Efficient Temporal Subgraph Management: A New Interval Index

Dian Ouyang
Guangzhou University
dian.ouyang@gzhu.edu.cn

Yikun Wang
University of New South Wales
yikun.wang1@unsw.edu.au

Dong Wen
University of New South Wales
dong.wen@unsw.edu.au

Wenjie Zhang
University of New South Wales
wenjie.zhang@unsw.edu.au

Yaping Liu
Guangzhou University
ypliu@gzhu.edu.cn

Xuemin Lin
Shanghai Jiao Tong University
xuemin.lin@sjtu.edu.cn

ABSTRACT

Many research efforts have been conducted to mine various substructures in temporal graphs. Given a set of temporal subgraphs and an arbitrary time window, we aim to design an index structure to efficiently retrieve all subgraphs contained in (sub-valid) or containing (super-valid) the window. The problem falls in the category of fundamental interval range queries studying the relationship between a set of intervals and a query interval. We propose a novel data structure that is tailored for real-world temporal subgraphs with high volumes, great overlaps, and frequent updates. We design a lightweight linear size index structure with a linear index construction time. The index enables us to answer queries in near optimal time. We also propose algorithms to maintain the index. Our running time to insert a subgraph is bounded by the size of the changed values in the index, which is optimal in the context. Deleting a subgraph takes constant time. Experiments on real-world datasets with numerous subgraph instances demonstrate our significant advantages compared with existing baselines.

PVLDB Reference Format:
Dian Ouyang, Yikun Wang, Dong Wen, Wenjie Zhang, Yaping Liu, and Xuemin Lin. Efficient Temporal Subgraph Management: A New Interval Index. PVLDB, 19(6): 1170-1183, 2026.
doi:10.14778/3797919.3797926

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/iykw/TEM-Graph.

1 INTRODUCTION

Temporal graphs naturally arise in real-world applications where interactions evolve over time. A widely studied model is the relational event (RE) graph [3, 5], which represents activities between entities as a sequence of timestamped edges. Temporal substructures are critical for many analytic tasks. For example, temporal motifs are used to model and analyze interactions among small groups of vertices [18, 33, 36]. A wide range of mining algorithms have been developed to identify time-sensitive subgraphs [28, 31, 42, 43, 50, 54, 56].

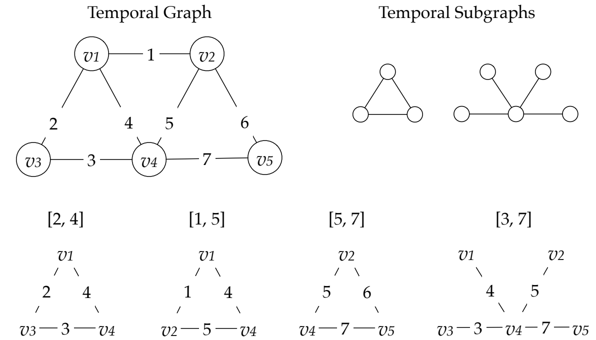


Figure 1: An example of temporal subgraphs.

We observe that retrieving subgraphs related to a specified time window is a common and fundamental task in temporal graph analysis [18, 30, 30, 32, 47, 48, 50, 51]. We call a subgraph *sub-valid* to a query window if the query window contains the time window of the subgraph. We select a few of the applications as follows:

- Social Media Community Detection [46, 52]. Online communities can be represented as dense subgraph patterns within a temporal network. The query window defines the time period of interest (e.g., the week before final exams), allowing us to retrieve communities that emerged and were active within that window.
- Financial Fraud Detection [18, 22]: In transaction networks, analysts monitor for suspicious motifs such as triangular arbitrage cycles or star-shaped fund aggregation patterns [27]. The query window represents a regulatory or investigative period (e.g., "Q3 2025" or "last 24 hours"), ensuring the entire suspicious activity occurred within that period.
- Cybersecurity Threat Detection [34, 55]: For malicious network attack detection, where vertices and edges represent IPs and connections, respectively, identify all Denial-of-Service attacks that occurred last night between 12 a.m. and 4 a.m.

In addition to sub-valid queries, certain temporal subgraph mining problems study elements covering the query window. We call this *super-valid* subgraph queries. For example, [17] mines k-cores keeping appear in a time duration. The time window of each resulting subgraph implies it exists persistently at every timestamp during the window. Therefore, all subgraphs containing a query window should be returned when people are interested in persistent subgraphs for the window. We mainly focus on sub-valid queries in this paper and will extend our techniques to super-valid queries with the same theoretical guarantees in Section 6.

Dong Wen is the corresponding author.
This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 6 ISSN 2150-8097.
doi:10.14778/3797919.3797926

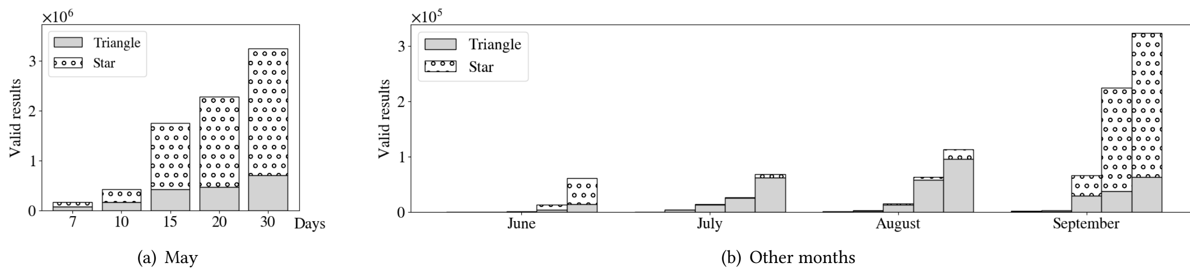


Figure 2: A case study on the College dataset. We analyze the trend of different community patterns (subgraphs) under query windows ranging from 7 to 30 days.

Example. In Figure 1, an instant messaging network is modeled as a temporal graph where each edge represents a message exchanged between two users. A triangle (or k-clique) in such a network indicates a tightly connected group, while a k-star (a central vertex u connected to k distinct vertices) reflects a user maintaining communication with multiple contacts. Service providers may track subgraph instances to detect communities or identify influential individuals within specific time periods. Given a query window $Q = [3, 7]$, two instances are found within this interval: a triangle occurring in $[5, 7]$ and a 4-star with window $[3, 7]$.

Case Study. The College dataset records messages exchanged in an online community of students at the University of California, Irvine [35]. Collected from April to October 2004, the platform required each user to create a personal profile that others could search and use to start conversations. We extract and analyze the temporal evolution of two representative patterns (triangles and 4-stars) by collecting their embeddings and active intervals. To observe temporal trends, we issue a series of sub-valid queries with window start times at the beginning of each month and window lengths varying from 7 to 30 days. The results are shown in Figure 2. We find that the number of communities appearing in May is an order of magnitude higher than in other months. Moreover, about 4‰–15‰of communities form within just 2–3 days in May, a phenomenon not observed in other periods (hence omitted from Figure 2). During the middle months, the number of both community types decreases noticeably, with triangles remaining dominant. This suggests that students mainly interact within stable, close-knit groups during this period. In September, the start of the fall term, the counts rise again, and star-shaped patterns become more active, reflecting renewed social expansion as students reconnect and form new relationships.

Baselines. The sub-valid query is essentially the Contains query in the category of fundamental interval range queries [2]. Interval tree [16] is a classical data structure for interval range query and HINT [8, 9] is the state-of-the-art in-memory interval index. The existing interval index suffers from two limitations. Firstly, these methods partition records along the time axis and are designed for the Overlaps query (where two windows intersect). The sub-valid query is more restrictive, making its results a subset of Overlaps. Although HINT introduces pruning techniques for sub-valid queries, it still scans unnecessary items, leading to significant overhead in our context. This is because one edge often appears in many temporal subgraphs (e.g., in temporal motif enumeration [18]), and any subgraph may contain, be contained in, or overlap with many others. Secondly, the entire dataset and its domain (the minimum and maximum timestamps) should be known apriori, but a real-world temporal graph is often represented as an edge stream where edges continuously arrive. Those two methods cannot efficiently insert items with an end time later than the current time period, while the setting is not acceptable in the context of dynamic temporal graphs. There also exist some other earlier solutions. They either perform poorly in query processing or take a non-linear index size not scalable for the great amount of temporal subgraphs in practice. More details can be found in Section 3.

Our Method. We propose a new interval index tailored for window queries on temporal subgraphs. Table 1 summarizes the theoretical results of our method compared with HINT and the interval tree. As the query time is output-sensitive, we use k to denote the result size. For the interval tree, the query time of a sub-valid (super-valid) query is bounded by the overlapping results instead of the actual result size. HINT does not provide a clear bound for either query type. In contrast, our method answers queries in $O(k + \log n)$ time and achieves optimal update time, even when the domain of the dataset expands. Our techniques focus on designing auxiliary data structures and query methods to avoid invalid subgraphs. We formulate a special graph structure called TEMS-Graph which is a directed acyclic planar graph. Each node in the graph corresponds to a subgraph. Directed edges in the graph are created based on the start times and end times of subgraphs. The edges are used to avoid scanning nodes that do not belong to the results. We also propose algorithms to update the index for subgraphs generated by new arriving edges and subgraphs deleted by expired edges.

Existing research on temporal subgraph mining and continuous subgraph matching focuses on discovering subgraphs that satisfy structural and temporal constraints [25, 41, 44]. Our work complements these methods by providing an efficient way to manage and query the discovered subgraphs over time. Specifically, once subgraph instances and their active intervals are generated by enumeration algorithms, our method can index and retrieve those that are valid within any query window. This enables seamless integration with existing temporal subgraph mining pipelines. We provide theoretical guarantees for window queries and design an optimal update mechanism for expanding time ranges, a capability that naturally aligns with the continuous evolution of temporal graphs.

Contributions. We summarize our contributions as follows.

Table 1: Comparison between different methods. n is the number of subgraphs, k is the number of query results. k_o is the number of subgraphs overlapping with the query window. ϕ is a small factor determined by the dataset domain and average interval length.

