Event Trend Aggregation Under Rich Event Matching Semantics

Olga Poppe[1], Chuan Lei[2], Elke A. Rundensteiner[3], and David Maier[4]

[1]Microsoft Gray Systems Lab, One Microsoft Way, Redmond, WA 98052
[2]IBM Research - Almaden, 650 Harry Road, San Jose, CA 95120
[3]Worcester Polytechnic Institute, 100 Institute Road, Worcester, MA 01609
[4]Portland State University, 1825 SW Broadway, Portland, OR 97201
olpoppe@microsoft.com,chuan.lei@ibm.com,rundenst@wpi.edu,maier@cs.pdx.edu

ABSTRACT

Streaming applications from cluster monitoring to algorithmic trading deploy Kleene queries to detect and aggregate event trends. Rich event matching semantics determine how to compose events into trends. The expressive power of state-of-the-art streaming systems remains limited since they do not support many of these semantics. Worse yet, they suffer from long delays and high memory costs because they maintain aggregates at a fine granularity. To overcome these limitations, our Coarse-Grained Event Trend Aggregation (Cogra) approach supports a rich variety of event matching semantics within one system. Better yet, Cogra incrementally maintains aggregates at the coarsest granularity possible for each of these semantics. In this way, Cogra minimizes the number of aggregates – reducing both time and space complexity. Our experiments demonstrate that Cogra achieves up to six orders of magnitude speed-up and up to seven orders of magnitude memory reduction compared to state-of-the-art approaches.

CCS CONCEPTS

• **Information systems → Data streaming**; • **Computing methodologies → Optimization algorithms**; • **Theory of computation → Online algorithms**;

KEYWORDS

Complex Event Processing; query optimization; incremental aggregation; event trend

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06.
https://doi.org/10.1145/3299869.3319862

ACM Reference Format:
Olga Poppe, Chuan Lei, Elke A. Rundensteiner, and David Maier. 2019. Event Trend Aggregation, Under Rich Event Matching Semantics. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3319862

1 INTRODUCTION

Complex Event Processing (CEP) is a technology for supporting streaming applications from cluster monitoring to algorithmic trading. CEP systems continuously evaluate Kleene pattern queries against high-rate streams of primitive events to detect higher-level event trends [38, 39]. In contrast to traditional event sequences of fixed length [29], event trends have an arbitrary length. A rich variety of event matching semantics were defined in the CEP literature to determine trend contiguity [11, 49, 50]. Aggregation functions are applied to these trends to derive summarized insights. CEP applications may need to react to critical changes of these aggregates in near real time. We now describe three use cases of time-critical event trend aggregation. Additional scenarios are presented in Appendix A.

```
q1 : RETURN mapper, SUM(L.cpu)
     PATTERN SEQ(Start S, Load L+, End E)
     SEMANTICS contiguous
     WHERE [job,mapper] AND L.cpu < NEXT(L).cpu
     GROUP-BY job,mapper
     WITHIN 1 minute SLIDE 30 seconds
```

• ***Cluster monitoring*** tools, such as Ganglia [32], gather load measurements, combine them with workflow-specific logs (e.g., start and end of Hadoop jobs) to form load distribution trends per job over time. These trends are aggregated to dynamically detect and then tackle cluster bottlenecks, unbalanced load distributions, and data queuing issues by automatically tuning the cluster configuration in near real time. Each event carries a time stamp in seconds, job and mapper identifiers. In addition, Load events carry various load

measurements. Query q_1 detects mappers that experience contiguously increasing CPU load trends per job and computes the total CPU cycles for such mappers and jobs during a time window of 1 minute that slides every 30 seconds. A load distribution trend consists of one Job-start event, any number of Load events, and one Job-end event. All events in one trend must carry the same job and mapper identifiers, as required by the predicate [job, mapper].[1] The predicate L.cpu < NEXT(L).cpu requires the CPU load to increase from one event to the next in a trend. No events may be skipped in between matched events per job and mapper, as expressed by the contiguous semantics.

```
q2 : RETURN district, COUNT(*), AVG(T.speed)
     PATTERN SEQ(Pickup P, Trip T+, Dropoff D)
     SEMANTICS skip-till-next-match
     WHERE [driver, rider, district] GROUP-BY district
     WITHIN 30 minutes SLIDE 1 minute
```

• ***Ridesharing service***. With thousands of drivers and over 150 requests per minute in New York City [10], real-time traffic analytics and ride management are challenging. Uber event stream analytics deploys event aggregation queries for price computation, forecasting, and optimization [8]. Each event carries a time stamp in seconds, district, speed, driver, and rider identifiers. Query q_2 computes the number and average speed of Uber trips per district during a time window of 30 minutes that slides every minute. Each Uber trip corresponds to a sequence of one Pickup event, any number of Trip events, and one Dropoff event. All events that constitute one trip must carry the same driver, rider, and district identifiers, as required by the predicate [driver, rider, district]. The skip-till-next-match semantics allows ignoring irrelevant events, e.g., trip cancellation.

```
q3 : RETURN sector, COUNT(*)
     PATTERN Stock S+
     SEMANTICS skip-till-any-match
     WHERE [company, sector] AND S.price > NEXT(S).price
     GROUP-BY sector
     WITHIN 10 minutes SLIDE 10 seconds
```

• ***Algorithmic trading*** platforms, such as MetaTrader4 [6], evaluate trend aggregation queries against high-rate streams of financial transactions to identify short-term profit opportunities and avoid pitfalls. Each transaction carries a time stamp in seconds, price, company and sector identifiers. Since stock trends of companies that belong to the same industrial sector tend to move as a group [20], query q_3 computes the number of down trends per sector to predict future trends for the sector. If the number of down trends exceeds certain threshold, a sell signal is triggered for the whole sector. Query q_3 detects down-trends within a time window of 10 minutes that slides every 10 seconds. All events that belong to the same trend must carry the same company and sector identifiers, as required by the predicate [company, sector]. The price in such a trend must decrease from one event to the next, as expressed by the predicate S.price>NEXT(S).price. The query may ignore local price fluctuations to detect longer and thus more reliable trends [20]. Such flexible behavior is enabled by the skip-till-any-match semantics.

Challenges. These application scenarios illustrate the following open problems.

• ***Real-time event trend aggregation***. Each event must be considered in the context of other events in the stream forming a trend to draw reliable conclusions. Kleene patterns detect event trends of arbitrary length. However, the number of these trends may grow exponentially in the number of events [50]. With these applications requiring near real-time responsiveness and thus efficient aggregation of event trends, this amounts to a critical dilemma. Thus, any real-time solution must aim to aggregate trends without first constructing them and ideally even without storing all matched events. At the same time, correctness must be guaranteed, i.e., the same aggregates must be returned as by the two-step approach.

• ***Rich event matching semantics***. Event trends are detected in high-rate streams under a rich variety of event matching semantics [11, 49, 50] depending on the application. These semantics range from the most restrictive contiguous semantics (query q_1 above) to the most flexible skip-till-any-match semantics (query q_3). Their execution strategies differ significantly, making the seamless support of online trend aggregation on top of these diverse semantics challenging.

• ***Expressive predicates*** on adjacent events in a trend determine whether an event is matched depending on other events in a trend. Since a new event may be adjacent to any previous event under the skip-till-any-match semantics, all matched events must be kept. The need to store all matched events contradicts the online aggregation requirement that aims to incrementally update aggregates upon event arrival and discard these events immediately thereafter.

In this paper, we tackle the open problems described above with the ultimate objective to minimize the latency of such expressive event trend aggregation queries.

State-of-the-Art Approaches to event aggregation can be divided into the following groups (Tables 1 and 2).

• ***CEP approaches*** such as SASE [50], Flink [2], Cayuga [15], and ZStream [33] support Kleene closure. However, Cayuga and ZStream do not consider diverse event matching semantics. While their languages support aggregation, they do not provide any optimization techniques to compute aggregation on top of Kleene patterns. Instead, they first constructs all trends and then compute their aggregation.

[1] The predicate [job] is commonly used syntactic sugar for the equivalence predicate (S.job = L.job AND L.job = E.job) [11, 38, 39, 49, 50].

This two-step approach suffers from long delays or even fails to return results within several hours due to the exponential time complexity of event trend construction (Section 8).

In contrast, A-Seq [40] Incrementally computes aggregation of fixed-length event sequences. However, A-Seq does not support Kleene closure. Thus, it does not tackle the exponential complexity of event trends. GRETA [39] introduces online event trend aggregation. Since it avoids the expensive event trend construction step, it reduces the time complexity from exponential to quadratic in the number of events compared to the two-step approaches. However, GRETA supports only one semantics, namely, skip-till-any-match. Moreover, it maintains aggregates at the finest granularity per each matched event. We will show that its complexity is not optimal in many cases. This also explains why GRETA returns aggregation results with over an hour long delay for 20 million events per window (Section 8.2). Such long delays are unacceptable for time-critical applications.

• ***Streaming approaches*** [12, 16, 23, 25, 47] evaluate traditional Select-Project-Join queries, i.e., their execution paradigm is set-based. They support neither event sequences nor Kleene closure. They construct join results prior to their aggregation. Thus, they define incremental aggregation of single raw events only. Industrial streaming systems such as Esper [1] and Oracle Stream Analytics [7] only support fixed-length event sequences. They do not explicitly support Kleene closure. They construct all sequences prior to their aggregation and thus follow the two-step approach.

	Event sequences	**Event trends**
Two-step	Oracle Stream Analytics [7], Esper [1]	SASE [50], Cayuga [15] Flink [2], ZStream [33]
Online	A-Seq [40]	GRETA [39]

Table 1: State-of-the-art event aggregation approaches

Our Proposed COGRA Approach is the first to define online event trend aggregation under rich event matching semantics at multiple granularities. COGRA pushes aggregation inside the Kleene closure computation. Thus, it avoids the event trend construction step which is known to suffer from exponential complexity [50]. Similarly to SQL queries computing aggregation over join, depending on query features, COGRA adaptively selects the coarsest possible granularity at which it incrementally computes trend aggregation. These granularities range from coarse (per pattern), to medium (per event type), to fine (per matched event) as per Figure 1. COGRA minimizes the number of maintained aggregates and succeeds to discard all events as soon as they have been used to update their respective type-grained or pattern-grained aggregates. Thus, our approach represents a win-win solution that reduces both time and space complexity of trend aggregation compared to state-of-the-art methods [2, 39, 40, 50].

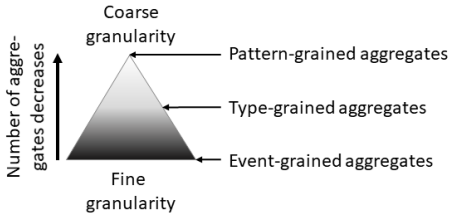


Figure 1: Event trend aggregation at different granularities

Contributions. The key innovations of COGRA include:

1) We define the problem of real-time event trend aggregation under rich event matching semantics. Based on these semantics and other query features, we determine the coarsest granularity at which event trend aggregates are maintained.

