All that Incremental is not Efficient: Towards Recomputation Based Complex Event Processing for Expensive Queries

ABSTRACT

Complex Event Processing (CEP) deals with matching an event stream with the query patterns to extract complex matches. These matches incrementally emerge over time while the partial matches accumulate in the memory. The number of partial matches for expressive CEP queries can be polynomial or exponential to the number of events within a time window. Hence, traditional strategies result in extensive memory and CPU utilisation.In this paper, we revisit the CEP problem through the lens of complex queries with expressive operators (skip-till-any-match and Kleene+). Our main result is that traditional approaches, based on the partial matches' storage, are inefficient for these types of queries. We advise a simple yet efficient query tree approach that experimentally outperforms traditional approach on both CPU and memory usage.

1 INTRODUCTION

Complex Event Processing (CEP) matches a sequence of events within a stream against a complex query pattern that specifies constraints on extent, order, values, and quantification of the matching events.Most of the CEP systems incrementally produce matched patterns, where partial matches are stored and then computed to avoid the recomputation cost [11, 14]. That is, with the arrival of an event, a CEP system (i) can generate a new partial match by matching incoming event with the prefix of the defined query pattern; (ii) checks with the existing partial matches if the incoming event can be part of them or complete them. The number of partial matches for such strategy can be polynomial or exponential to the number of events within a window [1, 9, 14]: some partial matches lead to the complete matches while others fail. This results in extensive memory and CPU utilisations. In the following, we present a real-world CEP query to showcase the issues of incrementally processing partial matches.

Example 1.1. A stock market application processes thousands of financial transactions per second to detect patterns that signify emerging profit opportunities. An example of such a pattern, called V-shaped pattern [12, 14], is described in Query 1 using the syntax from SASE [13]. Query 1 detects an increasing and then decreasing pattern per company. Hence, the price of the matched events must first increase from an initial value (a.price < b.price), then the pattern should show an uptrend (b.price < NEXT(b.price)) using Kleene+, and then the price should decrease such than it is less than the first reported increase (c.price < FIRST(b.price)). All the matched events should be within a window of size 30 minutes which slides every 2 minutes.

Query 1. PATTERN SEQ (a, b+, c) WHERE [companyID]
AND a.price < b.price AND
b.price < NEXT(b.price) AND c.price < FIRST(b.price)
WITHIN 30 minutes SLIDE 2 minutes

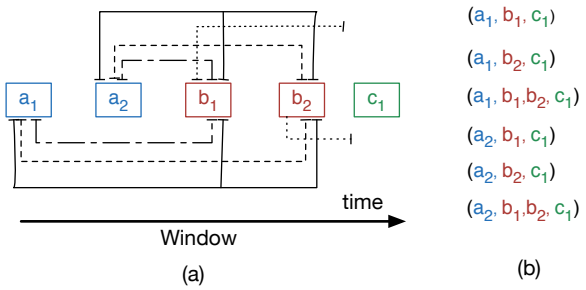

Figure 1: (a) Partial matches for Query 1 over the event stream, (b) complete matches for the Query 1

To reveal all the profit opportunities, Query 1 detects all the combinations of patterns using an operator called as *skip-till-any-match* event selection strategy in the literature [1, 12, 14]. This operator aid in ignoring local price fluctuations to preserve opportunities of detecting longer and thus more reliable patterns. Fig. 1 shows the evaluation of the Query 1 over an event stream. From Fig. 1 (a), before the arrival of an event c_1, six partial matches (shown with the connected lines) for a_1 and a_2 are produced, and one for each b_1 and b_2. Note that the prefix of the Query 1 is highly unselective (*a.price* < *b.price*) and any event can start a partial match. Consequently, there can be a large number of partial matches that would not produce full matches, which result in wasted computation. The complete set of matches, after the arrival of a *trigger* event c_1 are shown in Fig. 1 (b).

