SuSe: Summary Selection for Regular Expression Subsequence Aggregation over Streams

Technical Report

Anonymized Author(s)

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 StateSummary, 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 Author(s) . 2018. SuSe: Summary Selection for Regular Expression Subsequence Aggregation over Streams: Technical Report. 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

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 [40, 44], or infrastructure security [9, 23]. RegEx have been studied extensively in theoretical computer science [18, 20] and automata-based approaches for their evaluation, most prominently the Thompson construction [41] and its derivates, have been around for decades.

However, existing RegEx engines, such as PCRE2 [13], pcregrep [17], Oniguruma [1], Boost.Regex [25], RE2 [36], and TRE [42] are limited in their ability to consider subsequences during evaluation: First, they lack native support for skipping characters in a sequence when constructing matches. This is not a matter of expressiveness, though, 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. However, such explicit encoding significantly increases the per-match state and changes the underlying language. Second, they do not support enumerating *all* matches, allowing only greedy or lazy matching that returns a single match. REmatch [37] is a notable exception in that it supports the enumeration of *all* matches; yet, it suffers from performance issues due to the exponential growth of subsequence matches with input size, as we demonstrate later empirically.

Complex Event Processing (CEP) engines like FlinkCEP [14] allow for subsequence matching using automata-based methods. They evaluate sequence queries grounded in regular expressions, enhanced with predicates and time window constraints. However, these engines also face performance issues caused by the need to manage exponentially many matches. The CORE engine [7] seeks to address this by representing matches in an efficient data structure. Still, it exhaustively enumerates matches, leading to significant performance bottlenecks, as demonstrated by our experiments.

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

(1) Many 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 are 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 [33] and its successors [29, 32, 34, 38].
(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 an external trigger that determines when the result is relevant. For instance, users may load an analysis report, open a dashboard, or refresh a visualization at certain, generally unknown time points.

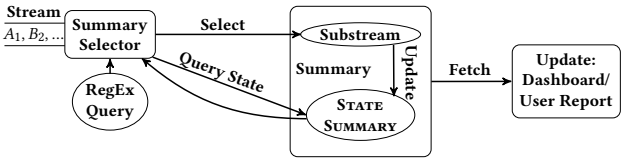


Figure 1: Architecture for RegEx evaluation over streams.

Combining these two observations, we present SuSe, a novel architecture for RegEx evaluation. As shown in Fig. 1, it incrementally maintains a query-specific summary of the stream that includes a substream of selected elements along with aggregated information about query matches. This summary provides an approximate answer, which is generally more accurate than existing evaluation strategies, and which can be fetched at any point in time. From a technical point of view, 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 provide a motivating example (§2) and introduce preliminaries (§3), before giving a formal problem statement (§4). Then, we define the StateSummary (§5), as the basis for summary selection (§6). We present evaluation results (§7), demonstrating that SuSe computes aggregates based on several orders of magnitude more matches than baseline techniques. Finally, we review related work (§8) and conclude the paper (§9).

2 MOTIVATING EXAMPLE

Consider a bike rental service that aims to enhance operational efficiency by analyzing patterns within trip data streams, e.g., targeting patterns of events denoting short trips in busy areas. These patterns, indicative of user behavior, are defined using regular expressions. The inherent variability and complexity of real-world data streams, where a strictly contiguous occurrence of events within a pattern may be rare or too restrictive, motivates the search for subsequences, i.e., events in a pattern can occur in a non-contiguous order in the stream. Furthermore, streams are partitioned by attributes, e.g., bike ID or start location, to facilitate targeted analysis.

To inform operational decision making, a dashboard gives an overview of the current number of patterns, i.e., it derives a simple aggregate over the matches of a query that describes the relevant trips. Whenever the dashboard is refreshed, at unknown time points, the aggregate is updated and a subsequence of the stream that induced many pattern matches is made available for a quick plausibility assessment. Here, it is fine to trade a minor information loss due to approximate results for increased efficiency, as decision making is based on general trends rather than the exact results.

Fig. 2 showcases three strategies for retaining stream subsequences: Random, First-In-First-Out (FIFO), and SuSe. Given is a stream s, a summary size n, and a regular expression $\gamma = AB^*C$.

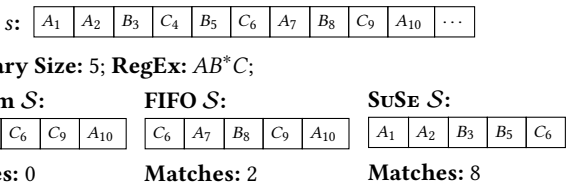


Figure 2: Comparison of random, First-In-First-Out (FIFO), and our SuSe summary selection strategy.

Assuming that all events of s fall within the time window of the RegEx query, we aim to select a subsequence $\mathcal{S}$ of s (i.e., an order-preserving selection of elements) no larger than n, that maximizes the count of subsequence matches upon evaluating the RegEx against the summary $\mathcal{S}$. For the stream in Fig. 2, we compare the strategies at the point in time after the event A_{10} was processed.

Random. A first baseline strategy randomly selects an element for eviction when appending a new one, once the summary size n is reached. This way, a subsequence was chosen which produces zero matches upon evaluating γ against $\mathcal{S}$, wasting the entire space.

FIFO. The FIFO strategy always keeps the latest n elements in the summary $\mathcal{S}$. Evaluating γ against $\mathcal{S}$ generates two matches, i.e., A_7C_9 and $A_7B_8C_9$. However, keeping C_6 in the summary, which cannot contribute to any matches, illustrates inefficient space usage.

SuSe. Using the SuSe framework presented in this work, we would arrive at a more effective stream summary resulting in eight matches: A_1C_6; A_2C_6; $A_1B_3C_6$; $A_2B_3C_6$; $A_1B_5C_6$; $A_2B_5C_6$; $A_1B_3B_5C_6$; and $A_2B_3B_5C_6$. Yet, the answer is still approximate (missing, for instance, matches with event B_8 of the stream).

3 PRELIMINARIES

3.1 Regular Expressions

Languages. An *alphabet* $\Sigma = \{a_1, \ldots, a_m\}$ is a set of *characters*. A *word* is a sequence of characters $w = \langle c_1, \ldots, c_n \rangle$ of size $n = |w|$ over Σ, i.e., $c_i \in \Sigma$. The character at the i-th index in word w is denoted by $w[i]$, for $1 \leq i \leq |w|$. 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 of length $\leq n$ that starts from its first character.

A *subsequence* y of a word $w = \langle c_1, \ldots, c_n \rangle$, denoted by $y \preceq w$, is obtained by deleting zero or more characters from w without altering the order of the remaining characters. There are 2^n possible subsequences of w. Formally, a subsequence y is represented as $y = \langle c_{i_1}, \ldots, c_{i_k} \rangle$, where $k \leq n$ and $1 \leq i_1 \leq \ldots \leq i_k \leq n$.

To establish the relation between the indices in y and w, a *mapping function* $m\colon \{1, \ldots, k\} \to \{1, \ldots, n\}$ is defined such that $j \mapsto i_j$. This function maps the k indices of y to the corresponding indices in 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 pm. Note that multiple (partial) mapping functions may exist from a subsequence to a word.

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: we have* $y \preceq_{m_1} w$ *with* $m_1\colon 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 4$*; and* $y \preceq_{m_2} w$ *with* $m_2\colon 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 7$.

The set of all words of length $n \in \mathbb{N}_0$ over an alphabet Σ is Σ^n; and the set of all possible words is $\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$. A *language* L over Σ is any subset $L \subseteq \Sigma^*$, and the set of all prefixes of L is denoted as L_{pre}. The *empty word* is $\varepsilon = \Sigma^0$.

Let L_1 and L_2 be languages defined over an alphabet Σ. Then, we define concatenation, union, and Kleene star for languages as:

- $L_1 L_2 = \{x.y \mid x \in L_1, y \in L_2\}$,
- $L_1 \cup L_2 = \{x \in \Sigma^* \mid x \in L_1 \vee x \in L_2\}$, and
- $L_1^* = \bigcup_{n \geq 0} L_1^n$.

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

- the empty language $L(\varnothing) = \varnothing$;
- 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 $L((\alpha)^*) = L(\alpha)^*$.

3.2 Regular Expression Queries over Streams

Natural numbers represent time points, denoted as $\mathbb{T} \subseteq \mathbb{N}$. A *stream* $s = (\omega = \langle c_1, c_2, \ldots \rangle, \lambda = \langle t_1, t_2, \ldots \rangle, \upsilon = \langle v_1, v_2, \ldots \rangle)$ comprises sequences of characters $c_i \in \Sigma$ (i.e., ω), timestamps $t_i \in \mathbb{T}$ (i.e., λ), and values $v_i \in \mathbb{N}$ (i.e., υ). They induce stream elements $e_i = (\omega[i], \lambda[i], \upsilon[i])$ for $1 \leq i$, with $\omega[i]$ (or c_i), $\lambda[i]$ (or t_i), and $\upsilon[i]$ (or v_i) being the i-th elements of the respective sequences. The stream is assumed to be totally ordered by timestamps, i.e., $t_i < t_j$ for $1 \leq i < j$, which, in practice, may be achieved by techniques for out-of-order handling and tie-breaking mechanisms [31, 43]. The value v_i models a payload of e_i and could also be defined over a different domain, depending on the application scenario. Let $t \in \mathbb{T}$ be a time instance. A *substream* $s(t) = (\langle c_1, \ldots, c_p \rangle, \langle t_1, \ldots, t_p \rangle, \langle v_1, \ldots, v_p \rangle)$ is the stream up to index p, containing only elements with timestamps $\leq t$.

A *regular expression query* $q = (\gamma, \tau, \mathscr{A})$ is a triple, comprising a regular expression γ (also q_γ), a time window $\tau \in \mathbb{T}$ (also q_τ), and an aggregate function $\mathscr{A}$ (also $q_{\mathscr{A}}$). A mapping $y \preceq_m \omega$ of a word $y \in L(\gamma)$ to ω is a *match* of a regular expression query q if the time window constraint is satisfied, i.e., $\lambda[m(k)] - \lambda[m(1)] \leq q_\tau$.