2) For each granularity, we propose efficient data structures and algorithms to incrementally compute event trend aggregation. We prove the correctness of these algorithms.

3) We prove that COGRA represents a win-win solution that guarantees optimal time complexity, while keeping space complexity linear in the number of events in the worst case.

4) Our experiments on real data sets [4, 10, 41] demonstrate that for 100 million events per window, the latency of COGRA stays within 3 seconds, while its throughput exceeds 39 million events per second. Based on these results, we conclude that COGRA enables real-time in-memory event trend aggregation as required for time-critical applications.

Outline. Section 2 describes our data and query model. Section 3 provides an overview of the COGRA framework. For each event matching semantics, Sections 4–6 define incremental event trend aggregation at different granularities. We consider the remaining query clauses in Section 7. Section 8 describes our experimental study. Section 9 discusses related work, while Section 10 concludes the paper.

2 DATA AND QUERY MODEL

2.1 Basic Notions and Assumptions

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

An event is a message indicating that something of interest to the application happened in the real world. An event e has a time stamp $e.time \in \mathbb{T}$ assigned by the event source. An event e belongs to a particular event type E, denoted $e.type{=}E$ and described by a schema that specifies the set of event attributes and the domains of their values.

Events are sent by event producers (e.g., sensors) on an event stream I. An event consumer (e.g., health care system) continuously monitors the stream with event queries. We borrow the query language and semantics from SASE [11, 49, 50]. Queries in Section 1 are expressed using this syntax.

Approaches	Kleene closure	Skip-till-any-match	Skip-till-next-match	Contiguous semantics	Predicates on adjacent events	Event trend grouping	Online sequence/ trend aggregation
Flink	+	+	+	+	+	+	–
SASE	+	+	+	+	+	+	–
GRETA	+	+	–	–	+	+	+
A-Seq	–	+	–	–	–	+	+
Cogra	**+**	**+**	**+**	**+**	**+**	**+**	**+**

Table 2: Expressive power of the event aggregation approaches

Definition 2.1 (Pattern). A pattern has the form E, $P+$, or SEQ(P_1, P_2), where E is an event type, P, P_1, P_2 are patterns, + is a Kleene plus operator, and SEQ is an event sequence operator. P is a sub-pattern of $P+$, while P_1 and P_2 are sub-pattern of SEQ(P_1, P_2). If a pattern contains a Kleene plus operator, it is called a Kleene pattern. It is an event sequence pattern otherwise. The length of a pattern is the number of event types in it. □

Assumptions. We focus on Kleene patterns that allow us to specify arbitrarily long event pattern matches, called event trends (Definitions 2.4, 2.6, and 2.8). To simplify our discussion, we first consider Kleene patterns that do not contain negation, Kleene star, optional sub-patterns, conjunction, nor disjunction. Also, an event type may appear at most once in a pattern. We later sketch extensions of our approach to relax these assumptions in Appendix C.

We focus on event queries with predicates on single events and on pairs of adjacent events in a trend (Section 3.2). Such predicates allow us to specify expressive queries in diverse application domains, e.g., queries q_1–q_3 in Section 1. Predicates on non-adjacent events are subject for future research.

Lastly, we assume that events arrive in order by their time stamps. Existing approaches can be applied to solve the orthogonal problem of out-of-order events [13, 27, 28, 45].

2.2 Event Matching Semantics

Event matching semantics [11, 49, 50] constrain event contiguity in a trend to express queries for diverse streaming applications. These semantics differentiate between relevant events, i.e., events that can extend an existing partial trend (Definition 3.1), and irrelevant events that cannot. Relevant events either must extend existing trends or can be skipped to preserve opportunities for alternative trends. Irrelevant events either invalidate partial trends or can be skipped.

Example 2.2. In Figure 2, the pattern P = *(SEQ(A+,B))+* is evaluated under various event matching semantics against the stream I (depicted at the bottom of the figure). In the stream, letters denote types, while numbers represent time stamps, e.g., $a1$ is an event of type A with time stamp 1. Matched trends are depicted above the stream. They range from the shortest contiguous trend $(a1, b2)$[2] to the longest non-contiguous trend $(a1, b2, a3, a4, b6, a7, b8)$.

Skip-Till-Any-Match Semantics (ANY) is the most flexible semantics that detects all possible trends as follows. For each event e and each partial trend tr that can be extended by e, two possibilities are considered: (1) e is appended to the trend tr to form a longer trend $tr' = (tr, e)$, and (2) e is skipped and the trend tr remains unchanged to preserve opportunities for alternative longer trends. If an event e can extend all existing trends, then e doubles the number of trends. Thus, the number of trends grows exponentially in the number of events in the worst case. Skip-till-any-match semantics skips irrelevant events. Query q_3 in Section 1 is evaluated under this semantics.

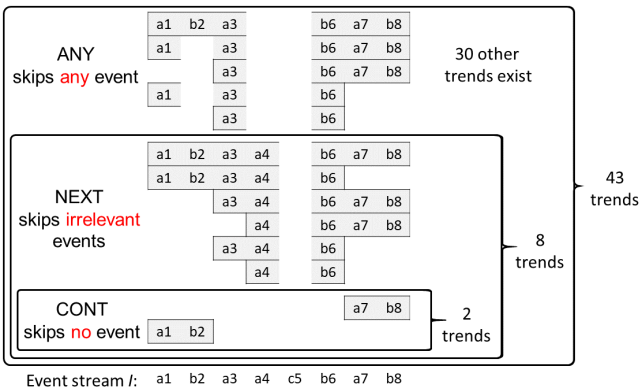


Figure 2: Event trends matched by the Kleene pattern *P=(SEQ(A+, B))+* under various event matching semantics

Example 2.3. In Figure 2, when $a7$ arrives, the trend $(a3, b6)$ is extended to $(a3, b6, a7)$ and the original trend $(a3, b6)$ is also kept. Based on only eight events in the stream, 43 trends are detected. Only some of them are shown for compactness. Irrelevant events are ignored, e.g., $c5$.

Definition 2.4 (Event Trend Under Skip-Till-Any-Match). An event type E matches an event $e \in I$ of type E under skip-till-any-match, denoted $e \in trends_{any}(E, I)$.

[2]Constraints on minimal trend length exclude too short and thus not meaningful trends as described in Appendix C.

Let P_1, P_2 be patterns and $e_1, \ldots, e_k \in I$ be events. An event sequence pattern SEQ(P_1, P_2) matches the trend $s = (e_1, \ldots, e_k)$ under skip-till-any-match, denoted $s \in trends_{any}($ SEQ$(P_1, P_2), I)$, if the following two constraints hold:

• Pattern constraint: $(e_1, \ldots, e_m) \in trends_{any}(P_1, I)$ and $(e_{m+1}, \ldots, e_k) \in trends_{any}(P_2, I)$.

• Temporal order constraint: $e_m.time < e_{m+1}.time$.

Let P be a pattern, $s_1, \ldots, s_k$ be trends, and ":" denote concatenation. A Kleene plus pattern $P+$ matches the trend $tr = (s_1 : \cdots : s_k)$ under skip-till-any-match, denoted $tr \in trends_{any}(P+, I)$, if the following constraints hold:

• Pattern constraint: $\forall s_l \in \{s_1, \ldots, s_k\}. s_l \in trends_{any}(P, I)$.

• Temporal order constraint: $s_l.end.time < s_{l+1}.start.time$.

Start and end events of a trend are defined in Table 3. □

Due to the temporal order constraint in Definition 2.4, an event e can appear at most once in any trend tr. But an event e can appear in k different trends ($k \in \mathbb{N}, k \geq 0$).

Skip-Till-Next-Match Semantics (NEXT) is more restrictive than ANY because NEXT requires that all relevant events are matched. It allows skipping irrelevant events however. Query q_2 in Section 1 is evaluated under this semantics.

Example 2.5. In Figure 2, the trend $(a3, b6)$ does not conform to this semantics since it skipped over the relevant event $a4$. In contrast, the trend $(a3, a4, b6)$ is valid since $c5$ is an irrelevant event. Hence the trend $(a3, a4, b6)$ skips no relevant events between $a3$ and $b6$.

Definition 2.6 (Event Trend Under Skip-Till-Next-Match). A pattern P matches a trend tr under skip-till-next-match, denoted $tr \in trends_{next}(P, I)$, if the following constraints hold:

• Pattern and temporal order constraints (Definition 2.4).

• Relevance constraint: Let $tr = (e_1, \ldots, e_n)$ be a finished trend and $(e_1, \ldots, e_m)$, $m \leq n$, be a partial trend of P (Definition 3.1). $\nexists e \in I$ such that $e.time < e_m.time$, e_{m-1} and e are adjacent under skip-till-next-match (Definition 3.2) and e is not part of the trend tr. □

Trend tr	$tr.start$	$tr.mid$	$tr.end$
e	e	$\emptyset$	e
$(e_1, \ldots, e_k)$	e_1	$\{e_2, \ldots, e_{k-1}\}$	e_k
$(s_1 : \cdots : s_k)$	$s_1.start$	$\{e \mid e$ is in $tr, e \neq s_1.start, e \neq s_k.end\}$	$s_k.end$

Table 3: Start, mid, and end events of a trend

Contiguous Semantics (CONT) is the most restrictive semantics since it does not skip events. Query q_1 in Section 1 detects contiguous trends.

Example 2.7. In Figure 2, $(a1, b2)$ and $(a7, b8)$ are the only contiguous trends. Since $c5$ cannot be ignored, $a1$–$a4$ cannot form contiguous trends with later events.

Definition 2.8 (Event Trend Under Contiguous Semantics). A pattern P matches a trend tr under the contiguous semantics, denoted $tr \in trends_{cont}(P, I)$, if the constraints below hold:

• Pattern and temporal order constraints (Definition 2.4).

• Contiguity constraint: $\nexists e \in I$ such that $tr.start.time < e.time < tr.end.time$ and e is not part of the trend tr. □

By Definitions 2.4, 2.6, and 2.8, the skip-till-next-match and contiguous semantics impose additional relevance and contiguity constraints on trends matched under skip-till-any-match. Contiguity constraint is more strict than relevance constraint. Thus, there is a containment relation among the sets of trends matched by a pattern P under these semantics as illustrated in Figure 2.

2.3 Event Trend Aggregation Query

An event trend aggregation query constrains the trends that are detected under various event matching semantics by predicates, grouping, and windows as follows.

Definition 2.9 (Event Trend Aggregation Query). An event trend aggregation query q consists of six clauses:

• Aggregation result specification (RETURN clause),

• Kleene pattern P (PATTERN clause),

• Event matching semantics S (SEMANTICS clause),

• Predicates θ (optional WHERE clause),

• Grouping G (optional GROUP-BY clause), and

• Window w (WITHIN and SLIDE clause).

A query q matches a trend tr if the following conditions hold:

• Pattern, temporal order, (relevance or contiguity) constraints (Definitions 2.4, 2.6, and 2.8).

• Predicate constraint: All events in tr satisfy the predicates θ.

• Grouping constraint: All events in tr carry the same values of the grouping attributes G.

