Efficient Temporal Edge-Core Maintenance in Streaming Graphs

Tongfeng Weng
National University of Singapore
tf.weng@nus.edu.sg

Mo Sha*
Alibaba Cloud, Singapore
shamo.sm@alibaba-inc.com

Xu Zhou
Hunan University
zhxu@hnu.edu.cn

Jingjing Lu*
Hunan University
lujingjing000@hnu.edu.cn

Kenli Li
Hunan University
lkl@hnu.edu.cn

Kian-Lee Tan
National University of Singapore
tankl@comp.nus.edu.sg

ABSTRACT

Temporal graphs are critical for modeling dynamic systems where interactions evolve over time, with a central challenge being the characterization of structural cohesion. The *temporal edge-core*, defined under a temporal proximity constraint Δ, quantifies the stability and density of connections within subgraphs and is essential for applications such as anomaly detection and information diffusion. Existing edge-core decomposition methods, however, are designed for static graphs and are computationally prohibitive in *streaming environments* due to frequent edge arrivals and deletions. We present **TECM**, an efficient framework for streaming temporal edge-core decomposition that leverages the localized impact of edge updates within Δ-incident neighbors. **TECM** incrementally updates core values through Δ-aware traversals and localized H-index analysis, and incorporates batch processing to handle high-velocity streams. Extensive experiments on real and synthetic temporal networks demonstrate that **TECM** delivers speedups of several orders of magnitude over state-of-the-art static baselines, providing a scalable and principled solution for real-time structural analysis in evolving temporal graphs.

PVLDB Reference Format:
Tongfeng Weng, Mo Sha, Xu Zhou, Jingjing Lu, Kenli Li, and Kian-Lee Tan. Efficient Temporal Edge-Core Maintenance in Streaming Graphs. PVLDB, 19(4): 726 - 739, 2025. doi:10.14778/3785297.3785312

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/naruto110/EdgeCoreMaintenance.

1 INTRODUCTION

Modern dynamic systems increasingly rely on temporal graphs to model time-varying interactions [8, 9, 67]. These graphs capture essential temporal patterns such as bursty transactions in fraud detection [15, 28, 33], ephemeral social connections in information diffusion [57], and time-dependent routes in urban mobility [29, 62]. A fundamental challenge lies in identifying cohesive substructures that persist both structurally and temporally, underlying critical applications including community evolution tracking [12, 14, 22, 60], anomaly detection [2, 44], and epidemic forecasting [45]. Traditional cohesion metrics like k-core and k-truss, while effective in static graphs [4, 10, 27, 53], suffer from temporal blindness: They overlook temporal proximity constraints inherent in real-world interactions. For instance, fraudulent transaction clusters often manifest as tightly connected edges within narrow time windows—patterns invisible to static decomposition methods [28, 31, 50, 71]. The temporal edge core directly addresses this limitation by quantifying how tightly an interaction event is embedded within short-term, locally cohesive substructures. In fraud detection, high core values indicate coordinated transactions within narrow time windows, characteristic of collusion schemes. In misinformation scenarios, rapidly rising core values signal burst-like amplification typical of coordinated disinformation campaigns. Similarly, disease spread risks depend on temporally proximate contacts, necessitating analyses that integrate structural density with temporal locality [3, 35].

Recent proposals for temporal cohesiveness metrics, particularly the (k, Δ)-temporal edge-core [47], address this need by requiring edge endpoints to reside in substructures with at least k temporal neighbors within a Δ-duration window. However, real-world temporal graphs operate in dynamic, streaming environments where edges continuously arrive and expire at high velocities [20, 21].

In such settings, existing traversal-based (k, Δ)-core decomposition methods become computationally prohibitive, requiring full recomputation upon each update. Even state-of-the-art incremental algorithms for time-agnostic graph cores fail to adapt [11, 27, 40, 51, 73], as they ignore temporal dependencies: Update impacts propagate through structural adjacency rather than temporally bounded paths, causing excessive recomputations or missed updates. This inefficiency stems from a fundamental mismatch—static methods prioritize spatial locality, while temporal cohesion demands simultaneous consideration of structural density and time-constrained reachability. Consequently, maintaining (k, Δ)-cores in streaming contexts remains an open challenge, with existing approaches lacking theoretical guarantees on update locality and practical mechanisms to contain cascading effects along temporal pathways.

To overcome these limitations, we propose **Temporal Edge-Core Maintenance (TECM)**, a framework grounded in *temporally bounded propagation locality* and Δ-*reachable influence containment*. Central to **TECM** is a task decomposition strategy that partitions edge updates into localized sub-edges constrained by Δ-incident proximity. The effect of edge insertions or deletions propagates along Δ-incident adjacency chains—edges sharing a vertex and occurring within a Δ-length time window. By restricting propagation to these chains, **TECM** confines recomputation to nearby, temporally cohesive subgraphs and contains cascading effects.

*Corresponding authors.

Proceedings of the VLDB Endowment, Vol. 19, No. 4 ISSN 2150-8097.
doi:10.14778/3785297.3785312

This approach is grounded in a novel theory of Δ-reachable candidate subgraphs that characterizes the minimal set of edges affected by core updates. We formally prove that any edge subject to potential core changes must lie on a temporal path originating from the updated edge, where successive edges differ in timestamp by at most Δ and initially share the root's core value. This enables **TECM** to identify candidate subgraphs through a provably minimal BFS over Δ-incident adjacencies, avoiding exhaustive scans.

Further, **TECM** advances incremental peeling by unifying temporal support thresholds with dynamic H-index maintenance [42]. Unlike conventional static approaches, each edge's Δ-support—the minimum number of Δ-incident neighbors from its endpoints with core values at least equal to its own—acts as a stability criterion. During insertions, edges iteratively "ascend" to higher cores when their Δ-support exceeds their current core; during deletions, edges are "stripped" once their Δ-support falls below the threshold. To mitigate overcomputation, **TECM** employs lightweight pruning that preemptively eliminates edges with sufficient Δ-support during BFS traversal, reducing candidate subgraph sizes by up to 65.2%. This design ensures that per-update complexity scales linearly with candidate subgraph size rather than the entire graph.

TECM addresses the critical challenge of maintaining (k, Δ)-temporal edge-cores in high-velocity streaming graphs through *temporal propagation locality*, formalized via Δ-reachable chains. This enables a unified model that provably bounds update impacts within minimal candidate subgraphs identified through localized searches along Δ-incident adjacencies.

TECM presents a new paradigm for dynamic temporal graph analysis by using Δ-incident proximity as a computational localization mechanism. This turns temporal constraints from a liability into an efficiency advantage, enabling real-time maintenance of (k, Δ)-temporal edge-cores in settings where spatiotemporal cohesion matters, such as evolving fraud networks and event-driven social interactions. Our incremental framework resolves the previously open problem of updating (k, Δ)-cores under edge insertions and deletions, delivering orders-of-magnitude speedups over batch recomputation while raising alerts when edge core values cross predefined thresholds and returning candidate subgraphs for investigation. In doing so, it connects advanced temporal cohesion models with the operational needs of high-velocity streaming systems and provides a basis for future work in streaming graph analytics.

The main contributions of this work are as follows:

- We formalize the problem of maintaining (k, Δ)-temporal edge-cores in dynamic streaming graphs, addressing fundamental limitations of prior static baseline methods. By introducing Δ-incident adjacency chains, we show that update propagation is confined to structurally and temporally coherent neighborhoods, resolving temporal insensitivity in earlier approaches.
- We propose the **TECM** framework, which incrementally maintains temporal edge-cores via localized task decomposition, dynamic peeling guided by Δ-support thresholds, and candidate pruning techniques. This design guarantees correctness equivalent to full recomputation while substantially reducing overhead.
- We empirically evaluate **TECM** on diverse real-world temporal graph streams, demonstrating significant performance gains over batch baselines with speedup ratios up to 5 orders of magnitude.

Table 1: Summary of Notations

Notation	Description
$G = (V, E, \tau)$	A temporal graph
$e = (u, v, t)$	A temporal edge e at time $t \in \mathbb{N}$
$d(v)$	The degree of vertex v
$\Delta \in \mathbb{N}$	The temporal proximity constraint
$N_\Delta(e)$	The Δ-incident adjacent edges
$d_\Delta(e)$	The edge degree of edge e (i.e., $\lvert N_\Delta(e)\rvert$)
$H = (V', E')$	An edge-induced subgraph of G
$H_{(k,\Delta)}$	A (k, Δ)-core
$H^e_{(k,\Delta)}$	Maximal (k, Δ)-core containing edge e
$\theta(e)$	Edge core number (max k s.t. $e \in H_{(k,\Delta)}$)

2 PRELIMINARIES

2.1 Definitions

We define key concepts used throughout the paper, with notations summarized in Table 1. An undirected temporal graph is denoted as $G = (V, E, \tau)$, where V and E are the vertex and edge sets, and τ is the discrete time domain (the set of active time units with fixed granularity used throughout). Each edge $e = (u, v, t) \in E$ represents an interaction between vertices $u, v \in V$ at time $t \in \tau$, where t is a discrete timestamp measured in the fixed time units. For instance, $(u, v, \text{day-1})$ and $(u, v, \text{day-2})$ indicate meetings on two separate days. An edge-induced subgraph is denoted $H = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$.

The degree of a temporal edge e, denoted $d_\Delta(e)$ [47], is defined under a temporal proximity constraint Δ. Two edges are Δ-incident if they share an endpoint and their timestamps differ by at most Δ. The degree of e is the minimum number of Δ-incident edges from its two endpoints (Definition 2.1). We also define the temporal (k, Δ)-core and the edge core number $\theta(e)$ in Definitions 2.2 and 2.3.

Definition 2.1. ***Edge Degree*** d_Δ**.** *Given a temporal graph $G = (V, E, \tau)$, $\Delta \in \mathbb{N}$, the degree of an edge $e = (u, v, t)$ can be defined as*

$$d_\Delta(e) = \min\Big(\big|\{(u, w, t') \in E \mid |t - t'| \le \Delta\}\big|, \big|\{(v, w, t') \in E \mid |t - t'| \le \Delta\}\big|\Big). \quad (1)$$

The temporal proximity between two edges is used to describe the temporal closeness of two edges that derive from the same vertex (i.e., the common vertex w in Equation (1)). It is defined by the absolute difference in their timestamps, i.e., $|t - t'|$. If this temporal proximity does not exceed the given parameter Δ and the two edges share an endpoint, the two edges are considered Δ-incident. It is important to note that each edge is Δ-incident to itself. The Δ-incident adjacent edges of e are denoted as $N_\Delta(e)$.

Definition 2.2. ***Temporal*** (k, Δ)***-core.*** *Given a temporal graph $G = (V, E, \tau)$, an edge-induced subgraph $H = (V', E')$ is a temporal (k, Δ)-core iff for $e \in E'$ its induced edge degree in H is at least k:*

$$d^H_\Delta(e) \ \ge \ k \quad \text{for all } e \in E',$$

where $d^H_\Delta(e)$ counts the number of edges $e' \in E'$ that are Δ-incident to e, i.e., e and e' share an endpoint and $\big|\tau(e) - \tau(e')\big| \le \Delta$. We denote such a subgraph by $H_{(k,\Delta)}$.

Definition 2.3. ***Edge Core*** $\theta(e)$**.** *Given a temporal graph $G = (V, E, \tau)$ and the temporal proximity constraint Δ, the edge core of a temporal edge $e \in E$, denoted as $\theta(e)$, is the maximum value of k for which e is present in the (k, Δ)-core of G, denoted as $H^e_{(k,\Delta)}$.*

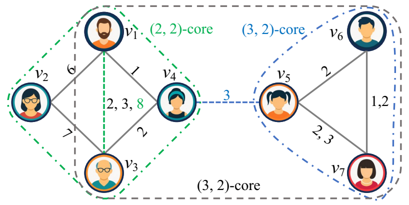

Figure 1: Temporal graph G with edge timestamps ($\Delta = 2$). The gray dashed box marks the initial $(3, 2)$-core.

