To Share, or not to Share Online Event Trend Aggregation Over Bursty Event Streams

Olga Poppe[1], Chuan Lei[2], Lei Ma[3], Allison Rozet[4], and Elke A. Rundensteiner[3]
[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]MathWorks, 1 Apple Hill Dr, Natick, MA 01760
olpoppe@microsoft.com,chuan.lei@ibm.com,lma5@wpi.edu,arozet@mathworks.com,rundenst@wpi.edu

ABSTRACT

Complex event processing (CEP) systems continuously evaluate large workloads of pattern queries under tight time constraints. Event trend aggregation queries with Kleene patterns are commonly used to retrieve summarized insights about the recent trends in event streams. State-of-art methods are limited either due to repetitive computations or unnecessary trend construction. Existing shared approaches are guided by statically selected and hence rigid sharing plans that are often sub-optimal under stream fluctuations. In this work, we propose a novel framework Hamlet that is the first to overcome these limitations. Hamlet introduces two key innovations. First, Hamlet adaptively decides at run time whether to share or not to share computations depending on the current stream properties to harvest the maximum sharing benefit. Second, Hamlet is equipped with a highly efficient shared trend aggregation strategy that avoids trend construction. Our experimental study on both real and synthetic data sets demonstrates that Hamlet consistently reduces query latency by up to five orders of magnitude compared to state-of-the-art approaches.

ACM Reference Format:
O.Poppe et al. 2021. To Share, or not to Share Online Event Trend Aggregation Over Bursty Event Streams. In *Xi'an '21: ACM SIGMOD/PODS International Conference on Management of Data, June 20–25, 2021, Xi'an, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Sensor networks, web applications, and smart devices produce high velocity event streams. Industries use Complex Event Processing (CEP) technologies to extract insights from these streams using Kleene queries [9, 15, 46], i.e., queries with Kleene plus "+" operator that matches event sequences of any length, a.k.a. *event trends* [32]. Since these trends can be arbitrarily long and complex and there also tends to be a large number of them, they are typically aggregated to derive summarized insights [37]. CEP systems must thus process large workloads of these event trend aggregation queries over high-velocity streams in near real-time.

Xi'an '21, June 20–25, 2021, Xi'an, China

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

Example 1.1. Complex event trend aggregation queries are used in Uber and DoorDash for price computation, forecasting, scheduling, and routing [30]. With hundreds of users per district, thousands of transactions, and millions of districts nationwide, real-time event analytics has become a challenging task.

In Figure 1, the query workload computes various trip statistics such as the number, total duration, and average speed of trips per district. Each event in the stream is of a particular event type, e.g., *Request*, *Pickup*, *Dropoff*. Each event is associated with attributes such as a time stamp, district, speed, driver, and rider identifiers.

Query q_1 focuses on trips in which the driver drove to a pickup location but did not pickup a rider within 30 minutes since the request. Each trip matched by q_1 corresponds to a sequence of one ride *Request* event, followed by one or more *Travel* events (expressed by the Kleene plus operator "+"), and not followed by a *Pickup* event. All events in a trip must have the same driver and rider identifiers as required by the predicate [driver, rider]. Query q_2 targets *Pool* riders who were dropped off at their destination. Query q_3 tracks riders who cancel their accepted requests while the drivers were stuck in slow-moving traffic. All three queries contain the expensive Kleene sub-pattern $T+$ that matches arbitrarily long event trends. Thus, one may conclude that sharing $T+$ always leads to computational savings. However, a closer look reveals that the actual sharing benefit depends on the current stream characteristics. Indeed, trips are affected by many factors, from time and location to specific incidents, as the event stream fluctuates.

```
q1: RETURN   T.district, COUNT(*), SUM(T.duration)
    PATTERN  SEQ(Request R, Travel T+, NOT Pickup P)
    WHERE    [driver, rider] GROUP-BY T.district
    WITHIN   30 min SLIDE 1 min
q2: RETURN   T.district, COUNT(*), AVG(T.speed)
    PATTERN  SEQ(Request R, Travel T+, Dropoff D)
    WHERE    [driver, rider] AND R.type=Pool GROUP-BY T.district
    WITHIN   30 min SLIDE 5 min
q3: RETURN   T.district, COUNT(*), SUM(T.duration)
    PATTERN  SEQ(Request R, Travel T+, Cancel C)
    WHERE    [driver, rider] AND T.speed<10 GROUP-BY T.district
    WITHIN   20 min SLIDE 1 min
```

Figure 1: Event trend aggregation queries

Challenges. To enable shared execution of trend aggregation queries, we must tackle the following open challenges.

Exponential complexity versus real-time response. Construction of event trends matched by a Kleene query has exponential time complexity in the number of matched events [32, 46]. To achieve real-time responsiveness, shared execution of trend aggregation

queries should thus adopt online strategies that compute trend aggregates on-the-fly while avoiding this expensive trend construction [34, 35]. However, shared execution applied to such online trend aggregation incurs additional challenges not encountered by the shared construction of traditional queries [22]. In particular, we must avoid constructing these trends, while capturing critical connections among shared sub-trends compactly to validate predicates of each query. For example, query q_1 in Figure 1 may match all events of type *Travel*, while queries q_2 and q_3 may only match some of them due to their predicates. Consequently, different trends will be matched by these queries. On first sight it appears that result validation requires the construction of all trends per query, which would defeat the goal of online aggregation. To address this dilemma, we must develop a correct yet efficient shared online trend aggregation strategy.

Benefit versus overhead of sharing. One may assume that the more sub-patterns are shared, the greater the performance improvement will be. However, this assumption does not always hold due to the overhead caused by maintaining intermediate aggregates of sub-patterns to ensure correctness of results. The computational overhead incurred by shared query execution does not always justify the savings achievable compared to baseline non-shared execution. For example, sharing query q_1 with the other two queries in Figure 1 will not be beneficial if there are only few *Pool* requests and the travel speed is above 10 mph. Hence, we need to devise a lightweight benefit model that accurately estimates the benefit of shared execution of multiple trend aggregation at runtime.

Bursty event streams versus light-weight sharing decisions. The actual sharing benefit can vary over time due to the nature of bursty event streams. Even with an efficient shared execution strategy and an accurate sharing benefit model, a static sharing solution may not always lead to computational savings. Worse yet, in some cases, a static sharing decision may do more harm than good. Due to different predicates and windows of queries in Figure 1, one may decide at compile time that these queries should not be shared. However, a large burst of *Pool* requests may arrive and the traffic may be moving slowly (i.e., speed below 10 mph) in rush hour, making sharing of these queries beneficial. For this, a dynamic sharing optimizer, capable of adapting to changing arrival rates, data distribution, and other cost factors, must be designed. Its runtime sharing decisions must be light-weight to ensure real-time responsiveness.

Approach	Kleene closure	Online aggregation	Sharing decisions
MCEP [22]	✓	–	static
Sharon [36]	–	✓	static
Greta [34]	✓	✓	not shared
Hamlet (ours)	✓	✓	dynamic

Table 1: Approaches to event trend aggregation

State-of-the-Art Approaches. While there are approaches to shared execution of multiple Kleene queries [19, 22], they first construct all trends and then aggregate them. Even if trend construction is shared, its exponential complexity is not avoided [32, 46]. Thus, even the most recent approach, MCEP [22] is 76–fold slower than Hamlet as the number of events scales to 10K events per window (Figure 8(a)). Recent work on event trend processing [34–36] addresses this performance bottleneck by pushing the aggregation computation into the pattern matching process. Such online methods manage to skip the trend construction step and thus reduce time complexity of trend aggregation from exponential to quadratic in the number of matched events. Among these online approaches, Greta [34] is the only approach that supports Kleene closure. Unfortunately, Greta neglects sharing opportunities in the workload and instead processes each query independently from others. On the other hand, while Sharon [36] considers sharing among queries, it does not support Kleene closure. Thus, it is restricted to fixed-length event sequences. Further, its shared execution strategy is static and thus misses runtime sharing opportunities. Our experiments confirm that these existing approaches fail to cope with high velocity streams with 100K events per window (Figures 10(a) and 10(b)). Table 1 summarizes these approaches.

Proposed Solution. Our Hamlet approach supports online aggregation over Kleene closure while dynamically deciding which subset of sub-patterns should be shared by which trend aggregation queries and for how long depending on the current characteristics of the event stream. The Hamlet optimizer leverages these stream characteristics to estimate the runtime sharing benefit. Based on the estimated benefit, it instructs the Hamlet executor to switch between shared and non-shared execution strategies. Such fine-grained decisions allow Hamlet to maximize the sharing benefit at runtime. The Hamlet runtime executor propagates shared trend aggregates from previously matched events to newly matched events in an online fashion, i.e., without constructing event trends.

Contributions. Hamlet offers the following key innovations.

1. We present a novel framework Hamlet for optimizing a workload of queries computing aggregation over Kleene pattern matches, called event trends. To the best of our knowledge, Hamlet is the first to seamlessly integrate the power of online event trend aggregation and adaptive execution sharing among queries.

2. We introduce the Hamlet graph to compactly capture trends matched by queries in the workload. We partition the graph into smaller graphlets by event types and time. Hamlet then selectively shares trend aggregation in some graphlets among multiple queries.

3. We design a lightweight sharing benefit model to quantify the trade-off between the benefit of sharing and the overhead of maintaining the intermediate trend aggregates per query at runtime.

4. Based on the benefit of sharing sub-patterns, we propose an adaptive sharing optimizer. It selects a subset of queries among which it is beneficial to share this sub-pattern and determines the time interval during which this sharing remains beneficial.

5. Our experiments on several real world stream data sets demonstrate that Hamlet achieves up to five orders of magnitude performance improvement over state-of-the-art approaches.