Example 2. *For $\Sigma = \{A, B, C\}$, consider the stream $s = (\omega, \lambda, \upsilon)$ with $\omega = \langle A_1, B_2, C_3, C_4 \rangle$ and $\lambda = \langle 1, 5, 6, 9 \rangle$. For the RegEx $q_\gamma = ABC$, two mappings between the only word $\langle A_1, B_2, C_3 \rangle$ in $L(q_\gamma)$ and ω exist: $m_1 \colon 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3$ and $m_2 \colon 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 4$. Yet, with a time window $q_\tau = 5$, only m_1 is a match of the RegEx query.*

A *partial match* occurs when a word $y \in L_{\text{pre}}(\gamma)$ maps to indices of ω while satisfying the time window constraint. It is a candidate for becoming a complete match as more characters of ω are examined. Notably, a complete match can also serve as a partial match. For instance, for $\gamma = A(B|C)D^*$, a match of the word $\langle A_1, B_2, D_3 \rangle \in L(\gamma)$ is also a partial match of $\langle A_1, B_2, D_3, D_4 \rangle \in L(\gamma)$.

Our work utilizes a streaming model of computation, in which a RegEx query $q = (\gamma, \tau, \mathscr{A})$ is received and preprocessed. We focus on the aggregate functions $\mathscr{A} \in \{\text{Count}, \text{Sum}, \text{Avg}\}$, which are defined over the matches of q. Count calculates the number of matches, Sum adds up the payload values of elements from all complete matches, and Avg is the ratio of Sum to Count.

Table 1: Overview of notations for RegEx queries.

Notation	Explanation
$\Sigma = \{a_1, \ldots, a_m\}$	Alphabet containing characters.
$w = \langle c_1, \ldots, c_n \rangle$	Word (sequence of characters) of size $n = \lvert w \rvert$.
$w[i]$	Character at i-th index of w.
$y \preceq_m w$	A subsequence y of w with a mapping m.
$q = (\gamma, \tau, \mathscr{A})$	Regular expression query q, composed of regular expression γ, a time window size τ, and an aggregate function $\mathscr{A}$.
$s = (\omega = \langle c_1, \ldots \rangle,$ $\lambda = \langle t_1, \ldots \rangle,$ $\upsilon = \langle v_1, \ldots \rangle)$	A *stream* of related sequences of characters $c_i \in \Sigma$ (i.e., ω), timestamps $t_i \in \mathbb{T}$ (i.e., λ), and values $v_i \in \mathbb{N}$ (i.e., υ), respectively.
$L(\gamma), L_{\text{pre}}(\gamma)$	The languages of *all words* and *all prefixes* of γ, respectively.
PM, CM	The sets of partial and complete matches, respectively, resulting from evaluating q over the stream s.

3.3 Regular Expression Query Matching

Automata-Based Matching. Traditional matching of a regular expression q_γ against a stream (word) relies on a Non-deterministic Finite Automaton (NFA). 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, $\delta \colon Z \times \Sigma \to 2^Z$ as the transition function, and $F \subseteq Z$ as the set of final states. The NFA for q_γ is obtained by the *Thompson construction* [41] that recursively composes NFAs of subexpressions, such that all words in $L(q_\gamma)$ are accepted.

As motivated, we aim to detect *all possible (partial) matches of words from $L(q_\gamma)$ to indices in a word w*. This mirrors the most challenging combination of consumption and selection policies [2, 15, 45] in event processing: *Reuse* and *Skip-Till-Any-Match (STAM)*. A simple automata-based matching method works as follows: Each incoming element is checked against the current set of automata. If an element enables a transition in an automaton, it is duplicated, with one copy disregarding the transition and the other one taking it, yielding an exponential growth of the number of automata in the number of processed elements.

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 stream $s = (\omega, \lambda, \upsilon)$ using two sets: complete matches CM and partial matches PM. These sets are based on the language $L(q_\gamma)$ and its prefix language $L_{\text{pre}}(q_\gamma)$, respectively. PM includes a partial match if it maps a word $y \in L_{\text{pre}}(q_\gamma)$ to indices in ω. CM includes a match if it maps a word $y \in L(q_\gamma)$ to indices in ω. Either way, the mapping must satisfy the time window q_τ.

Next, we formalize the construction of CM, while the construction of PM works analogously. Let i be the index of the last received stream element e_i and $[i] = \{1, \ldots, i\}$. We define $\mathcal{M}_{\text{CM}} = \{\varphi \mid \varphi \colon L(q_\gamma) \to 2^{[i]}\}$ as the set of all *word mapping functions* φ from words in $L(q_\gamma)$ to sets of up to i indices in ω. Note that the mapping function m introduced earlier can be depicted through the word mapping function φ_m, e.g., in Example 2, m_1 aligns with $\varphi_{m_1}(\langle A_1, B_2, C_3 \rangle) = \{1, 2, 3\}$. Analogously, the set of all *partial word mapping functions*, $\mathcal{M}_{\text{PM}} = \{\varphi' \mid \varphi' \colon L_{\text{pre}}(q_\gamma) \setminus L(q_\gamma) \to 2^{[i]}\}$, is based on $L_{\text{pre}}(q_\gamma)$.

A word mapping function φ for a word $y \in L(q_\gamma)$, represented by $\varphi(y) = \{i_1, i_2, \ldots, i_{|y|}\}$ with $i_1 < i_2 < \ldots < i_{|y|}$, is deemed *valid* if it satisfies the following conditions:

(1) (Reconstruction) The indices $\varphi(y) = \{i_1, i_2, \ldots, i_{|y|}\}$ in ω must reconstruct $y \in L(q_\gamma)$, i.e., $y = \omega[i_1]\omega[i_2] \ldots \omega[i_{|y|}]$.

(2) (Time Window Constraint) The mapping must adhere the time window constraint, i.e., $\lambda[i_{|y|}] - \lambda[i_1] \leq q_\tau$.

Based thereon, we define the set $CM = \{\varphi \mid \varphi \in \mathcal{M}_{\text{CM}}, \varphi \text{ is valid}\}$ as all valid word mapping functions φ. Analogously, PM denotes the set of all valid partial word mapping functions φ'.

Finally, using the set of complete matches as CM, we integrate the aggregate function $\mathscr{A}$. With $q_{\mathscr{A}} = \textsc{Count}$, the result is $|CM|$, i.e., the total number of complete matches. If $q_{\mathscr{A}} = \textsc{Sum}$, the result is $\sum_{\varphi \in CM} \sum_{j \in \text{range}(\varphi)} v[j]$, summing values from v of complete matches. For $q_{\mathscr{A}} = \textsc{Avg}$, the result is $\textsc{Sum}(CM)/\textsc{Count}(CM)$.

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 of q_γ is $L(q_\gamma) = \{\langle A_1, B_2, C_3\rangle\}$. For the stream $s = (\omega, \lambda, v)$, with $\omega = \langle A_1, C_2, B_3, A_4, C_5, B_6, C_7\rangle$, $\lambda = \langle 1, 2, 3, 4, 5, 6, 7\rangle$, and values $v = \langle 2, 3, 5, 6, 13, 7, 9\rangle$, potential matches are $\{\varphi_1 : \langle A_1, B_2, C_3\rangle \mapsto \{1, 3, 5\}, \varphi_2 : \langle A_1, B_2, C_3\rangle \mapsto \{1, 3, 7\}, \varphi_3 : \langle A_1, B_2, C_3\rangle \mapsto \{4, 6, 7\}\}$. Incorporating the time window q_τ, we obtain $CM = \{\varphi_1 : \langle A_1, B_2, C_3\rangle \mapsto \{1, 3, 5\}, \varphi_3 : \langle A_1, B_2, C_3\rangle \mapsto \{4, 6, 7\}\}$. For some aggregations $q_{\mathscr{A}}$, we get $\textsc{Count}(CM) = 2$ and $\textsc{Sum}(CM) = (2+5+13)+(6+7+9) = 42$.*

4 PROBLEM STATEMENT

We consider scenarios, in which the query evaluation is relevant at a random time instance (e.g., when a user refreshes a dashboard). 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*, which are drawn independently as a fixed number N of samples from $\mathbb{T}$ according to P_U, yielding $\mathcal{E} = \{\varepsilon_1, \ldots, \varepsilon_N\}$. Each $\varepsilon_i \in \mathcal{E}$ denotes a time instance when query results are relevant.

Then, we define a summary representation $\mathcal{S}$ as a subsequence of elements from the substream $s(\varepsilon)$ up to time $\varepsilon \in \mathcal{E}$, containing at most n elements, as $\mathcal{S}(\varepsilon, n) = (\langle c_{i_1}, \ldots, c_{i_k}\rangle, \langle t_{i_1}, \ldots, t_{i_k}\rangle, \langle v_{i_1}, \ldots, v_{i_k}\rangle)$ where $1 \leq i_1 \leq \cdots \leq i_k \leq p$, $k \leq n$, and each i_j is an index of a stream element in $s(\varepsilon)$. Here, p is the index of the latest stream element with timestamp $\leq \varepsilon$. Technically, the summary representation is constructed by a summary selection function that decides whether an element should be kept, discarded, or replaced.

We aim to assess how the projected elements in a summary affect a loss function $l_{\mathscr{A}} : \mathbb{N} \times \mathbb{N} \to \mathbb{R}$ upon an evaluation trigger. The loss quantifies the discrepancy between the values x_{opt} and x, obtained by the aggregate for query matches over either the stream $s(\varepsilon)$ (for x_{opt}) or the chosen subsequence $\mathcal{S}(\varepsilon, n)$ (for x) at time ε.

For aggregations Count and Sum, we employ the loss function $l_{\textsc{Count}}(x, x_{\text{opt}}) = l_{\textsc{Sum}}(x, x_{\text{opt}}) = |x - x_{\text{opt}}|$. For Avg, we define the loss as the relative error $l_{\textsc{Avg}}(x, x_{\text{opt}}) = |\frac{x - x_{\text{opt}}}{x_{\text{opt}}}|$. For each evaluation timestamp $\varepsilon \in \mathcal{E}$, we aim to minimize the loss $l_{\mathscr{A}}$.

