_2(1) + \ldots \end{aligned}$$

Figure 5: Extending the example from Fig. 2, we compute the expected benefit for $t \in [0, 1, 2]$. Resulting coefficients are cached.

time t. For every state counter $\#z_i$ in $\#Z$, we define an *influence set* $S_i \subseteq \{1, \ldots, |Z|\}$, which contains the indices of influence functions that affect $\#z_i$. In Fig. 5, the influence of A is described as $I_1(t) = 1 \cdot P_\Sigma(A)$, whereas for character C, it is defined as $I_3(t) = (\#z_1(t) + \#z_2(t)) \cdot P_\Sigma(C)$. Consequently, the influence set corresponding to $\#z_1$ is expressed as $S_1 = \{1, 3\}$, i.e., both A and C influence $\#z_1$.

The computation at time $t + 1$ relies on the results at time t. As such, we derive a linear recurrence relation that operates on the active local state counters $\mathcal{A}_{c_{\#Z}}$ of a character $c \in \mathcal{S}$ to compute the expected benefit and, therefore, the overall *benefit*.

Specifically, at time $t = 0$, we adopt $c_{\#z_i}(0) = c_{\#z_i}$ as the initial relation. Then, the general linear recurrence relation for each $t + 1$ and counter $c_{\#z_i}$ is defined as:

$$c_{\#z_i}(t + 1) = c_{\#z_i}(t) + \textstyle\sum_{k \in S_i} I_k(t) \tag{2}$$

For updating $c_{\#z_i}$, we add to the previous value $c_{\#z_i}(t)$ the total weighted influence on $c_{\#z_i}$ at time t, i.e., the sum over all influences of characters affecting c at time t.

Since the output of Eq. 2 depends on the initial active local state counters $\mathcal{A}_{c_{\#Z}}$, and these values only increase, the calculated output captures both the present and the expected benefit. Thus, we define the *benefit function* $\mathcal{B} : \mathcal{S} \times \mathbb{T} \to \mathbb{R}$ as:

$$\mathcal{B}(c, t) = \textstyle\sum_{c_{\#z_i} \in \mathcal{A}_{c_{CM}}(t)} c_{\#z_i} \tag{3}$$

estimating the expected count of complete matches c leads to.

Caching Coefficients. The computational cost for $\mathcal{B}(c, t)$ grows as the time window q_τ increases. We therefore employ a cache that is built during pre-processing using the inputs P_Σ and $\mathcal{S}$. Using Fig. 5 for illustration, we note that for each time step t and each state counter $\#z_i$, a linear combination with constant coefficients is derived. A growing time window τ modifies only these coefficients, leaving the state counter $\#z_i$ value anchored to its initial. Hence, for every $t \in [0, q_\tau]$, we store the coefficients corresponding to each state counter $\#z_i$. In Fig. 5, for instance, for $\#z_1(2)$, the cached coefficients are $1.3225, 0.3525$, and 1.105.

In total, for every $t \in [0, q_\tau]$, we keep track of all $O(|Z|)$ state counters by caching at most $O(|Z|)$ coefficients, leading to the memory requirement $O(q_\tau \cdot |Z| \cdot |Z|)$, simplifying to $O(q_\tau \cdot |Z|^2)$. After the cache is computed, for a given element c and t, we can compute $\mathcal{B}(c, t)$ in time $O(|Z|)$ since we have to factor in all cached coefficients for a given input tuple.

5.2 Selection Strategy

The *selection strategy*, modelled as $\psi : \mathcal{S} \times \Sigma \to \mathcal{S}$, derives a new summary from the current summary upon the arrival of a character $c \in \Sigma$. It may insert c, replace c_{old} in $\mathcal{S}$ by c, or discard c (here, $\mathcal{S}|c_{\text{old}}$ denotes the removal of character c_{old} from summary $\mathcal{S}$)

$$\psi(\mathcal{S}, c) = \begin{cases} \mathcal{S}.c & \text{if } c \text{ is inserted into } \mathcal{S} \\ (\mathcal{S}|c_{\text{old}}).c & \text{if } c_{\text{old}} \text{ is replaced by } c \\ \mathcal{S} & \text{if } c \text{ is discarded} \end{cases} \tag{4}$$

The decision of ψ relies on the benefit function $\mathcal{B}$. If the summary capacity has not been reached, a new character c is inserted. If the summary is filled already, the choice between replacement and discarding is done by iterating through the summary from the oldest to the newest element, identifying the character c' with the lowest benefit. For elements outside the active time window $\mathcal{A}$, it holds that $\mathcal{B}(c', 0) = \mathcal{B}_{\text{pres}}(c')$ is their benefit. Upon reaching $\mathcal{A}_{\text{init}}$, we set $i_{\text{old}} = i_{\text{new}} = \mathcal{A}_{\text{init}}$, marking both the oldest and initial newest initiators of $\mathcal{A}$. Note that i_{old} remains constant. As we proceed, encountering a new initiator updates i_{new}. We then determine values $\Delta_{\tau_{\text{old}}} = q_\tau - (c.t - i_{\text{old}}.t)$ and $\Delta_{\tau_{\text{new}}} = q_\tau - (c.t - i_{\text{new}}.t)$. The value $\Delta_{\tau_{\text{old}}}$ specifies the duration during which both c and i_{old} will be involved in matches. Conversely, $\Delta_{\tau_{\text{new}}}$ designates the overall period in which c will participate in matches at most. Based thereon, we define an *averaged benefit function* $\mathcal{B}' : \mathcal{S} \times \mathbb{T} \times \mathbb{T} \to \mathbb{R}$:

$$\mathcal{B}'(c, \Delta_{\tau_{\text{old}}}, \Delta_{\tau_{\text{new}}}) = \mathcal{B}_{\text{pres}}(c) + \frac{(\mathcal{B}(c, \Delta_{\tau_{\text{old}}}) - \mathcal{B}_{\mathcal{A}_{\text{pres}}}(c)) + (\mathcal{B}(c, \Delta_{\tau_{\text{new}}}) - \mathcal{B}_{\mathcal{A}_{\text{pres}}}(c))}{2}$$

The function offers a balanced measure of the benefit of c, averaging its potential benefits over the periods $\Delta_{\tau_{\text{old}}}$ and $\Delta_{\tau_{\text{new}}}$.

