Complete Event Trend Detection in High-Rate Event Streams

Olga Poppe*, Chuan Lei**, Salah Ahmed* and Elke A. Rundensteiner*
*Worcester Polytechnic Institute, 100 Institute Road, Worcester, MA 01609
**NEC Labs America, 10080 N Wolfe Rd, Cupertino, CA 95014
*opoppe|sahmed2|rundenst@wpi.edu, **chuan@nec-labs.com

ABSTRACT

Event processing applications from financial fraud detection to health care analytics continuously execute event queries with Kleene closure to extract event sequences of arbitrary, statically unknown length, called Complete Event Trends (CETs). Due to common event sub-sequences in CETs, either the responsiveness is delayed by repeated computations or an exorbitant amount of memory is required to store partial results. To overcome these limitations, we define the CET graph to compactly encode all CETs matched by a query. Based on the graph, we define the spectrum of CET detection algorithms from CPU-optimal to memory-optimal. We find the middle ground between these two extremes by partitioning the graph into time-centric graphlets and caching partial CETs per graphlet to enable effective reuse of these intermediate results. We reveal cost monotonicity properties of the search space of graph partitioning plans. Our CET optimizer leverages these properties to prune significant portions of the search to produce a partitioning plan with minimal CPU costs yet within the given memory limit. Our experimental study demonstrates that our CET detection solution achieves up to 42–fold speed-up even under rigid memory constraints compared to the state-of-the-art techniques in diverse scenarios.

1. INTRODUCTION

Complex Event Processing (CEP) has emerged as a prominent technology for supporting streaming applications from financial fraud detection to health care analytics. CEP systems consume high-rate streams of primitive events and evaluate expressive event queries to detect event sequences such as circular check kites and irregular heart rate trends in near real time. These event sequences may have arbitrary, statically unknown, and potentially unbounded length. They are expressed by event queries with Kleene closure and are henceforth called Complete Event Trends (CETs).

Motivating Examples. We now describe three application scenarios of time-critical CET detection.

SIGMOD'17, May 14-19, 2017, Chicago, Illinois, USA
© 2017 ACM. ISBN 978-1-4503-4197-4/17/05...$15.00
DOI: http://dx.doi.org/10.1145/3035918.3035947

• ***Financial Fraud Detection***. Circular check kiting is an example of event-trend detection for financial fraud. In a simple case, it involves writing a check for a value greater than the account balance from an account in Bank A, then writing a check from another account in Bank B, also with insufficient funds, with the second check serving to cover the non-existent funds from the first account. Fraudsters take advantage of the float and withdraw funds from the account before the banks can detect the scheme (Figure 1).

Complex versions of this scheme have occurred involving multiple fraudsters posing as large businesses, thereby masking their activity as normal business transactions. This way they coax banks to waive the limit of available funds [6]. To implement this scheme, fraudsters transfer millions among banks, using complex webs of worthless checks. As just one example, in 2014, 12 people were charged in a large-scale "bustout" scheme, costing banks over $15 million [5].

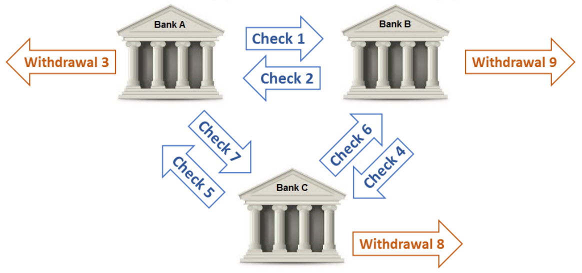


Figure 1: Circular check kiting

```
Q_1 : PATTERN Check+ c[ ]
      WHERE c.type = 'notcovered' AND
            c.destination = NEXT(c).source
      WITHIN 1 day SLIDE 10 minutes
```

Query Q_1 detects a chain (or circle) of any length formed by not covered check deposits during a time window of 1 day that slides every 10 minutes. The pattern of the query is the Kleene closure on check deposit events, denoted Check+ c[ ]. The predicates require the checks in a chain to be not covered. The destination of a check c must be the same as the source of the next check NEXT(c) to form a chain.

Since arbitrary many fraudsters, financial transactions and banks worldwide can be involved in this scheme, detection of circular check kites is a computationally expensive problem. To prevent cash withdrawal from an account that is involved in at least one check kiting scheme, the query continuously analyzes high-rate event streams with thousands of financial transactions per second and detects *all complete* check kiting trends in real time.

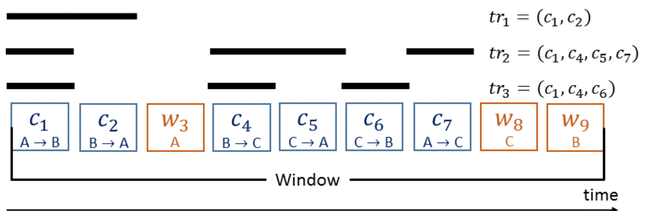


Figure 2: Three check kite trends detected by query Q_1

In Figure 2, ($c_1 : A \rightarrow B$) denotes a not-covered check deposit event from Bank A into Bank B at time 1 and ($w_3 : A$) denotes a cash withdrawal event from Bank A at time 3. Three check kiting trends are detected by query Q_1. They are shown as black lines above the event stream: (c_1, c_2), (c_1, c_4, c_5, c_7) and (c_1, c_4, c_6). Note that check c_2 is part of the first trend but is skipped to detect the second and third event trends. Such flexible way of finding all matches is called skip-till-any-match event selection strategy [10].

• ***Health Care Analytics***. Cardiac arrhythmia is a group of serious hearth diseases in which the heartbeat is irregular, too fast or too slow. It can lead to life-threatening complications such as stroke or cardiac death. There are more than 250k sudden cardiac deaths per year [9]. Thus, it is very important to detect extreme differences in heartbeat in high-rate streams of thousands of measurements per second and immediately trigger lifesaving measures.

```
Q2 : PATTERN SEQ(Activity a, Activity+ b[ ])
     WHERE [personID] AND b.rate < NEXT(b).rate AND
               a.rate*2 < b.rate AND b.type = 'passive'
     WITHIN 10 minutes SLIDE 1 minute
```

Query Q_2 monitors physical activity per patient and detects life-threatening conditions when the heart rate gradually increases until it doubles compared to the first measurement despite passive physical activity. The query pattern is the sequence of the first activity measurement event and the Kleene closure on all following activity events. The predicates require all events in a trend to be about the same person, denoted by [personID]. *All complete* irregular heartbeat trends must be detected in real time to enable lifesaving measures in critical stages of cardiac arrhythmia.

• ***Stock Trend Analytics*** platforms process thousands of financial transactions per second to detect all event trends that signify emerging profit opportunities. Examples of such event trends include increasing and decreasing stock market trends [10, 16], head-and-shoulders pattern [12], V-shaped [26], M-shaped [19], and W-shaped [27] trends.

```
Q3 : PATTERN Stock+ s[ ]
     WHERE [companyID] AND s.price < NEXT(s).price
     WITHIN 1 hour SLIDE 30 minutes
```

Query Q_3 detects increasing stock market trends per company using a Kleene closure on stock events. The predicates require all events in a trend to have the same value of the company identifier, denoted by [companyID]. The price of consecutive events in a trend must increase. To reveal all profit opportunities, the query detects *all complete* event trends – ignoring local price fluctuations to preserve opportunities of detecting longer and thus more reliable trends.

Challenges of time-critical CET detection.

• ***Exponentially Many CETs of Unbounded Length***. Not only is each CET of statically unknown and potentially unbounded length but also the number of CETs is proven to be exponential in the number of relevant events in the worst case (Appendix A). This complexity may jeopardize the real-time CET detection in high-rate event streams.

• ***CPU versus Memory Trade-off of CET Detection***. Due to the occurrence of many common event sub-sequences in CETs, either repeated computations are invoked or an exorbitant amount of memory is required to store partial CETs during CET detection. Consequently, the system may either fail to deliver the CET results with low-latency, or run out of memory due to high-rate event streams.

• ***NP-Hard Stream-Partitioning Problem***. The divide and conquer principle is a common solution to the above problem of trading between CPU and memory. Namely, we could partition the stream, cache results per partition and reuse them for final result construction. However, with the search space being exponential [21], an effective lightweight stream-partitioning algorithm must be developed to guarantee prompt system responsiveness.

State-of-the-Art Approaches. Existing event processing approaches recognize the importance of Kleene closure computation over event streams [10, 16, 26, 37, 38]. However, Cayuga [16] and ZStream [26] do not support the skip-till-any-match semantics required to express the above use cases. While SASE++ [38] supports Kleene closure computation under skip-till-any-match, it stores single events and forms matches at the end of each window. Since an event sub-sequence can be part of multiple matches, SASE++ recomputes a common event sub-sequence for each match that contains it. This approach suffers from repeated computations. For example, the query latency of SASE++ is 38 minutes when the event rate is 50k per second and the query window is 30 minutes (Section 7.2). Such long processing delay is unacceptable for time-critical applications which require responsiveness within a minute. In summary, the existing approaches do not fully address the above challenges of real-time CET detection over high-rate event streams.

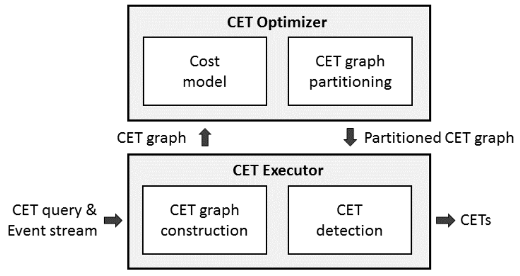


Figure 3: CET processing paradigm

Proposed CET Approach. Given an event query with a Kleene pattern, our CET processing paradigm extracts events matched by the query from a high-rate event stream and encodes their CET relationships in a compact data structure, called CET graph (Figure 3). Based on the graph, we propose a family of CET detection algorithms ranging from the memory-optimal M-CET to the CPU-time-optimal T-CET solution. M-CET avoids excessive storage of intermediate results and thus invokes repeated computations. T-CET accelerates CET detection by incrementally maintaining intermediate results but unfortunately requires an exorbitant amount of memory. To trade off between CPU and memory costs, we develop the Hybrid CET detection algorithm H-CET which is a middle ground between these two extremes.

Namely, we partition the CET graph into smaller graphlets. Based on the partitioned graph, H-CET caches partial CETs per graphlet using T-CET, and then stitches these partial CETs together to form the final CET results using M-CET. Partitioning faces the trade off that finer-grained partitioning plans reduce the memory consumption since CETs across graphlets are not stored while increasing the execution time, and vice versa. We thus design a cost-driven CET optimizer that finds an optimal partitioning plan with minimal CPU execution costs yet within the available memory limit.

Contributions. Our key innovations are the following:

1) We define the problem of real-time CET detection over high-rate event streams under memory constraints. We prove that the number of CETs is exponential in the number of relevant events in the worst case.

2) We introduce a compact data structure, called CET graph, to encode relevant events and their CET relationships. The spectrum of CET detection algorithms ranging from the CPU-time-optimal algorithm T-CET to the memory-optimal algorithm M-CET is introduced.

3) To trade-off between CPU and memory costs, we develop the Hybrid CET detection algorithm H-CET. We first partition the CET graph into time-centric graphlets. Then, H-CET computes CETs per graphlet using T-CET and reuses these partial results to detect all CETs using M-CET.

4) We establish a cost model for CET detection. Our analysis reveals cost monotonicity properties of the search space of candidate partitioning plans. We then design the CET optimizer that leverages these properties to effectively prune sub-optimal plans. Our optimizer is guaranteed to produce a graph-partitioning plan with minimal execution time within the memory bound.

5) We conduct an extensive performance evaluation of our CET approach using both synthetic and real data sets. Our CET solution achieves up to 42–fold speed-up compared to the state-of-the-art strategies [10, 37, 38] – bringing the time-critical CET detection over high-rate event streams into the realm of practicality.

Outline. We start with preliminaries in Section 2. We design our baseline CET detection algorithm in Section 3, while the CET-graph-based approaches are introduced in Section 4. Sections 5 and 6 are devoted to the foundation of CET-graph partitioning and the CET optimizer. We report on the performance study in Section 7. Related work is discussed in Section 8, while Section 9 concludes the article.