Definition 2.4. ***H-index Property.*** *Given a temporal graph $G = (V, E, \tau)$ and Δ, let $N_\Delta(e, u)$ be the set of Δ-incident adjacent edges of $e = (u, v, t)$ involving endpoint u (including e itself). The* H-index *of a multiset of integers is defined as the maximum integer h such that there are at least h elements in the multiset that are greater than or equal to h. The H-index for endpoint u of edge e, denoted $e.u._{h\text{-}index}$, is defined as the H-index of the multiset of edge cores $\{\theta(e') \mid e' \in N_\Delta(e, u)\}$. The edge core $\theta(e)$ satisfies the property $\theta(e) = \min(e.u._{h\text{-}index}, e.v._{h\text{-}index})$. This means that from each endpoint (e.g., u), there are at least $\theta(e)$ Δ-incident adjacent edges e' such that $\theta(e') \geq \theta(e)$.*

Problem 2.1. ***Edge Core Decomposition.*** *Given a temporal graph $G = (V, E, \tau)$ and Δ, the edge core decomposition problem is to calculate the edge core number of each edge $e \in E$.*

In real-world scenarios, temporal graphs evolve as new edges arrive or when counterfactual analysis requires evaluating the absence of a specific edge. This motivates dynamic algorithms for maintaining temporal graph decompositions, which aim to update the core structure incrementally without full recomputation. We formally define the problem as follows.

Problem 2.2. ***Edge Core Maintenance.*** *Given a temporal graph $G = (V, E, \tau)$, a temporal proximity constraint Δ, and an initial edge core decomposition, efficiently update the core numbers of affected temporal edges in response to edge insertions or deletions.*

2.2 Edge-based Decomposition Framework

Edge-based decomposition is a commonly used approach for computing temporal edge cores based on a temporal degree function (Definition 2.1). A representative algorithm is shown in Algorithm 1, following the design in [47]. In Line 1, the initial core value $\theta(e)$ of each edge is set to its Δ-degree $d_\Delta(e)$. Lines 2-4 iteratively remove the edge with the minimum current degree and update the degrees of its Δ-incident adjacent edges. Specifically, the core number $\theta(e)$ is assigned at the time the edge is removed (Line 2), and its removal affects the degrees of Δ-incident adjacent edges (Line 4). The algorithm computes core numbers for all edges with a time complexity of $O(|E| \cdot \max\{\log d_{max}, d_{\Delta max}\})$, where d_{max} is the maximum degree in G and $d_{\Delta max}$ is the maximum number of Δ-incident edges at any edge $e \in E$. While efficient for static graphs, repeated recomputation in response to edge insertions or deletions can still incur significant overhead in dynamic settings.

Example. We use the illustrative temporal graph G in Figure 1 to illustrate the challenges of temporal edge-core maintenance under dynamic updates. The numbers on edges represent time stamps, and the temporal constraint is set as $\Delta = 2$. Edges $(v_1, v_3, 2)$ and $(v_1, v_3, 3)$ are existing temporal edges, and edge $(v_1, v_3, 8)$ is a new edge to be inserted at time 8. We first compute edge degrees according to Definition 2.1. For example, edge $(v_4, v_5, 3)$ has three Δ-incident adjacent edges from v_4—$(v_1, v_4, 1)$, $(v_3, v_4, 2)$, and itself—and four from v_5—$(v_4, v_5, 3)$, $(v_5, v_6, 2)$, $(v_5, v_7, 2)$, and $(v_5, v_7, 3)$—giving $d_\Delta((v_4, v_5, 3)) = 3$. In contrast, for edge $(v_1, v_2, 6)$, only one Δ-incident adjacent edge exists from v_1 (i.e., itself), since $(v_1, v_3, 2)$ and $(v_1, v_3, 3)$ are temporally too distant. Although two Δ-incident adjacent edges exist from v_2, the edge degree remains 1.

Algorithm 1: (k, Δ)-core Decomposition Algorithm

```
    Input: Temporal graph $G = (V, E, \tau)$
    Output: Edge core $\theta(e)$ $\forall e \in E$
1   Initialize $\theta(e) = d_\Delta(e)$ by Definition 2.1
2   while $E \neq \emptyset$ do
3       Peel $e \in E$ with the minimum degree $\theta(e)$
4       for $e' \in N_\Delta(e)$ do
5           Update $\theta(e') \leftarrow max(\theta(e), \theta(e') - 1)$
6   return $\theta(e)$ $\forall e \in E$
```

The decomposition algorithm [47] assigns an edge core number $\theta(e)$ to each edge. For instance, edges within the gray dashed box have $\theta = 3$, while $\theta(v_1, v_2, 6) = \theta(v_2, v_3, 7) = 1$. After deleting edge $(v_4, v_5, 3)$, only edges in the blue dashed box retain the (3,2)-core structure, and the core values of $(v_1, v_4, 1)$, $(v_1, v_3, 2)$, $(v_1, v_3, 3)$, and $(v_3, v_4, 2)$ drop from 3 to 2. When edge $(v_1, v_3, 8)$ is inserted, edges within the green dashed box newly form a (2,2)-core.

This example highlights several difficulties: temporal edge degrees are sensitive to time and neighborhood structure, edge core numbers differ even among nearby edges (e.g., $\theta(v_1, v_2, 6) = 1$ vs. $\theta(v_1, v_4, 1) = 3$), and updates can cause both localized and cascading changes. As a result, it becomes non-trivial to identify which edges may have their core numbers affected and to incrementally maintain these values as the graph evolves.

3 THEORETICAL FOUNDATIONS

In this section, we present our theoretical findings on the incremental maintenance of edge cores in evolving temporal graphs. We establish our theoretical foundation by recalling the edge-core hierarchy induced by peeling, which defines the edge core number θ and the nested subgraphs $H_{(k,\Delta)}$. To eliminate computational redundancy, we employ peeling solely as a definitional construct rather than an operational procedure.

The fundamental challenge in temporal edge-core maintenance lies in characterizing the set of edges whose core values are affected by graph updates. Our theoretical framework provides a complete characterization through four principal results: **Theorem 3.1 (Bounded Impact):** Each edge update induces a core value change of at most one, establishing a tight bound on the propagation scope. **Theorem 3.2 (Deterministic Root Selection):** Under our task decomposition strategy, only the root edge (with minimal core value) may change, while non-root edges maintain stability as support elements. **Definition 3.2 + Theorem 3.3 (Candidate Characterization):** The candidate edge set is precisely the Δ-reachable subgraph, enabling efficient local enumeration. **Theorem 3.4 (Correctness):** Sequential sub-edge processing is equivalent to batch processing, ensuring algorithmic correctness.

Given the structural complexity of temporal graphs, the insertion or deletion of an updating edge can simultaneously impact the edge degrees of multiple Δ-incident adjacent edges, creating interference

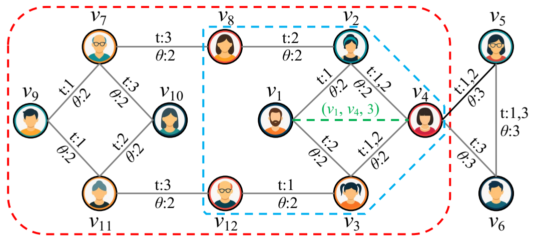

Figure 2: Candidate edges of different algorithms.

when constructing the candidate subgraph. We resolve this by task decomposition (Strategy 3.1): we transform the global update on e into a sequence of sub-edge updates (e, e') over $e' \in N_\Delta(e) \setminus \{e\}$, processing one pair at a time while buffering the others. Operationally (Algorithm 2), we keep e out of G until all sub-edges are processed and compute supports against the buffered sub-edges, which prevents cross-sub-edge contamination of support values.

Strategy 3.1. ***Task decomposition.*** *Given a temporal graph $G = (V, E, \tau)$ with temporal proximity constraint Δ and a target edge $e = (u, v, t)$ undergoing insertion/deletion: First generate the sub-edge subset $S = N_\Delta(e) \setminus \{e\}$ containing all Δ-incident adjacent edges excluding e itself. Each edge $e' \in S$ is called a* ***sub-edge*** *because it participates in the localized update process as a constituent element of the task decomposition strategy. Then iterate through each sub-edge $e' \in S$: when performing edge insertion, establish Δ-incident adjacency relations between e and e'; conversely, during edge deletion, remove the existing Δ-incident adjacency relations between e and e'.*

Example. We use Figure 2 to illustrate this strategy. In the temporal graph G, each edge is annotated with its time stamp and edge core, and the temporal constraint is set to $\Delta = 2$. A new edge $(v_1, v_4, 3)$ is to be inserted into G. Based on Definition 2.1 and Strategy 3.1, the sub-edge set $S = \{(v_1, v_2, 1), (v_1, v_3, 2), (v_2, v_4, 1), (v_2, v_4, 2), (v_3, v_4, 1), (v_3, v_4, 2), (v_4, v_5, 1), (v_4, v_5, 2), (v_4, v_6, 3)\}$. Our algorithms process sub-edges sequentially, where each computation builds upon previously processed results.

Lemma 3.1. *Given a new edge $e = (u, v, t)$ and a sub-edge set S, each sub-edge $e' \in S$ necessarily shares at least one endpoint with e (i.e., $e' = (u, w, t')$, $e' = (v, w, t')$, or $e' = (u, v, t')$). Upon insertion or deletion of a sub-edge, only the number of Δ-incident adjacent edges connected to the common endpoint(s) increases or decreases by exactly 1. Consequently, only the edge degrees of e and e' may change, with a maximum magnitude of 1.*

Proof. Since $S = N_\Delta(e) \setminus e$ constitutes a subset of the Δ-incident adjacent edge set $N_\Delta(e)$, by Definition 2.1, each $e' \in S$ must share at least one common endpoint with e. The update of an individual sub-edge exclusively affects the count of Δ-incident adjacent edges at common endpoints, incrementing or decrementing it by exactly 1. Consequently, by Equation (1), the edge degrees of only e and e' may undergo change, with a maximum magnitude of 1. □

According to Lemma 3.1, a sub-edge may lead to changes to the edge degree of that sub-edge $d_\Delta(e')$ and the originally updated edge $d_\Delta(e)$. The edge cores of e' and e may not be equal, which poses a challenge in identifying candidate edges where cores might change.

Definition 3.1. **Δ*-support.*** *For an edge $e = (u, v, t)$, we define its Δ-support as follows:*

$$Sup(u) = |\{(u, w, t') \in E \mid |t - t'| \le \Delta \wedge \theta((u, w, t')) \ge \theta(e)\}| \quad (2)$$

$$Sup(v) = |\{(v, w, t') \in E \mid |t - t'| \le \Delta \wedge \theta((v, w, t')) \ge \theta(e)\}| \quad (3)$$

$$\Delta\text{-}support(e) = \min(Sup(u), Sup(v)) \quad (4)$$

where $Sup(u)$ and $Sup(v)$ represent the support calculated from vertices u and v, respectively. We say that a Δ-incident adjacent edge e' supports e if the edge core of e' is at least $\theta(e)$.

In temporal graphs, edge cores quantify local structural cohesion. Single-edge modifications exhibit bounded propagation effects, constraining structural changes to local neighborhoods rather than affecting distant graph regions. This locality property is fundamental to our incremental maintenance approach, as it limits the computational scope of updates.