The idea is based on the following observation: the value of $\Delta_{\tau_{\text{old}}}$ can be small, indicating that c might not contribute to matches for long; however, $\Delta_{\tau_{\text{new}}}$ dictates how long c can lead to matches. On the other hand, relying solely on $\Delta_{\tau_{\text{new}}}$ might not capture the expected benefit accurately, given that c aligns with i_{old} in many (partial) matches, but not for the span of $\Delta_{\tau_{\text{new}}}$. Also, note that both $\mathcal{B}(c, \Delta_{\tau_{\text{old}}})$ and $\mathcal{B}(c, \Delta_{\tau_{\text{new}}})$ include the counts from $\mathcal{B}_{\mathcal{A}_{\text{pres}}}(c)$. Since we already accounted for that values by $\mathcal{B}_{\text{pres}}(c)$, we subtract them.

Finally, we compute $c_{\#Z}$ for c using $\mathcal{A}_{\#Z}$, as if inserting c into $\mathcal{S}$. If the benefit of c' is smaller than $\mathcal{B}'(c, \Delta_{\tau_{\text{old}}}, \Delta_{\tau_{\text{new}}})$, we replace c' by c; otherwise, c is discarded. Since we compute $\mathcal{B}'$ in time $O(|Z|)$ for each element in $\mathcal{S}$, the run time complexity of the selection strategy ψ is given by $O(|\mathcal{S}| \cdot |Z|)$ using $O(q_\tau \cdot |Z|^2)$ additional space.

5.3 Modifications for Other Aggregate Functions

When adopting the SUM aggregate function, the benefit function needs to be adapted. The function for the present benefit is defined analogously, i.e., summing up all values of complete matches. Turning to the expected benefit function, we note that for COUNT, we estimate the count of matches that character c may generate. For SUM, each of these future matches carries a value v. Therefore, the expected benefit for c is augmented by its current present benefit (for SUM), combined with the expected match count of c weighted by v. While this function underestimates the expected SUM value for c, the underestimation is consistent over all characters in the summary.

6 EXPERIMENTAL EVALUATION

To evaluate SUSE, we conducted experiments using the setup described in §6.1. Specifically, we present results on the overall effectiveness (§6.2); on the sensitivity of the approach (§6.3); on an ablation and recall study (§6.4); on a comparison against a state-of-the-art engine for regular expressions (§6.5); and on two studies with real-world datasets (§6.6 and §6.7).

6.1 Experimental Setup

Our implementation and experimental setup is publicly available[1].

Datasets and Parameters. We used two real-world datasets and synthetic data to assess SUSE regarding the COUNT aggregate. First, we used one month of the Citi Bike dataset [9] that captures information about bike rentals. It contains ≈ 3.8 million events, each of them describing the duration of a trip, the start and end station, a bike ID, and information about the driver.

Our second real-world dataset, NASDAQ [26], contains stock trading data for 462028 events on a minute-by-minute basis. Each entry lists the stock symbol, opening, closing, highest and lowest prices, and the trading volume for that minute.

In synthetic data experiments, we assessed the influence of different parameters on SUSE, i.e., the summary size $|\mathcal{S}|$, stream size $|s|$, number of evaluation timestamps $|\mathcal{E}|$, and the time window q_τ and length of q_γ of a query. To generate the data streams, characters were drawn from alphabets based on different distributions: Zipfian, normal, and uniform. The evaluation timestamps, see Problem 1, have been sourced from a Poisson process and a uniform distribution. Yet, the differences between both models have been negligible.

Baselines. We evaluated the performance of our approach against two baselines: A random baseline replaces a randomly chosen character in the summary once its capacity is met. The FIFO baseline consistently replaces the oldest element.

Our ablation study used a restricted SUSE model, where the benefit function $\mathcal{B}'$ is solely based on the present benefit $\mathcal{B}_{\text{pres}}$.

We examined SUSE's efficiency by comparing it to the state-of-the-art engine REmatch [36]. To ensure a fair comparison, we ran it with the *–mode=noutputs* parameter to avoid the output overhead.

Metrics. We assess the evaluation quality using the *average total match ratio*. At each evaluation time point $t \in \mathcal{E}$, we divide the number of complete matches from SUSE by the number of complete matches from a corresponding baseline. The average of the sum of these ratios denotes the average total match ratio.

[1] https://anonymous.4open.science/r/SuSe-8F12/README.md

We also evaluate the *average total match ratio recall*. At each evaluation timestamp, it is calculated by dividing the match count from SUSE by the ground truth, obtained by setting the summary size $|\mathcal{S}|$ equal to the stream size (limited to 2000 in these experiments). We also examine the *detected matches recall*, i.e., the ratio of complete matches in SUSE to *all* potential matches over time.

We further report standard performance metrics, such as *throughput* (average number of elements processed per second), *latency* (average processing duration per element), and *memory usage* (maximum resident set size).

Implementation. Our experiments were conducted using a C++ implementation, with 128-bit counters. They ran on a server with 4 Intel Xeon E7-4880 (60 cores, 1TB RAM).

6.2 Overall Effectiveness

We first compare against a random baseline and a FIFO baseline. For Fig. 6a and Fig. 6b, we assessed how variations in the summary size $|\mathcal{S}|$ (100 to 5000), time-window size q_τ (10 to 250), and alphabet probability distribution P_Σ (Zipfian, uniform, normal) affect the average total match ratio. We adopt a query with $q_\gamma = a(b^*c)^*d(e|f)g^*$.

SUSE outperforms its baselines by choosing stream projections that generate up to 10^{20} more matches. There is no parameter combination, where SUSE performs worse than a baseline (indicated by the black dashed line). A clear trend emerges for both plots: the larger q_τ and the smaller $|\mathcal{S}|$, the more significant the advantage.