• Window constraint: All events in tr belong to the same window w. □

Aggregation Functions. Within each window of query q, event trends are grouped by the values of grouping attributes G. Aggregates are 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 [17].

COUNT$(*)$ returns the number of trends per group. Let tr.COUNT*(E)* be the number of events of type E in a trend tr. COUNT*(E)* corresponds to the sum of tr.COUNT(E) of all trends tr per group.

Let tr.MIN*(E.attr)* be the minimal value of an attribute *attr* of events of type E in a trend tr. MIN*(E.attr)* returns the minimal value of tr.MIN$(E.attr)$ of all trends tr per group. MAX*(E.attr)* is defined analogously to MIN$(E.attr)$.

Let *tr*.SUM*(E.attr)* be the sum of values of an attribute *attr* of events of type E in a trend tr. SUM*(E.attr)* corresponds to the sum of *tr*.SUM*(E.attr)* of all trends tr per group. Lastly, AVG*(E.attr)* = SUM*(E.attr)* / COUNT*(E)* per group.

As discussed in Section 2.2, an event e may appear in k different trends ($k \in \mathbb{N}, k \geq 0$). In this case, e contributes k times to an aggregate. In contrast, distinct event aggregation enforces that e contributes at most once to an aggregate. Distinct event aggregation is subject to future research.

3 COGRA APPROACH OVERVIEW

Problem Statement. To support time-critical streaming applications, we solve the following event trend aggregation problem. Given an event trend aggregation query q (Definition 2.9) evaluated under an event matching semantics (Definitions 2.4, 2.6, and 2.8) over an event stream I, our goal is to compute the results of q with minimal latency. Figure 3 illustrates our Cogra framework. To select the granularity at which the aggregates are maintained for a query q (Section 3.3), the Static Query Analyzer analyzes the pattern of q (Section 3.1) and classifies the predicates of q (Section 3.2). The results of this query analysis are encoded into the Cogra configuration to guide our Runtime Executor (Sections 4–6).

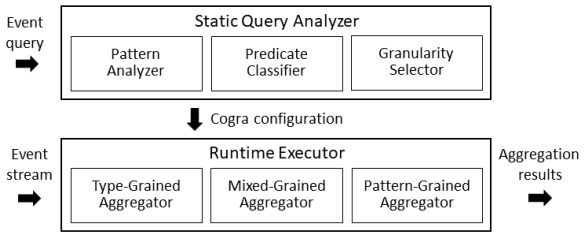


Figure 3: Cogra framework

3.1 Pattern Analyzer

To facilitate the analysis of a pattern P, we follow the traditional approach that translates P into its Finite State Automaton (FSA)-based representation [11, 15, 33, 39, 49, 50]. We briefly describe this translation here. The algorithm can be found in the literature [39].

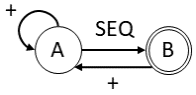


Figure 4: FSA representation of the pattern *P=(SEQ(A+,B))+*

States are labeled by event types in the pattern P. According to our assumption in Section 2.1, a type may occur at most once in P. Thus, state labels are distinct. The first state is the start type *start(P)* and the final state is the end type *end(P)*. All other states are labeled by middle types *mid(P)*. There is exactly one start type, exactly one end type, and any number of middle types in P [39]. In Figure 4, *start(P)* = A, *end(P)* = B, and $mid(P) = \emptyset$, meaning that a trend matched by P always starts with an event a (i.e., an event of type A) and ends with an event b (i.e., an event of type B).

Transitions are labeled by operators in P. They connect types of events that are adjacent in a trend matched by P (Definition 3.2). If a transition connects a type E' with a type E, then E' is called a predecessor type of E, denoted $E' \in P.predTypes(E)$. In Figure 4, $P.predTypes(A) = \{A, B\}$ and $P.predTypes(B) = \{A\}$, meaning that an event a may be preceded by previously matched a's and b's, while an event b is preceded by previously matched a's.

Definition 3.1 (Finished and Partial Event Trend). Given a pattern P evaluated under an event matching semantics S (Definitions 2.4, 2.6, and 2.8), a trend $tr = (e_1, \ldots, e_n) \in trends_S(P, I)$ is called a finished trend of P, while a trend $(e_1, \ldots, e_m)$, $m \leq n$, is called a partial trend of P. □

Definition 3.2 (Adjacent Events, Predecessor Event). Let $e_p, e \in I$ be events such that e_p is in a partial trend matched by a query q and e is new. The events e_p and e are adjacent under the skip-till-any-match semantics in a window w if they satisfy the following constraints:

(1) Pattern constraint: $e_p.type \in P.predTypes(e.type)$.

(2) Temporal order constraint: $e_p.time < e.time$.

(3) Predicate constraint: e_p and e satisfy the predicates θ.

(4) Grouping constraint: e_p and e have the same values of grouping attributes G.

(5) Window constraint: e_p and e belong to the same window w.

The events e_p and e are adjacent under the skip-till-next-match semantics in a window w if constraints 1–5 above and the relevance constraint hold:

(6) Relevance constraint: $\nexists e' \in I$ such that $e'.time < e.time$ and e_p and e' are adjacent in a window w under skip-till-any-match.

The events e_p and e are adjacent under the contiguous semantics in a window w if constraints 1–5 above and the contiguity constraint hold:

(7) Contiguity constraint: $\nexists e' \in I$ such that $e_p.time < e'.time < e.time$.

If e_p and e are adjacent in a window w, then e_p is called a predecessor event of e in the window w. □

3.2 Predicate Classifier

We distinguish between predicates on single events and predicates on adjacent events since they determine the granularity at which aggregates are maintained (Section 3.3).

Predicates on single events either filter or partition the stream. For example, query q_4 in Appendix A uses the predicate (A.type = passive) to select only passive activities and the predicate [patient] to partition the stream by patient.

Predicates on adjacent events restrict the adjacency relation between events in a trend. For example, the predicate (L.cpu < NEXT(L).cpu) of query q_1 requires CPU load measurements to increase from one event to the next in a trend.

3.3 Granularity Selector

The number of event trends matched by a pattern P is determined by the presence of Kleene plus in P and the semantics under which P is evaluated (Table 4). The number of event trends matched by P ranges from linear to exponential in the number of matched events in the worst case [40, 50].

Event matching semantics	Event sequence pattern	Kleene pattern
ANY	Polynomial	Exponential
NEXT, CONT	Linear	Polynomial

Table 4: Number of trends in the number of events [40, 50]

State-of-the-art two-step approaches [1, 2, 7, 15, 50] first construct all event trends (Figure 2) and then compute their aggregation. These approaches are not feasible in real-time applications, since they suffer from the exponential overhead of event trend construction in the worst case (Table 4).

To overcome this limitation, we omit the trend construction step and incrementally compute trend aggregation. Events are discarded once they have been used to update aggregates. Better yet, depending on the semantics of a query q, our granularity selector chooses the coarsest granularity at which trend aggregates are maintained by q such that both correctness and optimal time complexity of trend aggregation are guaranteed for q. More precisely, our granularity selector decides whether to maintain trend aggregates for q at the pattern, type, or mixed granularity as follows (Table 5).

Event matching semantics	Predicates on adjacent events	
	without	with
ANY	Type	Mixed
NEXT, CONT	Pattern	

Table 5: Granularity selection

Type-grained aggregator. If the query q is evaluated under the skip-till-any-match semantics and has no predicates on adjacent events, our executor maintains an aggregate per each event type in the pattern (Section 4).

Mixed-grained aggregator. If q is evaluated under skip-till-any-match and has predicates on adjacent events θ, our executor maintains the aggregates at mixed granularities, namely, either per event e if e is required to evaluate the predicates θ or per event type *e.type* otherwise (Section 5).

Pattern-grained aggregator. If the query q is evaluated under the contiguous or skip-till-next-match semantics, our executor adopts the pattern-grained aggregation strategy. Namely, only the final aggregate of q and the intermediate aggregate of the last matched event are kept (Section 6).

In Sections 4–6, we present the core Cogra techniques under the assumption that the number of input events is finite and they belong to the same window and same event trend group to keep the discussion focused. In Section 7, we then describe how Cogra can be generalized to support windows, grouping, and other query features.

4 TYPE-GRAINED AGGREGATOR

To overcome the exponential time overhead of trend construction under the skip-till-any-match semantics (Table 4), we now propose to incrementally compute event trend aggregation at the event type granularity. Coarse-grained trend aggregation under skip-till-any-match is complicated by the flexibility of this semantics, especially, by its ability to skip any event in between adjacent events in a trend (Definition 2.4). Thus, any previously matched event e_p may be a predecessor event of a new event e in a trend (Definition 3.2). Therefore, each previously matched event e_p must be kept in order to evaluate predicates on adjacent events while deciding whether e_p and a new event e are adjacent. This requirement contradicts our goal of incrementally computing trend aggregation and immediately discarding all events.

However, if the query has no predicates on adjacent events, then incremental trend aggregation is possible at the type granularity. Indeed, when a new event e of type E arrives, all previously matched events of predecessor types of E in a pattern P are adjacent to e (Definition 3.2).Thus, an aggregate can be assigned to each type in the pattern P. The event e updates the aggregate of E and is discarded thereafter. The final aggregate corresponds to the aggregate of the end type of P. In the following, we illustrate the type-grained trend count computation, i.e., COUNT(*). The same principles apply to other aggregation functions as defined in Appendix C.

Example 4.1. Continuing our running example in Figure 2, the type-grained trend count computation is shown in Table 6. For example, when event $a7$ is matched, the intermediate count $a7.count$ captures the number of partial trends that end at $a7$. According to our predecessor relationship analysis in Section 3.1, all previously matched a's and b's are adjacent to $a7$. Thus, $a7.count$ is set to the sum of the counts of all previously matched a's and b's to accumulate the number of trends extended by $a7$. We further increment $a7.count$ by one since $a7$ is of a start type of the pattern P and thus begins one new trend (Section 3.1).

$$a7.count = A.count + B.count + 1 = 10 + 11 + 1 = 22.$$

$A.count$ accumulates the number of partial trends that end at any a. Thus, each a increments $A.count$ by $a.count$.

$$A.count = A.count + a7.count = 10 + 22 = 32.$$

$B.count$ is computed analogously to $A.count$. Since an event b is of an end type of the pattern P, it finishes all trends that it extends. Thus, $B.count$ corresponds to the final count. 43 event trends are detected in this example (Figure 2).

$$b8.count = A.count = 32.$$
$$B.count = B.count + b8.count = 11 + 32 = 43.$$

Only the most recent values of the type-grained aggregates are stored at a time. They are highlighted in bold in Table 6. An event grained aggregate is discarded after it contributed to its respective type-grained aggregates.

Event	a.count	b.count	**A.count**	**B.count**
a1	1		1	
b2		1		1
a3	3		4	
a4	6		10	
b6		10		11
a7	22		32	
b8		32		43

Table 6: Type-grained trend count

Theorem 4.2. *(Type-Grained Trend Count). Let q be a query that is evaluated under skip-till-any-match and has no predicates on adjacent events, P be its pattern, E be an event type in P, and $e \in I$ be an event of type E. Then the event-grained count $e.count$ associated with the event e corresponds to the number of (partial) trends that end at e:*

$$e.count = \sum_{E' \in P.predTypes(E)} E'.count.$$

If $E = start(P)$, $e.count$ is incremented by one.

