Scalable Time-Range k-Core Query on Temporal Graphs

Junyong Yang
School of Computer Science, Wuhan University
Wuhan, China
thomasyang@whu.edu.cn

Ming Zhong*
School of Computer Science, Wuhan University
Wuhan, China
clock@whu.edu.cn

Yuanyuan Zhu
School of Computer Science, Wuhan University
Wuhan, China
yyzhu@whu.edu.cn

Tieyun Qian
School of Computer Science, Wuhan University
Wuhan, China
qty@whu.edu.cn

Mengchi Liu
South China Normal University
Guangzhou, China
liumengchi@scnu.edu.cn

Jeffrey Xu Yu
The Chinese University of Hong Kong
Hong Kong, China
yu@se.cuhk.edu.hk

ABSTRACT

Querying cohesive subgraphs on temporal graphs with various time constraints has attracted intensive research interests recently. In this paper, we study a novel Temporal k-Core Query (TCQ) problem: given a time interval, find all distinct k-cores that exist within any subintervals from a temporal graph, which generalizes the previous historical k-core query. This problem is challenging because the number of subintervals increases quadratically to the span of time interval. For that, we propose a novel Temporal Core Decomposition (TCD) algorithm that decrementally induces temporal k-cores from the previously induced ones and thus reduces "intra-core" redundant computation significantly. Then, we introduce an intuitive concept named Tightest Time Interval (TTI) for temporal k-core, and design an optimization technique with theoretical guarantee that leverages TTI as a key to predict which subintervals will induce duplicated k-cores and prunes the subintervals completely in advance, thereby eliminating "inter-core" redundant computation. The complexity of optimized TCD (OTCD) algorithm no longer depends on the span of query time interval but only the scale of final results, which means OTCD algorithm is scalable. Moreover, we propose a compact in-memory data structure named Temporal Edge List (TEL) to implement OTCD algorithm efficiently in physical level with bounded memory requirement. TEL organizes temporal edges in a "timeline" and can be updated instantly when new edges arrive in dynamical temporal graphs. We compare OTCD algorithm with the incremental historical k-core query on several real-world temporal graphs, and observe that OTCD algorithm outperforms it by three orders of magnitude, even though OTCD algorithm needs none precomputed index.

PVLDB Reference Format:
Junyong Yang, Ming Zhong, Yuanyuan Zhu, Tieyun Qian, Mengchi Liu, and Jeffrey Xu Yu. Scalable Time-Range k-Core Query on Temporal Graphs. PVLDB, 16(5): 1168 - 1180, 2023.
doi:10.14778/3579075.3579089

*The corresponding author.

Proceedings of the VLDB Endowment, Vol. 16, No. 5 ISSN 2150-8097.
doi:10.14778/3579075.3579089

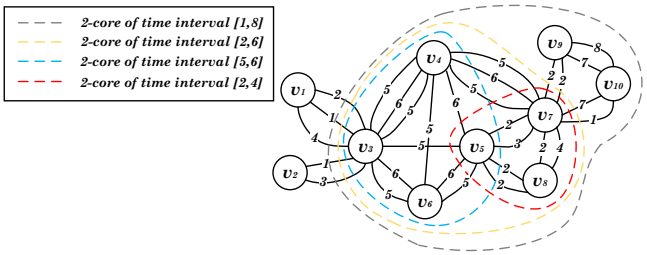


Figure 1: A running example of temporal graph.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/graphlab-whu/Temporal-k-Core-Query-Project.

1 INTRODUCTION

1.1 Motivation

Discovering communities or cohesive subgraphs from temporal graphs has great values in many application scenarios, thereby attracting intensive research interests [1, 5, 12, 19, 25, 27, 34, 37] in recent years. Here, a temporal graph refers to an undirected multigraph in which each edge has a timestamp to indicate when it occurred, as illustrated in Figure 1. For example, consider a graph consisting of bank accounts as vertices and fund transfer transactions between accounts as edges with natural timestamps. For applications such as anti-money-laundering, we would like to search communities like k-cores that contain a known suspicious account and emerge within a specific time interval like the FIFA World Cup, and investigate the associated accounts.

