GRETA: Graph-based Real-time Event Trend Aggregation

Olga Poppe[1], Chuan Lei[2], Elke A. Rundensteiner[1], and David Maier[3]
[1]Worcester Polytechnic Institute, 100 Institute Road, Worcester, MA 01609
[2]IBM Research, Almaden, 650 Harry Rd, San Jose, CA 95120
[3]Portland State University, 1825 SW Broadway, Portland, OR 97201
[1]opoppe|rundenst@wpi.edu, [2]chuan.lei@ibm.com, [3]maier@cs.pdx.edu

ABSTRACT

Streaming applications from algorithmic trading to traffic management deploy Kleene patterns to detect and aggregate arbitrarily-long event sequences, called event trends. State-of-the-art systems process such queries in two steps. Namely, they first construct all trends and then aggregate them. Due to the exponential costs of trend construction, this two-step approach suffers from both a long delays and high memory costs. To overcome these limitations, we propose the Graph-based Real-time Event Trend Aggregation (GRETA) approach that dynamically computes event trend aggregation without first constructing these trends. We define the GRETA graph to compactly encode all trends. Our GRETA runtime incrementally maintains the graph, while dynamically propagating aggregates along its edges. Based on the graph, the final aggregate is incrementally updated and instantaneously returned at the end of each query window. Our GRETA runtime represents a win-win solution, reducing both the time complexity from exponential to quadratic and the space complexity from exponential to linear in the number of events. Our experiments demonstrate that GRETA achieves up to four orders of magnitude speed-up and up to 50–fold memory reduction compared to the state-of-the-art two-step approaches.

PVLDB Reference Format:
Olga Poppe, Chuan Lei, Elke A. Rundensteiner, and David Maier. GRETA: Graph-based Real-time Event Trend Aggregation. *PVLDB*, 11(1): 80-92, 2017.
DOI: 10.14778/3136610.3136617

1. INTRODUCTION

Complex Event Processing (CEP) is a technology for supporting streaming applications from algorithmic trading to traffic management. CEP systems continuously evaluate event queries against high-rate streams composed of primitive events to detect event trends such as stock market down-trends and aggressive driving. In contrast to traditional event sequences of *fixed* length [19], event trends have *arbitrary* length [24]. They are expressed by Kleene closure. Aggregation functions are applied to these trends to provide valuable summarized insights about the current situation. CEP applications typically must react to critical changes of these aggregates in real time [6, 31, 32].

Proceedings of the VLDB Endowment, Vol. 11, No. 1

DOI: 10.14778/3136610.3136617

Motivating Examples. We now describe three application scenarios of time-critical event trend aggregation.

• ***Algorithmic Trading***. Stock market analytics platforms evaluate expressive event queries against high-rate streams of financial transactions. They deploy event trend aggregation to identify and then exploit profit opportunities in real time. For example, query Q_1 computes the *count* of down-trends per industrial sector. Since stock trends of companies that belong to the same sector tend to move as a group [12], the number of down-trends across different companies in the same sector is a strong indicator of an upcoming down trend for the sector. When this indicator exceeds a certain threshold, a sell signal is triggered for the whole sector including companies without down-trends. These aggregation-based insights must be available to an algorithmic trading system in *near real time* to exploit short-term profit opportunities or avoid pitfalls.

Query Q_1 computes the number of down-trends per sector during a time window of 10 minutes that slides every 10 seconds. These stock trends are expressed by the Kleene plus operator $S+$. All events in a trend carry the same company and sector identifier as required by the predicate $[company, sector]$. The predicate $S.price > \mathsf{NEXT}(S).price$ expresses that the price continually decreases from one event to the next in a trend. The query ignores local price fluctuations by skipping over increasing price records.

Q_1 : RETURN *sector*, COUNT(∗) PATTERN *Stock* $S+$
WHERE $[company, sector]$ AND $S.price > \mathsf{NEXT}(S).price$
GROUP-BY *sector* WITHIN 10 *minutes* SLIDE 10 *seconds*

• ***Hadoop Cluster Monitoring***. Modern computer cluster monitoring tools gather system measurements regarding CPU and memory utilization at runtime. These measurements combined with workflow-specific logs (such as start, progress, and end of Hadoop jobs) form load distribution trends per job over time. These load trends are aggregated to dynamically detect and then tackle cluster bottlenecks, unbalanced load distributions, and data queuing issues [32]. For example, when a mapper experiences increasing load trends on a cluster, we might measure the *total CPU cycles* per job of such a mapper. These aggregated measurements over load distribution trends are leveraged in *near real time* to enable automatic tuning of cluster performance.

Query Q_2 computes the total CPU cycles per job of each mapper experiencing increasing load trends on a cluster during a time window of 1 minute that slides every 30 seconds. A trend matched by the pattern of Q_2 is a sequence of a job-start event S, any number of mapper performance measurements $M+$, and a job-end event E. All events in a trend must carry the same job and mapper identifiers expressed by the predicate $[job, mapper]$. The predicate M.load < NEXT(M).load requires the load measurements to increase from one event to the next in a load distribution trend. The query may ignore any event to detect all load trends of interest for accurate cluster monitoring.

Q_2 : RETURN *mapper*, SUM($M.cpu$)
PATTERN SEQ($Start\ S$, $Measurement\ M+$, $End\ E$)
WHERE $[job, mapper]$ AND $M.load <$ NEXT(M).*load*
GROUP-BY *mapper* WITHIN 1 *minute* SLIDE 30 *seconds*

• ***Traffic Management*** is based on the insights gained during continuous traffic monitoring. For example, leveraging the *maximal* speed per vehicle that follows certain trajectories on a road, a traffic control system recognizes congestion, speeding, and aggressive driving. Based on this knowledge, the system predicts the traffic flow and computes fast and safe routes in real time to reduce travel time, costs, noise, and environmental pollution.

Query Q_3 detects traffic jams which are not caused by accidents. To this end, the query computes the number and the average speed of cars continually slowing down in a road segment without accidents during 5 minutes time window that slides every minute. A trend matched by Q_3 is a sequence of any number of position reports $P+$ without an accident event A preceding them. All events in a trend must carry the same vehicle and road segment identifiers expressed by the predicate $[P.vehicle, segment]$. The speed of each car decreases from one position report to the next in a trend, expressed by the predicate P.speed > NEXT(P).speed. The query may skip any event to detect all relevant car trajectories for precise traffic statistics.

Q_3 : RETURN *segment*, COUNT($*$), AVG($P.speed$)
PATTERN SEQ(NOT *Accident* A, *Position* $P+$)
WHERE $[P.vehicle, segment]$ AND
$P.speed >$ NEXT(P).*speed*
GROUP-BY *segment* WITHIN 5 *minutes* SLIDE 1 *minute*

State-of-the-Art Systems do not support incremental aggregation of event trends. They can be divided into:

• ***CEP Approaches*** including SASE [6, 32], Cayuga [9], and ZStream [22] support Kleene closure to express event trends. While their query languages support aggregation, these approaches do not provide any details on how they handle aggregation on top of nested Kleene patterns. Given no special optimization techniques, these approaches construct all event trends prior to their aggregation (Figure 1). These two-step approaches suffer from high computation costs caused by the exponential number of arbitrarily-long trends. Our experiments in Section 10 confirm that such two-step approaches take over two hours to compute event trend aggregation even for moderate stream rates of 500k events per window. Thus, they fail to meet the low-latency requirement of time-critical applications. A-Seq [26] is the only system we are aware of that targets incremental aggregation of event sequences. However, it is restricted to the simple case of *fixed-length* sequences such as SEQ(A, B, C). It supports neither Kleene closure nor expressive predicates. Therefore, A-Seq does not tackle the exponential complexity of event trends – which now is the focus of our work.

• ***Streaming Systems*** support aggregation computation over streams [8, 10, 13, 15, 30]. However, these approaches evaluate simple Select-Project-Join queries with windows, i.e., their execution paradigm is set-based. They support neither event sequence nor Kleene closure as query operators. Typically, these approaches require the construction of join-results prior to their aggregation. Thus, they define incremental aggregation of *single raw events* but focus on multi-query optimization techniques [13] and sharing aggregation computation between sliding windows [15].

Challenges. We tackle the following open problems:

• ***Correct Online Event Trend Aggregation***. Kleene closure matches an exponential number of arbitrarily-long event trends in the number of events in the worst case [32]. Thus, any practical solution must aim to aggregate event trends without first constructing them to enable real-time in-memory execution. At the same time, correctness must be guaranteed. That is, the same aggregation results must be returned as by the two-step approach.

• ***Nested Kleene Patterns***. Kleene closure detects event trends of arbitrary, statically unknown length. Worse yet, Kleene closure, event sequence, and negation may be arbitrarily-nested in an event pattern, introducing complex interdependencies between events in an event trend. Incremental aggregation of such arbitrarily-long and complex event trends is an open problem.

• ***Expressive Event Trend Filtering***. Expressive predicates may determine event relevance depending on other events in a trend. Since a new event may have to be compared to *any* previously matched event, all events must be kept. The need to store all matched events conflicts with the instantaneous aggregation requirement. Furthermore, due to the continuous nature of streaming, events expire over time – triggering an update of all affected aggregates. However, recomputing aggregates for each expired event would put *real-time* system responsiveness at risk.