The type-grained trend count $E.count$ associated with the type E captures the number of (partial) trends that end at an event of type E:

$$E.count = \sum_{e.type=E} e.count.$$

The final count corresponds to the number of finished trends matched by q:

$$final\_count = end(P).count.$$

Theorem 4.2 is proven by induction in Appendix B.

Type-Grained Trend Count Algorithm maintains a hash table H that maps each type E in the pattern P to the count for E. Initially, all counts are set to 0 (Lines 1–2 in Algorithm 1). For each event e of type E, $e.count = 1$ if E is a start type of P. Otherwise, $e.count = 0$ (Lines 3–5). For each

Algorithm 1 Type-grained trend count algorithm

Input: Query q with pattern P, event stream I
Output: Count of event trends matched by q in I

```
1: H ← empty hash table
2: for each event type E in P do H.put(E, 0)
3: for each e ∈ I of type E do
4:     if E = start(P) then e.count ← 1
5:     else e.count ← 0
6:     for each E' ∈ P.predTypes(E) do
7:         e.count += H.get(E')
8:     E.count ← H.get(E) + e.count; H.put(E, E.count)
9: return H.get(end(P))
```

predecessor type E' of E, $e.count$ is incremented by $E'.count$ (Lines 6–7). $E.count$ is incremented by $e.count$ (Line 8). The count of the end type of P is returned (Line 9).

Theorem 4.3 (Complexity). *Let q be a query that is evaluated under skip-till-any-match and has no predicates on adjacent events. Let P be its pattern of length l and n be the number of events per window of q. Algorithm 1 has linear time complexity: $O(n)$ and linear space complexity: $\Theta(l)$.*

Proof. For each matched event of type E, the type-grained trend counts of all predecessor types of E are accessed. E has l predecessor types in the worst case. The length of the pattern l is negligible compared to the number of events per window n for high-rate events streams [4, 10, 41] and meaningful patterns. Thus, the time complexity of Algorithm 1 is linear in n, i.e., $O(n)$. Its space complexity is determined by the number of counts. Since one count is stored per type, the space costs are linear in l, i.e., $\Theta(l)$. □

Theorem 4.4 (Time Optimality). *The linear time complexity $O(n)$ of Algorithm 1 is optimal.*

Proof. Any event aggregation algorithm must process n events to ensure correctness of the final count. Even if one event is not processed, the final result may be incorrect. Thus, the time complexity $O(n)$ of Algorithm 1 is optimal. □

5 MIXED-GRAINED AGGREGATOR

We now extend our coarse-grained trend aggregation techniques to the most general class of queries under skip-till-any-match with predicates on adjacent events θ. To this end, we propose to maintain aggregates at mixed granularities. Namely, we classify the event types in a pattern P into two disjoint sets $\mathcal{T}_e$ and $\mathcal{T}_t$. Events of types $\mathcal{T}_e$ must be stored to evaluate the predicates θ as new events arrive. Thus, an event-grained trend aggregate is computed for each event of type in $\mathcal{T}_e$. In contrast, events of types $\mathcal{T}_t$ do not have to be kept. Thus, a type-grained trend aggregate is maintained for

each type in $\mathcal{T}_t$. Only in the extreme case when expressive predicates restrict all events (i.e., $\mathcal{T}_t = \emptyset$), Cogra defaults to the fine-grained trend aggregation [39].

Example 5.1. Continuing our example in Figure 2, assume that the predicates θ restrict the adjacency relations between b's and a's. When an a arrives, we have to compare it to each previously matched b to select those b's that satisfy the predicates θ. Thus, event-grained trend counts must be maintained for b's (Table 7). In contrast, all a's are adjacent to the following b's. Thus, a type-grained trend count can be maintained for type A. Assuming that $a7$ is adjacent to $b2$ but not to $b6$, $a7.count$ is computed based on the mixed-grained trend counts as follows:

$$\begin{aligned} a7.count &= A.count + \textstyle\sum_{(b,a7) \text{ satisfy } \theta} b.count + 1 \\ &= A.count + b2.count + 1 = 10 + 1 + 1 = 12. \end{aligned}$$

All matched b's, their counts, and the most recent values of type-grained counts are stored at a time (Table 7).

Event	a.count	**b.count**	**A.count**	**B.count**
a1	1		1	
b2		1		1
a3	3		4	
a4	6		10	
b6		10		11
a7	12		22	
b8		22		33

Table 7: Mixed-grained trend count

Theorem 5.2 (Mixed-Grained Trend Count). *Let q be a query that is evaluated under skip-till-any-match, θ be its predicates on adjacent events, P be its pattern, attr and attr'' be attributes of types E and E'' in P respectively, and $\circ \in \{>, \geq, <, \leq, =, \neq\}$ be a comparison operator. A type-grained trend count is maintained for a type E if either there is no predicate of the form $(E.attr \circ E''.attr'')$ in θ or $E \notin P.predTypes(E'')$. Otherwise, an event-grained trend count is computed for each matched event of type E. Let $\mathcal{T}_t$ ($\mathcal{T}_e$) be the set of event types in P for which type-grained (event-grained) trend counts are maintained. Let $e \in I$ be an event of type E. A mixed-grained trend count is computed as follows:*

$$\begin{aligned} e.count = \; & \textstyle\sum_{E' \in \mathcal{T}_t \cap P.predTypes(E)} E'.count + \\ & \textstyle\sum_{e_p.type \in \mathcal{T}_e \cap P.predTypes(E),\ (e_p, e) \text{ satisfy } \theta} e_p.count. \end{aligned}$$

If $E = start(P)$, $e.count$ is incremented by one. $E.count$ and $final\_count$ are computed as defined in Theorem 4.2.

Theorem 5.2 can be proven similarly to Theorem 4.2.

Mixed-Grained Trend Count Algorithm. At compile time, for each type E in the pattern P, Algorithm 2 decides whether to maintain a single type-grained trend count $E.count$ or an event-grained trend count for each matched event of type E. Type-grained trend counts are maintained in a hash table H. Initially, the table H contains all types P with counts set to 0 (Lines 1–2). For each predicate that restricts the adjacency relation between events of type E and events of type E'', if E is a predecessor type of E'', then E is removed from the table H since event-grained trend counts must be computed for events of type E (Lines 3–4).

When event e of type E arrives at runtime, $e.count = 1$ if E is a start type of P. Otherwise, $e.count = 0$ (Lines 5–7). For each predecessor type E' of E, if a type-grained trend count $E'.count$ is maintained, $e.count$ is incremented by $E'.count$ (Lines 8–9). If event-grained trend counts are computed for events of type E', $e.count$ is incremented by the count of each predecessor event e' of type E' (Lines 10–11). If a type-grained trend count $E.count$ is maintained, $E.count$ is incremented by $e.count$ in the table H (Lines 12–13). If event-grained trend counts for events of type E are computed and E is an end type of P, then the final count is incremented by $e.count$ (Line 14). Lastly, if a type-grained trend count is maintained for the end type of P, it is returned as a result. Otherwise, $final\_count$ is returned (Lines 15–16).

Algorithm 2 Mixed-grained trend count algorithm

Input: Query q with pattern P and predicates θ, stream I
Output: Count of event trends matched by q in I

```
1:  H ← empty hash table; V ← ∅; final_count ← 0
2:  for each event type E in P do H.put(E, 0)
3:  for each (E.attr Op E''.attr'') ∈ θ do
4:      if E ∈ P.predTypes(E'') then H.remove(E)
5:  for each e ∈ I of type E do
6:      if E = start(P) then e.count ← 1
7:      else e.count ← 0
8:      for each E' ∈ P.predTypes(E) do
9:          if E' ∈ H then e.count += H.get(E')
10:         else for each e_p ∈ V.predEvents(e) of type E'
11:             do V ← V ∪ e; e.count += e_p.count
12:     if E ∈ H then E.count ← H.get(E) + e.count
13:         H.put(E, E.count)
14:     else if E = end(P) then final_count += e.count
15: if end(P) ∈ H then return H.get(end(P))
16: else return final_count
```

Theorem 5.3 (Complexity). *Let q be a query evaluated under skip-till-any-match, P be its pattern of length l, and n be the number of events per window of q. Let $\mathcal{T}_t$ ($\mathcal{T}_e$) be the set of event types for which type-grained (event-grained) trend counts are maintained. Let $t \leq l$ be the number of types in $\mathcal{T}_t$ and $n_e \leq n$ be the number of events of a type in $\mathcal{T}_e$ per window of q. Algorithm 2 has quadratic time complexity: $O(n\ n_e)$ and linear space complexity: $\Theta(t + n_e)$.*

Proof. The time complexity of the static analysis (Lines 1–4) is linear in the length of the pattern l and the number of predicates θ. These values are negligible compared to the number of events per window n for high-rate streams and meaningful queries. During runtime execution (Lines 5–16), for each event, $O(t)$ type-grained and $O(n_e)$ event-grained trend counts are accessed. Since the number of types $t \leq l$ is negligible compared to the number of events per window n, the time complexity is quadratic: $O(n\ n_e)$. The space complexity is linear in the number of counts: $\Theta(t + n_e)$. □

Theorem 5.4 (Time Optimality). *The quadratic time complexity $O(n\ n_e)$ of Algorithm 2 is optimal.*

Proof. By Theorem 4.4, the type-grained trend count computation has optimal time complexity $O(n)$.

By Definition 2.4, any event can be skipped in between adjacent events in a trend under skip-till-any-match. Thus, any previously matched event e_p may be a predecessor event of a new event e in a trend (Definition 3.2). Therefore, each previously matched event e_p must be kept in order to evaluate predicates on adjacent events while deciding whether e_p and a new event e are adjacent. Algorithm 2 maintains event-grained trend counts only for those events that are needed to verify their adjacency to future events. In the worst case, one event requires access to $O(n_e)$ event-grained trend counts to guarantee correctness. In summary, the quadratic time complexity $O(n\ n_e)$ is optimal. □

6 PATTERN-GRAINED AGGREGATOR

To avoid polynomial time overhead of trend construction under the skip-till-next-match and contiguous semantics (Table 4), we now propose to incrementally compute trend aggregation at the coarsest possible pattern granularity. This is possible because an event can have at most one predecessor event under these restrictive semantics.

Theorem 6.1 (Predecessor Event Uniqueness). *Let q be a query evaluated under the skip-till-next-match or contiguous semantics. For any event $e \in I$ that is part of an event trend matched by q, e has the same predecessor event in all trends matched by q (if e has a predecessor).*

Proof. Suppose e has different predecessors e_1 and e_2 in trends tr_1 and tr_2, respectively. By Definition 3.2, $e_1.time < e.time$ and $e_2.time < e.time$. Without loss of generality, assume $e_1.time < e_2.time$. By Definitions 2.8 and 2.6, relevant events must be matched under the skip-till-next-match and contiguous semantics. Thus, the relevant event e_2 cannot be skipped in tr_1 and e_1 cannot be predecessor event of e. □

By Theorem 6.1 and Definition 3.2, event adjacency can be determined based on the last matched event and a new event. Thus, only the last matched event must be stored.