2. PRELIMINARIES

2.1 CET Data and Query Model

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

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

Example 1. In the check kitting example, a check deposit event carries a status (covered or not), a source bank and a destination bank. Other attributes (such as account owner, balance, amount) are ignored here for simplicity.

Event Stream. Events are sent by event producers (e.g., ATM machines) to event consumers (e.g., financial fraud detection system) on an input *event stream I*.

CET Query. The input event stream is continuously monitored by event queries that detect CETs in near real time. To avoid reinventing the wheel, we borrow the query syntax and semantics from SASE [10]. We focus on event queries with Kleene closure. As demonstrated by our examples in Section 1, Kleene enables expressive event queries that detect matches of arbitrary, statically unknown length.

Definition 1. (**CET Query**.) A *CET query* has the form:

$$\mathsf{PATTERN}\ P\ [\mathsf{WHERE}\ \theta]\ \mathsf{WITHIN}\ l\ \mathsf{SLIDE}\ s$$

and consists of an event pattern P defined below, optional predicates θ, and a sliding window of length l time units that slides every s time units.

An *event pattern P* is a time-ordered sequence of one Kleene closure pattern and any number of event types. More exactly, let $E_1, ..., E_n$ be event types, $x[\ ]$ be an array, and $y_1, ..., y_n$ be variables. Then $P = p_i$ or $P = \mathsf{SEQ}(p_1, ..., p_n)$ where one sub-pattern $p_i = (E_i +\ x[\ ])$ is a Kleene closure pattern on events of type E_i while all other sub-patterns $p_j = (E_j\ y_j)$ are event types. Events matched by p_i and p_j are bound to the array $x[\ ]$ and the variable y_j respectively. $n \in \mathbb{N}, 1 \leq i, j \leq n, i \neq j$.

Complete Event Trend. We call the matches of a CET query *event trends* instead of using the traditional term of an *event sequence* since event sequences usually have fixed, statically known length [23, 30, 31].

A new event extends an existing event trend if the event is compatible with the trend. An event $e \in I$ is *compatible* with an event trend tr matched by q if for each event e_i in the trend tr $e_i.time < e.time$ holds, the trend (tr, e) is matched by the event pattern of q, satisfies the predicates of q, and is within the window of q.

Example 2. In Figure 2, the event c_4 is compatible with the trends tr_2 and tr_3 since c_4 is matched by the pattern of Q_1 ($c_4.type = Check$), satisfies the predicate of Q_1 ($c_1.destination = c_4.source$), and is within the window of Q_1. In contrast, c_4 is not compatible with tr_1 because the predicate of Q_1 is violated ($c_2.destination \neq c_4.source$).

The result of a CET query corresponds to the subset of all possible matches, namely, complete matches, called *complete event trends*. The shorter matches can then be easily derived from complete matches [38].

Definition 2. (**CET**.) Let $w \subseteq I$ be the set of events within the window of q. An event trend tr is *complete* (Complete Event Trend, CET for short) if $\nexists e \in w$ that the trend tr is compatible with and e is not part of the trend tr. A trend tr is said to be *incomplete* in w otherwise.

Example 3. The trends in Figure 2 are complete since no event can be added to them without violating query Q_1.

2.2 CET Detection Optimization Problem

Many CET detection applications are time-critical (Section 1). Thus, our goal is to minimize CPU processing time of CET detection in main-memory settings.

Definition 3. (**Problem Statement**.) Given a CET query q, a high-rate stream I, and available memory M, the CET detection optimization problem is to detect all CETs matched by the query q in the stream I while *minimizing the CPU costs* and staying within the memory limit M.

This problem is prohibitively expensive with respect to both CPU processing time and memory consumption since the number of CETs is exponential (Theorem 1).

THEOREM 1. *[**Maximal Number of CETs.**] $3^{\frac{n}{3}}$ CETs can be constructed from n events in the worst case.*

To keep the discussion brief, we prove Theorem 1 in Appendix A while we sketch the intuition here. The problem of determining the maximal number of CETs that can be constructed from n events is equivalent to the problem of dividing n elements into groups such that the product of all group sizes is maximal. To maximize the product, all groups must have the same size $x = 3$.

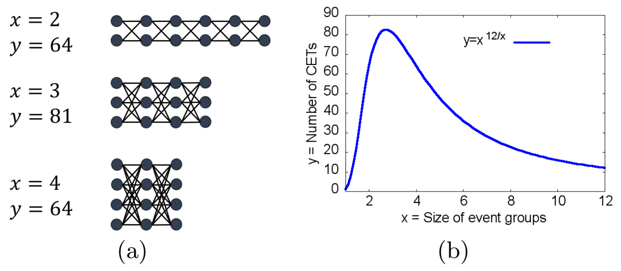


Figure 4: Maximal number of CETs given $n = 12$ events

Example 4. In Figure 4(a), we consider three scenarios with the same number of events $n = 12$. These events are divided into $\lceil \frac{n}{x} \rceil$ groups with at most x events in each group where $n, x \in \mathbb{N}$, $1 \leq x \leq n$. Events in one group are incompatible with each other. Each event in a group is compatible with each event in the following group. A pair of compatible events is connected by an edge. Then the number of CETs corresponds to the product of all group sizes: $y = \prod_{i=1}^{\frac{n}{x}} x = x^{\frac{n}{x}}$. Figure 4(b) shows the number of CETs $y = x^{\frac{n}{x}}$ given $n = 12$ events while varying the size of event groups x from 1 to 12. If $x = 3$, $y = 3^{\frac{n}{3}} = 81$ is maximal.

3. BASELINE CET DETECTION

We now design our baseline CET detection algorithm. It consumes a CET query and an event stream. As new events arrive, it incrementally constructs CETs and returns them when the query window ends.

Three Cases of the Baseline Algorithm. Three cases are possible for processing a new event e. If e is compatible with an existing CET tr, e is appended to tr (Case 2). e may be compatible not with a whole CET tr but with its prefix. In this case, a new CET is created by appending e to the prefix (Case 3). If tr is a CET of length n, its *prefix* $tr(i)$ is a CET that contains all events from the first till the i^{th} event of the trend tr in the same order, $1 \leq i \leq n$. Lastly, if e is compatible neither with an existing CET nor with its prefix, e starts a new CET (Case 1). We now illustrate this baseline algorithm by Example 5.

Example 5. Assume query Q_1 in Section 1 is evaluated against the events c_1-c_4 in Table 1. At the beginning, the set of CETs is empty. When c_1 arrives, a new CET $tr_1 = (c_1)$ is started (Case 1). When c_2 arrives, it is compatible with tr_1 and thus extends it: $tr_1 = (c_1, c_2)$ (Case 2). When c_3 arrives, it is compatible with c_1 but not c_2. In other words, c_3 is compatible with the prefix of tr_1. The newly formed trend is $tr_2 = (c_1, c_3)$ (Case 3). Lastly, when c_4 arrives it is compared to the existing CETs $tr_1 = (c_1, c_2)$

Event	Event trends	Explanation
$c_1 : A \rightarrow B$	(c_1)	Case 1: Create a new CET
$c_2 : B \rightarrow C$	(c_1, c_2)	Case 2: Append to a CET
$c_3 : B \rightarrow D$	(c_1, c_2), (c_1, c_3)	Case 3: Append to the compatible prefix of a CET
$c_4 : D \rightarrow E$	(c_1, c_2), (c_4), (c_1, c_3, c_4)	Eliminate incomplete event trends

Table 1: Three cases of the baseline algorithm

and $tr_2 = (c_1, c_3)$. Since c_4 can be appended neither to tr_1 nor to its prefix, it starts a new trend $tr_3 = (c_4)$ (Case 1). However, c_4 can extend $tr_2 = (c_1, c_3, c_4)$ which makes the trend tr_3 incomplete. Thus, tr_3 is eliminated.

Algorithm 1 Baseline CET detection algorithm

```
Input: CET query $q$, input event stream $I$
Output: CETs $T_{prev}$
 1: $T_{prev} \leftarrow \emptyset$
 2: for all $e \in I$ such that $e.isMatchedBy(q)$ do
 3:     $T_{new} \leftarrow \emptyset$
 4:     if $T_{prev} = \emptyset$ then
 5:         $T_{new} \leftarrow \{e\}$                                  // Case 1
 6:     else
 7:         for all $t \in T_{prev}$ do
 8:             if $isCompatible(q, t, e)$ then
 9:                 $T_{new} \leftarrow T_{new} \cup \{t, e\}$          // Case 2
10:             else
11:                 $p \leftarrow getCompatiblePrefix(q, t, e)$
12:                 if $p.length > 0$ then
13:                     $T_{new} \leftarrow T_{new} \cup t \cup \{p, e\}$ // Case 3
14:                 else
15:                     $T_{new} \leftarrow T_{new} \cup t \cup \{e\}$   // Case 1
16:         $T_{prev} \leftarrow eliminateIncomplete(T_{new})$
17: return $T_{prev}$
```

Drawbacks. As proven in Appendix A, Algorithm 1 has exponential CPU and memory costs. Indeed, a single event (or, more generally, an event sequence) can be part of several CETs. For example, the event c_1 is part of all CETs in Table 1. The baseline approach replicates such shared event sequences for each CET that contains them. Worse yet, when a new event arrives and is compared to the previously constructed CETs, repeated computations arise since the new event may need to be compared to each shared event sequence within each CET that contains this sequence.

4. THE GRAPH-BASED CET DETECTION

4.1 Compact CET Graph Encoding

The baseline algorithm is inefficient because it does not take common event sequences in CETs into account. To tackle this problem, we propose a compact data structure, called CET graph. The graph prevents event duplication by storing each relevant event exactly once. It avoids repeated computations since each new event is compared to a common event sequence at most once.

Given a CET query and an event stream, the nodes of the graph correspond to events in the stream that are matched by the query, while the edges connect events that are adjacent in a CET. Thus, a path in the CET graph from a node without ingoing edges ("first event") to a node without outgoing edges ("last event") corresponds to one CET.

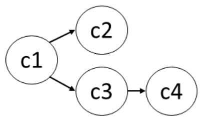
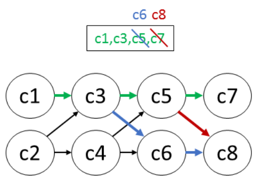
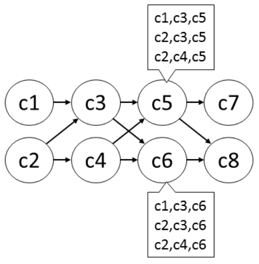


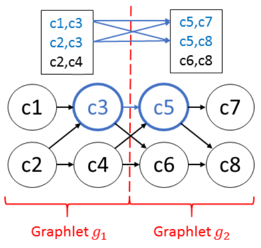
Figure 5: CET graph for Example 5



(a) M-CET algorithm



(b) T-CET algorithm



(c) H-CET algorithm

Figure 6: Approaches to CET detection

	M-CET algorithm	T-CET algorithm	H-CET algorithm
Memory cost = Graph + CET number * CET length	$\Theta(\lvert V\rvert + \lvert E\rvert) + O(\lvert V\rvert)$	$\Theta(\lvert V\rvert + \lvert E\rvert) + O(3^{\frac{\lvert V\rvert}{3}}\lvert V\rvert)$	$\Theta(\lvert V\rvert+\lvert E\rvert)+O(\sum_{i=1}^{k} 3^{\frac{\lvert V_i\rvert}{3}}\lvert V_i\rvert)+O(\lvert V\rvert)$
CPU cost = Graph construction + Graph traversal + CET update	$O(\lvert V\rvert^4)+O(3^{\frac{\lvert V\rvert}{3}}\lvert V\rvert)+ O(3^{\frac{\lvert V\rvert}{3}}\lvert V\rvert)$	$O(\lvert V\rvert^4)+\Theta(\lvert E\rvert)+O(3^{\frac{\lvert V\rvert}{3}})$	$O(\lvert V\rvert^4) + \Theta(\sum_{i=1}^{k} \lvert E_i\rvert) + O(\sum_{i=1}^{k} 3^{\frac{\lvert V_i\rvert}{3}}) + O(3^{\frac{\lvert V\rvert}{3}} k) + O(3^{\frac{\lvert V\rvert}{3}} k)$

Table 2: Cost model

Definition 4. (**CET Graph.**) Given a CET query q and an event stream I, the CET graph $G = (V, E)$ is a directed acyclic graph with a set of vertices V and a set of edges E. The vertices $V \subseteq I$ are events matched by q. For two events $e_1, e_2 \in V$, there is an edge $(e_1, e_2) \in E$ if e_1 and e_2 are adjacent in a CET matched by q.