Outline. Section 2 describes preliminaries. Sections 3 and 4 describe the core Hamlet techniques: online trend aggregation and dynamic sharing optimizer. We present experiments, review related work and conclude the paper in Sections 5, 6, and 7, respectively.

2 PRELIMINARIES

2.1 Basic Notions

Time is represented by a linearly ordered set of time points $(\mathbb{T}, \leq)$, where $\mathbb{T} \subseteq \mathbb{Q}^+$ are the non-negative rational numbers. An ***event*** e is a data tuple describing an incident of interest to the application.

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. A specific attribute $attr$ of E is referred to as $E.attr$. Table 2 summarizes the notation.

Events are sent by event producers (e.g., vehicles and mobile devices) to an ***event stream*** I. We assume that events arrive in order by their time stamps. Existing approaches to handle out-of-order events can be applied [11, 26, 27, 42].

An event consumer (e.g., Uber stream analytics) continuously monitors the stream with ***event queries***. We adopt the commonly used query language and semantics from SASE [9, 45, 46]. The query workload in Figure 1 is expressed in this language. We assume that the workload is static. Adding or removing a query from a workload requires migration of the execution plan to a new workload which can be handled by existing approaches [24, 49].

Notation	Description
$e.time$	Time stamp of event e
$e.type$	Type of event e
$E.attr$	Attribute $attr$ of event type E
$start(q)$	Start types of the pattern of query q
$end(q)$	End types of the pattern of query q
$pt(E, q)$	Predecessor types of event type E w.r.t query q
$pe(e, q)$	Predecessor events of event e w.r.t query q
n	Number of events per window
g	Number of events per graphlet
b	Number of events per burst
k	Number of queries in the workload Q
k_s	Number of queries share the graphlet G_E
k_n	Number of queries not share the graphlet G_E
t	Number of event types per query
s	Number of snapshots
s_c	Number of snapshots created from one event burst
s_p	Number of snapshots in one shared graphlet

Table 2: Table of notations

Definition 2.1. (**Kleene Pattern**) A pattern P can be in the form of E, P_1+, (NOT P_1), SEQ(P_1, P_2), ($P_1 \vee P_2$), or ($P_1 \wedge P_2$), where E is an event type, P_1, P_2 are patterns, + is a Kleene plus, NOT is a negation, SEQ is an event sequence, $\vee$ is a disjunction, and $\wedge$ is a conjunction. P_1 and P_2 are called sub-patterns of P. If a pattern P contains a Kleene plus operator, P is called a Kleene pattern.

Definition 2.2. (**Event Trend Aggregation Query**) An event trend aggregation query q consists of five clauses:

- Aggregation result specification (RETURN clause),
- Kleene pattern P (PATTERN clause) as per Definition 2.1,
- Predicates θ (optional WHERE clause),
- Grouping G (optional GROUPBY clause), and
- Window w (WITHIN/SLIDE clause).

Definition 2.3. (**Event Trend**) Let q be a query per Definition 2.2. An event trend $tr = (e_1, \ldots, e_k)$ corresponds to a sequence of events that conform to the pattern P of q. All events in a trend tr satisfy predicates θ, have the same values of grouping attributes G, and are within one window w of q.

Aggregation of Event Trends. Within each window specified by the query q, event trends are grouped by the values of grouping attributes G. Aggregates are then computed per group. Hamlet focuses on distributive (COUNT, MIN, MAX, SUM) and algebraic aggregation functions (AVG) since they can be computed incrementally [16]. Let E be an event type, $attr$ be an attribute of E, and e be an event of type E. While COUNT($*$) returns the number of all trends per group, COUNT(E) computes the number of all events e in all trends per group. SUM($E.attr$) (AVG($E.attr$)) calculates the summation (average) of the value of $attr$ of all events e in all trends per group. MIN($E.attr$) (MAX($E.attr$)) computes the minimal (maximal) value of $attr$ for all events e in all trends per group.

2.2 Hamlet Approach in a Nutshell

Given a workload of event trend aggregation queries Q and a high-rate event stream I, the ***Multi-query Event Trend Aggregation Problem*** is to evaluate the workload Q over the stream I such that the average query latency of all queries in Q is minimal. The latency of a query $q \in Q$ is measured as the difference between the time point of the aggregation result output by the query q and the arrival time of the last event that contributed to this result.

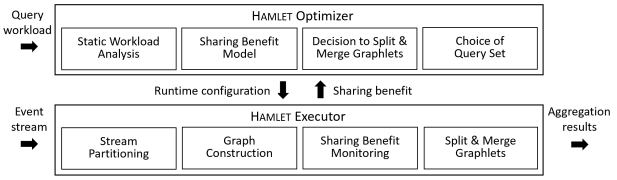


Figure 2: Hamlet Framework

We design the Hamlet framework (Figure 2). To reveal all sharing opportunities in the workload at compile time, the ***Hamlet Optimizer*** identifies sharable queries and translates them into a Finite State Automaton-based representation, called the merged query template (Section 3.1). Based on this template, the optimizer reveals which sub-patterns could potentially be shared by which queries. At runtime, the optimizer estimates the sharing benefit depending on the current stream characteristics to make fine-grained sharing decisions. Each sharing decision determines which queries share the processing of which Kleene sub-patterns and for how long (Section 4). These decisions along with the template are encoded into the runtime configuration to guide the executor.

Hamlet Executor partitions the stream by the values of grouping attributes. To enable shared execution despite different windows of sharable queries, the executor further partitions the stream into panes that are sharable across overlapping windows [10, 17, 24, 25]. Based on the merged query template for each set of sharable queries, the executor compactly encodes matched trends within a pane into the Hamlet graph. More precisely, matched events are modeled as nodes, while event adjacency relations in a trend are edges of the graph. Based on this graph, we incrementally compute trend aggregates by propagating intermediate aggregates along the edges from previously matched events to new events – without constructing the actual trends. This reduces the time complexity of trend aggregation from exponential to quadratic in the number of matched events compared to two-step approaches [19, 22, 32, 46].

The Hamlet graph is partitioned into sub-graphs, called graphlets, by event type and time stamps to maximally expose runtime opportunities to share these graphlets among queries. Since the value of aggregates may be differ for distinct queries, we capture these aggregate values per query as "snapshots" and share the propagation of snapshots through shared graphlets (Section 3.3).

Lastly, the executor implements the sharing decisions imposed by the optimizer. This may involve dynamically splitting a shared graphlet into several non-shared graphlets or, vice-versa, merging several non-shared graphlets into one shared graphlet (Section 4.2).

3 CORE HAMLET EXECUTION TECHNIQUES

Assumptions. To keep the discussion focused on the core concepts, we make simplifying assumptions in Sections 3 and 4. We drop them to extend Hamlet to the broad class of trend aggregation queries (Definition 2.2) in the technical report [33]. These assumptions include: (1) queries compute the number of trends per window COUNT(∗); (2) query patterns do not contain disjunction, conjunction nor negation; and (3) Kleene plus operator is applied to an event type and appears once per query.

In Section 3.1, we describe the workload and stream partitioning. We introduce strategies for processing queries without sharing in Section 3.2 versus with *shared* online trend aggregation in Section 3.3. In Section 4, we present the runtime optimizer that makes these sharing decisions.

3.1 Workload Analysis and Stream Partitioning

Given that the workload may contain queries with different Kleene patterns, aggregation functions, windows, and groupby clauses, Hamlet takes the following pre-processing steps: (1) it breaks the workload into sets of sharable queries at compile time; (2) it then constructs the Hamlet query template for each sharable query set; and (3) it partitions the stream by window and groupby clauses for each query template at runtime.

Definition 3.1. (**Shareable Kleene Sub-pattern**) Let Q be a workload and E be an event type. Assume that a Kleene sub-pattern $E+$ appears in queries $Q_E \subseteq Q$ and $|Q_E| > 1$. We say that $E+$ is shareable by queries Q_E.

However, sharable Kleene sub-patterns cannot always be shared due to other query clauses. For example, queries having COUNT(∗), MIN($E.$ *attr*) or MAX($E.attr$) can only be shared with queries that compute these same aggregates. In contrast, since AVG($E.attr$) is computed as SUM($E.attr$) divided by COUNT(E), queries computing AVG($E.attr$) can be shared with queries that calculate SUM($E.attr$) or COUNT(E). We therefore define sharable queries below.

Definition 3.2. (**Sharable Queries**) Two queries are *sharable* if their patterns contain at least one sharable Kleene sub-pattern, their aggregation functions can be shared, their windows overlap, and their grouping attributes are the same.

To facilitate the shared runtime execution of each set of sharable queries, each pattern is converted into its Finite State Automaton-based representation [9, 14, 45, 46], called ***query template***. We adopt the state-of-the-art algorithm [34] to convert each pattern in in the workload Q into its template.

Figure 3(a) depicts the template of query q_1 with pattern SEQ(A, $B+$). States, shown as rectangles, represent event types in the pattern. If a transition connects a type E_1 with a type E_2 in a template of a query q, then events of type E_1 precede events of type E_2 in a trend matched by q. E_1 is called a ***predecessor type*** of E_2, denoted $E_1 \in pt(E_2, q)$. A state without ingoing edges is a ***start type***, and a state shown as a double rectangle is an ***end type*** in a pattern.

Example 3.3. In Figure 3(a), events of type B can be preceded by events of types A and B in a trend matched by q_1, i.e., $pt(B, q_1) = \{A, B\}$. Events of type A are not preceded by any events, $pt(A, q_1) = \emptyset$. Events of type A start trends and events of type B end trends matched by q_1, i.e., $start(q_1) = \{A\}$ and $end(q_1) = \{B\}$.

Our Hamlet system processes the entire workload Q instead of each query in isolation. To expose all sharing opportunities in Q, we convert the entire workload Q into one ***Hamlet query template***. It is constructed analogously to a query template with two additional rules. First, each event type is represented in the merged template only once. Second, each transition is labeled by the set of queries for which this transition holds.