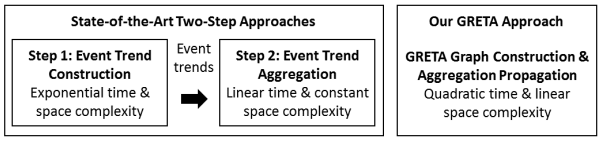


Figure 1: State-of-the-art versus our GRETA approach

Our Proposed GRETA Approach. We are the first to tackle these challenges in our Graph-based Real-time Event Trend Aggregation (GRETA) approach (Figure 1). Given an event trend aggregation query q and a stream I, the GRETA runtime compactly encodes all event trends matched by the query q in the stream I into a GRETA graph. During graph construction, aggregates are propagated from previous events to newly arrived events along the edges of the graph following the dynamic programming principle. This propagation is proven to assure incremental aggregation computation without first constructing the trends. The final aggregate is also computed incrementally such that it can be instantaneously returned at the end of each window of q.

Contributions. Our key innovations include:

1) We translate a nested Kleene pattern P into a GRETA template. Based on this template, we construct the *GRETA*

graph that compactly captures all trends matched by pattern P in the stream. During graph construction, the aggregates are dynamically propagated along the edges of the graph. We prove the correctness of the GRETA graph and the graph-based aggregation computation.

2) To handle nested patterns with negative sub-patterns, we split the pattern into positive and negative sub-patterns. We maintain a separate GRETA graph for each resulting sub-pattern and invalidate certain events if a match of a negative sub-pattern is found.

3) To avoid sub-graph replication between overlapping sliding windows, we share one GRETA graph between all windows. Each event that falls into k windows maintains k aggregates. Final aggregate is computed per window.

4) To ensure low-latency lightweight query processing, we design the *GRETA runtime data structure* to support dynamic insertion of newly arriving events, batch-deletion of expired events, incremental propagation of aggregates, and efficient evaluation of expressive predicates.

5) We prove that our GRETA approach reduces the time complexity from exponential to quadratic in the number of events compared to the two-step approach and in fact achieves *optimal time complexity*. We also prove that the space complexity is reduced from exponential to linear.

6) Our experiments using synthetic and real data sets demonstrates that GRETA achieves up to four orders of magnitude speed-up and consumes up to 50–fold less memory compared to the state-of-the-art strategies [2, 24, 32].

Outline. We start with preliminaries in Section 2. We overview our GRETA approach in Section 3. Section 4 covers positive patterns, while negation is tackled in Section 5. We consider other language clauses in Section 6. We describe our data structure in Section 7 and analyze complexity in Section 8. Section 9 discusses how our GRETA approach can support additional language features. Section 10 describes the experiments. Related work is discussed in Section 11. Section 12 concludes the paper.

2. GRETA DATA AND QUERY MODEL

Time. Time is represented by a linearly ordered *set of time points* $(\mathbb{T}, \leq)$, where $\mathbb{T} \subseteq \mathbb{Q}^+$ and $\mathbb{Q}^+$ denotes the set of non-negative rational numbers.

Event Stream. An *event* is a message indicating that something of interest happens in the real world. An event e has an *occurrence time* $e.time \in \mathbb{T}$ assigned by the event source. For simplicity, we assume that events arrive in-order by time stamps. Otherwise, an existing approach to handle out-of-order events can be employed [17, 18].

An event e belongs to a particular *event type* E, denoted $e.type = E$ and described by a *schema* which specifies the set of *event attributes* and the domains of their values.

Events are sent by event producers (e.g., brokers) on an *event stream* I. An event consumer (e.g., algorithmic trading system) monitors the stream with *event queries*. We borrow the query syntax and semantics from SASE [6, 32].

Definition 1. (**Kleene Pattern**.) Let I be an event stream. A ***pattern*** P is recursively defined as follows:

- An ***event type*** E matches an event $e \in I$, denoted $e \in matches(E)$, if $e.type = E$.
- An ***event sequence operator*** $\mathsf{SEQ}(P_i, P_j)$ matches an ***event sequence*** $s = (e_1, \ldots, e_k)$, denoted $s \in matches(\mathsf{SEQ}(P_i, P_j))$, if $\exists m \in \mathbb{N}$, $1 \leq m \leq k$, such that $(e_1, \ldots, e_m) \in matches(P_i)$, $(e_{m+1}, \ldots, e_k) \in matches(P_j)$, and $\forall l \in \mathbb{N}$, $1 \leq l < k$, $e_l.time < e_{l+1}.time$. Two events e_l and e_{l+1} are called ***adjacent*** in the sequence s. For an event sequence s, we define $s.start = e_1$ and $s.end = e_k$.
- A ***Kleene plus operator*** P_i+ matches an ***event trend*** $tr = (s_1, \ldots, s_k)$, denoted $tr \in matches(P_i+)$, if $\forall l \in \mathbb{N}$, $1 \leq l \leq k$, $s_l \in matches(P_i)$ and $s_l.end.time < s_{l+1}.start.time$. Two events $s_l.end$ and $s_{l+1}.start$ are called ***adjacent*** in the trend tr. For an event trend tr, we define $tr.start = s_1.start$ and $tr.end = s_k.end$.
- A ***negation operator*** $\mathsf{NOT}\ N$ appears within an event sequence operator $\mathsf{SEQ}(P_i, \mathsf{NOT}\ N, P_j)$ (see below). The pattern $\mathsf{SEQ}(P_i, \mathsf{NOT}\ N, P_j)$ matches an ***event sequence*** $s = (s_i, s_j)$, denoted $s \in matches(\mathsf{SEQ}(P_i, \mathsf{NOT}\ N, P_j))$, if $s_i \in matches(P_i)$, $s_j \in matches(P_j)$, and $\nexists s_n \in matches(N)$ such that $s_i.end.time < s_n.start.time$ and $s_n.end.time < s_j.start.time$. Two events $s_i.end$ and $s_j.start$ are called ***adjacent*** in the sequence s.

A ***Kleene pattern*** is a pattern with at least one Kleene plus operator. A pattern is ***positive*** if it contains no negation. If an operator in P is applied to the result of another operator, P is ***nested***. Otherwise, P is ***flat***. The ***size*** of P is the number of event types and operators in it.

All queries in Section 1 have Kleene patterns. The patterns of Q_1 and Q_2 are positive. The pattern of Q_3 contains a negative sub-pattern NOT *Accident A*. The pattern of Q_1 is flat, while the patterns of Q_2 and Q_3 are nested.

While Definition 1 enables construction of arbitrarily-nested patterns, nesting a Kleene plus into a negation and vice versa is not useful. Indeed, the patterns $\mathsf{NOT}\ (P+)$ and $(\mathsf{NOT}\ P)+$ are both equivalent to $\mathsf{NOT}\ P$. Thus, we assume that a negation operator appears within an event sequence operator and is applied to an event sequence operator or an event type. Furthermore, an event sequence operator applied to consecutive negative sub-patterns $\mathsf{SEQ}(\mathsf{NOT}\ P_i, \mathsf{NOT}\ P_j)$ is equivalent to the pattern $\mathsf{NOT}\ \mathsf{SEQ}(P_i, P_j)$. Thus, we assume that only a positive sub-pattern may precede and follow a negative sub-pattern. Lastly, negation may not be the outer most operator in a meaningful pattern. For simplicity, we assume that an event type appears at most once in a pattern. A straightforward extension of our GRETA approach allows to drop this assumption [25].

Definition 2. (**Event Trend Aggregation Query**.) An ***event trend aggregation query*** q consists of five clauses:

- Aggregation result specification (RETURN clause),
- Kleene pattern P ($\mathsf{PATTERN}$ clause),
- Predicates θ (optional WHERE clause),
- Grouping G (optional $\mathsf{GROUP\text{-}BY}$ clause), and
- Window w ($\mathsf{WITHIN/SLIDE}$ clause).

The query q requires each event in a trend matched by its pattern P (Definition 1) to be within the same window w, satisfy the predicates θ, and carry the same values of the grouping attributes G. These trends are grouped by the values of G. An aggregate is computed per group. We focus on distributive (such as COUNT, MIN, MAX, SUM) and algebraic aggregation functions (such as AVG) since they can be computed incrementally [11].

Let e be an event of type E and $attr$ be an attribute of e. $\mathsf{COUNT}(*)$ returns the number of all trends per group, while $\mathsf{COUNT}(E)$ computes the number of all events e in all trends per group. $\mathsf{MIN}(E.attr)$ ($\mathsf{MAX}(E.attr)$) computes the minimal (maximal) value of $attr$ for all events e in all

trends per group. SUM($E.attr$) calculates the summation of the value of $attr$ of all events e in all trends per group. Lastly, AVG($E.attr$) is computed as SUM($E.attr$) divided by COUNT(E) per group.