Theorem 3.1. *Given a temporal graph $G = (V, E, \tau)$ and a new edge $e = (u, v, t)$, $S = N_\Delta(e) \setminus e$ is the sub-edge set. Each sub-edge e' will change the edge cores of edges in G by at most 1.*

Proof. For sub-edge insertion, suppose that the edge core of an edge $\varepsilon \in H^{\varepsilon}_{m,\Delta}$ increases from $\theta(\varepsilon) = m$ to $\theta(\varepsilon)' = m + n$ where $n > 1$. At least one of e or e' must belong to $H^{\varepsilon}_{m+n,\Delta}$; otherwise, $H^{\varepsilon}_{m+n,\Delta}$ would form a $(m + n, \Delta)$-core prior to inserting e', yielding $\theta(\varepsilon) = \theta(\varepsilon)' = m + n$—a contradiction. By Lemma 3.1, the insertion of sub-edge e' increases the edge degree of either e or e' by at most 1. Consequently, the edge degrees in $H^{\varepsilon}_{m+n,\Delta} \setminus \{e, e'\}$ would be at most $m + n - 1$, implying $\theta(\varepsilon) \le m + n - 1 > m$, which contradicts our initial assumption.

For sub-edge deletion, assume the edge core of ε decreases by $n > 1$. Upon reinserting the sub-edge, ε would increase its edge core by at most 1, rendering full recovery impossible—a contradiction. □

This theorem establishes a crucial property of our task decomposition strategy: when processing a sub-edge pair (e, e'), only the edge with the smaller core value (the "root") will change. This deterministic behavior simplifies algorithm design, as we only need to focus on the root edge during each sub-edge update, while the non-root edge remains stable.

Theorem 3.2. *Given a new edge $e = (u, v, t)$ and a sub-edge e', only the one with the smaller edge core may have its edge core changed by 1, which is denoted as* ***root*** *edge. The larger one is called the non-root edge. The non-root edge must be in $H^{root}_{(\theta(root)', \Delta)}$, where $\theta(root)'$ is the updated edge core of the root edge.*

$$root = \begin{cases} e & if\, \theta(e) < \theta(e') \\ e, e' & if\, \theta(e) = \theta(e') \\ e' & if\, \theta(e) > \theta(e') \end{cases}$$

The edge core of root(s) is denoted as $\theta(root)$.

Proof. According to Strategy 3.1 and Definition 3.1, e and e' are Δ-incident, and only the one having a larger edge core could be a support of another. We prove the scenario $\theta(e) > \theta(e')$ where e' is the root edge. Based on Theorem 3.1, the edge core of e and e' will be increased by at most 1 after inserting e'. If e increases its edge core to $\theta(e)' = \theta(e) + 1$, as $\theta(e') \le \theta(e')' \le \theta(e') + 1 < \theta(e)'$, e' cannot be in $H^{e}_{(\theta(e)', \Delta)}$, leading to a contradiction. If e' increases its edge core to $\theta(e')' = \theta(e') + 1$, as e and e' are Δ-incident and $\theta(e) \ge \theta(e')' > \theta(e')$, e must be a support of e'. Therefore, only the root edge(s) may have the edge core changed by at most 1, and the non-root edge must be in $H^{root}_{(\theta(root)', \Delta)}$. □

Lemma 3.2. *An edge e can change its edge core if and only if either its edge degree $d_\Delta(e)$ changes or its Δ-$support(e)$ value changes. No other conditions can affect the edge core of e.*

Proof. According to Definition 2.4, $\theta(e)$ is determined by the H-index of its endpoints, which depends on the multiset of edge cores of Δ-incident adjacent edges. For sufficiency, if $d_\Delta(e)$ changes, the number of Δ-incident adjacent edges changes, altering the H-index. If Δ-support(e) changes, the number of supporting edges changes, also altering the H-index. For necessity, for $\theta(e)$ to change, the H-index must change, requiring either: (1) the number of Δ-incident adjacent edges to change (affecting $d_\Delta(e)$), or (2) the core values of existing Δ-incident adjacent edges to change (affecting Δ-support(e)). No other factors influence the H-index. □

Lemma 3.3. *Given a temporal graph $G = (V, E, \tau)$, the temporal proximity constraint Δ, a new edge $e = (u, v, t)$ will be inserted into or deleted from G. e' is a sub-edge generated from e. Only edge $\varepsilon \in E$ whose edge core is equal to $\theta(root)$ may have edge core changed.*

Proof. According to Theorem 3.2, only the root edge(s) can experience a change in edge core. The non-root edge necessarily preserves its edge core and must belong to $H^{root}_{(\theta(root)',\Delta)}$, thereby exerting no influence on the support values or edge cores of other edges. By Lemma 3.2, only the root edge undergoes a change in $d_\Delta(e)$. For any non-root edge to alter its edge core, the edge cores of its Δ-incident adjacent edges must undergo modification. Through recursive application, any edge core modification necessitates a preceding change in the root edge's core.

For sub-edge insertion, consider the relationship between $\theta(\varepsilon)$ and $\theta(root)$: (1) If $\theta(\varepsilon) > \theta(root)$, the root edge cannot participate in $H^{\varepsilon}_{(\theta(\varepsilon)+1,\Delta)}$ since $\theta(root) + 1 \leq \theta(\varepsilon) < \theta(\varepsilon) + 1$—a contradiction. (2) If $\theta(\varepsilon) < \theta(root)$, removing sub-edge e' would reduce the root's edge core to at most $\theta(root) \geq \theta(\varepsilon)'$, which remains sufficient to support edges with core $\theta(\varepsilon)'$—a contradiction. For sub-edge deletion, analogous reasoning applies. Therefore, only edges with edge cores equal to $\theta(root)$ may undergo changes. □

Building on the lemmas above, we can find candidate edges whose cores may be changed after inserting or deleting a sub-edge.

Definition 3.2. ***Δ-reachable path.*** *A Δ-reachable path p consisting of a sequence of $l \in \mathbb{N}$ temporal edges $p = \{e_1 = (v_1, v_2, t_1), e_2 = (v_2, v_3, t_2), ..., e_l = (v_l, v_{l+1}, t_l)\}$ for which $|t_{i+1} - t_i| \leq \Delta$, i.e., any pair of adjacent edges in the path are Δ-incident.*

Theorem 3.3. *Given a new edge $e = (u, v, t)$ and a sub-edge e', the root edge(s) is defined by Theorem 3.2. After inserting or deleting e', an edge ε may have edge core changed if and only if $\theta(\varepsilon) = \theta(root)$ and there is a Δ-reachable path that consists of edges with edge cores equal to $\theta(root)$ leading to the root edge(s).*

Proof. According to Lemma 3.2, the edge core of an edge changes, either the number of its Δ-incident adjacent edges has changed or at least one of its existing Δ-incident adjacent edges must have its edge core value altered. Applying this recursively, we must reach an edge whose edge core is changed due to gaining or losing a Δ-incident adjacent edge. Furthermore, only the edges with an edge core equal to $\theta(root)$ may have their edge cores changed (Lemma 3.3). Therefore, the impact of a sub-edge update on the edge cores of other edges can only propagate through Δ-reachable paths from the root in which all edge cores are equal to $\theta(root)$. □

Theorem 3.4. *Given a temporal graph $G = (V, E, \tau)$ with temporal proximity constraint Δ, and a new edge $e = (u, v, t)$ to be inserted into or deleted from G, let $S = N_\Delta(e) \setminus \{e\}$ denote the set of sub-edges. The edge cores of all edges in E can be accurately maintained through sequential processing of each sub-edge in S.*

Proof. We prove the case for insertion and deletion follows by symmetry. Let $S = N_\Delta(e) \setminus \{e\} = \{e'_1, ..., e'_m\}$ be the set of sub-edges. For $i \geq 0$, let $G^{(i)}$ denote the graph state after integrating the Δ-incidence between e and the first i sub-edges $\{e'_1, ..., e'_i\}$, followed by complete edge-core maintenance. Let $\theta^{(i)}$ denote the edge-core assignment in $G^{(i)}$. For each sub-edge e'_i (where $i \geq 1$), let r_i be the root edge and $k_i = \theta^{(i-1)}(r_i)$ be its core value. By Definition 3.2 and Theorem 3.3, the candidate edges whose cores may change from $\theta^{(i-1)}$ to $\theta^{(i)}$ are precisely those in the subgraph of $G^{(i-1)}$ consisting of edges with core value k_i that are Δ-reachable to r_i.

By Theorem 3.1, any affected edge changes its core by at most one. Since each update is monotone (cores only increase during insertion), the maintenance process on the candidate subgraph converges to a unique final assignment that is independent of the processing order within the subgraph.

Therefore, processing sub-edge e'_i sequentially yields the same core assignment $\theta^{(i)}$ as would be obtained by batch-integrating the first i sub-edges simultaneously. By induction on $i = 1, ..., m$, we obtain $\theta^{(m)} = \theta^*$, which equals the batch-maintained assignment after integrating all sub-edges in S. □

4 TECM FRAMEWORK

Building on the theoretical foundations in Section 3, we propose the **TECM** framework for incremental edge-core maintenance in temporal graphs. The framework systematically translates theoretical results into algorithmic components: (1) **Candidate identification** implements the Δ-reachable subgraph characterization (Definition 3.2, Theorem 3.3) to locate potentially affected edges; (2) **Root selection** applies the deterministic root property (Theorem 3.2) to identify the single edge requiring updates; (3) **Update bounds** enforce unit-step constraints (Theorem 3.1) ensuring at most one core level change per update; and (4) **Efficiency optimization** utilizes local counter mechanisms (Lemma 3.1) to track degree changes without full recomputation.

The **TECM** framework addresses both edge insertions and deletions through a unified architecture. Edge insertions and deletions exhibit fundamentally different propagation patterns, necessitating specialized approaches: insertion algorithms employ candidate-based enumeration with pruning strategies, while deletion uses direct propagation. The framework routes operations to the appropriate kernel through a unified mechanism. For edge insertions, **TECM** first identifies the potentially affected candidate edges (following the Δ-reachable subgraph characterization) and then applies a localized peel operation to update their edge cores. In contrast, edge deletion is handled more directly. Starting from the root edge (applying the deterministic root property), the algorithm computes the H-index to detect any decrease in edge core value and propagates the update if necessary (constrained by unit-step bounds).

4.1 Base Incremental Algorithm

Based on the theoretical foundation established in Section 3, we propose a Base Incremental Algorithm (**TECM**-Ini). To minimize computational redundancy, we exclusively compute the sub-edges generated from one endpoint vertex, as elaborated in Lemma 3.1.

Lemma 4.1. *Given a new edge $e = (u, v, t)$, the sub-edges can be generated from vertices u and v, denoted as S_u and S_v, respectively.*

Algorithm 2: Base Incremental Algorithm

```
Input: Temporal graph $G = (V, E, \tau)$, temporal proximity
    constraint $\Delta$, new edge $e = (u, v, t)$
Output: Edge core $\theta(e)$ $\forall e \in E$
1  Generate sub-edge set $S_u$ and $S_v$
2  insertListUset $\leftarrow S_u$, insertListVset $\leftarrow \varnothing$, $\theta(e) \leftarrow 1$
3  for $e' \in S_v$ do
4      Initialize all edge flags to false
5      insertListVset.insert($e'$)
6      root $\leftarrow \theta(e) \le \theta(e')$ ? $e : e'$
7      candidateVec $\leftarrow \{root\}$, candidateQueue$\leftarrow \{root\}$
8      while candidateQueue is not empty do
9          can_e $\leftarrow$ candidateQueue.pop()
10         for $\Delta{-}e \in N_\Delta(can\_e)$ do
11             if $\theta(\Delta{-}e) = \theta(root) \wedge !\Delta{-}e.flag$ then
12                 Push $\Delta{-}e$ into candidateQueue and
                     candidateVec
13                 $\Delta{-}e.flag \leftarrow true$
14     for $can\_e \in candidateVec$ do
15         Calculate $\Delta{-}support(can\_e)$ by Equations
             Equation (2), Equation (3), Equation (4)
16     for $can\_e \in candidateVec$ do
17         if $can\_e.flag \wedge \Delta{-}support(can\_e) \le \theta(root)$ then
18             $can\_e.flag \leftarrow false$
19             for $\Delta{-}e \in N_\Delta(can\_e)$ do
20                 if $\Delta{-}e.flag$ then
21                     Update the support value of $\Delta{-}e$
22     for $can\_e \in candidateVec$ do
23         if $can\_e.flag$ then
24             $\theta(can\_e) \leftarrow \theta(can\_e) + 1$
25 Insert $e$ into $G$
26 return $\theta(e)$ $\forall e \in E$
```

When processing sub-edges from only one set (either S_u or S_v), regardless of their quantity, the edge core of e will increase to exactly 1.