Example 6. The CET graph for Example 5 is depicted in Figure 5. The paths from the first event c_1 to the last events c_2 and c_4 correspond to two CETs.

The CET graph construction algorithm is analogous to the baseline algorithm (Section 3), except that it updates the CET graph instead of CETs. It has quadratic CPU and memory costs in the number of events (Appendix B). Our cost model in Table 2 and experiments in Section 7.2 demonstrate that the CET graph considerably accelerates CET detection compared to the baseline algorithm and the state-of-the-art techniques, while the graph maintenance cost is negligible compared to the overall costs of CET detection.

4.2 Spectrum of Graph-based CET Detection

Based on the CET graph, the traditional graph traversal algorithms such as Depth First Search (DFS) and Breadth First Search (BFS) can be applied to extract all CETs. These algorithms optimize the usage of critical resources such as memory and CPU. We distinguish between the following two algorithms at the extreme ends of the CPU versus memory trade-off spectrum:

The **M-CET algorithm** is *memory-optimal* since it maintains only one current CET during the DFS traversal. For example, Figure 6(a) shows the current CET after the colored edges have been traversed. Since no intermediate results are stored, the algorithm applies backtracking to find alternative paths through the graph. Thus, an edge is retraversed for each CET.

The **T-CET algorithm** is *CPU-time-optimal* since it stores all CETs found so far during the BFS traversal. For example, Figure 6(b) shows all CETs when events c_5 and c_6 are reached. Since all CETs found so far are stored, no backtracking is necessary. Thus, T-CET traverses each edge exactly once.

Complexity Analysis. Table 2 summarizes the costs of these algorithms further described in Appendix C. In a nutshell, by storing all CETs detected so far, T-CET reduces the CPU costs for graph traversal from exponential to linear and the CPU costs of CET updates by the multiplicative factor $|V|$. On the down side, the memory requirement of T-CET degrades from linear to exponential. Thus, it risks to exceed the available memory when event queries with long windows are evaluated against high-rate event streams.

4.3 Hybrid CET Detection Algorithm

To achieve our goal of minimizing the CPU costs without running out of memory (Definition 3), our CET optimizer partitions the CET graph into smaller graphlets (Section 6.1). Based on the partitioned graph, we now propose the **H-CET algorithm** (*H* for hybrid) that exploits the best of both M-CET and T-CET approaches in a divide-and-conquer fashion. The H-CET algorithm takes two steps:

1) The T-CET algorithm is applied to each graphlet to extract and cache CETs per graphlet.

2) The M-CET algorithm is applied to stitch these partial CETs within graphlets into final CETs across graphlets.

Complexity Analysis. Figure 6(c) illustrates that every edge in the cut between two graphlets requires concatenating the respective CETs within these graphlets to construct final results. This concatenation provokes additional CPU overhead. In other words, the smaller the graphlets, the fewer CETs per graphlet are computed and stored but the higher the CPU overhead of constructing the final CETs from these partial results becomes.

The memory (CPU) costs of H-CET correspond to the sum of the memory requirement to store the CET graph itself (CPU time to construct CET graph), the memory (CPU) costs of T-CET within graphlets, and the memory (CPU) cost of M-CET across graphlets. In Table 2, k denotes the number of graphlets. The cost of graph partitioning, ignored here, will be determined in Section 6.1.

The memory costs of H-CET are exponential in the number of events *per graphlet*, not in the number of events *per query window* as is for T-CET. This can result in several orders of magnitude reduction for high-rate event streams as demonstrated by our experiments in Section 7.2.

Example 7. Partitioning even the small graph in Figure 6(c) reduces the memory costs almost 3-fold. The CPU cost increases by 1.5% due to partitioning.

5. FOUNDATIONS OF CET-GRAPH PARTITIONING PROBLEM

In this section, we study the properties of the graph partitioning search space to efficiently find an optimal CET-graph partitioning plan (Section 6).

Definition 5. (**Optimal CET-Graph Partitioning Plan**.) Let $G = (V, E)$ be a CET graph. A *partitioning plan* p of G into k graphlets is a set of k sub-graphs $p = \{g_1 = (V_1, E_1) \dots, g_k = (V_k, E_k)\}$ such that $k \in \mathbb{N}, 1 \leq k \leq |V|$, $V = V_1 \uplus \dots \uplus V_k$ and $E = E_1 \uplus \dots \uplus E_k \uplus E_c$ where E_c is the set of cut edges that connect events in different graphlets.

Let M be a memory limit, G be a CET graph, and P be the set of all partitioning plans of G. For a partitioning plan $p \in P$, let $cpu(p)$ and $mem(p)$ denote the CPU and memory costs of H-CET for the graph G partitioned by p, respectively. An *optimal partitioning plan* of G is $p_{opt} \in P$ such that $mem(p_{opt}) \leq M$ and $\not\exists p \in P$ with $cpu(p) < cpu(p_{opt})$ and $mem(p) \leq M$.

5.1 CET-Graph Partitioning Search Space

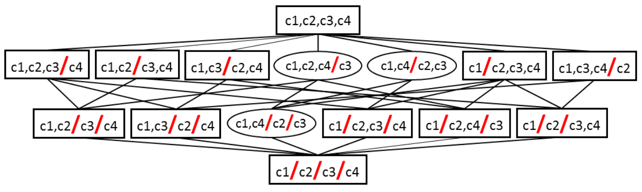


Figure 7: Search space of event-centric partitioning plans

Event-Centric Partitioning Plans. Figure 7 shows the search space of all partitioning plans for the CET graph in Figure 5. Each node in the search space is a partitioning plan. Events in different graphlets are separated by slashes. The number of graphlets per plan increases top to bottom in the search space. That is, the top node has only one graphlet with all events in it. The bottom node has as many graphlets as there are events since each event belongs to a separate graphlet. Generally, the size of the search space is exponential, described by the Bell number which represents the number of different partitions of a set of elements [21]. Thus, streaming graph partitioning is an NP-hard problem [11].

Time-Centric Partitioning Plans. Fortunately, we are not interested in all partitioning plans shown in Figure 7. We aim to partition a CET graph in such a way that we achieve the following goals:

1) ***Correct CET detection***: The H-CET algorithm can extract all CETs from the partitioned graph.

2) ***Expeditious CET detection***: Each final CET can be constructed after visiting each graphlet at most once.

3) ***Feasible memory requirement***: The memory cost of all graphlets satisfies the memory constraint.

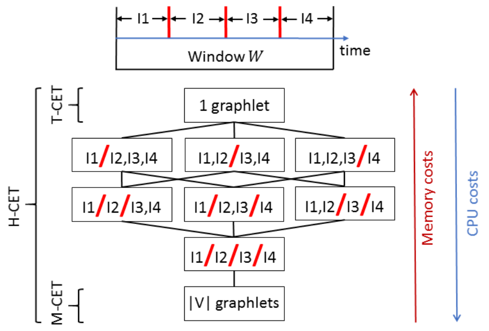


Figure 8: Search space of time-centric partitioning plans

Correct and Expeditious CET Detection. To achieve the first two goals, we consider only those partitioning plans which respect the order of events in a CET. We call such partitioning plans *effective*. A partitioning plan $p = \{g_1 = (V_1, E_1) \dots, g_k = (V_k, E_k)\}$ of a CET graph G is effective if for each CET tr in G that contains an event sequence (v_1, v_2) it holds that if $v_1 \in V_i, v_2 \in V_j$ then $i \leq j$. Effective partitioning plans are indicated by rectangular frames in Figure 7 in contrast to ineffective plans with round frames. An ineffective partitioning plan requires visiting the same graphlet multiple times to construct one final CET. This diminishes the gain of sharing intermediate results and introduces CPU overhead as illustrated by Example 8.

Example 8. Consider the partitioning into graphlets g_1 and g_2 with nodes $\{c_1, c_2\}$ and $\{c_3, c_4\}$ respectively. This plan is effective since the CET $tr_1 = (c_1, c_2)$ is extracted from g_1 and the CET $tr_2 = (c_1, c_3, c_4)$ results by concatenating c_1 from g_1 with (c_3, c_4) from g_2.

In contrast, consider the partitioning into graphlets g'_1 and g'_2 with nodes $\{c_1, c_2, c_4\}$ and $\{c_3\}$ respectively. This plan is ineffective because the CET $tr_2 = (c_1, c_3, c_4)$ requires visiting g'_1 twice: Once to extract c_1 and once to get c_4.

We can exclude ineffective partitioning plans by traversing the CET graph to determine the order of events in all CETs. However, this is an expensive process (Table 2). Instead, we partition the CET graph into non-overlapping consecutive time intervals. All events within a time interval are assigned to the same graphlet (similar to Chunking [34]). *Since consecutive time intervals contain consecutive sub-sequences of CETs, a time-centric partitioning plan is guaranteed to be effective.* The search space remains exponential, however, now in the number of *time intervals* (Figure 8), not is the number of *events* in the window (Figure 7). This results in a substantial reduction for high-rate event streams in which multiple events fall into the same time interval.

Feasible Memory Requirement. To achieve the third goal, we differentiate between first, middle, and last events in a graphlet. An event e is called the *last* (*first*) in its graphlet g if there are no outgoing (incoming) edges from (to) e in the same graphlet g. An event that is neither first nor last is called a *middle* event. For example, events c_3 and c_4 are the last events in graphlet g_1 while events c_5 and c_6 are the first events in graphlet g_2 in Figure 6(c).

To reduce the memory consumption, we store *complete* event trends per graphlet, i.e., an event trend from a first to a last event in the graphlet. No intermediate event trends per graphlet are stored. To guarantee correctness, cut edges may connect only a first and a last event in their respec-

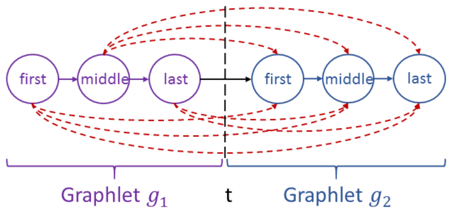


Figure 9: Cut of a CET graph

tive graphlets (Figure 9). The dashed edges are prohibited since they would require storing incomplete event trends per graphlet. To construct final results, for each edge from a last event e_l in a graphlet g_1 to a first event e_f in another graphlet g_2, we concatenate all CETs that end with e_l in g_1 with all CETs that start with e_f in g_2 (Figure 6(c)).

Definition 6. (**Cut of a CET Graph.**) A time point $t \in \mathbb{T}$ partitions a CET graph $G = (V, E)$ into two graphlets g_1 and g_2 such that events are assigned to these graphlets by their time stamps. That is, $\forall e \in V$ if $e.time \leq t$ then $e \in g_1$. Otherwise $e \in g_2$. The time point t is called a *cut* of the graph G if the following conditions hold:
• If a last event in the graphlet g_1 has an outgoing edge to an event e, then e must be the first event in its graphlet g_2.
• If a first event in the graphlet g_2 has an ingoing edge from an event e, then e must be the last event in its graphlet g_1.
• If a middle event in the graphlet g_2 has an ingoing edge from an event e, then e must belong to the graphlet g_2.

A graphlet is called *atomic* if it cannot be cut into smaller non-empty graphlets.

5.2 Monotonicity Properties Across Levels

Based on the cost model in Table 2, we now reveal the cost monotonicity properties across different levels and within the same level of the search space. We use these properties in Section 6.1 to design the branch-and-bound algorithm that effectively prunes the search space to find an optimal CET-graph partitioning plan.