These trends hold consistently across different P_Σ. We explain this as follows: SUSE quantifies each element's value using the underlying STATESUMMARY and the benefit function $\mathcal{B}$, exploiting parameters such as q_τ to choose better projections successively. Conversely, with FIFO, an element remains within the projection for a time equal to the summary size, even though it still satisfies the time-window constraint. While, for the random baseline, elements can stay longer than the summary size, both baselines do not quantify the value of individual elements; hence, elements involved in many matches or rare elements are replaced.

The larger the summary size, the more matches are found by chance by the baselines; nevertheless, for these parameters, SUSE chooses stream projections that generate at least three orders of magnitude more matches on average.

6.3 Sensitivity Analysis

In the sensitivity analysis, we analyze the effect of different parameters on the average total match ratio. We fix $q_\tau = 100$ and $|\mathcal{S}| = 500$ while varying the stream sizes between 10000 and 25000, the numbers of evaluations varying from 25 to 50, and P_Σ as either Zipfian or uniform. Each y-value represents an average of over 50 runs.

Number of Evaluation Timestamps. Fig. 6c examines the influence of increasing the number of evaluation timestamps $|\mathcal{E}|$ on the average total match ratio. As the boxplots indicate, increasing the number of evaluations does not impact the results.

Query Length. In Fig. 6d, we investigate the effect of increasing the query length. As the query length grows, selecting projections that yield matches becomes increasingly challenging since more elements are required for satisfying the query. SUSE addresses this by quantifying the importance of individual elements, prioritizing, for instance, rare elements that are essential to keep to obtain matches.

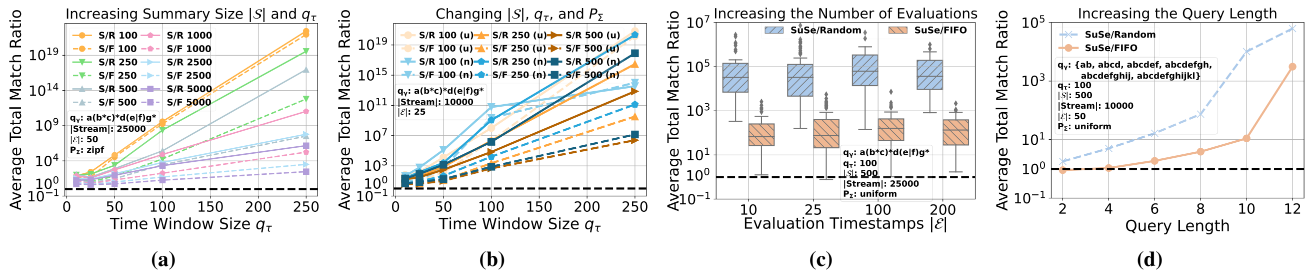


Figure 6: Examining the impact on average total match ratio (higher is better) by varying (a) summary and time-window sizes, (b) the alphabet probability distribution P_Σ, (c) the number of evaluation timestamps $|\mathcal{E}|$, and (d) the query length.

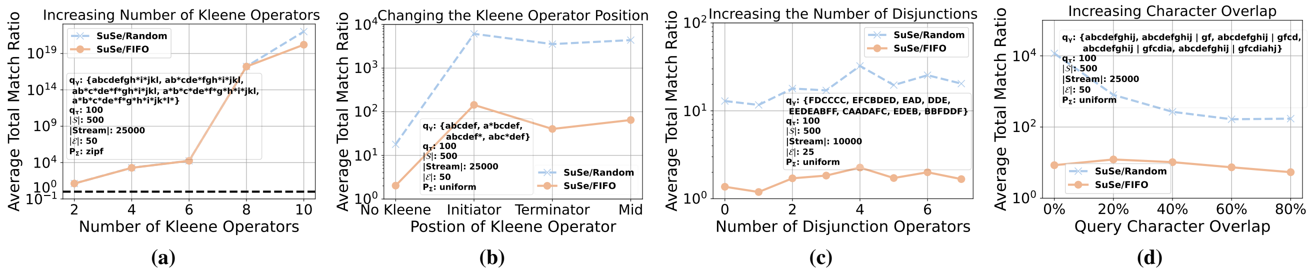


Figure 7: Examining the impact on average total match ratio (higher is better) by varying (a) the count of Kleene stars, (b) Kleene star position, (c) disjunction operator count, and (d) query character overlap.

In contrast, the baselines lack this level of insight as they are agnostic to the current state. While for query length two, FIFO performs slightly better, for query lengths above ten, SuSe is superior by at least one and up to almost five orders of magnitude.

Kleene Operator. We examine the influence of the Kleene operator. In Fig. 7a, we employed a regex of length twelve and incrementally increased the number of Kleene operators, distributing them randomly and uniformly. As SuSe is aware of characters with Kleene operators (by Count/Sum rules), it selects better summaries. An increasing number of Kleene operators amplifies the effect.

In this context, it is essential to note that the Count or Sum rules indicate which characters in the RegEx have associated Kleene operators, as these elements activate multiple rules and updates. Consequently, SuSe is aware of characters with Kleene operators and considers this when selecting projections. Specifically, more matches of up to three orders of magnitude are found when using four Kleene operators. This advantage amplifies further, reaching up to 10^{20} when the number of Kleene operators is increased to ten.

In Fig. 7b, we examined four regular expressions: one without a Kleene operator and three with the Kleene operator's position varied. Without a Kleene operator, SuSe's advantages are minimal, while there is a large difference for the other cases, independent of the position of the Kleene operator.

Disjunction Operator. We also explored the effect of the disjunction operator in Fig. 7c. We varied the number of disjunctions from zero to seven and generated random queries, which were then combined using the disjunction operators. Yet, their impact is negligible.

In Fig. 7d, we fixed a query of length ten and progressively increased the character overlap. We randomly selected two characters of the given query for each step, linking them to the query and a disjunction. While the FIFO result remains constant, the results regarding the random baseline become worse with increased overlap. Still, SuSe yields significant improvements.

6.4 Ablation and Recall Study

Ablation Study. We compared the benefit function $\mathcal{B}'$ against a setup using solely the present benefit $\mathcal{B}_{\text{pres}}$ in terms of the average total match ratio, the related recall, and the detected matches recall.