Problem 1. *Let Σ be an alphabet and $s(\varepsilon) = (\omega, \lambda, v)$ be a stream up to time instance $\varepsilon \in \mathbb{T}$. Also, let $q = (\gamma, \tau, \mathscr{A})$ be a regular expression query, n be the summary size, $\mathcal{S}(\varepsilon, n)$ the summary representation, and $\mathcal{E} \subseteq \mathbb{T}$ a finite set of evaluation timestamps. For each time point $\varepsilon \in \mathcal{E}$, we aim at a stream summary $\mathcal{S}(\varepsilon, n)$ with minimal loss:*

$$\text{minimize} \sum_{\varepsilon \in \mathcal{E}} \min_{\mathcal{S}(\varepsilon,n)} (l_{\mathscr{A}}(q_{\mathscr{A}}(CM_{\mathcal{S}(\varepsilon,n)}), q_{\mathscr{A}}(CM_{s(\varepsilon)})))$$

5 STATE SUMMARY DATA STRUCTURE

The traditional evaluation of regular expression queries over streams poses challenges for the problem introduced in §4. The number of automata to maintain may grow exponentially in the number of processed elements, which yields an exponential runtime of an evaluation algorithm. Yet, many matches are materialized only to be discarded before the occurrence of an evaluation trigger, which wastes computational resources.

In the SuSe framework, we therefore approach the evaluation of regular expression queries with a stream summary, as shown already in Fig. 1. 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 SuSe is a stream summary and the StateSummary data structure. In the remainder, we fix a time instance $\varepsilon \in \mathbb{T}$ and summary size n, and, thus, write $\mathcal{S}$ to represent the stream summary. Let $s = (\omega, \lambda, v)$ be a stream; the stream summary $\mathcal{S} = (\omega', \lambda', v')$ is a subsequence of its elements. Given a regular expression query q, the StateSummary stores (i) aggregated information about partial and complete matches from evaluating q over the elements in $\mathcal{S}$ (i.e., *global state counters*); and (ii) information on the contribution of each element in $\mathcal{S}$ to the matches (i.e., *local state counters*). Moreover, this information is kept for an *active time window* that denotes a temporal context.

To simplify the presentation, we first introduce the StateSummary for queries using Count as the aggregation function, in terms of its construction (§5.1) and maintenance (§5.2). Then, we discuss how it is adapted for Sum and Avg aggregations (§5.3).

5.1 StateSummary Compilation

Given a regular expression query $q = (\gamma, \tau, \mathscr{A})$, the StateSummary is constructed based on the NFA for q_γ obtained by the Thompson construction [41]. 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 $\mathcal{N} = (Z, \Sigma, \delta, z_0, F)$, is used to compile our StateSummary. The data structure includes state counters (summarized in Table 2) that are defined and derived as follows.

Global State Counter. For each state $z_i \in Z$ in $\mathcal{N}$, where $0 \leq i < |Z|$, 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 elements $e_i = (\omega'[i], \lambda'[i], v'[i])$ in $\mathcal{S}$. Initially, these counters are set to zero. Let $\mathcal{S}_{\#Z} = (\#z_0, \ldots, \#z_{|Z|})$ denote the tuple of *global state counters* of $\mathcal{S}$, derived from the states Z of the compiled NFA, where each component $\#z_i$ in $\mathcal{S}_{\#Z}$ denotes a global state counter. Referring to the example given in Fig. 3, with global state counters $(\#z_0, \#z_1, \#z_2, \#z_3)$, processing the stream s yields 8 partial matches in z_1, 4 partial matches in z_2, and 10 complete matches in z_3, represented by the global state counters $(0, 8, 4, 10)$.

Count Rules. The Count rules specify the number of (partial) matches in a given state, based on the evaluation of query q over the selected elements in $\mathcal{S}$. 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. 3, where $\delta(z_1, C) = \{z_1\}$, while $\delta(z_1, A) = \emptyset$.

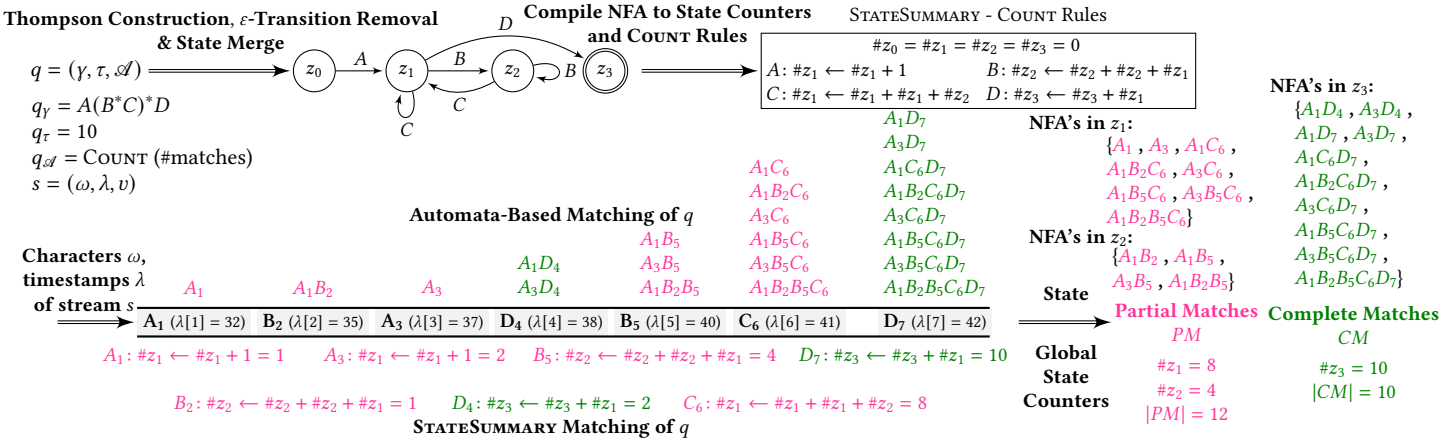


Figure 3: Characters ω and timestamps λ of a stream s and the NFA for the regular expression $q_\gamma = A(B^*C)^*D$. The counters of the StateSummary are contrasted with the (partial) matches constructed by traditional automata-based matching.

Table 2: Overview of different counter concepts.

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

With each character c, any (partial) match in a state z_i satisfying $\delta(z_i, c) \neq \emptyset$ can transition using c. This creates (partial) matches in the resulting state(s), as each (partial) match in z_i transitions with c, causing the current 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 $z_k \neq z_0$ 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:
if $\delta(z_i, c) = \{z_i\}$ then $\#z_i \leftarrow \#z_i + \#z_i$.
(2) If transitions from states $\{z_{i_1}, z_{i_2}, \ldots, z_{i_l}\}$ lead to z_k upon processing character c (excluding the self-loop case):
if $\delta(z_{i_j}, c) = \{z_k\}$ for $j \in \{1, ..., l\}$ then $\#z_k \leftarrow \#z_k + \sum_{j=1}^{l} \#z_{i_j}$.
(3) If the transition goes from the initial state z_0 to $z_k \neq z_0$:
if $\delta(z_0, c) = \{z_k\}$ then $\#z_k \leftarrow \#z_k + 1$.

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 ensures that no (partial) match undergoes two transitions.

Since δ yields a subset of 2^Z, a character c may update multiple state counters, with $O(|Z|)$ as an upper bound.

Example 4. *Consider the stream $s = (\omega, \lambda, \upsilon)$ with $\omega = \langle A_1, B_2, A_3, D_4, B_5, C_6, D_7 \rangle$ in Fig. 3. 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 zero. The Count rules are depicted in the upper box in Fig. 3. Processing s, 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. After processing s, it holds that $|PM| = \#z_1 + \#z_2 = 8 + 4 = 12$ and $|CM| = \#z_3 = 10$.*

Local State Counter. For each element e in $\mathcal{S}$, *local state counters*, denoted as $e_{\#Z} = (e_{\#z_0}, \ldots, e_{\#z_{|Z|}})$ and initialized to zero, quantify the number of partial and complete matches involving element e across each state. They are updated using the Count rules introduced above for global state counters. When an element e arrives, the local state counters $e_{\#Z}$ reflect the (partial) matches that e contributes to the global state counters; thus, $e_{\#Z}$ is updated first based on the values of the global state counters in $\mathcal{S}_{\#Z}$. Subsequent updates of $e_{\#Z}$, however, rely on the values in their respective local state counters, to incorporate new (partial) matches involving e.

Example 5. *Consider Fig. 3 and element B_5. Upon its arrival, B_5 is incorporated by all partial matches in states z_1 and z_2. 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: $B_{5\#z_1} \leftarrow B_{5\#z_1} + B_{5_{\#z_1}} + B_{5\#z_2} = 3$, resulting in $B_{5\#Z} = (0, 3, 3, 0)$.*

To discuss complexity, we overload notation and write $|\mathcal{S}|$ for the number of elements in $\mathcal{S}$. If each state is reachable from any other state, for each element e, 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 elements. If a stream element e is added to the summary, it is possible that not all summary elements fall within the same time window as e. Hence, the global state counters $\mathcal{S}_{\#Z}$ may not represent the correct values for initializing the local state counters $e_{\#Z}$. 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 elements in $\mathcal{S}$ that, upon adding an element e, satisfy the time window q_τ with e. Based thereon, we maintain separate *active* global and local state counters, denoted by $\mathcal{A}_{\#Z}$ and $\mathcal{A}_{e_{\#Z}}$, respectively. They reflect the match counts per element in $\mathcal{A}$ and are updated 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

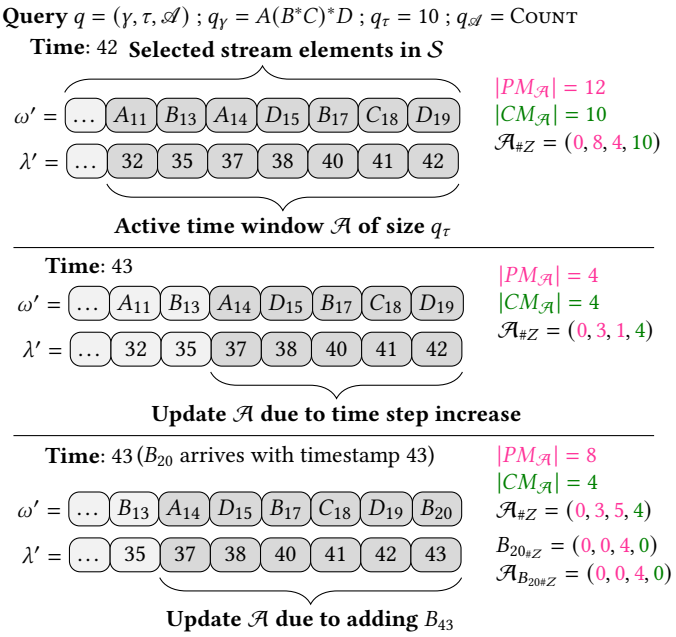


Figure 4: Evolution of a summary for a given regular expression query; shaded elements are in the active time window $\mathcal{A}$.

selected element, for which the character aligns with a transition in the automaton starting in the initial state z_0. There is no need to keep elements before this element $\mathcal{A}_{init}$ not linked to transitions from the initial state, as they cannot create *new* (partial) matches.