Skip-Till-Any-Match Semantics. We focus on Kleene patterns evaluated under the most flexible semantics, called *skip-till-any-match* in the literature [6, 31, 32]. Skip-till-any-match detects *all possible trends* by allowing to skip *any* event in the stream as follows. When an event e arrives, it extends each existing trend tr that can be extended by e. In addition, the unchanged trend tr is kept to preserve opportunities for alternative matches. Thus, an event doubles the number of trends in the worst case and the number of trends grows exponentially in the number of events [26, 32]. While the number of all trends is exponential, an application selects a subset of trends of interest using predicates, windows, grouping, and negation (Definition 2).

Detecting all trends is necessary in some applications such as algorithmic trading (Section 1). For example, given the stream of price records $I = \{10, 2, 9, 8, 7, 1, 6, 5, 4, 3\}$, skip-till-any-match is the only semantics that detects the down-trend (10,9,8,7,6,5,4,3) by ignoring local fluctuations 2 and 1. Since longer stock trends are considered to be more reliable [12], this long trend[1]. can be more valuable to the algorithmic trading system than three shorter trends (10,2), (9,8,7,1), and (6,5,4,3) detected under the skip-till-next-match semantics that does not skip events that can be matched (Section 9). Other use cases of skip-till-any-match include financial fraud, health care, logistics, network security, cluster monitoring, and e-commerce [6, 31, 32].

3. GRETA APPROACH IN A NUTSHELL

Our ***Event Trend Aggregation Problem*** to compute event trend aggregation results of a query q against an event stream I with *minimal latency*.

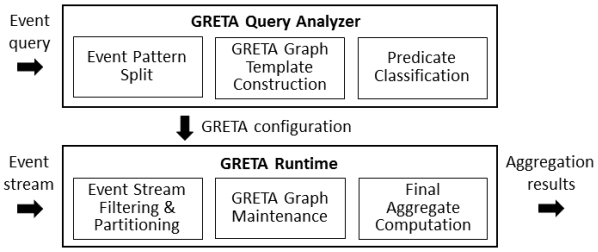


Figure 2: GRETA framework

Figure 2 provides an overview of our GRETA framework. The ***GRETA Query Analyzer*** statically encodes the query into a GRETA configuration. In particular, the pattern is split into positive and negative sub-patterns (Section 5.1). Each sub-pattern is translated into a GRETA template (Section 4.1). Predicates are classified into vertex and edge predicates (Section 6). Guided by the GRETA configuration, the ***GRETA Runtime*** first filters and partitions the stream based on the vertex predicates and grouping attributes of the query. Then, the GRETA runtime encodes matched event trends into a GRETA graph. During the graph construction, aggregates are propagated along the edges of the graph in a dynamic programming fashion. The final aggregate is updated incrementally, and thus is returned immediately at the end of each window (Sections 4.2, 5.2, 6).

[1]We sketch how constraints on minimal trend length can be supported by GRETA in [25]

Algorithm 1 GRETA template construction algorithm

Input: Positive pattern P
Output: GRETA template $\mathcal{T}$

```
generate(P) {
S ← event types in P, T ← ∅, 𝒯 = (S, T)
for each SEQ(P_i, P_j) in P do
    t ← (end(P_i), start(P_j)), t.label ← "SEQ"
    T ← T ∪ {t}
for each P_i+ in P do
    t ← (end(P_i), start(P_i)), t.label ← "+"
    T ← T ∪ {t}
return 𝒯 }
start(P) {
switch P do
    case E return E
    case P_i+ return start(P_i)
    case SEQ(P_i, P_j) return start(P_i) }
end(P) {
switch P do
    case E return E
    case P_i+ return end(P_i)
    case SEQ(P_i, P_j) return end(P_j) }
```

4. POSITIVE NESTED PATTERNS

We statically translate a positive pattern into a GRETA template (Section 4.1) At runtime, the GRETA graph is maintained according to this template (Section 4.2).

4.1 Static GRETA Template

The GRETA query analyzer translates a Kleene pattern P into a Finite State Automaton that is then used as a template during GRETA graph construction at runtime. For example, the pattern P=(SEQ(A+,B))+ is translated into the GRETA template in Figure 3.

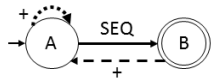
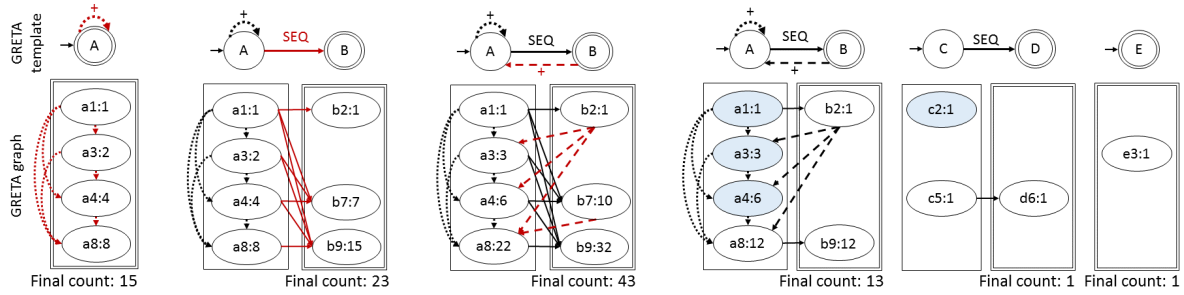


Figure 3: GRETA template for (SEQ(A+,B))+

States correspond to event types in P. The start state is labeled by the first type in P, denoted $start(P)$. Events of type $start(P)$ are called START events. The end state has label $end(P)$, i.e., the last type in P. Events of type $end(P)$ are called END events. All other states are labeled by types $mid(P)$. Events of type $E \in mid(P)$ are called MID events. In Figure 3, $start(P) = A$, $end(P) = B$, and $mid(P) = \emptyset$.

Since an event type may appear in a pattern at most once (Section 2), state labels are distinct. There is one $start(P)$ and one $end(P)$ event type per pattern P [25]. There can be any number of event types in the set $mid(P)$. $start(P) \notin mid(P)$ and $end(P) \notin mid(P)$. An event type may be both $start(P)$ and $end(P)$, for example, in the pattern A+.

Transitions correspond to operators in P. They connect types of events that may be adjacent in a trend matched by



(a) $A+$ (b) $\mathsf{SEQ}(A+, B)$ (c) $(\mathsf{SEQ}(A+, B))+$ (d) $(\mathsf{SEQ}(A+, \mathsf{NOT}\ \mathsf{SEQ}(C, \mathsf{NOT}E, D), B))+$

Figure 4: Count of trends matched by the pattern P in the stream $I = \{a1, b2, c2, a3, e3, a4, c5, d6, b7, a8, b9\}$

P. If a transition connects an event type E_i with an event type E_j, then E_i is a *predecessor event type* of E_j, denoted $E_i \in P.predTypes(E_j)$. In Figure 3, $P.predTypes(A) = \{A, B\}$ and $P.predTypes(B) = \{A\}$.

GRETA Template Construction Algorithm. Algorithm 1 consumes a positive pattern P and returns the automaton-based representation of P, called *GRETA template* $\mathcal{T} = (S, T)$. The states S correspond to the event types in P (line 2), while the transitions T correspond to the operators in P. Initially, the set T is empty (line 2). For each event sequence $\mathsf{SEQ}(P_i, P_j)$ in P, there is a transition from $end(P_i)$ to $start(P_j)$ with label "SEQ" (lines 3–5). Analogously, for each Kleene plus P_i+ in P, there is a transition from $end(P_i)$ to $start(P_i)$ with label "+" (lines 6–8). Start and end event types of a pattern are computed by the auxiliary methods in lines 10–19.

Complexity Analysis. Let P be a pattern of size s (Definition 1). To extract all event types and operators from P, P is parsed once in $\Theta(s)$ time. For each operator, we determine its start and event event types in $O(s)$ time. Thus, the time complexity is quadratic $O(s^2)$. The space complexity is linear in the size of the template $\Theta(|S| + |T|) = \Theta(s)$.

4.2 Runtime GRETA Graph

The GRETA graph is a runtime instantiation of the GRETA template. The graph is constructed on-the-fly as events arrive (Algorithm 2). The graph compactly captures all matched trends and enables their incremental aggregation.

Compact Event Trend Encoding. The graph encodes all trends and thus avoids their construction.

Vertices represent events in the stream I matched by the pattern P. Each state with label E in the template is associated with the sub-graph of events of type E in the graph. We highlight each sub-graph by a rectangle frame. If E is an end state, the frame is depicted as a double rectangle. Otherwise, the frame is a single rectangle. An event is labeled by its event type, time stamp, and intermediate aggregate (see below). Each event is stored once. Figure 4(c) illustrates the template and the graph for the stream I.

Edges connect adjacent events in a trend matched by the pattern P in a stream I (Definition 1). While transitions in the template express predecessor relationships between event types in the pattern, edges in the graph capture predecessor relationships between events in a trend. In Figure 4(c), we depict a transition in the template and its respective edges in the graph in the same way. A path from a START to an END event in the graph corresponds to a trend. The length of these trends ranges from the shortest $(a1, b2)$ to the longest $(a1, b2, a3, a4, b7, a8, b9)$.