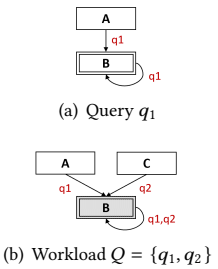
Example 3.4. Figure 3(b) depicts the template for the workload $Q = \{q_1, q_2\}$ where query q_1 has pattern SEQ(A, $B+$) and query q_2 has pattern SEQ(C, $B+$). The transition from B to itself is labeled by two queries q_1 and q_2. This transition corresponds to the shareable Kleene sub-pattern $B+$ in these queries (highlighted in gray).

The event stream is first partitioned by the grouping attributes. To enable shared execution despite different windows of sharable queries, Hamlet further partitions the stream into ***panes*** that are sharable across overlapping windows [10, 17, 24, 25]. The size of a pane is the greatest common divisor (gcd) of all window sizes and window slides. For example, for two windows (WITHIN 10 *min* SLIDE 5 *min*) and (WITHIN 15 *min* SLIDE 5 *min*), the gcd is 5 minutes. In this example, a pane contains all events per 5 minutes interval. For each set of sharable queries, we apply the Hamlet optimizer and executor within each pane.

3.2 Non-Shared Online Trend Aggregation

For the non-shared execution, we describe below how the Hamlet executor leverages state-of-the-art online trend aggregation approach [34] to compute trend aggregates for each query *independently from all other queries*. Given a query q, it encodes all trends matched by q in a query graph. The nodes in the graph are events matched by q. Two events e' and e are connected by an edge if e' and e are adjacent in a trend matched by q. The event e' is called a ***predecessor event*** of e. At runtime, trend aggregates are propagated along the edges. In this way, we aggregate trends online, i.e., without actually constructing them.

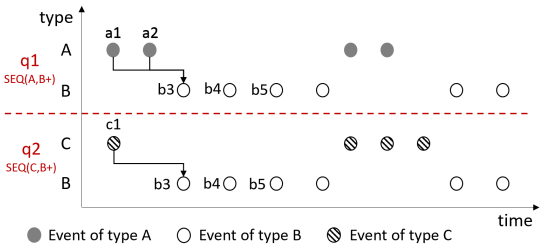
Assume a query q computes the number of trends COUNT(∗). When an event e arrives, e is inserted in the graph for q and the ***intermediate trend count*** of e (denoted $count(e, q)$) is computed. $count(e, q)$ corresponds to the number of trends that are matched by q and end at e. If e is of start type of q, e starts a new trend. Thus, $count(e, q)$ is incremented by one. In addition, e extends all trends that were previously matched by q. Thus, $count(e, q)$ is incremented by the sum of the intermediate trend counts of the



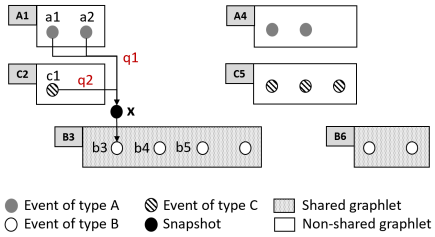
(a) Query q_1

(b) Workload $Q = \{q_1, q_2\}$

Figure 3: Template



(a) Non-shared Greta graph



(b) Shared Hamlet graph

Figure 4: Non-shared vs shared execution

predecessor events of e that were matched by q (denoted $pe(e, q)$).

$$start(e, q) = (e.type \in start(q))?\ 1\ :\ 0$$

$$count(e, q) = start(e, q) + \sum_{e' \in pe(e,q)} count(e', q) \quad (1)$$

The ***final trend count*** of q is the sum of intermediate trend counts of all matched events of $end(q)$ (Equation 2).

$$fcount(q) = \sum_{e.type \in end(q)} count(e, q) \quad (2)$$

Example 3.5. Continuing Example 3.4, a graph is maintained per each query in the workload $Q = \{q_1, q_2\}$ in Figure 4(a). For readability, we sort all events by their types and timestamps. Events of types A, B, and C are displayed as gray, white, and striped circles, respectively. We highlight the predecessor events of event b_3 by edges. All other edges are omitted for compactness. When b_3 arrives, two trends (a_1, b_3) and (a_2, b_3) are matched by q_1. Thus, $count(b_3,q_1) = count(a_1, q_1) + count(a_2, q_1) = 2$. However, only one trend (c_1, b_3) is matched by q_2. Thus, $count(b_3, q_2) = count(c_1, q_2) = 1$.

Complexity Analysis. Figure 4(a) illustrates that each event of type B is stored and processed once for each query in the workload Q, introducing significant re-computation and replication overhead. Let k denote the number of queries in the workload Q and n the number of events. Each query q stores each matched event e and computes the intermediate count of e per Equation 1. All predecessor events of e must be accessed, with e having at most n predecessor events. Thus, the time complexity of non-shared online trend aggregation is computed as follows:

$$NonShared(Q) = k \times n^2 \quad (3)$$

Events that are matched by k queries are replicated k times (Figure 4(a)). Each event stores its intermediate trend count. In addition, one final result is stored per query. Thus, the space complexity is $O(k \times n + k) = O(k \times n)$.

3.3 Shared Online Trend Aggregation

In Equation 3, the overhead of processing each event once per query in the workload Q is represented by the multiplicative factor k. Since the number of queries in a production workload may reach hundreds to thousands [38, 44], this re-computation overhead can be significant. Thus, we design an efficient shared online trend aggregation strategy that encapsulates bursts of events of the same type in a graphlet such that the propagation of trend aggregates within these graphlets can be shared among several queries.

Definition 3.6. (**Graphlet**) Let $q \in Q$ be a query and T be a set of event types that appear in the pattern of q. A *graphlet* G_E is a graph of events of type E, if no events of type $E' \in T$, $E' \neq E$, are matched by q during the time interval $(e_f.time, e_l.time)$, where $e_f.time$ and $e_l.time$ are the timestamps of the first and the last events in G_E, respectively. If new events can be added to a graphlet G_E without violating the constraints above, the graphlet G_E is called *active*. Otherwise, G_E is called *inactive*.

Definition 3.7. (**Shared Graphlet, Hamlet Graph**) Let $E+$ be a Kleene sub-pattern that is shareable by queries $Q_E \subseteq Q$ (Definition 3.1). We call a graphlet G_E of events of type E a shared graphlet. The set of all interconnected shared and non-shared graphlets for a workload Q is called a Hamlet graph.

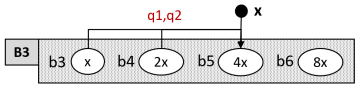
Example 3.8. In Figure 4(b), matched events are partitioned into six graphlets A_1–B_6 by their types and timestamps. For example, graphlets B_3 and B_6 are of type B. They are shared by queries q_1 and q_2. In contrast to the non-shared strategy in Figure 4(a), each event is stored and processed once for the entire workload Q. Events in A_1–C_2 are predecessors of events in B_3, while events in A_1–C_5 are predecessors of events in B_6. For readability, only the predecessor events of b_3 are highlighted by edges in Figure 4(b). All other edges are omitted. a_1 and a_2 are predecessors of b_3 only for q_1, while c_1 is a predecessor of b_3 only for q_2.

Example 3.8 illustrates the following two challenges of online shared event trend aggregation.

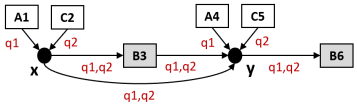
Challenge 1. Given that event b_3 has different predecessors for queries q_1 and q_2, the computation of the intermediate trend count of b_3 (and all other events in graphlets B_3 and B_6) cannot be directly shared by queries q_1 and q_2.

Challenge 2. If queries q_1 or q_2 have predicates, then not all previously matched events are qualified to contribute to the trend count of a new event. Assume that the edge between events b_4 and b_5 holds for q_1 but not for q_2 due to predicates, and all other edges hold for both queries. Then $count(b_4, q_1)$ contributes to $count(b_5, q_1)$, but $count(b_4, q_2)$ does not contribute to $count(b_5, q_2)$.

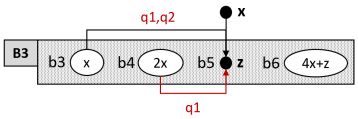
We tackle these challenges by introducing ***snapshots***. Intuitively, a snapshot is a variable that its value corresponds to an intermediate trend aggregate per query. In Figure 4(b), the propagation of a snapshot x within graphlet B_3 is shared by queries q_1 and q_2. We store the values of x per query (e.g., $x = 2$ for q_1 and $x = 1$ for q_2).



(a) Snapshot x at graphlet level



(b) Snapshots x and y at graphlet level



(c) Snapshot z at event level

Figure 5: Snapshots at graphlet and event levels

	Intermediate trend count
b_3	x
b_4	$x + count(b_3, Q) = 2x$
b_5	$x+count(b_3, Q)+count(b_4, Q) = 4x$
b_6	$x + count(b_3, Q) + count(b_4, Q)+$ $count(b_5, Q) = 8x$

Table 3: Shared propagation of x within B_3

	Query q_1	**Query q_2**
x	$sum(A_1, q_1) = 2$	$sum(C_2, q_2) = 1$
y	$value(x, q_1)+$ $sum(B_3, q_1)+$ $sum(A_4, q_1)$ = $2+15*2+2 = 34$	$value(x, q_2)+$ $sum(B_3, q_2)+$ $sum(C_5, q_2)$ = $1+15*1+3 = 19$

Table 4: Values of snapshots x and y per query

	Query q_1	**Query q_2**
z	$value(x, q_1)+$ $count(b_3, q_1)+$ $count(b_4, q_1) = 8$	$value(x, q_2)+$ $count(b_3, q_2) = 2$
y	$value(x, q_1)+$ $sum(B_3, q_1)+$ $sum(A_4, q_1) = 34$	$value(x, q_2)+$ $sum(B_3, q_2)+$ $sum(C_5, q_2) = 15$

Table 5: Values of snapshots z and y per query

Definition 3.9. (**Snapshot at Graphlet Level**) Let E' and E be distinct event types. Let $E+$ be a Kleene sub-pattern that is shared by queries $Q_E \subseteq Q$, $q \in Q_E$. Let $E' \in pt(E, q)$ and $G_{E'}$ and G_E be graphlets of events of types E' and E, respectively. Assume for any events $e' \in G_{E'}, e \in G_E$, $e'.time < e.time$ holds. A snapshot x of the graphlet $G_{E'}$ is a variable whose value is computed per query q and corresponds to the intermediate trend count of the query q at the end of the graphlet $G_{E'}$.

$$value(x, q) = sum(G_{E'}, q) = \sum_{e' \in G_{E'}} count(e', q) \tag{4}$$

The propagation of snapshot x through the graphlet G_E follows Equation 1 and is shared by queries Q_E.