Proof. Consider the scenario where we process sub-edges in S_u first. According to Strategy 3.1 and Theorem 3.4, sub-edges in S_v remain uninserted at this stage. Consequently, vertex v has precisely one Δ-incident adjacent edge, namely e itself. By Definition 2.1, this implies $d_\Delta(e) = 1$, and hence $\theta(e) = 1$. With this minimal edge core value, e cannot support edge core elevation for other edges. □

Lemma 4.2. *If the support value of a candidate ε is not larger than $\theta(\varepsilon)$, $\theta(\varepsilon)$ cannot be increased.*

Proof. According to Definition 2.2, Definition 2.3, Definition 2.4, and Definition 3.1, if an edge ε can participate in a $(\theta(\varepsilon)+1, \Delta)$-core, the support value of ε must be not less than $\theta(\varepsilon) + 1 > \theta(\varepsilon)$. □

The pseudocode detailing the **TECM**-Ini algorithm is presented in Algorithm 2. Given a temporal graph $G = (V, E, \tau)$, a temporal proximity constraint Δ, and a new edge $e = (u, v, t)$ designated for insertion into G, the algorithm proceeds as follows. Initially, Line 2 generates sub-edge sets S_u and S_v corresponding to the endpoints u and v of e, respectively, based on Strategy 3.1.

Crucially, the edge e is not integrated into G until the processing of all its sub-edges is complete. Two lists, *insertListUset* and *insertListVset*, are initialized to buffer the processed sub-edges. Following Lemma 4.1, we designate one set, arbitrarily chosen as S_u, as the treatment-free set, buffer its contents into *insertListUset*, and initialize $\theta(e) = 1$ (Line 1). Subsequently, the algorithm sequentially processes each sub-edge e' within S_v.

The core processing of a sub-edge encompasses four distinct phases: **Phase 1** (Lines 7-13) iteratively identifies all candidate edges using a Breadth-First Search (BFS) approach, guided by Theorem 3.3. **Phase 2** (Lines 13-15) computes the support value for every identified candidate, consulting the buffered sub-edges for Δ-incident neighbors. **Phase 3** (Lines 15-21) implements the peeling process, removing candidates whose support value does not exceed $\theta(root)$. **Phase 4** (Lines 21-24) increments the edge core by 1 for all candidates that survived the peeling phase. After processing all sub-edges in S_v, the new edge e is inserted into G (Line 24), and the edge cores across the graph are updated accordingly.

The algorithm operates in three phases: (1) candidate enumeration via BFS over Δ-incident neighbors, (2) support computation for all candidates, and (3) peeling with support updates. Each Δ-incident adjacency is accessed at most twice (once in enumeration, once in peeling), yielding time complexity $O(m_C)$, where $m_C = \sum_{e \in C} d_\Delta(e)$ denotes the total Δ-incident adjacencies within the candidate set C. Space complexity is $O(m_C)$ for storing candidates and support counters.

4.2 Optimized Candidate Pruning

Although **TECM**-Ini can maintain edge cores incrementally, the process may often involve a significant amount of redundant calculations. When the edge core distribution of the graph is quite concentrated, inserting a sub-edge may result in finding many candidates whose edge cores are unlikely to change, leading to substantial redundant calculations and data access overhead. In this section, we explore optimized algorithms to maintain edge cores.

Lemma 4.3. *If the support value of a candidate edge ε is less than or equal to its edge core, i.e., $\Delta{-}support(\varepsilon) \le \theta(\varepsilon)$, then ε cannot cause any of its Δ-incident adjacent edges to become candidates.*

Proof. According to Lemma 4.2, if $\Delta{-}support(\varepsilon) \le \theta(\varepsilon)$, the edge core $\theta(\varepsilon)$ will not increase. Consequently, ε cannot support increasing the core of any Δ-incident neighbor $\Delta{-}e$ where $\theta(\Delta{-}e) = \theta(\varepsilon)$. Since an edge only becomes a candidate if its core might increase, ε cannot propagate the candidate status to its neighbors under this condition. This justifies pruning the candidate search path originating from ε without compromising correctness. □

The overall framework of the Optimized Candidate Pruning Algorithm (**TECM**-Ref) is similar to that of Algorithm 2 (**TECM**-Ini), with efficiency improvements through targeted optimizations. Based on Lemma 4.1, sub-edges with higher frequencies in S_u and S_v are prioritized to reduce candidate traversal. Phases 1 and 2 are replaced by Algorithm 3, which employs a key-value map `candidateMap` to record, for each candidate *can_e*, its Δ-incident adjacent edge Δ-*e* and their common endpoint. When traversing Δ-*e* for *can_e*, if $\theta(\Delta\text{-}e) \ge \theta(root)$, the support of the corresponding endpoint of *can_e* is updated, and $(\Delta\text{-}e, v')$ is stored in `candidateMap[`*can_e*`]`. Potential candidates that meet the core value criteria are temporarily buffered in `tempCandidateVec`, and the support value of *can_e* is computed. According to Lemma 4.3,

Algorithm 3: Optimized Candidate Subgraph Construction

```
Input: root
Output: candidateVec, candidateMap
1  Initialize candidateMap:$edge \longrightarrow (edge, vertex)$
2  candidateVec $\longleftarrow \{root\}$, candidateQueue$\longleftarrow \{root\}$
3  while $candidateQueue$ is not empty do
4      Initialize tempCandidateVec $\longleftarrow \varnothing$
5      can_e $\longleftarrow$ candidateQueue.pop()
6      for $\Delta{-}e \in N_\Delta(can\_e)$ do
7          if $\theta(\Delta{-}e) \geq \theta(root)$ then
8              Accumulate $Sup(can\_e.u)$ and $Sup(can\_e.v)$
9              // $v'$ selection: the common endpoint between
                 can_e and $\Delta{-}e$
10             Add $(\Delta{-}e, v')$ to candidateMap[can_e]
11         if $\theta(\Delta{-}e) = \theta(root) \wedge !\Delta{-}e.flag$ then
12             Push $\Delta{-}e$ into tempCandidateVec
13     $\Delta{-}support(can\_e)$ = min($Sup(can\_e.u)$, $Sup(can\_e.v)$)
14     if $\Delta{-}support(can\_e) > \theta(root)$ then
15         for $tmp\_can\_e \in tempCandidateVec$ do
16             Push $tmp\_can\_e$ into candidateQueue and
                 candidateVec
17             $tmp\_can\_e.flag \longleftarrow true$
18 return candidateVec, candidateMap
```

only candidates with support greater than $\theta(root)$ can expand further candidates. In the peeling phase, support values of related candidates are directly updated via `candidateMap`. Due to space constraints, we omit the full pseudocode.

The algorithm fuses candidate discovery with support computation using a key-value map, eliminating redundant traversals through Lemma 4.3. The theoretical complexities remain $O(m_C)$ time and $O(m_C)$ space, but the effective candidate set size is substantially reduced in practice due to early pruning.

4.3 Batch Insertion Algorithm

In this section, we present the batch-oriented sub-edge processing algorithm (**TECM**-Batch-Ins). The batch insertion algorithm leverages Lemma 4.5 to provide efficient pruning during candidate enumeration. This lemma establishes an a priori upper bound $\theta(e)_{max}$ on the possible edge core of the new edge e, which is directly utilized in Algorithm 5, Lines 4-5, to prune candidate enumeration. Specifically, for any edge whose current edge core exceeds $\theta(e)_{max}$, we skip candidate set construction since its core cannot change, making the batch design efficient and parallel-friendly by eliminating unnecessary candidate accessing operations.

Lemma 4.4. *Given a temporal graph $G = (V, E, \tau)$ with temporal proximity constraint Δ, when a new edge $e = (u, v, t)$ is inserted into G, all edges except e will increase their edge cores by at most 1.*

Proof. By Theorem 3.1, each edge's core can increase by at most 1 during any sub-edge insertion. For sub-edges with identical root core $\theta(root)$, only edges with current core $\theta(root)$ are candidates. Once a candidate's core increases to $\theta(root) + 1$, it cannot serve as a candidate for subsequent sub-edges with the same root core, preventing cumulative increases. For sub-edges with different root core values, candidate sets are disjoint: when the difference exceeds 1, edges affected by one sub-edge cannot be candidates for others due to the unit increase limit; when the difference equals 1, processing sub-edges with larger root cores first ensures updated edges cannot become candidates for sub-edges with smaller root cores. Therefore, all edges increase their cores by at most 1. □

Algorithm 4: Maximum Core Value Calculation

```
Input: $G = (V, E, \tau)$, $e = (u, v, t)$
Output: The maximum of $\theta(e)$
1  Initialize uNeiCoreVec, vNeiCoreVec$\longleftarrow \varnothing$
2  for $(u, w, t') \in E \mid |t - t'| \leq \Delta$ do
3      Push $\theta((u, w, t')) + 1$ into uNeiCoreVec
4  sort uNeiCoreVec in non-decreasing order
5  Push uNeiCoreVec.back() into uNeiCoreVec
6  $\theta(e)_{umax} \longleftarrow H{-}index(uNeiCoreVec)$
7  // $\theta(e)_{vmax}$ can be calculated similarly
8  $\theta(e)_{vmax} \longleftarrow H{-}index(vNeiCoreVec)$
9  $\theta(e)_{max} \longleftarrow min(\theta(e)_{umax}, \theta(e)_{vmax})$
10 return $\theta(e)_{max}$
```

Lemma 4.5. *Given a temporal graph $G = (V, E, \tau)$, a temporal proximity constraint Δ, and a new edge $e = (u, v, t)$ to be inserted into G, let $\theta(e)_{max}$ be the maximum possible edge core value for e, as computed by Algorithm 4. The edge cores of existing edges $\varepsilon \in E$ with $\theta(\varepsilon) \geq \theta(e)_{max}$ will remain unchanged after the insertion of e.*

Proof. The edge core of an edge is defined by the minimum H-index of its endpoints (Definition 2.4). Lemma 4.4 establishes that the insertion of e increases the edge core of any other edge by at most 1. Algorithm 4 calculates $\theta(e)_{max}$ by hypothetically incrementing the core of each Δ-incident neighbor of u and v by 1 and including a virtual maximum core representing e itself before its insertion, then computing the H-index. Since the maximum potential core value of the new edge e is $\theta(e)_{max}$, e cannot participate in any (k, Δ)-core where $k > \theta(e)_{max}$. Consequently, e cannot provide the necessary support to elevate the edge core of any other edge beyond $\theta(e)_{max}$. Therefore, edges $\varepsilon \in E$ with $\theta(\varepsilon) \geq \theta(e)_{max}$ are unaffected by the insertion of e. □

The algorithm examines all Δ-incident neighbors of both endpoints u and v to determine the upper bound. The time complexity is $O(d_\Delta \cdot \log d_\Delta)$ where $d_\Delta = \max(d_\Delta(u), d_\Delta(v))$, due to the sorting operations on the neighbor core vectors. Space complexity is $O(d_\Delta)$ for storing the neighbor core vectors.

Leveraging the preceding lemmas, we present the batch sub-edge processing algorithm in Algorithm 5. Line 5 computes $\theta(e)_{\max}$ using Algorithm 4 and assigns it as the edge core of the new edge e. Lines 1-5 collect the edge cores of $N_\Delta(e) \setminus \{e\}$. Line 8 groups all sub-edges into $candidateVecSet$ based on their edge cores. For each distinct edge core, Line 8 identifies the corresponding candidate set in parallel. Lines 11-12 apply the peel operation to each candidate set in non-decreasing order of their $root\_core$. If the inserted edge e is peeled while processing a candidate set with $root\_core$, then all candidates with edge cores greater than or equal to $root\_core$ remain unchanged. Once all candidate sets are processed, edge cores are correctly maintained. Finally, after inserting e into G (Line 13), its actual edge core is recalculated using Definition 2.4 (Line 14).