In summary, the GRETA graph in Figure 4(c) compactly captures all 43 event trends matched by the pattern P in the stream I. In contrast to the two-step approach, the graph avoids repeated computations and replicated storage of common sub-trends such as $(a1, b2)$.

Dynamic Aggregation Propagation. Intermediate aggregates are propagated through the graph from previous events to new events in dynamic programming fashion. Final aggregate is incrementally computed based on intermediate aggregates. In the examples below, we compute event trend count COUNT(*) as defined in Section 2. Same principles apply to other aggregation functions [25].

Intermediate Count $e.count$ of an event e corresponds to the number of (sub) trends in G that begin with a START event in G and end at e. When e is inserted into the graph, all predecessor events of e connect to e. That is, e extends all trends that ended at a predecessor event of e. To accumulate the number of trends extended by e, $e.count$ is set to the sum of counts of the predecessor events of e. In addition, if e is a START event, it starts a new trend. Thus, $e.count$ is incremented by 1. In Figure 4(c), the count of the START event $a4$ is set to 1 plus the sum of the counts of its predecessor events $a1, b2$, and $a3$.

$$a4.count = 1 + (a1.count + b2.count + a3.count) = 6$$

$a4.count$ is computed once, stored, and reused to compute the counts of $b7$, $a8$, and $b9$ that $a4$ connects to. For example, the count of $b7$ is set to the sum of the counts of all predecessor events of $b7$.

$$b7.count = a1.count + a3.count + a4.count = 10$$

Final Count corresponds to the sum of the counts of all END events in the graph.

$$final\_count = b2.count + b7.count + b9.count = 43$$

In summary, the count of a new event is computed based on the counts of previous events in the graph following the dynamic programming principle. Each intermediate count is computed once. The final count is incrementally updated by each END event and instantaneously returned at the end of each window.

Definition 3. (**GRETA Graph.**) The *GRETA graph* $G = (V, E, final\_count)$ for a query q and a stream I is a directed

Algorithm 2 GRETA algorithm for positive patterns

Input: Positive pattern P, stream I
Output: Count of trends matched by P in I

```
1: process_pos_pattern(P, I) {
2: V ← ∅, final_count ← 0
3: for each e ∈ I of type E do
4:     Pr ← V.predEvents(P.predTypes(E))
5:     if e.type = start(P) or Pr ≠ ∅ then
6:         V ← V ∪ e, e.count ← (E = start(P)) ? 1 : 0
7:         for each p ∈ Pr do e.count += p.count
8:         if E = end(P) then final_count += e.count
9: return final_count }
```

acyclic graph with a set of vertices V, a set of edges E, and a $final\_count$. Each vertex $v \in V$ corresponds to an event $e \in I$ matched by q. A vertex v has the label ($e.type$ $e.time$: $e.count$) (Theorem 2). For two vertices $v_i, v_j \in V$, there is an edge $(v_i, v_j) \in E$ if their respective events e_i and e_j are adjacent in a trend matched by q. Event v_i is called a *predecessor event* of v_j.

The GRETA graph has different shapes depending on the pattern and the stream. Figure 4(a) shows the graph for the pattern $A+$. Events of type B are not relevant for it. Events of type A are both START and END events. Figure 4(b) depicts the GRETA graph for the pattern SEQ$(A+, B)$. There are no dashed edges since b's may not precede a's.

Based on the GRETA graph, Theorems 1 and 2 define the event trend count computation, i.e., COUNT(*) as defined in Definition 2.

THEOREM 1 (**Correctness of the GRETA Graph**). *Let G be the GRETA graph for a query q and a stream I. Let $\mathcal{P}$ be the set of paths from a START to an END event in G. Let $\mathcal{T}$ be the set of trends detected by q in I. Then, the set of paths $\mathcal{P}$ and the set of trends $\mathcal{T}$ are equivalent. That is, for each path $p \in \mathcal{P}$ there is a trend $tr \in \mathcal{T}$ with same events in the same order and vice versa.*

THEOREM 2 (**Event Trend Count Computation**). *Let G be the GRETA graph for a query q and a stream I and $e \in I$ be an event with predecessor events Pr in G. The intermediate count $e.count$ is the number of (sub) trends in G that start at a START event and end at e. $e.count = \sum_{p \in Pr} p.count$. If e is a START event, $e.count$ is incremented by one. Let End be the END events in G. The final count is the number of trends captured by G. $final\_count = \sum_{end \in End} end.count$.*

The proofs of Theorems 1 and 2 are omitted here due to space constraints. We refer the reader to the extended version of this paper [25].

GRETA Algorithm for a Positive Patterns (Algorithm 2) computes the number of trends matched by the pattern P in the stream I. The set of vertices V in the GRETA graph is initially empty (line 2). Since each edge is traversed exactly once, edges are not stored. When an event e of type E arrives, the method $V.predEvents(P.predTypes(E))$ returns the predecessor events of e in the graph, i.e., previous events of the predecessor types of E (line 4). A START event is always inserted into the graph since it always starts a new trend, while a MID or an END event is inserted only if it has predecessor events (lines 5–6). The count of e is

Algorithm 3 Pattern split algorithm

Input: Pattern P with negative sub-patterns
Output: Set S of sub-patterns of P

```
1: S ← {P}
2: split(P) {
3: switch P do
4:     case P_i+ : S ← S ∪ split(P_i)
5:     case SEQ(P_i, P_j) : S ← S ∪ split(P_i) ∪ split(P_j)
6:     case NOT P_i :
7:         Parent ← S.getPatternContaining(P)
8:         P_i.previous ← Parent.getPrevious(P)
9:         P_i.following ← Parent.getFollowing(P)
10:        S.replace(Parent, Parent − P)
11:        S ← S ∪ {P_i} ∪ split(P_i)
12: return S }
```

increased by the counts of its predecessor events (line 7). If e is a START event, its count is incremented by 1 (line 6). If e is an END event, the final count is increased by the count of e (line 8). This final count is returned (line 9). We prove the correctness of Algorithm 2 in [25]. Its complexity is analyzed in Section 8.

5. PATTERNS WITH NESTED NEGATION

To handle nested patterns with negation, we split the pattern into positive and negative sub-patterns at compile time (Section 5.1). At runtime, we then maintain the GRETA graph for each of these sub-patterns (Section 5.2).

5.1 Static GRETA Template

According to Section 2, negation appears within a sequence preceded and followed by positive sub-patterns. Thus, we focus on the patterns of the form $P =$ SEQ$(P_i,$ NOT $N, P_j)$ below and consider the special cases SEQ$(P_i,$ NOT $N)$ and SEQ(NOT N, P_j) in [25]. The pattern P means that no matches of N may occur between the matches of P_i and P_j. A match of N disqualifies the current matches of P_i from contributing to a trend detected by P. A match of N marks all events in the graph of the *previous* event type $end(P_i)$ as *invalid* to connect to any future event of the *following* event type $start(P_j)$. Only valid events of type $end(P_i)$ connect to events of type $start(P_j)$.

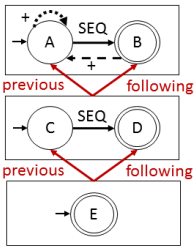


Figure 5: GRETA template for the pattern *(SEQ(A+, NOT SEQ(C, NOT E, D), B))+*

Example 1. The pattern *(SEQ(A+,NOT SEQ(C,NOT E, D),B))+* is split into a positive sub-pattern (SEQ$(A+, B))+$ and two negative sub-patterns SEQ(C, D) and E. Figure 5 illustrates the previous and following connections between a

template for a negative sub-pattern and event types in the template for its parent pattern.

Pattern Split Algorithm. Algorithm 3 consumes a pattern P, splits it into positive and negative sub-patterns, and returns the set S of these sub-patterns. At the beginning, S contains the pattern P (line 1). The algorithm traverses P top-down. If it encounters a negative sub-pattern $P = \mathsf{NOT}\ P_i$, it finds the sub-pattern containing P, called *Parent* pattern, computes the previous and following event types of P_i, and removes P from *Parent* (lines 7–10). The pattern P_i is added to S and the algorithm is called recursively on P_i (line 11). Since the algorithm traverses the pattern P top-down once, the time and space complexity are linear in the size of the pattern s, i.e., $\Theta(s)$.

Definition 4. (**Dependent GRETA Graph**.) Let G_N and G_P be GRETA graph that are constructed according to templates $\mathcal{T}_N$ and $\mathcal{T}_P$ respectively. The GRETA graph G_P is *dependent* on the graph G_N if there is a previous or following connection from $\mathcal{T}_N$ to an event type in $\mathcal{T}_P$.

5.2 Runtime GRETA Graphs

Definition 5. (**Invalid Event**.) Let G_P and G_N be GRETA graphs such that G_P is dependent on G_N. Let $tr = (e_1, \ldots, e_n)$ be a *finished* trend captured by G_N, i.e., e_n is an END event. The trend tr marks all events of the previous event type that arrived before $e_1.time$ as *invalid* to connect to any event of the following event type that will arrive after $e_n.time$.