In Fig. 8a, a trend similar to that in Fig. 6a is observed: smaller $|\mathcal{S}|$ and larger q_τ lead to better projection selections. This is attributed to $\mathcal{B}'$ providing a more accurate assessment of a character's future potential. In contrast, $\mathcal{B}_{\text{pres}}$ only considers the current match count of elements, neglecting any potential future contributions from new elements. For $q_\tau = 250$, SuSe with $\mathcal{B}'$ obtains at least three orders of magnitude more matches on average over solely using $\mathcal{B}_{\text{pres}}$, up to 10^{18} more for smaller values of $|\mathcal{S}|$, showing the effectiveness of $\mathcal{B}'$.

Average Total Match Ratio Recall. In Fig. 8b and Fig. 8c, we investigated how close SuSe using $\mathcal{B}'$ and $\mathcal{B}_{\text{pres}}$ comes to the optimal average total match ratio value by computing ground truths for stream sizes 2000 and 3000. Since the counters for the ground truth are increasing rapidly, we used small stream sizes and 256 bits integers for this experiment to avoid overflows.

For both plots, an increase in summary size positively affects recall for all methods. This improvement can be attributed to the reduced difference between the ground truth's summary size (equal to the stream size) and SuSe's. Nonetheless, it is evident that SuSe, when employing expected benefits $\mathcal{B}'$, attains significantly higher recall values compared to using just $\mathcal{B}_{\text{pres}}$. Additionally, a noticeable trend is that when $|\mathcal{S}| \geq q_\tau$, the expected benefit $\mathcal{B}'$ method selects substantially better projections, leading to a considerable increase in recall up to 80% − 95%.

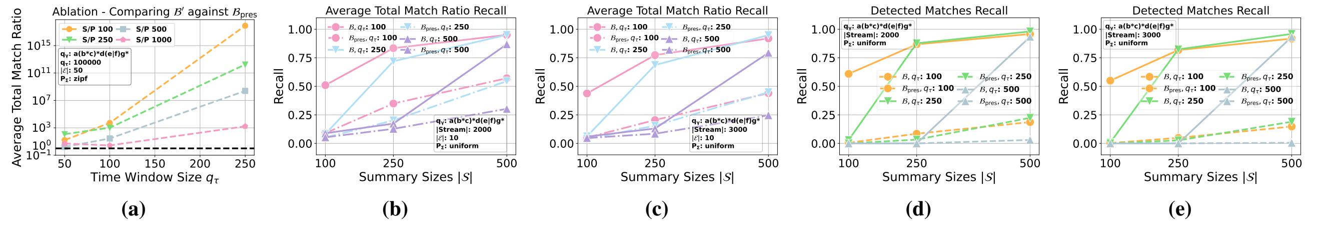


Figure 8: Examining (a) the impact of expected benefit $\mathcal{B}'$ on the average total match ratio, (b) proximity to lower bound in average total match ratio recall, and (c) proximity to lower bound in detected matches recall (higher is better in all plots).

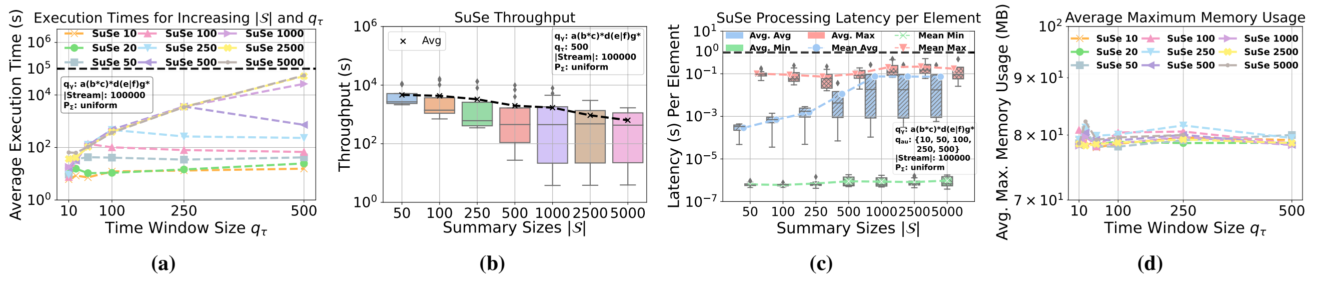


Figure 9: (a) Execution time (lower is better) for processing 10^5 elements across different summary and time window sizes, (b) average minimum, average maximum, and average processing latency per element (lower is better).

This trend can be attributed to our assumption in §5.2 regarding the expected benefit $\mathcal{B}'$, where we implicitly assumed due to $\Delta_{\tau_{\text{old}}}$ and $\Delta_{\tau_{\text{new}}}$ that at least one time-window size of elements fits into the summary. For values $|\mathcal{S}| \geq q_\tau$, this assumption is valid, leading to more accurate estimates and an enhanced recall, representing a boost of approximately 70% compared to the recall achieved using $\mathcal{B}_{\text{pres}}$. Furthermore, as the stream size grows, all recalls in Fig. 8c diminish, attributable to the corresponding increase in the ground truth's summary size.

Detected Matches Recall. For Fig. 8d and Fig. 8e, we examined how many of all possible matches were present in SuSE, again comparing $\mathcal{B}'$ and $\mathcal{B}_{\text{pres}}$. The trends align closely with those observed for the average total match recall. SuSE using $\mathcal{B}'$ yields a recall between $90\% - 98\%$ when $|\mathcal{S}| = 500$, indicating that SuSE chooses excellent stream projections. In contrast, the recalls by $\mathcal{B}_{\text{pres}}$ are consistently lower, with $\mathcal{B}'$ generating recall values being higher more than several hundreds of percent. For this experiment, it is also evident that when $|\mathcal{S}| \geq q_\tau$, there is a significant rise in the recall, which we again attribute to the discussed assumption. Once again, the trend of diminishing recalls with increasing stream size is apparent in Fig. 8e.

6.5 Efficiency

State-of-the-art Comparison. We compared SuSE's efficiency with the state-of-the-art engine REmatch in Fig. 10. Given the inferior performance of other RegEx engines (see the results in [36]), along with their limitation in computing *all* matches of a subsequence, our experiments include solely REmatch.