Algorithm 5: Batch Insertion Algorithm

```
Input: Temporal graph $G = (V, E, \tau)$, temporal proximity
       constraint $\Delta$, new edge $e = (u, v, t)$
Output: Edge core $\theta(e)\ \forall e \in E$
1  Set $\theta(e) \leftarrow \theta(e)_{max}$ by Algorithm 4
2  Initialize: candidateVecSet, candidateMapSet
3  for $\Delta$-$e \in N_\Delta(e)$ do
4      if $\theta(\Delta\text{-}e) < \theta(e)_{max}$ then
5          rootCoreSet.insert($\theta(\Delta\text{-}e)$) in non-decreasing order
6          Push $\Delta$-$e$ into candidateVecSet[$\theta(\Delta\text{-}e)$]
7  #pragma omp parallel
8  for $root\_core \in rootCoreSet$ do
9      Invoke Algorithm 3 to find candidate sub-graph
10 *** Phase 2: Peel Candidates ***
11 for $root\_core \in rootCoreSet$ do
12     Peel candidateVecSet(root_core) with
         candidateMapSet[root_core]
13     Update Edge Cores
14 Insert $e$ into $G$
15 Update $\theta(e)$ according to Definition 2.4
16 return $\theta(e)\ \forall e \in E$
```

Theorem 4.1. *Given a temporal graph $G = (V, E, \tau)$, a temporal proximity constraint Δ, and a new edge $e = (u, v, t)$ to be inserted into G, Algorithm 5 maintains the edge cores for all edges in $E \cup \{e\}$.*

Proof. Lemma Lemma 4.5 determines the maximum possible edge core for the new edge e, denoted $\theta(e)_{max}$. The corresponding lower bound for the final core value of e is $\theta(e)_{min} = \theta(e)_{max} - 1$.

Consider candidate edges in $N_\Delta(e)$ with edge cores less than $\theta(e)_{min}$. The edge e inherently provides support for these candidates, irrespective of whether its final core value settles at $\theta(e)_{max}$ or $\theta(e)_{min}$. Thus, the algorithm correctly maintains the edge cores for these candidates.

Now, consider candidates in $N_\Delta(e)$ with edge cores equal to $\theta(e)_{min}$. The algorithm processes candidate sets in non-decreasing order of their root core value (*root_core*). When processing the set where $root\_core = \theta(e)_{min}$: (1) If edge e is peeled, it cannot attain a core of $\theta(e)_{min}+1$, and neighboring candidates' edge cores correctly remain unchanged. (2) If edge e is not peeled, it possesses sufficient support to potentially participate in a $(\theta(e)_{min}+1, \Delta)$-core, correctly permitting e to contribute support. Therefore, the edge cores for candidates at the $\theta(e)_{min}$ level are correctly maintained in both cases. Finally, Algorithm 5 determines the final edge core of the newly inserted edge e based on the updated cores of its Δ-incident neighbors (Line 14), consistent with Definition 2.4. □

The algorithm processes candidates in parallel across different core levels. For each level, the complexity follows **TECM**-Ref, resulting in the same $O(m_C)$ time and $O(m_C)$ space bounds, with additional parallelization benefits. The upper bound calculation (Algorithm 4) requires $O(d_\Delta \cdot \log d_\Delta)$ time, where $d_\Delta = \max(d_\Delta(u), d_\Delta(v))$ is the maximum Δ-degree of the new edge's endpoints.

4.4 Batch Deletion Algorithm

In this section, we study the edge-core maintenance algorithm for edge deletion (**TECM**-Batch-Del). Unlike insertions, deletions do not affect the core of the removed edge itself. The algorithm starts by checking whether the edge cores of its Δ-incident adjacent edges (i.e., sub-edges) change. If so, the changes are propagated via BFS.

Algorithm 6: Batch Deletion Algorithm

```
Input: Temporal graph $G = (V, E, \tau)$, temporal proximity
       constraint $\Delta$, delete edge $e = (u, v, t)$
Output: Edge core $\theta(e)\ \forall e \in E$
1  Delete $e$ from $G$
2  Initialize candidateQueue $\leftarrow e$
3  Initialize all edge flags to false
4  Set $H\text{-}index(e) \leftarrow 0$
5  while $candidateQueue$ is not empty do
6      can_e = candidateQueue.pop()
7      if $!can\_e.flag \wedge H\text{-}index(can\_e) < \theta(can\_e)$ then
8          $\theta(can\_e) \leftarrow \theta(can\_e) - 1$
9          $can\_e.flag \leftarrow true$
10         for $\Delta$-$e \in N_\Delta(can\_e)$ do
11             if $\theta(\Delta\text{-}e) \le \theta(can\_e)$ then
12                 Push $\Delta$-$e$ into candidateQueue
13 return $\theta(e)\ \forall e \in E$
```

Lemma 4.6. *Given a temporal graph $G = (V, E, \tau)$, the temporal proximity constraint Δ, an edge $e = (u, v, t)$ will be deleted from G. All edges will reduce their edge cores by at most 1.*

Proof. Assume there is an edge ε whose edge core $\theta(\varepsilon) = k$, and that its edge core decreases by $m > 1$. If we insert ε back to G, according to Lemma 4.4, we have $\theta(\varepsilon) \le k - m + 1$. Because $m > 1$, $k - m + 1 < k$, which is a contradiction. □

Based on Lemma 4.6, we propose the incremental algorithm for edge deletion. The pseudocode is shown in Algorithm 6. The deleted e is selected as the initial *can_e* and pushed into *candidateQueue* (Lines 6-3). Then, we calculate the edge core of each $can\_e \in candidateQueue$ based on the H-index of the two endpoints of *can_e*. If there is a change, add its Δ-incident adjacent edges that have edge cores not larger than $\theta(can\_e)$ to the *candidateQueue* (Lines 5-11). Since the edge core of each edge can decrease by at most 1 (Lemma 4.6), once the edge core of an edge has changed, subsequent accesses can skip that edge (Line 7).

The algorithm employs BFS propagation from the deleted edge. In the worst case, all edges require core updates, resulting in time complexity $O(M_\Delta)$ and space complexity $O(M_\Delta)$, where $M_\Delta = \sum_{e \in E} d_\Delta(e)$ represents the total Δ-incident adjacencies in the graph. Each edge is processed at most once due to the flag mechanism, ensuring efficient propagation.

5 EVALUATIONS

5.1 Experimental Setup

Algorithms. We compare our approach with existing algorithms from two perspectives: performance and practical applicability:

- **Static edge core decomposition (SECD)** [47] computes edge cores via global traversal. We measure efficiency by the average time to process each updated edge, normalized by the baseline time for full decomposition, yielding the speedup ratio. This metric is widely used to assess incremental algorithms [51, 64].

Table 2: Graph datasets

Graph	Vertices	Edges	Domain
FacebookMsg	1,899	59,795	social network
Enron	86,806	1,133,968	email network
AskUbuntu	134,035	257,305	question answering
Twitter	346,573	2,131,270	retweets
StackOverflow	2,584,164	47,902,566	interaction network
Reddit	3,007,854	84,272,870	social network

- **Incremental batch edge core decomposition (BECD)** [16] updates edge cores by recomputing H-index values of candidate edges, following standard core decomposition principles [46, 52, 64]. BECD aligns with Phase 1 of Algorithm 2. Since insertions are more complex than deletions, requiring comprehensive candidate enumeration and support recalculation as they can potentially increase edge cores, while deletions only decrease edge cores and follow direct propagation patterns, we focus on insertion comparisons; deletion results are omitted for brevity.
- **History k-core** [69, 70] constructs a snapshot by merging edges with identical endpoints within a time window into a single unlabeled edge. We provide a case study demonstrating its limitations in temporal analysis, underscoring the value of our edge core definition and the necessity of efficient edge core maintenance.

For convenience, we denote the Base Incremental Algorithm (Section 4.1) as **TECM**-Ini, the refined version Optimized Candidate Pruning Algorithm (Section 4.2) as **TECM**-Ref, batch insertion algorithm (Section 4.3) as **TECM**-Batch-Ins, and batch deletion algorithm (Section 4.4) as **TECM**-Batch-Del. Only **TECM**-Batch-Del is designed for edge deletion, while other algorithms focus on insertions. Due to the fundamental complexity asymmetry between these operations, we present both results but emphasize that they should not be directly compared. Deletion operations, with their deterministic propagation patterns, are evaluated to demonstrate deletion-specific optimizations, while insertion results provide comprehensive analysis of our algorithmic innovations. Throughout our experiments, whenever we delete edges, we apply **TECM**-Batch-Del, and whenever we insert edges, we apply one of the insertion algorithms (**TECM**-Ini, **TECM**-Ref, or **TECM**-Batch-Ins).

Datasets. We collect 6 real-world temporal graph datasets from various domains. These datasets are widely used for evaluating temporal graph analysis algorithms, which are summarized in Table 2.

Parameter choosing. We configure two parameters: Δ-percentile and θ-percentile. The Δ-percentile, derived from node-level inter-event times (IETs) [47], determines Δ, where a higher percentile yields more Δ-neighbors per edge and thus higher edge cores. The θ-percentile controls the core value of update edges: given the maximum edge core $\theta_{\max}$, each new edge is assigned a core value no smaller than $\theta_{\max} \times \theta$-percentile.

5.2 Performance Evaluation

Exp-1: Static baseline evaluation. We first evaluate the time cost of static edge core decomposition (SECD) with $\Delta-percentile \in \{0.25, 0.5, 0.75\}$, which is also applied in [47]. The results are shown in Figure 3. It can be observed that as $\Delta-percentile$ increases, the time cost of the baseline algorithm also rises. This is because a larger $\Delta-percentile$ results in each edge having more Δ-incident

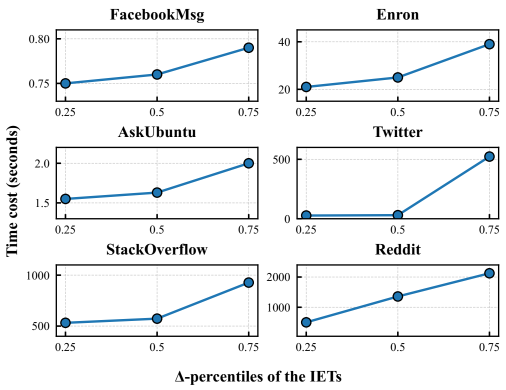

Figure 3: The time cost of SECD.

adjacent edges (Definition 2.1), which in turn increases the number of edges that need to be peeled, thereby leading to greater time consumption. For Twitter, the time cost at $\Delta-percentile = 0.75$ is 19x larger than that of $\Delta-percentile = 0.25$. Therefore, repeatedly calling the baseline algorithm to completely decompose the entire graph with each update would be highly time-consuming, especially for larger $\Delta-percentile$, making it impractical for meeting the real-time update and query needs in actual applications.

Exp-2: Scalability on different (Δ, θ)-percentiles. We evaluate algorithm speedup over baseline under different (Δ, θ)-percentile combinations. For each Δ-percentile $\in \{0.25, 0.5, 0.75\}$, we first compute all edge cores using baseline. Then, for each θ-percentile, we randomly sample ten edges, apply **TECM**-Batch-Del to delete them, and reinsert them using the competing insertion algorithms (**TECM**-Ini, **TECM**-Ref, and **TECM**-Batch-Ins). We report the average processing time per edge and corresponding speedup over baseline in Figure 4. Note that **TECM**-Batch-Del results represent deletion performance only, while **TECM**-Ini, **TECM**-Ref, and **TECM**-Batch-Ins results represent insertion performance only. Across all datasets, the baseline BECD achieves speedup ratios between 504× and 8,721×. **TECM**-Ini and **TECM**-Ref reach 879×–131,450× and 1,393×–131,833×, respectively. **TECM**-Batch-Ins and **TECM**-Batch-Del attain the highest efficiency, with speedups of 1,783×–256,855× and 1,526×–187,612×. These results confirm the effectiveness of our approach in maintaining edge cores on dynamic temporal graphs. Figure 4 employs a logarithmic scale with $10^0 = 1$ as the baseline reference. Zero-height bars occur when algorithms achieve speedup ratios ≤ 1 or close to 1, causing them to visually collapse to the baseline line. For instance, BECD achieves 0.5× speedup on StackOverflow with $(0.75, 0.75)$-percentiles, appearing as a zero-height bar. SECD serves as the primary baseline for incremental vs. static comparison, while BECD provides implementation comparison.