Example 2. Figure 4(d) depicts the graphs for the sub-patterns from Example 1. The match $e3$ of the negative sub-pattern E marks $c2$ as invalid to connect to any future d. Invalid events are highlighted by a darker background. Analogously, the match $(c5, d6)$ of the negative sub-pattern $\mathsf{SEQ}(C, D)$ marks all a's before $c5$ $(a1, a3, a4)$ as invalid to connect to any b after $d6$. $b7$ has no valid predecessor events and thus cannot be inserted. $a8$ is inserted and all previous a's are connected to it. The marked a's are valid to connect to new a's. $b9$ is inserted and its valid predecessor event $a8$ is connected to it. The marked a's may not connect to $b9$.

Event Pruning. Negation allows us to purge events from the graph to speed-up insertion of new events and aggregation propagation. The following events can be deleted:

• ***Finished Trend Pruning***. A finished trend that is matched by a negative sub-pattern can be deleted once it has invalidated all respective events.

• ***Invalid Event Pruning***. An invalid event of type $end(P_i)$ will never connect to any new event if events of type $end(P_i)$ may precede only events of type $start(P_j)$. The aggregates of such invalid events will not be propagated. Thus, such events may be safely purged from the graph.

Example 3. Continuing Example 2 in Figure 4(d), the invalid $c2$ will not connect to any new event since c's may connect only to d's. Thus, $c2$ is purged. $e3$ is also deleted. Once a's before $c5$ are marked, $c5$ and $d6$ are purged. In contrast, the marked events $a1, a3$, and $a4$ may not be removed since they are valid to connect to future a's.

THEOREM 3. *(**Correctness of Event Pruning**.) Let G_P and G_N be GRETA graphs such that G_P is dependent on G_N. Let G'_P be the same as G_P but without invalid events of type $end(P_i)$ if $P.predTypes(start(P_j)) = \{end(P_i)\}$. Let G'_N be the same as G_N but without finished event trends. Then, G'_P returns the same aggregation results as G_P.*

The proof of Theorem 3 is omitted due to space limitations. Please refer to the extended version of this paper [25].

GRETA Algorithm for Patterns with Negation. Algorithm 2 is called on each event sub-pattern with the following modifications. First, only valid predecessor events are returned in line 4. Second, if the algorithm is called on a negative sub-pattern N and a match is found in line 12, then all previous events of the previous event type of N are either deleted or marked as incompatible with any future event of the following event type of N. Afterwards, the match of N is purged from the graph. GRETA concurrency control is described in Section 7.

6. OTHER LANGUAGE CLAUSES

We now expand our GRETA approach to handle sliding windows, predicates, and grouping.

Sliding Windows. Due to continuous nature of streaming, an event may contribute to the aggregation results in several overlapping windows. Furthermore, events may expire in some windows but remain valid in other windows.

• ***GRETA Sub-Graph Replication***. A naive solution would build a GRETA graph for each window independently from other windows. Thus, it would replicate an event e across all windows that e falls into. Worse yet, this solution introduces repeated computations, since an event p may be predecessor event of e in multiple windows.

Example 4. In Figure 6(a), we count the number of trends matched by the pattern $(\mathsf{SEQ}(A+, B))+$ within a 10-seconds-long window that slides every 3 seconds. The events $a1$–$b9$ fall into window W_1, while the events $a4$–$b9$ fall into window W_2. If a GRETA graph is constructed for each window, the events $a4$–$b9$ are replicated in both windows and their predecessor events are recomputed for each window.

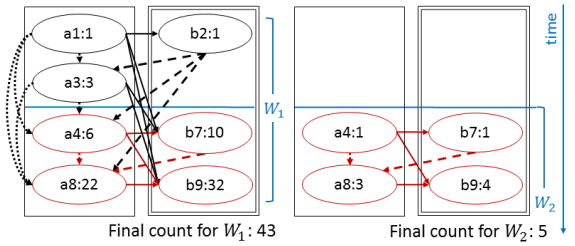
• ***GRETA Sub-Graph Sharing***. To avoid these drawbacks, we share a sub-graph G across all windows to which G belongs. Let e be an event that falls into k windows. The event e is stored once and its predecessor events are computed once across all k windows. The event e maintains a count fro each window. To differentiate between k counts maintained by e, each window is assigned an identifier wid [16]. The count with identifier wid of e ($e.count_{wid}$) is computed based on the counts with identifier wid of e's predecessor events (line 10 in Algorithm 2). The final count for a window wid ($final\_count_{wid}$) is computed based on the counts with identifier wid of the END events in the graph (line 12). In Example 4, the events $a4$–$b9$ fall into two windows and thus maintain two counts in Figure 6(b). The first count is for W_1, the second one for W_2.

Predicates on vertices and edges of the GRETA graph are handled differently by the GRETA runtime.

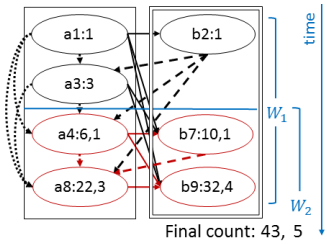
• ***Vertex Predicates*** restrict the vertices in the GRETA graph. They are evaluated on single events to either filter or partition the stream [26].

Local predicates restrict the attribute values of events, for example, *companyID=IBM*. They purge irrelevant events early. We associate each local predicate with its respective state in the GRETA template.

Equivalence predicates require all events in a trend to have the same attribute values, for example, *[company, sector]* in query Q_1. They partition the stream by these attribute values. Thereafter, GRETA queries are evaluated against each sub-stream in a divide and conquer fashion.



(a) GRETA sub-graph replication



(b) GRETA sub-graph sharing

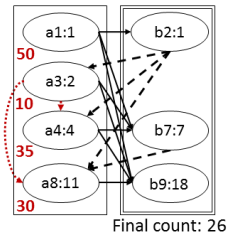


Figure 7: Edge predicate $A.attr < \mathsf{NEXT}(A).attr$

Figure 6: Sliding window WINDOW 10 seconds SLIDE 3 seconds

• ***Edge Predicates*** restrict the edges in the graph (line 4 of Algorithm 2). Events connected by an edge must satisfy these predicates. Therefore, edge predicates are evaluated during graph construction. We associate each edge predicate with its respective transition in the GRETA template.

Example 5. The edge predicate $A.attr < \mathsf{NEXT}(A).attr$ in Figure 7 requires the value of attribute *attr* of events of type A to increase from one event to the next in a trend. The attribute value is shown in the bottom left corner of a vertex. Only two dotted edges satisfy this predicate.

Event Trend Grouping. As illustrated by our motivating examples in Section 1, event trend aggregation often requires event trend grouping. Analogously to A-Seq [26], our GRETA runtime first partitions the event stream into sub-streams by the values of grouping attributes. A GRETA graph is then maintained separately for each sub-stream. Final aggregates are output per sub-stream.

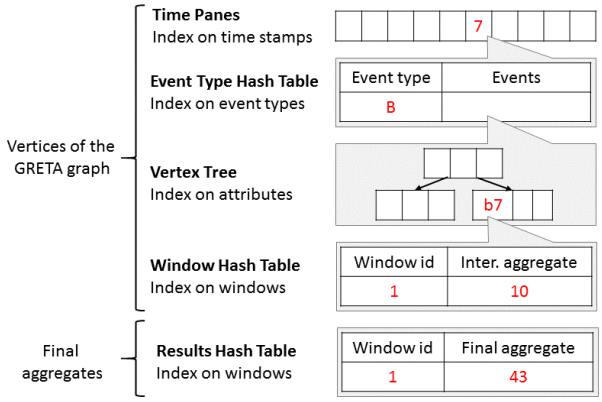


Figure 8: Data structure for a single GRETA graph

7. GRETA FRAMEWORK

Putting Setions 4–6 together, we now describe the GRETA runtime data structures and parallel processing.

Data Structure for a Single GRETA Graph. Edges logically capture the paths for aggregation propagation in the graph. Each edge is traversed *exactly once* to compute the aggregate of the event to which the edge connects (lines 8–10 in Algorithm 2). Hence, edges are not stored.

Vertices must be stored in such a way that the predecessor events of a new event can be efficiently determined (line 4). To this end, we leverage the following data structures. To quickly locate *previous* events, we divide the stream into non-overlapping consecutive time intervals, called ***Time Panes*** [15]. Each pane contains all vertices that fall into it based on their time stamps. These panes are stored in a time-stamped array in increasing order by time (Figure 8). The size of a pane depends on the window specifications and stream rate such that each query window is composed of several panes – allowing panes to be shared between overlapping windows [8, 15]. To efficiently find vertices of *predecessor event types*, each pane contains an ***Event Type Hash Table*** that maps event types to vertices of this type.