Methods	Query Time		Index	
	Sub-Valid	Super-Valid	Space	Update
Interval Tree	$O(k_o + \log n)$	$O(k_o + \log n)$	$O(n)$	–
HINT	–	–	$O(n\phi)$	–
Ours	$O(k + \log n)$	$O(k + \log n)$	$O(n)$	✓

- *Linear Index Size and Near Optimal Query Time.* We propose a new interval index for sub-valid (super-valid) subgraph queries. Given the input size n and the result size k, our index takes linear size $O(n)$ and answers queries in $O(k + \log n)$, which is optimal when the result size is large. To the best of our knowledge, no existing study achieves them simultaneously.
- *Linear Index Construction and Optimal Update Time.* We propose an algorithm to construct our index in linear time. We also propose algorithms for index maintenance. For inserting a subgraph, our time complexity is bounded by the amount of changed values in the index, which is optimal in the context. For deleting a subgraph, our time complexity is constant.
- *Extensive Performance Studies.* We conduct extensive performance studies to test the performance of proposed algorithms on large temporal graphs, including over 4 billion subgraphs. Experimental results demonstrate that our algorithm can achieve a speedup of up to two orders of magnitude sub-valid queries compared to the state-of-the-art while consuming linear space.

2 PRELIMINARY

A temporal graph is a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ with a set of vertices $\mathcal{V}$ and a set of edges $\mathcal{E}$, such that each edge e is a triplet (u, v, t), where t is a timestamp indicating the interaction time between two vertices. A temporal subgraph s is a graph whose vertex set and edge set are subsets of those of $\mathcal{G}(\mathcal{V}, \mathcal{E})$. The window of a subgraph s is denoted as $[s.start, s.end]$, where $s.start$ and $s.end$ represent the minimum and maximum timestamps of the edges in s, respectively.

Definition 2.1 (Sub-Validity). Given a query time window $Q = [t_s, t_e]$, a subgraph s is sub-valid for the query window if its window is covered by Q, i.e., $s.start \geq t_s \land g.end \leq t_e$.

Problem Statement. Given a set of subgraphs S and an arbitrary query window $Q = [t_s, t_e]$, we aim to efficiently compute all subgraphs in S that are sub-valid for Q.

Example 2.2. Consider the temporal graph in Figure 1 with 5 vertices and 7 edges. We examine triangles and 4-stars, where a 4-star has a central vertex u connected to four distinct vertices. The triangle $\{v_1, v_2, v_3\}$ has window $[1, 5]$, and the 4-star $\{v_4, v_1, v_2, v_3, v_5\}$ (centered at v_4) has window $[3, 7]$. Given an query window $Q = [2, 7]$, the sub-valid temporal subgraphs are the triangle $\{v_1, v_3, v_4\}$ with window $[2, 4]$, triangle $\{v_2, v_4, v_5\}$ with window $[5, 7]$ and the 4-star with window $[3, 7]$.

In addition to sub-validity, the other query type, namely super-validity is also adopted in temporal subgraph queries.

Definition 2.3 (Super-Validity). Given a query time window $Q = [t_s, t_e]$, a subgraph s is super-valid for the query window if its window covers Q, i.e., $s.start \leq t_s \land s.end \geq t_e$.

An immediate online solution for the problem is to linearly scan all subgraphs in S and pick all valid results. Let n denote the total number of subgraphs and k the number of query results. The query efficiency is limited when k is significantly smaller than n. Queries are often issued repeatedly with varying time windows over commonly interested substructures. This motivates us to pursue an index-based solution in this paper. We will mainly focus on querying sub-valid subgraphs and will extend our techniques for super-validity queries in Section 6.

When clear from the context, we simply call a subgraph valid if it is sub-valid to the query window. We assume subgraphs have unique start and end times, with ties broken by subgraph ID in practice. We assume inclusive window boundaries, which can be trivially adapted to open intervals. Due to space limitations, proofs are provided in the complete version included in the artifact.

3 EXISTING STUDIES

The research problem essentially falls in the category of fundamental interval range queries which aim to retrieve data intervals with a certain relationship to the query intervals. A set of precise relationships between intervals was formally defined in [2], called Allen's algebra, where the two relationships considered in this paper are the most representative and challenging ones in the algebra. For example, the sub-validity is equivalent to the Contains predicate in Allen's algebra.

3.1 Interval Tree

One of the most popular data structures handling interval range query is Edelsbrunner's interval tree [16], which is a special binary search tree. The tree hierarchically divides the domain by placing all intervals strictly before (or after) the domain's center to the left (right) subtree and all intervals that overlap with the domain's center at the root. This process is recursively applied to both left and right subtrees, using the centers of the corresponding sub-domains. The intervals at each tree node are sorted into two lists: one based on their start values and the other based on their end values.

For the sub-valid query, the interval tree detects all candidates whose start (or end) time is in the query time window. Given a query $Q = [t_s, t_e]$ and a tree node x with center c, we check the candidate subgraphs of x if $t_s \leq c \leq t_e$. We select the start list and find each subgraph s such that $t_s \leq g.start \leq t_e$ by binary search. For the candidate subgraphs, we select the subgraphs with $t_s \leq g.end \leq t_e$ from the candidates. Then, we recursively query the left (resp. right) subtree if $t_s \leq c$(resp. $c \leq t_e$).

Drawbacks. The interval tree maintains a good balance when intervals are uniformly distributed over the domain. The centers of tree nodes effectively partition the time range, ensuring a balanced assignment of intervals across the left and right subtrees. However, in temporal subgraphs, it is common for one interval to fully contain or be fully contained by another. For sub-valid queries, while the interval tree can exclude subgraphs s with $g.end \leq t_s$ or $t_e \leq g.start$

using the center c, each remaining candidate must still be verified if its interval $[g.start, g.end]$ overlaps with $[t_s, t_e]$. Let k_o be the number of Overlaps results for a query Q; the query time remains bounded by $O(k_o + \log n)$ instead of k, where $k \leq k_o$ is the number of truly valid results. This leads to inefficiency when the number of overlapping subgraphs is large. Updating an interval tree requires reordering the start and end time lists within tree nodes, which can be highly time-consuming.

3.2 HINT

HINT [8] is a hierarchical index for intervals. It supports multiple predicates in Allen's algebra, including sub-valid and super-valid [9]. HINT defines a hierarchy of $m + 1$ levels, where each level $l, 0 \leq l \leq m$ uniformly divides the domain into 2^m partitions. Each interval s is normalized and discretized in the $[0, 2^m - 1]$ domain, and assigned to the smallest set of partitions from all levels to ensure the non-redundant coverage[10]. The intervals in each partition are divided into two classes: those that start before the partition (replicas) and those that start inside the partition (originals).

This design enables efficient range queries by reducing the number of accessed partitions. Various techniques are applied to minimize comparisons and cache misses. For a sub-valid query Q, the expected number of partitions requiring comparisons is four parts in the index. These four parts contains results overlapping with Q. To answer the sub-valid query of Q, HINT identifies all these results and performs necessary checks to determine whether each subgraph satisfies the sub-valid constraint.

Drawbacks. For sub-valid queries, the query process requires scanning all intervals from the first to last partitions associated with Q, thus the cost cannot be bounded by k. Secondly, to build the HINT index, we must know the domain of the datasets. If the data domain expands (as in a temporal graph), the partitions must potentially be updated to accommodate the new domain. Additionally, the assignment of data intervals to partitions needs corresponding adjustment to preserve the index's beneficial properties.

3.3 Other Related Works

Temporal Subgraph Mining. Many models have been proposed for time-related graph data, differing in graph formalization (e.g., snapshot vs. incremental changes based on the initial graph) and the type of temporal information associated with edges (e.g., timestamps vs. time intervals) [3, 5, 24, 37, 53]. We discuss one of the most commonly used temporal graph models, i.e., relational event (RE) graph [3]. Subgraph mining has been widely discussed in recent years [17, 19, 28, 38]. In temporal graphs, given a query window and an integer k, the historical k-core query aims to find the vertices of k-core over the window [49, 51]. In real-world applications, the duration of a subgraph is also a key constraint. Durable subgraph matching finds all subgraphs in the temporal graph that not only match the given query graph but also have a duration longer than a user-specified duration threshold [29, 49]. These works focus on a certain pattern with tailored indexing solutions. In [13], the authors study space-query trade-offs for range subgraph counting and listing (enumeration) problems across various patterns (e.g., wedge, l-clique, l-star). A temporal graph can be viewed as an attributed graph, allowing the conclusions in [14] to be applied in the temporal setting. A few studies investigate the temporal subgraph counting problems. [18] proposes a scalable parallel framework for exactly counting the δ-temporal motifs, where the temporal edges are in chronological order within a δ time constraint. [46] addresses the δ-temporal triangle counting problems and binary δ-temporal triangle counting, which only considers the existence of δ-temporal triangle among three vertices. We observe that temporal subgraph queries often involve time range constraints, such as retrieving subgraphs within a specific window or those covering a given range. Motivated by this, we propose an index structure fit with the characteristics of temporal subgraphs, enabling more efficient support for range queries.

Interval Range Query. [11] introduces several geometry indices for orthogonal range search, aiming to report all records whose value falls in a certain range. The sub-valid query over intervals is equivalent to the 2-dimensional rectangular range searching problem. A kd-tree takes $O(n)$ space and answers a query in $O(\sqrt{n} + k)$ time [40]. A range tree uses $O(n \cdot \log n)$ storage and takes $O(\log^2 n + k)$ query time [26]. Chazelle's structure [6] is proposed to solve the range counting and reporting (enumeration) problems. A space-query tradeoff for the 2-dimensional range reporting problem is possible (Theorem 2 [6]). Boolean expression matching involves identifying whether a given input satisfies a Boolean condition composed of multiple attribute constraints [20, 21, 39]. It can be interpreted as storing intervals and supports efficient point containment queries, which determine whether a given point falls within any of the stored intervals. In contrast, our work focuses on more general range queries, where the input is itself an interval rather than a single point. The domain-expanded version of HINT is discussed in [10], but it only implements overlaps query and do not support deletions yet. Temporal databases often support operations like temporal aggregation and interval joins [7, 12, 15, 45]. The timeline index [23] and period index [4] have been adapted to support sub-valid and super-valid queries. HINT has been shown to outperform both by 1–2 orders of magnitude in query performance [9]. In this paper, we propose an index structure with $O(n)$ space complexity and optimal $O(\log n + k)$ query time for various query types, where k is the result size. We design an optimal update mechanism for expanding time ranges, which naturally fits the continuous evolution of temporal graphs.