Example 3.10. When graphlet B_3 starts, a snapshot x is created. x captures the intermediate trend count of query q_1 (q_2) based on the intermediate trend counts of all events in graphlet A_1 (C_2). x is propagated through graphlet B_3 as shown in Figure 5(a) and Table 3.

Analogously, when graphlet B_6 starts, a new snapshot y is created. The value of y is computed for queries q_1 (q_2) based on the value of x for q_1 (q_2) and graphlets B_3 and A_4 (C_5). Figure 5(b) illustrates the connections between snapshots and graphlets. The edges from graphlets A_1 and A_4 (C_2 and C_5) hold only for query q_1 (q_2). Other edges hold for both queries q_1 and q_2.

Table 4 captures the values of snapshots x and y per query. For compactness, $sum(A_1, q_1)$ denotes the sum of intermediate trend counts of all events in A_1 that are matched by q_1 (Equation 4). When the snapshot y is created, the value of x per query is plugged in to obtain the value of y per query. The propagation of y through B_6 is shared by q_1 and q_2. In this way, only one snapshot is propagated at a time to keep the overhead of snapshot maintenance low.

To enable shared trend aggregation despite expressive predicates, we now introduce snapshots at the event level.

Definition 3.11. (**Snapshot at Event Level**) Let G_E be a graphlet that is shared by queries $Q_E \subseteq Q$. Let $q_1, q_2 \in Q_E$ and $e_1, e_2 \in G_E$ be events such that the edge (e_1, e_2) holds for q_1 but does not hold for q_2 due to predicates. A snapshot z is the intermediate trend count of e_2 that is computed for q_1 and q_2 per Equation 1 and propagated through the graphlet G_E for all queries in Q_E.

Example 3.12. In Figure 5(c), assume that the edge between events b_4 and b_5 holds for query q_1 but not for query q_2 due to predicates. All other edges hold for both queries. Then, $count(b_4, q_1)$ contributes to $count(b_5, q_1)$, but $count(b_4, q_2)$ does not contribute to $count(b_5, q_2)$. To enable shared processing of graphlet B_3 despite predicates, we introduce a new snapshot z as the intermediate trend count of b_5 and propagate both snapshots x and z within graphlet B_3. Table 5 summarizes the values of z and y per query.

Shared Online Trend Aggregation Algorithm computes the number of trends per query $q \in Q$ in the stream I. For simplicity, we assume that the stream I contains events within one pane. For each event $e \in I$ of type E, Algorithm 1 performs two steps:

(1) ***Hamlet graph construction*** (Lines 4–15). When an event e of type E arrives and is matched by a query $q \in Q$, e is inserted into a graphlet G_E that stores events of type E (Line 15). if there is no active graphlet G_E of events of type E, we create a new graphlet G_E, mark it as active and store it in the Hamlet graph G. If the graphlet G_E is shared by queries $Q_E \subseteq Q$, then we create a snapshot x at graphlet level. x captures the values of intermediate trend counts per query per Equation 4 at the end of graphlet $G_{E'}$ that stores events of type E', $E' \in pt(E, q)$. We save these values of x per query in the table of snapshots S (Lines 7–14). Also, for each query $q \in Q$ with event types T, we mark all graphlets $G_{E'}$ of events of type $E' \in T$, $E' \neq E$, as inactive (Lines 4–6).

(2) ***Trend count computation*** (Lines 17–25). If G_E is shared by queries $Q_E \subseteq Q$ and the set of predecessor events of e is identical for all queries Q_E, then we compute the count of e for queries Q_E per Equation 1 (Lines 17–19). If G_E is shared but the sets of predecessor events of e differ among the different queries in Q_E due to predicates, then we create a snapshot y as the intermediate trend count of e. We compute the value of y for each query $q \in Q_E$ per Equation 1 and save it in the table of snapshots S (Lines 20–22). If G_E is not shared, the algorithm defaults to non-shared trend count propagation (Section 3.2) (Line 23). If E is an end type for a query $q \in Q$ (Section 3.1), we increment the final count of q in the table

of results R by the intermediate trend count of e for q (Lines 24–25). Lastly, we return the table of results R (Line 26). Due to the space constraints, correctness of Algorithm 1 is proven in [33].

Data Structures. Algorithm 1 utilizes the following physical data structures.

(1) ***Hamlet graph*** G is a set of all graphlets. Each graphlet has two metadata flags *active* and *shared* (Definitions 3.6 and 3.7).

(2) ***A hash table of snapshot coefficients*** per event e. The intermediate trend count of e may be an expression composed of several snapshots. In Figure 5(c), $count(b_6, Q) = 4x + z$. Such composed expressions are stored in a hash table per event that maps a snapshot to its coefficient. In this example, $x \mapsto 4$ and $z \mapsto 1$ for b_6.

(3) ***A hash table of snapshots*** S is a mapping from a snapshot x and a query q to the value of x for q (Tables 4 and 5).

(4) ***A hash table of trend count results*** R is a mapping from a query q to its corresponding trend count.

Algorithm 1 Hamlet shared online trend aggregation

Input: Query workload Q, event stream I, Hamlet graph G, hash table of snapshots S
Output: Hash table of results R

1: $G \leftarrow \emptyset$, $S, R \leftarrow$ empty hash tables
2: **for each** event $e \in I$ with $e.type = E$ **do**
3: // **Step 1: Hamlet graph construction**
4: **for each** $q \in Q$ with event types T **do**
5: **for each** $E' \in T,\ E' \neq E$ **do**
6: $G_{E'} \leftarrow getGraphlet(G, E')$, $G_{E'}.active \leftarrow false$
7: **if not** $G_E.active$ **then**
8: $G_E \leftarrow createGraphlet(), G_E.active \leftarrow true, G \leftarrow G \cup G_E$
9: **if** $G_E.shared$ by $Q_E \subseteq Q$ **then**
10: $x \leftarrow createSnapshot()$
11: **for each** $q \in Q_E$ **do**
12: **for each** $E' \in pt(E, q), E' \neq E$ **do**
13: $G_{E'} \leftarrow getGraphlet(G, E')$
14: $S(x, q) \leftarrow S(x, q) + sum(G_{E'}, q)$
15: insert e into G_E
16: // **Step 2: Trend count computation**
17: **if** $G_E.shared$ by $Q_E \subseteq Q$ **then**
18: **if** $\forall q \in Q_E$ $pe(e, q)$ are identical **then**
19: $count(e, Q_E) \leftarrow x + \sum_{e' \in pe(e, Q_E)} count(e', Q_E)$
20: **else** $y \leftarrow createSnapshot()$, $count(e, Q_E) = y$
21: **for each** $q \in Q_E$ **do**
22: $S(y, q) \leftarrow value(x, q) + \sum_{e' \in pe(e,q)} count(e', q)$
23: **else** $count(e, q) \leftarrow \sum_{e' \in pe(e,q)} count(e', q)$
24: **if** $E \in end(q)$ for a query $q \in Q$ **then**
25: $R(q) \leftarrow R(q) + count(e, q)$
26: **return** R

Complexity Analysis. We use the notations in Table 2. For each event e that is matched by a query $q \in Q$, Algorithm 1 computes the intermediate trend count of e in an online fashion. This requires access to all predecessor events of e. In the worst case, n previously matched events are the predecessor events of e. Since the intermediate trend count of e can be an expression that is composed of s snapshots, the trend count of e is stored in the hash table that maps snapshots to their coefficients. Therefore, the time complexity of trend count computation is $O(n^2 \times s)$.

In addition, Algorithm 1 maintains snapshots to enable shared trend count computation. To compute the values of s snapshots for each query q in the workload of k queries, the algorithm accesses g events in t graphlets $G_{E'}$ of events of type $E' \in T,\ E' \neq E$. Thus, the time complexity of snapshot maintenance is $O(s \times k \times g \times t)$. In summary, time complexity of Algorithm 1 is computed as follows:

$$Shared(Q) = n^2 \times s + s \times k \times g \times t \quad (5)$$

Algorithm 1 stores each matched event in the Hamlet graph once for the entire workload. Each shared event stores a hash table of snapshot coefficients. Each non-shared event stores its intermediate trend count. In addition, the algorithm stores snapshot values per query. Lastly, the algorithm stores one final result per query. Thus, the space complexity is $O(n+n\times s+s\times k+k) = O(n\times s+s\times k)$.

4 DYNAMIC SHARING OPTIMIZER

We first model the runtime benefit of sharing trend aggregation (Section 4.1). Based on this benefit model, our Hamlet optimizer makes runtime sharing decisions for a given set of queries (Section 4.2). Lastly, we describe how to choose a set of queries that share a Kleene sub-pattern (Section 4.3).