To support *edge predicates*, we utilize a tree index that enables efficient range queries. The overhead of maintaining ***Vertex Trees*** is reduced by event sorting and a pane purge mechanism. An event is inserted into the Vertex Tree for its respective pane and event type. This sorting by time and event type reduces the number of events in each tree. Furthermore, instead of removing single expired events from the Vertex Trees, a whole pane with its associated data structures is deleted after the pane has contributed to all windows to which it belongs. To support *sliding windows*, each vertex e maintains a ***Window Hash Table*** storing an aggregate per window that e falls into. Similarly, we store final aggregates per window in the ***Results Hash Table***.

Data Structure for GRETA Graph Dependencies. To support negative sub-patterns, we maintain a ***Graph Dependencies Hash Table*** that maps a graph identifier G to the identifiers of graphs upon which G depends.

Parallel Processing. The grouping clause partitions the stream into sub-streams that are processed in parallel *independently* from each other. Such stream partitioning enables a highly scalable execution as demonstrated in Section 10.4.

In contrast, negative sub-patterns require concurrent maintenance of *inter-dependent* GRETA graphs. To avoid race conditions, we deploy the time-based transaction model [21]. A *stream transaction* is a sequence of operations triggered by all events with the same time stamp on the same GRETA graph. The application time stamp of a transaction (and all its operations) coincides with the application time stamp of the triggering events. For each time stamp t and each GRETA graph G, our time-driven scheduler waits till the processing of all transactions with time stamps smaller than t on the graph G and other graphs that G depends upon is completed. Then, the scheduler extracts all events with the time stamp t, wraps their processing into transactions, and submits them for execution.

8. OPTIMALITY OF GRETA APPROACH

We now analyze the complexity of GRETA. Since a negative sub-pattern is processed analogously to a positive sub-pattern (Section 5), we focus on positive patterns below.

THEOREM 4 (**Complexity**). *Let q be a query with edge predicates, I be a stream, G be the GRETA graph for q and I, n be the number of events per window, and k be the number of windows into which an event falls. The time complexity of GRETA is $O(n^2k)$, while its space complexity is $O(nk)$.*

PROOF. **Time Complexity**. Let e be an event of type E. The following steps are taken to process e. Since events arrive in-order by time stamps (Section 2), the Time Pane to which e belongs is always the latest one. It is accessed in constant time. The Vertex Tree in which e will be inserted is found in the Event Type Hash Table mapping the event type E to the tree in constant time. Depending on the attribute values of e, e is inserted into its Vertex Tree in logarithmic time $O(log_b m)$ where b is the order of the tree and m is the number of elements in the tree, $m \leq n$.

The event e has n predecessor events in the worst case, since each vertex connects to each following vertex under the skip-till-any-match semantics. Let x be the number of Vertex Trees storing previous vertices that are of predecessor event types of E and fall into a sliding window $wid \in windows(e)$, $x \leq n$. Then, the predecessor events of e are found in $O(log_b m + m)$ time by a range query in one Vertex Tree with m elements. The time complexity of range queries in x Vertex Trees is computed as follows:

$$\sum_{i=1}^{x} O(log_b m_i + m_i) = \sum_{i=1}^{x} O(m_i) = O(n).$$

If e falls into k windows, a predecessor event of e updates at most k aggregates of e. If e is an END event, it also updates k final aggregates. Since these aggregates are maintained in hash tables, updating one aggregate takes constant time. GRETA concurrency control ensures that all graphs this graph G depends upon finishing processing all events with time stamps less than t before G may process events with time stamp t. Therefore, all invalid events are marked or purged before aggregates are updated in G at time t. Consequently, an aggregate is updated at most once by the same event. Putting it all together, the time complexity is:

$$O(n(log_b m + nk)) = O(n^2k).$$

Space Complexity. The space complexity is determined by x Vertex Trees and k counts maintained by each vertex.

$$\sum_{i=1}^{x} O(m_i k) = O(nk). \qquad \square$$

THEOREM 5 (**Time Optimality**). *Let n be the number of events per window and k be the number of windows into which an event falls. Then, GRETA has optimal worst-case time complexity $O(n^2k)$.*

PROOF. *Any* event trend aggregation algorithm must process n events to guarantee correctness of aggregation results. Since *any* previous event may be compatible with a new event e under the skip-till-any-match semantics [31], the edge predicates of the query q must be evaluated to decide the compatibility of e with n previous events in worst case. While we utilize a tree-based index to sort events by the most selective predicate, other predicates may have to be evaluated in addition. Thus, each new event must be compared to each event in the graph in the worst case. Lastly, a final aggregate must be computed for each window of q. An event that falls into k windows contributes to k aggregates. In summary, the time complexity $O(n^2k)$ is optimal. $\square$

9. DISCUSSION

In this section, we sketch how our GRETA approach can be extended to support additional language features.

Disjunction and **Conjunction** can be supported by our GRETA approach without changing its complexity because the count for a disjunctive or a conjunctive pattern P can be computed based on the counts for the sub-patterns of P as defined below. Let P_i and P_j be patterns (Definition 1). Let P_{ij} be the pattern that detects trends matched by both P_i and P_j. P_{ij} can be obtained from its DFA representation that corresponds to the intersection of the DFAs for P_i and P_j [28]. Let $\mathsf{COUNT}(P)$ denote the number of trends matched by a pattern P. Let $C_{ij} = \mathsf{COUNT}(P_{ij})$, $C_i = \mathsf{COUNT}(P_i) - C_{ij}$, and $C_j = \mathsf{COUNT}(P_j) - C_{ij}$. In contrast to event sequence and Kleene plus (Definition 1), disjunctive and conjunctive patterns do not impose a time order constraint upon trends matched by their sub-patterns.

Disjunction $(P_i \vee P_j)$ matches a trend that is a match of P_i or P_j. $\mathsf{COUNT}(P_i \vee P_j) = C_i + C_j - C_{ij}$. C_{ij} is subtracted to avoid counting trends matched by P_{ij} twice.

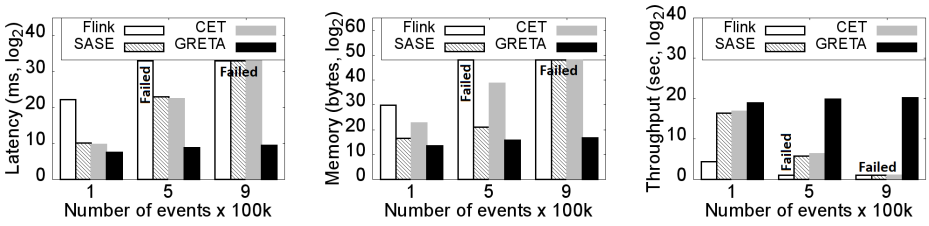
Conjunction $(P_i \wedge P_j)$ matches a pair of trends tr_i and tr_j where tr_i is a match of P_i and tr_j is a match of P_j. $\mathsf{COUNT}(P_i \wedge P_j) = C_i * C_j + C_i * C_{ij} + C_j * C_{ij} + \binom{C_{ij}}{2}$ since each trend detected only by P_i (not by P_j) is combined with each trend detected only by P_j (not by P_i). In addition, each trend detected by P_{ij} is combined with each other trend detected only by P_i, only by P_j, or by P_{ij}.

Kleene Star and **Optional Sub-patterns** can also be supported without changing the complexity because they are syntactic sugar operators. Indeed, $\mathsf{SEQ}(P_i*, P_j) = \mathsf{SEQ}(P_i+, P_j) \vee P_j$ and $\mathsf{SEQ}(P_i?, P_j) = \mathsf{SEQ}(P_i, P_j) \vee P_j$.

Table 1: Event selection semantics

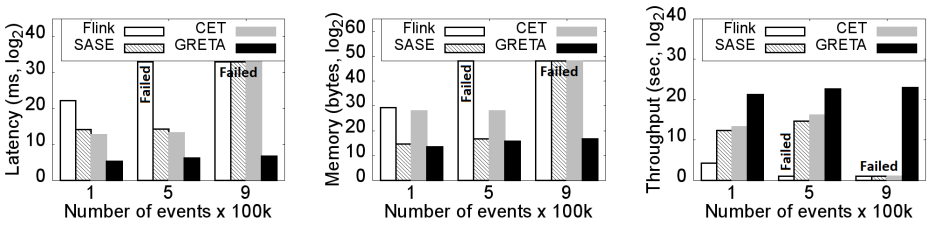
Event selection semantics	**Skipped events**	**Number of trends**
Skip-till-any-match	Any	Exponential
Skip-till-next-match	Irrelevant	Polynomial
Contiguous	None	

Event Selection Semantics are summarized in Table 1. As explained in Section 2, we focus on Kleene patterns evaluated under the most flexible semantics returning all matches, called *skip-till-any-match* in the literature [6, 31, 32]. Other semantics return certain subsets of matches [6, 31, 32]. *Skip-till-next-match* skips only those *events that cannot be matched*, while *contiguous* semantics skips *no* event. To support these semantics, Definition 1 of adjacent events in a trend must be adjusted. Then, fewer edges would be established in the GRETA graph than for skip-till-any-match resulting in fewer trends. Based on this modified graph, Theorem 2 defines the event trend count computation.