State of the art and limitations. The issues with the large number of partial matches and their effect on the memory and CPU resources have been acknowledged in the literature [1, 9, 12, 14]. The usual remedy proposed for such issues is to factorise the commonalities between the partial matches that originate from the same set of events [12, 14]. Hence, the query evaluation is broken into two phases. The first phase tracks the commonalities between the partial matches and compresses them using an additional data structure, e.g. an events graph. The second phase constructs the complete matches while decompressing the set of common partial matches, e.g. through *depth-first-search*. This strategy can reduce the memory requirements from exponential to polynomial, at the cost of the compression/decompression operations. Moreover, since a sub-partial match can be part of multiple complete matches, this strategy recomputes common sub-partial matches for each match that contains one. This results in redundant computations, and the high cost of maintaining additional structure and reconstruction of matches remains quite significant. For instance, in Fig.1 (a), all the partial matches are kept – and computed – even if a trigger event, i.e. c_1, never arrives within a window. Finally, due to the high space complexity, these systems spend considerable amount of time in repeated memory allocation and reclamation with the expiration of a window.

Contributions. In this paper, we initiate the study of a recomputation based CEP that addresses the following two main point. (1) Since storing partial matches is expensive, only events should be stored in a window. This reduces the memory cost from polynomial or exponential to linear. (2) The matches should be recomputed (i) only when the trigger events' arrive, (ii) recomputation should be performed in a batch manner and results should be directly stored on the disk. Hence, redundant computations are not performed and subsets of events are joined to produce matches. Furthermore, since matches are directly stored on the disk, repeated in-memory allocation and reclamation operations are avoided. In theory, both recomputation-based and incremental techniques have the same worst-case run-time cost. However, in practice, recomputation process is (1) performed for a fewer number of times (depending on the trigger events), (2) alleviates the redundant computations that arrives due to the partial matches that would not produce a complete match, and (3) avoids the cost of creation and deletion of partial matches.

The detailed structure and contributions of this paper are as follow. We first provide the compilation of the query tree for a CEP query (§ 2.2). We then present the design of an algorithm for recomputing matches and analyse its complexity(§ 2.3). Based on this algorithm, we present the techniques to process joins between events and the Kleene+ operator (§ 2.4). We implemented our solution and demonstrate that it outperforms existing solutions both in terms of memory and CPU cost (§ 3).

2 RECOMPUTATION BASED CEP

2.1 Preliminaries

In this section, we present CEP specifics definitions, query representation and query evaluation techniques.

Event. An *event* e is tuple (A, t), where $A = \{A_1, A_2, \ldots, A_m\}$ $(m \geq 1)$ is a set of attributes and $t \in \mathbb{T}$ is an associated timestamps that belongs to a totally ordered set of time points $(\mathbb{T}, \leq)$.

Event Stream. An *event stream* $\mathcal{S}$ is a possibly infinite set of events such that for any given timestamps t and t', there is a finite amount of events occurring between them.

Event Sequence. A chronological ordered *sequence of events*, with a total ordering given by $\mathbb{T}$ is represented as $\vec{E} = \langle e_1, e_2, \ldots, e_n \rangle$ with e_1 refer to the *first* event and e_n to the *last*.

CEP Query. A CEP query Q has the following form:

$$\text{PATTERN } P \ [\text{ WHERE } \Theta \ ] \text{ WITHIN } w \text{ SLIDE } s$$

where $P = \text{SEQ}(p_1, \ldots, p_k)$ is a sequence of pairwise disjoint variables of the form p and p^+, $\Theta = \theta_1, \ldots, \theta_l$ is a set of predicates (constant and variable) over the variables in P (see Query 1 in Example 1), ω is the time window and s is slide of the window to define the scope of event stream. A variable $p \in P$ binds a sequence of a single event $\langle e_i \rangle$, while the qualified variables $p^+ \in P$ binds a sequence of one or more events $\langle e_1, \ldots, e_n \rangle$, $n \geq 1$ for a query match.