4.1 Dynamic Sharing Benefit Model

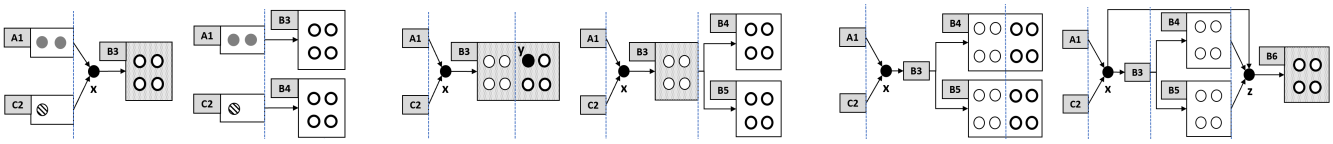
On the up side, shared trend aggregation avoids the re-computation overhead for each query in the workload. On the down side, it introduces overhead to maintain snapshots. Next, we quantify the trade-off between shared versus non-shared execution.

Equations 3 and 5 determine the cost of non-shared and shared strategies of all events within the window for the entire workload Q based on stream statistics. In contrast to these coarse-grained static decisions, the Hamlet optimizer makes ***fine-grained runtime decisions*** for each burst of events for a sub-set of queries $Q_E \subseteq Q$. Intuitively, a burst is a set of consecutive events of type E, the processing of which can be shared by queries Q_E that contain a E+ Kleene sub-pattern. The Hamlet optimizer decides at runtime if sharing a burst is beneficial. In this way, beneficial sharing opportunities are harvested for each burst at runtime.

Definition 4.1. (**Burst of Events**) Let E+ be a sub-pattern that is sharable by queries Q_E. Let T be the set of event types that appear in the patterns of queries Q_E, $E \in T$. A set of events of type E within a pane is called a *burst* B_E, if no events of type $E' \in T,\ E' \neq E$, are matched by the queries Q_E during the time interval $(e_f.time, e_l.time)$, where $e_f.time$ and $e_l.time$ are the timestamps of the first and the last events in B_E, respectively. If no events can be added to a burst B_E without violating the above constraints, the burst B_E is called *complete*.

Within each pane, events that belong to the same burst are buffered until a burst is complete. The arrival of an event of type E' or the end of the pane indicates that the burst is complete. In the following, we refer to complete bursts as bursts for compactness.

Hamlet restricts event types in a burst for the following reason. Assuming that a burst contained an event e of type E', the event e could be matched by one query q_1 but not by another query q_2 in Q_E. Snapshots would have to be introduced to differentiate between the aggregates of q_1 and q_2 (Section 3.3). Maintenance



(a) Shared B_3 (b) Non-shared B_3, B_4 (c) Shared B_3 (d) Non-shared B_4, B_5 (e) Non-shared B_4, B_5 (f) Shared B_6

Figure 6: Dynamic sharing decisions. Decision to merge B_3 in (a) and (b). Decision to split B_3 in (c) and (d). Decision to merge B_6 in (e) and (f).

of these snapshots may reduce the benefit of sharing. Thus, the previous sharing decision may have to be reconsidered as soon as the first event arrives that is matched by some queries in Q_E.

Definition 4.2. (**Dynamic Sharing Benefit**) Let E+ be a Kleene sub-pattern that is shareable by queries Q_E, B_E be a burst of events of type E, b be the number of events in B_E, s_c be the number of snapshots that are created from this burst B_E, and s_p be the number of snapshots that are propagated to compute the intermediate trend counts for the burst B_E. Let G_E denote a shared graphlet and G_E^i denote a set of non-shared graphlets (one graphlet per each query in Q_E). Other notations are consistent with previous sections (Table 2).

The *benefit* of sharing a graphlet G_E by the queries Q_E is computed as the difference between the cost of the non-shared and shared execution of the burst B_E.

$$\begin{aligned} &Shared(G_E, Q_E) = b \times n \times s_p + s_c \times k \times g \times t \\ &NonShared(G_E^i, Q_E) = k \times b \times n \\ &Benefit(G_E, Q_E) = NonShared(G_E^i, Q_E) - Shared(G_E, Q_E) \end{aligned} \quad (6)$$

If $Benefit(G_E, Q_E) > 0$, then it is beneficial to share trend aggregation within the graphlet G_E by the queries Q_E.

Based on Definition 4.2, we conclude that the more queries k share trend aggregation, the more events g are in shared graphlets, and the fewer snapshots s_c and s_p are maintained at a time, the higher the benefit of sharing will be. Based on this conclusion, our dynamic Hamlet optimizer decides to share or not to share online trend aggregation (Section 4.2).

4.2 Decision to Split and Merge Graphlets

Our dynamic Hamlet optimizer monitors the sharing benefit depending on changing stream conditions at runtime. Let B+ be a sub-pattern sharable by queries $Q_B = \{q_1, q_2\}$. In Figure 6, pane boundaries are depicted as dashed vertical lines and bursts of newly arrived events of type B are shown as bold empty circles. For each burst, the optimizer has a choice of sharing (Figure 6(a)) versus not sharing (Figure 6(b)). It concludes that it is beneficial to share based on calculations in Equation 7.

$$\begin{aligned} &Shared(B_3, Q_B) = 4 \times 7 \times 1 + 1 \times 2 \times 4 \times 2 = 44 \\ &NonShared(\{B_3, B_4\}, Q_B) = 2 \times 4 \times 7 = 56 \\ &Benefit(B_3, Q_B) = 56 - 44 = 12 > 0 \end{aligned} \quad (7)$$

Decision to Split. However, when the next burst of events of type B arrives, a new snapshot y has to be created due to predicates during the shared execution in Figure 6(c). In contrast, the non-shared strategy processes queries q_1 and q_2 independently from each other (Figure 6(d)). Now the overhead of snapshot maintenance is no longer justified by the benefit of sharing (Equation 8).

$$\begin{aligned} &Shared(B_3, Q_B) = 4 \times 11 \times 2 + 1 \times 2 \times 8 \times 2 = 120 \\ &NonShared(\{B_4, B_5\}, Q_B) = 2 \times 4 \times 11 = 88 \\ &Benefit(B_3, Q_B) = 88 - 120 = -32 < 0 \end{aligned} \quad (8)$$

Thus, the optimizer decides to split the shared graphlet B_3 into two non-shared graphlets B_4 and B_5 for the queries q_1 and q_2 respectively in Figure 6(d). Newly arriving events of type B then must be inserted into both graphlets B_4 and B_5. Their intermediate trend counts are computed separately for the queries q_1 and q_2. The snapshot x is replaced by its value for the query q_1 (q_2) within the graphlet B_4 (B_5). The graphlets A_1 and C_2 are collapsed.

Decision to Merge. When the next burst of events of type B arrives, we could either continue the non-shared trend count propagation within B_4 and B_5 (Figure 6(e)) or merge B_4 and B_5 into a new shared graphlet B_6 (Figure 6(f)). The Hamlet optimizer concludes that the latter option is more beneficial in Equation 9. As a consequence, a new snapshot z is created as input to B_6. z consolidates the intermediate trend counts of the snapshot x and the graphlets B_3–B_5 per query q_1 and q_2.

$$\begin{aligned} &Shared(B_6, Q_B) = 4 \times 15 \times 1 + 1 \times 2 \times 4 \times 2 = 76 \\ &NonShared(\{B_4, B_5\}, Q_B) = 2 \times 4 \times 15 = 120 \\ &Benefit(B_6, Q_B) = 120 - 76 = 44 > 0 \end{aligned} \quad (9)$$

Complexity Analysis. The runtime sharing decision per burst has constant time complexity because it simply plugs in locally available stream statistics into Equation 6. A graphlet split comes for free since we simply continue graph construction per query (Figure 6(d)). Merging graphlets requires creation of one snapshot and calculation of its values per query (Figure 6(f)). Thus, the time complexity of merging is $O(k \times g \times t)$ (Equation 5). Since our workload is fixed (Section 2), the number of queries k and the number of types t per query are constants. Thus, the time complexity of merge is linear in the number of events per graphlet g. Merging graphlets requires storing the value of one snapshot per query. Thus, its space complexity is $O(k)$.

4.3 Choice of Query Set

To relax the assumption from Section 4.2 that a set of queries Q_E that share a Kleene sub-pattern E+ is given, we now select a sub-set of queries Q_E from the workload Q for which sharing E+ is the most beneficial among all other sub-sets of Q. In general, the search space of all sub-sets of Q is exponential in the number of queries in Q since all combinations of shared and non-shared queries in Q are considered. For example, if Q contains four queries, Figure 7

illustrates the search space of 12 possible execution plans of Q. Groups of queries in braces are shared. For example, the plan (134)(2) denotes that queries 1, 3, 4 share their execution, while query 2 is processed separately. The search space ranges from maximally shared (top node) to non-shared (bottom node) plans. Each plan has its execution cost associated with it. For example, the cost of the plan (134)(2) is computed as the sum of $Shared(G_E, \{1, 3, 4\})$ and $NonShared(G_E^i, 2)$ (Equation 6). The goal of the dynamic Hamlet optimizer is to find a plan with minimal execution cost.

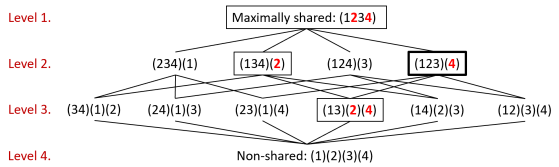


Figure 7: Search space of sharing plans

Traversing the exponential search space for each Kleene sub-pattern and each burst of events would jeopardize real-time responsiveness of Hamlet. Fortunately, most plans in this search space can be pruned without loosing optimality (Theorems 4.3 and 4.5). Intuitively, Theorem 4.3 states that it is always beneficial to share the execution of a query that introduces no new snapshots.