Example 6. *Consider the selected characters ω' of the elements at time 42 in Fig. 4 and the query q from Fig. 3. Here, D_{19} is the most recently appended element. Due to q_τ, $A_{11} = \mathcal{A}_{init}$ remains the earliest initiator matching D_{19} and stays in $\mathcal{A}$ as it satisfies the time window. Yet, at time 43, A_{11} is no longer valid and, hence, is removed from $\mathcal{A}$, but it remains in $\mathcal{S}$. Although B_{13} fits within the time window, it is discarded from $\mathcal{A}$ since it is not an initiator; we set $\mathcal{A}_{init} = A_{14}$. The elements in $\mathcal{A}$ correspond to ω in Fig. 2; thus, the matches corresponding to the counters are also depicted there.*

As we add B_{20}, the local state counters $B_{20\#Z}$ are initialized from the active time window state counters $\mathcal{A}_{\#Z} = (0, 3, 1, 4)$, so that we arrive at $B_{20\#Z} = (0, 0, \mathcal{A}_{\#z_1} + \mathcal{A}_{\#z_2}, 0) = (0, 0, 4, 0)$. These counts represent the current number of (partial) matches. Then, we update the global state counter values by adding the contributions from B_{20}: $\mathcal{S}_{\#Z} \leftarrow \mathcal{S}_{\#Z} + B_{20\#Z}$, meaning $\mathcal{S}_{\#z_i} \leftarrow \mathcal{S}_{\#z_i} + B_{20\#z_i}$ for $0 \le i < |Z|$.

In addition, we maintain *active local state counters* $\mathcal{A}_{e_{\#Z}}$ for elements e in $\mathcal{A}$ to track the number of matches each element participates in within $\mathcal{A}$ (whereas the local state counters $e_{\#Z}$ also cover elements no longer in $\mathcal{A}$). This nuanced tracking is crucial for estimating the benefit of an element in $\mathcal{A}$ to yield future matches.

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

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 $O(min\{q_\tau, |\mathcal{S}|\} \cdot |Z|)$.

5.2 Operations

In this section, we detail the Initialize, Insert, and Remove operations that manipulate the summary.

Initialize. The StateSummary is initialized for a regular expression query $q = (\gamma, \tau, \mathcal{A})$, the NFA $\mathcal{N} = (Z, \Sigma, \delta, z_0, E)$ derived for it, and a user-defined *summary size* n (from Problem 1). We create tuples $\mathcal{S}_{\#Z}, \mathcal{A}_{\#Z}$ 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.

We allocate $O(n \cdot |Z|)$ memory for all future elements and their local state counters. Consequently, the runtime of the Initialize operation is dominated by initializing the local state counter tuples for these elements, which requires $O(|\mathcal{S}| \cdot |Z|)$ time and space.

We use an array to hold the elements 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(e) operation appends an element e from the stream to the stream summary $\mathcal{S}$ and is detailed in Alg. 1. To incorporate e, we need to adjust the (active) global state counters $\mathcal{S}_{\#Z}$ and $\mathcal{A}_{\#Z}$, and for each element $\hat{e}$ in $\mathcal{A}$ its (active) local state counters $\hat{e}_{\#Z}$ and $\mathcal{A}_{\hat{e}_{\#Z}}$. Finally, we have to initialize the active local state counter $\mathcal{A}_{e_{\#Z}}$ of e and its local state counters $e_{\#Z}$.

To comprehend the influence of e's character c on the NFA, 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 tuple *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 tuple 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* (i.e., by adding tuple components, line 17). From line 4 to line 7, for each element $\hat{e}$ in $\mathcal{A}$, we determine the impact of c on the active local state counters of $\hat{e}$, denoted by *localChange* (line 5), to update its active local state counters $\mathcal{A}_{\hat{e}_{\#Z}}$ and local state counters $e_{\#Z}$ by *localChange*. Finally, after appending the new element e to $\mathcal{S}$ (line 8), denoted by $\mathcal{S} \oplus e$, we initialize its active local state counters $\mathcal{A}_{e_{\#Z}}$ (line 9) and local state counters $e_{\#Z}$ (line 10).

The time complexity of Insert is dominated by the updates of the (active) local state counters of elements 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* elements in $\mathcal{S}$. Since computeStateCounterChange requires $O(|Z|^2)$ time, Insert takes $O(min\{q_\tau, |\mathcal{S}|\} \cdot |Z|^2)$ time using $O(|Z|)$ space.

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

Algorithm 1: INSERT.

```
input : Summary S; element e to insert and its character c; NFA
        N = (Z, Σ, δ, z_0, E); active time window A
1  globCounterChange ← computeStateCounterChange(A_#Z, N, c);
2  S_#Z ← addStateCounters(S_#Z, globCounterChange, N);
3  A_#Z ← addStateCounters(A_#Z, globCounterChange, N);
4  for ê in A do
5      localChange ← computeStateCounterChange(A_ê#Z, N, c);
6      A_ê#Z ← addStateCounters(A_ê#Z, localChange, N);
7      ê_#Z ← addStateCounters(ê_#Z, localChange, N);
8  S ← S ⊕ e;
9  A_e#Z ← globCounterChange;
10 e_#Z ← globCounterChange;
11 function computeStateCounterChange(counters, N, c)
12     newCounters ← 0^|Z|;
13     for from_z ∈ Z do
14         for to_z ∈ δ(from_z, c) do
15             newCounters_#to_z ← newCounters_#to_z + counters_#from_z;
16     return newCounters;
17 function addStateCounters(stateCounters1, stateCounters2, N)
18     addCounters ← 0^|Z|;
19     for z ∈ Z do addCounters_#z ← stateCounters1_#z + stateCounters2_#z;
20     return addCounters;
```

Algorithm 2: REMOVE.

```
input : Summary S; element e_i to remove; query q = (γ, τ, A) NFA
        N = (Z, Σ, δ, z_0, E); active time window A
1  S_#Z ← S_#Z − e_i#Z;
2  A_#Z ← A_#Z − A_e_i#Z;
3  S_included.INITIALIZE(q, 2 · q_τ);
4  S_excluded.INITIALIZE(q, 2 · q_τ − 1);
5  affectedElements ← {e_j in S | |λ'[j] − λ'[i]| ≤ q_τ};
6  for e_j in affectedElements do
7      S_included.INSERT(e_j);
8      if e_j ≠ e_i then
9          S_excluded.INSERT(e_j);
10 for e in affectedElements do
11     e_#Z ← e_#Z − (S_included e_#Z − S_excluded e_#Z);
12 if |λ'[i] − currentTimestamp| ≤ q_τ then
13     replayActiveTimeWindow();
14 function replayActiveTimeWindow()
15     S_temp.INITIALIZE(q, q_τ);
16     for e in A do
17         S_temp.INSERT(e);
18     for e in A do
19         A_e#Z = S_temp e_#Z;
```

The STATESUMMARY provides the count of (partial) matches for each element and state but lacks information regarding the joint matches between two elements e_1, e_2 in $\mathcal{S}$. The removal of e_1 may necessitate adjustments to the (active) local state counters of other elements in $\mathcal{S}$ if they had (partial) matches with e_1. Consequently, determining the proportion of joint (partial) matches in the respective counters $e_{1\#Z}$ and $e_{2\#Z}$ is required for updating $e_{2\#Z}$.

The immediate question that follows is if we can discern the mutual (partial) matches based on the values in $e_{1\#Z}$ and $e_{2\#Z}$. Unfortunately, this is impossible. The reason being that e_1 and e_2 can match many elements that are exclusively matched with either e_1 or e_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

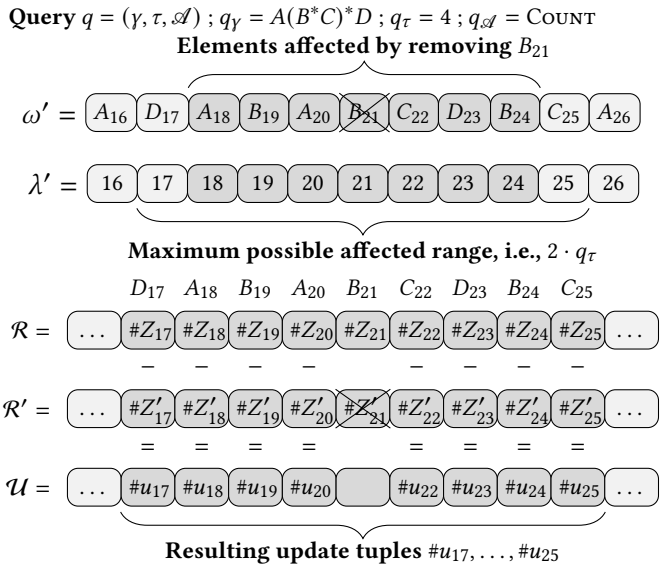


Figure 5: Evaluation of a summary when removing element B_{21}; for shaded elements, the local state counters are updated.