To address the community query/search problem for a fixed time interval, the concept of historical k-core [37] is proposed recently, which is the k-core induced from a projected subgraph of the temporal graph in which all edges occurred out of the time interval have been excluded and the parallel edges between each pair of vertices have been merged. Also, the PHC-Query method is proposed to deal with historical k-core query/search by using a precomputed index efficiently.

However, we usually do not know the exact time interval of targeted historical k-core in real-world applications. Actually, if we can know the exact time interval, a traditional core decomposition

on the projected subgraph over the given time interval is efficient enough to address the problem. Thus, it is more reasonable to assume that we can only offer a flexible time interval and need to induce cores from all its subintervals. For example, for detecting money laundering by soccer gambling during the FIFA World Cup, the k-cores emerged over a few of hours around one of the matches are more valuable than a large k-core emerged over the whole month.

Therefore, we aim to generalize historical k-core query by allowing the result k-cores to be induced by any subinterval of a given time interval, like "flexible versus fixed". The historical k-core query can be seen as a special case of our problem that only evaluates the whole interval. Consider the following example.

Example 1. *As illustrated in Figure 1, given a time interval [1,8], historical k-core query only returns the largest core marked by the grey dashed line. In contrast, our temporal k-core query returns four cores marked by dashed lines with different colors. These cores can reveal various insights unseen by the largest one. For example, some cores like red and blue that emerge in bursty periods may be caused by special events. Also, some persistent or periodic cores may be found. Further, we can analyze the interaction between cores and how they evolve over time, such as the small cores like red and blue are merged to the large cores like yellow. Lastly, some underlying details may be found. During the merge, the vertex v_5 may play a vital role because it appears in all the cores.*

The general and flexible temporal k-core query we study is naturally a generalization of existing query models like historical k-core and potentially supports various temporal graph analytics tasks mentioned in the above example.

1.2 Contribution

In this paper, we study a novel *temporal k-core query* problem: given a time interval, find all distinct k-cores that exist within any subintervals from a temporal graph. Although the existing PHC-Query returns the historical k-core of a fixed time interval efficiently, it cannot be trivially applied to deal with the new problem. Because inducing k-cores for each subinterval individually from scratch is not scalable, since the number of subintervals increases quadratically to the span of time interval. Moreover, PHC-Query suffers from two other intrinsic shortcomings. Firstly, it relies on a PHC-Index that precomputes the coreness of all vertices over all time intervals, thereby incurring heavy offline time and space overheads. Secondly, due to its sophisticated construction, it is unclear if PHC-Index can be updated dynamically. It is against the dynamic nature of temporal graphs.

In order to overcome the above challenges, we present a novel *temporal core decomposition* algorithm and auxiliary optimization and implementation techniques. Our contributions can be summarized as follows.

- We formalize a general time-range cohesive subgraph query problem on ubiquitous temporal graphs, namely, temporal k-core query. Many previous typical k-core query models on temporal graphs can be equivalently represented by temporal k-core query with particular constraints.
- To address temporal k-core query, we propose a simple and yet efficient algorithm framework based on a novel temporal core decomposition operation. By using temporal core decomposition, our algorithm always decrementally induces a temporal k-core from the previous induced temporal k-core except the initial one, thereby reducing redundant computation significantly.
- Moreover, we propose an intuitive concept named tightest time interval for temporal k-core. According to the properties of tightest time intervals, we design three pruning rules with theoretical guarantee to directly skip subintervals that will not induce distinct temporal k-core. As a result, the optimized algorithm is scalable in terms of the span of query time interval, since only the necessary subintervals are enumerated.
- For physical implementation of our algorithm, we propose a both space and time efficient data structure named temporal edge list to represent a temporal graph in memory. It can be manipulated to perform temporal core decomposition and tightest time interval based pruning rapidly with bounded memory. More importantly, temporal edge list can be incrementally updated with evolving temporal graphs, so that our approach can support dynamical graph applications naturally.
- Lastly, we evaluate the efficiency and effectiveness of our algorithm on real-world datasets. The experimental results demonstrate that our algorithm outperforms the improved PHC-Query by three orders of magnitude.

