Complex event analytics: Online aggregation of stream sequence patterns

Article · June 2014

DOI: 10.1145/2588555.2593684

Yingmei Qi

2 PUBLICATIONS **42** CITATIONS

SEE PROFILE

Lei Cao

Massachusetts Institute of Technology

32 PUBLICATIONS **298** CITATIONS

SEE PROFILE

Elke Rundensteiner

Worcester Polytechnic Institute

655 PUBLICATIONS **9,443** CITATIONS

SEE PROFILE

Complex Event Analytics: Online Aggregation of Stream Sequence Patterns [*]

Yingmei Qi[†], Lei Cao[‡], Medhabi Ray[‡], Elke A. Rundensteiner[‡]
[†]Google Inc Seattle, WA 98103, USA
[‡]Worcester Polytechnic Institute Worcester, MA 01609, USA
yingmei.qi@gmail.com, lcao|medhabi|rundenst@cs.wpi.edu

ABSTRACT

Complex Event Processing (CEP) is a technology of choice for high performance analytics in time-critical decision-making applications. Yet while effective technologies for complex pattern detection on continuous event streams have been developed, the problem of scalable online aggregation of such patterns has been overlooked. Instead, aggregation is typically applied as a post processing step after CEP pattern detection, leading to an extremely ineffective solution. In this paper, we demonstrate that CEP aggregation can be pushed into the sequence construction process. Based on this insight our *A-Seq* strategy successfully aggregates sequence pattern online without ever constructing sequence matches. This drives down the complexity of the CEP aggregation problem from *polynomial to linear*. We further extend our *A-Seq* strategy to support the shared processing of concurrent CEP aggregation queries. The *A-Seq* solution is shown to achieve over four orders of magnitude performance improvement for a wide range of tested scenarios compared to the state-of-the-art solution.

Categories and Subject Descriptors

H.2 [**Information Systems**]: Database Management

Keywords

Complex Event Processing, Aggregation, Sequence Pattern

1. INTRODUCTION

Motivation. High performance **C**omplex **E**vent **P**rocessing (CEP), which detects complex sequence patterns over high speed event streams, forms the core of many modern business processes [9, 11, 19]. Current research in CEP has focused primarily on how to efficiently detect time-order aware sequence patterns [3, 11, 19]. Yet the optimization of CEP aggregation, critical for high performance analytics, has been overlooked. CEP aggregation asks analytics queries about matched sequence patterns, such as how frequently a certain sequence pattern occurs over some period of time or what the maximum value of particular event attributes in a pattern is among all matched sequences. Such CEP aggregation queries are prevalent in every aspect of both our digital daily lives and the business world alike, as motivated below.

Application I: Network Security. Consider a network security system designed to prevent illegal web-access.

```
PATTERN <SEQ(TypeUsername,TypePassword,ClickSubmit)>
WHERE <TypePassword.value!=TypeUsername.Password>
GROUP BY <IP>
[AGG COUNT] [WITHIN 10s]
```

The query above counts the web click sequence pattern corresponding to a wrong password immediately being submitted after entering a user name from the same IP. A brute-force attack would cause this count to rise abnormally. Then an appropriate action should be taken to block the IP from further attacking the system. It is necessary to respond to such situations in near real-time to protect user accounts from being compromised. In this network security system the *count of the matches* must be rapidly determined.

Application II: E-Commerce. Consider an online shopping website that monitors user click patterns for optimally timing targeted promotions, product recommendations, and customized web display layout. For instance, an analyst, hypothesizing user behavior in online shopping, might be interested in knowing the number of users who buy a Kindle followed by a Kindle Case and then a Stylus within 1 hour. Such a query could be expressed as sequence aggregation over web click stream with a 1 hour window.

```
PATTERN <SEQ(Kindle, KindleCase, Stylus)>
WHERE <Kindle.userId=KindleCase.userId=Stylus.userId>
[AGG COUNT] [WITHIN 1hour]
```

Given this insight, a prudent market strategy may be to display Kindles and cases on the same web page when customers search for either product, recommending cases when customers view Kindle, as well as bundling a Kindle and a matching case together as a package with a promotion price.

Application III: Fraud Detection. In a credit card protection system, a susceptible credit card fraud might be defined as a particular online purchase pattern repeatedly arising with respect to the same credit card with the total value over $10,000 within a short time, such as a 10 minute-window. Once the occurrences of this pattern surpasses a given threshold, an alert must be issued immediately to block the suspicious transactions.

In spite of CEP aggregation being a core feature of analytics systems, its optimization has been overlooked. For such aggregation queries on high-volume event streams, a timely response is of paramount importance. Even a one-second delay may lead to a loss of huge funds, investment opportunities, or even human life. We thus target the design of a high-performance CEP aggregation solution in this work.

[*]This work was done when Yingmei Qi was a master student at Worcester Polytechnic Institute. NSF grants IIS-1018443 & 0917017 support this project.

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.
Copyright 2014 ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2593684.

State-of-the-Art. To the best of our knowledge no research work to date has focused on this CEP aggregation problem. In current systems CEP aggregation queries are either not considered [3] or are supported as a by-product of *sequence detection queries* [11, 19]. That is, they first employ a *sequence detection* algorithm to find event instances that match the CEP pattern query to construct all sequence matches. Thereafter the aggregation function, such as COUNT, is applied to count the number of matched sequences. As analyzed in Sec. 2 the CPU complexity of *sequence detection* is *polynomial* in the number of active events − thus not scalable for high volume event streams. Worst yet, it is *exponential* in the length of the sequence pattern to be matched, therefore scaling poorly to *complex* sequence pattern queries. Furthermore sequence detection also requires the system to store all active events. In a system which receives millions of events per second, such an approach poses a significant strain on memory resource. Therefore, a highly customized lightweight approach must be developed to overcome this critical performance bottleneck.

Unlike CEP aggregation, traditional aggregation over sliding window streams has been studied [7, 8]. The key innovation is that upon the arrival of each individual data point its contribution to the aggregation result can be instantly pre-computed for each future window. Hence newly arriving events can be safely discarded once processed. However unlike traditional stream aggregation which aggregates *independent* data points, in CEP aggregation the contribution of the new arrival to future windows depends on whether in the future it could form sequence matches with any later arrival. Thus it is not predictable. Therefore the techniques in [7, 8] cannot be applied to solve our CEP aggregation problem.

Technical Challenges. The design of a lightweight CEP aggregation algorithm is challenging. Given that sequence construction mechanism is the bottleneck in processing sequence aggregation queries [11, 19], if we were able to eliminate the sequence construction step while computing CEP aggregation, a major benefit would be reaped. However, any technique capable of correctly supporting CEP aggregation must also be *sequence-match aware*. Thus we need to design a preemptive algorithm which meets the contradicting design goals of not only *instantly aggregating and pruning* each event upon arrival, but also *not missing any potential sequence match* formed in the future under the premise that no event is preserved.

In addition, a lightweight CEP aggregation solution must be able to effectively handle the *event expiration problem.* Namely the aggregation result needs to be continuously updated upon the expiration of events from the current window. However one event expiration might cause an arbitrary number of sequence matches to be come invalid. Worst yet, independently expiring each event would lead to erroneous aggregation results due to the correlation among the events in each sequence match. Purging expired events is thus difficult if the sequence matches themselves are not maintained, while explicitly maintaining all sequence matches conflicts with the objective of eliminating the expensive sequence construction process.

Furthermore CEP queries require *support for the negation (!) operator*. In a sequence pattern *negation* requires the *non-occurrence* of instances of the negated event types at certain positions within the matched sequence. In contrast to positive events, the arrival of events of a negated type instead could *invalidate* some potential sequence matches. Thus one integrated strategy must be designed to effectively process both positive and negative events.

Lastly, CEP applications often experience huge workloads of similar but not identical queries over shared event streams [9]. The design of a solution that *optimizes the shared computation* of aggregation queries is vital for scalability. Intuitively common sub-patterns, if extracted and aggregated independently, could be shared by multiple queries. However not all sub-pattern matches would form valid final sequence matches due to time order constraints of pattern queries. How we can leverage the computation sharing over arbitrary sub-patterns common within multi-queries, while still assuring correctness remains an open technical problem.

Proposed Solution. In this work we propose the *A-Seq* strategy that successfully tackles all the above challenges. A-Seq completely eliminates the construction and maintenance of any actual sequence match. It is therefore fundamentally more lightweight compared to state-of-the-art approach [19]. In fact we prove that CEP aggregation can be correctly handled by dynamically maintaining a compact *Prefix Counter* structure. We also design a novel algorithm for updating the *Prefix Counter* which instantly processes and discards each new event upon its arrival. It is proven to be optimal in both CPU and memory costs.

Furthermore A-Seq elegantly handles the *problem of event expiration* while supporting sliding windows by first locating the minimum subset of events whose expiration could indeed affect the aggregation result. A-Seq, by pre-isolating the influence of this event set, eliminates the need for purging expired sequence matches.

To *support negation (!)* in sequence aggregation, we establish the novel *Prefix Invalidation property* which successfully bounds the influence of negation to the update of the *Prefix Counter* structure. It empowers A-Seq to correctly process queries with negation in *constant time*. Overall the complete A-Seq approach is shown to successfully drive down the CEP aggregation costs from *polynomial to linear* complexity.

Finally we also explore the problem of *sharing computations among multiple CEP aggregation queries*. First, since A-Seq dynamically maintains the aggregations of prefix patterns, this computation can be naturally shared among queries with common prefixes. We further extend A-Seq to share the computation on common sub-patterns at arbitrary locations within the patterns. This approach assures the correctness of the aggregation queries with only a compact *aggregation snapshot* structure introduced.

Contributions. Contributions of our work include:

1) We design the first solution to effectively aggregate sequence patterns called *A-Seq*. *A-Seq*, which is proven to correctly push the aggregation computation into the sequence detection process, features linear time complexity (Sec. 3).