In Fig. 10a, we tested with $q_\gamma = ABCD$ (the REmatch syntax being $!x\{A\}.*!y\{B\}.*!z\{C\}.*!w\{D\}$) for the required processing time of input words of varying sizes. As input, we choose words of the form $A^iB^iC^iD^i$ with $i \in \mathbf{N}^+$, e.g., $A^8B^8C^8D^8$, such that the number of resulting matches increases for increasing x-values.

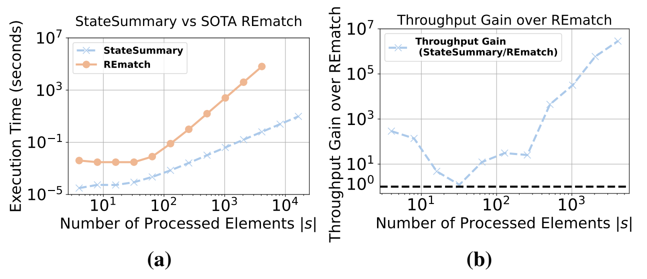


Figure 10: Comparison against the state-of-the-art RegEx engine REmatch, focusing on (a) execution time (lower is better) and (b) throughput gain (higher is better) when processing varying numbers of elements.

SuSE's processing time is consistently at least an order of magnitude less than REmatch, due to the StateSummary. Particularly for words exceeding 128 characters, the performance gap between SuSE and REmatch widens considerably. For instance, while REmatch takes nearly 10^3 seconds to process a 1024-character word, SuSE requires under 10^{-1} seconds. When the word size is 4096, REmatch's processing time increases to 10^5 seconds, whereas SuSE finishes in just 10^0 seconds.

The performance differences also induce throughput gains, as shown in Fig. 10b. The ratio indicates how much more quickly SuSE processed a word than REmatch. The throughput gain is always greater than one, indicating the superiority of SuSE. Notably, this gain consistently increases for word lengths beyond 32. At $x = 1024$, the throughput gain is close to five orders of magnitude, and this jumps to nearly seven orders of magnitude for word lengths of 4096, showcasing the efficiency of SuSE.

Overall Efficiency. In Fig. 9, we examined how varying $|\mathcal{S}|$ (see legend) and q_τ affects the execution time, throughput, processing latency per element, and memory usage, for processing a stream of 10^5 elements. Fig. 9a illustrates the average execution time for 3

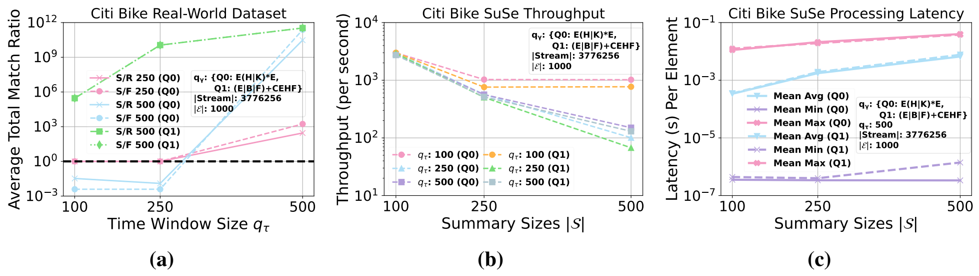


Figure 11: Citi Bike: (a) average total match ratio (higher is better), (b) throughput (higher is better), and (c) latency (lower is better).

runs to process the stream, whereby it can be seen that the larger $|\mathcal{S}|$ and the larger q_τ, the longer the execution time.

Also, a time window size exists for all procedures up to a summary size of 500, so that the execution time no longer increases but slightly decreases. We explain this as follows: as shown in §4.2, the REMOVE operation is the most expensive. Its cost is dictated by $min(q_\tau, |\mathcal{S}|)$. Once either $|\mathcal{S}| > q_\tau$ or $q_\tau > |\mathcal{S}|$, the smaller value of the two parameters becomes the limiting factor for runtime, rendering the influence of the other parameter's increase marginal. With larger q_τ, SUSE chooses better projections, which reduces the number of REMOVE operations and explains the drop in runtime for $q_\tau \geq |\mathcal{S}|$.

Besides the REMOVE operation, memory shifts primarily influence the execution time, which is reflected for configurations where $|\mathcal{S}| \geq 1000$ and $q_\tau = 500$. The complexity analysis does not consider such dimensions even if $q_\tau < |\mathcal{S}|$ holds. As $|\mathcal{S}|$ increases, the frequency of memory shifts rises, causing the overall execution time to rise; thus, in practice, the increasing size of $|\mathcal{S}|$ dictates the execution time.

In Fig. 9b, we examined the resulting throughput values for a fixed $q_\tau = 500$, the most computation-intensive window size we tested. For the same reasons as discussed above, for the execution time observations, there is a decline in throughput as the summary size expands. However, for summary sizes up to 5000, the average and median throughput consistently stay around 10^3. With smaller summary sizes, the throughput approaches 10^4.

In Fig. 9c, we investigated the resulting average minimum, maximum, and average processing latency per element per second. A boxplot contains the latencies that arose over increasing q_τ, while the corresponding line plots represent the respective average. A larger $\mathcal{S}$ leads to elevated processing latencies. However, the median of the average processing latencies consistently remains below $10^{-1}s$. The average maximum latency exhibits a similar but less pronounced trend, rising modestly with larger $|\mathcal{S}|$ and only marginally surpassing 10^{-1} seconds.

Finally, we examined the induced memory footprint depicted in Fig. 9d. Neither increasing $|\mathcal{S}|$ nor q_τ influences average maximum memory usage. With our fixed size of 128 bits for counters, the impact on memory from increasing $|\mathcal{S}|$ is negligible. Consequently, the STATESUMMARY maintains a modest memory footprint of 80MB while achieving compact state representation and summarization.