2 PRELIMINARY

2.1 Data Model

A *temporal graph* is normally an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with parallel temporal edges. Each temporal edge $(u, v, t) \in \mathcal{E}$ is associated with a timestamp t that indicates when the interaction happened between the vertices $u, v \in \mathcal{V}$. For example, the temporal edges could be transfer transactions between bank accounts in a finance graph. Without a loss of generality, we use continuous integers that start from 1 to denote timestamps. Figure 1 illustrates a temporal graph as our running example.

There are two useful concepts derived from the temporal graph. Given a time interval $[ts, te]$, we define the *projected graph* of $\mathcal{G}$ over $[ts, te]$ as $\mathcal{G}_{[ts,te]} = (\mathcal{V}_{[ts,te]}, \mathcal{E}_{[ts,te]})$, where $\mathcal{V}_{[ts,te]} = \mathcal{V}$ and $\mathcal{E}_{[ts,te]} = \{(u, v, t) | (u, v, t) \in \mathcal{E}, t \in [ts, te]\}$. Moreover, we define the *detemporalized graph* of $\mathcal{G}_{[ts,te]}$ as a simple graph $G_{[ts,te]} = (V_{[ts,te]}, E_{[ts,te]})$, where $V_{[ts,te]}$=$\mathcal{V}_{[ts,te]}$ and $E_{[ts,te]} = \{(u, v) | (u, v, t) \in \mathcal{E}_{[ts,te]}\}$.

2.2 Query Model

For revealing communities in graphs, the k-core query is widely adopted. Given an undirected graph G and an integer k, k-core is the maximal induced subgraph of G in which all vertices have degrees at least k, which is denoted by $C^k(G)$. The *coreness* of a vertex v in a graph G is the largest value of k such that $v \in C^k(G)$.

For temporal graphs, the Historical k-Core Query (HCQ) [37] is proposed recently. It aims to find a k-core that appears during a specific time interval. Formally, a historical k-core $\mathcal{H}^k_{[ts,te]}(\mathcal{G})$ is a

k-core in the detemporalized projected graph $G_{[ts,te]}$ of $\mathcal{G}$. Thus, HCQ can be defined as follows.

Definition 1 (Historical k-Core Query). *For a temporal graph $\mathcal{G}$, given an integer k and a time interval $[ts, te]$, return $\mathcal{H}^k_{[ts,te]}(\mathcal{G}) = C^k(G_{[ts,te]})$.*

In this paper, we propose a novel query model called Temporal k-Core Query (TCQ) that generalizes HCQ. The main difference is that the query time interval $[Ts, Te]$ of TCQ is a range but not fixed query condition like $[ts, te]$ of HCQ. In TCQ, Ts and Te are the minimum start time and maximum end time of query time interval respectively, and thereby the k-cores induced by each subinterval $[ts, te] \subseteq [Ts, Te]$ are all potential results of TCQ. Moreover, TCQ directly returns the maximal induced subgraphs of $\mathcal{G}$ in which all vertices have degrees (note that, the number of neighbor vertices but not neighbor edges) at least k as results. We call these subgraphs as *temporal k-cores* and denote by $\mathcal{T}^k_{[ts,te]}(\mathcal{G})$ a temporal k-core that appears over $[ts, te]$ on $\mathcal{G}$. Obviously, a historical k-core $\mathcal{H}^k_{[ts,te]}(\mathcal{G})$ is the detemporalized temporal k-core $\mathcal{T}^k_{[ts,te]}(\mathcal{G})$. Therefore, TCQ can be seen as a group of HCQ and HCQ can be seen as a special case of TCQ. The formal definition of TCQ is as follows.