Figure 8 illustrates the cost variations across levels of the search space. There are two extreme cases represented by the top and the bottom nodes. In the top node, all events are in one graphlet. Thus, CET detection takes place only in this graphlet. In other words, H-CET coincides with T-CET. In the bottom node, each event is in a separate graphlet. Thus, CET detection takes place across graphlets only. That is, H-CET coincides with M-CET. To further study the cost variations across levels of the search space, we define the parent-child relationship between nodes.

Definition 7. (**Parent Partitioning Plan.**) Let p be a partitioning plan of a CET graph G. Let p' be a partitioning plan of G that is the same as p except that a graphlet $g_i \in p$ is further partitioned into two graphlets $g_{i1}, g_{i2} \in p'$. Then p is called a parent of p' and p' is called a child of p.

Partitioning a graphlet g_i into two smaller graphlets g_{i1} and g_{i2} reduces the memory requirement because fewer and shorter CETs are stored in the smaller graphlets g_{i1} and g_{i2} than in the original graphlet g_i. In other words, memory costs monotonically decrease from parent to child in the search space (Theorem 2).

THEOREM 2. *[**Memory Cost Monotonicity.**] Let p and p' be partitioning plans of a CET graph G such that p is a parent of p'. The H-CET algorithm in G partitioned by p' has lower memory costs than in G partitioned by p.*

Partitioning a graphlet g_i into two smaller graphlets g_{i1} and g_{i2} introduces additional CPU overhead because CETs in the smaller graphlets g_{i1} and g_{i2} have to be combined to final results. In other words, CPU cost monotonically increases from parent to child in the search space (Theorem 3).

THEOREM 3. *[**CPU Cost Monotonicity.**] Let p and p' be partitioning plans of a CET graph G such that p is a parent of p'. The H-CET algorithm in G partitioned by p' has higher CPU costs than in G partitioned by p.*

The proofs of Theorems 2 and 3 are in Appendix D.

5.3 Monotonicity Properties Within One Level

The number of partitioning plans at one level of the search space is exponential, described by the Stirling number that represents the number of ways to partition n elements into k partitions [21]. To restrict the search at one level, we differentiate between balanced, nearly balanced, and unbalanced graph partitioning plans and compare their costs.

A *balanced partitioning plan* divides a CET graph into graphlets such that the number of events in any two graphlets differs by at most 1. However, such perfect partitioning plan is not always possible since an atomic graphlet cannot be divided (Definition 6). Thus, we define the notion of a *nearly balanced partitioning plan* that allows a graphlet to exceed the ideal size by less than the size of its first or last atomic graphlet. Otherwise a partitioning plan is *unbalanced.* These notions are formally defined in Appendix D.

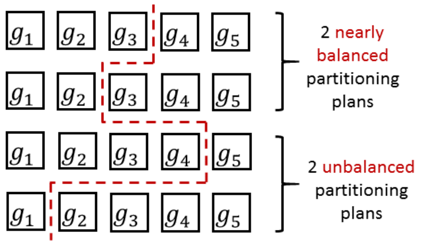

g_1 g_2 g_3 g_4 g_5
g_1 g_2 g_3 g_4 g_5
g_1 g_2 g_3 g_4 g_5
g_1 g_2 g_3 g_4 g_5
2 nearly balanced partitioning plans
2 unbalanced partitioning plans

Figure 10: Nearly balanced vs. unbalanced partitioning plans

Example 9. Assume a CET graph consists of atomic graphlets g_1-g_5 with equal number of events for the sake of simplicity (Figure 10). Assume we want to partition it into 2 graphlets. Obviously, no balanced partitioning plan exists since the atomic graphlet g_3 cannot be divided. However, 2 nearly balanced partitioning plans are possible. One of them assigns g_3 to the first graphlet and the other one assigns g_3 to the second graphlet. Note that the size of these nearly balanced graphlets exceeds the ideal size by less than the size of one atomic graphlet g_3. In contrast to that, if the graphlets are unbalanced, their size exceeds the ideal size by the size of more than one atomic graphlet.

The closer a partitioned CET graph is to balanced, the lower the CPU and memory costs of CET detection are. Intuitively, if a partitioned graph is unbalanced, many CETs are stored in its large graphlets. All of them have to be combined to final results. Hence, the costs of large graphlets dominate the overall costs. In contrast, a balanced partitioning plan has the minimal costs among all partitioning plans with the same number of graphlets (Theorem 4).

THEOREM 4. *[**Cost of Balanced Partitioning Plan.**] Let G be a CET graph and $k \in N$. The H-CET algorithm on* *a balanced partitioning of G into k graphlets has the minimal CPU and memory costs compared to all other partitionings of G into k graphlets.*

If no balanced partitioning plan exists, a nearly balanced partitioning plan has lower costs than unbalanced partitioning plans with the same number of graphlets (Theorem 5).

THEOREM 5. *[**Cost of Nearly Balanced Partitioning Plan**.] Let G be a CET graph and $k \in N$. The H-CET algorithm on a nearly balanced partitioning of G into k graphlets has lower CPU and memory costs than on an unbalanced partitioning of G into k graphlets.*

The proofs of Theorems 4 and 5 are in Appendix D.

6. CET DETECTION OPTIMIZATION

6.1 Branch-and-Bound Partitioning Algorithm

As explained in Section 5.1, the search space of partitioning plans is exponential in the number of atomic graphlets. Thus, we now utilize the cost monotonicity properties identified in Sections 5.2 and 5.3 to define an efficient branch-and-bound CET-graph partitioning algorithm (B&B).

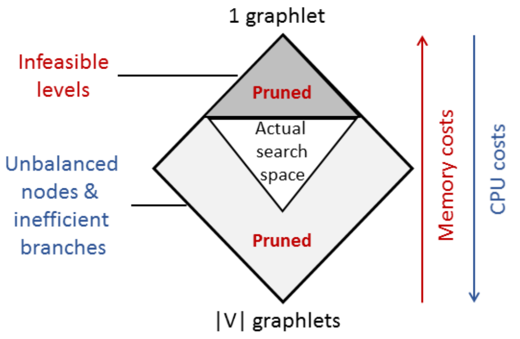


Figure 11: Search space of the branch-and-bound algorithm

The search space is depicted in Figure 8. Each node in it is a CET-graph partitioning plan that is connected to its parents and children (Definition 7). Our B&B algorithm (Algorithm 2) traverses the search space top down. Since memory costs decrease and CPU costs increase from parent to child in the search space, B&B prunes top levels with too high memory costs and brunches with too high CPU costs (Figure 11). In addition, the algorithm disregards unbalanced partitioning plans at each level. B&B consists of the following 2 phases:

1. Level Search Algorithm. Since the memory requirement is high at the top levels, we skip infeasible levels that are guaranteed to contain no partitioning plan that fits into memory. To achieve this goal, the algorithm compares the estimated memory cost of a hypothetical balanced partitioning plan of the input CET graph G at level i to the memory limit M. The memory cost of this hypothetical balanced partitioning plan with i graphlets is the lower bound of the actual minimal memory cost at level i (Theorem 4).

Infeasible level pruning principle: If a balanced partitioning plan with i graphlets does not fit into memory, then no other partitioning plan with i graphlets will. Thus, level i can be safely purged.

The algorithm returns the highest level that satisfies the memory constraint M with i balanced graphlets (lines 1–9).

Algorithm 2 Branch-and-bound partitioning algorithm

```
Input: Graph $G$, Memory limit $M$, Heap $heap$, List $pruned$
Output: Node $solution$
1: $s = 1$, $e = atomicGraphletNumber(G)$   // Level search
2: while $s \leq e$ do
3:     $m = s + (e - s)/2$
4:     Node $b = balanced(G, m)$
5:     if $mem(b) \leq M$ then
6:         $level = m$
7:         $e = m - 1$
8:     else
9:         $s = m + 1$
10: $minCPU = +\infty$   // Node search
11: Node[ ] $nodes = nearlyBalanced(G, level, pruned)$
12: $PushAll(heap, nodes)$
13: while $!isEmpty(heap)$ do
14:     $temp = Pop(heap)$
15:     if $temp.isPruned(pruned)$ then
16:         continue
17:     if $memory(temp) < M$ then
18:         if $cpu(temp) < minCPU$ then
19:             $solution = temp$
20:             $minCPU = cpu(temp)$
21:         $pruned.add(temp.cuts)$
22:     if $temp.level = level$ then
23:         $level++$
24:         $nodes = nearlyBalanced(G, level, pruned)$
25:         $PushAll(heap, nodes)$
26: return $solution$
```

2. Node Search Algorithm. Knowing the minimal number i of required graphlets, this algorithm aims to find an optimal partitioning plan with at least i graphlets. To this end, the algorithm keeps track of the (nearly) balanced nodes to consider in a *heap*. It disregards all other (unbalanced) partitioning plans since according to Theorem 4 (Theorem 5) they are less efficient.

Unbalanced node pruning principle: If a (nearly) balanced partitioning plan with i graphlets exists at a level i, then any other (unbalanced) partitioning plans with i graphlets will have higher CPU and memory costs. Thus, all other nodes at level i can be safely purged.

Starting from the highest feasible level i, in every step the algorithm reduces the memory utilization (Theorem 2) at the expense of increased CPU time (Theorem 3). As soon as a feasible solution is found, we can disregard the descendants of this feasible node since these descendants will have higher CPU costs than their ancestor (lines 11, 15–16, 21, 24).

Inefficient branch pruning principle: If a graph partitioning plan p that satisfies the memory constraint is found, then p's descendants will have higher CPU cost than p. Thus, they can be safely purged.

Since CPU cost increases from parent to child (not from level i to level $i-1$), it is possible that a node at level i has lower CPU than a node at level $i-1$. Thus, the algorithm considers all nearly balanced not pruned nodes. They are shown as white area in Figure 11. The figure conveys that the number of such nodes decreases top down. Node search terminates when it either reaches the lowest level or a level at which all nearly balanced nodes are pruned.

All descendants of a node n will have the cuts that are present in the node n (Figure 8). The cuts of feasible nodes are maintained in the list *pruned*. In lines 11 and 24, we avoid generating nearly balanced nodes from pruned branches. In lines 15–16, we disregard pruned nodes since a node may be pruned after it is added to the *heap*.

B&B keeps track of the best partitioning plan found so far (lines 18–20) and returns it at the end of the node search (line 26). H-CET in the graph partitioned by this plan is guaranteed to be optimal, i.e., have the minimal CPU processing time and stay within the memory limit (Definition 5).

THEOREM 6. ***[Partitioning Plan Optimality.]*** *B&B returns an optimal CET-graph partitioning plan.*

Theorem 6 follows from Theorems 2-5. Details are omitted here to keep the discussion brief.

Complexity Analysis. Let k be the number of levels in the search space, i.e., the number of atomic graphlets in the input CET graph. Then, level search has logarithmic CPU cost in k. It utilizes binary search and computes the memory cost of exactly one perfectly balanced partitioning plan at each level. Thus, the CPU time is $O(log\ k)$.

The node search algorithm has exponential CPU cost in k. The while-loop in lines 13–25 is called at most for every nearly balanced node in the search space, i.e., $O(2^k)$ times. Computing all nearly balanced not pruned partitioning plans at a level i in lines 11 and 24 has exponential CPU cost $O(2^i)$. Checking whether a node is pruned in line 15 is also exponential. All other auxiliary methods have constant CPU complexity. Thus, the CPU cost is $O(2^k)$.

The memory cost of level search is constant. In contrast, the memory cost of node search is exponential in k. It is determined by the size of the list *pruned* and the *heap*. Both are bounded by the maximal number of nearly balanced nodes in one level of the search space, namely, $O(2^k)$.

Despite three pruning principles, the CPU and memory costs of B&B are exponential in the number of atomic graphlets in the worst case.

6.2 Greedy Partitioning Algorithm

In this section, we thus propose a greedy search algorithm for high efficiency. It reuses the level search strategy from Algorithm 2 without change while the node search is simplified. Namely, the greedy search considers only one greedily constructed nearly balanced partitioning plan at level i. If it satisfies the memory constraint, the algorithm returns it as a result. Otherwise, a nearly balanced partitioning plan at level $i-1$ is considered. Thus, the result of the greedy search is guaranteed to be a nearly balanced partitioning plan. However, this result might not be optimal since it might have more graphlets than necessary to satisfy the memory constraint. In other words, the greedy search algorithm does not utilize the entire available memory to speed up CET detection.