Theorem 4.3. *Let E+ be a Kleene sub-pattern that is shared by a set of queries Q_E and not shared by a set of queries Q_N, $Q_E \cap Q_N = \emptyset$, $k_s = |Q_E|$, and $k_n = |Q_N|$. For a burst of events of type E, let $q \in Q_E$ be a query that does not introduce new snapshots due to predicates for this burst of events (Definition 3.11). Then the following follows:*

$$Shared(Q_E) + NonShared(Q_N) \leq$$
$$Shared(Q_E \setminus \{q\}) + NonShared(Q_N \cup \{q\})$$

Due to space limitations, proof of Theorem 4.3 is in [33]. We formulate the following pruning principle per Theorem 4.3.

Snapshot-Driven Pruning Principle. Plans at Level 2 of the search space that do not share queries that introduced no snapshots are pruned. All descendants of such plans are also pruned.

Example 4.4. In Figure 7, assume queries 1 and 3 introduced no snapshots, while queries 2 and 4 introduced snapshots. Then, four plans are considered because they share queries 1 and 3 with other queries. These plans are highlighted by frames. The other eight plans are pruned since they have higher execution costs.

Theorem 4.5 below states that if it is beneficial to share the execution of a query q with other queries Q, a plan that processes the query q separately from other queries $Q_E \subseteq Q$ will have higher execution costs than a plan that shares q with Q_E. The reverse of the statement also holds. Namely, if it is not beneficial to share the execution of a query q with other queries Q, a plan that shares the execution of q with other queries $Q_E \subseteq Q$ will have higher execution costs than a plan that processes q separately from Q_E.

Theorem 4.5. *Let E+ be a Kleene sub-pattern that is shareable by a set of queries Q, $Q = Q_E \cup Q_N$, and $q \in Q_E$. Then:*

$$If\ Shared(Q) \leq Shared(Q \setminus \{q\}) + NonShared(q),$$
$$then\ Shared(Q_E) + NonShared(Q_N) \leq$$
$$Shared(Q_E \setminus \{q\}) + NonShared(Q_N \cup \{q\})$$

This statement also holds if we replace all $\leq$ by $\geq$.

Proof of Theorem 4.5 is in the technical report [33]. We formulate the following pruning principle per Theorem 4.5.

Benefit-Driven Pruning Principle. Plans at Level 2 of the search space that do not share a query that is beneficial to share are pruned. Plans at Level 2 of the search space that share a query that is not beneficial to share are pruned. All descendants of such plans are also pruned.

Example 4.6. In Figure 7, if it is beneficial to share query 2, then we can safely prune all plans that process query 2 separately. That is, the plan (134)(2) and all its descendants are pruned. Similarly, if it is not beneficial to share query 4, we can safely exclude all plans that share query 4. That is, all siblings of (123)(4) and their descendants are pruned. The plan (123)(4) is chosen (highlighted by a bold frame).

Consequence of Pruning Principles. Based on all plans at Levels 1 and 2 of the search space, the optimizer classifies each query in the workload as either shared or non-shared. Thus, it chooses the optimal plan without considering plans at levels below 2.

Complexity Analysis. Given a burst of new events, let m be the number of queries that introduce new snapshots to share the processing of this burst of events. The number of plans at Levels 1 and 2 of the search space is $m + 1$. Thus both time and space complexity of sharing plan selection is $O(m)$.

Granularity of Hamlet Sharing Decision. Hamlet runtime sharing decisions are made per burst of events (Section 4.2). There can be several bursts per window (Definition 4.1). Within one burst, Hamlet has optimal time complexity [33]. According to the complexity analysis in Section 4.2, the choice of the query set has linear time complexity in the number of queries m that introduce snapshots due to predicates. By Section 4.3, the merge of graphlets has linear time complexity in the number of events g per graphlet. Hamlet would be optimal per window if it could make sharing decisions at the end of each window. However, waiting until all events per window arrive could introduce delays and jeopardise real-time responsiveness. Due to this low latency constraint, Hamlet makes sharing decisions per burst, achieving significant performance gain over competitors (Section 5.2).

5 EXPERIMENTAL EVALUATION

5.1 Experimental Setup

Infrastructure. We have implemented Hamlet in Java with JDK 1.8.0_181 running on Ubuntu 14.04 with 16-core 3.4GHz CPU and 128GB of RAM. Our code is available online [1]. We execute each experiment three times and report their average results here.

Data Sets. We evaluate Hamlet using four data sets.

- *New York city taxi and Uber real data set* [8] contains 2.63 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 average number of events per minute is 200.
- *Smart home real data set* [2] contains 4055 million measurements for 2125 plugs in 40 houses. Each event carries a timestamp in seconds, measurement, house identifiers, and voltage measurement value. The average number of events per minute is 20K.

• *Stock real data set* [5] contains up to 20 years of stock price history. Our sample data contains 2 million transaction records of 220 companies for 8 hours. Each event carries a time stamp in minutes, company identifier, price, and volume. The average number of events per minute is 4.5K.

• *Ridesharing data set* was created by our stream generator to control the rate and distribution of events of different types in the stream. This stream contains events of 20 event types such as request, pickup, travel, dropoff, cancel, etc. Each event carries a time stamp in seconds, driver and rider ids, request type, district, duration, and price. The attribute values are randomly generated. The average number of events per minute is 10K.

Event Trend Aggregation Queries. For each data set, we generated workloads similar to queries q_1–q_3 in Figure 1. We experimented with the two types of workloads described below.

• The first workload focuses on sharing Kleene closure because this is the most expensive operator in event trend aggregation queries (Definition 2.2). Further, the sharing of Kleene closure is a much overlooked topic in the literature; while the sharing of other query clauses (windows, grouping, predicates, and aggregation) has been well-studied in prior research and systems [10, 17, 24, 25, 28]. Thus, queries in this workload are similar to Examples 3.3–4.6. Namely, they have different patterns but their sharable Kleene subpattern, window, groupby clause, predicates, and aggregates are the same. We evaluate this workload in Figures 8–10.

• The second workload is more diverse since the queries have sharable Kleene patterns of length ranging from 1 to 3, windows sizes ranging from 5 to 20 minutes, different aggregates (e.g., COUNT, AVG, MAX, etc.), as well as groupbys and predicates on a variety of event types. We evaluate this workload in Figures 11–12.

The rate of events differs in different real data sets [2, 5, 8] that we used in our experiments. The window sizes are also different in the query workloads per data set. To make the results comparable across data sets, we vary the number of events per minute by a speed-up factor; which corresponds to the number of events per window divided by the window size in minutes. The default number of events per minute per data set is included in the description of each data set. Unless stated otherwise, the workload consists of 50 queries. We vary major cost factors (Definition 4.2), namely, the number of events and the number of queries.

Methodology. We experimentally compare Hamlet to the following state-of-the-art approaches:

• *MCEP* [22] is the most recently published state-of-the-art shared two-step approach. MCEP constructs all event trends prior to computing their aggregation. As shown in [22], it shares event trend construction. It outperforms other shared two-step approaches SPASS [39] and MOTTO [48].

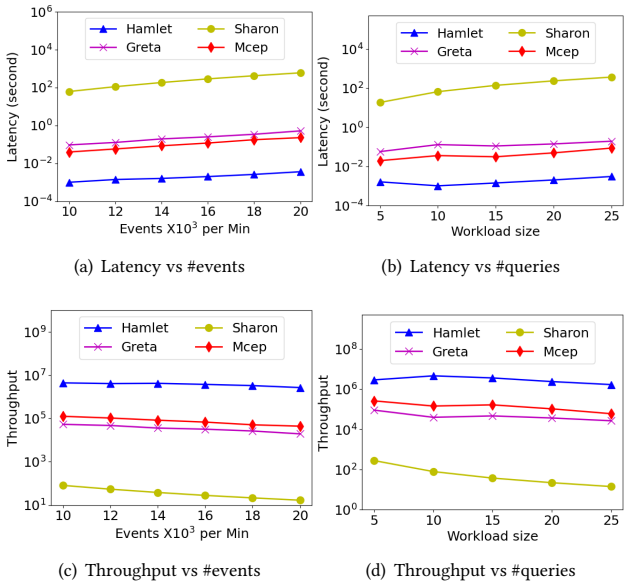
• *Sharon* [36] is a shared approach that computes event sequence aggregation online. That is, it avoids sequence construction by incrementally maintaining a count for each pattern. Sharon does not support Kleene closure. To mimic Kleene queries, we flatten them as follows. For each Kleene pattern $E+$, we estimate the length l of the longest match of $E+$ and specify a set of fixed-length sequence queries that cover all possible lengths up to l.

• *Greta* [34] supports Kleene closure and computes event trend aggregation online, i.e, without constructing all event trends. It achieves this online event trend aggregation by encoding all matched events and their adjacency relationships in a graph. However, Greta does not optimize for sharing a workload of queries. That is, each query is processed independently as described in Section 3.2.

Metrics. We measure *latency* in seconds as the average time difference between the time point of the aggregation result output by a query in the workload and the arrival time 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 consumption*, measured in bytes, corresponds to the maximal memory required to store snapshot expressions for Hamlet, the current event trend for MCEP, aggregates for Sharon, and matched events for Hamlet, MCEP, and Greta.

5.2 Experimental Results

Hamlet versus State-of-the-art Approaches. In Figures 8 and 9, we measure all metrics of all approaches while varying the number of events per minute from 10K to 20K and the number of queries in the workload from 5 to 25. We intentionally selected this setting to ensure that the two-step approach MCEP, the non-shared approach Greta, and the fixed-length sequence aggregation approach Sharon terminate within hours.