CEP Query Match. To define the matching of a CEP query Q, we use a substitution $\gamma = \{p_1/\vec{E_1}, \ldots, p_k/\vec{E_k}\}$ to bind the event sequences $(\vec{E})$ with the variables. Given Q and the event stream $\mathcal{S}$, a substitution γ is a match of Q in $\mathcal{S}$, iff (i) all the predicates in Q evaluate to true, (ii) for events in the two event sequences $e \in \vec{E_i}$ and $e' \in \vec{E}_{i+1}$, $e.t < e'.t$ and (iii) all the events in each event sequence $\vec{E_i}$ has timestamps less than the defined window w. Since no order is imposed on the selected events, it complies to the skip-till-any-match selection strategy.

2.2 Query Tree

Given the CEP query, we need to compile it from the high-level language into some form of automaton [1, 4, 6, 13] or a tree-like [5, 11] structure to package the semantics and executional framework. Since we are working with the recomputation-based model, a traditional tree structure customised for the streaming and recomputation settings would suit our needs. Given Q, we construct a tree, where leaf nodes are the substitution pairs, i.e. $(p_i/\vec{E_i})$ to store the primitive events and the internal nodes represent the joins on the defined predicates Θ and temporal ordering. We call it a query tree $\mathcal{T}_q$. Our model differ from other tree-structures [5, 11], since we do not store any partial matches. An example of such a tree for Query 1 is shown in Fig 2, where we have three leaf nodes for the variables bindings $a/\vec{E_1}$, $b/\vec{E_2}$ and $c/\vec{E_3}$. The internal nodes in Fig. 2 evaluate the defined Θ in terms of joins (denoted as $\bowtie_\Theta$) for all the variables $p \in P$. Furthermore, since for a CEP query, the matched events should follow the sequential order, joins on the timestamps (denoted as $\bowtie^t$) are also provided in the $\mathcal{T}_q$.

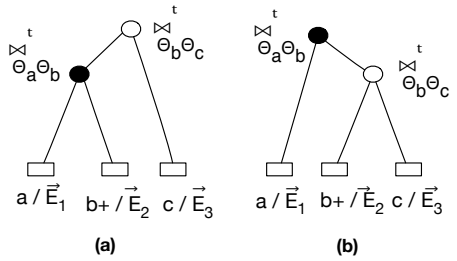

Figure 2: (a) Left-deep and (b) Right-deep Query tree for Query 1 in Example 1.1

2.3 Query Evaluation

We now present the algorithm to evaluate the query tree over the event stream without storing the partial matches. Algorithm 1 shows the query evaluation and is divided into three main steps.

Algorithm 1: Baseline Algorithm

```
   Input: Query Tree T_q and an event stream S
   Output: A set of query matches
1  Q ← (P, Θ, ω, s) ;                                   // CEP Query
2  E⃗ ← {E⃗_1, E⃗_2, ..., E⃗_k}, k = |P| ;                 // Event sequences for T_q
3  for each e ∈ S do
4      for each E⃗_i ∈ E⃗ do
5          if isCompatible(E⃗_i,e) then
6              E⃗_i = E⃗_i ∪ e ;                           // Step 1
7      if isCompatible(E⃗_k,e) then
8          ExecuteJoins(E⃗, Θ);                           // Step 2
9          ExecuteKleenePlus(E⃗, Θ);                      // Step 3
```

Step 1. For each incoming event e, we add e to the compatible event sequence $\vec{E_i}$, such that constant predicates (e.g. $a.price > 10$) filter the unwanted events for each $\vec{E_i}$ in $\mathcal{T}_q$. For instance, for a constant predicate $a.price > 10$, all the events in the $a/\vec{E_i}$, should have $price > 10$. This step (*lines 4-6*) constitutes to the accumulation of events within a defined window.