As (Δ, θ)-percentiles rise, larger Δ increases temporal degrees and edge-core values, while larger θ selects higher-core edges, yielding denser communities with more Δ-incident neighbors. This expands candidate sets and prolongs updates. On Twitter, at $(0.25, 0.25)$ BECD attains 5.4×10^5 speedup, **TECM**-Ini and **TECM**-Ref reach 6.75×10^5, and **TECM**-Batch-Ins/Del achieve 1.35×10^6 and 9×10^5, respectively, due to small edge-cores and limited candidates. At $(0.75, 0.75)$, speedups drop to 0.8–74.7×, sometimes below 1, with negligible bars in Figure 4, reflecting high-core edges that inflate candidate sets and costs.

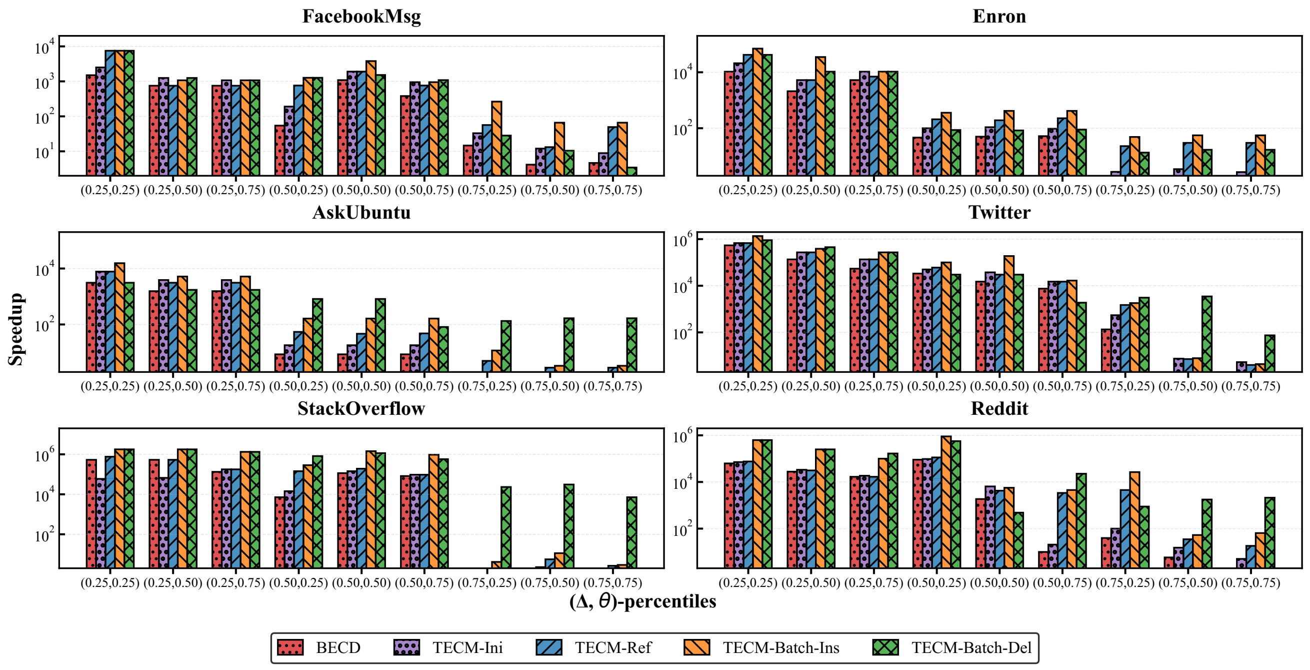

Figure 4: Algorithm speedup compared to baseline.

Among insertion algorithms, **TECM**-Batch-Ins is consistently superior under low percentiles, as each new edge $e = (u, v, t)$ spawns multiple sub-edges across regions. By grouping sub-edges by core values and batching their processing (Section 4.3, Strategy 3.1), it exploits Lemma 4.4 to reduce redundancy and parallelize, markedly lowering overhead. **TECM**-Batch-Del accelerates deletions by bypassing candidate search and applying localized H-index propagation. In contrast, **TECM**-Ref, though using pruning, may underperform **TECM**-Ini when candidate sets are already small, since overhead from subgraph construction and candidate transfer (Algorithm 3) dominates. For example, on Enron at $(0.25, 0.75)$, **TECM**-Ini yields 1.05×10^4 speedup versus 7×10^3 for **TECM**-Ref.

To provide absolute runtime context for the speedup ratios, we demonstrate the calculation using the Reddit dataset with (Δ, θ)-percentiles = $(0.75, 0.75)$. From Figure 3 and Figure 4, SECD requires 2,135 seconds, BECD achieves a 1.5× speedup, and **TECM**-Batch-Ins achieves a 65× speedup. The absolute runtimes are calculated as $T_{\text{method}} = T_{\text{SECD}}/S_{\text{method}}$, yielding BECD time ≈ 1,423 seconds and **TECM**-Batch-Ins time ≈ 33 seconds. This demonstrates that our algorithms achieve not only significant relative improvements but also practically meaningful absolute performance gains.

In addition, we assess the memory efficiency of BECD and TECM-Batch-Ins across multiple datasets under the $(0.5, 0.25)$-percentile configuration with 10 edge updates, measuring their peak Maximum Resident Set Size (RSS). The results demonstrate similar memory footprints between both methods, with only minor differences. This similarity arises from the localized nature of our candidate-based approach, where each edge update processes only a small subgraph, and temporary data structures are efficiently managed and released after processing.

Exp-3: Effectiveness of candidate pruning. To assess the impact of the candidate pruning strategy, we compare the total number of candidates in **TECM**-Ini and **TECM**-Ref. As **TECM**-Batch-Ins employs the same strategy as **TECM**-Ref, its results are excluded. **TECM**-Batch-Del updates edge cores directly, without invoking candidate search. Figure 5 shows the number of candidates found by **TECM**-Ini and **TECM**-Ref under different $(\Delta, \theta)-$*percentiles*. Overall, **TECM**-Ref reduces the number of traversed candidates by 16.8%–65.2% across datasets and configurations, leading to better efficiency than **TECM**-Ini in most cases. However, when the pruning strategy is ineffective—i.e., few candidates exist and none can be preemptively eliminated—**TECM**-Ref incurs extra overhead. For instance, on Enron with $(0.25, 0.75)-$*percentiles*, both algorithms traverse 790 candidates, resulting in **TECM**-Ini achieving higher speedup than **TECM**-Ref. Furthermore, as the $(\Delta, \theta)-$percentiles increase, the number of traversed candidates grows, leading to a reduced speedup ratio for the proposed incremental algorithm. In particular, for Twitter with $(0.75, 0.75)-$percentiles, **TECM**-Ini and **TECM**-Ref identify 5,533,140 and 4,284,880 candidates, respectively. Each edge update requires traversing, on average, 553,314 and 428,488 edges, while the source graph contains only 2,131,270 edges. This extensive candidate traversal substantially degrades the efficiency of the incremental algorithms.

Table 3: Peak memory usage (Maximum RSS, in kB).

Memory usage (kB)	Twitter	StackOverflow	Reddit
BECD	1,015,704	22,055,776	38,612,796
TECM-Batch-Ins	1,015,656	22,055,620	38,612,700

Exp-4: Scalability with different numbers of tasks. In this set of experiments, we assess the scalability of the proposed algorithm under varying numbers of edge updates. We employ a mixed workload design that simulates realistic streaming scenarios where both edge insertions and deletions occur. For each task size K in $\{100, 200, 300, 400, 500\}$, we first apply **TECM**-Batch-Del to

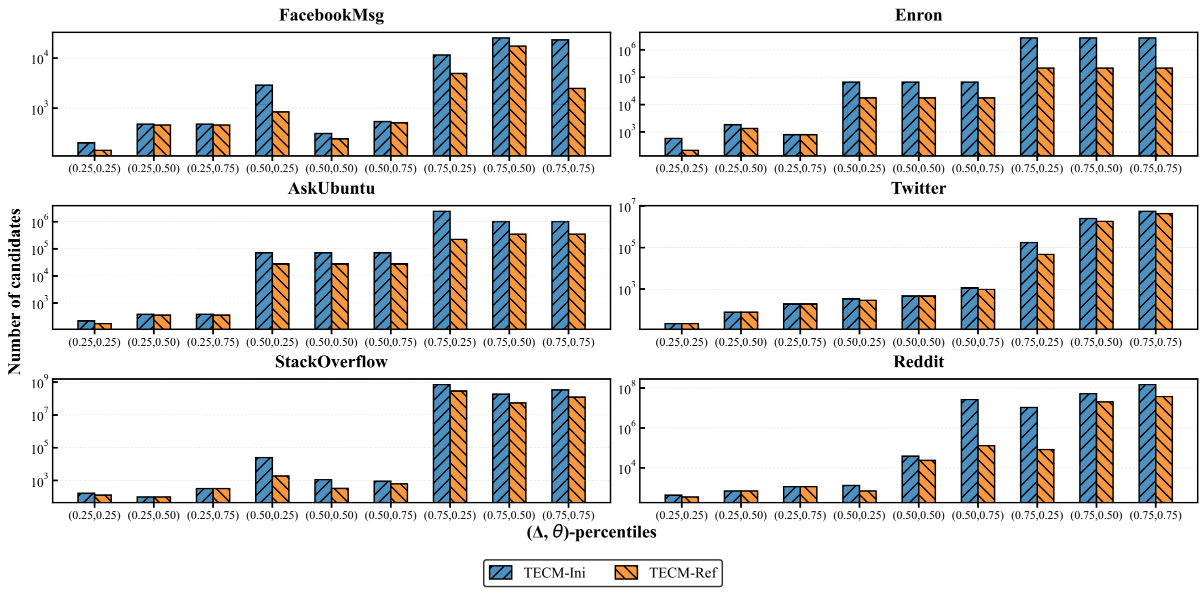

Figure 5: The number of candidates on different settings.

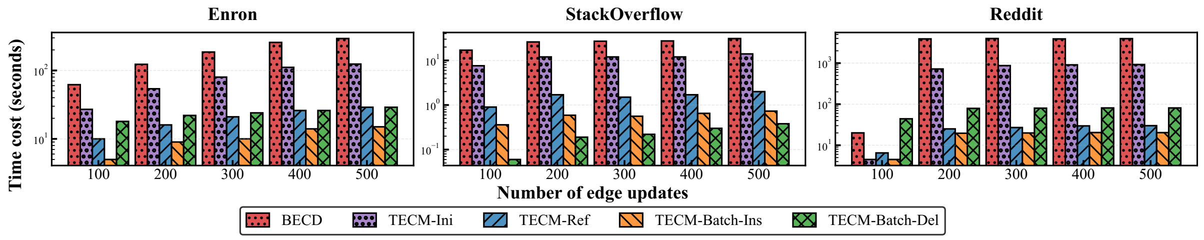

Figure 6: Evaluate the scalability on different number of tasks. (Δ, θ)-percentiles: (0.5, 0.25).

delete K edges sampled uniformly without replacement from the graph, then reinsert the same K edges using the competing insertion algorithms (**TECM**-Ini, **TECM**-Ref, and **TECM**-Batch-Ins). This delete-insert cycle methodology tests algorithms' ability to handle both operation types without graph reset between phases, ensuring the graph state evolves naturally as in real streaming systems. To ensure reproducibility, we use fixed random seeds for edge sampling. We measure and report runtime separately for deletion and insertion phases, enabling detailed analysis of each operation type's scalability characteristics. We randomly generate tasks under the configuration $(0.5, 0.25)-$ *percentiles*. Because we cannot generate enough new edges from AskUbuntu under this configuration, we conduct validation only on the other three datasets.