2) *A-Seq* effectively handles a variety of core CEP features, including negation, predicates, and GROUP BY (Sec. 3).

3) Lightweight algorithm is devised to share computation of common sub-patterns at arbitrary positions among queries in a given workload (Sec. 4).

4) *A-Seq* effectively supports popular aggregation functions like COUNT, MAX/MIN, and SUM/AVG (Sec. 5).

5) Our extensive experiments on real world event streams illustrate that our proposed *A-Seq* approach demonstrates over four orders of magnitude efficiency improvement for a wide range of tested scenarios over the state-of-the-art approach (Sec. 6).

2. PRELIMINARIES

2.1 CEP Aggregation Query

Basic Terminology. An *event instance* is an occurrence of interest denoted by a lower-case letter (e.g.,'e'). The time of occurrence of an event e_i is denoted by ts or simply by the subscript i for compactness. An *event type* **E** of an instance e_i denoted by $e_i.type$ describes the essential features associated with e_i.

Query Specification. We use a CEP query language commonly used in the literature [9, 19] and extend it to support aggregation over pattern queries.

```
PATTERN <event pattern>
[WHERE <qualification>]
[GROUP By <attribute>]
[AGG <aggregation function>]
[WITHIN <window>]
```

The PATTERN clause is composed of a CEP operator that defines an event pattern to be matched against an event stream. The WHERE clause contains predicates on attributes of the events, such as the E-Reader's model. The GROUP BY clause allows the user to partition sequence aggregation results by a certain attribute. The AGG clause specifies an aggregation function such as COUNT the number or SUM an attribute value over the set of sequence matches. The WITHIN clause stipulates that the time difference between the first to the last event instances matched by a pattern query falls within the window constraint.

CEP Operators. Several complex event operators have been studied in the literature [3, 11, 19]. In this work we focus on the sequential pattern queries denoted by the SEQ operator and the negation "!" operator which form a core feature of most event processing systems [3, 19]. A *SEQ operator* imposes an order on the instances of specific event types in terms of their timestamps to be considered a valid match.

$$\begin{aligned} SEQ(E_1, E_2, ..., E_n) = \{< e_1, e_2, ..., e_n > | e_1.ts < e_2.ts < \\ ... < e_n.ts \wedge (e_1.type = E_1) \wedge (e_2.type \\ = E_2) \wedge ... \wedge (e_n.type = E_n)\}. \end{aligned} \tag{1}$$

A *! operator* before an event type E_i indicates that the instance of type E_i is not allowed to appear in the stream *between* the matched events for the event types specified in the "SEQ" operator [19]. We call such E_i a *negative* event type. Consequently we call an event type E_i used in a SEQ construct without "!" a *positive* event type.

$$\begin{aligned} SEQ(E_1, !E_i, E_n) = \{< e_1, e_n > | (e_1.ts < e_n.ts) \\ \wedge (e_1.type = E_1) \wedge (e_n.type = E_n) \\ \wedge (\neg \exists e_i \; where (e_i.type = E_i) \wedge (e_1.ts < e_i.ts \\ < e_n.ts))\}. \end{aligned} \tag{2}$$

Sliding Window Constraint. Often continuous queries over streams are computed over a finite subset of events extracted from the stream by the "sliding window" constraint. The sliding window size is a parameter specified in the query to denote that the user is interested in the aggregated results over the most recent set of events. Here we adopt the window semantics most commonly used for CEP pattern detection queries [3, 9, 19]. That is, the query window has a fixed window size $Q._{win}$. It slides from W_i to W_{i+1} whenever a new event instance arrives. Query results are output whenever the aggregation result changes as the window slides.

2.2 Stack-Based Pattern Evaluation

State-of-the-art CEP engines employ an NFA based technique [19, 3] to process CEP sequence queries. Each event type specified in the sequence operator SEQ(E_1 ,..., E_n) is associated with a stack. Each new event instance of E_i is appended to the end of its corresponding stack. Each event of E_i is augmented with a pointer ptr_i to its adjacent event in stack E_{i-1}. When an event instance e_n of the last event type E_n arrives, the sequences are constructed by employing a depth-first search along instance pointers ptr_i rooted at e_n. Once sequences have been computed, the aggregation operator is applied. The following example evaluates a CEP aggregation with a NFA approach proposed in [19].

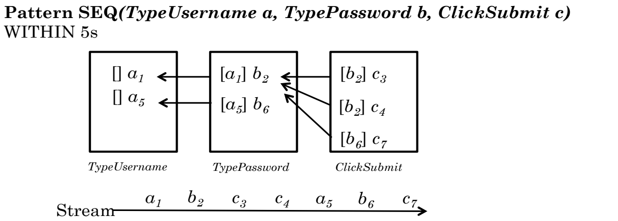


Figure 1: Stack Based Pattern Evaluation

EXAMPLE 1. *Fig. 1 depicts a sequence aggregation query and the event stacks. When a new event arrives it is appended to the end of its respective stack. For example when* c_3 *arrives it is inserted into the ClickSubmit stack. The CEP engine first checks if any event should be expired from the current window. Then a depth-first search is triggered to construct the sequence matches. Sequence* $<a_1, b_2, c_3>$ *is formed and the count is updated to 1. When* c_4 *arrives, a new sequence, namely* [$<a_1, b_2, c_4>$] *is formed. The count is now 2. When* b_6 *arrives,* a_1 *is purged out of the window. No valid sequence survives. Thus the count is updated to zero.*

Term	Definition
C_q	The cost of computing results for a query q
Pt_{E_i,E_j}	Selectivity of the implicit time predicate of sub-sequence (E_i, E_j).
$\|E_i\|$	Number of instances of type E_i in a time window

Table 1: Terminology Used in Cost Estimation

Cost Model for Stack-Based Execution. For an event pattern query q = SEQ(E_1, E_2 ,..., E_i ,..., E_n), E_i is an event type for 1 < i < n. Using stack-based pattern evaluation, the computation costs of C_q are formulated in Eq. 3 with the terms explained in Table 1.

$$C_q = \sum_{i=0}^{n-1} |E_{i+1}| * [\prod_{j=0}^{i} |E_j| * Pt_{E_j,E_{j+1}}] \tag{3}$$

For ease of comparison, given a sequence pattern with 'n' event types, assume every event type E_i receives an equal number of event instances $|E_i|$, then Eq. 3 is reduced to $|E_i|^n$. In other words the computation costs of the stack-based approach grow exponentially with the length of the sequence pattern and polynomially with the number of the event instances. Clearly this technology thus cannot provide real time responsiveness for CEP aggregation queries evaluated over high volume event streams.

3. THE A-SEQ SOLUTION

We now introduce our proposed *A-Seq* methodology that computes sequence aggregation on-the-fly. Here we first focus on the COUNT aggregation, being one of the most popular aggregation operations in the context of sequence analysis [10, 14, 17]. The other aggregation types, like SUM, AVG, MAX and MIN, can be similarly supported by *A-Seq* as described in Sec. 5.

3.1 A-seq Approach: Dynamic Prefix Counting

The key idea of A-seq is to push the aggregation computation into the sequence detection process, called the *Dynamic Prefix Counting (DPC)* method. Given a pattern $p = (E_1, E_2, ..., E_n)$, the pattern $p_m = (E_1, E_2, ..., E_m)$ $1 \leq m < n$ is called a *prefix pattern* of *p* of length 'm'. *DPC* exploits the key insight that sequence count aggregation can be handled as a counting operation rather than a sequence pattern construction operation. In other words to answer the sequence aggregation query q with pattern p, it is indeed not necessary to construct any complete or even partial sequence. Instead q can be solved by progressively counting the prefix patterns of p.

To better understand how *DPC* works, let us start by examining the sequence match formation process using the concrete example of pattern (A, B, C) in Fig. 2. At time t_i, one match (a_1, b_1, c_1) has been found, while a_2 is waiting for further event instances that can participate in the formation of future matches.

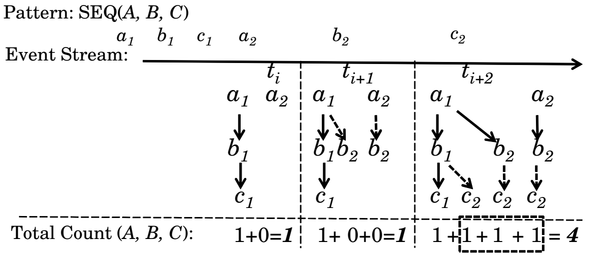


Figure 2: Sequence Forming Process for (A,B,C)

When b_2 arrives at time t_{i+1}, two new partial sub-sequences (a_1, b_2) and (a_2, b_2) are formed together with a_1 and a_2. When c_2, the instance of the last event type to form sequence (A, B, C), arrives at time t_{i+2}, we append it to subsequences (a_1, b_1), (a_1, b_2) and (a_2, b_2) to form 3 new (A, B, C) sequences. Thus, the total count of sequences constructed that match pattern (A, B, C) at time t_{i+2} is 4, including the 3 newly formed sequences and the one we have found before. From this, we observe that when c_i arrives, we can obtain the count of pattern (A, B, C) by adding two counts, namely, 1) the count of sub-pattern (A, B), to which c_i can be appended to form new (A, B, C) matches, and 2) the count of the previously detected pattern (A, B, C). This observation can be generalized as follows.

LEMMA 1. *Given a sequence pattern* $p = (E_1, E_2, ..., E_n)$ *and its Longest Prefix Pattern* $p_{n-1} = (E_1, E_2, ..., E_{n-1})$, *when one event* e_i *of* $e_i.type = E_n$ *arrives at time* t_i, *the count of the sequence matches of p can be calculated by Eq. 4.*

$$count(E_1, E_2, ..., E_n)_{t_i} = count(E_1, E_2, ..., E_{n-1})_{t_{i-1}} + count(E_1, E_2, ..., E_n)_{t_{i-1}} \quad (4)$$

Proof. Lemma 1 can be justified by proving the following two arguments: (1) all sequence matches of p_{n-1} equally contribute to the formation of the full sequence match; (2) when e_i of $e_i.type = E_n$ arrives, no any other prior sequences can lead to the construction of p except the sequence matches of p_{n-1}.