Algorithm 3 Pattern-grained trend count algorithm

Input: Query q with pattern P, event stream I
Output: Count of event trends matched by q in I

```
1:  e_l ← null; e_l.count ← 0; final_count ← 0
2:  for each e ∈ I of type E do
3:      if isMatched(e_l, e) then
4:          if E = start(P) then e.count ← 1
5:          else e.count ← 0
6:          if isAdjacent(e_l, e) then e.count += e_l.count
7:          if E = end(P) then final_count += e.count
8:          e_l ← e
9:      else if q.semantics = CONT then
10:             e_l ← null; e_l.count ← 0
11: return final_count
```

Example 6.2. Under skip-till-next-match, the trend count computation for our running example is shown in Table 8. Each a increments the intermediate count $a.count$ by one since it is of a start type of P. Each b increments the final count since it is of an end type of P. The irrelevant event $c5$ is skipped and eight trends are detected (Figure 2). Only the most recent values of the final count and the count of the last matched event are stored at a time.

Under the contiguous semantics, the trend count is computed analogously. The only difference is that $c5$ cannot be ignored. To invalidate all partial trends that end at the last matched event $a4$, we set the last matched event to null and its intermediate count to 0. Since $b6$ is neither of a start type of P, nor adjacent to the last matched event (null), $b6$ cannot be matched. Two contiguous trends are detected (Figure 2).

Event	a.count	b.count	**Final count**
a1	1		
b2		1	1
a3	2		
a4	3		
c5			
b6		3	4
a7	4		
b8		4	8

Table 8: Pattern-grained trend count

Theorem 6.3 (Pattern-Grained Trend Count). *Let q be a query evaluated under the contiguous or skip-till-next-match semantics and P be its pattern. Let $e_l, e \in I$ be events such that e_l is the predecessor event of e in a trend matched by q. Then, the pattern-grained counts are computed as follows.*

$$e.count = e_l.count.$$
$$final\_count = \sum_{end.type=end(P)} end.count.$$

If $E = start(P)$, $e.count$ is incremented by one.

Theorem 6.3 can be proven similarly to Theorem 4.2.

Pattern-Grained Trend Count Algorithm. Initially, the last matched event e_l is null, $e_l.count$ and the final count are set to 0 (Line 1 in Algorithm 3). An event e is matched by the query q if one of the following conditions holds: (1) If e is of a start type of P, e starts a new trend and $e.count$ is set to one (Lines 2–4). Otherwise, $e.count$ is set to 0 (Line 5). (2) If e_l and e are adjacent, e continues existing partial trends and $e.count$ is increased by $e_l.count$ to accumulate the number of previously started partial trends (Line 6). If e is of an end type of P, the final count is increased by $e.count$ (Line 7). The event e becomes the new last event (Line 8). If e is not matched by the query q that detects contiguous trends, then the last matched event e_l and its count are reset to invalidate partial trends that end at e_l (Lines 9–10). The final count is not reset, however, because it accumulates the number of contiguous trends that were detected before the irrelevant event e arrived. The final count is returned (Line 11).

Theorem 6.4 (Complexity). *Let q be a query evaluated under the contiguous or skip-till-next-match semantics and n be the number of events per window of q. Algorithm 3 has linear time complexity: $O(n)$ and constant space complexity.*

Proof. The time complexity is determined by the number of events per window, i.e., $O(n)$. The space complexity is constant since only two counts and the last matched event are stored. □

Theorem 6.5 (Time Optimality). *The linear time complexity $O(n)$ of Algorithm 3 is optimal.*

Theorem 6.5 can be proven analogously to Theorem 4.4.

7 OTHER QUERY CLAUSES

So far we have focused on Kleene patterns, matching semantics, and predicates on adjacent events. We now generalize Cogra to support other query clauses (Definition 2.9). Extensions of the language are discussed in Appendix C.

Windows partition the unbounded stream into intervals with a finite number of events in each interval. Our approach is applicable to any window type (e.g., sliding, hopping, count-based). Since windows may overlap (aka sliding windows), an event e may fall into $k \geq 1$ windows. This implies that an event e may expire in some windows but remain valid in others. To tackle this contiguous nature of streaming, we maintain aggregates per window [26]. Each aggregates is assigned a window identifier and is computed based on previous aggregates within the same window [39].

Predicates on Single Events are classified into local and equivalence predicates [39, 40]. Local predicates restrict the attribute values of matched events. For example, the predicate (A.type = passive) in query q_4 in Appendix A selects only passive physical activities. Such predicates filter the stream, before our approach applies. Equivalence predicates require that all events in a trend have the same attribute value. For example, query q_2 has a predicate [driver] that requires all events in an Uber pool trip to have the same driver identifier [11, 49, 50]. Such predicates partition the stream into non-overlapping sub-streams by the values of this attribute. Thereafter, our approach applies to each sub-stream.

Event Trend Grouping ensures that all events in a trend carry the same values of grouping attributes [38–40]. Similarly to equivalence predicates, event trend grouping partitions the stream into non-overlapping sub-streams. Thereafter, our approach applies to each sub-stream.

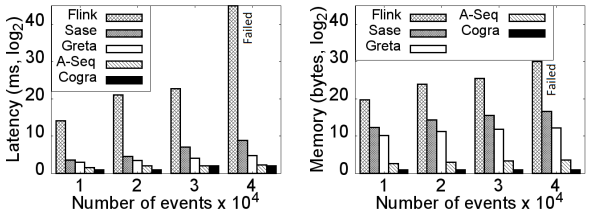
8 PERFORMANCE EVALUATION

8.1 Experimental Setup

Infrastructure. We have implemented our approach in Java with JDK 1.8.0_181 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 results here.

Methodology. We compare our Cogra approach to two two-step approaches (Flink version 1.6.1 [2] and SASE [50]) and two online approaches (A-Seq [40] and GRETA [39]) to cover the spectrum of state-of-the-art event aggregation approaches (Tables 1 and 2). We run Flink on the same hardware as our platform. To achieve a fair comparison, we implemented SASE, A-Seq, and GRETA on top of our platform.

- Flink [2] is a popular open-source streaming platform that supports pattern matching. In contrast to other industrial systems [1, 7], Flink supports Kleene closure and various semantics. We express our queries using Flink operators such as Kleene closure, event sequence, window, grouping, and filtering [3]. Flink implements a two-step approach that constructs all event trends prior to their aggregation.
- SASE [50] supports Kleene closure and all event matching semantics. It implements the two-step approach. Namely, it first stores each event e in a stack and computes the pointers to e's previous events in a trend. For each window, a DFS-based algorithm traverses these pointers to construct all trends. One current trend is saved during this traversal. Once the trend is complete, it is aggregated, and backtracking is applied to construct other trends.
- GRETA [39] captures all matched events and their adjacency relationships as a graph. Based on the graph, it computes event trend aggregation online, that is, it avoids trend construction. However, it supports only skip-till-any-match semantics and maintains aggregates at a fine granularity.
- A-Seq [40] avoids event sequence construction by dynamically maintaining a count for each prefix of a pattern. Since A-Seq does not support Kleene closure, we flatten our



(a) Latency (b) Memory

Figure 5: Skip-till-any-match (Physical activity data)

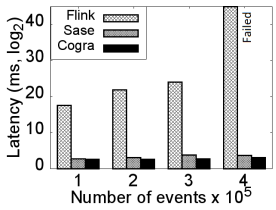


Figure 6: Skip-till-next-match (Physical activity data)

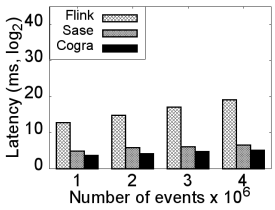


Figure 7: Contiguous semantics (Physical activity data)

queries as follows. For each Kleene pattern P, we first determine the length l of the longest match of P. We then specify a set of fixed-length event sequence queries that cover all possible lengths up to l. A-Seq supports only the skip-till-any-match semantics. It does not support arbitrary predicates on adjacent events beyond equivalence predicates (Section 7).

Data Sets. We compare our Cogra approach to the state-of-the-art techniques using the following data sets.

• New York city taxi and Uber real data set [10] contains 1.3 billion taxi and Uber trips in New York City in 2014–2015. Each event carries a time stamp in seconds, driver and rider identifiers, pick-up and drop-off locations, number of passengers, and price. The size of the data set is 330GB.

• Stock real data set [4] contains transaction records of more than 3K companies for one week. Each event carries a time stamp in minutes, company identifier, sector identifier, transaction identifier, and price. The size of the data is 1.3GB.

• Physical activity real data set [41] contains physical activity reports for 14 people during 1 hour 15 minutes. 18 activities are considered. A report carries a time stamp in seconds, person identifier, activity identifier, and heart rate. The size of the data set is 1.6GB.

Event Trend Aggregation Queries. We evaluate variations of q_2 in Section 1 against the Uber data, variations of q_3 in Section 1 against the stock data, and variations of q_4 in Appendix A against the physical activity data. These queries vary the event matching semantics and the event rates per window in Section 8.2, the number of event trend groups in Appendix D.1, and the predicate selectivity in Appendix D.2. Since A-Seq does not support predicates on adjacent events (Table 2), we exclude such predicates by default.

Metrics. Latency is measured in milliseconds as the average time difference between the result output and the arrival of the latest event that contributed to this result. Throughput corresponds to the average number of events processed by all queries per second. Peak memory is measured in bytes as the memory for storing aggregates and sub-graphs for Cogra, the graph for GRETA, the prefix counters for A-Seq, the events in stacks, the pointers between them, and one current trend for SASE, and all trends for Flink.

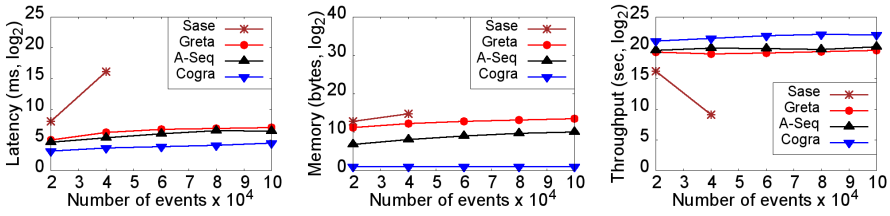
8.2 Event Matching Semantics and Event Rate per Window

In Figures 5–11, we compare the performance of Cogra to the state-of-the-art approaches under diverse event matching semantics, while varying the event rate per window. If an approach does not support a semantics (Table 2), the approach is not shown in the chart for this semantics.