Definition 2 (Temporal k-Core Query). *For a temporal graph $\mathcal{G}$, given an integer k and a time interval $[Ts, Te]$, return all distinct $\mathcal{T}^k_{[ts,te]}(\mathcal{G})$ with $[ts, te] \subseteq [Ts, Te]$.*

Note that, TCQ only returns the distinct temporal k-cores that are not identical to each other, since multiple subintervals of $[Ts, Te]$ may induce an identical subgraph of $\mathcal{G}$. For brevity, $\mathcal{T}^k_{[ts,te]}(\mathcal{G})$ is abbreviated as $\mathcal{T}^k_{[ts,te]}$ if the context is self-evident.

2.3 Baseline Algorithm

A straightforward solution to TCQ is to enumerate each subinterval $[ts, te] \subseteq [Ts, Te]$ and induce $\mathcal{T}^k_{[ts,te]}$ respectively, which takes $O(|Te - Ts|^2|\mathcal{E}|)$ time. However, the span of query time interval (namely, $Te - Ts$) can be extremely large in practice, which results in enormous time consumption for inducing all temporal k-cores from scratch independently. Therefore, we start from a non-trivial baseline based on the existing PHC-Query.

2.3.1 A Short Review of PHC-Query. PHC-Query relies on a heavy-weight index called PHC-Index that essentially precomputes the coreness of all vertices in the projected graphs over all possible time intervals. The index is logically a table that stores a set of timestamp pairs for each vertex $v \in \mathcal{V}$ (column) and each reasonable coreness k (row). Given a value of k, the coreness of a vertex v is exactly k in the projected graph over $[ts, te]$ for each timestamp pair ts and te in the cell (k, v). In particular, due to the monotonicity of coreness of a vertex with respect to te when ts is fixed, PHC-Index can reduce its space cost significantly by only storing the necessary but not all possible timestamp pairs. Specifically, for a vertex v, a coreness k and a start time ts, only a discrete set of *core time* need to be recorded, since the coreness of the vertex over $[ts, te]$ will not change with the increase of te until te is a core time. Consequently, given an HCQ instance, PHC-Query leverages PHC-Index to directly determine whether a vertex has the coreness no less than the required k, by comparing the query time interval with the retrieved timestamp pairs, and then induces historical k-cores with qualified vertices.

2.3.2 Incremental PHC-Query Algorithm. The main idea of our baseline algorithm is to induce temporal k-cores incrementally, thereby reducing redundant computation. With a temporal k-core $\mathcal{T}^k_{[ts,te]}$, we induce $\mathcal{T}^k_{[ts,te+1]}$ simply by appending new vertices to $\mathcal{T}^k_{[ts,te]}$, whose coreness has become no less than k due to the expand of time interval. Those vertices can be directly identified by using core time retrieved from PHC-Index since ts is fixed. The correctness of baseline algorithm is guaranteed while the correctness of PHC-Query holds.

The pseudo code of incremental PHC-Query (iPHC-Query) algorithm is presented in Algorithm 1. It enumerates all subintervals of a given $[Ts, Te]$ in a particular order for fulfilling efficient incremental temporal k-core induction. Specifically, it anchors the value of ts (line 1), and increases the value of te from ts to Te (line 5), so that $\mathcal{T}^k_{[ts,te+1]}$ can always be incrementally generated from an existing $\mathcal{T}^k_{[ts,te]}$. For each ts anchored and the input k, the algorithm firstly retrieves the core time of all vertices from PHC-Index, and pushes the vertices into a minimum heap $\mathbb{H}_v$ ordered by their core time (line 3). Moreover, all temporal edges with timestamps in $[ts, Te]$ are pushed into another minimum heap $\mathbb{H}_e$ ordered by their timestamp (line 4). Then, the algorithm maintains a vertex set $\mathbb{V}$ and an edge set $\mathbb{E}$, which represent the vertices and edges of $\mathcal{T}^k_{[ts,te]}$ respectively, whenever te is increased by the following steps. It pops remaining vertices with core time no greater than te from $\mathbb{H}_v$ and adds them to $\mathbb{V}$ (line 6), since the corenesss of these vertices are no less than k according to PHC-Index. Also, it pops remaining edges with timestamp no greater than te from $\mathbb{H}_e$ and adds them to $\mathbb{E}$ if both vertices linked by the edges are in $\mathbb{V}$ (line 7). Then, it puts back the popped edges that are not in $\mathbb{E}$ into $\mathbb{H}_e$ (line 8), because they could still be contained by other temporal k-cores induced later. Lastly, a temporal k-core comprised of $\mathbb{V}$ and $\mathbb{E}$ that are not empty is collected if it has not been induced before (line 9). The complexity analysis of baseline algorithm can be found in our full technical report [35].