(a) Latency (b) Memory (c) Throughput

Figure 9: Positive patterns (Stock real data set)



(a) Latency (b) Memory (c) Throughput

Figure 10: Patterns with negative sub-patterns (Stock real data set)

10. PERFORMANCE EVALUATION

10.1 Experimental Setup

Infrastructure. We have implemented our GRETA approach in Java with JRE 1.7.0_25 running on Ubuntu 14.04 with 16-core 3.4GHz CPU and 128GB of RAM. We execute each experiment three times and report their average.

Data Sets. We evaluate the performance of our GRETA approach using the following data sets.

• ***Stock Real Data Set***. We use the real NYSE data set [5] with 225k transaction records of 19 companies in 10 sectors. Each event carries volume, price, time stamp in seconds, type (sell or buy), company, sector, and transaction identifiers. We replicate this data set 10 times.

• ***Linear Road Benchmark Data Set***. We use the traffic simulator of the Linear Road benchmark [7] for streaming systems to generate a stream of position reports from vehicles for 3 hours. Each position report carries a time stamp in seconds, a vehicle identifier, its current position, and speed. Event rate gradually increases during 3 hours until it reaches 4k events per second.

Table 2: Attribute values

Attribute	Distribution	min–max
Mapper id, job id	Uniform	0–10
CPU, memory	Uniform	0–1k
Load	Poisson with $\lambda = 100$	0–10k

• ***Cluster Monitoring Data Set***. Our stream generator creates cluster performance measurements for 3 hours. Each event carries a time stamp in seconds, mapper and job identifiers, CPU, memory, and load measurements. The distribution of attribute values is summarized in Table 2. The stream rate is 3k events per second.

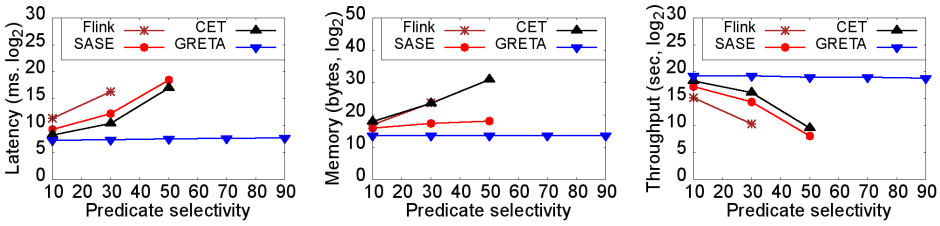
Event Queries. Unless stated otherwise, we evaluate query Q_1 (Section 1) and its nine variations against the stock data set. These query variations differ by the predicate $S.price * X < \mathsf{NEXT}(S).price$ that requires the price to increase (or decrease with $>$) by $X \in \{1, 1.05, 1.1, 1.15, 1.2\}$ percent from one event to the next in a trend. Similarly, we evaluate query Q_2 and its nine variations against the cluster data set, and query Q_3 and its nine variations against the Linear Road data set. We have chosen these queries because they contain all clauses (Definition 2) and allow us to measure the effect of each clause on the number of matched trends. The number of matched trends ranges from few hundreds to trillions. In particular, we vary the number of events per window, presence of negative sub-patterns, predicate selectivity, and number of event trend groups.

Methodology. We compare GRETA to CET [24], SASE [32], and Flink [2]. To achieve a fair comparison, we have implemented CET and SASE on top of our platform. We execute Flink on the same hardware as our platform. While Section 11 is devoted to a detailed discussion of these approaches, we briefly sketch their main ideas below.

• ***CET*** [24] is the state-of-the-art approach to event trend detection. It stores and reuses partial event trends while constructing the final event trends. Thus, it avoids the recomputation of common sub-trends. While CET does not explicitly support aggregation, we extended this approach to aggregate event trends upon their construction.

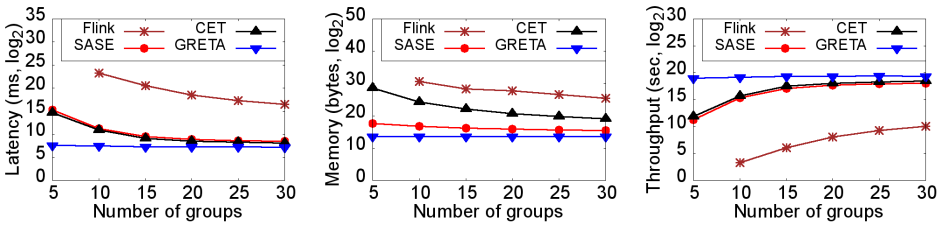
• ***SASE*** [32] supports aggregation, nested Kleene patterns, predicates, and windows. It implements the two-step approach as follows. (1) Each event e is stored in a stack and pointers to e's previous events in a trend are stored. For each window, a DFS-based algorithm traverses these pointers to construct all trends. (2) These trends are aggregated.

• ***Flink*** [2] is an open-source streaming platform that supports event pattern matching. We express our queries using Flink operators. Like other industrial systems [1, 3, 4], Flink



(a) Latency (b) Memory (c) Throughput

Figure 11: Selectivity of edge predicates (Linear Road benchmark data set)



(a) Latency (b) Memory (c) Throughput

Figure 12: Number of event trend groups (Cluster monitoring data set)

does not explicitly support Kleene closure. Thus, we flatten our queries, i.e., for each Kleene query q we determine the length l of the longest match of q. We specify a set of fixed-length event sequence queries that cover all possible lengths from 1 to l. Flink is a two-step approach.

Metrics. We measure common metrics for streaming systems, namely, *latency, throughput*, and *memory*. *Latency* measured in milliseconds corresponds to the peak time difference between the time of the aggregation result output and the arrival time of the latest event that contributes to the respective result. *Throughput* corresponds to the average number of events processed by all queries per second. *Memory* consumption measured in bytes is the peak memory for storing the GRETA graph for GRETA, the CET graph and trends for CET, events in stacks, pointers between them, and trends for SASE, and trends for Flink.

10.2 Number of Events per Window

Positive Patterns. In Figure 9, we evaluate positive patterns against the stock real data set while varying the number of events per window.

Flink does not terminate within several hours if the number of events exceeds 100k because Flink is a two-step approach that evaluates a set of event sequence queries for each Kleene query. Both the unnecessary event sequence construction and the increased query workload degrade the performance of Flink. For 100k events per window, Flink requires 82 minutes to terminate, while its memory requirement for storing all event sequences is close to 1GB. Thus, Flink is neither real time nor lightweight.

SASE. The latency of SASE grows exponentially in the number of events until it fails to terminate for more than 500k events. Its throughput degrades exponentially. Delayed responsiveness of SASE is explained by the DFS-based stack traversal which re-computes each sub-trend tr for each longer trend containing tr. The memory requirement of SASE exceeds the memory consumption of GRETA 50–fold because DFS stores the trend that is currently being constructed. Since the length of a trend is unbounded, the peak memory consumption of SASE is significant.

CET. Similarly to SASE, the latency of CET grows exponentially in the number of events until it fails to terminate for more than 700k events. Its throughput degrades exponentially. In contrast to SASE, CET utilizes the available memory to store and reuse common sub-trends instead of recomputing them. To achieve almost double speed-up compared to SASE, CET requires 3 orders of magnitude more memory than SASE for 500k events.

GRETA consistently outperforms all above two-step approaches regarding all three metrics because it does not waste computational resources to construct and store exponentially many event trends. Instead, GRETA incrementally computes event trend aggregation. Thus, it achieves 4 orders of magnitude speed-up compared to all above approaches. GRETA also requires 4 orders of magnitude less memory than Flink and CET since these approaches store event trends. The memory requirement of GRETA is comparable to SASE because SASE stores only one trend at a time. Nevertheless, GRETA requires 50–fold less memory than SASE for 500k events.

Patterns with Negative Sub-Patterns. In Figure 10, we evaluate the same patterns as in Figure 9 but with negative sub-patterns against the stock real data set while varying the number of events. Compared to Figure 9, the latency and memory consumption of all approaches except Flink significantly decreased, while their throughput increased. Negative sub-patterns have no significant effect on the performance of Flink because Flink evaluates multiple event sequence queries instead of one Kleene query and constructs all matched event sequences. In contrast, negation reduces

the GRETA graph, the CET graph, and the SASE stacks *before* event trends are constructed and aggregated based on these data structures. Thus, both CPU and memory costs reduce. Despite this reduction, SASE and CET fail to terminate for over 700k events.

10.3 Selectivity of Edge Predicates

In Figure 11, we evaluate positive patterns against the Linear Road benchmark data set while varying the selectivity of edge predicates. We focus on the selectivity of edge predicates because vertex predicates determine the number of trend groups (Section 6) that is varied in Section 10.4. To ensure that the two-step approaches terminate in most cases, we set the number of events per window to 10k.

The latency of Flink, SASE, and CET grows exponentially with the increasing predicate selectivity until they fail to terminate when the predicate selectivity exceeds 50%. In contrast, the performance of GRETA remains fairly stable regardless of the predicate selectivity. GRETA achieves 2 orders of magnitude speed-up and throughput improvement compared to CET for 50% predicate selectivity.

The memory requirement of Flink and CET grows exponentially (these lines coincide in Figure 11(b)). The memory requirement of SASE remains fairly stable but almost 22–fold higher than for GRETA for 50% predicate selectivity.