6.6 Case Study: Citi Bike

Character Types and Queries. Fig. 11 denotes our results for the Citi Bike real-world dataset. The character types encompass rides on frequented routes, pinpoint brief rides at busy stations, rides at central stations, and member rides at quieter stations, giving a thorough understanding of the Citi Bike dynamics. We defined a query with $Q0_\gamma = E(H|K)^*E$, recognizing patterns where rides originate from busy stations, transition through a combination of central and quieter stations, and then return to busy stations, suggesting probable areas for station enhancements or upkeep. The second query, $Q1_\gamma = (E|B|F)^+CEHF$, captures sequences of rides that start at busy stations, popular routes, or extended rides at central spots, followed by short and consecutive rides at busy stations, a trip at a central location, and concluding with a prolonged ride there, hinting at peak demand transitioning to central areas.

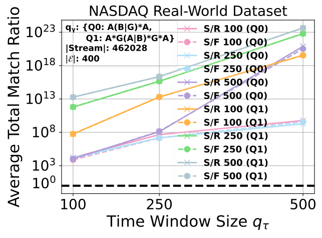
Effectiveness. To examine the average total match ratio, we varied $|\mathcal{S}|$ and q_τ ranging from $100 - 500$ in Fig. 11a. For both queries, not all summary sizes yielded matches. Generally, there is an upward trend in the results with a larger q_τ; SUSE identifies projections that are at least three orders and up to twelve orders of magnitude superior for $q_\tau = 500$. Furthermore, for $Q1$, both baselines discovered more matches at 100 and 250 time-window sizes than SUSE.

Throughput and Latency. We analyzed the throughput and processing latency of SUSE in Fig. 11b and Fig. 11c. For smaller values of $|\mathcal{S}|$ and q_τ (refer to legend), SUSE's throughput increases, processing up to 3000 elements per second. However, with larger values for $|\mathcal{S}|$ and q_τ, the lower throughput is attributed to the REMOVE operation and memory shift overheads.

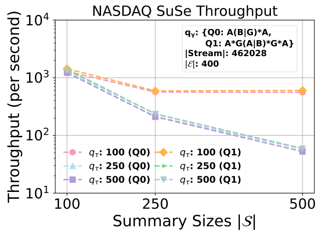
Fig. 11c shows the avg min, avg max, and avg processing latency of SUSE, with $q_\tau = 500$. An increase in summary size leads to higher latencies. Yet, the average maximum latency hovers around 10^{-2} seconds, while the average processing latency remains under 10^{-2} seconds for $|\mathcal{S}| = 500$, indicating real-time computing latencies.

6.7 Case Study: NASDAQ

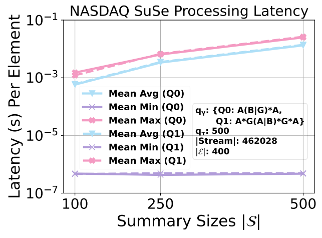
Character Types and Queries. Fig. 12 represents our results for the NASDAQ real-world dataset. We derived character types based on stock market activities, e.g., significant price changes, trade volumes, daily peak or lowest prices, periods of consistent behaviour or fluctuations, and market uncertainty. We formulated a query with $Q0_\gamma = A(B|G)^*A$, detecting an initial price increase, succeeded by any number of price reductions or uncertain market periods, and ending with a subsequent price surge, potentially signalling a fluctuating stock price ascent. The second query, $Q1_\gamma = A^*G(A|B)^*G^*A$, captures zero or more price rises, followed by market uncertainty, zero or more occurrences of either significant price rises or drops, a



(a)



(b)



(c)

Figure 12: NASDAQ: (a) average total match ratio (higher is better), (b) throughput (higher is better), and (c) latency (lower is better).

potential period of market uncertainty, and ends with a price rise, indicating a trend that, despite its volatility and periods of uncertainty, ends with a bullish behaviour.

Effectiveness. In Fig. 12a, we varied $|\mathcal{S}|$ and q_τ between $100-500$ using the $Q0$ and $Q1$ above. Here, for all $|\mathcal{S}|$, matches were obtained. Generally, larger $|\mathcal{S}|$ and q_τ led SuSe to derive superior stream projections, resulting in up to 10^{23} additional matches. However, even at smaller parameter values, SuSe's projections surpassed both baselines by magnitudes ranging from 10^4 to 10^{13}. Notably, for $Q1$, SuSe identified better projections due to the increased count of Kleene operators.

Throughput and Latency. In our examination of throughput and processing latency of SuSe, as shown in Fig. 12b and Fig. 12c, a similar trend as for Citi Bike emerges: when decreasing the summary and time-window size (see legend), SuSe's throughput enhances, reaching peaks of up to 1500 elements per second. However, when the $|\mathcal{S}|$ and q_τ increase, the throughput decreases to around $70-80$ elements.

Turning to Fig. 12c, we fixed $q_\tau = 500$ and increased the summary size, resulting in an upward trend of the latencies. Nevertheless, while the average maximum latency approaches 10^{-2} seconds, the average processing latency for $\mathcal{S} = 500$ stays just below 10^{-2} seconds, again denoting latencies suitable for real-time computing.

7 RELATED WORK

Subsequences. Subsequences are crucial tools in fields like biological sequence analysis [34] and event stream processing [4, 15, 43]. They are associated with algorithmic challenges including the longest common subsequence [5], shortest common supersequence [30], or subsequence counting [12], and are studied in contexts of gap-size [10] and wildcard constraints [21, 22]. However, obtaining aggregated information about subsequence *mappings* within a text has barely been researched. Solely in [12] and [16], dynamic programming algorithms are presented which for a given subsequence u and a text x count the number of mappings $u \preceq_m x$. We complement their work using a more holistic approach: We provide aggregated information for a set of patterns derived from a regular expression γ and simultaneously account for partial mappings from $L_{\text{pre}}(\gamma)$. While prior works assume static text, SuSe processes (valued) character streams, which require online computation.

Complex Event Processing (CEP). CEP facilitates reactive programming by analyzing queries over event streams. CEP queries primarily utilize regular expressions, enabling pattern recognition against incoming events. CEP engines like FlinkCEP [14] or CORE [6] incorporate the *skip-till any match* event selection strategy [2]. This strategy permits skipping characters during the match process, allowing for subsequence search. Nonetheless, a challenge arises as these systems adopt an automata-based approach for query matching, leading to exponential state growth (i.e., matches). In response, Poppe et al. proposed methods for event trend aggregation like GRETA [32] and successors [29, 31, 33, 37], which were already discussed earlier.