Two-Step Approaches. Flink is not optimized for the type of queries we target by this work. In particular, Flink first constructs all event trends and only then aggregates them. This event trend construction introduces exponential (polynomial) overhead under the skip-till-any-match (skip-till-next-match) semantics (Table 4). Consequently, Flink does not return results within several hours for high-rate event streams under these semantics (Figures 5 and 6). Cogra outperforms Flink by 6 orders of magnitude with respect to latency and 7 orders of magnitude with respect to memory usage for 30k events per window under skip-till-any-match (Figure 5). Similar performance difference can be observed under skip-till-next-match (Figure 6). Flink performs better under the contiguous semantics because the number and length of detected trends is much less than under other semantics. Nevertheless, the latency of Flink is 4 orders of magnitude higher than the latency of Cogra for 4 million events per window (Figure 7). Figures 8–11 do not show measurements for Flink because Flink does not return results for several hours for these high-rate event streams.

SASE also implements a two-step approach that constructs all trends prior to aggregating them. Such an approach suffers from exponential complexity under skip-till-any-match (Table 4). Thus, the latency and memory usage of SASE grow exponentially in the number of events, while its throughput degrades exponentially (Figure 8). SASE fails to return results when the number of events exceeds 40k. For 40k events, Cogra achieves 3 orders of magnitude speed-up and 4 orders of magnitude memory reduction compared to SASE.

Under the skip-till-next-match semantics the time complexity of SASE is polynomial. While for 4 million events SASE returns results with an over 3 hours long delay – which



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

Figure 8: Skip-till-any-match semantics (Stock data)

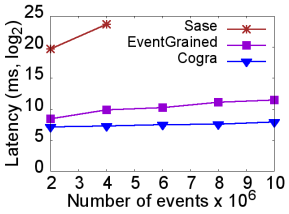
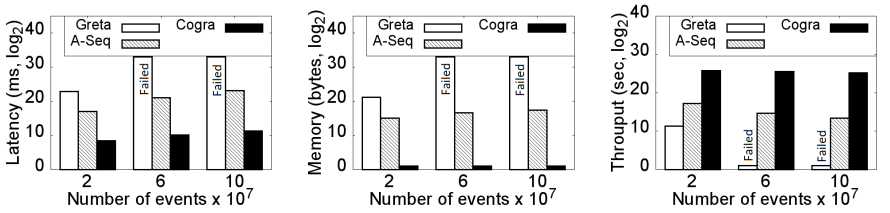


Figure 9: Skip-till-next-match semantics (Uber data)



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

Figure 10: Skip-till-any-match semantics (Stock data)

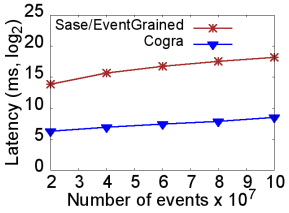


Figure 11: Contiguous semantics (Uber data)

is unacceptable for time-critical applications, it fails to produce results if a window contains over 4 million events (Figure 9). Cogra achieves 4 orders of magnitude speed-up and 5 orders of magnitude memory reduction compared to SASE for 4 million events.

SASE performs well for high-rate streams only under the most restrictive contiguous semantics when the number and length of trends are relatively small (Figure 11). Nevertheless, Cogra achieves 3 orders of magnitude speed up and 6 orders of magnitude memory reduction compared to SASE when the number of events per window reaches 10 million.

Online Approaches perform well for low-rate streams (Figures 5–8), while for high-rate streams the differences between them are revealed (Figure 10). GRETA captures all matched events and their trend relationships as a graph. While the overhead of graph construction is negligible for low-rate streams, it becomes a bottleneck when the stream rate is high. Under skip-till-any-match, GRETA fails to compute results within several hours if the stream rate exceeds 20 million events per window (Figure 10(a)). For 20 million events, GRETA suffers from over an hour long delay which is 4 orders of magnitude higher compared to Cogra.

A-Seq performs best among the state-of-the-art approaches because it neither constructs event trends nor stores events. However, its expressive power is limited (Table 2). In particular, it does not support Kleene closure. Thus, A-Seq must evaluate a workload of event sequence queries to express one Kleene pattern. The number of queries grows linearly in the number of events. Thus, the latency of A-Seq is 3 orders of magnitude higher compared to Cogra when the number of events per window reaches 100 million (Figure 10(a)). Each event sequence query maintains a fixed number of aggregates [40]. Thus, the memory usage of A-Seq grows linearly with the number of queries (i.e., with the number of events). A-Seq requires 4 orders of magnitude more memory than Cogra for 100 million events (Figure 10(b)).

In summary, our Cogra approach performs well across all semantics, data sets, and stream rates because it guarantees optimal time complexity. Its latency grows linearly in the number of events. For 100 million events per window, the latency of Cogra stays within 3 seconds (Figure 10(a)), while its throughput is over 39 million events per second (Figure 10(c)). Since Cogra maintains a fixed number of aggregates per Kleene pattern, its memory usage is constant (Figures 8(b) and 10(b)). Thus, we conclude that Cogra achieves real-time in-memory event trend aggregation.

Trend Aggregation at Different Granularities under the Same Event Matching Semantics. GRETA corresponds to the event-grained aggregation, while Cogra implements the type-grained aggregation under the skip-till-any-match semantics in Figures 5, 8, and 10. The pattern-grained aggregation would not be correct under skip-till-any-match since the pattern constraint may be violated (Definition 2.4).

Unfortunately no state-of-the-art approach implements online trend aggregation under the skip-till-next-match nor the contiguous semantics (Table 2). Thus, Figure 9 shows how our proposed pattern-grained aggregation Cogra compares

to the suboptimal event-grained aggregation under the skip-till-next-match semantics. The event-grained aggregation does not construct all trends. Consequently, its latency is 4 orders of magnitude lower than the latency of SASE for 4 million events. However, its latency is 12-fold higher than the latency of Cogra due to the overhead of storing all matched events and propagating event-grained aggregates.

Under the contiguous semantics in Figure 11, the event-grained aggregation and SASE have the same latency. Indeed, since the number and length of contiguous trends is small, the overhead of storing all matched events and propagating event-grained aggregates by the event-grained aggregation equals to the overhead of constructing all trends by SASE.

The type-grained aggregation performs similarly to the pattern-grained aggregation Cogra because the number of event types is small in meaningful queries (q_2 in Section 1).

9 RELATED WORK

Complex Event Processing approaches including SASE [11, 49, 50], Flink [2], Cayuga [15], ZStream [33], AFA [13], E-Cube [29], and CET [38] solve orthogonal problems. Many of them deploy a Finite State Automaton-based query execution paradigm [11, 13–15, 36, 49, 50]. AFA supports dynamic pattern detection over disordered streams. ZStream translates an event query into an operator tree optimized using rewrite rules. Similarly, Kolchinsky et al. apply traditional join-query optimization techniques to evaluate event sequence and Kleene closure queries [22] and adapt an optimization plan to changing data characteristics [21]. E-Cube employs hierarchical event stacks to share events across different event queries. CET solves the trade-off between the CPU and memory during event trend detection. Some approaches focus on XPath-based query processing over XML streams [14, 36]. Others implement hardware-based CEP on FPGAs at gigabit wire speed [48]. The expressive power of these approaches is limited since they do not support Kleene closure [29], nor aggregation [13, 21, 22, 36, 38, 48], nor various event matching semantics [13–15, 21, 29, 33, 36, 38, 48]. In contrast, SASE and Flink support all these language constructs but they do not design any optimization techniques for event trend aggregation. Thus, they require trend construction with exponential time complexity (Table 4) and fail to respond within a few seconds (Section 8).

In contrast, A-Seq [40] proposes online aggregation of fixed-length event sequences under skip-till-any-match. However, it supports neither Kleene closure, nor various semantics, nor expressive predicates. Thus, A-Seq does not tackle the challenges of this work. GRETA [39] defines online event trend aggregation under skip-till-any-match. GRETA does not support other semantics. It captures all event trends and aggregates as a graph. It avoids event trend construction and thus reduces the time complexity from exponential to quadratic in the number of events in the worst case. Since GRETA maintains aggregates at the event granularity, it does not achieve optimal time complexity for several query classes and thus suffers from long delays (Section 8).

Traditional Data Streaming approaches [12, 16, 23, 25, 26, 44, 46, 47, 51, 52] support aggregation computation over data streams. Some incrementally aggregate raw input events for single-stream queries [25, 26]. Others share aggregation results among overlapping sliding windows [12, 25, 46] or multiple queries [23, 51, 52] or both [44]. However, these approaches are restricted to Select-Project-Join queries with window semantics. That is, their execution paradigm is set-based. They support neither various event matching semantics nor CEP-specific operators such as event sequence and Kleene closure that treat the order of events as a first-class citizen. These approaches require the construction of join results prior to their aggregation, i.e., they define incremental aggregation of single raw events but implement a two-step approach for join results.

Industrial streaming systems such as Esper [1], and Oracle Stream Analytics [7] support fixed-length event sequences. They support neither Kleene closure nor different semantics. While Kleene closure computation can be simulated by a set of event sequence queries covering all possible lengths of event trends, this approach is possible only if the maximal length of a trend is known apriori. This is rarely the case in practice. Furthermore, this approach is highly inefficient for two reasons. One, it must execute a set of event sequence queries for each Kleene query. This increased workload degrades system performance. Two, since this approach requires trend construction prior to their aggregation, it has exponential time complexity in the worst case (Table 4).

Additional related work is discussed in Appendix E.

10 CONCLUSIONS

Our Cogra approach is the first to aggregate Kleene pattern matches under various event matching semantics with optimal time complexity. To this end, Cogra incrementally maintains trend aggregates at the coarsest possible granularity among several alternative design strategies. Thus, it achieves up to six orders of magnitude speed-up and up to seven orders of magnitude memory reduction compared to state-of-the-art solutions.

ACKNOWLEDGEMENTS

This work was supported by NSF grants IIS-1815866, CRI-1305258, and IIS-1018443.

REFERENCES

[1] 2019. Esper. http://www.espertech.com/.

[2] 2019. Flink. https://flink.apache.org/.

[3] 2019. Flink queries. https://www.dropbox.com/sh/1rafk6ce75pdrr4/AABv4WBPxuBolonWJjZ4wJw_a?dl=0.

[4] 2019. Historical Stock Data. http://www.eoddata.com.

[5] 2019. KardiaMobile. https://www.alivetec.com/pages/alivecor-heart-monitor.

[6] 2019. MetaTrader4. https://www.metatrader4.com/.

[7] 2019. Oracle Stream Analytics. http://www.oracle.com/technetwork/middleware/complex-event-processing/documentation/index.html.

[8] 2019. Scalable Real-time Complex Event Processing at Uber. https://eng.uber.com/athenax/.

[9] 2019. Sudden Cardiac Death. https://my.clevelandclinic.org/health/articles/sudden-cardiac-death.

[10] 2019. Unified New York City Taxi and Uber data. https://github.com/toddwschneider/nyc-taxi-data.

[11] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. 2008. Efficient pattern matching over event streams. In *SIGMOD*. 147–160.

[12] Arvind Arasu and Jennifer Widom. 2004. Resource Sharing in Continuous Sliding-window Aggregates. In *VLDB*. 336–347.