4 OUR INDEX

We propose a novel solution that improves the query time to optimal while keeping the linear index size. By optimal, we mean the query time complexity is bounded by the number of resulting subgraphs when the result size is a dominant factor.

4.1 Warming Up: A New $O(k \cdot \log n)$ Approach

We start from a simplified setting where the query specifies only an end time t_e, and the goal is to retrieve all subgraphs whose end times do not exceed t_e. A natural solution is to sort all subgraphs by increasing end time, which we refer to as the *end time order*. Note that the subgraph with an earlier start time ranks higher (placed in front) in the end time order if two subgraphs have the same end time in practice. An end-time query can then be answered by scanning this order from the beginning until a subgraph with end

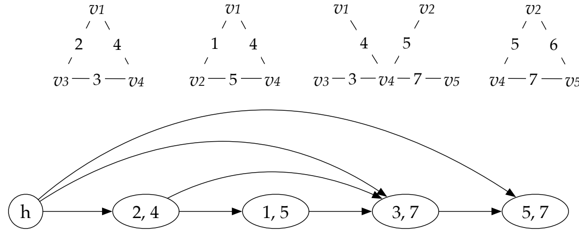


Figure 3: TEM-Graph constructed from temporal subgraphs in Figure 1.

time greater than t_e is reached. We adopt this end-time order for general window queries. Given a query window $[t_s, t_e]$, we can locate all subgraphs with end times within the window using a binary search followed by a sequential scan. The remaining challenge is to avoid scanning subgraphs whose start times are earlier than t_s. Our proposed techniques are motivated precisely by this need to efficiently skip such invalid subgraphs.

Finding the Next One. Given a valid subgraph s, we define the TEM-Graph (Temporal subgraph Management index) below to enable searching the next valid subgraph of s in the end time order.

Definition 4.1 (TEM-Graph). Given a set of temporal subgraphs S, the TEM-Graph is a directed graph defined as follows:

(1) Nodes correspond to subgraphs in S;

(2) Given two nodes (subgraphs) u and v, v is an out-neighbor of u if (i) $v.end > u.end$; and (ii) there does not exist $w \in S$ such that $v.end > w.end > u.end$ and $w.start > \min(u.start, v.start)$;

(3) Out-neighbors of each node are arranged in increasing order of end times.

We use the term node below in the context of TEM-Graph to refer to an input subgraph. Given a node u in a TEM-Graph, we use $N_{out}(u)$ and $N_{in}(u)$ to denote the out-neighbors and in-neighbors of u, respectively. An edge (u, v) represents a directed edge from u to v. The condition (2) in the definition implies that the start times of all nodes between u and v in the end time order are earlier than those of u and v. The last two conditions guarantee that the out-going neighbors of each node are also in the increasing order of start times.

Example 4.2. In Figure 3, we show the TEM-Graph for all temporal subgraphs from Figure 1. The nodes are ordered by the end times of subgraph windows, and h is a virtual head node. Node $u = [2, 4]$ has two out-neighbors $v = [1, 5]$ and $w = [3, 7]$, both satisfying condition (2). Node v has one out-neighbor $w = [3, 7]$. The nodes behind w cannot become the out-neighbor of v, as $w.start > v.start$ breaks condition (2).

Lemma 4.3. *The TEM-Graph is a planar graph.*

Lemma 4.3 enables the following space guarantee for our index.

Theorem 4.4. *The space complexity of TEM-Graph is $O(n)$.*

The Query Algorithm. We now discuss our query algorithm given the TEM-Graph. Assuming we have already found the first valid node u (will be discussed later), we iteratively apply the following theorem to get the next valid one for each valid node u.

Algorithm 1: FindNextValid

Input: the query window $[t_s, t_e]$ and a valid node u in TEM-Graph
Output: the next valid subgraph after u

1 $l \leftarrow 0,\ r \leftarrow |N_{out}(u)| - 1, m \leftarrow 0$;
2 **while** $l < r$ **do**
3 $m \leftarrow (l + r)/2$;
4 **if** $N_{out}(u)[m].start > t_s$ **then**
5   $r \leftarrow m$;
6 **else** $l \leftarrow m + 1$;
7 $v \leftarrow N_{out}(u)[l]$;
8 **if** $v.start < t_s \vee v.end > t_e$ **then return** Null;
9 **return** v;

Theorem 4.5. *Given a query window $[t_s, t_e]$ and a valid node u to the window, let v be the first out-neighbor of u in TEM-Graph such that $v.start \geq t_s$. The nearest valid node behind u in the end time order is v if $v.end \leq t_e$. Otherwise, there is no valid node behind u.*

Given the ordered out-neighbors, finding the out-neighbors in Theorem 4.5 needs a binary search and takes $O(\log n)$ time complexity since the number of out-neighbors of any node is bounded by n. We iteratively find the next one until we cannot find any satisfied outgoing neighbor. Therefore, we can compute all resulting subgraphs in $O(k \cdot \log n)$ time complexity, where k is the number of resulting subgraphs. The formal pseudocode to find the next valid node is shown in Algorithm 1. The input u is a valid node. The binary search (lines 1–6) finds the next node v whose start time is not smaller than t_s. If it does not exist, v is the last out-neighbor of u. The algorithm returns v if it has a valid end time. The time complexity of Algorithm 1 is clearly $O(\log n)$.

Example 4.6. Given a query window $Q = [2, 7]$, suppose we have already found the first valid item $[2, 4]$ and now search through the out-neighbors to find the next one. There are two out-neighbors of $[2, 4]$, which are sorted in ascending order of start times, as shown in Figure 3. We output $[3, 7]$ by binary search as the next valid item, and continuously search the out-neighbors of $[3, 7]$. We finish the query when we reach $[5, 7]$.

Correctness Analysis. We prove the correctness of Theorem 4.5. Given a query window $[t_s, t_e]$ and a valid node u (i.e., $u.start \geq t_s, u.end \leq t_e$), let v be the next valid node of u in the end time order. Based on Definition 4.1, v must be an out-neighbor of u since the second condition in the definition is satisfied. We then prove the out-neighbor picked in Theorem 4.5 is the correct next valid node. Note that we assume the subgraphs mentioned below are all located before s, i.e., the last subgraph with $end \leq t_e$ in the end time order. Otherwise, the search will be terminated immediately.

Lemma 4.7. *Given a query window $[t_s, t_e]$ and a valid node u for the window, a node v is valid if v is the out neighbor of u in TEM-Graph and $v.start \geq t_s \wedge v.end \leq t_e$.*

Lemma 4.8. *Given a node u in TEM-Graph, let v, w be a pair of adjacent out-neighbors of u ($v.end < w.end$). For any node x between v and w in the end time order, we have $x.start < v.start$.*

Algorithm 2: Sub-Query

```
Input: the query window [t_s, t_e] and the TEMS-Graph TG
Output: subgraphs in [t_s, t_e]
1  u ← FindNextValid(t_s, t_e, TG, h);
2  if u = Null then return Null;
3  output u;
4  i ← N_out(h)[l].successor, where l is the offset of u in
     N_out(h);
5  while i ≠ Null do
6      while i > 0 ∧ N_out(u)[i − 1].start ≥ t_s do
7          i ← i − 1;
8      v ← N_out(u)[i].id;
9      if v.start < t_s ∨ v.end > t_e then break;
10     i ← N_out(u)[i].successor;
11     u ← v;
12     output u;
```

Given a valid node u, let v be the first out-neighbor of u such that $v.start \geq t_s$ where t_s is the query start time. Lemma 4.7 guarantees that v is valid. Lemma 4.8 guarantees that we do not lose any result between u and v in the end time order. Specifically, let w be the next valid one of v in out-neighbors of u, and we have $w.start \geq t_s$. Lemma 4.8 proves the start time of all nodes between v and w in the end time order is smaller than t_s. We can apply Lemma 4.8 for all previous out-neighbors of v and prove that the start times of all nodes between v and u are smaller than t_s.

Finding the First Valid Item. TEM-Graph also allows us to find the first valid item using the same way. To this end, we introduce an additional virtual node h with $h.start = \infty, h.end = -\infty$. According to the definition of TEM-Graph, h is a head node in front of the end time order, and the first out-neighbor of h is the node with the smallest end time in S. The subsequent out-neighbors of h are added in an increasing order of start times and end times (the last two conditions in Definition 4.1). Given $Q = [t_s, t_e]$, the first valid node u can be found via a binary search from the out-neighbors of h with $u.start \leq t_s \wedge u.end \leq t_e$. If we cannot find such u, there is no valid subgraph for Q. We prove the correctness as follows.

Lemma 4.9. *Given a query window $Q = [t_s, t_e]$, there is no valid subgraph for Q if one of the conditions holds, (i) $u.start < t_s, \forall u \in N_{out}(h)$, or (ii) $u.end > t_e$ where u is the first subgraph in $N_{out}(h)$ with $u.start \geq t_s$.*

4.2 From $O(k \cdot \log n)$ To $O(k + \log n)$

We now propose a simple but effective technique to eliminate the factor of $O(\log n)$ in the time complexity of query processing. Our idea is to avoid the binary search of out-neighbors and locate the correct out-neighbor of each node based on the valid ones found in earlier rounds. To this end, we store an additional value for each edge in TEM-Graph, called successor, defined as follows.

Definition 4.10 (Successor). Given a node u and an out-neighbor v of u, the successor of the edge (u, v) is an edge (v, w) from v such that (1) the start time of w is not smaller than $\min(u.start, v.start)$; (2) there does not exist an edge (v, w') such that $\min(u.start, v.start) \leq w'.start < w.start$.