Complexity Analysis. Since the greedy search algorithm considers only one (nearly) balanced partitioning plan at a level, its memory cost is constant and its CPU complexity is linear $O(k)$ in the number of atomic graphlets k.

6.3 Graphlet Sharing Across Windows

Our CET executor effectively shares graphlets between overlapping sliding windows. When the window slides, graphlets are categorized into the following three groups (Figure 12): (1) Partially expired graphlet, (2) Shared graphlets among previous and new window, and (3) (Partially) new graphlets. Since graphlets store consecutive stream portions, there can be at most one partially expired graphlet ($I2$ in Figure 12) and at most one partially new graphlet ($I5$). There can be however any number of (completely) expired, shared, or new graphlets. The shared graphlets can be reused across overlapping windows such that repeated CET graph partitioning, construction, traversal, and CET detection in the shared graphlets can be avoided. The construction of partially expired graphlets can be shared between several windows by ignoring expired events. A partially new graphlet must be updated by new events.

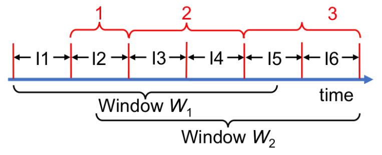


Figure 12: Graphlet sharing technique

Analogously to sharing graphlets between overlapping windows of the *same CET query*, the execution of a workload of *multiple CET queries* can be optimized by sharing the processing of common graphlets across sub-queries. However, sharing graphlets may not always be beneficial due to different window parameters or predicates associated with each CET query. Hence, multi-CET-query optimization is left as subject for future research.

7. PERFORMANCE EVALUATION

7.1 Experimental Setup and Methodology

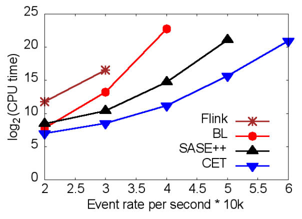
Experimental Infrastructure. We have implemented our CET detection approach in Java with JRE 1.7.0_25. We run our experiments on a cluster machine with Ubuntu 14.04, 16-core 3.4GHz QEMU Virtual CPU, and 128GB of RAM. We execute each experiment three times and report their average results here.

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

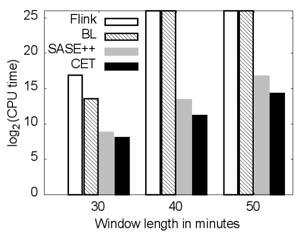
• ***PA: Physical Activity Real Data Set***. The physical activity monitoring real data set [32] (1.6GB) contains physical activity reports for 14 people during 1 hour 15 minutes. There are 20 physical activities which can be classified into active (e.g., running, soccer playing) and passive (e.g., watching TV, working on computer). Other attributes are time stamp, heart rate, temperature and person identifier.

• ***ST: Stock Real Data Set***. The NYSE data set [1] contains 225k transaction records of 10 companies in 1 sector during 12 hours. Each event carries company, sector, and transaction identifiers, volume, price, time stamp, and type (sell or buy). We replicate this data set 10 times with adjusted company, sector, and transaction identifiers such that the resulting data set contains transactions for 110 companies in 11 sectors. No other attributes except identifiers were changed in the replicas compared to the original.

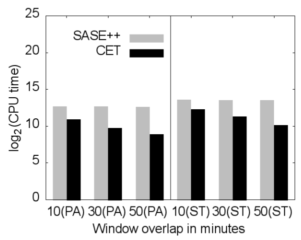
• ***FT: Financial Transaction Data Set***. We developed an event stream generator that creates check deposit events for 3 days. Each event carries time stamp in seconds, source bank, destination bank, and status (covered or not). The generator allows us to vary event rate per second R, event compatibility C, and predicate selectivity S. Given these 3 parameters, each of $R*S$ events with time stamp t is



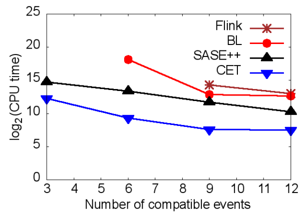
(a) Event rate (FT)



(b) Window length (ST)

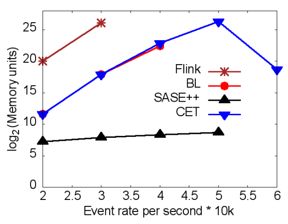


(c) Window overlap (PA,ST)

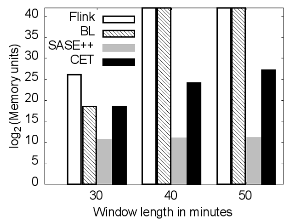


(d) Event compatibility (FT)

Figure 13: CPU costs of CET detection algorithms



(a) Event rate (FT)



(b) Window length (ST)

Figure 14: Memory costs of CET detection algorithms

compatible with C events with earlier time stamps than t. Unless stated otherwise, the default event rate is 3.5k events per second and the default event compatibility is 3.

CET Queries. We evaluate the workload of 10 appropriate CET queries against each of these event streams. These queries are variations of queries Q_1, Q_2, and Q_3 in Section 1. They differ by event patterns and predicates.

Methodology. To show the efficiency of our CET approach, we compare it to the baseline algorithm (BL, Section 3) and SASE++ [38] because both approaches support Kleene closure computation over event streams under the skip-till-any-match semantics. To achieve fair comparison, we have implemented SASE++ [38] including its optimization techniques [10, 37] on top of our platform. In a nutshell, SASE++ stores each matched event e in a stack and computes pointers to e's previous events in a CET. At the end of each window, DFS-based CET extraction is applied to these stacks. Section 8 contains a more detailed discussion.

We also compare our approach to the open-source streaming system Apache Flink [7] that supports event pattern matching under skip-till-any-match. We express our CET queries using Flink operators. Flink guarantees that events are processed in parallel but in-order by their time stamps. Each event is processed exactly once. However, like most industrial streaming systems [2, 3, 4], the Kleene operator is not explicitly supported by Flink. It is currently being developed for the next release [8]. To express our CET queries in Flink, each CET query q with a Kleene operator is flattened into a set of event sequence queries that cover all possible lengths of event trends matched by the query q. We vary the event rate, event compatibility, window length, and window overlap size in these experiments.

To demonstrate the effectiveness of our branch-and-bound graph partitioning (B&B, Section 6.1), we compare it to the exhaustive (Exh) and greedy search (Greedy, Section 6.2) by varying the event rate and window length. Additional experiments comparing the performance of CET-graph partitioning algorithms are in Appendix E.

Under certain parameter settings, *Flink*, *BL*, *SASE++*, and *Exh* are unable to produce results within several hours. Thus, the results are either discontinued in the line charts or highlighted by bars with maximal height in the bar charts.

Since event rate and event compatibility cannot be varied for the real data sets, we ran these experiments on the FT data set. We vary window length and window overlap size on the PA and ST real data sets.

Metrics. We measure two metrics common for streaming systems, namely, average *CPU time* and *memory requirement* per window [10, 23, 30, 31, 38]. Average CPU time is measured in milliseconds as the sum of total elapsed time in all windows divided by the number of windows. Average memory requirement is measured as the sum of peak memory units in all windows divided by the number of windows. For CET detection algorithms, an event in a stored CET is a memory unit. Additionally, a vertex or an edge in our CET graph as well as an event or a pointer in a SASE++ stack [38] are also memory units. The size of one memory unit is 8 bytes. For CET graph partitioning algorithms, a node in the search space is a memory unit. Its size is 20 bytes * k where k is the number of graphlets.

7.2 CET Detection Algorithms

In Figures 13–14, we compare our CET approach to other CET detection solutions.

Varying Number of Events per Window. CPU costs of all approaches grow exponentially with an increasing stream rate and window length (Figures 13(a)–13(b)). For large numbers of events per window, Flink, BL, and SASE++ do not terminate within several hours. These approaches do not scale because they construct all CETs from scratch – provoking repeated computations. In contrast, our CET approach stores partial CETs and reuses them while constructing final CETs. Thus, CET is 42–fold faster than SASE++ if stream rate is 50k events per second and 2 orders of magnitude faster than Flink if stream rate is 3k events per second. Flink is even slower than BL in all experiments because Flink expresses each Kleene query as a set of event sequence queries. Thus, the workload of Flink is considerably higher than other approaches tailored for Kleene computation.

Figure 14 demonstrates that the CET approach adapts to the available memory. That is, it partitions the graph in such a way that CET detection in the partitioned graph is guaranteed to stay within the given memory limit. If the stream rate is below 50k events per second or the window is shorter than 50 minutes, H-CET coincides with T-CET (Section 4.3). Otherwise, T-CET would run out of memory since it stores CETs for the *whole graph*. Instead, our CET approach partitions the graph into smaller graphlets and stores CETs *per each graphlet*.

CET and BL have almost the same memory cost (Figure 14). Thus, the amount of memory required for the CET graph is negligible compared to the amount of memory required for CET storage. We thus conclude that CET graph is worth maintaining to compactly capture all CETs.

SASE++ requires 5 orders of magnitude less memory than the CET approach if event rate is 50k events per second since SASE++ stores events in stacks and pointers to their previous events. When the window ends, these pointers are traversed using DFS to extract all CETs. Hence, SASE++ is conceptually close to the M-CET algorithm, namely, it is lightweight but slow (Figures 13 and 14).

Flink requires 2 orders of magnitude more memory than CET if stream rate is 3k events per second because Flink stores all trends of all possible lengths since it expresses Kleene closure by a set of sequence queries.

We also ran the experiment on the PA data set varying window lengths. These charts have similar trends as in Figures 13(b) and 14(b). Thus, they are omitted here due to the space constraints.

Varying Window Overlap. Figure 13(c) measures the effect of the graphlet sharing technique. As the window overlap increases, the CPU time of the CET approach decreases exponentially. This is explained by the fact that CET graph construction, partitioning and CET detection within graphlets is shared between overlapping windows. The larger the overlap, the more the gain of sharing. In contrast, the average CPU time per window of SASE++ remains fairly constant since no intermediate results are shared among overlapping windows. Our CET approach outperforms SASE++ 13–fold if window overlap is 50 minutes. Since Flink and BL do not keep up with such long windows, they are omitted in Figure 13(c).

Varying Event Compatibility. Figure 13(d) experimentally confirms Theorem 1. Namely, the number of CETs (and thus the cost of CET detection) is maximal if event compatibility is 3. When event compatibility increases, the CPU time of all algorithms decreases exponentially. When event compatibility is 3, our CET approach is up to 5.4–fold faster than SASE++ since it avoids repeated computations by caching and reusing intermediate results. Neither Flink nor BL can produce any results in this worst case scenario.

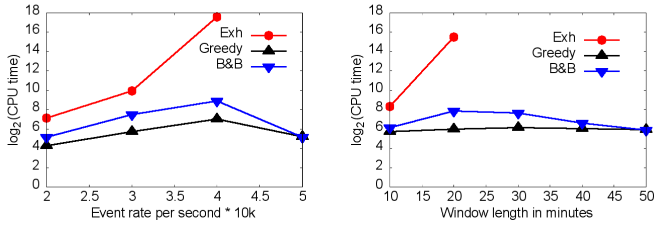
7.3 CET-Graph Partitioning Algorithms

Varying the Number of Events per Window. In Figures 15–16, we compare the performance of the CET graph partitioning algorithms used by our optimizer.

The search space for an optimal CET graph partitioning plan has a lattice shape (Figure 8). Thus, the B&B optimizer performs best if it searches the top or the bottom of the search space where the number of nodes is relatively small. This happens if the memory constraint is loose or tight for the given number of events (first and last cases on the X-axis). Otherwise, B&B searches the middle of the search space where the number of nodes is large. This causes higher CPU and memory costs (middle cases on the X-axis).