(a) Latency vs #events (b) Latency vs #queries

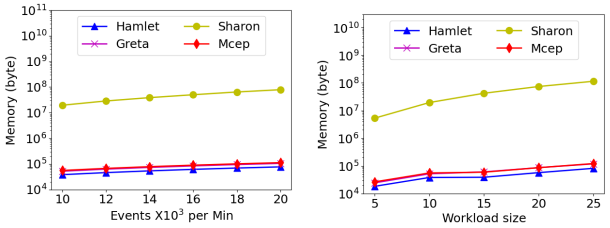
(c) Throughput vs #events (d) Throughput vs #queries

Figure 8: Hamlet versus state-of-the-art approaches (Ridesharing)

With respect to throughput, Hamlet consistently outperforms Sharon by 3–5 orders of magnitude, Greta by 1–2 orders of magnitude, and MCEP 7–76-fold (Figures 8(c) and 8(d)). We observe similar improvement with respect to latency in Figures 8(a) and 8(b). While Hamlet terminates within 25 milliseconds in all cases, Sharon needs up to 50 minutes, Greta up to 3 seconds, and MCEP up to 1 second. With respect to memory consumption, Hamlet, Greta, and MCEP perform similarly, while Sharon requires 2–3 orders of magnitude more memory than Hamlet in Figure 9.

Such poor performance of Sharon is not surprising because Sharon does not natively support Kleene closure. To detect all

Kleene matches, Sharon runs a workload of fixed-length sequence queries for each Kleene query. As Figure 8 illustrates, this overhead dominates the latency and throughput of Sharon. In contrast to Sharon, Greta and MCEP terminate within a few seconds in this low setting because both approaches not only support Kleene closure but also optimize its processing. In particular, Greta computes trend aggregation without constructing the trends but does not share trend aggregation among different queries in the workload. MCEP shares the construction of trends but computes trend aggregation as a post-processing step. Due to these limitations, Hamlet outperforms both Greta and MCEP with respect to all metrics.



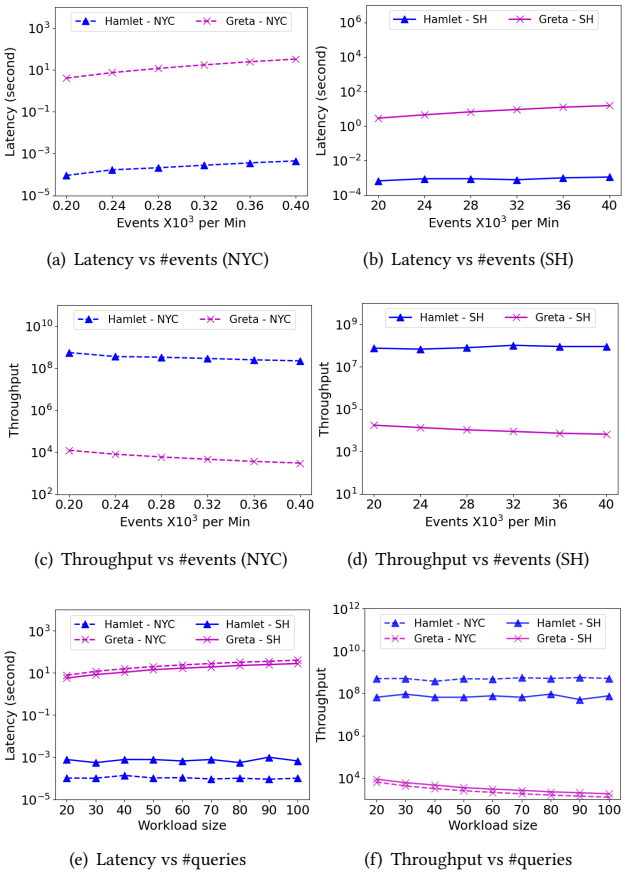
(a) Memory vs #events (b) Memory vs #queries

Figure 9: Hamlet vs state-of-the-art (Ridesharing)

However, the low setting in Figures 8 and 9 does not reveal the full potential of Hamlet. Thus in Figure 10, we compare Hamlet to the most advanced state-of-the-art online trend aggregation approach Greta using two real data sets. We measure latency and throughput, while varying the number of events per minute and the number of queries in the workload. Hamlet consistently outperforms Greta with respect to throughput and latency by 3–5 orders of magnitude. In practice this means that the response time of Hamlet is within half a second, while Greta runs up to 2 hours and 17 minutes for 400 events per minute in Figure 10(a).

Dynamic versus Static Sharing Decisions. Figures 11 and 12 compare the effectiveness of Hamlet dynamic sharing decisions to static sharing decisions. Each burst of events that can be shared contains 120 events on average in the stock data set. Our Hamlet dynamic optimizer makes sharing decisions at runtime per each burst of events (Section 4.1). The Hamlet executor splits and merges graphlets at runtime based on these optimization instructions (Section 4.2). The number of all graphlets ranges from 400 to 600, while the number of shared graphlets ranges from 360 to 500. In this way, Hamlet efficiently shares the beneficial Kleene sub-patterns within a subset of queries during its execution.

In Figures 11(a), 11(c) and 12(a), as the number of events per minute increases from 2K to 4K, the number of snapshots maintained by the Hamlet executor grows from 4K to 8K. As soon as the overhead of snapshot maintenance outweighs the benefits of sharing, the Hamlet optimizer decides to stop sharing. The Hamlet executor then splits these shared graphlets (Section 4.2). The Hamlet dynamic optimizer shares approximately 90% of bursts. The rest 10% of the bursts are not shared which substantially reduces the number of snapshots by around 50% compared to the shared execution.



(a) Latency vs #events (NYC) (b) Latency vs #events (SH)

(c) Throughput vs #events (NYC) (d) Throughput vs #events (SH)

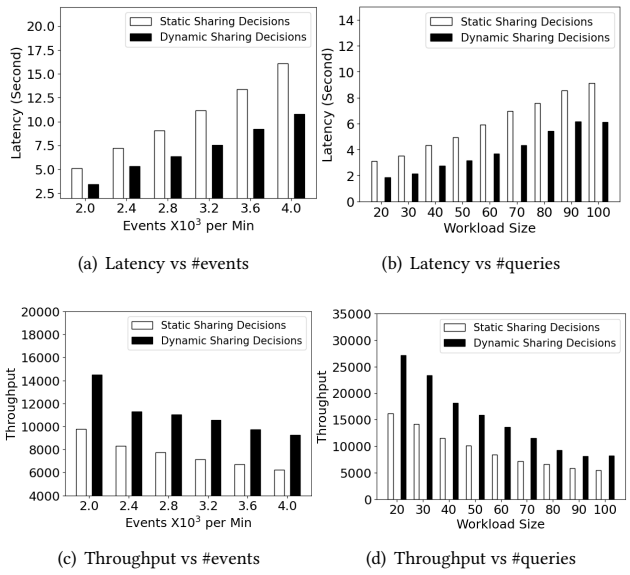
(e) Latency vs #queries (f) Throughput vs #queries

Figure 10: Hamlet versus state-of-the-art approaches (NY City Taxi (NYC) and Smart Home (SH) data sets)

In contrast, the static optimizer decides to share certain Kleene sub-patterns by a fixed set of queries during the entire window. Since these decisions are made at compile time, they do not incur overhead at runtime. However, these static decisions do not take the stream fluctuations into account. Consequently, these sharing decisions may do more harm than good by introducing significant CPU overhead of snapshot maintenance, causing non-beneficial shared execution. During the entire execution, the static optimizer always decides to share, and the number of snapshots grows dramatically from 10K to 20K. Therefore, our Hamlet dynamic sharing approach achieves 21–34% speed-up and 27–52% throughput improvement compared to the executor that obeys to static sharing decisions.

We observe similar gains of Hamlet with respect to memory consumption in Figure 12(a). Hamlet reduces memory by 25% compared to the executor based on static sharing decisions because the number of snapshots introduced by Hamlet dynamic sharing decisions is much less than the number of snapshots introduced by the static sharing decisions.

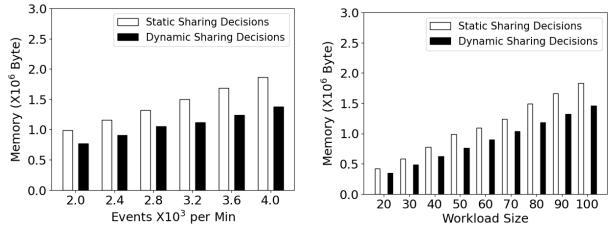
We also vary the number of queries in the workload from 20 to 100, and we observe similar gains by Hamlet dynamic sharing optimizer in terms of latency, throughput, and memory (depicted



(a) Latency vs #events (b) Latency vs #queries

(c) Throughput vs #events (d) Throughput vs #queries

Figure 11: Dynamic versus static sharing decisions (Stock data set)



(a) Memory vs #events (b) Memory vs #queries

Figure 12: Dynamic versus static sharing decisions (Stock data set)

in Figures 11(b), 11(d), and 12(b)). Hamlet can effectively leverage the beneficial sharing opportunities within a large query workload.

Lastly, we measured the runtime overhead of the Hamlet dynamic sharing decisions. Even though the number of sharing decisions ranges between 400 and 600 per window, the latency incurred by these decisions stays within 20 milliseconds (less than 0.2% of total latency per window) because these decisions are light-weight (Section 4.2). Also, the latency of one-time static workload analysis (Section 3.1) stays within 81 milliseconds. Thus, we conclude that the overhead of dynamic decision making and static workload analysis are negligible compared to their gains.