[13] Badrish Chandramouli, Jonathan Goldstein, and David Maier. 2010. High-Performance Dynamic Pattern Matching over Disordered Streams. *PVLDB* 3, 1 (2010), 220–231.

[14] Denis Debarbieux, Olivier Gauwin, Joachim Niehren, Tom Sebastian, and Mohamed Zergaoui. 2015. Early Nested Word Automata for XPath Query Answering on XML Streams. *Theor. Comput. Sci.* 578, C (2015), 100–125.

[15] Alan Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker White. 2007. Cayuga: A general purpose event monitoring system. In *CIDR*. 412–422.

[16] Thanaa Ghanem, Moustafa Hammad, Mohamed Mokbel, Walid Aref, and Ahmed Elmagarmid. 2007. Incremental Evaluation of Sliding-Window Queries over Data Streams. *IEEE Trans. on Knowl. and Data Eng.* 19, 1 (2007), 57–72.

[17] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Min. Knowl. Discov.* (1997), 29–53.

[18] Jiawei Han, Jian Pei, Behzad Mortazavi-asl, Qiming Chen, Umeshwar Dayal, and Meichun Hsu. 2000. FreeSpan: frequent pattern-projected sequential pattern mining. In *KDD*. 355–359.

[19] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. 2004. Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach. In *KDD*. 53–87.

[20] Arshad Khan. 2002. *Stock Market Tips and Guidelines*. Writers Club Press.

[21] Ilya Kolchinsky and Assaf Schuster. 2018. Efficient Adaptive Detection of Complex Event Patterns. *Proc. VLDB Endow.* 11, 11 (July 2018), 1346–1359.

[22] Ilya Kolchinsky and Assaf Schuster. 2018. Join Query Optimization Techniques for Complex Event Processing Applications. *Proc. VLDB Endow.* 11, 11 (July 2018), 1332–1345.

[23] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. 2006. On-the-fly sharing for streamed aggregation. In *SIGMOD*. 623–634.

[24] Alberto Lerner and Dennis Shasha. 2003. AQuery: Query language for ordered data, optimization techniques, and experiments. In *VLDB*. 345–356.

[25] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. No pane, no gain: Efficient evaluation of sliding window aggregates over data streams. In *SIGMOD*. 39–44.

[26] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*. 311–322.

[27] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. 2008. Out-of-Order Processing: a New Architecture for High-Performance Stream Systems. In *VLDB*. 274–288.

[28] Mo Liu, Ming Li, Denis Golovnya, Elke A. Rundensteiner, and Kajal T. Claypool. 2009. Sequence Pattern Query Processing over Out-of-Order Event Streams. In *ICDE*. 784–795.

[29] Mo Liu, Elke Rundensteiner, Kara Greenfield, Chetan Gupta, Song Wang, Ismail Ari, and Abhay Mehta. 2011. E-Cube: Multi-dimensional event sequence analysis using hierarchical pattern query sharing. In *SIGMOD*. 889–900.

[30] Eric Lo, Ben Kao, Wai-Shing Ho, Sau Dan Lee, Chun Kit Chui, and David W. Cheung. 2008. OLAP on sequence data. In *SIGMOD*. 649–660.

[31] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. 1997. Discovery of Frequent Episodes in Event Sequences. *Data Min. Knowl. Discov.* 1, 3 (Jan. 1997), 259–289.

[32] Matt Massie, Bernard Li, Brad Nicholes, Vladimir Vuksan, Robert Alexander, Jeff Buchbinder, Frederiko Costa, Alex Dean, Dave Josephsen, Peter Phaal, and Daniel Pocock. 2012. *Monitoring with Ganglia* (1st ed.). O'Reilly Media, Inc.

[33] Yuan Mei and Samuel Madden. 2009. ZStream: A Cost-based Query Processor for Adaptively Detecting Composite Events. In *SIGMOD*. 193–206.

[34] Luiz F. Mendes, Bolin Ding, and Jiawei Han. 2008. Stream Sequential Pattern Mining with Precise Error Bounds. In *ICDM*. 941–946.

[35] Iakovos Motakis and Carlo Zaniolo. 1997. Temporal Aggregation in Active Database Rules. In *SIGMOD*. 440–451.

[36] Barzan Mozafari, Kai Zeng, and Carlo Zaniolo. 2012. High-performance complex event processing over XML streams. In *SIGMOD*. 253–264.

[37] Jian Pei, Jiawei Han, Behzad Mortazavi-asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Meichun Hsu. 2001. PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth. In *ICDE*. 215–224.

[38] Olga Poppe, Chuan Lei, Salah Ahmed, and Elke Rundensteiner. 2017. Complete Event Trend Detection in High-rate Streams. In *SIGMOD*. 109–124.

[39] Olga Poppe, Cuan Lei, Elke A. Rundensteiner, and David Maier. 2017. GRETA: Graph-based Real-time Event Trend Aggregation. In *VLDB*. 80–92.

[40] Yingmei Qi, Lei Cao, Medhabi Ray, and Elke A. Rundensteiner. 2014. Complex Event Analytics: Online Aggregation of Stream Sequence Patterns. In *SIGMOD*. 229–240.

[41] Attila Reiss and Didier Stricker. 2012. Creating and Benchmarking a New Dataset for Physical Activity Monitoring. In *PETRA*. 40:1–40:8.

[42] Reza Sadri, Carlo Zaniolo, Amir Zarkesh, and Jafar Abidi. 2004. Expressing and Optimizing Sequence Queries in Database Systems. In *ACM Trans. on Database Systems*. 282–318.

[43] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. 1996. SEQ: Design and Implementation of a Sequence Database System. In *VLDB*. 99–110.

[44] Anatoli U. Shein, Panos K. Chrysanthis, and Alexandros Labrinidis. 2018. SlickDeque: High Throughput and Low Latency Incremental Sliding-Window Aggregation. In *EDBT*. 397–408.

[45] Utkarsh Srivastava and Jennifer Widom. 2004. Flexible Time Management in Data Stream Systems. In *PODS*. 263–274.

[46] Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. 2017. Low-Latency Sliding-Window Aggregation in Worst-Case Constant Time. In *DEBS*. 66–77.

[47] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. 2015. General Incremental Sliding-window Aggregation. In *VLDB*. 702–713.
[48] Louis Woods, Jens Teubner, and Gustavo Alonso. 2010. Complex Event Detection at Wire Speed with FPGAs. *VLDB Endow.* 3, 1-2 (2010), 660–669.
[49] Eugene Wu, Yanlei Diao, and Shariq Rizvi. 2006. High-performance Complex Event Processing over streams. In *SIGMOD*. 407–418.
[50] Haopeng Zhang, Yanlei Diao, and Neil Immerman. 2014. On Complexity and Optimization of Expensive Queries in CEP. In *SIGMOD*. 217–228.
[51] Rui Zhang, Nick Koudas, Beng Chin Ooi, and Divesh Srivastava. 2005. Multiple Aggregations over Data Streams. In *SIGMOD*. 299–310.
[52] Rui Zhang, Nick Koudas, Beng Chin Ooi, Divesh Srivastava, and Pu Zhou. 2010. Streaming Multiple Aggregations Using Phantoms. In *VLDB*. 557–583.

A ADDITIONAL USE CASES

q_4 : RETURN patient, COUNT(∗)
PATTERN SEQ(Activity S, Activity A+, Activity F)
SEMANTICS contiguous
WHERE [patient] AND S.rate < 60 AND F.rate > 100 AND
A.type = passive GROUP-BY patient
WITHIN 10 minutes SLIDE 30 seconds

Health Care Analytics devices, such as KardiaMobile [5], allow users to record, review, and analyse electrocardiograms in real time. It helps to detect, monitor, and cure serious heart diseases such as cardiac arrhythmia, i.e., too fast, too slow, or irregular heartbeat. Cardiac arrhythmia can lead to life-threatening complications causing 325K sudden cardiac deaths in United States per year [9]. Thus, the prompt detection of abnormal heartbeat is critical to enable immediate lifesaving measures. Query q_4 consumes a stream of activities of intensive care patients. Each event carries a time stamp in seconds, a patient identifier, an activity type, and a heart rate. The query computes the number of heart rate changes from too slow (below 60) to too fast (above 100) per patient despite passive physical activities during a time window of 10 minutes that slides every 30 seconds. All events that belong to one trend must carry the same patient identifier as required by the predicate [patient]. No events may be skipped in between matched events per patient, as expressed by the contiguous semantics.

B ADDITIONAL PROOFS

Theorem 4.2. *(Type-Grained Trend Count). Let q be a query that is evaluated under skip-till-any-match and has no predicates on adjacent events, P be its pattern, E be an event type in P, and $e \in I$ be an event of type E. Then the event-grained count $e.count$ associated with the event e corresponds to the number of (partial) trends that end at e:*

$$e.count = \sum_{E' \in P.predTypes(E)} E'.count.$$

If $E = start(P)$, $e.count$ is incremented by one.

The type-grained trend count $E.count$ associated with the type E captures the number of (partial) trends that end at an event of type E:

$$E.count = \sum_{e.type=E} e.count.$$

The final count corresponds to the number of finished trends matched by q:

$$final\_count = end(P).count.$$

Proof by induction. Let n be the number of input events.

Induction Basis. $n = 1$. If only one event e of type E is matched, then there are no trends to extend and e starts a new trend. Thus, $e.count = 1$ and $E.count = 1$. If E is the end type of P, $final\_count = 1$. Otherwise, $final\_count = 0$.

Induction Assumption. This statement holds for n events.

Induction Step. Assume $n + 1$ events arrive. According to the induction assumption, for a type E' in P, $E'.count$ corresponds to the number of (partial) trends that end at an event of type E' after n events have been processed. Let e be a new event of type E. By Definition 2.4 without predicates on adjacent events, e is adjacent to all previously matched events of type $E' \in P.predTypes(E)$, i.e., e continues all these trends. To accumulate the number of trends extended by e, $e.count$ is computed as the sum of counts of all predecessor types of E, i.e., $E'.count$. In addition, if E is a start type of P, e begins a new trend and $e.count$ is incremented by 1. Since events arrive in order by time stamps (Section 2), no event after e can change $e.count$. $E.count$ is increased by $e.count$ to accumulate the number of trends extended by e. Lastly, the query q counts the number of finished trends only. By Definition 3.1 and Section 3.1, only events of end type of P may finish trends. Thus, the count of the end type of P captures the number of finished trends. □

C OTHER LANGUAGE FEATURES

We now discuss how to relax our simplifying assumptions.

Aggregation Functions. While Sections 4–6 focused on event trend count computation (i.e., COUNT(*)), Table 9 defines coarse-grained computation of COUNT(E), MIN($E.attr$), and SUM($E.attr$) as per Section 2.3. In contrast to COUNT(*), only matched events e of type E update these aggregates. All other matched events x of type $X \neq E$ propagate the aggregates from previous to more recent events in a trend matched by a pattern P (or event types in P). MAX(E.attr) is maintained analogously to MIN($E.attr$). AVG($E.attr$) is computed based on SUM($E.attr$) and COUNT(E).