CEP optimizations. Various strategies address system overload, including load shedding techniques [7, 38, 44]. These techniques selectively discard events or remove partial matches unlikely to complete, reducing information loss while decreasing overload. Similarly, filtering methods [3] transform a stream s into a filtered stream s', retaining events more likely to lead to complete matches and omitting others. SuSe also functions as a filter, with its selected projection representing stream s'. It resembles these methods in (1) state-based decisions, (2) reducing system load, and (3) minimizing information loss. Yet, SuSe bases decisions on the StateSummary without materializing the state, evading the exponential complexity of automata-based evaluation, and its decisions, unlike [3, 44], are not reliant on a learned model, enhancing streaming practicality.

8 CONCLUSIONS

We proposed SuSe, an architecture for regular expression subsequence summarization over data streams. It incorporates a StateSummary data structure, capturing a query-specific stream summary through maintaining aggregated subsequence match information. We presented a summary selection algorithm, leveraging the StateSummary for choosing stream projections, which aim to minimize the loss in the aggregated results over time. Our evaluation on real-world and synthetic data validates the efficiency and effectiveness of SuSe: it processes inputs by multiple orders of magnitude faster than leading RegEx engines while generating aggregates based on substantially richer stream projections than baseline approaches.

REFERENCES

[1] Oniguruma – a modern and flexible regular expressions library. 2022. https://github.com/kkos/oniguruma Accessed on 2023-10-15.

[2] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. 2008. Efficient pattern matching over event streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, Jason Tsong-Li Wang (Ed.). ACM, 147–160. https://doi.org/10.1145/1376616.1376634

[3] Adar Amir, Ilya Kolchinsky, and Assaf Schuster. 2022. DLACEP: A Deep-Learning Based Framework for Approximate Complex Event Processing. In *Proceedings of the 2022 International Conference on Management of Data*. ACM. https://doi.org/10.1145/3514221.3526136

[4] Alexander Artikis, Alessandro Margara, Martín Ugarte, Stijn Vansummeren, and Matthias Weidlich. 2017. Complex Event Recognition Languages: Tutorial. In

Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017, Barcelona, Spain, June 19-23, 2017. ACM, 7–10. https://doi.org/10.1145/3093742.3095106

[5] Ricardo A. Baeza-Yates. 1991. Searching subsequences. *Theoretical Computer Science* 78, 2 (Jan. 1991), 363–376. https://doi.org/10.1016/0304-3975(91)90358-9

[6] Marco Bucchi, Alejandro Grez, Andrés Quintana, Cristian Riveros, and Stijn Vansummeren. 2022. CORE: a complex event recognition engine. *Proceedings of the VLDB Endowment* 15, 9 (May 2022), 1951–1964. https://doi.org/10.14778/3538598.3538615

[7] Koral Chapnik, Ilya Kolchinsky, and Assaf Schuster. 2021. DARLING: Data-Aware Load Shedding in Complex Event Processing Systems. *Proc. VLDB Endow.* 15, 3 (2021), 541–554. http://www.vldb.org/pvldb/vol15/p541-chapnik.pdf

[8] Hao Chen, Yu Chen, and Douglas H. Summerville. 2011. A Survey on the Application of FPGAs for Network Infrastructure Security. *IEEE Communications Surveys &amp Tutorials* 13, 4 (2011), 541–561. https://doi.org/10.1109/surv.2011.072210.00075

[9] citi Bike. 2022. http://www.citibikenyc.com/system-data..

[10] Joel D. Day, Maria Kosche, Florin Manea, and Markus L. Schmid. 2022. Subsequences with Gap Constraints: Complexity Bounds for Matching and Analysis Problems. In *33rd International Symposium on Algorithms and Computation (ISAAC 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 248)*, Sang Won Bae and Heejin Park (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 64:1–64:18. https://doi.org/10.4230/LIPIcs.ISAAC.2022.64

[11] Bartłomiej Dudek, Paweł Gawrychowski, Garance Gourdel, and Tatiana Starikovskaya. 2022. Streaming Regular Expression Membership and Pattern Matching. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Society for Industrial and Applied Mathematics, 670–694. https://doi.org/10.1137/1.9781611977073.30

[12] Cees Elzinga, Sven Rahmann, and Hui Wang. 2008. Algorithms for subsequence combinatorics. *Theoretical Computer Science* 409, 3 (Dec. 2008), 394–404. https://doi.org/10.1016/j.tcs.2008.08.035

[13] PCRE2 – Perl-Compatible Regular Expressions. 2022. https://github.com/PCRE2Project/pcre2 Accessed on 2023-10-15.

[14] Apache Software Foundation. 2021. Apache Flink. https://nightlies.apache.org/flink/flink-docs-release-1.14/ Accessed on 2023-10-15.

[15] Nikos Giatrakos, Elias Alevizos, Alexander Artikis, Antonios Deligiannakis, and Minos N. Garofalakis. 2020. Complex event recognition in the Big Data era: a survey. *VLDB J.* 29, 1 (2020), 313–352. https://doi.org/10.1007/s00778-019-00557-w

[16] Ronald I. Greenberg. 2003. Computing the Number of Longest Common Subsequences. arXiv:arXiv:cs/0301034

[17] PCREgrep – A grep program that uses the PCRE regular expression library. 2014. https://github.com/vmg/pcre/blob/master/pcregrep.c/ Accessed on 2023-10-15.

[18] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2001. Introduction to automata theory, languages, and computation, 2nd edition. *ACM SIGACT News* 32, 1 (March 2001), 60–65. https://doi.org/10.1145/568438.568455

[19] Theodore Johnson, S. Muthukrishnan, and Irina Rozenbaum. 2007. Monitoring Regular Expressions on Out-of-Order Streams. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE. https://doi.org/10.1109/icde.2007.369001

[20] Stephen C. Kleene. 1951. *Representation of events in nerve nets and finite automata*. Technical Report. RAND Corporation, Santa Monica, CA.