Note that if there does not exist a satisfied edge, the successor of (u, v) is set as the last out-going edge of v. If v does not have any out-neighbor, the successor of (u, v) is Null. Definition 4.10 implies w is the first out-neighbor of v with a non-smaller start time compared to (u, v), given that out-neighbors are arranged in increasing order of start times. We call the updated structure TEMS-Graph. In summary, for each out-neighbor of a node u in TEMS-Graph, we maintain a pair $\langle id, successor \rangle$. id represents the out-neighbor id, which is the same as the basic TEM-Graph. $successor$ represents the offset of the successor in its out-neighbor list.

Example 4.11. In Figure 3, let $u = [2, 4]$, $v = [1, 5]$, and $w = [3, 7]$, the successor of (h, u) is (u, w), because v has smaller start time than $\min(h.start, u.start) = 2$. The successor of (u, v) is (v, w), as w is the first out-neighbor of v and $w.start \geq \min(u.start, v.start) = 1$.

Assume that we have a valid node v and the previous valid node u of v (v must be an out-neighbor of u). We aim to find the next valid node of v in the end time order. Let (v, w) be the successor of (u, v). It is clear to see that w is a candidate since $w.start \geq \min(u.start, v.start)$ based on Definition 4.10. However, there may exist valid nodes between v and w based on Theorem 4.5. In other words, there may exist a valid out-neighbor w' of v such that $w.start > w'.start \geq t_s$. Therefore, we first locate w (via the successor) in the out-neighbors of v. We move backward from w in the decreasing order of out-neighbors of v to find the first valid out-neighbor of v.

Based on the TEMS-Graph, we present our query algorithm in Algorithm 2. We find the first valid subgraph u via the head node (lines 1–2). Then we locate the successor of (h, u) (line 4) and move backward from the successor to find the first valid item v after u (lines 6–8). In the next iteration, we start from the successor of (u, v) (lines 10–11). The search completes when we reach a node whose end time is greater than t_e or cannot find any out-neighbor satisfying the start time (line 9).

Theorem 4.12. *The time complexity of Algorithm 2 is $O(k+\log n)$.*

5 INDEX CONSTRUCTION

In this section, we first propose the algorithm to construct the TEMS-Graph and then propose techniques to maintain the TEMS-Graph to insert or delete subgraphs in the temporal graph scenario.

5.1 Constructing TEMS-Graph

TEM-Graph Construction. Based on Definition 4.1, each pair of adjacent nodes in the end time order forms a directed edge in TEM-Graph. The challenge is to create other edges that adhere to the order of out-neighbors. We design an elegant method to construct them. The purpose of out-neighbors is to skip items with smaller start times. We iteratively remove the node with the smallest start time from the end-time order and add an edge from u to w, where u and w are the predecessor and successor of v in the end-time order. This guarantees that, for every directed edge from u to w, all items between them in the end-time order have smaller start times, satisfying the second condition of Definition 4.1. We repeat this process until no subgraphs remain in the start-time order.

To implement this idea, we utilize an auxiliary list of nodes in increasing order of start times (called *start time order* for convenience). For two nodes with the same start time, the node with the smaller end time ranks in front in the start time order. We organize the end time order as a doubly linked list (DLL) so that any item can be deleted with $O(1)$ cost. The formal index construction is shown in Algorithm 3. We start with the head node h and end time order O (lines 1–2). In DLL, each item is connected with its previous and next item in O (lines 5–6). The initial out-neighbor (in-neighbor) is the next (previous) item in the DLL (lines 7–8). Note that the in-neighbor list of each node is maintained for efficiently computing successors, which will be discussed later. Then we prepare the start time order O' and iteratively delete items from the DLL. The previous and next item of v in DLL are connected directly, and also becomes the new out-neighbor and in-neighbor of each other (lines 12–17). Given $u.end < v.end < w.end$, w is an out-neighbor of u since $w.start > \min(u.start, v.start) = v.start$ (v is deleted earlier). The construction completes after deleting all items in O'.

Example 5.1. In Figure 3, the first node in the start time order O', $v = [1, 5]$, is deleted from DLL, making $u = [2, 4]$ become the previous node of $w = [3, 7]$. We add w to the out-neighbor list of u and u to the in-neighbor list of w. In the next round, we delete u by connecting h to w, and w become the second out-neighbor of h.

Computing Successors. We propose a linear-time solution for computing successors. We observe that both in-neighbors and out-neighbors are organized in increasing order of start times after construction. The successors of in-neighbors also grow in a non-decreasing order. This motivates us to compute the successors of all in-neighbors by scanning out-neighbors only once for each node v. We implement this idea with two pointers i, j for in-neighbors and out-neighbors, respectively. In Algorithm 3 (line 18), for each node v, we start from the beginning of $N_{in}(v)$ and $N_{out}(v)$ (lines 19–21). If the current out-neighbor w has a smaller start time than u, v, we move j to the next one (line 22). Otherwise, we assign j (the offset) as the successor of (u, v) and move i to the next one (line 24). Specially, some of the successor of $N_{in}(v)$ are still Null while j has been moved to the end of $N_{out}(v)$. We need to set the last out-neighbor of v as the successor of these edges, or set Null if v has no out-neighbors (lines 25-28).

Example 5.2. In Figure 3, we start from $u = [2, 4]$ and compute successors for its incoming edges. The first in-neighbor and the first out-neighbor of u are h and $v = [1, 5]$, respectively. Since $v.start \geq \min(h.start, u.start) = 2$, v cannot be the successor for (h, u). We then move to the second out-neighbor of u, i.e., $w = [3, 7]$, keeping the same in-neighbor h. We assign $N_{out}(h)[0].successor = 1$, indicating that the successor is the second out-neighbor of u. After processing all in-neighbors of u, the iteration moves to $v = [1, 5]$, where the successor of (u, v) is also 0, as $w.start \geq \min(u.start, v.start) = 1$.

Theorem 5.3. *The time complexity of Algorithm 3 is $O(n)$.*

5.2 Subgraph Insertion

Real-world temporal graphs are commonly modeled as edge streams with continuously arriving edges. Each incoming edge has an arrival time larger than all existing timestamps. Consequently, any new subgraph must include this edge and have an end time greater than all existing subgraphs. We focus on adding a single subgraph (node). When multiple new nodes share the same end time, we assume they are inserted in increasing order of start time; otherwise, a radix sort can be used to enforce this order in linear time. The running time is bounded by the size of the changed values in our index, which is optimal in the context.

Given a new node v, v must be the last one in the end time order so that there is no outgoing edge from v in TEMS-Graph. The insertion of v only produces new incoming edges to v in TEMS-Graph. These new edges do not have successors because v is the last node in the end-time order. Therefore, we mainly discuss two tasks to insert a new node v: (1) which nodes are the in-neighbors of v, and (2) for each incoming edge e of v, whose successor is e.

Algorithm 3: TEMS-Construct

```
   Input: a subgraph set S
   Output: TEMS-Graph TG
 1 h.start ← ∞, h.end ← −∞;
 2 O ← h and all items in S in increasing order of end times;
   /* construct the TEM-Graph                         */
 3 foreach g ∈ O do  N_in(g) ← ∅, N_out(g) ← ∅ ;
 4 foreach 0 < i < |O| do
 5     u ← O[i − 1], v ← O[i];
 6     u.next ← v, v.pre ← u;
 7     N_in(v).push(⟨u, |N_out(u)|⟩);
 8     N_out(u).push(⟨v, Null⟩);
 9 O′ ← items in S by increasing order of start times;
10 foreach v ∈ O′ do
11     u ← v.pre, w ← v.next;
       /* delete v and create edge (u,w)      */
12     if w ≠ Null then w.pre ← u;
13     if u ≠ Null then u.next ← w;
14     if u = Null ∨ w = Null then continue;
15     l ← |N_out(u)|;
16     N_in(w).push(⟨u, l⟩);
17     N_out(u).push(⟨w, Null⟩);
   /* computing successors                            */
18 foreach v ∈ O do
19     i ← 0, j ← 0;
20     while i < |N_in(v)| and j < |N_out(v)| do
21         ⟨u, x⟩ ← N_in(v)[i], ⟨w, y⟩ ← N_out(v)[j];
22         if w.start < min(u.start, v.start) then j ← j + 1;
23         else
24             N_out(u)[x].successor ← j, i ← i + 1;
25     foreach i < |N_in(v)| do
26         ⟨u, x⟩ ← N_in(v)[i];
27         if N_out(v) = Null then
               N_out(u)[x].successor ← Null ;
28         else N_out(u)[x].successor ← |N_out(v)| − 1;
```

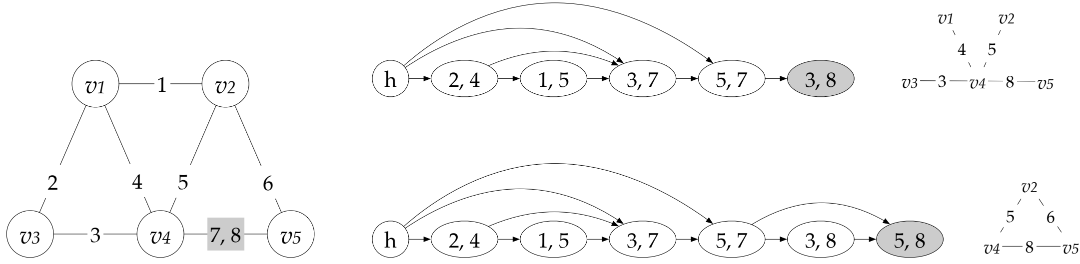


Figure 4: TEMS-Graph constructed from temporal subgraphs in Figure 1. Nodes and edges are inserted into TEMS-Graph when new temporal subgraphs are formed by arriving edges in the original temporal graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$.