Although the baseline algorithm can achieve incremental induction of temporal k-core for each start time, PHC-Index incurs a huge amount of extra space and time overheads. Moreover, its incremental induction only offers a kind of "intra-core" optimization that reduces the redundant computation in each temporal k-core induction, and lacks of a kind of "inter-core" optimization that can directly avoids inducing some temporal k-cores.

3 ALGORITHM

In this section, we propose a novel efficient algorithm to address TCQ. Our algorithm leverages a fundamental operation called *temporal core decomposition* to induce $\mathcal{T}^k_{[ts,te]}$ from $\mathcal{T}^k_{[ts,te+1]}$ decrementally. More importantly, our algorithm does not require any precomputation and index space, and can still outperform the baseline algorithm.

Algorithm 1: Baseline iPHC-Query algorithm.

Input: $\mathcal{G}$, k, $[Ts, Te]$
Output: all distinct $\mathcal{T}^k_{[ts,te]}$ with $[ts, te] \subseteq [Ts, Te]$

```
for ts ← Ts to Te do
    V ← ∅, E ← ∅, H_v ← ∅, H_e ← ∅
    for k and ts, retrieve the core time of each vertex in G
      from PHC-Index and push them into H_v
    push the temporal edges with timestamps in [ts, Te] in
      G into H_e
    for te ← ts to Te do
        pop a vertex from H_v and add it to V, until the min
          core time of H_v exceeds te
        pop an edge from H_e and add it to E if both vertices
          linked by this edge are in V, until the min
          timestamp of H_e exceeds te
        push all edges that have been popped from H_e and
          are not added to E back to H_e
        collect T^k_[ts,te] = (V, E) if it is neither empty nor
          identical to other existing results
```

3.1 Temporal Core Decomposition (TCD)

Firstly, we introduce Temporal Core Decomposition (TCD) as a basic operation on temporal graphs, which is derived from the traditional *core decomposition* [2] on ordinary graphs. TCD refers to a two-step operation of inducing a temporal k-core $\mathcal{T}^k_{[ts,te]}$ of a given time interval $[ts, te]$ from a given temporal graph $\mathcal{G}$. The first step is *truncation*: remove temporal edges with timestamps not in $[ts, te]$ from $\mathcal{G}$, namely, induce the projected graph $\mathcal{G}_{[ts,te]}$. The second step is *decomposition*: iteratively peel vertices with degree (the number of neighbor vertices but not neighbor edges) less than k and the edges linked to them together. The correctness of TCD is as intuitive as core decomposition.

An excellent property of TCD operation is that, it can induce a temporal k-core $\mathcal{T}^k_{[ts,te]}$ from another temporal k-core $\mathcal{T}^k_{[ts',te']}$ with $[ts, te] \subset [ts', te']$, so that we can develop a decremental algorithm based on TCD operation to achieve efficient processing of TCQ. To prove the correctness of this property, let us consider the following Theorem 1. All proofs of the following lemmas and theorems can be found in our full technical report [35].