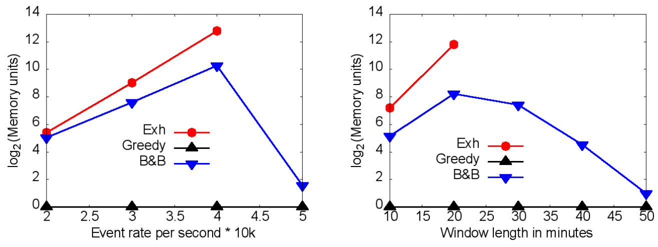
The CPU and memory costs of the exhaustive algorithm grow exponentially with an increasing event rate or window length. Due to 3 effective pruning principles, our B&B is up to 2 orders of magnitude faster if stream rate is 40k events per second and requires up to 12–fold less memory if window length is 20 minutes than the exhaustive optimizer yet B&B returns an optimal CET graph partitioning plan.

The greedy optimizer has fairly constant CPU and memory costs. It is up to 3–fold faster and requires up to 3 orders of magnitude less memory than B&B if the stream rate is 40k events per second.



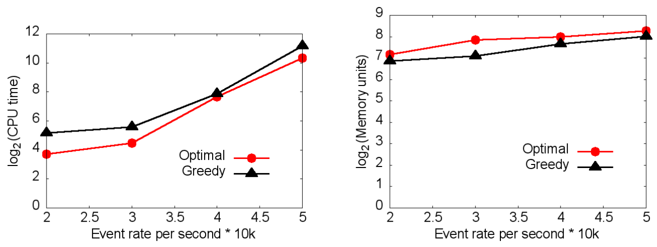
(a) Event rate (FT) (b) Window length (PA)

Figure 15: CPU costs of CET graph partitioning algorithms



(a) Event rate (FT) (b) Window length (PA)

Figure 16: Memory costs of CET graph partitioning algorithms



(a) CPU time (FT) (b) Memory cost (FT)

Figure 17: Partitioned graph quality

Partitioned Graph Quality. The greedy optimizer tends to return a sub-optimal CET graph partitioning plan because it considers only one nearly balanced partitioning plan per level until it finds a plan that satisfies the memory constraint. Thus, the greedy search tends to partition a CET graph more than necessary. In other words, it does not utilize the entire memory resources to speed up CET detection. Figure 17 compares the CPU and memory costs of CET detection in an optimally-partitioned versus a greedily-partitioned CET graph. A greedy partitioning plan is 2.8–fold slower than an optimal partitioning solution if event rate is 20k events per second.

8. RELATED WORK

Complex Event Processing. Most existing approaches either compute event sequences of fixed length [23, 30, 31] or assume a known upper bound on the length of event sequences [12]. They do not support event queries with Kleene patterns. Thus, their expressive power is limited.

Several approaches [10, 16, 26, 37, 38] support Kleene closure computation over event streams. However, ZStream [26] and Cayuga [16] do not support Kleene closure computation under skip-till-any-match. In contrast, SASE [10, 37] optimize Kleene queries under various semantics. SASE adapts a Finite State Automaton (FSA)-based execution paradigm. Each event query is translated into an FSA. Each state of an FSA is associated with a stack with single events stored in each stack. To speed up stack traversal, each event is augmented with a pointer to its previous event in a match.

SASE^{++} [38] further optimizes query evaluation by breaking it into pattern matching and result construction phases. Pattern matching computes the main runs of an FSA (equivalent to CETs) with certain predicates postponed. Result construction derives all matches of the Kleene pattern by applying the postponed predicates to remove non-viable runs. The pattern matching phase in SASE^{++} simply reuses the FSA-based approach from SASE [10, 37] without any optimization. Computing matches for the main runs incurs repeated computations since the same common event subsequence is re-traversed for each match that contains it. In contrast, we introduce a compact CET graph to capture all CETs and partition the graph to trade off between CPU and memory costs of CET detection.

Recursive Queries. Kleene closure computation has been studied in recursive query processing as well [14, 15, 17, 24, 25, 27, 28, 33]. However, most existing solutions either focus on achieving high expressive power for recursive queries [28], or ensuring correctness of such queries [14, 24]. These approaches incur additional execution costs for supporting more expressive queries than CEP queries. The optimization techniques proposed in [15, 17, 25, 27] neither support the skip-till-any-match semantics nor take memory constraints into consideration. Therefore, none of the existing solutions fully tackles the challenges of event queries with Kleene closure we target.

Streaming Graph Partitioning approaches [29, 34, 36] consider dynamic graphs in which vertexes or edges or both change over time. These approaches aim at a balanced one-pass dynamic graph partitioning algorithm. Since the problem is NP-hard [11], several approximation algorithms [11, 22] and heuristics [13, 18, 20] have been developed.

While these strategies also aim to find a balanced graph partitioning, they focus on optimization criteria that are distinct from our problem. Namely, they minimize the total number [18, 20, 34, 36] or weight [11] of cut edges. If we were to apply their method and minimize the number of cut edges, it would not necessarily reduce the cost of CET detection since one cut edge may need to be traversed multiple times. Potentially, we could address this problem by defining an edge weight as the number of edge traversals. However, CETs within and across graphlets would have to be computed to determine the number of edge traversals. This risks making our CET-graph partitioning algorithm more expensive than the CET detection algorithm itself.

Furthermore, existing graph partitioning approaches do not take the order of events in a CET into consideration – which is crucial to CET detection. Indeed, if a graph partitioning algorithm is oblivious to the event order, then correct CET detection that visits each graphlet at most once for each CET would not be possible (Section 5.1). Also, graphlets could not be shared between overlapping sliding windows (Section 6.3). Lastly, the properties of the graph partitioning search space we uncover are specific to the cost model for CET detection, and thus lead us to unique pruning principles of our B&B algorithm.

9. CONCLUSIONS

To the best of our knowledge, we are the first to guarantee optimal CPU time of complete event trend detection over high-rate event streams given limited memory. We compactly encode the relevant events and their CET relationships into the CET graph. To trade-off between CPU and memory costs, our CET optimizer partitions the CET graph into time-centric graphlets. To this end, our CET optimizer leverages the cost monotonicity properties that we have discovered to find an optimal CET-graph partitioning plan in practical time. Given the partitioned CET graph, our CET executor caches CETs per each graphlet and reuses them while constructing CETs within one window and between overlapping sliding windows. Our experimental results demonstrate that our CET approach is up to 42-fold faster than the state-of-the-art solutions in various scenarios.

Acknowledgments

We sincerely thank Prof. David Maier for his suggestions to improve this paper. This work was supported by NSF grants IIS 1018443, IIS 1343620, IIS 1560229, and CRI 1305258.

10. REFERENCES

[1] Stock trade traces. http://davis.wpi.edu/datasets/Stock_Trace_Data/.
[2] Esper. http://www.espertech.com/, 2015. [Online; accessed 20-April-2015].
[3] Storm. https://storm.apache.org/, 2015. [Online; accessed 9-January-2015].
[4] StreamInsight. https://technet.microsoft.com/en-us/library/ee362541%28v=sql.111%29.aspx, 2015. [Online; accessed 20-April-2015].
[5] The Press Enterprise. http://www.pe.com/articles/checks-694614-people-bank.html, 2015. [Online; accessed 6-October-2015].
[6] Wikipedia. https://en.wikipedia.org/wiki/Check_kiting, 2015. [Online; accessed 6-October-2015].
[7] Apache Flink. https://flink.apache.org/, 2016. [Online; accessed 14-October-2016].
[8] Apache Flink Forum. https://issues.apache.org/jira/browse/FLINK-3318, 2016. [Online; accessed 14-October-2016].
[9] Boulder Community Health. http://www.bch.org/cardiac-care/arrhythmia-electrophysiology.aspx, 2016. [Online; accessed 6-July-2016].
[10] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, pages 147–160, 2008.
[11] K. Andreev and H. Räcke. Balanced graph partitioning. In *SPAA*, pages 120–124, 2004.
[12] C. Balkesen, N. Dindar, M. Wetter, and N. Tatbul. RIP: Run-based intra-query parallelism for scalable Complex Event Processing. In *DEBS*, pages 3–14, 2013.
[13] S. T. Barnard. PMRSB: Parallel Multilevel Recursive Spectral Bisection. In *Supercomputing*, 1995.
[14] B. Chandramouli, J. Goldstein, and D. Maier. On-the-fly progress detection in iterative stream queries. *VLDB*, 2(1):241–252, Aug. 2009.
[15] Y. Chen, S. B. Davidson, and Y. Zheng. An Efficient XPath Query Processor for XML Streams. In *ICDE*, pages 1–12, 2006.
[16] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. White. Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422, 2007.
[17] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004.

[18] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Supercomputing*, 1995.

[19] M. Hirzel. Partition and compose: Parallel Complex Event Processing. In *DEBS*, pages 191–200, 2012.

[20] G. Karypis and V. Kumar. Multilevel graph partitioning schemes. In *Parallel Processing*, pages 113–122, 1995.

[21] M. Klazar. Bell numbers, their relatives, and algebraic differential equations. *J. Comb. Theory, Ser. A*, 102(1):63–87, 2003.

[22] R. Krauthgamer, J. S. Naor, and R. Schwartz. Partitioning graphs into balanced components. In *SODA*, pages 942–949, 2009.

[23] 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.

[24] M. Liu, N. E. Taylor, W. Zhou, Z. G. Ives, and B. T. Loo. Recursive computation of regions and connectivity in networks. In *ICDE*, pages 1108–1119, 2009.

[25] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A transducer-based XML query processor. In *VLDB*, pages 227–238, 2002.

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

[27] B. Mozafari, K. Zeng, and C. Zaniolo. From regular expressions to nested words: Unifying languages and query execution for relational and XML sequences. *VLDB*, 3(1-2):150–161, Sept. 2010.

[28] B. Mozafari, K. Zeng, and C. Zaniolo. High-performance complex event processing over XML streams. In *SIGMOD*, pages 253–264, 2012.

[29] J. Nishimura and J. Ugander. Restreaming graph partitioning: Simple versatile algorithms for advanced balancing. In *KDD*, pages 1106–1114, 2013.

[30] M. Ray, C. Lei, and E. A. Rundensteiner. Scalable pattern sharing on event streams. In *SIGMOD*, pages 495–510, 2016.

[31] M. Ray, E. A. Rundensteiner, M. Liu, C. Gupta, S. Wang, and I. Ari. High-performance complex event processing using continuous sliding views. In *EDBT*, pages 525–536, 2013.

[32] A. Reiss and D. Stricker. Creating and benchmarking a new dataset for physical activity monitoring. In *PETRA*, pages 40:1–40:8, 2012.

[33] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo. Big data analytics with Datalog queries on Spark. In *SIGMOD*, pages 1135–1149, 2016.

[34] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *KDD*, pages 1222–1230, 2012.

[35] J. Stewart. *Calculus: Early Transcendentals*. Thompson Brooks/Cole, 8th edition, 2015.

[36] C. E. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. Technical report, 2012.

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

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

APPENDIX

A. COMPLEXITY OF CET DETECTION

THEOREM 1. ***[Maximal Number of CETs.]*** *$3^{\frac{n}{3}}$ CETs can be constructed from n events in the worst case.*

PROOF. Let $x \in \mathbb{N}$ be the number of events that all current CETs are compatible with. We first show that the number of CETs to be constructed for a given set I of n events is maximal when the number of events x_i in I that each CET is compatible with is identical. Then the maximum number of CETs is given by $maximize \prod x_i$, subject to $\sum x_i = n$. The method of Lagrange multiplier in mathematical optimization [35] introduces an auxiliary function $\mathcal{L}(x_i, \lambda) = \prod x_i + \lambda \cdot \sum x_i$. Solving the following equation $\bigtriangledown_{x_i,\lambda}\mathcal{L}(x_i, \lambda) = 0$, we have $x_1 = ... = x_i = x_{i+1} = ... = x_{n/x}$. This proves that the maximum number of CETs that can be constructed from n events is $y = x^{\frac{n}{x}}$.