(partial) matches. Thus, we will now present a procedure that determines the joint counts.

Let e_i represent the element we intend to remove from $\mathcal{S}$. Due to the time window q_τ, element e_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 e_i, while the latter omits e_i. Through this approach, we obtain two sets: $\mathcal{R}$, representing the local state counter tuples within the specified range post the first replay (e_i included) (line 7), and $\mathcal{R}'$, which stands for the local state counter tuples in the same range following post the second replay (e_i excluded) (line 9).

Next, we aim to determine the set of update tuples, $\mathcal{U}$, for the affected elements. For each update tuple $\#u \in \mathcal{U}$, the values within $\#u$ indicate the number of (partial) matches the corresponding element has in common with e_i for a given state z in Z. For an element e_j ($j \neq i$) in the affected range, we determine the corresponding replay tuples $\#r_j \in \mathcal{R}$ and $\#r_{j'} \in \mathcal{R}'$ such that the difference between $\#r_j - \#r_{j'}$ represents the shared match counts e_j and e_i have in common for each state. For instance, in Fig. 5 for the element D_{17} the respective replay tuples are $\#Z_{17} \in \mathcal{R}$ and $\#Z_{17'} \in \mathcal{R}'$.

The update tuple is formulated as $\#u_j = \#r_j - \#r_{j'}$ and $\#u_j$ is then added to $\mathcal{U}$. Once $\mathcal{U}$ is computed, the local state counters, $e_{j\#Z}$, are adjusted for each affected e_j using the respective update tuple $\#u_j \in \mathcal{U}$ with the update operation $e_{j\#Z} \leftarrow e_{j\#Z} - \#u_j$ (line 11).

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).

Table 3: Time and space complexity of summary operations.

Operation	Time complexity	Space complexity
Initialize	$O(\|\mathcal{S}\| \cdot \|Z\|)$	$O(\|\mathcal{S}\| \cdot \|Z\|)$
Insert	$O(min\{q_\tau, \|\mathcal{S}\|\} \cdot \|Z\|^2)$	$O(\|Z\|)$
Remove	$O(min\{q_\tau, \|\mathcal{S}\|\}^2 \cdot \|Z\|^2)$	$O(min\{q_\tau, \|\mathcal{S}\|\} \cdot \|Z\|)$

Example 7. *Consider Fig. 5, where $q_\tau = 4$ and we remove the element corresponding to B_{21}. Due to q_τ, the possible affected range $2 \cdot q_\tau$ needs revision, i.e., $[21 - q_\tau, 21 + q_\tau]$. We replay the affected range twice, first including B_{21}, obtaining the replay tuples $\mathcal{R}$, the second excluding B_{21}, obtaining $\mathcal{R}'$. Let $j \in \mathbb{N}$. When subtracting $\#R'_j$ in $\mathcal{R}'$ from $\#R_j$ in $\mathcal{R}$, we obtain the update tuples $\mathcal{U}$, with $\#u_j$ in $\mathcal{U}$ being the update tuple for e_j in $\mathcal{S}$, which contains the joint counts of e_j and the removed character for each state counter. Based on these tuples, we update the local state counters of affected elements. If the removed element was present in $\mathcal{A}$, an additional replay of $\mathcal{A}$ yields updates to the active local state counters.*

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 tuples from the (active) local state counters. (2) The replay produces a zero tuple for elements in the area without matches to the removed element, as both replays yield identical tuples. For instance, this occurs for D_{17} in Fig. 5 where $\#Z_{17} = \#Z'_{17}$.

As for the insertion, removing an element e_i could affect the local state counters of *all* elements in $\mathcal{S}$, if $q_\tau > |\mathcal{S}|$.

Complexity. The Remove operation implies a series of Insert operations. For every new element inserted, all prior elements are updated. This results in a quadratic time complexity of $O(min\{q_\tau, |\mathcal{S}|\}^2 \cdot |Z|^2)$ and a space complexity of $O(min\{q_\tau, |\mathcal{S}|\} \cdot |Z|)$. We list an overview of our operations' time and space complexities in Table 3.

5.3 Model Extensions

Aggregate Functions. Having described the StateSummary for queries involving Count aggregations, we turn to support Sum, which then also facilitates Avg.

As before, when an element e is added to $\mathcal{S}$, transitions of the automaton are triggered, creating new matches in subsequent states. For all affected (partial) matches, the character c of element e and its payload value v must 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 tuple of *global sum counters* $\mathcal{S}_{\#Z} = (\mathcal{S}_{\#z_0}, \ldots, \mathcal{S}_{\#z_{|Z|}})$. 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: For each $z_i \in Z$, each $\mathcal{S}_{\#z_i}$ in $\mathcal{S}_{\#Z}$, and character c with value v:

(1) If z_i has a self-loop with c:
if $\delta(z_i, c) = \{z_i\}$ then $\mathcal{S}_{\#z_i} \leftarrow \mathcal{S}_{\#z_i} + \#z_i \cdot v$.
(2) If transitions from states $\{z_{i_1}, z_{i_2}, \ldots, z_{i_l}\}$ lead to z_k upon processing character c (excluding the self-loop case):
if $\delta(z_{i_j}, c) = \{z_k\}$ for $j \in \{1, 2, ..., l\}$ then
$\mathcal{S}_{\#z_k} \leftarrow \mathcal{S}_{\#z_k} + \sum_{j=1}^{l} \mathcal{S}_{\#z_{i_j}} + \#z_{i_j} \cdot v$.
(3) If the transition goes from the initial state z_0 to $z_k \neq z_0$:
if $\delta(z_0, c) = \{z_k\}$ then $\mathcal{S}_{\#z_k} \leftarrow \mathcal{S}_{\#z_k} + 1 \cdot v$.

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 8 (StateSummary Matching for Sum). *Consider Fig. 3, where we will now set $q_{\mathcal{A}}$ = Sum. Due to the underlying NFA, we have the Sum counters $\mathcal{S}_{\#Z} = \{\mathcal{S}_{\#z_0}, \mathcal{S}_{\#z_1}, \mathcal{S}_{\#z_2}, \mathcal{S}_{\#z_3}\}$. Let $v \in \mathbb{N}$ be a value of an element, the resulting Sum rules are:*
$A : \mathcal{S}_{\#z_1} \leftarrow \mathcal{S}_{\#z_1} + 1 \cdot v$,
$B : \mathcal{S}_{\#z_2} \leftarrow \mathcal{S}_{\#z_2} + \mathcal{S}_{\#z_2} + \#z_2 \cdot v + \mathcal{S}_{\#z_1} + \#z_1 \cdot v$,
$C : \mathcal{S}_{\#z_1} \leftarrow \mathcal{S}_{\#z_1} + \mathcal{S}_{\#z_1} + \#z_1 \cdot v + \mathcal{S}_{\#z_2} + \#z_2 \cdot v$, *and*
$D : \mathcal{S}_{\#z_3} \leftarrow \mathcal{S}_{\#z_3} + \mathcal{S}_{\#z_1} + \#z_1 \cdot v$.
We add to the given character stream $\omega = \langle A_1, B_2, A_3, D_4, B_5, C_6, D_7 \rangle$ a stream of values $v = \langle 5, 3, 2, 1, 7, 3, 8 \rangle$. After inserting A_1, we update $\mathcal{S}_{\#z_1} \leftarrow 0 + 1 \cdot v[1] = 0 + 1 \cdot 5 = 5$. When B_2 occurs, $\mathcal{S}_{\#z_2} \leftarrow 0 + 0 + 0 \cdot v[2] + 5 + 1 \cdot v[2] = 0 + 0 + 0 \cdot 3 + 5 + 1 \cdot 3 = 8$ and A_3 causes $\mathcal{S}_{\#z_1} \leftarrow 5 + 1 \cdot v[3] = 5 + 1 \cdot 2 = 7$. Due to D_4, we update $\mathcal{S}_{\#z_3} \leftarrow 0 + 7 + 2 \cdot v[4] = 0 + 7 + 2 \cdot 1 = 9$. B_5 induces $\mathcal{S}_{\#z_2} \leftarrow 8+8+1 \cdot v[5]+7+2 \cdot v[5] = 8+8+4 \cdot 7+2 \cdot 7 = 44$. When C_6 arrives, we obtain $\mathcal{S}_{\#z_1} \leftarrow 7+7+2 \cdot v[6]+44+4 \cdot v[6] = 7+7+2 \cdot 3+44+4 \cdot 3 = 76$. Finally, D_7 arrives, causing $\mathcal{S}_{\#z_3} = 9+76+8 \cdot v[7] = 9+76+8 \cdot 8 = 149$, obtaining the Sum counters: $(0, 76, 44, 149)$.

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 $z_i \in Z$.

Predicates. Patterns often materialize in stream partitions (e.g., per sensor, per person, per vehicle), making our model directly applicable for equality predicates over corresponding attribute values per partition. When defining predicates over subsets of query characters, e.g., $q_\gamma = ABC$ where B's and C's (but not A's) shall have equal attribute values, characters for which the predicate does not apply (here, the A characters) are replicated in each partition. Without partitioning, equality predicates can still be processed by maintaining counters for each attribute value previously considered for partitioning, tracking matches specifically for that value.

Selection Policies. The semantics underlying our model is known as the STAM policy [2, 15, 45]. To integrate a stricter policy, such as Skip-Till-Next-Match (STNM) [15], we adapt the Count rules to reflect STNM's key distinction: it prevents skipping *relevant* elements, i.e., (partial) matches are not duplicated; thus, the increase in matches in subsequent states is balanced by an equivalent decrease in the originating states when an element arrives. Applying STNM to our running example in Fig. 3, when A_1 occurs, it increases $\#z_1$ by one. B_2 causes all partial matches in z_1 to transition to z_2, increasing $\#z_2$ by one. However, at the same time, $\#z_1$ is reduced by one (from one to zero) since an instance is not duplicated upon transition, which would reflect skipping the relevant character B_2.