Our experimental design leverages the composable nature of our single-edge update kernels, where mixed workloads are handled by applying the appropriate kernel (insertion or deletion) for each individual edge update while maintaining correctness guarantees. This design ensures that our algorithms can effectively handle various mixed workload patterns, with correctness preserved regardless of processing order. The results presented in Figure 6 show that as the number of tasks increases linearly, the time consumption of each algorithm also grows linearly. It is worth noting that **TECM**-Batch-Del results represent deletion performance only, while **TECM**-Ini, **TECM**-Ref, and **TECM**-Batch-Ins results represent insertion performance only, as only **TECM**-Batch-Del is designed for the edge deletion scenario. This indicates that the proposed algorithms demonstrate good scalability with respect to both the number of edge updates and the diversity of update operations.

The observed performance characteristics in Figure 6 demonstrate the output-sensitive nature of our algorithms. For **TECM**-Batch-Del on the Reddit dataset, the runtime stabilizes beyond 300 tasks because additional deletions no longer trigger edge-core changes, as the remaining edges have reached their minimum possible core values within the temporal window constraints. This behavior reflects the efficiency advantage of our candidate-based approach, where computational resources are allocated only to edges requiring actual core value updates, consistent with our theoretical analysis that algorithms scale with the affected candidate set size rather than the total number of operations.

Exp-5: Case study. We conduct a case study on the Twitter dataset with second-level temporal resolution, where each edge denotes a retweet labeled as either $fact-checking$ or $misinformation$ (also called *claim*). Misinformation accounts for 81.9% of all edges, reflecting its dominance and virality. We visualize the edge-core distributions using kernel density estimation (KDE), where the x-axis represents discrete edge-core levels (integer values) and the y-axis shows probability density (normalized to unit area per curve), not raw counts. The smooth curves result from KDE smoothing, providing more interpretable visualization of the distribution patterns.

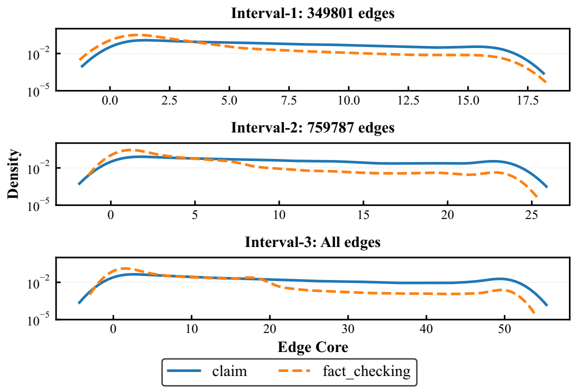

Figure 7: History k-core distribution of *claim* and *fact–checking* edges in Twitter.

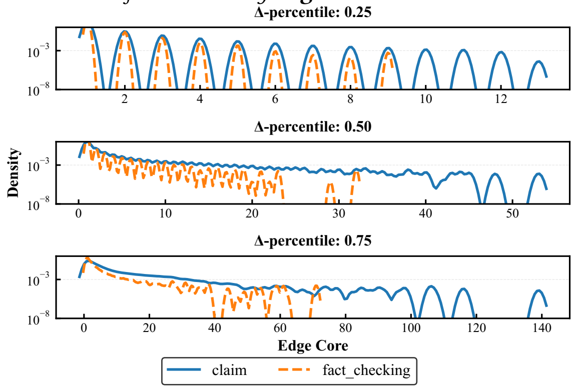

Figure 8: Edge core distribution of *claim* and *fact–checking* edges in Twitter.

We compare temporal edge-core decomposition with the history k-core baseline [69, 70]. In the history k-core setting, each edge is assigned the minimum core of its endpoints, yielding nearly overlapping distributions of *claim* and *fact–checking* edges across core levels (Figure 7) and thus limited discriminative power. In contrast, temporal edge cores produce a clear separation: high-core regions are dominated by *claim* edges, while *fact–checking* edges are largely absent from inner cores (Figure 8). This indicates that misinformation tends to propagate within dense, echo-chamber-like structures, whereas fact-checking information remains more peripheral and cross-community, consistent with established theories on information correction dynamics.

To quantify discrimination more precisely, we compute the Jensen–Shannon divergence (JSD, base 2) between the label-wise core distributions restricted to edges with $k \geq \alpha$ for a range of thresholds α. Figure 9 shows JSD as a function of α, using solid curves for temporal edge cores at different Δ-percentiles and dashed curves for the history k-core. Horizontal dotted lines at 0.1/0.3/0.5 denote weak, moderate, and strong separation. Temporal edge cores yield substantially larger JSD values in high-core regions, most notably at $\Delta = 0.75$, where JSD exceeds 0.5 and approaches 1 at the largest cores, while the history k-core remains near zero in the tail. This pattern shows that temporal edge-core decomposition offers stronger discriminative power.

These results demonstrate the value of edge-centric analysis. Edge core decomposition enables precise discrimination of content

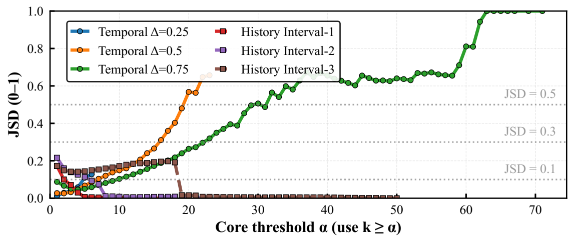

Figure 9: JSD analysis comparing temporal edge-core and history k-core discrimination effectiveness.

structures and filtering of unreliable information by modeling edge resilience. It confirms prior observations about misinformation clustering and provides a scalable method for distinguishing propagation patterns, supporting downstream detection and intervention tasks. Our proposed algorithm enables efficient updates, supporting real-time structural analysis in dynamic temporal graphs.

6 RELATED WORK

Community detection is fundamental in network science and can be seen as a coarse-grained form of subgraph matching [26, 32, 36–38, 61] with broad applications [5, 17, 41]. Communities represent functional modules in protein-protein interaction [48] and metabolic networks [23], user groups in social networks [1], topic-focused websites [6], and large-scale e-commerce user alignment [74]. Classical cohesiveness metrics include k-core [4, 53, 72], k-truss [10, 27], k-clique [59], and k-ECC [19]. For temporal graphs, Wu et al. [66] introduced (k, h)-core decomposition, with subsequent incremental algorithms [11, 27, 40, 51, 58, 73] for dynamic maintenance.

Temporal graphs capture time-varying interactions essential for real-world applications [7, 34, 65]. Community detection on temporal graphs is crucial for understanding dynamic systems [39, 49, 63]. Galimberti et al. [18] proposed (k, I)-span-cores with time intervals, while Hung et al. [30] introduced (L, K)-lasting cores persisting for L time steps. Oettershagen et al. [47] proposed the (k, Δ)-core paradigm for hierarchical decomposition, explicitly accounting for temporal edge correlations. However, maintaining (k, Δ)-core communities in dynamic temporal graphs remains unaddressed.

On the system side, parallel and GPU-accelerated methods have been proposed for dynamic graph analytics, traversal, and subgraph enumeration [13, 24, 25, 43, 54–56, 68], but they mainly target static or time-agnostic settings and do not model temporal edge-core constraints. We address this gap by focusing on incremental maintenance of (k, Δ)-core decomposition, proposing efficient algorithms for dynamic updates in temporal networks.

7 CONCLUSION

This paper presents **TECM**, a framework for efficient incremental maintenance of temporal edge-cores in streaming graphs. Built on theoretical insights that localize update impacts via root edges and Δ-reachable paths, **TECM** integrates task decomposition, Δ-aware traversal, and localized support analysis. Candidate pruning and batch processing strategies further enhance efficiency. Experiments demonstrate orders-of-magnitude speedups over static baselines, offering a scalable solution for real-time temporal graph analysis.

REFERENCES

[1] Alessandro Acquisti and Ralph Gross. 2006. Imagined communities: Awareness, information sharing, and privacy on the Facebook. In *International workshop on privacy enhancing technologies*. Springer, 36–58.

[2] Leman Akoglu, Hanghang Tong, and Danai Koutra. 2015. Graph based anomaly detection and description: a survey. *Data mining and knowledge discovery* 29 (2015), 626–688.

[3] Rasim Alguliyev, Ramiz Aliguliyev, and Farhad Yusifov. 2021. Graph modelling for tracking the COVID-19 pandemic spread. *Infectious disease modelling* 6 (2021), 112–122.

[4] Vladimir Batagelj and Matjaz Zaversnik. 2003. An o (m) algorithm for cores decomposition of networks. CoRR. *arXiv preprint cs.DS/0310049* 37 (2003).

[5] Angela Bonifati, M Tamer Özsu, Yuanyuan Tian, Hannes Voigt, Wenyuan Yu, and Wenjie Zhang. 2024. The future of graph analytics. In *Companion of the 2024 International Conference on Management of Data*. 544–545.

[6] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. 2000. Graph structure in the web. *Computer networks* 33, 1-6 (2000), 309–320.

[7] Liuyi Chen, Yi Ding, Xushuo Tang, Fangyue Chen, Siyuan Gong, Xu Zhou, and Zhengyi Yang. 2025. Accelerating Streaming Subgraph Matching via Vector Databases. *Intelligent Computing* (2025).

[8] Xin Chen, Jieming Shi, You Peng, Wenqing Lin, Sibo Wang, and Wenjie Zhang. 2024. Minimum Strongly Connected Subgraph Collection in Dynamic Graphs. *Proceedings of the VLDB Endowment* 17, 6 (2024), 1324–1336.

[9] Deming Chu, Fan Zhang, Wenjie Zhang, Xuemin Lin, Ying Zhang, Yinglong Xia, and Chenyi Zhang. 2024. Discovering and Maintaining the Best k in Core Decomposition. *IEEE Transactions on Knowledge and Data Engineering* (2024).

[10] Jonathan Cohen. 2008. Trusses: Cohesive subgraphs for social network analysis. *National security agency technical report* 16, 3.1 (2008), 1–29.

[11] Apurba Das, Michael Svendsen, and Srikanta Tirthapura. 2019. Incremental maintenance of maximal cliques in a dynamic graph. *The VLDB Journal* 28 (2019), 351–375.

[12] Zeineb Dhouioui and Jalel Akaichi. 2014. Tracking dynamic community evolution in social networks. In *2014 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2014)*. IEEE, 764–770.

[13] Yi Ding, Zhengyi Yang, Shunyang Li, Liuyi Chen, Haoran Ning, Kongzhang Hao, and Yongfei Liu. 2024. FGAQ: Accelerating Graph Analytical Queries Using FPGA. In *Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint International Conference on Web and Big Data*. Springer, 357–361.

[14] Jinyu Duan, Haicheng Guo, Fan Zhang, Kai Wang, Zhengping Qian, and Zhihong Tian. 2025. The k-Trine Cohesive Subgraph and Its Efficient Algorithms. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 1*. 271–282.

[15] Yifan Duan, Guibin Zhang, Shilong Wang, Xiaojiang Peng, Wang Ziqi, Junyuan Mao, Hao Wu, Xinke Jiang, and Kun Wang. 2024. Cat-gnn: Enhancing credit card fraud detection via causal temporal graph neural networks. *arXiv preprint arXiv:2402.14708* (2024).

[16] Wenfei Fan, Chao Tian, Ruiqi Xu, Qiang Yin, Wenyuan Yu, and Jingren Zhou. 2021. Incrementalizing graph algorithms. In *Proceedings of the 2021 International Conference on Management of Data*. 459–471.