Lemma 1. *Given time intervals $[ts, te]$ and $[ts', te']$ such that $[ts, te] \subset [ts', te']$, we have $\mathcal{T}^k_{[ts,te]}$ is a subgraph of $\mathcal{T}^k_{[ts',te']}$.*

Theorem 1. *Given a time interval $[ts, te]$ and a temporal k-core $\mathcal{T}^k_{[ts',te']}$ with $[ts, te] \subset [ts', te']$, the subgraph induced by using TCD operation from $\mathcal{T}^k_{[ts',te']}$ for $[ts, te]$ is $\mathcal{T}^k_{[ts,te]}$.*

For example, Figure 2 illustrates the procedure of TCD from $\mathcal{T}^2_{[2,6]}$ to $\mathcal{T}^2_{[5,6]}$ on our running example graph in Figure 1. The edges with timestamps not in $[5, 6]$ (marked by dashed lines) are firstly removed from $\mathcal{T}^2_{[2,6]}$ by truncation, which results in the decrease of degrees of vertices v_5, v_7 and v_8. Then, the vertices with degree less than 2 (marked by dark circles), namely, v_7 and v_8 are further peeled by decomposition, together with their edges. The remaining temporal graph is $\mathcal{T}^2_{[5,6]}$.

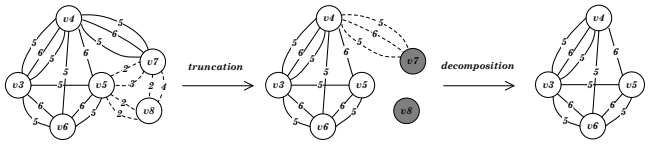


Figure 2: Temporal core decomposition from $\mathcal{T}^2_{[2,6]}$ to $\mathcal{T}^2_{[5,6]}$.

Algorithm 2: TCD algorithm.

Input: $\mathcal{G}$, k, $[Ts, Te]$
Output: all distinct $\mathcal{T}^k_{[ts,te]}$ with $[ts, te] \subseteq [Ts, Te]$

```
for ts ← Ts to Te do          // anchor a new start time
    te ← Te                   // reset the end time
    if ts = Ts then
        T^k_[ts,te] ← TCD(G_[Ts,Te], k, [ts, te])
    else
        T^k_[ts,te] ← TCD(T^k_[ts−1,te], k, [ts, te])
    collect T^k_[ts,te] if it is distinct
    for te ← Te − 1 to ts do          // iterative TCD
        T^k_[ts,te] ← TCD(T^k_[ts,te+1], k, [ts, te])
        collect T^k_[ts,te] if it is distinct
```

3.2 TCD Algorithm

We propose a TCD algorithm to address TCQ by using the above TCD operation. In general, given a TCQ instance, the TCD algorithm enumerates each subinterval of $[Ts, Te]$ in a particular order, so that the temporal k-cores of each subinterval are induced decrementally from previously induced temporal k-cores except the initial one.

Specifically, we enumerate a subinterval $[ts, te]$ of $[Ts, Te]$ as follows. Initially, let $ts = Ts$ and $te = Te$. It means we induce the largest temporal k-core $\mathcal{T}^k_{[Ts,Te]}$ at the beginning. Then, we will anchor the start time $ts = Ts$ and decrease the end time te from Te until ts gradually. As a result, we can always leverage TCD to induce the temporal k-core of current subinterval $[ts, te]$ from the previously induced temporal k-core of $[ts, te + 1]$ but not from $\mathcal{G}_{[ts,te]}$ or even $\mathcal{G}$. Whenever the value of te is decreased to ts, the value of ts will be increased to $ts + 1$ until $ts = Te$, and the value of te will be reset to Te. Then, we induce $\mathcal{T}^k_{[ts+1,te]}$ from $\mathcal{T}^k_{[ts,te]}$, and start over the decremental TCD procedure. The pseudo code of TCD algorithm is given in Algorithm 2. Note that, the details of TCD($\mathcal{G}$, k, $[ts, te]$) function is left to Section 5.2, in which we design a specific data structure to implement TCD operation efficiently in physical level.