Step 2. For each incoming event e, check if it can trigger the query evaluation to produce matches. That is, if e can be part of

$\vec{E}_k$ ($k = |P|$), it can complete a set of matches; since it contains the highest timestamp within the window. For instance, Query 1 produces matches only when an event of type c arrives. Hence, we execute the query tree for a trigger event. By execution, we mean executing the joins between events within each $\vec{E}_i$ using the predicates Θ and timestamps. This step (*lines 7-8*) assembles all the events, in a batch manner, for each $\vec{E}_i$ that can produce the set of matches.

Step 3. For a $p_i^+ \in P$, we need to compute all the combinations for the events in $p_i^+/\vec{E}_i$, i.e. a power set of events in $\vec{E}_i$. For instance, in Fig. 1 and using Query 1, each a event has $2^{|2|} - 1$ matches for two b events. This step (*line 9*) groups all the combinations by following the one or more semantics of the Kleene+ operator.

2.4 Detailed Analysis

We now present the details of the two main processes of Algorithm 1, i.e. joining the set of events and computing the power set of events for the Kleene+ operator.

Execution of Joins. Let $\vec{E}_i$ and $\vec{E}_j$ are two event sequences with theta-join $\vec{E}_i \bowtie_{\Theta}^{t} \vec{E}_j$ over the timestamp t and predicates Θ. Hence, we have joins on multiple relations for the **Step 2**. The generic cost of such joins, i.e. pairwise join, is $O(|\vec{E}_i||\vec{E}_j|)$ and the problem of its efficient evaluation resembles to the traditional theta-joins with inequality predicates [8]. The wide range of methods for this problem include: the textbook merge-sort, hash-based, band-join and various indices such as Bitmap [7]. These techniques are mostly focused on equality joins using a single join relation, however. The inequality joins on multiple join relations are notoriously slow and multi-pass *projection-based* strategies [3, 8] are usually employed. These strategies, however, require multiple sorting operations, each for a distinct relation, and are only optimised for the static datasets, where indexing time is not of much importance. Considering this, we employ the general nested-loop join for our preliminary algorithm. Our experimental analysis showcase that even such an algorithm provide competitive performance.

Execution of Kleene+ Operator. For **Step 3**, we need to create all the possible combinations of matches over the joined events. That is, enumerating the powerset of event sequences' with p^+ bindings. A traditional solution in this context would be to generate Gray code sequence of events with p^+ bindings, where a new match can be constructed from its immediate predecessor by adding or removing an event. However, this would require storing the predecessor matches to produce the next one and would result in an extra load on the memory resources. To implement Kleene+ operator efficiently, we use the joined results (from **Step 2**) while generating the binary representation of the possible matches using the Banker's sequence [10]. That is, we check the number of events in event sequences' with p^+ bindings after the join process. For $|m|$ number of such events, we need to create 1 to $2^m - 1$ matches. This means if we generate all binary numbers from 1 to $2^{|m|} - 1$, and translate the binary representation of numbers according to the location of the events in the $p^+/\vec{E}_i$, we can produce all the matches for the Kleene+ operator in a batch manner. For instance, consider Fig. 3 (using Query 1), where there are three b events for the Kleene+ operator. Hence, we generate binary numbers from 1 to $2^3 - 1$, now equates 1 as take element at the specified location of the $\vec{E}_2$ and 0 as do not take the element. Then using the generated binary numbers, we generate all the combinations of matched events.

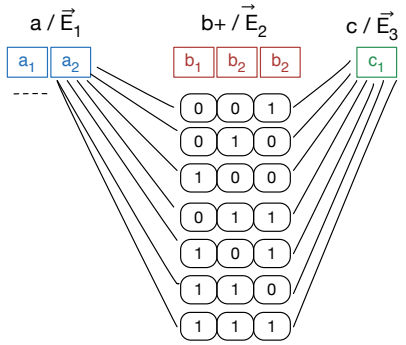

Figure 3: Execution of the Kleene+ operator using the Banker's sequence and generated binary numbers

Complexity Analysis. Herein, we briefly present the complexity analysis for the three steps described in Algorithm 1. **Step 1** result in a constant time operation, since an incoming event can be directly added to an event sequence. **Step 2** has a polynomial time cost (pair-wise joins) and depends on the number of patterns P defined in a CEP query. For n events in a window and $k = |P|$, we have $O(n^k)$. **Step 3** requires to produce exponential number of matches for a Kleene+ operators. For n events in a window, we have $O(2^n)$. The memory cost for the Algorithm 1 is linear to the number of events within a window.