The two arguments naturally follow from the definition of sequence pattern (Sec. 2.1). To match pattern p the occurrence of $e_i(e_i.type = E_n)$ must be later than the occurrence of any other event participating in the sequence. Since at t_{i-1} each p_{n-1} sequence match s is constructed by the events arriving earlier than e_i, a full sequence match of p will then be formed by appending e_i to s. Argument 1 is proven. On the contrary appending e_i to the matches of other prefix pattern $p_m (m < n - 1)$ would not lead to a construction of p at time t_i due to the missing of the event instances e_j ($e_j.type \in \{E_{m+1} ... E_{n-1}\}$). Argument 2 is proven. ■

Lemma 1 has two implications. First, to count the matches of a sequence pattern *p*, only the *counts* of the previously formed sequence matches of *p* and the matches of p_{n-1} are required rather than the actual matches. Therefore these actual sequence matches and even the raw events can be safely pruned once processed. Second, the count of the sequence matches at time t_i only relies on the state at the previous time point, namely the counts at time t_{i-1}. For example the count at t_{i+2} when c_2 arrives only depends on the sequence counts at t_{i+1} ($count(A, B, C)_{t_{i+1}} = 1$, $count(A, B)_{t_{i+1}} = 3$). Therefore the states (the counts) at the older time points ($< t_{i-1}$) can be safely discarded.

However in event stream environments, only keeping the count of the matches for p and p_{n-1} is not sufficient to guarantee that we always get the correct aggregation. As new events continuously arrive, a match of any prefix pattern of p potentially can evolve to a match of its longest prefix pattern if the missing events are acquired. A valid match to *p* will then be formed whenever an E_n instance arrives later. Next we introduce Lemma 2 to establish the *minimum* information necessary to guarantee the correctness of the sequence aggregation query processing.

LEMMA 2. *Given a sequence pattern* $p = (E_1, E_2, ..., E_n)$, *to correctly count the matches to p at each time* t_i, *it is sufficient and necessary to continuously maintain a* ***Prefix Counter (PreCntr)*** *set:* $PreCntr_{t_{i-1}} = \{prefixCnt(p_m)_{t_{i-1}} \mid 1 \leq m \leq n\}$, *where* $prefixCnt(p_m)_{t_{i-1}}$ *indicates the count of the matches for the prefix pattern* $p_m = (E_1, E_2, ..., E_m)$ *constructed at time* t_{i-1}.

Proof. Sufficiency: Once we acquire an event e_i with $e_i.type = E_m$ at time t_i, by Lemma 1 $prefixCnt(p_m)_{t_i}$ is calculated as: $prefixCnt(p_m)_{t_i} = prefixCnt(p_m)_{t_{i-1}} + prefixCnt(p_{m-1})_{t_{i-1}}$. Since $prefixCnt(p_m)_{t_{i-1}}, prefixCnt(p_{m-1})_{t_{i-1}} \in PreCntr_{t_{i-1}}$, any prefix sequence match ending at e_i that potentially leads to the construction of a match of the full pattern *p* will not be missed by maintaining $PreCntr_{t_{i-1}}$. The sufficiency of $PreCntr_{t_{i-1}}$ is proven.

Necessity: By contradiction. Assume $prefixCnt(p_m)_{t_{i-1}}$, which represents the count of sequence matches for $p_m = (E_1, E_2, ..., E_m)$, misses from the $PreCntr_{t_{i-1}}$ set. Assume it is *k*. If events e_i ($e_i.type = E_{m+1}$), e_{i+1} ($e_{i+1}.type = E_{m+2}$) ,..., $e_{i+n-m-1}$ ($e_{i+n-m-1}.type = E_n$) continuously arrive between time t_i to $t_{i+\delta}$, then k sequence matches for *p* will be formed at $t_{i+\delta}$ by appending these new arrivals to the sequence matches of p_m constructed previously at t_i. Since $prefixCnt(p_m)_{t_{i-1}}$ is not kept, the k matches for *p* will be missed at $t_{i+\delta}$. Therefore every element in $PreCntr_{t_{i-1}}$ must be saved. The necessity of $PreCntr_{t_{i-1}}$ is thus proven. ■

By Lemma 2 given a sequence pattern $p = (A, B, C, D)$, we can progressively compute its count from the singleton prefix (A), until we get the count for the full pattern (A,B,C,D). That is, to get the count of the whole pattern, we count all its prefix patterns incrementally and store their most recent counts. The algorithm called *Dynamic Prefix Counting (DPC)* is shown in Fig. 3.

Given a CEP aggregation query *q* with the sequence pattern specified as $p = (E_1, E_2, ..., E_n)$. DPC first creates a PreCntr structure representing the counts of the *n* prefix patterns of *p*. Then DPC classifies each new arrival into three classes, namely **Start Event Type (START)**, **Update Event Type (UPD)**, and **Trigger Event Type (TRIG)** as shown in Lines 3, 5, and 7. START indicates the first event type E_1 of *p*. When a *START* instance arrives, we simply increase the count of E_1 by 1 since there is no prefix pattern prior to it (Line 4). *UPD* instead represents all other event types except START in *p*. When a *UPD* instance arrives, the count of the prefix pattern it triggers (ending with this *UPD* type) will be updated according to Lemma 1 (Line 6). For example, the count of (A, B, C) should be updated when a C instance arrives. The last event type

```
Basic A-Seq: Dynamic Prefix Counting
(Input: pattern query $q$, stream $S$)

  $e_i$ : an event instance
  $e_{i.cat}$ : event category $e_i$ falls in (START, UPD, TRIG)
  $L_q$ : the number of event types in $q$

1. initialize a PreCntr of size $L_q$
2. for each arriving event $e_i$ in $S$
3.   if $e_{i.cat}$ = START
4.     count(START) ++
5.   if $e_{i.cat}$ = UPD
6.     apply Lemma 1
7.   if $e_{i.cat}$ = TRIG
8.     return the full sequence count
```

Figure 3: Dynamic Prefix Counting Algorithm

E_n in *p* is called TRIG. It not only performs the *UPD* update operation, but it also indicates the completion of the full pattern *p*. Thus, TRIG will trigger the delivery of the output of the just generated aggregation result to users (Line 8).

EXAMPLE 2. *Fig. 4 depicts the prefix count update process based on the DPC algorithm. The number at the lower right corner of each circle represents the count of this prefix pattern at the respective moment in time indicated at the top of each column. When event instance b arrives at time t_{i+1}, new matches of the prefix pattern that end in B will be triggered, namely, (A, B). For this, we simply add the existing counts of (A) = 3 and (A, B) = 2 to get the new count of (A, B) = 5. The counts of all other prefix patterns remain unchanged. Similarly, when the instance d arrives, the same update process is applied to compute the new count of its corresponding pattern (A, B, C, D). Since d is a TRIG, this new count will be returned to the user.*

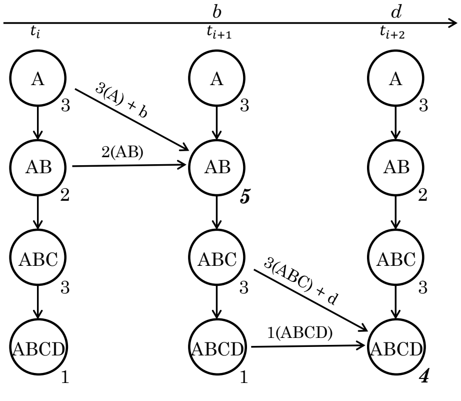


Figure 4: Prefix Pattern Count for Pattern (A, B, C, D)

Complexity Analysis. DPC is optimal in both CPU and memory utilization. Each incoming event is processed only once with only one element of the *PreCntr* updated. Clearly this is optimal, since without touching each event at least once it is impossible to correctly answer any counting problem. In terms of memory DPC does not store any event. Instead events are immediately discarded once processed. At any time instance *PreCntr* is the only data structure that is stored, which by Lemma 2 is the minimum information required to guarantee the correctness of sequence counting.

3.2 Expiration Support for A-Seq

Sliding Window Problem. *DPC* is able to aggregate the sequence patterns formed by continuously arriving events with minimum CPU and memory costs. However, this technique does not support sliding window clause. That is, as new events continuously arrive, old events continuously expire. Every event would eventually expire and no longer participate in future CEP sequence construction. Thus, they should be purged. However purging only the expired events is not sufficient for CEP sequence aggregation. After an event expires, any sequence match that contains the expired event would also become invalid and should be purged from the aggregation result.

Intuitively we may be able to support the removal of expired events by maintaining a count for each event instance e_i, which indicates how many sequence matches e_i participates in. Thus when e_i expires, this count is deducted from the aggregation result.

However only maintaining the count for each e_i is not sufficient to correctly handle event expiration, although it adds significant memory overheads to *A-Seq*. Each sequence match is composed of *l* individual events with *l* indicating the length of the sequence pattern. If we independently expire each event by removing all the sequences involving it (the count), one sequence will be expired *l* times. To solve this problem we have to be aware of other participants in each sequence. This in fact equals to keeping each actual sequence match. This requirement clearly invalidates the core principle of DPC, namely answering the aggregation query without constructing actual sequence matches.

Start Event Marking (SEM) Solution. The key insight is that the expiration of most of the event instances makes no impact on the sequence aggregation query result. The only case when we need to purge the sequence count is when the *START* events expires.