Figure 3 gives a demonstration of TCD algorithm for finding temporal 2-cores of time interval [1,8] on our running example graph. The temporal k-cores are induced line by line and from left to right. Each arrow between temporal k-cores represents a TCD operation from tail to head. We can see that, compared with

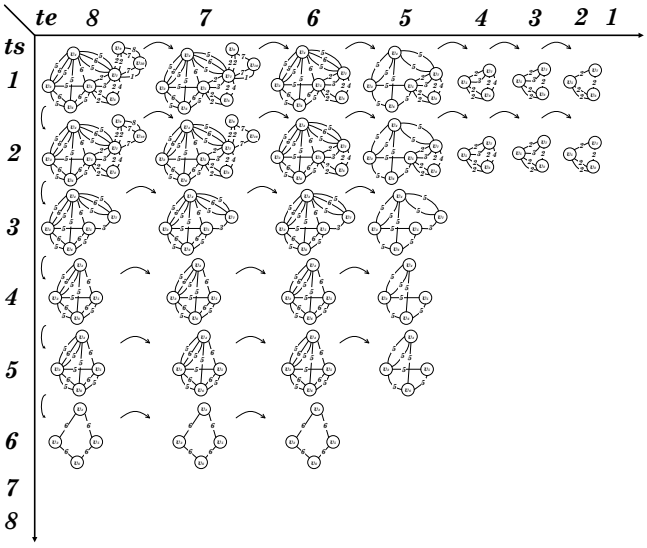


Figure 3: A demonstration of TCD algorithm for finding temporal 2-cores of time interval [1,8].

inducing each temporal k-core independently, the TCD algorithm reduces the computational overhead significantly. For most induced temporal k-cores, a number of vertices and edges have already been excluded while inducing the previous temporal k-cores. Moreover, with the increase of ts and the decrease of te when ts is fixed, the size of $\mathcal{T}^k_{[ts,te]}$ will be reduced monotonically until no temporal k-core exists over $[ts, te]$, so that the time and space costs of TCD operation will also be reduced gradually.

Lastly, we compare TCD algorithm with baseline algorithm. When ts is fixed, Baseline algorithm conducts an incremental procedure, in which each vertex is popped once and each edge may be popped and pushed back many times, and in contrast, TCD algorithm conducts a decremental procedure, in which each vertex is peeled once and each edge is also removed once due to Lemma 1. Therefore, TCD algorithm that is well implemented in physical level (see Section 5.2) can be even more efficient than Baseline algorithm, though it does not need any precomputed index.

4 OPTIMIZATION

In this section, we dive deeply into the procedure of TCD algorithm and optimize it dramatically by introducing an intuitive concept called *tightest time interval* for temporal k-cores. In a nutshell, we directly prune subintervals without inducing their temporal k-cores if we can predict that the temporal k-cores are identical to other induced temporal k-cores, and tightest time interval is the key to fulfill prediction. In this way, the optimized TCD algorithm only performs TCD operations that are necessary for returning all distinct answers to a given TCQ instance. Conceptually, the new pruning operation of optimized algorithm eliminates the "inter-core" redundant computation, and the original TCD operation eliminates the "intra-core" redundant computation. Thus, the computational complexity of optimized algorithm no longer depends on the span of query time interval $[Ts, Te]$ like the baseline algorithm and the original TCD algorithm but only depends on the scale of final results.

4.1 Tightest Time Interval (TTI)

We have such an observation, a temporal k-core of $[ts, te]$ may only contain edges with timestamps in a subinterval $[ts', te'] \subset [ts, te]$, since the edges in $[ts, ts')$ and $(te', te]$ have been removed by core decomposition. For example, consider a temporal k-core $\mathcal{T}^2_{[4,8]}$ illustrated in Figure 3. We can see that it does not contain edges with timestamps 4, 7 and 8. As a result, if we continue to induce $\mathcal{T}^2_{[4,7]}$ from $\mathcal{T}^2_{[4,8]}$ and to induce $\mathcal{T}^2_{[4,6]}$ from $\mathcal{T}^2_{[4,7]}$, the returned temporal k-cores remain unchanged. The sameness of temporal k-cores induced by different subintervals inspires us to further optimize TCD algorithm by pruning subintervals directly. As illustrated in Figure 3, the subintervals such as [4,7], [4,6], [5,8], [5,7] and [5,6] all induce the identical temporal k-cores to [4,8], so that they can be potentially pruned in advance.