3 EXPERIMENTAL EVALUATION

In this section, we report the results of our experimental study on both incremental and recomputation-based methods for CEP. Our proposed techniques have been implemented in Java and our system is called RCEP. All the experiments were performed on a machine equipped with Intel Xeon E3 1246v3 processor and 32 GB of memory. For robustness, each experiment was performed 3 times and we report median values.

Datasets. We employ both real and synthetic datasets to compare the performance of our proposed techniques.

Synthetic Stock Dataset (S-SD): We use the SASE++ generator, as used in [14], to produce the synthetic dataset. Each event carries timestamp, company-id, volume and price of a stock. This dataset enables us to tweak the selectivity measures of matches $\frac{\#ofMatches}{\#ofevents}$ to evaluate the performance of the systems at different workloads. In total, the generated dataset contains 1 million events.

Real Credit Card Dataset (R-CCD): We use a real dataset of credit card transactions [2]. Each event is a transaction accompanied by several arguments, such as the time of the transaction, the card ID, the amount of money spent, etc. This dataset contains around 1.5 million events.

Queries. We consider 8 different variations of Query 1 (Section 1) against the stock dataset. These queries variations differ by the constant predicates and time window to control the selectivity of produced matches. For the credit card dataset, we use a CEP query describing "Big after Small"pattern [2] using the SEQ(a, b+) template. That is, an outstandingly large amount of transactions after one or a series of small amounts.

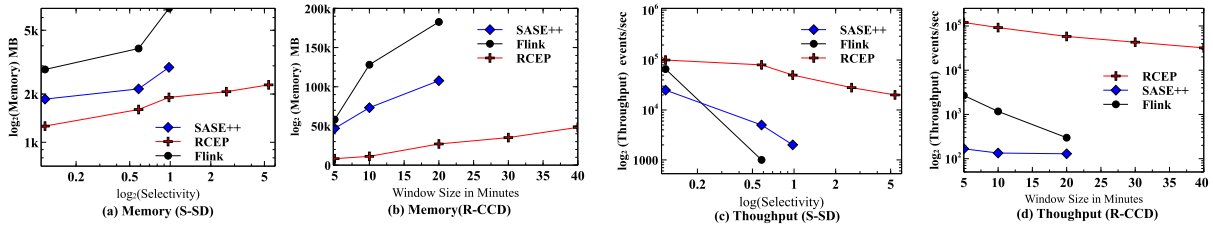

Figure 4: Memory cost and throughput analysis of the CEP systems over the two datasets S-SD and R-CCD

Methodology. We compare RCEP with the SASE++ [14] and the open source industrial system Apache Flink [6]. All of these system support skip-till-any-match and Kleene+ operator: both SASE++ and Flink employ incremental evaluation of partial matches. Unless otherwise specified, all experiments use a slide granularity $s = 1$. We measure two standard metrics common for the CEP systems: throughput and memory requirements [1, 12, 14]. The memory requirements were measured by considering the resident set size (RSS) in MBs. RSS was measured using a separate process that polls the `/proc` Linux file system, once a second. We use the selectivity measures $\frac{\#ofMatches}{\#ofEvents}$ and window size to test different workloads.

Memory Cost. Figs. 4 (a) and (b) show the memory consumption of all the systems for both datasets. As expected, increase in the selectivity measures (subsequently window size) result in a large number of partial matches and extensive memory cost for both SASE++ and Flink. In particular, Apache Flink consumption is exponential in terms of the number of events in the window. SASE++ managed to sustain memory requirement due to the it superior compression of Kleene+ matches. However, with the increase in the number of events that prefixed a new partial match, its memory utilisation increases to about two orders of magnitude compared to our recomputation-based approach. This phenomenon is largely observed in Fig. 4 (b), where the credit card dataset contains a large number of events that can initiate a new partial match. In contrast, RCEP scales linearly to the number of events within a window and not the partial matches.