LEMMA 3. *Given a sequence pattern query $q = (E_1, E_2, ..., E_n)$ and an event instance e_i, $e_i.type \neq E_1$, when e_i expires at time t_j, then before processing the new event instance arriving at t_j: $count(q)_{t_{j-1}} = count(q)_{t_j}$*

Proof. Suppose e_i participates in at least one sequence match s of q. By the definition of sequence pattern, in each s at least one START event instance e_k, $e_k.type = E_1$ must exist which had arrived before e_i. Otherwise s would not be a valid sequence match of q. In event streams earlier arrivals always expire before later arrivals. Therefore when e_i expires at t_j, all matches that includes e_i would already have become invalid due to the expiration of some prior e_k within its s. Hence the expiration of e_i will not impact the sequence count at t_j. This proves Lemma 3. ■

We now leverage Lemma 3 to design a **Start Event Marking (SEM)** technique which pre-isolates the influence of each *START* instance upon its arrival. It successfully avoids the expensive process of having to eliminate the impact of instances once they expire from the current window.

The SEM algorithm (Fig. 5) is composed of the following steps:

(1) When a *START* instance e_i arrives, a *PreCntr* is created for e_i (denoted as $PreCntr(e_i)$) to record the number of sequence matches formed on it (Lines 3-6).

(2) When a *UPD* instance arrives, the same update process as in basic *A-Seq* approach (Fig. 3) is applied. However, this update is applied to the *PreCntr* of all active *START* instances (Lines 7-11).

(3) When a *TRIG* instance arrives, counts on all active *PreCntr* are summed together and output as aggregation result. Expired *PreCntr* (if any) is simply removed (Lines 12-18).

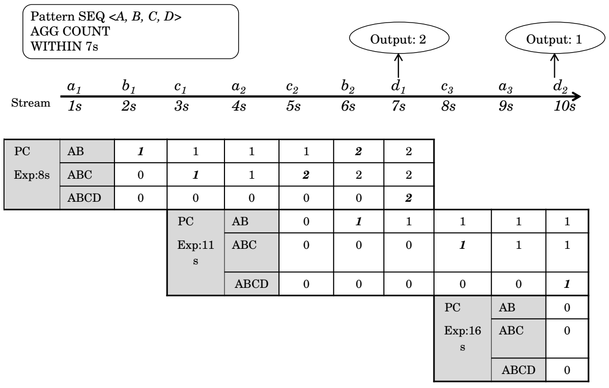
```
A-Seq SEM
$e_{i.ts}$: arrival timestamp of $e_i$
$PreCntr._{exp}$: expiration timestamp of a $PreCntr$ (a START instance $e_i$)
$ts$ : current system timestamp
$Win$ : query window size (time based)

agg = 0
for each arriving event  in stream S
//step 1: Create PreCntr
    if $e_{i.cat}$ = START
        create a PreCntr of size $L_q$-1
        mark $PreCntr._{exp}$ = $e_{t.ts}$ + Win
//step 2: Update Count
    else
        for each PreCntr
            if $PreCntr._{exp}$< ts
                apply Lemma 1
//step 3: Sum aggregation result
    if $e_{i.cat}$ = TRIG
        for each PreCntr
            if $PreCntr._{exp}$< ts
                agg = agg + PreCntr.count
            else
                remove this expired PreCntr
return agg
```

Figure 5: A-Seq SEM Algorithm

(4) When the window slides, any expired *START* event with its corresponding *PreCntr* is removed. If an output result were to be required at this time, then the count on this *PreCntr* will be simply subtracted from the total count result.

Next we use an example to illustrate the SEM counting process.



PC	AB	***1***	1	1	1	***2***	2			
Exp:8s	ABC	0	***1***	1	***2***	2	2			
	ABCD	0	0	0	0	0	***2***			
			PC	AB	0	***1***	1	1	1	1
			Exp:11 s	ABC	0	0	0	***1***	1	1
				ABCD	0	0	0	0	0	***1***
								PC	AB	0
								Exp:16 s	ABC	0
									ABCD	0

Figure 6: Pushing Windows Down into A-Seq Using SEM

EXAMPLE 3. *In Fig. 6, when a_1 arrives at time $t = 1s$, we first calculate its expiration timestamp (Exp), where $Exp = arrTime + Q.win$. Thus, $a_1.Exp = 1s + 7s = 8s$. That is, a_1 and its aggregation result will become invalid at $t = 8s$. Then we create a PreCntr for a_1. We remove the element for prefix A from the PreCntr set. Now each instance of A has its own PreCntr structure. Thus, the count for prefix A will always be 1, hence not necessary to explicitly maintain it.*

The update process of the basic A-Seq method is applied when b_1 and c_1 arrive. When a_2 arrives at $t = 4s$, we now have to create a new PreCntr for a_2 and mark it with its expiration timestamp. Subsequently, when c_2 and b_2 arrive, the PreCntrs on both a_1 and a_2 are updated.

When d_1 arrive at $t = 7s$, first we update the count on both PreCntrs. Since the instance of the TRIG event type D completes the full sequence, we compute the final aggregation result by summing the counts of pattern (A, B, C, D) from all active PreCntrs. Thus, the output result is $2 = 2(a_1 PreCntr) + 0(a_2 PreCntr)$.

Next, c_3 arrives at time $t = 8s$. When we perform the update on each PreCntr, we find that a_1 expires at $8s$ and a_1's PreCntr should be purged. If users require a result at this moment, the output would be 0 instead of 2. The same steps as above are applied when a_3 and d_2 arrive. The output result becomes 1 after d_2 is processed.

Next we are ready to prove the correctness of SEM.

LEMMA 4. *Given a sequence pattern query $q = (E_1, E_2, ..., E_n)$, then at any time t_i:*

$$count(q)_{t_i} = \sum_{e_1^j.type=E_1, e_1^j.exp>t_i} PreCntr(e_1^j).count(q)_{t_i} \tag{5}$$

Proof. By the definition of sequence pattern, we have

$$count(q)_{t_i} = \sum_{e_1^j.type=E_1, e_1^j.exp>t_i} count(e_1^j, E_2, ..., E_n)_{t_i} \tag{6}$$

where $count(e_1^j, E_2, ..., E_n)_{t_i}$ represents the number of sequence matches starting at START event instance e_1^j.

By Lemma 2, $count(e_1^j, E_2, ..., E_n)_{t_i} = PreCntr(e_1^j).count(q)_{t_i}$. Therefore Eq. 5 holds. Lemma 4 is thus proven. ■

Complexity Analysis. As shown in Fig. 5 the CPU costs of SEM originate from the update operation on each PreCntr triggered by each new arrival. This update costs depend on the number of active start events *k*. Since the update of each PreCntr takes constant time, the time complexity of SEM is thus *k*. Similarly the memory cost of SEM is $k \times sizeof(PreCntr) = kl$ with l indicating the length of the sequence pattern. That is, the memory complexity of SEM is linear in *k*. In short, SEM successfully drives down the sequence aggregation cost from *polynomial* to *linear*.

3.3 Negation Support for A-Seq

The Invalid Sequence Check Problem. *Negation* in CEP queries requires us to assert the non-occurrence of instances of the negated event types at certain positions in a sequence pattern. For example, to plan an effective web advertising strategy online retailers might be interested in pattern $p_n = (VK, BK, !REC, VC, BC)$. This pattern tracks how many customers purchase a case after the purchase of a kindle, yet without first clicking through the "Recommendation" link. When an event instance of a negated type occurs, it will *invalidate* some otherwise potential sequence matches. While the occurrence of the positive event instances instead would advance the formation of longer sequence matches. The design of one single algorithm that accommodates these two contradictory requirements (invalidation vs. advancement) is challenging.

One intuitive way from the literature [3, 19] to solve this problem would be to add a negation filter on top of the query plan to discard all positive matched sequences (vk_i, bk_i, vc_i, bc_i) that have rec_i between bk_i and vc_i. An obvious problem with this later-filter-step solution is that it generates a potentially huge number of intermediate results, many of which may be filtered out eventually. Worst yet it is incompatible with our core *A-Seq* approach as we avoid constructing any full or even partial sequence matches. We now propose a solution that addresses this problem by *pushing this negation check into A-Seq*.

The Immediate Re-Count Solution. We now illustrate that the *invalidation effect* of the negative events can be naturally handled in the *dynamic prefix counting* process. Negation can be supported by simply plugging a *recounting rule* in A-Seq. In other words to support negation, it is still not necessary to construct any actual sequence matches.

First we introduce the ***Prefix Invalidation Property*** which defines the boundary of the influence of negative events.

LEMMA 5. *Given a negation pattern query $q = (E_1, E_2, ..., !E_i, ..., E_n)$, when a negative event instance e_j with $e_j.type = E_i$ arrives, then all the prefix sequences of q remain valid except those Longest Positive Prefix Sequences LPPS with LPPS representing the sequences that match prefix pattern $(E_1, E_2, ..., E_{i-1})$.*

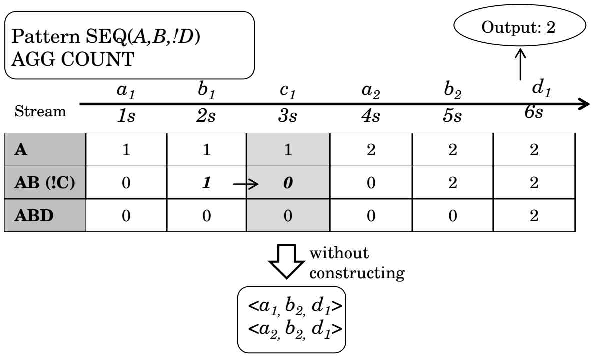
Proof. With the arrival of e_j even if an event instance e_k of E_{i+1} arrives later, LPPS cannot form any longer prefix pattern of q with e_k. Therefore LPPS is not valid any more. All other prefix matches remain effective. First the previously formed matches of the prefix patterns longer than LPPS are not affected. This is so because e_j arrives later than the last event of these matches, hence it does not satisfy the time order of q. Second, the sequences matching the prefix pattern shorter than LPPS are also not affected. Without loss of generality given such a pattern $p = (E_1, E_2, ..., E_{i-2})$, the sequences of p is still able to form valid matches of the longer prefix pattern $(E_1, E_2, ..., E_{i-1})$ as an new event of E_{i-1} arrives. ■

Since our *A-Seq* solution preserves the count of each prefix pattern, *A-Seq* can elegantly leverage Lemma 5 to support *negation* by applying the ***Recounting Rule (RR)***.