6 SUMMARY SELECTION

To address Problem 1, we shall maintain a summary that aims at minimizing the loss in query evaluation. To this end, we first introduce a benefit function to quantify the potential of an element to contribute to a substantial amount of complete matches (§6.1),

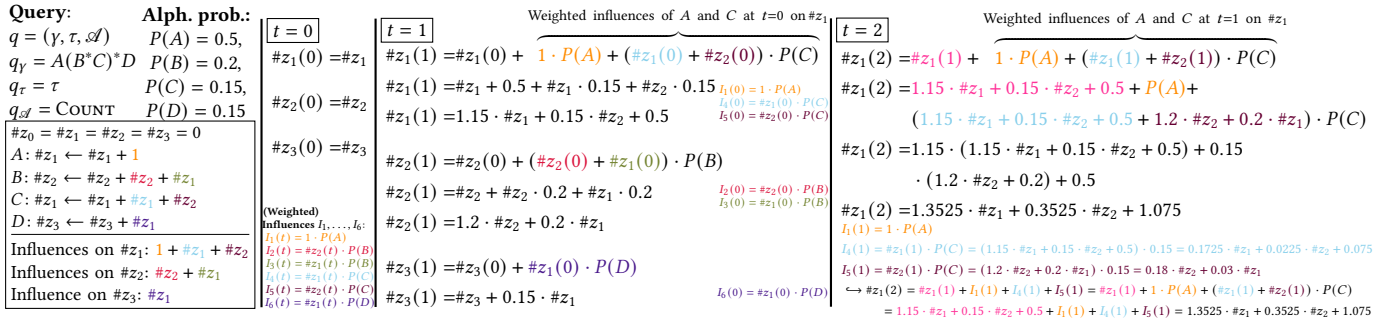


Figure 6: Extending the example from Fig. 3, we compute the expected benefit for $t \in [0, 1, 2]$. Resulting coefficients are cached.

which then guides our selection strategy (§6.2). Again, we introduce the concepts for Count, and later discuss Sum and Avg (§6.3).

6.1 Benefit Function

To build a stream summary, we define a benefit function $\mathcal{B}$ that scores an element based on the *present benefit* for current aggregation results, and the *expected benefit* for future results.

Present Benefit. The *present benefit* $\mathcal{B}_{\text{pres}}: \mathcal{S} \to \mathbb{N}$ (for Count) of an element e in $\mathcal{S}$ is defined by the number of *complete matches* involving e, i.e., sum of local state counters for final states F in $\mathcal{N}$:

$$\mathcal{B}_{\text{pres}}(e) = \sum_{z_i \in F} e_{\#z_i} \tag{1}$$

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

Expected Benefit. The *expected benefit* $\mathcal{B}_{\text{exp}}: \mathcal{S} \times \mathbb{T} \to \mathbb{R}_{\geq 0}$ quantifies the expected count of complete matches involving e using its active local state counter $\mathcal{A}_{e_{\#Z}}$ (i.e., (partial) matches with e that may lead to further matches) and the remaining time span Δ_τ, in which e can participate in matches.

We assume that the probability distribution over Σ, coined *alphabet probability* and denoted as $P_\Sigma: \Sigma \to [0, 1]$, is known. That is, $P_\Sigma(c)$ represents the probability that character $c \in \Sigma$ is the character of the subsequent stream element e. 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 time step $t \in [0, q_\tau]$, 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 $\#z_i$ in $\mathcal{A}_{e_{\#Z}}$ (denoted here as $\#z_i$ instead of $\mathcal{A}_{\#z_i}$ to ease the presentation), we compute how each $\#z_i$ is affected per time step. For $\#z_i$, the resulting count $\#z_i(t)$ represents the averaged count over all possible words Σ^t weighted by $P_\Sigma(c)$, i.e., the expected match count after t time steps.

Consider Fig. 6, with the regular expression $q_\gamma = A(B^*C)^*D$, the corresponding Count rules, and their influences on state counters. Each rule models how a state counter is *influenced*. Since we create a Count rule for each transition in $\mathcal{N}$, the number of influences equals the number of transitions $T = \{(z, x, z') \mid z \in Z, x \in \Sigma, z' \in \delta(z, x)\}$. For each active state counter $\#z_i$ in $\mathcal{A}_{e_{\#Z}}$, we determine the *total influence* on $\#z_i$ for a single time step. That is, we determine all influences on $\#z_i$ that correspond to all transitions leading to z_i, i.e., $T_{z_i} = \{(z, x) \mid z \in Z, x \in \Sigma, z_i \in \delta(z, x)\}$. Each tuple $(z, x) \in T_{z_i}$ represents an influence on z_i. For the automaton in Fig. 2, $T_{z_1} = \{(z_0, A), (z_1, C), (z_2, C)\}$, yielding the influences 1, $\#z_1$, and $\#z_2$, respectively, on $\#z_1$, which results in the total influence $1+\#z_1+\#z_2$. Also, each influence is weighted by probability $P_\Sigma(c)$ of character c occurring, e.g., for $\#z_1$, we get $1 \cdot P(A) + (\#z_1 + \#z_2) \cdot P(C)$.

For each transition in T, we define an *influence function*, $I_k: \mathbb{T} \to \mathbb{R}_{\geq 0}$, with $1 \leq k \leq |T|$, indicating the weighted influence of a tuple $(z, x, z') \in T$ at time t, yielding the set of all influences $\mathcal{I} = \{I_1, \ldots, I_{|T|}\}$. For every active state counter $\#z_i$ in $\mathcal{A}_{\#Z}$, we define an *influence set* $S_i \subseteq \{1, \ldots, |T|\}$, which contains the indices of influence functions that affect $\#z_i$. In Fig. 6, $\#z_1$'s influence set is determined by $S_1 = \{1, 4, 5\}$, indicating that I_1, I_4, and I_5 affect $\#z_1$.

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 $\#z_i$ in $\mathcal{A}_{e_{\#Z}}$ of an element e in $\mathcal{S}$ to compute e's expected benefit and, therefore, the overall *benefit*.

Specifically, at time $t = 0$, we adopt $\#z_i(0) = \#z_i$ as the initial relation, signifying $\#z_i$'s current value. Then, the general linear recurrence relation for each $t + 1$ and counter $\#z_i$ is defined as:

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

To update $\#z_i(t + 1)$, we add to its previous value, $\#z_i(t)$, the sum of weighted influences on $\#z_i$ at time t. In Fig. 6, at $t = 2$, $\#z_1(2)$ is updated to $1.3525 \cdot \#z_1 + 0.3525 \cdot \#z_2 + 1.075$, that is, $\#z_1(2) = \#z_1(1) + I_1(1) + I_4(1) + I_5(1) = \#z_1(1) + 1 \cdot P(A) + (\#z_1(1) + \#z_2(1)) \cdot P(C)$.

The output of Eq. 2 relies on the initial active local state counters values $\#z_i$ in $\mathcal{A}_{e_{\#Z}}$ of e, which only increase. Hence, the result also captures e's active present benefit. Thus, to define the *expected benefit function* $\mathcal{B}_{\text{exp}}: \mathcal{S} \times \mathbb{T} \to \mathbb{R}$, we subtract $\mathcal{B}_{\mathcal{A}_{\text{pres}}}(e)$, estimating the expected count of complete matches e leads to:

$$\mathcal{B}_{\text{exp}}(e, t) = \sum_{z_i \in F} \mathcal{A}_{e_{\#z_i}}(t) - \mathcal{B}_{\mathcal{A}_{\text{pres}}}(e) \tag{3}$$

For instance, let $\mathcal{A}_{e_{\#Z}} = (0, 5, 10, 15)$ in Fig. 6; at $t = 0$, $\mathcal{A}_{e_{\#Z}}$ remains unchanged, while at $t = 1$, $\mathcal{A}_{e_{\#Z}}$ updates to $(0, 7.75, 13, 15.75)$.

Caching Coefficients. The computational cost for $\mathcal{B}_{\text{exp}}(e, t)$ 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. 6 for illustration, we note that for each time step t and each active state counter $\#z_i$, a linear combination with constant coefficients is derived. A growing time window q_τ modifies only these coefficients, leaving the active 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. 6, for instance, for $\#z_1(2)$, the cached coefficients are 1.3525, 0.3525, and 1.075.

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 summary element e and time instance t, we can compute $\mathcal{B}_{\text{exp}}(e, t)$ in time $O(|Z|)$ since we have to factor in all cached coefficients for a given input tuple.

6.2 Selection Strategy

The *selection strategy*, denoted as $\psi(\mathcal{S}, e)$, derives a new summary from the current one $\mathcal{S}$ upon the arrival of a stream element e. It may insert e, replace e_{old} in $\mathcal{S}$ by e, or discard e (here, $\mathcal{S}|e_{\text{old}}$ denotes the removal of element e_{old} from $\mathcal{S}$)

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

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

$$\mathcal{B}(e, \Delta_{\tau_{\text{old}}}, \Delta_{\tau_{\text{new}}}) = \mathcal{B}_{\text{pres}}(e) + \frac{\mathcal{B}_{\text{exp}}(e, \Delta_{\tau_{\text{old}}}) + \mathcal{B}_{\text{exp}}(e, \Delta_{\tau_{\text{new}}})}{2} \tag{5}$$

The function offers a balanced measure of the benefit of e, 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 e might not contribute to matches for long; however, $\Delta_{\tau_{\text{new}}}$ dictates how long e can lead to matches. On the other hand, relying solely on $\Delta_{\tau_{\text{new}}}$ might not capture the expected benefit accurately, given that e aligns with i_{old} in many (partial) matches, but not for the span of $\Delta_{\tau_{\text{new}}}$.

Finally, we compute $e_{\#Z}$ and $\mathcal{A}_{e_{\#Z}}$ for e, to estimate its benefit $\mathcal{B}(e, \Delta_{\tau_{\text{old}}}, \Delta_{\tau_{\text{new}}})$. If the benefit of e' is smaller, we replace e' with e; otherwise, e is discarded. Since we compute $\mathcal{B}_{\text{exp}}$ 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 $O(q_\tau \cdot |Z|^2)$ space.

6.3 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 element e may generate. For Sum, each of these future matches carries a value v. Therefore, the expected benefit for e is augmented by its current present benefit (for Sum), combined with the expected match count of e weighted by v. While this function underestimates the expected Sum value for e, the underestimation is consistent across all summary elements.