Algorithm 4: Insert

Input: TEMS-Graph $\mathcal{TG}$, and a new subgraph w
Output: updated $\mathcal{TG}$

/* insert w into DLL */
1 $v \leftarrow O[|O| - 1]$;
2 $v.next \leftarrow w, w.pre \leftarrow v$;
/* create new edges to w */
3 **while** $v \neq$ Null **do**
  /* update the successors */
4   **foreach** i *from* $|N_{in}(v) - 1|$ *to* 0 **do**
5     $\langle u, j\rangle \leftarrow N_{in}(v)[i]$;
6     $l \leftarrow N_{out}(u)[j].successor$;
7   **if** $N_{out}(v)[l].start > \min(u.start, v.start)$ **then break**;
8     $N_{out}(u)[j].successor \leftarrow |N_{out}(v)|$;
9   $N_{in}(w).push(\langle v, |N_{out}(v)|\rangle)$;
10   $N_{out}(v).push(\langle w, \text{Null}\rangle)$;
11   **if** $v.start \geq w.start$ **then break**;
12   $v \leftarrow N_{in}(v)[|N_{in}(v)| - 1].id$;

Constructing New Edges. The previous node of the new node v in the end-time order is the first in-neighbor of v. We start from the initial in-neighbor and compute all in-neighbors progressively. We say the *latest incoming edge* of a node v as an incoming edge (u, v) such that the start time of u is maximized, and u in this case is called the *latest in-neighbor* of v. The last in-neighbor is clearly the front-most node in the end time order that is linked to v. Our idea to compute new edges is guided by the following lemmas.

Lemma 5.4. *Given an incoming edge (v, w) of w with $v.start < w.start$, there exists an incoming edge (u, w) of w such that (u, v) is the latest incoming edge of v.*

Lemma 5.5. *Given an incoming edge (u, w) of w where u and w are not adjacent in the end time order, there exists an incoming edge (v, w) of w such that (u, v) is the latest incoming edge of v.*

Both Lemma 5.4 and Lemma 5.5 are inferred by the second condition of Definition 4.1. Now assume we have a new node w. We start from the initial in-neighbor v of w and then locate the latest in-neighbor u of v. We add u as an in-neighbor of w. We repeat this step until we find the first u with $u.start > w.start$. Lemma 5.4 shows expanding from one incoming edge to the other incoming edge is safe and the result is correct. Lemma 5.5 guarantees any incoming edge can be decomposed into one latest incoming edge and one existing incoming edge, which proves the result is complete.

Updating Successors. For each new edge (v, w), we need to identify the incoming edge of v whose successor is (v, w). Before insertion, the incoming edge (u, v) of v has three possible successor cases: (1) it has a successor (v, w') with $\min(u.start, v.start) < w'.start$; (2) it has a successor (v, w') with $\min(u.start, v.start) > w'.start$, where w' is the last out-neighbor of v; (3) the successor is Null. We show by the following lemma that the successors of all edges in the first case remain unchanged.

Lemma 5.6. *For any in-neighbor v of the last node w in the end time order, w has the latest start time among all out-neighbors of v.*

Assume a new edge (v, w) is added. The start time of w must be later than w'. Therefore, (v, w') is still the successor of (u, v). Case 2 happens when we cannot find an outgoing edge meeting the first condition in Definition 4.10. The last outgoing edge of v is assigned as the successor of (u, v). When a new edge (v, w) is inserted, all edges falling in case 2 should change their successors to (v, w). Note that some case 2 edges may be transformed to case 1 edges after the update when $\min(u.start, v.start) < w.start$. The third case happens only when v is the last node in the original TEMS-Graph. Now w is the last one and v becomes the second last one. The successors of all case 3 edges should be the only outgoing edge (v, w). Recall that the incoming edges of each node are organized in increasing order of the start time of the source nodes. Therefore, all case 3 edges must appear behind case 2 edges, and all case 2 edges appear behind case 1 edges in order. This observation motivates us to scan the incoming edges of v in reverse order. We update the successor for each edge falling in cases 3 and 2. The iteration terminates when a case 1 edge is found.

We show the pseudocode in Algorithm 4. Firstly, we append w at the end of O. Before creating each edge, we check $u \in N_{in}(v)$ backwardly to see whether a new successor evolves (line 4–8). The check stops earlier when we meet an in-neighbor u whose current successor falls in case 1 (line 7). For case 2 and case 3 edges, the successors are changed to the offset of w in $N_{out}(v)$ (line 8). After that, we create a new edge (v, w) and check if it is the latest incoming edge of w (lines 9–11). If not, we move to the latest in-neighbor of v and repeat the process.

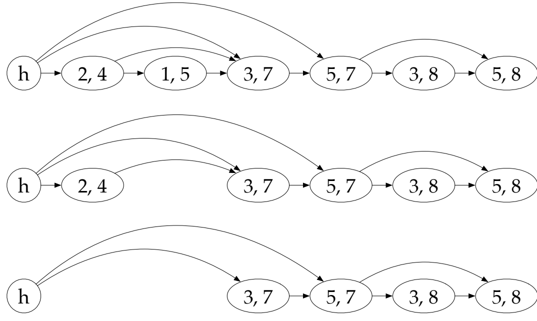


Figure 5: Delete nodes and edges from TEMS-Graph after all subgraphs before $t_o = 2$ become outdated.

Example 5.7. In Figure 4, a new edge between v_4 and v_5 arrives at time 8, forming two subgraphs: 4-star $\{v_4, v_1, v_2, v_3, v_5\}$ ([3, 8]) and triangle $\{v_2, v_4, v_5\}$ ([5, 8]). We check the successor of the incoming edges of $v = [5, 7]$ reversely. The successor of (h, v) is Null (case 3) because v does not have any out-neighbor now. We change this successor to 0 and repeat this operation to another in-neighbor $u' = [3, 7]$. The successor update for v is done. Now we can insert $w = [3, 8]$ and set its initial in-neighbor to $v = [5, 7]$. No other in-neighbor exists because (v, w) is the *latest incoming edge* of w. In the next round, the successor of (v, w) is changed from Null to 0 after inserting $w' = [5, 8]$. w' has two in-neighbors w and v, where (v, w') is the *latest incoming edge.*

Let Δ be the size of changed values in the TEMS-Graph after the insertion, i.e., Δ is the sum of the number of in-neighbors of w and the number of edges whose successors are incoming edges of w in Algorithm 4. We summarize the time complexity below.

Theorem 5.8. *Given a new node w with the latest end time, the time complexity of inserting w into the TEMS-Graph is $O(\Delta)$.*

5.3 Subgraph Deletion

We discuss how to maintain the index when the earliest edge expires. This edge appears in all subgraphs with the earliest start time, so the corresponding nodes must be removed from the index. As in insertion, we focus on deleting a single node from TEMS-Graph. When multiple expired nodes share the same start time, we always delete the head node in the start-time order. The strategy enables certain elegant properties for efficient deletion.

Locating the head. A straightforward method is to directly maintain an array by increasing order of start times, but this way incurs extra maintenance costs when a new node (with the latest end time) is inserted. We maintain all nodes in a two-dimensional array by increasing order of their start times. The offset in the first dimension corresponds to the start times. Nodes with the same start time are arranged in the same array by increasing order of their end times. When a new node w is inserted, we locate the corresponding array for the start time $w.start$ and add w to the end because the end time of w must be the latest in the array. Now we can sequentially process nodes in the two-dimensional array, which guarantees the head node in the start time order is always deleted.

Deleting edges from TEMS-Graph. Given the head node v in the start-time order, deleting v requires removing the directed edges incident to it and identifying the affected successors. Let u and w be the predecessor and successor of v in the end-time order. Then v has exactly one in-neighbor u and one out-neighbor w. We delete v from the out-neighbor list of u and from the in-neighbor list of w, as shown in Figure 5. Since v has the earliest start time, it is the first element in both lists, allowing these deletions to be performed in constant time. We implement the deletion by using an indicator to record the start position of the neighbor list. We increase the indicator by one to remove the first value. In this way, we do not incur extra cost in deletion and query processing (scanning neighbor lists from the indicator). Now for the two deleted edges, (u, v) cannot be the successor of any remaining edges because v has the earliest start time. (v, w) cannot be the successor of any remaining edges because all in-neighbors of v are deleted.

Theorem 5.9. *Given the head node w in the start time order, deleting w from TEMS-Graph takes $O(1)$ time.*

6 SUPER-VALID QUERY PROCESSING

The general idea of handling super-valid queries is to scan nodes in the decreasing order of end times while avoiding nodes with start times larger than the query start time. To this end, we make the following modification and call the index Sup-TEMS-Graph.

Definition 6.1 (Sup-TEMS-Graph). Given a set of temporal subgraphs S, the Sup-TEMS-Graph is a directed graph as follows:

(1) Nodes correspond to subgraphs in S;

(2) Given two nodes (subgraphs) u and v, v is an out-neighbor of u if (i) $v.end < u.end$; and (ii) there does not exist $w \in S$ such that $u.end > w.end > v.end$ and $w.start < \max(u.start, v.start)$;

(3) Out-neighbors of each vertex are arranged in decreasing order of end times.

(4) Given a vertex u and an out-neighbor v of u, the successor of the edge (u, v) is an edge (v, w) from v such that (1) the start time of w is not bigger than $\max(u.start, v.start)$; and (2) there does not exist an edge (v, w') such that $\min(u.start, v.start) \geq w'.start > w.start$.

Condition (2) assures that the out-neighbors of a node u are arranged in decreasing order of start times. We also create a virtual head node h in Sup-TEMS-Graph with to help search the first valid item u, where $h.start = -\infty, h.end = \infty$. After that, we can find the first out-neighbor of u whose start time is smaller than t_s as the next valid node by the successor, as illustrated in Figure 6.

The pseudocode is shown in Algorithm 5. We binary search $N_{out}(h)$ to find the first valid node u with $u.start < t_s$. In later rounds, we use the successors to locate the next valid item (lines 6–7). The query terminates when we reach the last valid item in the decreasing end time order, or there is no more valid item satisfying the time constraint (line 9). The space and query time complexity is the same as Theorem 4.12. The algorithms for index construction, subgraph insertion, and subgraph deletion are also similar to those for sub-valid queries. All their time complexities still hold. We omit details given the space limitation.