LEMMA 6. *Given a pattern query with negation $q = (E_1, E_2, ..., !E_i, ..., E_n)$, when a negative event instance e_j with $e_j.type = E_i$ arrives at time t_j, then:*

(1) $count(E_1, E_2, ..., E_{i-1})_{t_j} = 0$; (2) For other prefix pattern p of q, $count(p)_{t_i} = count(p)_{t_{i-1}}$.

That is, when a negative event instance arrives, we only need to reset the count of the previous prefix pattern adjacent to the negative event type to 0. This simple reset corresponds to an effective recounting. In other words A-Seq can support negation by applying the *RR* rule on each negative event without any further change.



Stream	a_1 1s	b_1 2s	c_1 3s	a_2 4s	b_2 5s	d_1 6s
A	1	1	1	2	2	2
AB (!C)	0	*1*	*0*	0	2	2
ABD	0	0	0	0	0	2

Figure 7: Pushing Negation Check Down

EXAMPLE 4. *Fig. 7 illustrates how RR works using pattern $(A, B, !C, D)$. When c_1 arrives at $t = 3s$, the count of (A, B) is cleared to 0 while the (A) and (A, B, D) counts are kept. The output result is 2 when d_1 arrives. The sequence $< a_1, b_1, d_1 >$ is not counted, as a C instance c_1 exists between b_1 and d_1.*

3.4 Predicates and GROUP BY for A-Seq

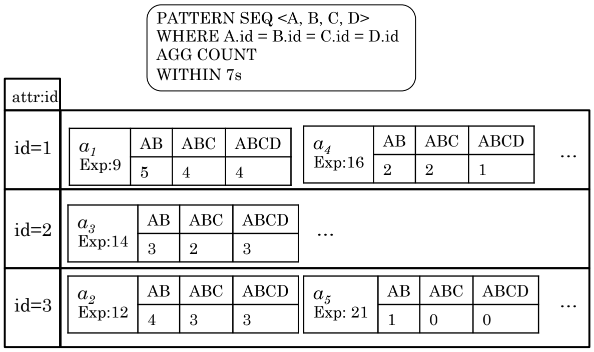
Below, we present solutions for pushing the predicate evaluation into our sequence aggregation process.

Local Predicates. The most common predicates over event data are local predicates, which impose constraints on the attribute values of an event instance (e.g. Kindle.model = "touch"). Such predicates can be evaluated on the relevant event instances before these instances are involved in the aggregation process. Event instances that do not satisfy the predicates are immediately discarded .

Equivalence Predicates. CEP queries often use equivalence predicates to correlate events in a sequence pattern [19]. For example, in our online shopping pattern tracking scenario, the clicks should be from the same customer. Similarly, in the stock market example, the pattern of price going up and down should be of the same stock. An equivalence test essentially partitions an event stream into several sub-streams, where events in the same partition have the same value for the attribute used in the equivalence test (i.e. equivalence attribute). Here, we propose a technique to dynamically partition the event stream during the sequence aggregation, henceforth called ***Hashed Prefix Counter (HPC)***.

The basic idea of *HPC* is that the aggregation process is applied separately to each equivalent partition. Prefix counters for a *START* instance are created upon their arrival and hashed into the corresponding partition based on the equivalence attribute value of this instance. Other event instances are similarly hashed to their corresponding partition. Aggregation results are computed based on each partition using our regular *A-Seq* method.

GROUP BY. Aggregation queries are often used in conjunction with GROUP BY clause. For instance, in Application I described in Sec. 1, the query counts the web click sequences coming from a certain IP address. The *HPC* technique naturally supports partitioning the aggregation by attribute values. Here instead of summing the aggregation from different *HPC* partitions, the results are output separately for each partition.



attr:id		AB	ABC	ABCD
id=1	a_1 Exp:9	5	4	4
	a_4 Exp:16	2	2	1
id=2	a_3 Exp:14	3	2	3
id=3	a_2 Exp:12	4	3	3
	a_5 Exp: 21	1	0	0

Figure 8: Hashed Prefix Counter (HPC) Data Strcture

EXAMPLE 5. *Fig. 8 illustrates the HPC data structure proposed to cope with the processing of equivalence predicates in the query in Fig. 8. The equivalence test is on attribute id. The id value of each incoming event will be evaluated. Assuming three distinct id values (1,2,3) exist, then three hash partitions are created as depicted in Fig. 8 with id value as key and the prefix counters as the value. For instance, the id values of instances a_1 and a_4 are 1. Thus their prefix counters are created in the ID_1 partition. The instances of type B, C, D will activate the update of the prefix counters in the partition determined by their respective id values.*

4. THE MULTI-QUERY A-SEQ SOLUTION

For rich data streams from web-clicks to stock ticker streams, large workloads of similar queries may be processed. Executing each query separately could lead to repeated computations causing scalability and performance issues. Thus, in this section we explore the opportunity to share the computation across multiple CEP aggregation queries [9].

EXAMPLE 6. *Consider the running example in Sec. 1, where merchants are interested in various purchase patterns to determine customer shopping habits, including:* [1]

```
Q1 = SEQ(VKindle,BKindle,VCase,BCase)
Q2 = SEQ(VKindle,BKindle,VKindleFire)
Q3 = SEQ(VKindle,BKindle,VCase,BCase,VeBook,BeBook)
Q4 = SEQ(VKindle,BKindle,VCase,BCase,VLight,BLight)
Q5 = SEQ(ViPad,VKindleFire,VKindle,BKindle)
```

Several common sub-queries (substrings) arise across these 5 pattern queries, such as $(VKindle, BKindle)$.

The intuition is that if we were to compute intermediate aggregates for the $(VKindle,\ BKindle)$ substring, then we would be able to share this result among the queries that contain this substring. This way, redundant aggregation computation for common substrings would be avoided. In principle, the more queries share common substrings, the more computational resources could potentially be saved. This promise of scalability leads us to adapting our A-Seq approach to support sharing-aware execution of multiple aggregation queries.

4.1 The Prefix Sharing Strategy

We first investigate how to achieve multiple aggregation query sharing among queries with *common prefixes*. The observation here is that once the aggregation of a full sequence pattern *p* is computed by *A-Seq*, then the aggregations of all its prefix patterns p_m would also have been obtained as side-effect. These aggregation results thus can be naturally reused by other queries.

Based on this observation, we first propose a ***Prefix Tree (PreTree)*** structure in support of the ***Prefix Sharing***.

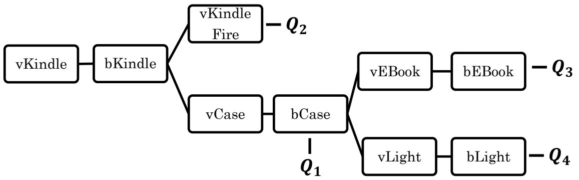


Figure 9: PreTree Structure

EXAMPLE 7. *Fig. 9 shows an example of the PreTree structure for* $Q_1 \sim Q_4$*, namely, these four prefix counters are organized into a PreTree structure. For computation of these four patterns, the result of substring* $(vKindle, bKindle)$ *is pipelined into both* Q_1 *and* Q_2*, while the* Q_1 *result is pipelined to both* Q_3 *and* Q_4.

Based on this *PreTree* structure A-Seq can easily be adapted for this common prefix patterns sharing as follows.

(1) When a START instance arrives, a ***PreTree*** structure is created instead of the old *PreCntr* structure.

(2) When a UDP instance arrives, the count update occurs on the corresponding location in each *PreTree* only once. For example, when a $bKindle$ instance arrives, one update on *PreTree* is needed in place of four updates on each of the four *PreCntrs*.

(3) When a TRIG instance arrives, results for the respective query can be directly acquired from the appropriate locations of *PreTree*.

Apparently no overhead is introduced in this process. That is, A-Seq effectively shares the computation on the common prefix patterns *for free*.

4.2 Sub-pattern Sharing Strategy

Overall Idea of Chop-Connect Approach. In general, queries may feature common sub-expressions at arbitrary locations rather than only at their prefix positions. For instance in Example 6 Q_5 shares $(VKindle,\ BKindle)$ with the other four query patterns at the tail. As shown in Sec. 4.1 we can obtain the count of $(VKindle,\ BKindle)$ from the prefix tree that has been set up for $Q_1 \sim Q_4$. Intuitively if we were to compute the remainder of Q_5 $(ViPad,\ VKindleFire)$ separately, then we could potentially acquire the count of the full pattern by connecting the counts of these two substrings together. This way, the computation of $(VKindle,\ BKindle)$ would be shared by all five queries. This leads us to propose the ***Chop-Connect (CC)*** method to tackle this general problem of common sub-expression computation. The idea is to ***chop*** a query into substrings, with each substring being computed separately. The results of all substrings are later ***connected*** together to get the count of the whole sequence pattern. Hence this approach is called *Chop-Connect*, or in short *CC*.

The Connect Problem. Compared to the prefix sharing of Sec. 4.1, an extra *connect* operation is introduced. However to get the count of the longer connected sequence $(sub_1\ +\ sub_2)$, we cannot simply construct a cartesian product of two previously computed sub-counts. Instead the time order between the matches of the two substrings sub_1 and sub_2 matters. That is, the sub_1 match must be formed before the starting time of the sub_2 match for the two sequences to construct a valid longer sequence match. Since the *A-Seq* approach does not maintain any time information (except START for expiration), additional ordering knowledge must be recorded to correctly connect two substrings. One straightforward solution is to maintain the time when each sub_1 match is constructed and compare it with the arrival time of the START event of sub_2 once the connect operation is required. However keeping the time information of each single sequence match contradicts the core of the A-Seq technology.