Now our goal is to determine the global maximum of this continuous function on the interval $[1, n]$. Below we derive the critical value of x following the standard approach in [35]. First we take the logarithm of both sides:

$$\ln y = \ln x^{\frac{n}{x}} = \frac{n}{x} \ln x.$$

We then differentiate both sides:

$$\frac{y^t}{y} = (\frac{n}{x})' \ln x + (\ln x)' \frac{n}{x} = \frac{0 \cdot x - n \cdot 1}{x^2} \ln x + \frac{n}{x^2} = \frac{n}{x^2}(1 - \ln x).$$

Lastly, we multiply by y and substitute y with $\frac{n}{x} \ln x$.

$$y^t = y\frac{n}{x^2}(1 - \ln x) = x^{\frac{n}{x}}\frac{n}{x^2}(1 - \ln x) = x^{\frac{n}{x}-2}n(1 - \ln x).$$

We have $y^t = 0$ if $x = e$ (e is Euler's number here). Since $x \in \mathbb{N}$, we round x to 3. Thus, $y = 3^{\frac{n}{3}}$ is the absolute maximum value of the function. $\Box$

Complexity Analysis of the Baseline CET Detection Algorithm. Let n be the number of events in the query window, y be the number of CETs, l be the maximal length of a CET, and p be the number of predicates in the query q. The for-loop in lines 2–16 in Algorithm 1 is called n times. The for-loop in lines 7–15 is executed for each CET, y times. The $isCompatible(q, t, e)$ method has the CPU cost $O(p)$ since each predicate needs to be executed to conclude compatibility. The $getCompatiblePrefix(q, t, e)$ method has the CPU cost $O(lp)$ since in the worst case the compatibility is checked for each possible prefix of the trend tr. Finally, the $eliminateIncomplete(T_{new})$ method has the CPU cost $O(y^2l)$ since each event trend is compared to all other event trends and the cost of one comparison is $O(l)$. Altogether, the CPU cost is $O(nyp^2l + ny^2l)$.

According to Theorem 1, $y = 3^{\frac{n}{3}}$ in the worst case and thus y is the dominant factor in the overall CPU complexity. Thus, the execution time of Algorithm 1 is exponential in the number of events n in the window:

$$CPU_{BL} = O(n(yp^2l + y^2l)) = O(n(3^{\frac{n}{3}}p^2l + 3^{\frac{2n}{3}}l)).$$

The memory consumption of Algorithm 1 is also exponential in n since all CETs are stored:

$$Mem_{BL} = O(yl) = O(3^{\frac{n}{3}}l).$$

B. CET GRAPH CONSTRUCTION

The CET graph construction algorithm in Algorithm 3 avoids comparing each new matched event with each previously matched event by maintaining the set V_{last} of all last events in the CETs constructed so far. Each new matched event e becomes a vertex in the CET graph (line 5). The edges are drawn as follows: The event e has no ingoing edges if it is incompatible with all events in the graph. Then e starts a new CET (Case 1). If the event e is compatible with a last event e_l of a CET tr, an edge from e_l to e is built and e becomes the last event of the trend tr (Case 2). If the event e is incompatible with the last event e_l of a CET tr, we backtrack along the edges from this event e_l until we find a compatible event which is the last event in the prefix of the trend tr compatible with e, if any (Case 3).

Complexity Analysis. Let q be a CET query, n be the number of events in the window of q, and p be the number

Algorithm 3 CET graph construction algorithm

```
Input: CET query $q$, input event stream $I$
Output: CET graph $G$
1: $V_{last} \leftarrow \emptyset,\ V \leftarrow \emptyset,\ E \leftarrow \emptyset,\ G \leftarrow (V, E)$
2: for all $e \in I$ such that $e.isMatchedBy(q)$ do
3:     $V \leftarrow V \cup e$
4:     if $V_{last} = \emptyset$ then
5:         $V_{last} \leftarrow e$                                  // Case 1
6:     else
7:         for all $e_l \in V_{last}$ do
8:             if $isCompatible(q, e_l, e)$ then
9:                 $E \leftarrow E \cup (e_l, e)$                   // Case 2
10:                $V_{last} \leftarrow (V_{last} - e_l) \cup e$
11:            else
12:                $C \leftarrow getLatestCompEvents(q, e_l, e)$
13:                for all $e_c \in C$ do
14:                    $E \leftarrow E \cup (e_c, e)$               // Case 3
15:                $V_{last} \leftarrow V_{last} \cup e$            // Case 1
16: return $G$
```

of predicates in q. The outermost for-loop is called for each matched event, i.e., $O(n)$ times. The following two cases are possible for a new event e:

• e is compatible with each last event. The *isCompatible*(q, e_l, e) method has the CPU cost $O(p)$ since each predicate needs to be executed to conclude compatibility. This method is called at most n times since in a extreme case each event in the graph is a last event.

• e is compared to earlier events to find latest compatible events. The *getLatestCompEvents*(q, e_l, e) method returns the set of all events that are compatible with e and are latest in the event trends which end with e_l. The CPU costs are $O(np)$ since in the worst case the compatibility with each event in the graph has to be checked.

Altogether, the CPU costs are quadratic in n: $O(n^2p) = O(n^2)$ since $p < n$ for high-rate event streams. The memory costs are also quadratic in n because the number of nodes is $O(n)$, while the number of edges is $O(n^2)$ if each event is compatible with all other events in the graph.

C. CPU- AND MEMORY-OPTIMIZED CET DETECTION ALGORITHMS

Complexity Analysis. In terms of memory utilization, both algorithms store the CET graph itself, that is, $|V|$ vertexes and $|E|$ edges. Storing the current CET requires at most $|V|$ space since in an extreme case all events build one CET. Thus, the memory cost of the M-CET algorithm is:

$$mem_{M-CET} = \Theta(|V| + |E|) + O(|V|).$$

The memory requirement for all CETs in the extreme case can be bounded by multiplying the maximal number of CETs $3^{\frac{|V|}{3}}$ by the maximal length of a CET $|V|$. Hence, the memory cost of the T-CET algorithm is:

$$mem_{T-CET} = \Theta(|V| + |E|) + O(3^{\frac{|V|}{3}}|V|).$$

Both algorithms construct the CET graph in $O(|V|^4)$ time (Appendix B). The M-CET algorithm stores only one current CET and thus performs at most $3^{\frac{|V|}{3}}|V|$ edge traversals and at most $3^{\frac{|V|}{3}}|V|$ CET updates. Altogether, the CPU cost of the M-CET algorithm thus is:

$$cpu_{M-CET} = O(|V|^4) + O(3^{\frac{|V|}{3}}|V|) + O(3^{\frac{|V|}{3}}|V|).$$

In contrast, the T-CET algorithm stores all CETs found so far and thus traverses an edge exactly once. In addition, it updates the current CETs (Figure 6(b)). According to Theorem 1, the number of CETs is maximal if there are $\frac{|V|}{3}$ groups with 3 events at each group. Thus, the T-CET algorithm performs $O(\sum_{i=1}^{\frac{|V|}{3}} 3^i) = O(3^{\frac{|V|}{3}})$ CET updates in total. Its CPU cost is:

$$cpu_{T-CET} = O(|V|^4) + \Theta(|E|) + O(3^{\frac{|V|}{3}}).$$

D. PROPERTIES OF THE CET GRAPH PARTITIONING SEARCH SPACE

THEOREM 6. *[**Partitioning Plan Optimality**.] B&B returns an optimal CET-graph partitioning plan.*

PROOF. Since the memory costs for storing the CET graph are the same for all algorithms in Table 2, we ignore this cost factor in the following. Let $G = (V, E)$ be a CET graph such that $V = V_1 \uplus V_2$. The memory cost of T-CET in the non-partitioned graph G is:

$$O(3^{\frac{|V_1|+|V_2|}{3}}(|V_1| + |V_2|)) = O(3^{\frac{|V_1|+|V_2|}{3}}|V_1| + 3^{\frac{|V_1|+|V_2|}{3}}|V_2|).$$

The memory cost of H-CET in the partitioned graph $p2 = \{g_1 = (V_1, E_1), g_2 = (V_2, E_2)\}$ is:

$$O(3^{\frac{|V_1|}{3}}|V_1| + 3^{\frac{|V_2|}{3}}|V_2|) + O(|V_1| + |V_2|) =$$
$$O((3^{\frac{|V_1|}{3}} + 1)|V_1| + (3^{\frac{|V_2|}{3}} + 1)|V_2|).$$

The latter is considerably smaller than the former since:

$$(3^{\frac{|V_1|}{3}} + 1)|V_1| \leq 3^{\frac{|V_1|+|V_2|}{3}}|V_1|, (3^{\frac{|V_2|}{3}} + 1)|V_2| \leq 3^{\frac{|V_1|+|V_2|}{3}}|V_2|.$$

Assume that the graphlet $g_2 = (V_2, E_2)$ is further partitioned into two smaller graphlets $g_{21} = (V_{21}, E_{21})$ and $g_{22} = (V_{22}, E_{22})$. Then the memory cost of the H-CET algorithm in the partitioned graph $p3 = \{g_1, g_{21}, g_{22}\}$ is:

$$O(3^{\frac{|V_1|}{3}}|V_1| + 3^{\frac{|V_{21}|}{3}}|V_{21}| + 3^{\frac{|V_{22}|}{3}}|V_{22}|) + O(|V_1| + |V_{21}| + |V_{22}|).$$

Memory cost of CET detection in $p3$ is lower than the memory cost of CET detection in $p2$ since:

$$3^{\frac{|V_2|}{3}}|V_2| = 3^{\frac{|V_{21}|+|V_{22}|}{3}}(|V_{21}| + |V_{22}|) = 3^{\frac{|V_{21}|+|V_{22}|}{3}}|V_{21}|+$$
$$3^{\frac{|V_{21}|+|V_{22}|}{3}}|V_{22}| \geq 3^{\frac{|V_{21}|}{3}}|V_{21}| + 3^{\frac{|V_{22}|}{3}}|V_{22}|. \quad \Box$$

THEOREM 3. *[**CPU Cost Monotonicity**.] Let p and p' be partitioning plans of a CET graph G such that p is a parent of p'. The H-CET algorithm in G partitioned by p' has higher CPU costs than in G partitioned by p.*

PROOF. Since the CPU time for constructing the CET graph is the same for all algorithms in Table 2, we ignore this cost factor in the following. Let $G = (V, E)$ be a CET graph such that $V = V_1 \uplus V_2$ and $E = E_1 \uplus E_2 \uplus E_c$. The CPU cost of T-CET in the non-partitioned graph G is:

$$O(|E_1| + |E_2| + \underline{|E_c|}) + \underline{O(3^{\frac{|V_1|+|V_2|}{3}})}$$

The CPU cost of H-CET using the partitioning plan $p2 = \{g_1 = (V_1, E_1), g_2 = (V_2, E_2)\}$ is:

$$O(|E_1| + |E_2|) + \underline{O(3^{\frac{|V_1|}{3}} + 3^{\frac{|V_2|}{3}})} + \underline{O(3^{\frac{|V_1|+|V_2|}{3}} * 4)}$$

The factors that differentiate costs are underlined. They reveal the trade-off between the CPU costs within and across graphlets. Namely, a large number of small graphlets reduces the CPU cost for CET update within graphlets. The *CPU overhead within graphlets* is the difference between the CPU cost within graphlets in a partitioned graph and the CPU cost within graphlets in a non-partitioned graph:

$$CPU_{within} = O(\sum_{i=1}^{k} 3^{\frac{|V_i|}{3}} - |E_c|)$$

On the other hand, a small number of large graphlets reduces the CPU cost for traversing edges across graphlets. The *CPU overhead across graphlets* is the difference between the CPU costs across graphlets in a partitioned graph and the CPU cost across graphlets in a non-partitioned graph:

$$CPU_{across} = O(3^{\frac{|V|}{3}}(4k - 1))$$

Since $\forall k \in \mathbb{N},\ k \geq 2.\ CPU_{across} \geq CPU_{within}$, we conclude that the CPU cost degrades due to partitioning. $\square$

Definition 8. (**Balanced, Nearly Balanced and Unbalanced CET-Graph Partitioning Plans.**) A partitioning plan p of a CET graph $G = (V, E)$ into k graphlets is called *balanced* if $\forall i \in \mathbb{N},\ 1 \leq i \leq k,\ |V_i| \leq \lceil \frac{|V|}{k} \rceil$ holds.