Other Allen's Predicates. Our methods also support other Allen's predicates [2, 9]. For example, the Overlaps query retrieves all the

Algorithm 5: Super-Query

Input: the query window $[t_s, t_e]$ and the Sup-TEMS-Graph $\mathcal{TG}'$

Output: super-valid subgraphs in $[t_s, t_e]$

```
1  u ← FindNextValid′(ts, te, TG′, h);
2  if u = Null then return Null;
3  output u;
4  i ← N_out(h)[l].successor, where l is the offset of u in N_out(h);
5  while i ≠ Null do
6      while i > 0 ∧ N_out(u)[i − 1].start ≤ ts do
7          i ← i − 1;
8      v ← N_out(u)[i].id;
9      if v.start > ts ∨ v.end < te then break;
10     i ← N_out(u)[i].successor;
11     u ← v;
12     output u;
```

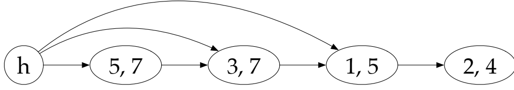


Figure 6: Sup-TEMS-Graph constructed from temporal subgraphs in Figure 1.

subgraphs where the interval overlaps with the query window. We can trivially modify Algorithm 5 to answer the Overlaps query, with the complexity also bounded by $O(k+\log n)$. The Starts query requires $t_s = s.start \wedge t_e < s.end$, which is a special case of super-valid query. Similarly, the Finishes query ($t_e = s.end \wedge t_s > s.start$) is also a special case. Both types can be supported by Sup-TEMS-Graph. To sum up, other predicates are easy to support and can be straightforwardly extended by the techniques studied in this paper.

7 PERFORMANCE STUDIES

We compare the query efficiency, update efficiency and index cost of our method against state-of-the-art methods. All algorithms are implemented in C++, compiled with the g++ compiler at the -O3 optimization level, and run on a Linux machine with an Intel Xeon 2.80GHz CPU and 500GB RAM.

Datasets. We evaluate our approach on six real-world temporal graphs from the SNAP repository[1] and generate the temporal subgraphs by uniform sampling common subgraph patterns (e.g., triangle, square, k-clique, and k-star). Table 2 summarizes the dataset characteristics. For each dataset, we report its time domain, the number of vertices and edges in $\mathcal{G}$, and the size of the subgraph set ($O(n)$ in complexity analysis). These datasets capture diverse types of interactions, and collectively cover a broad range of temporal and structural characteristics, with time spans from 193 to 2,774 days (7.6 years). They also vary significantly in density. College and Math are relatively sparse, while Email and Wiki exhibit dense cores formed by high-degree users.

[1] https://snap.stanford.edu/data/

Table 2: Characteristics of real datasets.

Datasets	Domain (D) [days]	#Vertices $\lvert\mathcal{V}\rvert$	#Edges $\lvert\mathcal{E}\rvert$	#Subgraphs [$\times 10^6$]	Avg D_s/D [%]
College	193	1,899	59,835	239	21.30
Email	803	986	332,334	1,469	47.97
Math	2,350	24,818	506,550	1,091	46.44
Super	2,773	194,085	1,443,339	1,758	31.80
Wiki	2,320	1,140,149	7,833,140	4,055	27.04
Stack	2,774	2,601,977	63,497,050	4,295	27.21

Queries. We generate query groups by fixing the query start time t_s and varying the window size (extent). t_s is randomly sampled from the time domain. The window sizes range from 1 hour to 90 days (approximately one quarter), covering diverse time spans commonly observed in real-world applications. Each query group contains 1,000 queries, which are executed sequentially. We report the average time cost per query. We also report the corresponding hit rate, defined as the ratio of valid results to the total number of subgraphs accessed during query processing. The number of valid subgraphs in each query group is annotated above the histograms.

Baselines. We compare TEMS-Graph with interval tree[2] and HINT[3]. We also report the query time of an index-free method (Brute-force), where subgraphs are sorted by increasing order of start times. For sub-validity queries, we perform a binary search to find the first subgraph with $start \geq t_s$, then scan sequentially to collect the valid results until $start > t_e$. For super-validity queries, we scan from the beginning until encountering a subgraph with $start > t_s$.

Exp-1: Test on sub-validity query. We report the results in Figure 7. As the query window expands, the running time increases because larger windows usually include more valid results. Interval tree cannot effectively prune results for Overlaps queries, so its running time is dominated by many intersecting but non-sub-valid records. For small query windows, TEMS-Graph significantly outperforms the baselines, achieving 1–2 orders of magnitude speedup. This advantage is reflected in the hit rates: while all three baselines remain below 10%, TEMS-Graph achieves over 40%. This is because TEMS-Graph can directly jump to valid results through out-neighbors, avoiding unnecessary visits to invalid entries. When the query window reaches 90 days, the performance gap narrows across all methods. In this regime, brute-force scanning becomes a simple and effective solution, avoiding index space overhead and benefiting from good cache locality during sequential access.

Exp-2: Test on super-validity query. The performance is shown in Figure 8. Due to space limitations, we omit results that exhibit the same trend. The running time decreases as the query window widens because fewer subgraphs fully cover longer time spans. HINT exhibits a sharp drop in hit rate, which leads to lower query efficiency than the other baselines. This behavior is evident on the Wiki dataset, where the hit rate falls from nearly 100% to about 10% for large windows. In contrast, TEMS-Graph consistently outperforms all baselines and scales well with dataset size. This advantage stems from the structure of Sup-TEMS-Graph, where edges directly connect super-valid results, allowing efficient skipping of irrelevant candidates. Moreover, the optimal $O(k + \log n)$ query complexity of

[2] https://github.com/ekg/intervaltree
[3] https://github.com/pbour/hint.

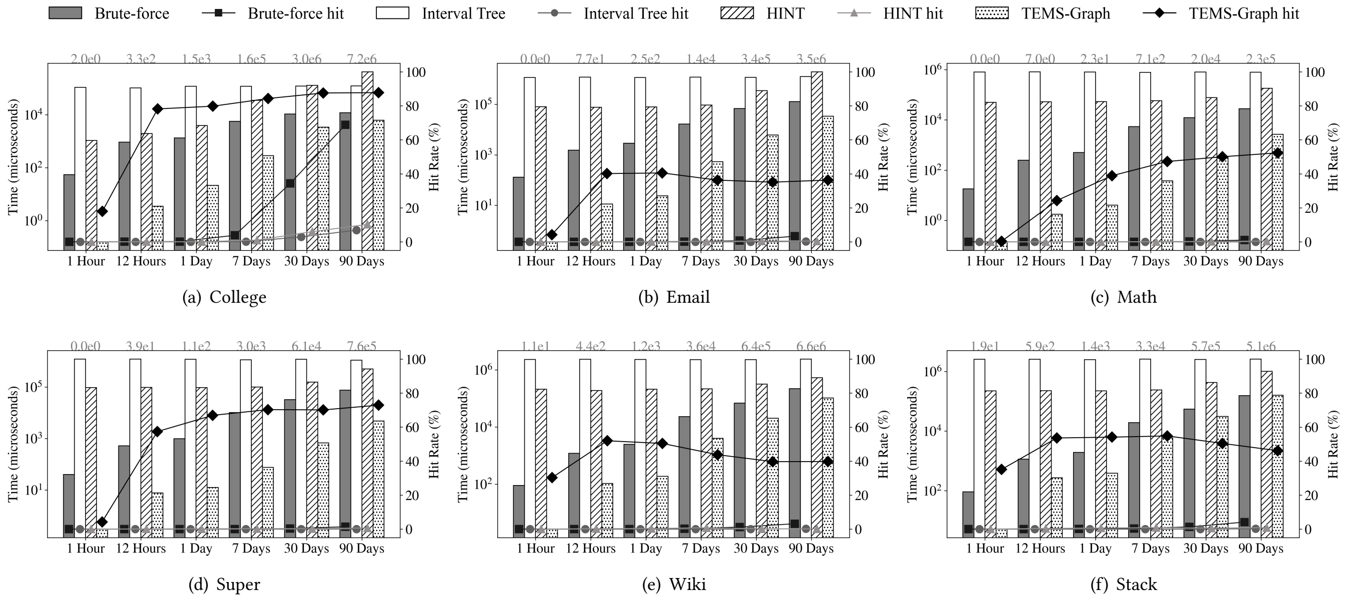


Figure 7: Comparing running time and hit rate of sub-validity query with different query window extents

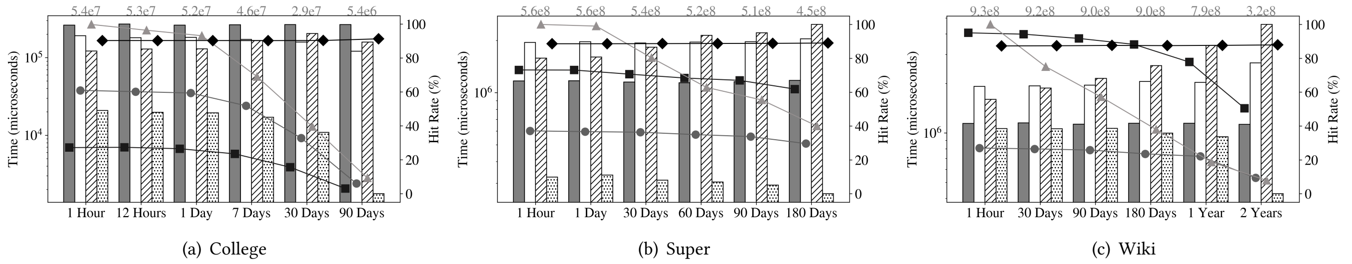


Figure 8: Comparing running time and hit rate of super-validity query with different query window extents

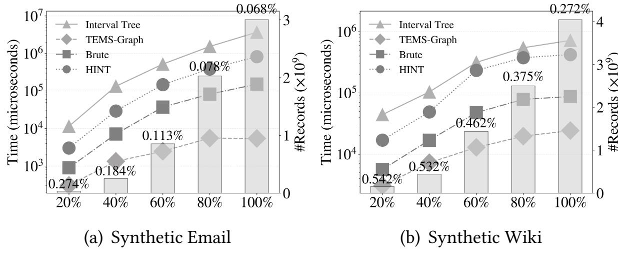
Sup-TEMS-Graph ensures a stable hit rate, averaging around 80% across all test cases.