10.4 Number of Event Trend Groups

In Figure 12, we evaluate positive patterns against the cluster monitoring data set while varying the number of trend groups. The number of events per window is 10k.

The latency and memory consumption of Flink, SASE, and CET decrease exponentially with the increasing number of event trend groups, while their throughput increases exponentially. Since trends are constructed per group, their number and length decrease with the growing number of groups. Thus, both CPU and memory costs reduce. In contrast, GRETA performs equally well independently from the number of groups since event trends are never constructed. Thus, GRETA achieves 4 orders of magnitude speed-up compared to Flink for 10 groups and 2 orders of magnitude speed-up compared to CET and SASE for 5 groups.

11. RELATED WORK

Complex Event Processing. CEP approaches like SASE [6, 32], Cayuga [9], ZStream [22], and E-Cube [19] support aggregation computation over event streams. SASE and Cayuga deploy a Finite State Automaton (FSA)-based query execution paradigm, meaning that each query is translated into an FSA. Each run of an FSA corresponds to an event trend. ZStream translates an event query into an operator tree that is optimized based on the rewrite rules and the cost model. E-Cube employs hierarchical event stacks to share events across different event queries.

However, the expressive power of all these approaches is limited. E-Cube does not support Kleene closure, while Cayuga and ZStream do not support the skip-till-any-match semantics nor the GROUP-BY clause in their event query languages. Furthermore, these approaches define no optimization techniques for event trend aggregation. Instead, they handle aggregation as a post-processing step that follows trend construction. This trend construction step delays the system responsiveness as demonstrated in Section 10.

In contrast to the above approaches, A-Seq [26] proposes *online* aggregation of *fixed-length* event sequences. The expressiveness of this approach is rather limited, namely, it supports neither Kleene closure, nor arbitrarily-nested event patterns, nor edge predicates. Therefore, it does not tackle the exponential complexity of event trends.

The CET approach [24] focuses on optimizing the *construction of event trends*. It does not support aggregation, grouping, nor negation. In contrast, our GRETA approach focuses on *aggregation of event trends* without trend construction. Due to the exponential time and space complexity of trend construction, the CET approach is neither real-time nor lightweight as confirmed by our experiments.

Data Streaming. Streaming approaches [8, 10, 13, 15, 16, 30, 33, 34] support aggregation computation over data streams. Some approaches incrementally aggregate *raw input events for single-stream queries* [15, 16]. Others share aggregation results between overlapping sliding windows [8, 15], which is also leveraged in our GRETA approach (Section 4.2). Other approaches share intermediate aggregation results between multiple queries [13, 33, 34]. However, these approaches evaluate simple Select-Project-Join queries with window semantics. Their execution paradigm is set-based. They do not support CEP-specific operators such as event sequence and Kleene closure that treat the order of events as first-class citizens. Typically, these approaches require the *construction of join results* prior to their aggregation. Thus, they define incremental aggregation of *single raw events* but implement a two-step approach for join results.

Industrial streaming systems including Flink [2], Esper [1], Google Dataflow [3], and Microsoft StreamInsight [4] do not explicitly support Kleene closure. However, Kleene closure computation can be simulated by a set of event sequence queries covering all possible lengths of a trend. This approach is possible only if the maximal length of a trend is known apriori – which is rarely the case in practice. Furthermore, this approach is highly inefficient for two reasons. First, it runs a set of queries for each Kleene query. This increased workload degrades the system performance. Second, since this approach requires event trend construction prior to their aggregation, it has exponential time complexity and thus fails to compute results within a few seconds.

Static Sequence Databases. These approaches extend traditional SQL queries by order-aware join operations and support aggregation of its results [14, 20]. However, they do not support Kleene closure. Instead, *single data items* are aggregated [14, 23, 27, 29]. Furthermore, these approaches assume that the data is statically stored and indexed prior to processing. Hence, these approaches do not tackle challenges that arise due to dynamically streaming data such as real-time responsiveness and event expiration.

12. CONCLUSIONS

To the best of our knowledge, our GRETA approach is the first to aggregate event trends that are matched by nested Kleene patterns without constructing these trends. We achieve this goal by compactly encoding all event trends into the GRETA graph and dynamically propagating the aggregates along the edges of the graph during graph construction. We prove that our approach has optimal time complexity. Our experiments demonstrate that GRETA achieves up to four orders of magnitude speed-up and requires up to 50–fold less memory than the state-of-the-art solutions.

13. REFERENCES

[1] Esper. http://www.espertech.com/.

[2] Flink. https://flink.apache.org/.

[3] Google Dataflow. https://cloud.google.com/dataflow/.

[4] Microsoft StreamInsight. https://technet.microsoft.com/en-us/library/ee362541%28v=sql.111%29.aspx.

[5] Stock data. http://davis.wpi.edu/datasets/Stock_Trace_Data/.

[6] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, pages 147–160, 2008.

[7] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: A stream data management benchmark. *PVLDB*, 30(1):480–491, 2004.

[8] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. *PVLDB*, 30(1):336–347, 2004.

[9] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. White. Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422, 2007.

[10] T. M. Ghanem, M. A. Hammad, M. F. Mokbel, W. G. Aref, and A. K. Elmagarmid. Incremental evaluation of sliding-window queries over data streams. *IEEE Trans. on Knowl. and Data Eng.*, 19(1):57–72, 2007.

[11] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

[12] A. Khan. *501 Stock Market Tips and Guidelines.* Writers Club Press, 2002.

[13] S. Krishnamurthy, C. Wu, and M. J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, pages 623–634, 2006.

[14] A. Lerner and D. Shasha. AQuery: Query language for ordered data, optimization techniques, and experiments. *PVLDB*, 29(1):345–356, 2003.

[15] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: Efficient evaluation of sliding window aggregates over data streams. In *SIGMOD*, pages 39–44, 2005.

[16] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*, pages 311–322, 2005.

[17] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier. Out-of-order processing: a new architecture for high-performance stream systems. *PVLDB*, 1(1):274–288, 2008.

[18] M. Liu, M. Li, D. Golovnya, E. A. Rundensteiner, and K. T. Claypool. Sequence pattern query processing over out-of-order event streams. In *ICDE*, pages 784–795, 2009.

[19] M. Liu, E. A. Rundensteiner, K. Greenfield, C. Gupta, S. Wang, I. Ari, and A. Mehta. E-Cube: Multi-dimensional event sequence analysis using hierarchical pattern query sharing. In *SIGMOD*, pages 889–900, 2011.

[20] E. Lo, B. Kao, W.-S. Ho, S. D. Lee, C. K. Chui, and D. W. Cheung. OLAP on sequence data. In *SIGMOD*, pages 649–660, 2008.

[21] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, M. Stonebraker, K. Tufte, and H. Wang. S-Store: Streaming Meets Transaction Processing. *PVLDB*, 8(13):2134–2145, 2015.

[22] Y. Mei and S. Madden. ZStream: A Cost-based Query Processor for Adaptively Detecting Composite Events. In *SIGMOD*, pages 193–206, 2009.

[23] I. Motakis and C. Zaniolo. Temporal aggregation in active database rules. In *SIGMOD*, pages 440–451, 1997.

[24] O. Poppe, C. Lei, S. Ahmed, and E. A. Rundensteiner. Complete event trend detection in high-rate event streams. In *SIGMOD*, pages 109–124, 2017.

[25] O. Poppe, C. Lei, E. A. Rundensteiner, and D. Maier. GRETA: Graph-based Real-time Event Trend Aggregation. http://users.wpi.edu/~opoppe/papers/Greta-full.pdf, 2017. Technical report in progress.

[26] Y. Qi, L. Cao, M. Ray, and E. A. Rundensteiner. Complex event analytics: Online aggregation of stream sequence patterns. In *SIGMOD*, pages 229–240, 2014.

[27] R. Sadri, C. Zaniolo, A. Zarkesh, and J. Abidi. Expressing and optimizing sequence queries in database systems. In *ACM Trans. on Database Systems*, pages 282–318, 2004.

[28] U. Schöning. *Theoretische Informatik - kurzgefaßt (3. Aufl.)*. Spektrum Akademischer Verlag, 1997.

[29] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: Design and Implementation of a Sequence Database System. *PVLDB*, 22(1):99–110, 1996.

[30] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu. General incremental sliding-window aggregation. *PVLDB*, 8(7):702–713, 2015.

[31] E. Wu, Y. Diao, and S. Rizvi. High-performance Complex Event Processing over streams. In *SIGMOD*, pages 407–418, 2006.

[32] H. Zhang, Y. Diao, and N. Immerman. On complexity and optimization of expensive queries in Complex Event Processing. In *SIGMOD*, pages 217–228, 2014.

[33] R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava. Multiple aggregations over data streams. In *SIGMOD*, pages 299–310, 2005.

[34] R. Zhang, N. Koudas, B. C. Ooi, D. Srivastava, and P. Zhou. Streaming multiple aggregations using phantoms. *PVLDB*, 19(4):557–583, 2010.