Count Attachment Observation. Fortunately our analysis reveals that it is not necessary to store the time information of each sub_1 match. Instead, it is sufficient to only attach a count of sub_1 to each START instance of sub_2 upon its arrival.

LEMMA 7. *Given a sequence pattern p chopped into two substrings* sub_1 *and* sub_2*, a START event* e_i *of* sub_2 *arriving at* t_i*, when a TRIG event* e_j *of* s_2 *arrives at* t_j*, then:*

$$count(p, e_i)_{t_j} = count(sub_1)_{t_i} * count(sub_2, e_i)_{t_j} \qquad (7)$$

where $count(p,\ e_i)_{t_j}$ *represents the count of full sequence matches of pattern p at* t_j *that contains the event* e_i *(a START event of* sub_2*),* $count(sub_1)_{t_i}$ *represents the count of sub-sequences* sub_1 *at* t_i*, and* $count(sub_2,\ e_i)_{t_j}$ *represents the count of sub-sequences* sub_2 *starting at* e_i *at time* t_j

Proof. This lemma can be proven by the fact that only the sub_1 sequence matches constructed before the arrival of e_i (t_i) can be connected to the sub_2 sequence matches starting at e_i to form valid full sequence matches, while those sub_1 sequence matches constructed after t_i cannot. ■

Lemma 7 indicates that keeping only the count of sub_1 matches formed before the arrival of each START event of sub_2 is sufficient to correctly connect two substrings. In other words the *connect* problem can be safely solved by attaching to each START event e_i of sub_2 a *count* upon its arrival, where the count indicates the total number of sub_1 matches detected before e_i arrives. We note that this sub_1 count would already have been calculated and stored in *PreCntr* by our *A-Seq* algorithm.

[1] Capital letter *V* represents *View*, and *B* represents *Buy*

Challenges Caused by Expiration. However, the above connect solution assumes that events never expire. Suppose that the $\#(sub_1)$ (count of sub_1 matches) is attached to a START instance of sub_2 as it arrives at time t_i. When a TRIG instance of sub_2 arrives at time t_j, this $\#(sub_1)$ might be invalid due to the expiration of some sub_1 matches. This causes erroneous aggregation.

SnapShot Solution. We now propose a solution to this connect expiration problem. The idea is to leverage the observation of Lemma 3 introduced in Sec. 3.2. Namely, when a START instance of sub_2 arrives, instead of attaching the current overall sub_1 count, we record the count of each $PreCntr$ on sub_1 corresponding to each start event of sub_1 separately along with their expiration timestamps. Therefore, when a TRIG instance arrives, the respective $PreCntr$ to be expired by its arrival can be discovered by checking the expiration time of each partial sub_1 count.

We then design the ***SnapShot*** data structure to represent the counts on all sub_1's $PreCntr$s. The snapshot is organized as a table. Each row represents the count snapshot of a sub_1's $PreCntr$. The columns contain metadata: 1) $PreCntr$ tag, indicating the *START* instance to which this $PreCntr$ belongs, 2) the expiration time of this $PreCntr$, and 3) the count of this $PreCntr$.

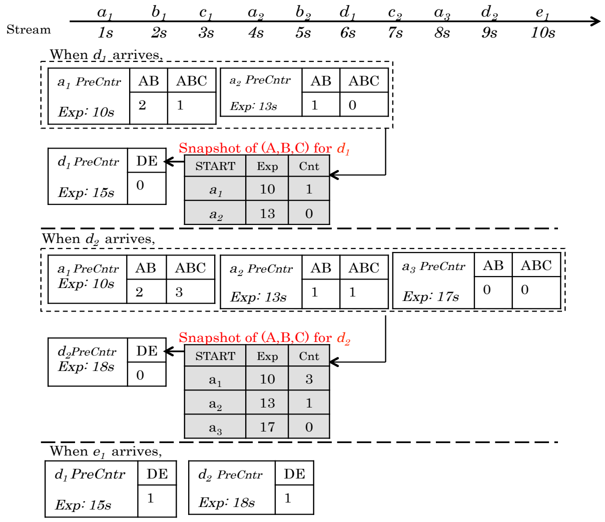

Stream
a_1 1s
b_1 2s
c_1 3s
a_2 4s
b_2 5s
d_1 6s
c_2 7s
a_3 8s
d_2 9s
e_1 10s
When d_1 arrives,
a_1 PreCntr Exp: 10s AB 2 ABC 1
a_2 PreCntr Exp: 13s AB 1 ABC 0
d_1 PreCntr Exp: 15s DE 0
Snapshot of (A,B,C) for d_1
START Exp Cnt
a_1 10 1
a_2 13 0
When d_2 arrives,
a_1 PreCntr Exp: 10s AB 2 ABC 3
a_2 PreCntr Exp: 13s AB 1 ABC 1
a_3 PreCntr Exp: 17s AB 0 ABC 0
d_2PreCntr Exp: 18s DE 0
Snapshot of (A,B,C) for d_2
START Exp Cnt
a_1 10 3
a_2 13 1
a_3 17 0
When e_1 arrives,
d_1 PreCntr Exp: 15s DE 1
d_2 PreCntr Exp: 18s DE 1

Figure 10: SnapShot Maintenance

EXAMPLE 8. *Fig. 10 illustrates the snapshot maintenance of* sub_1 (A, B, C) *for* sub_2 (D, E)*. In Fig. 10, when* d_1 *arrives, a snapshot of* d_1 *with counts and expiration timestamps of all* sub_1 *prefix counters is created, and attached to* d_1*. When* d_2 *arrives, similarly, another snapshot is created to record the prefix counter status of* sub_1 *at that moment and attached to* d_2*. When* e_1 *arrives at* $t = 10s$*, after the PreCntrs of* (D, E) *is updated, the snapshot expiration check is applied to each table row of* sub_2 $PreCntr$*. Expired rows are discarded, and thus won't be involved in any future aggregation. For example, when* e_1 *arrives at* $t = 10s$*, the snapshot check finds that* a_1 *expires at that time. Thus, only the counts of the PreCntrs of* a_2 *and* a_3 *will be used. The total count* $\#(A.B, C, D, E)$ *at* $t = 10s$ *thus is calculated as:*

$count_1 = \#(d_1\ PreCntr) = \#(A, B, C, d_1, E) = 1 \times 0 = 0;$
$count_2 = \#(d_2\ PreCntr) = \#(A, B, C, d_2, E) = 1 \times 0 + 1 \times 1 = 1;$
$\#(A, B, C, D, E) = count_1 + count_2 = 0 + 1 = 1$

We now introduce a new event category, called **Connect Event Type (CNET)**. *CNET* is the *START* event type in sub_2 when connecting sub_1 and sub_2. The arrival of a *CNET* instance will trigger the process of checking counts on sub_1's *PreCntr*s and creating the sub_1 snapshot for sub_2. For example, event type D falls into the category *CNET*, meaning when connecting (A, B, C) and (D, E), a snapshot of (A, B, C) count is created for (D, E) whenever a new d_i instance of type *D* arrives. By this the snapshot strategy can be seamlessly plugged into our *A-Seq* algorithm (Fig. 5).

Multi-Connect Process. Based on the sharing plan produced by a multi-query optimizer, a query might be chopped into multiple pieces rather than just only 2. For example, the pattern query (A, B, C, D, E, F, G) might be chopped into: $sub_1 = (A, B, C)$, $sub_2 = (D, E)$, and $sub_3 = (F, G)$, as sub_2 and sub_3 are shared by other queries respectively. In this case all three substrings have to be concatenated. Intuitively this can be achieved by recursively connecting adjacent substrings.

In support of expiration, when we create the snapshot for one substring, we always use the START event of the full sequence as the *tag*. Since the START event is always the first one to expire in a sequence, by this it is very convenient to check and exclude the expired sequences. For example, when we create the snapshot for $sub_3 = (F, G)$, we calculate the snapshot counts of (A, B, C, D, E) on each A instance, that is, counts of (a_i, B, C, D, E) for all active a_i. When F instance arrives, these counts can be calculated as follows:

(1) Calculate count of (a_i, B, C, D, E) on **each** (D, E) *PreCntr*. For each (D, E) *PreCntr*, to which the snapshot of (A, B, C) is attached, we multiply its (D,E) count with the count in its snapshot one by one. That is, we calculate the count for each (a_i, B, C, d_j, E).

(2) Calculate the count of (a_i, B, C, D, E) on **all** (D, E) *PreCntr*s. Given the counts calculated by step 1, we apply a sum over those counts with the same a_i tag across all (D, E) *PreCntr*s.

EXAMPLE 9. *Fig. 11 illustrates the multi-connect calculation process of the above example. When* f_1 *arrives at* $t = 12s$*, we first multiply the count on each* (D, E) *PreCntr with its corresponding snapshot counts of* (A, B, C)*. Then, we calculate the overall count of tag* a_2 *by adding the count* 1 *on* d_1*'s PreCntr to the count* 2 *on* d_2*'s PreCntr with the same tag* a_2*. Similarly, we sum up the counts with same tag across all* (D, E) *PreCntrs. Lastly we attach these counts to the* f_1 *PreCntr representing the snapshot of* (A, B, C, D, E)*.*

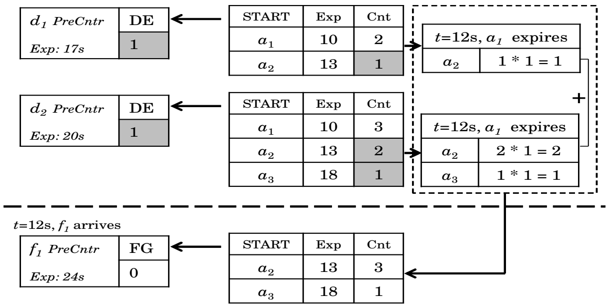


Figure 11: Snapshot Computation of Multi-connect

Cost Analysis. The memory overhead of Chop-Connect originates from the snapshot structure attached to each CNET event. The size of the snapshot structure is determined by the number of active start events *k*. Therefore the memory overhead per CNET event is linear in *k*. The CPU overhead depends on the costs of creating the snapshot structure as each CNET event arrives. It also depends on *k*. Therefore the CPU overhead per CNET event is also linear in *k*. This is identical to the CPU costs of processing each event by the single A-Seq. Since the number of CNET events is far less than the total number of events, the overhead of Chop-Connect is minor compared to supporting multiple queries by applying the single A-Seq on each query to process each event separately.