Exp-3: Test on synthetic datasets. We investigate the impact of graph size and density on query performance. Similar to [1], we fix the vertex set and time domain of the original datasets and sample 20%, 40%, ..., 100% of the edges. We then report the average running time of the mixed query workload used in previous experiments, as shown in Figure 9. The number of subgraphs in each dataset is shown as a histogram and the proportion of valid results is also reported. As graph density increases, the number of valid results grows, leading to longer running times for all methods. The performance gap between TEMS-Graph and the baselines widens on denser graphs, demonstrating the robustness of our approach.

Exp-4: Test on pattern query. We evaluate query performance when both a time window and a subgraph pattern type are specified. For comparison, the *Mix* method builds a single TEMS-Graph for all patterns, while the *Separate* method builds an individual TEMS-Graph for each pattern type. The query time and hit rate are reported in Figure 10. Across all datasets, *Separate* outperforms *Mix* by a factor of 2–5. This is because *Mix* first retrieves all records valid within the query window and then filters by pattern type, resulting in a lower hit rate and higher query cost.

Exp-5: Test on index construction. Figure 11 presents the index construction time and index space of each method. For Brute-force, the indexing time refers to the cost of sorting all subgraphs by start time, and its space cost consists only of subgraph IDs and their corresponding windows, serving as a baseline. The construction time of TEMS-Graph is comparable to that of interval tree, yet TEMS-Graph achieves significantly better query performance. HINT incurs the highest construction time across all datasets. Both interval tree and TEMS-Graph use linear space and demonstrate similar memory usage. In contrast, although HINT is also near-linear in space complexity, its overhead is higher due to an additional factor ϕ related to the domain size and the average subgraph window length, resulting in the highest space consumption among the methods.

Exp-6: Test on index update. To simulate batch updates as new subgraphs are formed by the latest temporal edges, we split each



(a) Synthetic Email (b) Synthetic Wiki

Figure 9: Comparing the running time with different densities on the synthetic datasets.

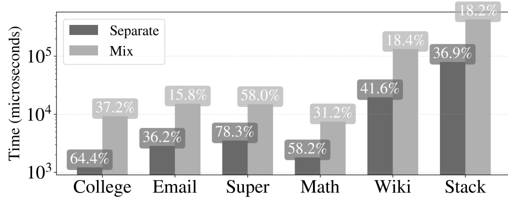


Figure 10: Comparing the running time and hit rate for window and pattern queries.

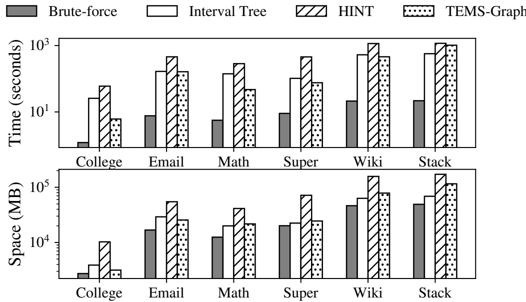
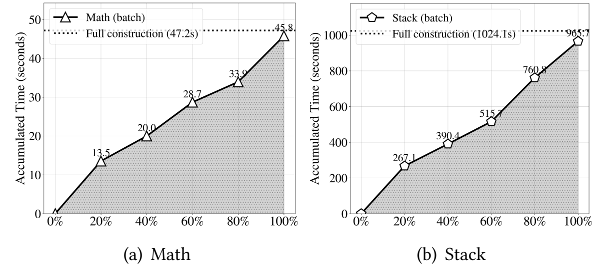


Figure 11: Comparing the index construction time and index space cost (memory usage).

data set into five parts according to the end-time order. In Figure 12, we report the accumulated index construction time as data is incrementally loaded from 0% to 100%. For comparison, the full construction time using the entire dataset is shown as a dotted horizontal line. The index update time grows linearly with data size and eventually converges to the full construction time, which aligns with our theoretical analysis in Theorem 5.8.

Following the previous works [8, 9], we also report the average insertion and deletion time per subgraph in Table 3. We first index 80% of the original subgraph set, then insert the remaining 20%. To evaluate deletion, we select an outdated timestamp in each dataset such that 20% of the subgraphs are removed from the index. In all datasets, the average cost remains below 1 microsecond. The domain-expanded version of HINT is discussed in [10], but it only implements OVERLAPS query and do not support deletions yet. Thus, we omit the comparison here. We also measure the time cost of generating subgraphs per incoming edge using a basic streaming



(a) Math (b) Stack

Figure 12: Accumulated indexing time with batch updates compared to full index construction on the entire dataset.

Table 3: Average time cost (microseconds) per subgraph of updating TEMS-Graph and average cost (seconds) to generate subgraphs per incoming edge.

Dataset	College	Email	Math	Super	Wiki	Stack
Construction	0.026	0.111	0.027	0.043	0.113	0.238
Insertion	0.051	0.064	0.061	0.043	0.105	0.310
Deletion	0.021	0.028	0.011	0.017	0.026	0.078
Streaming Edge	0.015	0.034	1.166	1.869	6.969	25.328

graph model. In dense or large-scale graphs, subgraph enumeration becomes the primary bottleneck, while updating TEMS-Graph remains extremely fast with negligible overhead. Existing continuous subgraph matching techniques [42, 43, 50] can be integrated to further enhance scalability.

8 CONCLUSION

Given a temporal subgraphs set, we propose a novel index structure to efficiently retrieve all subgraphs contained (sub-valid) or containing (super-valid) an arbitrary query time window. Our index achieves near-optimal query complexity with linear index size. We also design efficient index construction and maintenance algorithms to support subgraph updates with optimal cost. Experiments on real-world datasets demonstrate that our algorithm can achieve a speedup of up to two orders.

Our work can be extended to support additional Allen's temporal predicates. From the interval-index perspective, we are working on efficient maintenance algorithms to handle out-of-order insertions and deletions (ad hoc interval updates). Developing either a theoretically bounded or a practically efficient solution would make our index a general-purpose dynamic interval index.

ACKNOWLEDGMENTS

Yaping Liu is supported by New Generation Artificial Intelligence-National Science and Technology Major Project 2025ZD0122203. Dian Ouyang is supported by NSFC No.62502105. Yikun Wang is supported by China Scholarship Council No.202308440225. Dong Wen is supported by ARC DP260100709 and ARC DE240100668. Wenjie Zhang is supported by ARC DP230101445 and FT210100303. Xuemin Lin is supported by NSFC U2241211 and U20B2046.

REFERENCES

[1] Amir Aghasadeghi, Jan Van Den Bussche, and Julia Stoyanovich. 2024. Temporal graph patterns by timed automata. *The VLDB Journal* 33, 1 (Jan. 2024), 25–47.

[2] James F Allen. 1983. Maintaining knowledge about temporal intervals. *Commun. ACM* 26, 11 (1983), 832–843.

[3] Michael J. Bannister, Christopher DuBois, David Eppstein, and Padhraic Smyth. 2013. Windows into Relational Events: Data Structures for Contiguous Subsequences of Edges. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*. SIAM, 856–864.

[4] Andreas Behrend, Anton Dignös, Johann Gamper, Philip Schmiegelt, Hannes Voigt, Matthias Rottmann, and Karsten Kahl. 2019. Period index: A learned 2d hash index for range and duration queries. In *Proceedings of the 16th International Symposium on Spatial and Temporal Databases*. 100–109.

[5] Farah Chanchary, Anil Maheshwari, and Michiel H. M. Smid. 2020. Querying relational event graphs using colored range searching data structures. *Discret. Appl. Math.* 286 (2020), 51–61.

[6] Bernard Chazelle. 1988. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.* 17, 3 (1988), 427–462.

[7] Melisachew Wudage Chekol, Giuseppe Pirrò, and Heiner Stuckenschmidt. 2019. Fast Interval Joins for Temporal SPARQL Queries. In *Companion of The 2019 World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*. ACM, 1148–1154.

[8] George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. 2022. HINT: A Hierarchical Index for Intervals in Main Memory. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 1257–1270.

[9] George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. 2024. HINT: a hierarchical interval index for Allen relationships. *The VLDB Journal* 33, 1 (2024), 73–100.

[10] George Christodoulou, Panagiotis Bouros, and Nikos Mamoulis. 2024. LIT: Lightning-fast In-memory Temporal Indexing. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–27.

[11] Mark de Berg. 2000. *Computational geometry: algorithms and applications, 2nd Edition*. Springer.

[12] Ariel Debrouvier, Eliseo Parodi, Matías Perazzo, Valeria Soliani, and Alejandro Vaisman. 2021. A model and query language for temporal graph databases. *The VLDB Journal* 30, 5 (2021), 825–858.

[13] Shiyuan Deng, Shangqi Lu, and Yufei Tao. 2023. Space-Query Tradeoffs in Range Subgraph Counting and Listing. In *26th International Conference on Database Theory, ICDT 2023, March 28-31, 2023, Ioannina, Greece (LIPIcs)*, Vol. 255. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:25.

[14] Shiyuan Deng, Shangqi Lu, and Yufei Tao. 2023. Space-query tradeoffs in range subgraph counting and listing. *arXiv preprint arXiv:2301.03390* (2023).

[15] Anton Dignös, Michael H. Böhlen, Johann Gamper, and Christian S. Jensen. 2016. Extending the Kernel of a Relational DBMS with Comprehensive Support for Sequenced Temporal Queries. *ACM Trans. Database Syst.* 41, 4 (2016), 26:1–26:46.

[16] Herbert Edelsbrunner. 1983. A new approach to rectangle intersections part I. *International Journal of Computer Mathematics* 13, 3-4 (1983), 209–219.