6 RELATED WORK

Complex Event Processing Systems (CEP) have gained popularity in the recent years [3, 4, 6, 7]. Some approaches use a Finite State Automaton (FSA) as an execution framework for pattern matching [9, 14, 45, 46]. Others employ tree-based models [31]. Some approaches study lazy match detection [23], compact event graph encoding [32], and join plan generation [21]. We refer to the recent survey [15] for further details. While these approaches support trend aggregation, they construct trends prior to their aggregation. Since the number of trends is exponential in the number of events per window [37, 46], such two-step approaches do not guarantee real-time response [34, 35]. Worse yet, they do not leverage sharing opportunities in the workload. The re-computation overhead is substantial for workloads with thousands of queries.

Online Event Trend Aggregation. Similarly to single-event aggregation, event trend aggregation has been actively studied. A-Seq [37] introduces online aggregation of event sequences, i.e., sequence aggregation without sequence construction. Greta [34] extends A-Seq by Kleene closure. Cogra [35] further generalizes online trend aggregation by various event matching semantics. However, none of these approaches addresses the challenges of multi-query workloads, which is our focus.

CEP Multi-query Optimization follows the principles commonly used in relational database systems [41], while focusing on pattern sharing techniques. RUMOR [19] defines a set of rules for merging queries in NFA-based RDBMS and stream processing systems. E-Cube [28] inserts sequence queries into a hierarchy based on concept and pattern refinement relations. SPASS [39] estimates the benefit of sharing for event sequence construction using intra-query and inter-query event correlations. MOTTO [48] applies merge, decomposition, and operator transformation techniques to re-write pattern matching queries. Kolchinsky et al. [22] combine sharing and pattern reordering optimizations for both NFA-based and tree-based query plans. However, these approaches do not support online aggregation of event sequences, i.e., they construct all event sequences prior to their aggregation, which degrades query performance. To the best of our knowledge, Sharon [36] and Muse [40] are the only solutions that support shared online aggregation. However, Sharon does not support Kleene closure. Worse yet, Sharon and Muse make static sharing decisions. In contrast, Hamlet harnesses additional sharing benefit thanks to dynamic sharing decisions depending on the current stream properties.

Multi-query Processing over Data Streams. Sharing query processing techniques are well-studied for streaming systems. NiagaraCQ [13] is a large-scale system for processing multiple continuous queries over streams. TelegraphCQ [12] introduces a tuple-based dynamic routing for inter-query sharing [29]. AStream [20] shares computation and resources among several queries executed in Flink [4]. Several approaches focus on sharing optimizations given different predicates, grouping, or window clauses [10, 17, 18, 24, 25, 43, 47]. However, these approaches evaluate Select-Project-Join queries with windows and aggregate single events. They do not support CEP-specific operators such as event sequence and Kleene closure that treat the order of events as a first-class citizen. Typically, they require the construction of join results prior to their aggregation. In contrast, Hamlet not only avoids the expensive event trend construction, but also exploits the sharing opportunities among trend aggregation queries with diverse Kleene patterns.

7 CONCLUSIONS

Hamlet integrates a shared online trend aggregation execution strategy with a dynamic sharing optimizer to maximize the benefit of sharing. It monitors fluctuating streams, recomputes the sharing benefit, and switches between shared and non-shared execution at runtime. Our experimental evaluation demonstrates substantial performance gains of Hamlet compared to state-of-the-art.

REFERENCES

[1] https://github.com/LeiMa0324/Hamlet.
[2] DEBS 2014 grand challenge: Smart homes. https://debs.org/grand-challenges/2014/.
[3] Esper. http://www.espertech.com/.
[4] Flink. https://flink.apache.org/.
[5] Historical stock data. http://www.eoddata.com.
[6] Microsoft StreamInsight. https://technet.microsoft.com/en-us/library/ee362541%28v=sql.111%29.aspx.
[7] Oracle Stream Analytics. https://www.oracle.com/middleware/technologies/stream-processing.html.
[8] Unified New York City taxi and Uber data. https://github.com/toddwschneider/nyc-taxi-data.
[9] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, pages 147–160, 2008.
[10] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, pages 336–347, 2004.
[11] B. Chandramouli, J. Goldstein, and D. Maier. High-performance dynamic pattern matching over disordered streams. *PVLDB*, 3(1):220–231, 2010.
[12] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.
[13] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD*, page 379–390, 2000.
[14] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. White. Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422, 2007.
[15] N. Giatrakos, E. Alevizos, A. Artikis, A. Deligiannakis, and M. Garofalakis. Complex event recognition in the Big Data era: A survey. *PVLDB*, 29(1):313–352, 2020.
[16] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, pages 29–53, 1997.
[17] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Three-level processing of multiple aggregate continuous queries. In *ICDE*, pages 929–940, 2012.
[18] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *VLDB*, page 297–308, 2003.
[19] M. Hong, M. Riedewald, C. Koch, J. Gehrke, and A. Demers. Rule-based multi-query optimization. In *EDBT*, pages 120–131, 2009.
[20] J. Karimov, T. Rabl, and V. Markl. AStream: Ad-hoc shared stream processing. In *SIGMOD*, page 607–622, 2019.
[21] I. Kolchinsky and A. Schuster. Join query optimization techniques for complex event processing applications. In *PVLDB*, pages 1332–1345, 2018.
[22] I. Kolchinsky and A. Schuster. Real-time multi-pattern detection over event streams. In *SIGMOD*, pages 589–606, 2019.
[23] I. Kolchinsky, I. Sharfman, and A. Schuster. Lazy evaluation methods for detecting complex events. In *DEBS*, pages 34–45, 2015.
[24] S. Krishnamurthy, C. Wu, and M. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, pages 623–634, 2006.
[25] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: Efficient evaluation of sliding window aggregates over data streams. In *SIGMOD*, pages 39–44, 2005.
[26] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier. Out-of-order processing: A new architecture for high-performance stream systems. In *VLDB*, pages 274–288, 2008.
[27] M. Liu, M. Li, D. Golovnya, E. A. Rundensteiner, and K. T. Claypool. Sequence pattern query processing over out-of-order event streams. In *ICDE*, pages 784–795, 2009.
[28] M. Liu, E. Rundensteiner, K. Greenfield, C. Gupta, S. Wang, I. Ari, and A. Mehta. E-Cube: Multi-dimensional event sequence analysis using hierarchical pattern query sharing. In *SIGMOD*, pages 889–900, 2011.
[29] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, page 49–60, 2002.
[30] H. Mai, B. Liu, and N. Cherukuri. Introducing AthenaX, Uber engineering's open source streaming analytics platform. https://eng.uber.com/athenax/, 2017.
[31] Y. Mei and S. Madden. ZStream: A cost-based query processor for adaptively detecting composite events. In *SIGMOD*, pages 193–206, 2009.
[32] O. Poppe, C. Lei, S. Ahmed, and E. Rundensteiner. Complete event trend detection in high-rate streams. In *SIGMOD*, pages 109–124, 2017.
[33] O. Poppe, C. Lei, L. Ma, A. Rozet, and E. A. Rundensteiner. To share, or not to share online event trend aggregation over bursty event streams. https://arxiv.org/abs/2101.00361, 2021. Technical report.
[34] O. Poppe, C. Lei, E. A. Rundensteiner, and D. Maier. Greta: Graph-based real-time event trend aggregation. In *VLDB*, pages 80–92, 2017.
[35] O. Poppe, C. Lei, E. A. Rundensteiner, and D. Maier. Event trend aggregation under rich event matching semantics. In *SIGMOD*, pages 555–572, 2019.
[36] O. Poppe, A. Rozet, C. Lei, E. A. Rundensteiner, and D. Maier. Sharon: Shared online event sequence aggregation. In *ICDE*, pages 737–748, 2018.
[37] Y. Qi, L. Cao, M. Ray, and E. A. Rundensteiner. Complex event analytics: Online aggregation of stream sequence patterns. In *SIGMOD*, pages 229–240, 2014.
[38] R. Ramakrishnan, B. Sridharan, J. R. Douceur, P. Kasturi, B. Krishnamachari-Sampath, K. Krishnamoorthy, P. Li, M. Manu, S. Michaylov, R. Ramos, N. Sharman, Z. Xu, Y. Barakat, C. Douglas, R. Draves, S. S. Naidu, S. Shastry, A. Sikaria, S. Sun, and R. Venkatesan. Azure Data Lake Store: A hyperscale distributed file service for big data analytics. In *SIGMOD*, page 51–63, 2017.
[39] M. Ray, C. Lei, and E. A. Rundensteiner. Scalable pattern sharing on event streams. In *SIGMOD*, pages 495–510, 2016.
[40] A. Rozet, O. Poppe, C. Lei, and E. A. Rundensteiner. Muse: Multi-query event trend aggregation. In *CIKM*, page 2193–2196, 2020.
[41] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.
[42] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *PODS*, pages 263–274, 2004.
[43] G. Theodorakis, A. Koliousis, P. Pietzuch, and H. Pirk. LightSaber: Efficient window aggregation on multi-core processors. In *SIGMOD*, page 2505–2521, 2020.
[44] C. Wu, A. Jindal, S. Amizadeh, H. Patel, W. Le, S. Qiao, and S. Rao. Towards a learning optimizer for shared clouds. *PVLDB*, 12(3):210–222, 2018.
[45] E. Wu, Y. Diao, and S. Rizvi. High-performance Complex Event Processing over streams. In *SIGMOD*, pages 407–418, 2006.
[46] H. Zhang, Y. Diao, and N. Immerman. On complexity and optimization of expensive queries in CEP. In *SIGMOD*, pages 217–228, 2014.
[47] R. Zhang, N. Koudas, B. C. Ooi, D. Srivastava, and P. Zhou. Streaming multiple aggregations using phantoms. In *VLDB*, pages 557–583, 2010.
[48] S. Zhang, H. T. Vo, D. Dahlmeier, and B. He. Multi-query optimization for complex event processing in SAP ESP. In *ICDE*, pages 1213–1224, 2017.
[49] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic plan migration for continuous queries over data streams. In *SIGMOD*, pages 431–442, 2004.