5. DISCUSSION ON AGGREGATION TYPES

Aggregation functions like COUNT, MAX/MIN, and SUM/AVG are all common. In this paper, we first focused on COUNT, as COUNT is the most popular and important aggregation type for pattern detection in CEP. Now we briefly sketch how the MAX/MIN and SUM/AVG operators could also be supported by A-Seq.

First, we interpret SUM/AVG/MAX/MIN aggregation over sequences. These functions aggregate the attribute value of a certain event type over all sequence matches. For example, assume for all sequence matches of pattern (A, B, C, D), we want the SUM value on event type $C._{weight}$ with weight an attribute of event type C. Then a SUM operation will be applied to add $c._{weight}$ values of all sequence matches together. Similarly, the AVG requires the average $c._{weight}$ among all sequence matches, while MAX/MIN requires the maximum or minimum $c._{weight}$.

We now illustrate that how above aggregation types can be pushed into our *A-Seq* approach. For SUM, we maintain an extra SUM field in each PreCntr structure. Whenever a C instance arrives and causes a relevant count to be updated, the SUM field of each PreCntr will also be updated by a rule similar to Lemma 1: $sum_{t_i} = sum_{t_{i-1}} + count(A, B)_{t_{i-1}} * c._{weight}$. When a TRIG event D arrives at t_j, the final SUM per PreCntr will be calculated as $sum_{t_i} * \frac{count(A,B,C,D)_{t_j}}{count(A,B,C)_{t_i}}$. Then the overall SUM can be simply derived by adding up the sum of each PreCntr.

The AVG can be directly obtained by our $SUM/COUNT$ solution. That is, we can simply maintain both SUM and COUNT at same time. The computation of MAX/MIN is similar as SUM. We maintain a result field, which always tracks the max/min value of $c._{weight}$. Whenever a C instance arrives, we check whether its value is greater or less than the maintained value. We then update this field accordingly.

6. PERFORMANCE EVALUATION

6.1 Experimental Setup

We implement our sequence aggregation techniques inside HP Lab's stream engine CHAOS [18]. Experiments were conducted on a PC with Intel Pentium IV 2.8 GHz CPU and 4GB RAM running Windows 7 system. We first compare our proposed *A-Seq* approach (Sec. 3) with the state-of-the-art two-step approach [19]. We also compare our multi *A-Seq* technique by comparing it against the non-shared A-Seq technique and the state-of-the-art multi-query sharing strategy for sequence queries [9] to demonstrate the effectiveness of our sub-pattern sharing strategy.

Data Sets. We evaluate A-Seq using real stock trades data from [6], which contains stock ticker and timestamp information. The portion of the trace holds 120,000 event instances. For evaluating the multi-query techniques, to make longer queries and to form larger query workloads, we also generate synthetic stock streams with more event types and their corresponding instances.

Evaluation Metrics. We compare the average execution time for computing sequence aggregation per window slide. $Execution\ Time = T_{elapsed} / |Window\ Slides|$, where $|Window\ Slides|$ represents the number of windows that have been processed. Peak memory usage is measured by the maximum number of active Java objects or references. Namely, for the state-of-the-art approach, we count the sum of three types of objects: active event references inserted in the stacks, pointers, as well as intermediate sequence matches for later aggregation. For our *A-Seq* approach, we count the number of active prefix counters (*PreCntr*), in which all the information for aggregation computation is stored. The maximum object counts for both approaches are reported as peak memory usage.

6.2 Single Query Evaluation

Effect of Query Length on A-Seq. In this set of experiments, the length of the sequence pattern is varied from l = 2 to 5. The window size is fixed at $1000ms$. The above patterns are aggregated using the state-of-the-art stack-based technique and our A-Seq approach. The average execution time per window slide is shown in Fig. 12(a) using logarithmic scale on the Y-axis. As confirmed by the cost analysis of the stack-based technique (Sec. 2.2), its execution costs grow exponentially with the length of the pattern query. On the other hand, the average execution time for A-Seq stays fairly constant with varying pattern lengths. At length 5, *A-Seq* is 16,736-fold faster than the state-of-the-art approach.

Fig. 12(b) shows the peak memory usage of the two approaches when varying pattern lengths with Y-axis in logarithmic scale. A-Seq also wins in storage consumption, i.e., it uses significantly less memory, because it stores only prefix counters, whose cardinality is equal to the number of active *START* events. While the state-of-art approach must store all relevant event instances along with pointers indicating the time order among the events.

Effect of Window Size on A-Seq. In this set of experiments, the window size w is varied from $100ms$ to $1000ms$. The query pattern length is fixed at 3. We again measure the average execution time and peak memory usage and compare our A-Seq technique with the state-of-the-art method. Fig. 13(a) shows that as the window size increases, the average execution time of both methods increases. As expected, the state-of-the-art approach degrades significantly faster than *A-Seq*, because A-Seq successfully drives down the CPU costs from polynomial to linear in the number of active events per window. As shown in Fig. 13(b), memory utilization increases in a similar manner with the CPU costs.

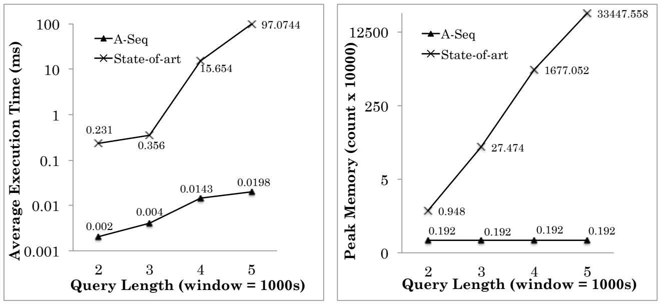
Scalability Test for A-Seq. As demonstrated above, the performance of the state-of-the-art approach degrades dramatically with the growth in either the pattern length or the window size. Now we examine the scalability of *A-Seq* under scalability tests in which the traditional stack-based join method fails in our system (i.e., memory overflow). We vary the pattern length from 6 to 10. The window size is extended to 2000ms.

Fig. 14 (a) illustrates the scalability test result of *A-Seq* in terms of the average processing time. No significant performance degradation of *A-Seq* is observed even in the most extreme case (at length=10 and window=2000). The performance ($0.0219ms/event$) at the extreme case is almost the same as that of the-state-of-the-art performance at the lowest load case of length=2 and window=100 ($0.02ms/event$) in Fig. 12(a).

Negation Test for A-Seq. In this experiment, we study *A-Seq*'s performance for processing queries with negation. We compare the performance of *A-Seq* (negation pushed down) approach with the state-of-the-art (post-filtering negation check) approach, for queries q_1 and q_2 below:

```
q1 = (DELL,IPIX,AMAT)
q2 = (DELL,IPIX,!QQQ,AMAT)
```

q_1 and q_2 have the same positive pattern *(DELL,IPIX,AMAT)* while q_2 has the additional negative event type *QQQ* filter inserted. When processing q_2, the state-of-the-art approach will first collect all matches of the positive pattern *(DELL,IPIX,AMAT)*, and then filter out those matches that contain *QQQ* instances between *IPIX*



(a) Average Processing Time (b) Peak Memory *(objects count)*

Figure 12: A-Seq Performance by Varying Pattern Length

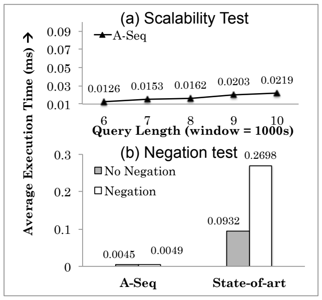


Figure 14: Scalability and Negation

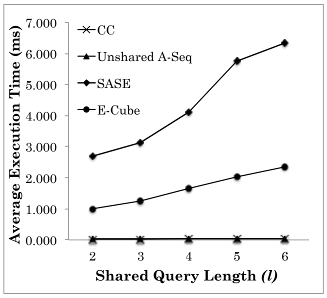


Figure 15: Shared A-Seq v/s State-of-the-art

and *AMAT* instances. While *A-Seq* only needs to reset the count of a particular prefix pattern once a negative event instance arrives. Fig. 14 (b) depicts the average processing time of these two methods. *A-Seq* experiences almost no overhead for processing the negation query, while the state-of-art approach suffers from significant overhead introduced by the post-filtering negation check.

6.3 Multi-query Evaluation

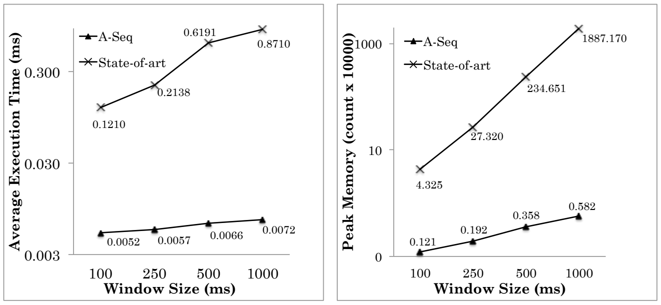
Next we demonstrate the savings in CPU time obtained by sharing common substrings among multiple queries either via Prefix sharing or by Chop-Connect method.