[17] Edoardo Galimberti, Alain Barrat, Francesco Bonchi, Ciro Cattuto, and Francesco Gullo. 2018. Mining (maximal) Span-cores from Temporal Networks. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management, CIKM 2018, Torino, Italy, October 22-26, 2018*. ACM, 107–116.

[18] Zhongqiang Gao, Chuanqi Cheng, Yanwei Yu, Lei Cao, Chao Huang, and Junyu Dong. 2022. Scalable Motif Counting for Large-scale Temporal Graphs. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 2656–2668.

[19] Qiuyu Guo, Jianye Yang, Wenjie Zhang, Hanchen Wang, Ying Zhang, and Xuemin Lin. 2025. Efficient and Accurate Subgraph Counting: A Bottom-up Flow-learning based Approach. *Proc. VLDB Endow.* 18, 8 (2025), 2695–2708.

[20] Shuping Ji and Hans-Arno Jacobsen. 2018. PS-Tree-based Efficient Boolean Expression Matching for High Dimensional and Dense Workloads. *Proc. VLDB Endow.* 12, 3 (2018), 251–264. https://doi.org/10.14778/3291264.3291270

[21] Shuping Ji and Hans-Arno Jacobsen. 2021. A-Tree: A Dynamic Data Structure for Efficiently Indexing Arbitrary Boolean Expressions. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 817–829.

[22] Jiawei Jiang, Hao Huang, Zhigao Zheng, Yi Wei, Fangcheng Fu, Xiaosen Li, and Bin Cui. 2025. Detecting and Analyzing Motifs in Large-Scale Online Transaction Networks. *IEEE Trans. Knowl. Data Eng.* 37, 2 (2025), 584–596.

[23] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter Michael Fischer, Donald Kossmann, Franz Färber, and Norman May. 2013. Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1173–1184.

[24] Udayan Khurana and Amol Deshpande. 2019. Historical Graph Management. In *Encyclopedia of Big Data Technologies*. Springer.

[25] Dongjin Lee, Kijung Shin, and Christos Faloutsos. 2020. Temporal locality-aware sampling for accurate triangle counting in real graph streams. *The VLDB Journal* 29, 6 (2020), 1501–1525.

[26] D. T. Lee and C. K. Wong. 1980. Quintary Trees: A File Structure for Multidimensional Database Systems. *ACM Trans. Database Syst.* 5, 3 (1980), 339–353.

[27] Xiaoyu Leng, Guang Zeng, Hongchao Qin, Longlong Lin, and Rong-Hua Li. 2025. On Temporal-Constraint Subgraph Matching. In *41st IEEE International Conference on Data Engineering, ICDE 2025, Hong Kong, May 19-23, 2025*. IEEE, 2493–2506.

[28] Faming Li and Zhaonian Zou. 2021. Subgraph matching on temporal graphs. *Inf. Sci.* 578 (2021), 539–558.

[29] Faming Li, Zhaonian Zou, and Jianzhong Li. 2023. Durable Subgraph Matching on Temporal Graphs. *IEEE Trans. Knowl. Data Eng.* 35, 5 (2023), 4713–4726.

[30] Shunyang Li, Kai Wang, Xuemin Lin, Wenjie Zhang, Yizhang He, and Long Yuan. 2024. Querying Historical Cohesive Subgraphs Over Temporal Bipartite Graphs. In *ICDE*. IEEE, 2503–2516.

[31] Youhuan Li, Lei Zou, M Tamer Özsu, and Dongyan Zhao. 2019. Time constrained continuous subgraph search over streaming graphs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1082–1093.

[32] Youhuan Li, Lei Zou, M Tamer Özsu, and Dongyan Zhao. 2020. Space-efficient subgraph search over streaming graph with timing order constraint. *IEEE Transactions on Knowledge and Data Engineering* 34, 9 (2020), 4453–4467.

[33] Penghang Liu, Valerio Guarrasi, and Ahmet Erdem Sariyüce. 2023. Temporal Network Motifs: Models, Limitations, Evaluation. *IEEE Trans. Knowl. Data Eng.* 35, 1 (2023), 945–957.

[34] Min Lu, Qianzhen Zhang, and Xianqiang Zhu. 2025. Temporal Multi-Query Subgraph Matching in Cybersecurity. *Technologies* 13, 8 (2025), 335.

[35] Pietro Panzarasa, Tore Opsahl, and Kathleen M. Carley. 2009. Patterns and dynamics of users' behavior and interaction: Network analysis of an online community. *Journal of the American Society for Information Science and Technology* 60, 5 (2009), 911–932.

[36] Ashwin Paranjape, Austin R Benson, and Jure Leskovec. 2017. Motifs in temporal networks. In *Proceedings of the tenth ACM international conference on web search and data mining*. 601–610.

[37] Evaggelia Pitoura. 2017. Historical Graphs: Models, Storage, Processing. In *Business Intelligence and Big Data - 7th European Summer School, eBISS 2017, Bruxelles, Belgium, July 2-7, 2017, Tutorial Lectures (Lecture Notes in Business Information Processing)*, Vol. 324. Springer, 84–111.

[38] Hongchao Qin, Rong-Hua Li, Ye Yuan, Yongheng Dai, and Guoren Wang. 2023. Densest periodic subgraph mining on large temporal graphs. *IEEE Transactions on Knowledge and Data Engineering* 35, 11 (2023), 11259–11273.

[39] Mohammad Sadoghi and Hans-Arno Jacobsen. 2011. BE-tree: an index structure to efficiently match boolean expressions over high-dimensional discrete space. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*. ACM, 637–648.

[40] Hanan Samet. 1990. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley.

[41] Ilie Sarpe and Fabio Vandin. 2021. PRESTO: Simple and Scalable Sampling Techniques for the Rigorous Approximation of Temporal Motif Counts. In *Proceedings of the 2021 SIAM International Conference on Data Mining, SDM 2021, Virtual Event, April 29 - May 1, 2021*. SIAM, 145–153.

[42] Shixuan Sun, Xibo Sun, Bingsheng He, and Qiong Luo. 2022. Rapidflow: An efficient approach to continuous subgraph matching. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2415–2427.

[43] Xibo Sun, Shixuan Sun, Qiong Luo, and Bingsheng He. 2022. An in-depth study of continuous subgraph matching. *Proceedings of the VLDB Endowment* 15, 7 (2022), 1403–1416.

[44] Jingjing Wang, Yanhao Wang, Wenjun Jiang, Yuchen Li, and Kian-Lee Tan. 2020. Efficient Sampling Algorithms for Approximate Temporal Motif Counting. In *CIKM '20: The 29th ACM International Conference on Information and Knowledge Management, Virtual Event, Ireland, October 19-23, 2020*. ACM, 1505–1514.

[45] Li Wang, Ruichu Cai, Tom Z. J. Fu, Jiong He, Zijie Lu, Marianne Winslett, and Zhenjie Zhang. 2018. Waterwheel: Realtime Indexing and Temporal Range Query Processing over Massive Data Streams. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 269–280.

[46] Yuyang Xia, Yixiang Fang, and Wensheng Luo. 2025. Efficiently Counting Triangles in Large Temporal Graphs. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–27.

[47] Haoxuan Xie, Yixiang Fang, Yuyang Xia, Wensheng Luo, and Chenhao Ma. 2023. On Querying Connected Components in Large Temporal Graphs. *SIGMOD* 1, 2 (2023), 170:1–170:27.

[48] Jianye Yang, Sheng Fang, Zhaoquan Gu, Ziyi Ma, Xuemin Lin, and Zhihong Tian. 2024. Tc-match: Fast time-constrained continuous subgraph matching. *Proceedings of the VLDB Endowment* 17, 11 (2024), 2791–2804.

[49] Junyong Yang, Ming Zhong, Yuanyuan Zhu, Tieyun Qian, Mengchi Liu, and Jeffrey Xu Yu. 2023. Scalable Time-Range k-Core Query on Temporal Graphs. *Proc. VLDB Endow.* 16, 5 (2023), 1168–1180. https://doi.org/10.14778/3579075.3579089

[50] Rongjian Yang, Zhijie Zhang, Weiguo Zheng, and Jeffrey Xu Yu. 2023. Fast continuous subgraph matching over streaming graphs via backtracking reduction. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.

[51] Michael Yu, Dong Wen, Lu Qin, Ying Zhang, Wenjie Zhang, and Xuemin Lin. 2021. On querying historical k-cores. *Proceedings of the VLDB Endowment* (2021).

[52] Yuanhang Yu, Dong Wen, Michael Yu, Lu Qin, Ying Zhang, Wenjie Zhang, and Xuemin Lin. 2025. Querying historical K-cores in large temporal graphs. *VLDB J.* 34, 2 (2025), 26.

[53] Xiangjun Zai, Xingyu Tan, Xiaoyang Wang, Qing Liu, Xiwei Xu, and Wenjie Zhang. 2025. PRoH: Dynamic Planning and Reasoning over Knowledge Hypergraphs for Retrieval-Augmented Generation. *arXiv preprint arXiv:2510.12434* (2025).

[54] Ming Zhong, Junyong Yang, Yuanyuan Zhu, Tieyun Qian, Mengchi Liu, and Jeffrey Xu Yu. 2024. A Unified and Scalable Algorithm Framework of User-Defined Temporal $(k,\mathcal {X})$(k,X)-Core Query. *IEEE Trans. Knowl. Data Eng.* 36, 7 (2024), 2831–2845.

[55] Kaijie Zhu, George Fletcher, and Nikolay Yakovets. 2021. Leveraging temporal and topological selectivities in temporal-clique subgraph query processing. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 672–683.

[56] Kaijie Zhu, George H. L. Fletcher, Nikolay Yakovets, Odysseas Papapetrou, and Yuqing Wu. 2019. Scalable temporal clique enumeration. In *Proceedings of the 16th International Symposium on Spatial and Temporal Databases, SSTD 2019, Vienna, Austria, August 19-21, 2019*. ACM, 120–129.