CPU Cost. Figs. 4 (c) and (d) show the relative performance of the CEP systems over both datasets. We can see that, in general, RCEP have much higher throughput (more than an order of magnitude) than Flink and SASE++. As a matter of fact, SASE++ and Flink do not produce results for several hours for the moderate selectivities and window sizes. This is because the cost of SASE++ and Flink is highly dependent on the number of partial matches within a window. As the window size (subsequently selectivity) increases, both systems produce a large number of partial matches and spend most of their time in compressing and decompressing of common events within the partial matches. That is, traversing through the stack of pointers using depth-first-search to extract all the matches. In contrast, (i) RCEP initiates the recompuation of matches only if the triggered events' arrive; (ii) the execute joins over the stored events; and (iii) the Kleene+ operator is executed only for the events that can be part of the final matches. Hence, RCEP performs much better and consume less memory than SASE++ and Flink, often by 1-2 orders of magnitude.

4 LOOKING AHEAD

In this preliminary study, we have highlighted the utility of recomputation-based CEP for expensive CEP queries. We have proposed our first algorithm for recomputing the matches with the arrival of new events. To our knowledge, ours is the first algorithm of this kind in the context of CEP. Our experimental results show that recomputation-based approach outperforms the incremental approach used by the existing systems.

Our study opens up several directions for future work. A major direction is to establish techniques to efficiently store and index events within a defined window. Without this, we cannot discard events within an event sequence unless it is accessed and compared with all the other events. Hence, indexing of events would enable us to prune irrelevant events before the joining process. Further, we plan to consider new algorithms for multi-relational and inequality joins in streaming settings, since existing algorithms are only efficient for the static workloads and require extensive indexing time. Finally, we would like to incorporate our solution in the open-source Apache Flink framework.

REFERENCES

[1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, pages 147–160, 2008.
[2] A. Artikis, N. Katzouris, I. Correia, C. Baber, N. Morar, I. Skarbovsky, F. Fournier, and G. Paliouras. A prototype for credit card fraud management: Industry paper. In *DEBS*, pages 249–260, 2017.
[3] J. L. Bentley and J. H. Friedman. Data structures for range searching. *ACM Comput. Surv.*, 11(4):397–409, Dec. 1979.
[4] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White. Cayuga: A high-performance event processing engine. In *SIGMOD*, pages 1100–1102, 2007.
[5] ESPER. http://www.espertech.com/esper/.
[6] A. Flink. https://flink.apache.org/.
[7] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems - the complete book (international edition).* Pearson Education, 2002.
[8] Z. Khayyat, W. Lucia, M. Singh, M. Ouzzani, J.-A. Papotti, N. Tang, and P. Kalnis. Fast and scalable inequality joins. *The VLDB Journal*, pages 125–150, 2017.
[9] I. Kolchinsky, I. Sharfman, and A. Schuster. Lazy evaluation methods for detecting complex events. In *DEBS*, pages 34–45, 2015.
[10] J. Loughry, J. van Hemert, and L. Schoofs. Efficiently enumerating the subsets of a set. *Applied-math*, 2000.
[11] Y. Mei and S. Madden. Zstream: A cost-based query processor for adaptively detecting composite events. In *SIGMOD*, pages 193–206, 2009.
[12] O. Poppe, C. Lei, S. Ahmed, and E. A. Rundensteiner. Complete event trend detection in high-rate event streams. In *SIGMOD*, pages 109–124, 2017.
[13] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIDMOD*, 2006.
[14] H. Zhang, Y. Diao, and N. Immerman. On complexity and optimization of expensive queries in complex event processing. In *SIGMOD*, 2014.