Shared A-Seq v/s ECube Sharing. We first compare our multiple aggregation queries sharing strategy against the state-of-the-art CEP multi-query computation strategy [9]. Fig. 15 shows the average execution time for aggregating 3 queries using four approaches: 1) applying SASE [19] to each query independently, 2) sharing the computation of common substrings in sequence construction followed by independent counting (ECube) [9], 3) applying A-Seq to each query independently, 4) multi-query A-Seq using Chop-Connect(CC). Although ECube outperforms SASE 2 to 3 fold by sharing the computation for common substrings, it is still at least 100 times slower than our A-Seq and CC techniques, leading to the overlapped lines of A-Seq and CC. To clearly show the effectiveness of our multi-query sharing strategy, in the rest of the experiments we only compare it against A-Seq.

6.3.1 Prefix Sharing Strategy

Effect of Varying Length of Shared Prefix: We vary the length of the shared common prefix pattern from 2 to 6. To exclude the effect of the query numbers, each workload only has three queries. This effectively is our worst case, as clearly for more sharing opportunities introduced by more queries, more saving can be reaped.

While the length of the overall pattern does not affect the performance of A-Seq, the length of the common prefix significantly affects the gain in performance achievable by sharing the prefix. As shown in Fig. 16(b) for even only a 3 query workload with a sharing prefix length of 2, A-Seq is 3 times faster than the un-shared A-Seq approach, while the last workload which shares a prefix of length 6 gains more (around 5 times).



(a) Average Processing Time (b) Peak Memory *(objects count)*

Figure 13: A-Seq Performance by Varying Window Size

Effect of Varying Number of Queries Sharing the Prefix: The next set of experiments shows the amount of gain obtained by sharing the prefix among larger number of queries. As shown in Fig. 16(a) when the workload size increases from 2 to 6 queries each sharing a prefix of length 3, Prefix-Share consistently wins around 2 times. The saved average execution time per event increases from 0.027 ms to 0.051 ms.

6.3.2 Chop-Connect (CC) Sharing Strategy

Similar to prefix sharing experiments the length of the common substring and the number of queries sharing the substring both impact the performance of the Chop-Connect sharing strategy.

Effect of Varying Length of Shared Substring l. Similar to the experiment in Sec. 6.3.1, the length of the shared substring is varied from 2 to 6.

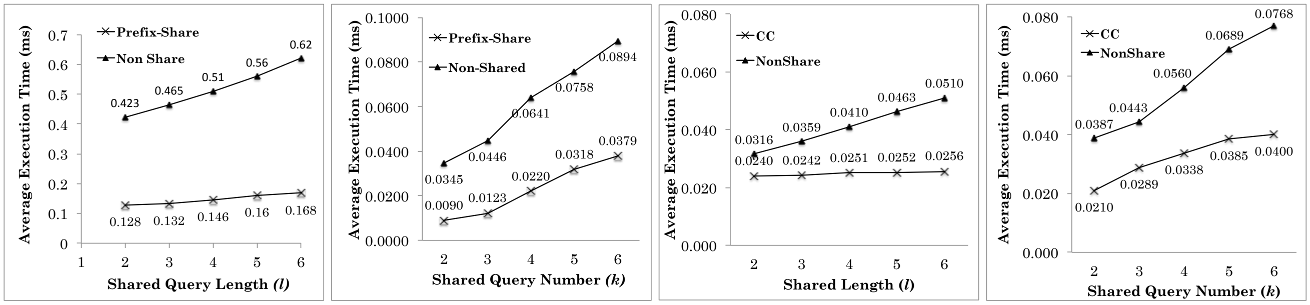
As shown in Fig. 16(c) with the increase in the length of the shared substring, similar to the trend of prefix sharing in Fig. 16(a) the gain obtained by our CC methodology increases from 1.3 times to 2.6 times. This confirms that our CC strategy is lightweight and requires very little CPU overhead to achieve the computation sharing on the common sub-patterns at arbitrary positions.

Effect of Varying the Number of Shared Queries k. Next we add more queries to the workload namely from 2 to 6 queries. The results of *CC* and *NonShare* methods are illustrated in Fig. 16(d). Once again the difference in processing time between the shared and unshared approaches is increasing. This indicates that compared to the *NonShare* method, the performance of *CC* improves with the growth of k. The performance gain is as pronounced as in the prefix sharing experiments (2 times), again confirming our CC strategy is very lightweight.

7. RELATED WORK

Complex Event Processing. Many CEP systems have been developed for scalable pattern detection over high-speed event streams. SASE [2, 19] employs an NFA-based matching model for stack-based sequence construction. Cayuga [3] employs a more general NFA system for processing complex events. These two systems inherit the limitations of the NFA-based model including the late negation-filter processing. ZStream [11] optimized this CEP sequence matching process by selecting a flexible tree-based cost-aware query execution plan in place of the fixed-order NFA evaluation. However, no particular technique has been proposed to address the sequence aggregation problem. Rather, in these CEP systems, aggregations would be applied as a post-pattern-detection step, resulting in inefficient solution.

Aggregations over Stream Data. Traditional aggregation over data streams has also been extensively studied in the literature.



(a) Sharing Prefix Varying l (b) Sharing Prefix Varying k (c) Chop-Connect Varying l (d) Chop-Connect Varying k

Figure 16: Evaluating performance of shared A-Seq

[7, 8] propose incremental techniques that avoid re-computation among overlapping sliding windows. In the stream context without applying sliding window, [20] maintains aggregates using multiple levels of temporal granularity: older data is aggregated using coarser granularity while more recent data is aggregated with fine detail. However, these state-of-the-art aggregation techniques computed aggregation over individual data events of the stream rather than over detected complex sequences. In other words, in our CEP context, these methods could only be plugged in the two-step aggregation operators as the second step – resulting in inefficient performance as demonstrated in our experiments.

Aggregation in Static Sequence Databases. For static sequence databases, SQL-style languages support order-aware join operations among data records along with aggregation functions [10, 1]. However, these works assume that the data is statically stored and indexed prior to processing. Also, the notion of sequence is defined by time-based predicates instead of by continuously arriving event streams. Thus, they focus on designing operators that effectively utilize disk-and-buffer resources to support such time-based predicates. In contrast, *A-Seq* targets dynamic stream data where results are produced instantaneously and continuously upon the arrival of data. In fact A-Seq aims to discard all data upon arrival when possible. [1, 16] support range-based aggregation, where independent data records within a certain time range are aggregated. *A-Seq* instead works at a higher level, where aggregates are over multi-records matches rather than an individual record match. Moreover, in [13, 15], aggregations are specified for patterns with recursion, however, again on independent data records.

Data Mining of Sequential Patterns. Unlike query processing that supports the efficient processing of a particular pattern, sequential pattern mining aims to discover all subsequences of any length that frequently arise over sequential data. Typically, they use a PF-Tree structure and the Apriori principle to find all frequent subsequences [4, 5, 12, 14]. Clearly, the problem introduced by those works is distinct from ours. First, in CEP context, the pattern query of interest is pre-specified by users. Thus our task is to search for occurrences of one given sequence pattern rather than to discover all possible frequent patterns. Second, the notion of sliding window semantics is typically not adopted in sequential pattern mining. Thus, solutions to tackle efficient data purging or result updating are not required. Lastly, existing sequential pattern mining does not handle CEP specific problems including negation, predicates, etc.

8. CONCLUSION

This work is the first to support high performance processing of CEP aggregation queries. Our *A-Seq* solution pushes the aggregation computation into the pattern detection process. It gracefully tackles the CEP-specific challenges including window constraints, negation, and predicates. Compared to the state-of-the-art two-step solution, *A-Seq* is lightweight, thus achieving several orders of magnitude savings in both CPU and memory resources. Effective techniques for aggregation computation sharing among multi-query workloads are also proposed for multi-query optimization. Currently, we assume that stream events arrive in order. In the future we will extend our A-Seq solution to also handle out-of-order event stream.

9. REFERENCES

[1] L. A and S. D. and. Aquery: Query language for ordered data, optimization techniques, and experiments. In *VLDB*, pages 345–356, 2003.

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

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

[4] J. Han, J. Pei, B. Mortazavi-asl, Q. Chen, U. Dayal, and M. Hsu. Freespan: frequent pattern-projected sequential pattern mining. In *KDD*, pages 355–359, 2000.

[5] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. In *KDD*, pages 53–87, 2004.

[6] I. inetats. Stock trade traces. In *http://davis.wpi.edu/dsrg/stockData/eventstream3.txt*.

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

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

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

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

[11] Y. Mei and S. Madden. Zstream: a cost-based query processor for adaptively detecting composite events. In *SIGMOD*, pages 193–206, 2009.

[12] L. F. Mendes and B. D. J. Han. Stream sequential pattern mining with precise error bounds. In *ICDM*, pages 941–946, 2008.

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

[14] J. Pei, J. Han, B. Mortazavi-asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *ICDE*, pages 215–224, 2001.

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

[16] P. Seshadri, M. Livny, and R. Ramakrishnan. Seq: Design and implementation of a sequence database system. In *VLDB*, pages 99–110, 1996.

[17] I. Timko, M. H. Böhlen, and J. Gamper. Sequenced spatio-temporal aggregation in road networks. In *EDBT*, pages 48–59, 2009.

[18] S. Wang, M. Hao, et al. Chaos: A data stream analysis architecture for enterprise applications. In *2009 IEEE conference on commerce and enterprise computing*, pages 33–40, 2009.

[19] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, pages 407–418, 2006.

[20] D. Zhang, D. Gunopulos, V. J. Tsotras, and B. Seeger. Temporal aggregation over data streams using multiple granularities. In *EDBT*, pages 646–663, 2002.