7 EXPERIMENTAL EVALUATION

To evaluate SuSe, we conducted experiments using the setup described in §7.1. Specifically, we present results on the overall effectiveness, including an ablation and recall study (§7.2); on the sensitivity of the approach (§7.3); on a comparison against state-of-the-art engines for regular expressions and complex event processing (§7.4); and on two real-world datasets (§7.5 and §7.6).

7.1 Experimental Setup

Our implementation and experimental setup is publicly available [4].

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 [10] that captures information about bike rentals. It contains $\approx$ 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, 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, were 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 element in the summary once its capacity is met. The FIFO baseline consistently replaces the oldest element once its capacity is met.

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 StateSummary's efficiency by comparing it to the state-of-the-art RegEx engine REmatch [37] and CEP engines FlinkCEP [14] and CORE [7]. For REmatch and CORE, we conducted experiments without outputting matches to minimize overhead and ensure fairness. In our Flink setup, while match printing was required to measure processing time for implementation reasons, subsequent evaluations confirmed a negligible impact on the results.

Metrics. We assess the evaluation quality using the *relative recall improvement*. At each evaluation time point $t \in \mathcal{E}$, we divide the current number of complete matches from SuSe by the current number of complete matches from a corresponding baseline. Averaging the sum of these ratios denotes the relative recall improvement.

We also evaluate the *absolute 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 that were present in SuSe to *all* potential matches over time.

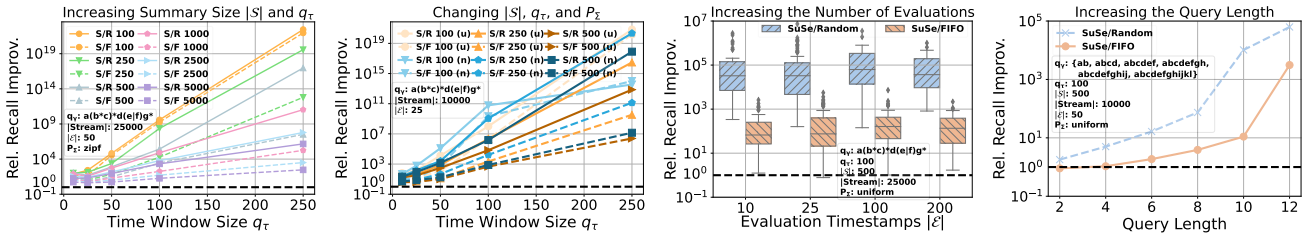


Figure 7: Examining the impact on relative recall improvement (higher is better) by varying (a) summary and time window sizes, (b) the alphabet probability distribution P_Σ, (c) the number of evaluation timestamps $|\mathcal{E}|$, and (d) the query length.

We further report performance metrics, such as *throughput* (avg number of elements processed per second), *latency* (avg processing duration per element), and *memory usage* (max 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).

7.2 Overall Effectiveness

We first compare against a random baseline (SuSe/Random - S/R) and a FIFO baseline (SuSe/FIFO - S/F). In Fig. 7a- 7b, 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 (u), normal (n)) affect the relative recall improvement, using a query with $q_\gamma = a(b^*c)^*d(e|f)g^*$.

SuSe outperforms its baselines by choosing summaries that generate up to 10^{20} more matches. There is no parameter combination, in which SuSe performs worse than a baseline (black dashed line). A clear trend emerges for both plots: the larger q_τ and the smaller $|\mathcal{S}|$, the more significant the advantage. As the summary size increases, the baselines find more matches by chance; nevertheless, for these parameters, SuSe chooses stream subsequences that generate at least three orders of magnitude more matches on average.

Ablation Study. We compared SuSe's benefit function $\mathcal{B}$ against a setup using solely the present benefit $\mathcal{B}_{\text{pres}}$, i.e., denoted by S/P, in terms of the relative recall improvement, the related absolute recall, and the detected matches recall. In Fig. 8a, a trend similar to that in Fig. 7a is observed: smaller $|\mathcal{S}|$ and larger q_τ lead to better subsequence selections. This is attributed to $\mathcal{B}$ providing a more accurate assessment of an element's future potential. For $q_\tau = 250$, SuSe with $\mathcal{B}$ obtains at least three orders of magnitude more matches on average over solely using $\mathcal{B}_{\text{pres}}$ and up to 10^{18} more for smaller values of $|\mathcal{S}|$, showing the effectiveness of $\mathcal{B}$.

Absolute Recall. Fig. 8b shows how closely SuSe approaches the optimal recall value. Increasing $|\mathcal{S}|$ positively affects recall. This is attributed to the reduced difference between the ground truth's summary size (equal to the stream size) and SuSe's summary size. SuSe clearly attains higher recall values when employing expected benefits $\mathcal{B}$, compared to using solely $\mathcal{B}_{\text{pres}}$. Also, when $|\mathcal{S}| \geq q_\tau$, with $\mathcal{B}$, SuSe selects substantially better subsequences, increasing recall by up to 80% – 95%. This trend can be attributed to our assumption in §6.2 regarding the expected benefit $\mathcal{B}$, where we implicitly assumed 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.

Detected Matches Recall. For Fig. 8c, 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 absolute recall. SuSe with $\mathcal{B}$ yields a recall of 90%–98% for $|\mathcal{S}| = 500$, indicating that SuSe chooses rich stream subsequences.

7.3 Sensitivity Analysis

Next, we assess the effect of various parameters on the relative recall improvement. We fix $q_\tau = 100$ and $|\mathcal{S}| = 500$, and vary stream sizes (10000 - 25000), the numbers of evaluations (25 - 50), and P_Σ (Zipfian or uniform). We report averages of over 50 runs.

Number of Evaluation Timestamps. Fig. 7c examines the influence of increasing the number of evaluation timestamps $|\mathcal{E}|$ on the relative recall improvement. As the boxplots indicate, increasing the number of evaluations does not impact the results.

Query Length. Fig. 7d shows that as the query length grows, selecting subsequences that yield matches becomes increasingly challenging. SuSe addresses this by quantifying the importance of individual elements, prioritizing, for instance, rare elements that are essential to keep to obtain matches. 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. In Fig. 9a, we employed a RegEx of length twelve and incrementally increased the number of Kleene operators, distributing them randomly. As SuSe is aware of characters with Kleene operators (by Count/Sum rules), it selects better summaries.

In Fig. 9b, we examined four regular expressions: one without a Kleene operator and three with the Kleene's position varied. Without Kleene, SuSe's advantages are minimal, while there is a large difference for the other cases, independent of the Kleene position.

Disjunction Operator. We also explored the effect of the disjunction operator in Fig. 9c. We varied the number of disjunctions from zero to seven and generated and combined random sequences for q_γ's with disjunction operators. Yet, the impact is negligible.

In Fig. 9d, 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 baseline show different trends, in all configurations, SuSe yields significant improvements.

7.4 Efficiency

StateSummary State-of-the-art Comparison. We compared StateSummary's efficiency against the state-of-the-art RegEx engine REmatch and two CEP engines, FlinkCEP and CORE, in Fig. 11.

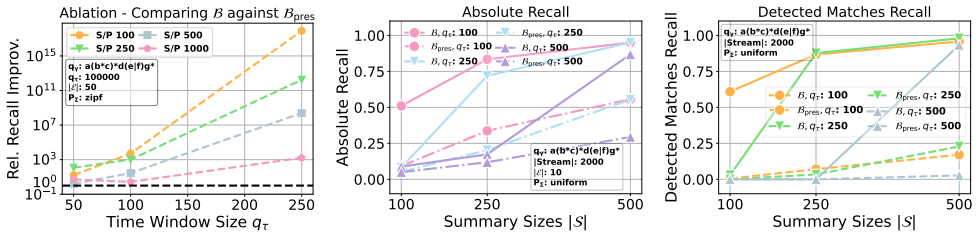


Figure 8: Examining the impact of the benefit function $\mathcal{B}$ compared to $\mathcal{B}_{\text{pres}}$ (a) on the relative recall improvement, (b) proximity to lower bound in absolute recall, and (c) proximity to lower bound in detected matches recall (higher is better in all plots).

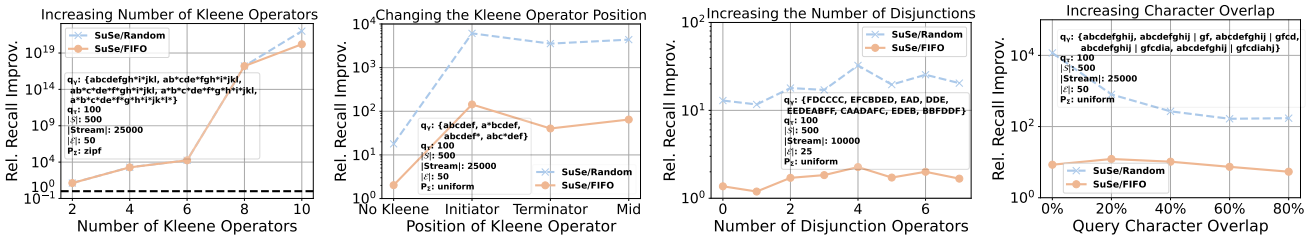


Figure 9: Examining the impact on relative recall improvement (higher is better) by varying (a) the count of Kleene stars, (b) Kleene star position, (c) disjunction operator count, and (d) query character overlap.

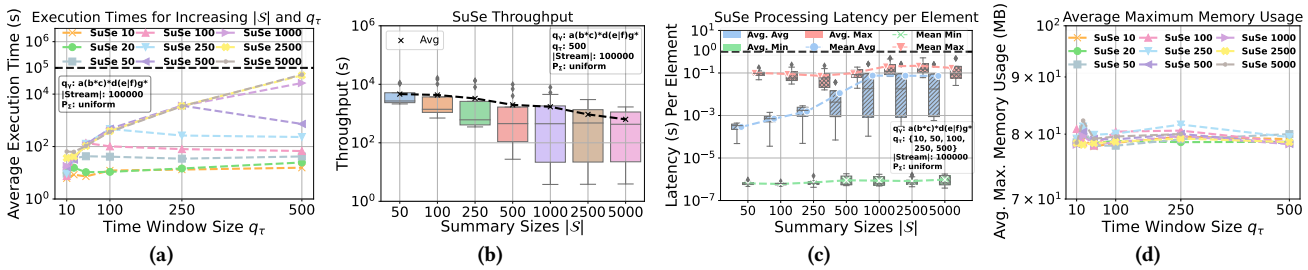


Figure 10: (a) Execution time (lower is better) for processing 10^5 elements across different summary and time window sizes, (b) average minimum, average maximum, and average processing latency per element (lower is better).