[17] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. 2020. A survey of community search over big graphs. *The VLDB Journal* 29 (2020), 353–392.

[18] Edoardo Galimberti, Martino Ciaperoni, Alain Barrat, Francesco Bonchi, Ciro Cattuto, and Francesco Gullo. 2020. Span-core decomposition for temporal networks: Algorithms and applications. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 15, 1 (2020), 1–44.

[19] Alan Gibbons. 1985. *Algorithmic graph theory*. Cambridge university press.

[20] Jennifer Golbeck. 2015. *Introduction to social media investigation: A hands-on approach*. Syngress.

[21] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. 2020. FastFabric: Scaling hyperledger fabric to 20 000 transactions per second. *International Journal of Network Management* 30, 5 (2020), e2099.

[22] Derek Greene, Donal Doyle, and Padraig Cunningham. 2010. Tracking the evolution of communities in dynamic social networks. In *2010 international conference on advances in social networks analysis and mining*. IEEE, 176–183.

[23] Roger Guimera and Luís A Nunes Amaral. 2005. Functional cartography of complex metabolic networks. *nature* 433, 7028 (2005), 895–900.

[24] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. 2020. GPU-Accelerated Subgraph Enumeration on Partitioned Graphs. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, 1067–1082.

[25] Wentian Guo, Yuchen Li, Mo Sha, and Kian-Lee Tan. 2017. Parallel Personalized Pagerank on Dynamic Graphs. *Proc. VLDB Endow.* 11, 1 (2017), 93–106.

[26] Kongzhang Hao, Zhengyi Yang, Longbin Lai, Zhengmin Lai, Xin Jin, and Xuemin Lin. 2019. PatMat: a distributed pattern matching engine with cypher. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 2921–2924.

[27] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1311–1322.

[28] Xuanwen Huang, Yang Yang, Yang Wang, Chunping Wang, Zhisheng Zhang, Jiarong Xu, Lei Chen, and Michalis Vazirgiannis. 2022. Dgraph: A large-scale financial dataset for graph anomaly detection. *Advances in Neural Information Processing Systems* 35 (2022), 22765–22777.

[29] Xiaoke Huang, Ye Zhao, Chao Ma, Jing Yang, Xinyue Ye, and Chong Zhang. 2015. TrajGraph: A graph-based visual analytics approach to studying urban network centralities using taxi trajectory data. *IEEE transactions on visualization and computer graphics* 22, 1 (2015), 160–169.

[30] Wei-Chun Hung and Chih-Ying Tseng. 2021. Maximum (l, k)-lasting cores in temporal social networks. In *International Conference on Database Systems for Advanced Applications*. Springer, 336–352.

[31] Paul Irofti, Andrei Pătraşcu, and Andra Băltoiu. 2021. Fraud detection in networks. *Enabling AI Applications in Data Science* (2021), 517–536.

[32] Xin Jin, Zhengyi Yang, Xuemin Lin, Shiyu Yang, Lu Qin, and You Peng. 2021. Fast: Fpga-based subgraph matching on massive graphs. In *2021 IEEE 37th international conference on data engineering (ICDE)*. IEEE, 1452–1463.

[33] Yejin Kim, Youngbin Lee, Minyoung Choe, Sungju Oh, and Yongjae Lee. 2024. Temporal graph networks for graph anomaly detection in financial networks. *arXiv preprint arXiv:2404.00060* (2024).

[34] Vassilis Kostakos. 2009. Temporal graphs. *Physica A: Statistical Mechanics and its Applications* 388, 6 (2009), 1007–1023.

[35] Valerio La Gatta, Vincenzo Moscato, Marco Postiglione, and Giancarlo Sperli. 2020. An epidemiological neural network exploiting dynamic graph structured data applied to the COVID-19 outbreak. *IEEE Transactions on Big Data* 7, 1 (2020), 45–55.

[36] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, et al. 2019. Distributed subgraph matching on timely dataflow. *Proceedings of the VLDB Endowment* 12, 10 (2019), 1099–1112.

[37] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, et al. 2019. A survey and experimental analysis of distributed subgraph matching. *arXiv preprint arXiv:1906.11518* (2019).

[38] Zhengmin Lai, Zhengyi Yang, and Longbin Lai. 2019. Improving Distributed Subgraph Matching Algorithm on Timely Dataflow. In *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 266–273.

[39] Ling Li, Yuhai Zhao, Yuan Li, Fazal Wahab, and Zhengkui Wang. 2022. The most active community search in large temporal graphs. *Knowledge-Based Systems* 250 (2022), 109101.

[40] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. 2013. Efficient core maintenance in large dynamic graphs. *IEEE transactions on knowledge and data engineering* 26, 10 (2013), 2453–2465.

[41] Yu Liu, Qi Luo, Yanwei Zheng, Wenjie Zhang, Xuemin Lin, and Dongxiao Yu. 2025. Effective and Efficient Community Search over Large-Scale Hypergraphs. (2025).

[42] Linyuan Lü, Tao Zhou, Qian-Ming Zhang, and H Eugene Stanley. 2016. The H-index of a network node and its relation to degree and coreness. *Nature communications* 7, 1 (2016), 10168.

[43] Qiuyi Lyu, Mo Sha, Bin Gong, and Kuangda Lyu. 2021. Accelerating Depth-First Traversal by Graph Ordering. In *SSDBM 2021: 33rd International Conference on Scientific and Statistical Database Management, Tampa, FL, USA, July 6-7, 2021*. ACM, 13–24.

[44] Xiaoxiao Ma, Jia Wu, Shan Xue, Jian Yang, Chuan Zhou, Quan Z Sheng, Hui Xiong, and Leman Akoglu. 2021. A comprehensive survey on graph anomaly detection with deep learning. *IEEE transactions on knowledge and data engineering* 35, 12 (2021), 12012–12038.

[45] Yihong Ma, Patrick Gerard, Yijun Tian, Zhichun Guo, and Nitesh V Chawla. 2022. Hierarchical spatio-temporal graph neural networks for pandemic forecasting. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*. 1481–1490.

[46] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. 2011. Distributed k-core decomposition. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on principles of distributed computing*. 207–208.

[47] Lutz Oettershagen, Athanasios L Konstantinidis, and Giuseppe F Italiano. 2025. An Edge-Based Decomposition Framework for Temporal Networks. In *Proceedings of the Eighteenth ACM International Conference on Web Search and Data Mining*. 735–743.

[48] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. 2005. Uncovering the overlapping community structure of complex networks in nature and society. *nature* 435, 7043 (2005), 814–818.

[49] Hongchao Qin, Rong-Hua Li, Ye Yuan, Guoren Wang, Lu Qin, and Zhiwei Zhang. 2022. Mining bursting core in large temporal graphs. *Proceedings of the VLDB Endowment* (2022).

[50] Subhajit Sahu. 2024. DF Louvain: Fast Incrementally Expanding Approach for Community Detection on Dynamic Graphs. *arXiv preprint arXiv:2404.19634* (2024).

[51] Ahmet Erdem Saríyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V Çatalyürek. 2013. Streaming algorithms for k-core decomposition. *Proceedings of the VLDB Endowment* 6, 6 (2013), 433–444.

[52] Ahmet Erdem Sarıyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V Çatalyürek. 2016. Incremental k-core decomposition: algorithms and evaluation. *The VLDB Journal* 25, 3 (2016), 425–447.

[53] Stephen B Seidman. 1983. Network structure and minimum degree. *Social networks* 5, 3 (1983), 269–287.

[54] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. 2017. Accelerating Dynamic Graph Analytics on GPUs. *Proc. VLDB Endow.* 11, 1 (2017), 107–120.

[55] Mo Sha, Yuchen Li, and Kian-Lee Tan. 2019. GPU-based Graph Traversal on Compressed Graphs. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.* ACM, 775–792.

[56] Mo Sha, Yuchen Li, and Kian-Lee Tan. 2021. Self-adaptive Graph Traversal on GPUs. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021.* ACM, 1558–1570.

[57] Guodao Sun, Tan Tang, Tai-Quan Peng, Ronghua Liang, and Yingcai Wu. 2017. Socialwave: visual analysis of spatio-temporal diffusion of information on social media. *ACM Transactions on Intelligent Systems and Technology (TIST)* 9, 2 (2017), 1–23.

[58] Anxin Tian, Alexander Zhou, Yue Wang, and Lei Chen. 2023. Maximal D-truss search in dynamic directed graphs. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2199–2211.

[59] William Thomas Tutte. 2001. *Graph theory.* Vol. 21. Cambridge university press.

[60] Kai Wang, Shuting Wang, Xin Cao, and Lu Qin. 2020. Efficient radius-bounded community search in geo-social networks. *IEEE Transactions on Knowledge and Data Engineering* 34, 9 (2020), 4186–4200.

[61] Ran Wang, Zhengyi Yang, Wenjie Zhang, and Xuemin Lin. 2020. An empirical study on recent graph database systems. In *International Conference on Knowledge Science, Engineering and Management.* Springer International Publishing Cham, 328–340.

[62] Yishu Wang, Ye Yuan, Yuliang Ma, and Guoren Wang. 2019. Time-dependent graphs: Definitions, applications, and algorithms. *Data Science and Engineering* 4, 4 (2019), 352–366.

[63] Tongfeng Weng, Yumeng Liu, Mo Sha, Xinyuan Chen, Xu Zhou, Kenli Li, and Kian-Lee Tan. 2025. Efficient Projection-Based Algorithms for Tip Decomposition on Dynamic Bipartite Graphs. *IEEE Trans. Knowl. Data Eng.* 37, 2 (2025), 626–640.

[64] Tongfeng Weng, Xu Zhou, Kenli Li, Peng Peng, and Keqin Li. 2021. Efficient distributed approaches to core maintenance on large dynamic graphs. *IEEE Transactions on Parallel and Distributed Systems* 33, 1 (2021), 129–143.

[65] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. 2014. Path problems in temporal graphs. *Proceedings of the VLDB Endowment* 7, 9 (2014), 721–732.

[66] Huanhuan Wu, James Cheng, Yi Lu, Yiping Ke, Yuzhen Huang, Da Yan, and Hejun Wu. 2015. Core decomposition in large temporal graphs. In *2015 IEEE international conference on big data (Big data).* IEEE, 649–658.

[67] Yanping Wu, Renjie Sun, Xiaoyang Wang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2024. Efficient maximal temporal plex enumeration. In *2024 IEEE 40th International Conference on Data Engineering (ICDE).* IEEE, 3098–3110.

[68] Zhengyi Yang, Longbin Lai, Xuemin Lin, Kongzhang Hao, and Wenjie Zhang. 2021. Huge: An efficient and scalable subgraph enumeration system. In *Proceedings of the 2021 international conference on management of data.* 2049–2062.

[69] 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).

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

[71] Anita Zakrzewska and David A Bader. 2015. A dynamic algorithm for local community detection in graphs. In *Proceedings of the 2015 IEEE/ACM international conference on advances in social networks analysis and mining 2015.* 559–564.

[72] Wenqian Zhang, Zhengyi Yang, Dong Wen, Wentao Li, Wenjie Zhang, and Xuemin Lin. 2025. Accelerating Core Decomposition in Billion-Scale Hypergraphs. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–27.

[73] Yikai Zhang, Jeffrey Xu Yu, Ying Zhang, and Lu Qin. 2017. A fast order-based approach for core maintenance. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE).* IEEE, 337–348.

[74] Vincent W. Zheng, Mo Sha, Yuchen Li, Hongxia Yang, Yuan Fang, Zhenjie Zhang, Kian-Lee Tan, and Kevin Chen-Chuan Chang. 2018. Heterogeneous Embedding Propagation for Large-Scale E-Commerce User Alignment. In *IEEE International Conference on Data Mining, ICDM 2018, Singapore, November 17-20, 2018.* IEEE Computer Society, 1434–1439.