Let $g_i = (V_i, E_i)$ be a graphlet in a partitioning plan p such that $|V_i| > \lceil \frac{|V|}{k} \rceil$. Let $g_f = (V_f, E_f)$ be the first and $g_l = (V_l, E_l)$ be the last atomic graphlet in g_i. A partitioning plan p is called *nearly balanced* if two conditions hold:

- If $i > 1$ then $|V_i| < \lceil \frac{|V|}{k} \rceil + |V_f|$ and
- If $i < k$ then $|V_i| < \lceil \frac{|V|}{k} \rceil + |V_l|$.

Otherwise, a partitioning plan p is called *unbalanced.*

THEOREM 4. ***[Cost of Balanced Partitioning Plan.]*** *Let G be a CET graph and $k \in N$. The H-CET algorithm on a balanced partitioning of G into k graphlets has the minimal CPU and memory costs compared to all other partitionings of G into k graphlets.*

PROOF. Since the memory cost for storing one CET across graphlets and the CPU cost for computing CETs across graphlets are independent from the quality of partitioning a CET graph into k graphlets, we ignore these costs below.

We now prove that moving one vertex from one balanced graphlet to another balanced graphlet increases both memory and CPU costs. Let $p_b = (g_{b1} = (V_{b1}, E_{b1}),\ g_{b2} = (V_{b2}, E_{b2}))$ be a balanced partitioning plan of a CET graph. Assume for simplicity that $|V_{b1}| = |V_{b2}| = v$ holds. Let $p_o = (g_{o1} = (V_{o1}, E_{o1}), g_{o2} = (V_{o2}, E_{o2}))$ be a partitioning plan which is the same as p_b except that one vertex is moved from the first graphlet into the second one, i.e., $|V_{o1}| = v - 1$ and $|V_{o2}| = v + 1$. Then, the memory cost of CET detection in these two partitioning plans is computed as follows:

$$mem(p_b) = 2(\sqrt[3]{3})^v v$$
$$mem(p_o) = (\sqrt[3]{3})^{v-1}(v-1) + (\sqrt[3]{3})^{v+1}(v+1)$$
$$= \frac{(\sqrt[3]{3})^v v}{\sqrt[3]{3}} - \frac{(\sqrt[3]{3})^v}{\sqrt[3]{3}} + (\sqrt[3]{3})^v \sqrt[3]{3}v + (\sqrt[3]{3})^v \sqrt[3]{3}.$$

We verify the fact that $mem(p_b) < mem(p_o)$ as follows:

$$2(\sqrt[3]{3})^v v < \frac{(\sqrt[3]{3})^v v}{\sqrt[3]{3}} - \frac{(\sqrt[3]{3})^v}{\sqrt[3]{3}} + (\sqrt[3]{3})^v \sqrt[3]{3}v + (\sqrt[3]{3})^v \sqrt[3]{3}.$$

We multiply both sides of the equation by $\frac{\sqrt[3]{3}}{(\sqrt[3]{3})^v v}$.

$$2\sqrt[3]{3} < 1 - \frac{1}{v} + (\sqrt[3]{3})^2 + \frac{(\sqrt[3]{3})^2}{v}.$$

For a high-rate input event stream, the number of events per graphlet v will be large. Therefore, $\frac{(\sqrt[3]{3})^2 - 1}{v} \approx \frac{1}{v} \approx 0$.

$$2\sqrt[3]{3} \approx 2.88 < 1 + (\sqrt[3]{3})^2 \approx 3.08 \qquad (1)$$

Analogously, we verify that $cpu(p_b) < cpu(p_o)$ below.

$$2(\sqrt[3]{3})^v < \frac{(\sqrt[3]{3})^v}{\sqrt[3]{3}} + (\sqrt[3]{3})^v \sqrt[3]{3}$$

We multiply both sides by $\frac{\sqrt[3]{3}}{(\sqrt[3]{3})^v}$ and get Equation 1. $\square$

THEOREM 5. ***[Cost of Nearly Balanced Partitioning Plan.]*** *Let G be a CET graph and $k \in N$. The H-CET algorithm on a nearly balanced partitioning of G into k graphlets has lower CPU and memory costs than on an unbalanced partitioning of G into k graphlets.*

PROOF. Since the memory cost for storing one CET across graphlets and the CPU cost for computing CETs across graphlets are independent from the quality of partitioning a CET graph into k graphlets, we ignore these costs below. We now prove that even moving one atomic graphlet from one nearly balanced graphlet to another nearly balanced graphlet increases both the memory and CPU costs.

Let $G = (V, E)$ be a CET graph. Let $p_n = \{g_{n1} = (V_{n1}, E_{n1}), g_{n2} = (V_{n2}, E_{n2})\}$ be a nearly balanced partitioning plan of G and $p_u = \{g_{u1} = (V_{u1}, E_{u1}), g_{u2} = (V_{u2}, E_{u2})\}$ be an unbalanced partitioning plan of G. The unbalanced partitioning plan p_u is same as the nearly balanced one p_n except that one atomic graphlet g with m nodes ($m \in \mathbb{N}, m \geq 1$) is copied from the first graphlet to the second. That is, $|V_{u1}| = |V_{n1}| - m$ and $|V_{u2}| = |V_{n2}| + m$ (Figure 18).

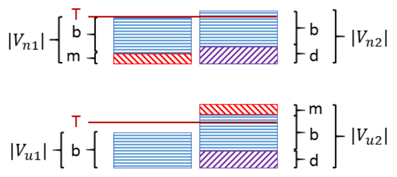


Figure 18: Nearly balanced vs. unbalanced partitioning plan

According to Definition 8, $T = \lceil \frac{|V|}{2} \rceil$ is the threshold for the number of events in the nearly balanced graphlets g_{n1} and g_{n2}. Assume $|V_{n1}| = b + m$ where m is the number of events in the atomic graphlet g that we move from g_{n1} to g_{n2} and $b \in \mathbb{N}, b \geq 1$. Since g_{n1} is nearly balanced, removing the atomic graphlet g with m events form g_{n1} causes an underflow in g_{n1}, i.e., $|V_{u1}| = b < T$.

According to Definition 8, the partitioning plan p_u is unbalanced if at least two atomic graphlets cause an overflow in G_{u2}. Since $b < T$ and all m nodes belong to the same atomic graphlet g, $|V_{u2}| > b + m$. Let $|V_{u2}| = b + d + m$ for some $d \in \mathbb{N}, d \geq 1$. Then $|V_{n2}| = b + d$.

Let $x = \sqrt[3]{3}$. Then the memory cost of CET detection according to these 2 partitioning plans is computed as follows:

$$\begin{aligned} mem(p_b) &= x^{b+m}(b+m) + x^{b+d}(b+d) \\ &= x^b x^m b + x^b x^m m + x^b x^d b + x^b x^d d \\ &= x^b b(x^m + x^d) + x^b x^m m + x^b x^d d \\ mem(p_u) &= x^b b + x^{b+d+m}(b+d+m) \\ &= x^b b + x^b x^d x^m b + x^b x^d x^m d + x^b x^d x^m m \\ &= x^b b(1 + x^d x^m) + x^b x^d x^m d + x^b x^d x^m m \end{aligned}$$

$mem(p_b) < mem(p_u)$ since $x^b b(x^m + x^d) < x^b b(1 + x^d x^m)$, $x^b x^d d < x^b x^d x^m d$ and $x^b x^m m < x^b x^d x^m m$.

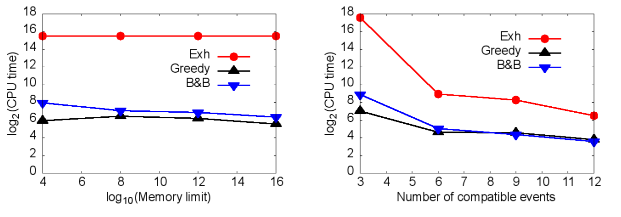
Analogously, we verify that $cpu(p_b) < cpu(p_u)$ as follows:

$$x^b(x^m + x^d) < x^b(1 + x^{d+m}). \qquad \square$$

E. ADDITIONAL EXPERIMENTS

E.1 CET-Graph Partitioning Algorithms

Continuing our experiments in Section 7.3, in Figure 19 we measure the CPU time of the CET-graph partitioning algorithms while varying the memory limit and the number of compatible events.



(a) Memory limit (PA) (b) Event compatibility (FT)

Figure 19: CPU costs of CET graph partitioning algorithms

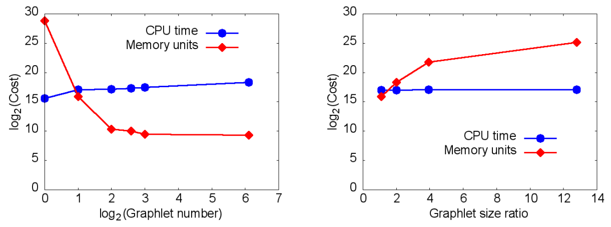
Varying Memory Limit. As Figure 19(a) illustrates, the exhaustive optimizer is indifferent to the memory limit. It traverses the entire search space to find an optimal partitioning plan. In contrast, both the B&B and greedy algorithms utilize the memory constraint to skip infeasible levels. Thus, B&B is 2 orders of magnitude faster and requires 12–fold less memory than the exhaustive optimizer if memory limit is 10k memory units.

When the memory constraint is loose, a few cuts are enough to satisfy the memory constraint. Thus, the B&B and greedy algorithms perform almost equally well. However, when the memory constraint is tight, a CET graph has to be partitioned more to avoid an out-of-memory error. While our B&B optimizer considers all nearly balanced partitioning plans at a level to find an optimal partitioning plan, the greedy optimizer considers only one per level. Thus, the greedy optimizer is up to 4–fold faster than B&B if memory limit is 10k memory units at the expense of finding a sub-optimal partitioning plan.

Varying Event Compatibility. With the increasing number of compatible events the size of graphlets increases and their number decreases (Figure 19(b)). Thus, the size of the search space also decreases. Consequently, the CPU time of the algorithms rapidly decreases as well until it becomes almost constant if more than 10 events are compatible. If 3 events are compatible, our B&B optimizer is 2 orders of magnitude faster than the exhaustive optimizer thanks to our effective pruning principles.

E.2 Properties of Partitioning Search Space

In Figure 20, we experimentally confirm the cost monotonicity properties of the CET graph partitioning search space (Sections 5.2 and 5.3) while varying the number of graphlets and their size ratio.



(a) Cost variation across levels (b) Cost variation at one level

Figure 20: Search space properties (PA)

Varying Graphlet Number. The goal of this experiment is to confirm the properties across levels of the search space (Theorems 2 and 3) which support the infeasible level and inefficient branch pruning by our B&B optimizer.

In Figure 20(a), we observe the monotonic CPU and memory cost variation while varying the number of graphlets. The first case on the X-axis (1 graphlet) corresponds to T-CET since all events belong to 1 graphlet. The last case on the X-axis (69 graphlets) corresponds to M-CET since each relevant event is a separate graphlet. In all other cases, H-CET is applied to an optimally-partitioned CET graph into the given number of graphlets.

As the number of graphlets increases, the memory requirement decreases exponentially (5 orders of magnitude comparing 1 and 69 graphlets) and CPU time increases exponentially (7–fold comparing 1 and 69 graphlets). Such cost variation is expected since the larger the number of graphlets, the smaller their size, the less CETs are stored per graphlet and the more CPU overhead is provoked while computing CETs across graphlets.

Varying Graphlet Size Ratio. The goal of this experiment is to confirm the properties at one level of the search space (Theorems 4 and 5) which lead to the unbalanced node pruning by our B&B optimizer.

In Figure 20(b), we observe the monotonic cost variation while varying the graphlet size ratio. The first case on the X-axis (the ratio is 1) corresponds to CET detection in a CET graph that is partitioned in a balanced way. The last case on the X-axis (the ratio is 13) shows the costs of CET detection in a CET graph that is partitioned in an unbalanced way, namely, one graphlet has 13 times more events than another.

As the graphlet size ratio increases, the memory requirement increases exponentially (2 orders of magnitude comparing the balanced and the unbalanced partitioning plans). Such memory increase is expected since the larger the size of a graphlet the more CETs are stored in it. In contrast, the CPU time increases only slightly (1.03–fold comparing these extreme cases). This is explained by the fact that the same number of graphlets causes the same amount of CPU overhead for CET construction across graphlets.