[21] Sarah Kleest-Meißner, Rebecca Sattler, Markus L. Schmid, Nicole Schweikardt, and Matthias Weidlich. 2022. Discovering Event Queries from Traces: Laying Foundations for Subsequence-Queries with Wildcards and Gap-Size Constraints. In *25th International Conference on Database Theory, ICDT 2022, March 29 to April 1, 2022, Edinburgh, UK (Virtual Conference) (LIPIcs, Vol. 220)*, Dan Olteanu and Nils Vortmeier (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 18:1–18:21. https://doi.org/10.4230/LIPIcs.ICDT.2022.18

[22] Sarah Kleest-Meißner, Rebecca Sattler, Markus L. Schmid, Nicole Schweikardt, and Matthias Weidlich. 2023. Discovering Multi-Dimensional Subsequence Queries from Traces - From Theory to Practice. In *Datenbanksysteme für Business, Technologie und Web (BTW 2023), 20. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 06.-10, März 2023, Dresden, Germany, Proceedings (LNI, Vol. P-331)*, Birgitta König-Ries, Stefanie Scherzinger, Wolfgang Lehner, and Gottfried Vossen (Eds.). Gesellschaft für Informatik e.V., 511–533. https://doi.org/10.18420/BTW2023-24

[23] Sailesh Kumar, Balakrishnan Chandrasekaran, Jonathan Turner, and George Varghese. 2007. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*. ACM. https://doi.org/10.1145/1323548.1323574

[24] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. 2006. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM. https://doi.org/10.1145/1159913.1159952

[25] Boost Regex Library. 2022. https://github.com/boostorg/regex Accessed on 2023-10-15.

[26] NASDAQ Data Link. 2023. https://data.nasdaq.com/ Accessed: 2023-10-15.

[27] Alex X. Liu and Eric Norige. 2019. A De-Compositional Approach to Regular Expression Matching for Network Security. *IEEE/ACM Transactions on Networking* 27, 6 (Dec. 2019), 2179–2191. https://doi.org/10.1109/tnet.2019.2941920

[28] Tingwen Liu, Yong Sun, Alex X. Liu, Li Guo, and Binxing Fang. 2012. A Prefiltering Approach to Regular Expression Matching for Network Security Systems. In *Applied Cryptography and Network Security*. Springer Berlin Heidelberg, 363–380. https://doi.org/10.1007/978-3-642-31284-7_22

[29] Lei Ma, Chuan Lei, Olga Poppe, and Elke A. Rundensteiner. 2022. Gloria: Graph-based Sharing Optimizer for Event Trend Aggregation. In *Proceedings of the 2022 International Conference on Management of Data*. ACM. https://doi.org/10.1145/3514221.3526145

[30] David Maier. 1978. The Complexity of Some Problems on Subsequences and Supersequences. *Journal of the ACM* 25, 2 (April 1978), 322–336. https://doi.org/10.1145/322063.322075

[31] Olga Poppe, Chuan Lei, Lei Ma, Allison Rozet, and Elke A. Rundensteiner. 2021. To Share, or not to Share Online Event Trend Aggregation Over Bursty Event Streams. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1452–1464. https://doi.org/10.1145/3448016.3452785

[32] Olga Poppe, Chuan Lei, Elke A. Rundensteiner, and David Maier. 2017. GRETA: graph-based real-time event trend aggregation. *Proceedings of the VLDB Endowment* 11, 1 (Sept. 2017), 80–92. https://doi.org/10.14778/3151113.3151120

[33] Olga Poppe, Chuan Lei, Elke A. Rundensteiner, and David Maier. 2019. Event Trend Aggregation Under Rich Event Matching Semantics. In *Proceedings of the 2019 International Conference on Management of Data*. ACM. https://doi.org/10.1145/3299869.3319862

[34] Sven Rahmann. 2006. Subsequence Combinatorics and Applications to Microarray Production, DNA Sequencing and Chaining Algorithms. In *Combinatorial Pattern Matching*. Springer Berlin Heidelberg, 153–164. https://doi.org/10.1007/11780441_15

[35] RE2 regular expression library. 2022. https://github.com/google/re2 Accessed on 2023-10-15.

[36] Cristian Riveros, Nicolás Van Sint Jan, and Domagoj Vrgoč. 2023. REmatch: A Novel Regex Engine for Finding All Matches. *Proceedings of the VLDB Endowment* 16, 11 (July 2023), 2792–2804. https://doi.org/10.14778/3611479.3611488

[37] Allison Rozet, Olga Poppe, Chuan Lei, and Elke A. Rundensteiner. 2020. Muse: Multi-query Event Trend Aggregation. In *Proceedings of the 29th ACM International Conference on Information &amp Knowledge Management*. ACM. https://doi.org/10.1145/3340531.3412138

[38] Ahmad Slo, Sukanya Bhowmik, and Kurt Rothermel. 2020. hSPICE: state-aware event shedding in complex event processing. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*. ACM. https://doi.org/10.1145/3401025.3401742

[39] Peter Snyder and Chris Kanich. 2015. No Please, After You: Detecting Fraud in Affiliate Marketing Networks.. In *WEIS*.

[40] Ken Thompson. 1968. Programming Techniques: Regular expression search algorithm. *Commun. ACM* 11, 6 (June 1968), 419–422. https://doi.org/10.1145/363347.363387

[41] robust TRE – a lightweight and efficient POSIX compliant regexp matching library. 2021. https://github.com/laurikari/tre Accessed on 2023-10-15.

[42] Mohamed Zaki and Babis Theodoulidis. 2013. Analyzing Financial Fraud Cases Using a Linguistics-Based Text Mining Approach. *SSRN Electronic Journal* (2013). https://doi.org/10.2139/ssrn.2353834

[43] Haopeng Zhang, Yanlei Diao, and Neil Immerman. 2014. On complexity and optimization of expensive queries in complex event processing. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 217–228. https://doi.org/10.1145/2588555.2593671

[44] Bo Zhao, Nguyen Quoc Viet Hung, and Matthias Weidlich. 2020. Load Shedding for Complex Event Processing: Input-based and State-based Techniques. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 1093–1104. https://doi.org/10.1109/ICDE48307.2020.00099