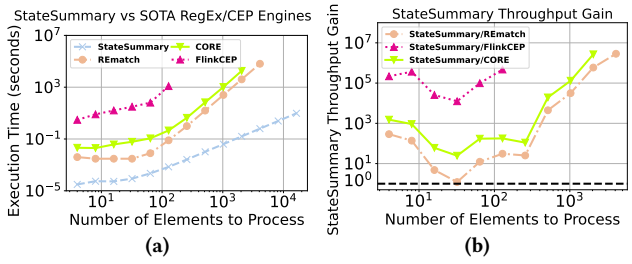
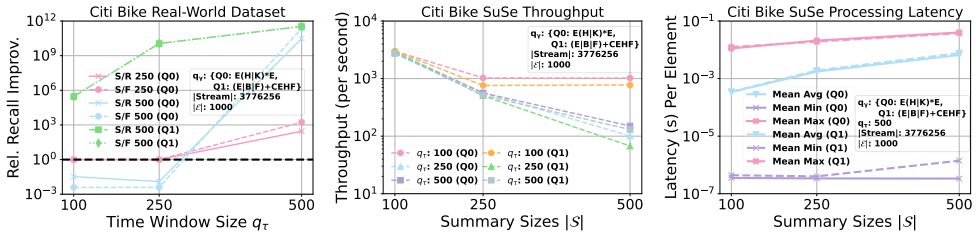


Figure 11: Comparison against the state of the art, focusing on (a) execution time (lower is better) and (b) throughput gain (higher is better), varying the number of processed elements.

We summarize the idea of the experiment as follows: To assess the current state, i.e., the current number of matches and involvements of individual events, we would need to use a RegEx/CEP engine. However, such engines implement a traditional two-step approach: first building the state, then evaluating aggregate functions over it. Given the inferior performance of other RegEx engines (see [37]) and their limitation in computing *all* subsequence matches, we focus on REmatch as a representative RegEx engine. While FlinkCEP employs a traditional automata-based evaluation, CORE employs compact match representations, avoiding exponential state growth, and provides match enumeration with output-linear delay.

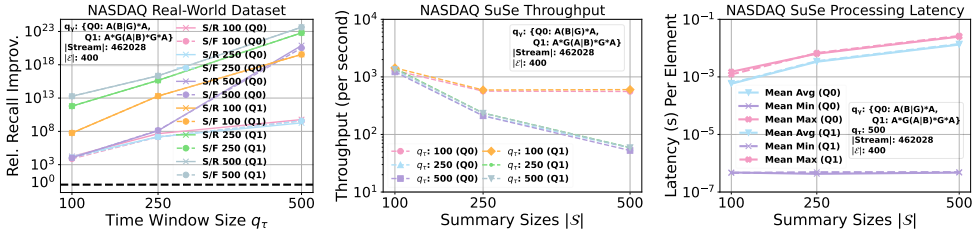
We set the summary size to the corresponding x-value, ensuring that the processed word fits entirely within the summary. Thus, we focused exclusively on StateSummary operations without employing summary selection, computing exact values instead of approximations. We set the time window size such that all elements fit into a single window, maximizing the number of matches to be detected. Also, we ensured that all methods produced the same result when evaluating the aggregate function. We tested with $q_Y = ABCD$ (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 $i \in \mathbb{N}^+$, e.g., $A^8B^8C^8D^8$, ensuring an increase in matches for rising x-values.

StateSummary's processing time is consistently at least an order of magnitude less than REmatch and CORE, and at least four orders of magnitude less than Flink. Particularly for words longer than 128 characters, the performance gap between StateSummary and both REmatch and CORE significantly widens, while Flink did not terminate for these values due to overload.



(a) (b) (c)

Figure 12: Citi Bike: (a) relative recall improvement (higher is better), (b) throughput (higher is better), and (c) latency (lower is better).



(a) (b) (c)

Figure 13: NASDAQ: (a) relative recall improvement (higher is better), (b) throughput (higher is better), and (c) latency (lower is better).

The performance differences induce throughput gains, as shown in Fig. 11b. The ratio indicates how much more quickly StateSummary processed a word than REmatch, CORE, or Flink. The throughput gain always exceeds one, indicating StateSummary's superiority; for longer words, this gain increases significantly.

SuSe's Overall Efficiency. In Fig. 10, we examined how varying $|\mathcal{S}|$ (see legend) and q_τ affects the execution time, throughput, processing latency per element, and memory usage of SuSe (employing summary selection). Fig. 10a shows 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, where execution time stabilizes or slightly decreases. The reasoning being that the Remove operation is the most expensive one (see §5.2), with cost $min(q_\tau, |\mathcal{S}|)$. 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 parameters marginal. With larger q_τ, SuSe chooses better subsequences, which reduces the number of Remove operations and explains the drop in runtime for $q_\tau \geq |\mathcal{S}|$.

In Fig. 10b, we examined the resulting throughput values for a fixed $q_\tau = 500$. For the same reasons as discussed above, there is a decline in throughput as the summary size expands.

In Fig. 10c, we investigated the resulting average min, max, and average processing latency per element per second. A boxplot shows the latencies for increasing q_τ, while the corresponding line plots denote the averages. A larger $|\mathcal{S}|$ leads to elevated processing latencies. However, the median of the average processing latencies consistently remains below 10^{-1}s, showing streaming feasibility.

Finally, the induced memory footprint turned out to be negligible and constant for various $|\mathcal{S}|$ and q_τ, see Fig. 10d.

7.5 Case Study: Citi Bike

Character Types and Queries. Fig. 12 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 a ride starts at a busy station, followed by a combination of rides at central and quieter stations, ending with a ride at a busy station, 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.

Effectiveness. To examine the relative recall improvement, we varied $|\mathcal{S}|$ and q_τ in Fig. 12a. 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 subsequences that are between three and twelve orders of magnitude superior for $q_\tau = 500$.

Throughput and Latency. Throughput and latency of SuSe are shown in Fig. 12b and Fig. 12c. For smaller values of $|\mathcal{S}|$ and q_τ (refer to legend), SuSe's throughput increases. However, with larger values for $|\mathcal{S}|$ and q_τ, the lower throughput is attributed to the Remove operation. Fig. 12c shows the avg min, avg max, and avg latency of SuSe, with $q_\tau = 500$. An increase in summary size leads to higher latencies. Yet, the average max latency hovers around 10^{-2} seconds, while the average processing latency remains under 10^{-2} seconds for $|\mathcal{S}| = 500$, which meets real-time demands.

7.6 Case Study: NASDAQ

Character Types and Queries. Fig. 13 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 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. 13a, 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 subsequences, resulting in up to 10^{23} additional matches. However, even at smaller parameter values, SuSe's subsequences surpassed both baselines by magnitudes ranging from 10^4 to 10^{13}. Notably, for $Q1$, SuSe identified better subsequences 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. 13b and Fig. 13c, 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. 13c, 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.

8 RELATED WORK

Subsequences. Subsequences are crucial tools in fields like biological sequence analysis [35] and event stream processing [5, 15, 45]. They are linked to longest common subsequences [6], shortest common supersequences [30], or subsequence counting [12], and are studied in contexts of gap-size [11] 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 single word y and a text w count the number of mappings $y \preceq_m w$. 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 streams, which require online computation.

CEP Optimizations. Queries in Complex Event Processing (CEP) use regular expressions for pattern recognition. CEP optimization include load shedding techniques [8, 39, 46] that discard events or partial matches unlikely to complete. 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 subsequence 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, 46], are not reliant on a learned model, enhancing streaming practicality.

9 CONCLUSIONS

We proposed SuSe, an architecture for regular expression subsequence summarization over data streams. It incorporates a StateSummary data structure that captures a query-specific stream summary through aggregated subsequence match information. We presented a summary selection algorithm, leveraging StateSummary for choosing stream subsequences, which aim to minimize the loss in the aggregated results over time. SuSe is multiple orders of magnitude faster than leading RegEx and CEP engines, while generating aggregates based on richer subsequences 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] Anonymized Author(s). 2024. SuSe: Summary Selection for Regular Expression Subsequence Aggregation over Streams. https://anonymous.4open.science/r/SuSe-8F12/.

[5] 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

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

[7] 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

[8] 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

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

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

[11] 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) (LIPIcs)*, Sang Won Bae and Heejin Park (Eds.), Vol. 248. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 64:1–64:18. https://doi.org/10.4230/LIPIcs.ISAAC.2022.64

[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 (LIPIcs)*, Dan Olteanu and Nils Vortmeier (Eds.), Vol. 220. Schloss Dagstuhl, 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), Proceedings (LNI)*, Birgitta König-Ries, Stefanie Scherzinger, Wolfgang Lehner, and Gottfried Vossen (Eds.), Vol. P-331. GI 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] Christopher Mutschler and Michael Philippsen. 2014. Adaptive Speculative Processing of Out-of-Order Event Streams. *ACM Trans. Internet Techn.* 14, 1 (2014), 4:1–4:24. https://doi.org/10.1145/2633686

[32] 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

[33] 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

[34] 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

[35] 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

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

[37] 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

[38] 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

[39] 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

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

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

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

[43] Wolfgang Weiss, Víctor Juan Expósito Jiménez, and Herwig Zeiner. 2020. Dynamic Buffer Sizing for Out-of-order Event Compensation for Time-sensitive Applications. *ACM Trans. Sens. Networks* 17, 1 (2020), 1:1–1:23. https://doi.org/10.1145/3410403

[44] 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

[45] 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

[46] 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