Negated Sub-Patterns. We split the pattern into positive and negative sub-patterns and maintain aggregates for each sub-pattern separately [39]. If aggregates are maintained per matched event, whenever a negative sub-pattern N finds a

Type-grained aggregates	Mixed-grained aggregates	Pattern-grained aggregates
$\boldsymbol{e.count_E} = e.count + \sum_{E' \in \mathcal{T}} E'.count_E$	$\boldsymbol{e.count_E} = e.count + \sum_{E' \in \mathcal{T}_t} E'.count_E + \sum_{e".type \in \mathcal{T}_e,\ (e",e) \text{ satisfy } \theta} e".count_E$	$\boldsymbol{e_l.count_E}$ += $e.count$
$\boldsymbol{x.count_E} = \sum_{X' \in \mathcal{Y}} X'.count_E$	$\boldsymbol{x.count_E} = \sum_{X' \in \mathcal{Y}_t} X'.count_E + \sum_{x".type \in \mathcal{Y}_e,\ (x",x) \text{ satisfy } \theta} x".count_E$	
$\boldsymbol{T.count_E} = \sum_{t.type=T} t.count_E$		
$\boldsymbol{COUNT(E)} = end(P).count_E$		$\boldsymbol{COUNT(E)} = \sum_{e_l.type=end(P)} e_l.count_E$
$\boldsymbol{e.min} = \min_{E' \in \mathcal{T}}(e.attr, E'.min)$	$\boldsymbol{e.min} = \min_{E' \in \mathcal{T}_t,\ e".type \in \mathcal{T}_e,\ (e",e) \text{ satisfy } \theta}(e.attr, E'.min, e".min)$	$\boldsymbol{e_l.min} = \min(e.attr, e_l.min)$
$\boldsymbol{x.min} = \min_{X' \in \mathcal{Y}}(X'.min)$	$\boldsymbol{x.min} = \min_{X' \in \mathcal{Y}_t,\ x".type \in \mathcal{Y}_e,\ (x",x) \text{ satisfy } \theta}(X'.min, x".min)$	
$\boldsymbol{T.min} = \min_{t.type=T}(t.min)$		
$\boldsymbol{MIN(E.attr)} = end(P).min$		$\boldsymbol{MIN(E.attr)} = \min_{e_l.type=end(P)}(e_l.min)$
$\boldsymbol{e.sum} = e.attr * e.count + \sum_{E' \in \mathcal{T}} E'.sum$	$\boldsymbol{e.sum} = e.attr * e.count + \sum_{E' \in \mathcal{T}_t} E'.sum + \sum_{e".type \in \mathcal{T}_e,\ (e",e) \text{ satisfy } \theta} e".sum$	$\boldsymbol{e_l.sum}$ += $e.attr * e.count$
$\boldsymbol{x.sum} = \sum_{X' \in \mathcal{Y}} X'.sum$	$\boldsymbol{x.sum} = \sum_{X' \in \mathcal{Y}_t} X'.sum + \sum_{x".type \in \mathcal{Y}_e,\ (x",x) \text{ satisfy } \theta} x".sum$	
$\boldsymbol{T.sum} = \sum_{t.type=T} t.sum$		
$\boldsymbol{SUM(E.attr)} = end(P).sum$		$\boldsymbol{SUM(E.attr)} = \sum_{e_l.type=end(P)} e_l.sum$

Table 9: Coarse-grained event trend aggregation ($e, x, e", x", t \in I$ are matched events, $e.type = E$, $x.type = X \neq E$, T is any event type in P, $P.predTypes(E) = \mathcal{T} = \mathcal{T}_t \cup \mathcal{T}_e$, $P.predTypes(X) = \mathcal{Y} = \mathcal{Y}_t \cup \mathcal{Y}_e$ where type-grained aggregates are maintained for types $\mathcal{T}_t$ and $\mathcal{Y}_t$ and event-grained aggregates are maintained for events of types $\mathcal{T}_e$ and $\mathcal{Y}_e$)

match (i.e., its COUNT(*) = 1), all previously matched events of predecessor types $\mathcal{T}_p$ of N are marked as incompatible with all future events of following types $\mathcal{T}_f$ of N. If aggregates are maintained per type, the aggregates of all predecessor types $\mathcal{T}_p$ are marked as invalid to contribute to aggregates of the following types $\mathcal{T}_f$. Lastly, if aggregates are maintained per pattern, the last matched event e_l of the previous sub-pattern of N is set to null.

Disjunction and Conjunction can be supported by Cogra without changing its complexity because the aggregates for a disjunctive or a conjunctive pattern P can be computed based on the aggregates for the sub-patterns of P [39].

Kleene Star and Optional Sub-Patterns can also be supported without changing the complexity since they are syntactic sugar operators. Indeed, $\text{SEQ}(P_1*, P_2) = \text{SEQ}(P_1+, P_2) \vee P_2$ and $\text{SEQ}(P_1?, P_2) = \text{SEQ}(P_1, P_2) \vee P_2$.

Multiple Event Type Occurrences in a Pattern. If a type appears several time in a pattern, our approach applies with the following modifications. (1) We assign an identifier to each type. For example, $\text{SEQ}(A+, B, A)$ is translated into $\text{SEQ}(A1+, B, A2)$. Then, each state in an FSA representation of the pattern has a unique label (Section 3.1). (2) An event e may be inserted into several sub-graphs (or update several type-grained aggregates) under the skip-till-any-match semantics. However, e may not be its own predecessor event since an event may occur in a trend at most once [39].

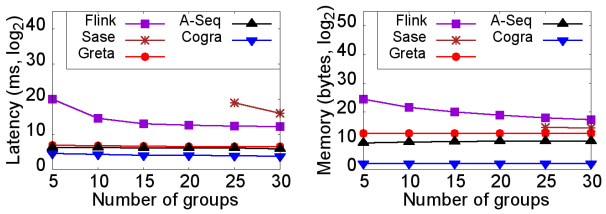
Predicates on Minimal Trend Length exclude too short and thus unreliable trends. One way of modeling such constraints is to unroll a pattern to its minimal length. For example, if we are interested in trends matched by the pattern $A+$ and with $length \geq 3$, then we unroll the pattern $A+$ to length 3 as follows: $\text{SEQ}(A, A, A+)$. Our approach applies thereafter.

D ADDITIONAL EXPERIMENTS

In all experiments below, we evaluate our queries under skip-till-any-match since all state-of-the-art approaches support this semantics (Table 2). To ensure that the two-step approaches return results in most cases, we run these experiments against a low-rate stream of 50k events per window.

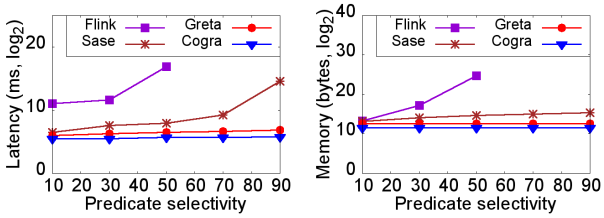
D.1 Event Trend Grouping

As described in Section 7, event trends are constructed per group. Since the event rate is constant 50k events per window, each group has a lot of events when the number of groups is small. Thus, the number and length of trends increase with a decreasing number of groups. As Figure 12 illustrates, the latency and memory usage of all approaches increase with a decreasing number of groups.



(a) Latency (b) Memory

Figure 12: Event trend grouping (Uber data)



(a) Latency (b) Memory

Figure 13: Predicate selectivity (Stock data)

Two-Step Approaches. Since Flink is optimized for parallel execution, it returns results in all cases. Due to the trend construction in Flink, Cogra wins 4 orders of magnitude with respect to latency and 6 orders of magnitude regarding memory compared to Flink for 5 groups in Figure 12.

SASE constructs all trends but stores only one current trend at a time. Thus, its latency reduces exponentially, while its memory consumption decreases linearly with the growing number of trend groups in Figure 12. SASE fails to respond within hours for fewer than 25 groups. For 25 groups, Cogra achieves 4 orders of magnitude speed-up and 3 orders of magnitude memory reduction compared to SASE.

Online Approaches perform well independently from the number of groups because trends are not constructed. Since A-Seq evaluates a workload of event sequence queries for each Kleene query, its latency is 5–fold higher than the latency of Cogra for 5 groups. A-Seq maintains aggregates per group. Thus, its memory costs increase linearly in the number of groups. Cogra wins 2 orders of magnitude regarding memory usage compared to A-Seq for 30 groups.

Due to the GRETA graph construction overhead and edge traversal, the latency of GRETA is 7–fold higher than the latency of Cogra for 5 groups. The memory usage of GRETA remains stable when varying the number of groups because the same number of events is stored in the GRETA graphs. Edges are traversed exactly once and thus are not stored. The memory costs of GRETA are 3 orders of magnitude higher compared to Cogra in all cases in Figure 12(b).

D.2 Predicate Selectivity

Two-Step Approaches. When the predicate selectivity increases, more and longer trends are constructed and stored by Flink. Thus, its latency and memory usage grow exponentially (Table 4, Figure 13) until it fails to return results within several hours when the predicate selectivity exceeds 50%. In this case with predicate selectivity 50%, Cogra achieves 3 orders of magnitude speed-up and 3 orders of magnitude memory reduction compared to Flink.

Since SASE constructs all trends, its latency grows exponentially. It is 2 orders of magnitude higher than the latency of Cogra for 90% predicate selectivity. The number of stored pointers between events increases linearly with growing predicate selectivity. SASE uses 13–fold more memory than Cogra for 90% selectivity.

Online Approaches perform well for low-rate streams of 50k events per window. When the predicate selectivity increases, the number of edges in the GRETA graph grows. Since each edge is traversed exactly once, latency of GRETA grows linearly in the number of edges. Memory consumption remains stable because edges are not stored.

The number of aggregates maintained by Cogra stays the same with the growing predicate selectivity. Since Cogra maintains aggregates per event type (not per event), it achieves double speed-up and memory reduction compared to GRETA when the predicate selectivity reaches 90%.

Since A-Seq does not support predicates on adjacent events, it is excluded from this experiment.

E ADDITIONAL RELATED WORK

Static Sequence Databases extend traditional SQL queries by order-aware join operations and support aggregation of their results [24, 30]. However, they do not support Kleene closure. Instead, single data items are aggregated [24, 35, 42, 43]. They also do not support various matching semantics. Lastly, these approaches assume that the data is statically stored and indexed prior to processing. Hence, they do not tackle challenges that arise due to dynamically streaming data such as event expiration and real-time processing.

Sequential Pattern Mining aims at detecting top-k item sequences that frequently occur in a data set. The sequence patterns of interest are not known prior to processing. These approaches employ a Prefix tree and the Apriori algorithm [18, 19, 31, 34, 37]. This problem is distinct from our problem in which the event patterns of interest are specified by the user prior to processing. Frequent pattern mining approaches do not tackle real-time aggregation of event trends detected by Kleene patterns under various semantics. However, our approach can be used to count the number of sequences and then select the top-k frequent among them.