For that, we propose the concept of Tightest Time Interval (TTI) for temporal k-cores. Given a temporal k-core of $[ts, te]$, its TTI refers to the minimal time interval $[ts', te']$ that can induce an identical temporal k-core to $\mathcal{T}^k_{[ts,te]}$, namely, there is no subinterval of $[ts', te']$ that can induce an identical temporal k-core to $\mathcal{T}^k_{[ts,te]}$. We formalize the definition of TTI as follows.

Definition 3 (Tightest Time Interval). *Given a temporal k-core $\mathcal{T}^k_{[ts,te]}$, its tightest time interval $\mathcal{T}^k_{[ts,te]}$.TTI is $[ts', te']$, if and only if*
1) $\mathcal{T}^k_{[ts',te']}$ is an identical temporal k-core to $\mathcal{T}^k_{[ts,te]}$;
2) there does not exist $[ts'', te''] \subset [ts', te']$, such that $\mathcal{T}^k_{[ts'',te'']}$ is an identical temporal k-core to $\mathcal{T}^k_{[ts,te]}$.

It is easy to prove the TTI of a temporal k-core of $[ts, te]$ is surely a subinterval of $[ts, te]$. To evaluate the TTI of a given $\mathcal{T}^k_{[ts,te]}$, we have the following theorem.

Theorem 2. *Given a temporal k-core $\mathcal{T}^k_{[ts,te]}$, $\mathcal{T}^k_{[ts,te]}$.TTI = $[t_{min}, t_{max}]$, where t_{min} and t_{max} are the minimum and maximum timestamps in $\mathcal{T}^k_{[ts,te]}$ respectively.*

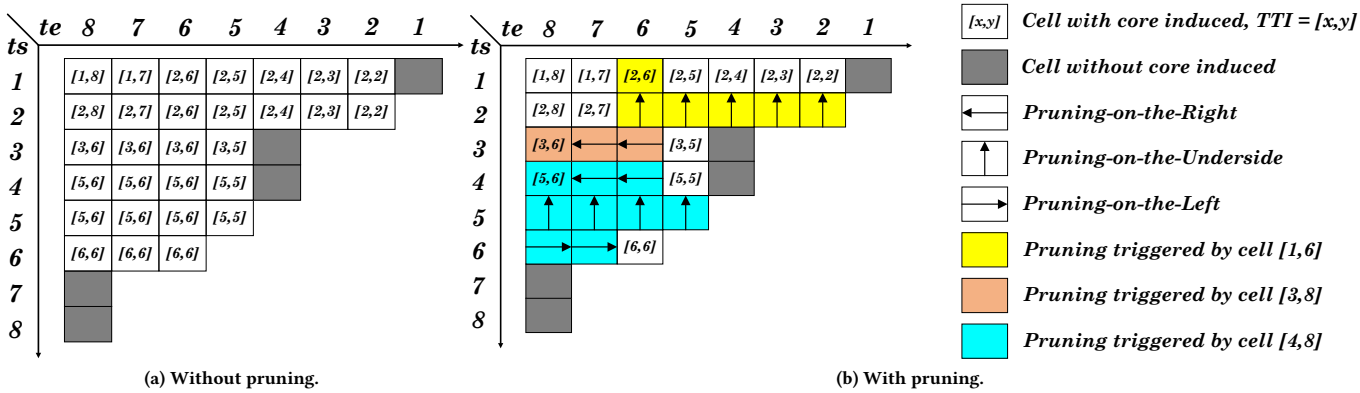
With Theorem 2, we can evaluate the TTI of a given temporal k-core instantly (by $O(1)$ time, see Section 5), which guarantees the following optimization based on TTI will not incur extra overheads.

Moreover, there are the following important properties of TTI that support our pruning strategies.

Property 1 (Uniqueness). *Given a temporal k-core $\mathcal{T}^k_{[ts,te]}$, there exists no other time interval than $\mathcal{T}^k_{[ts,te]}$.TTI evaluated by Theorem 2 that is also a TTI of $\mathcal{T}^k_{[ts,te]}$.*

Property 2 (Equivalence). *Given two temporal k-cores $\mathcal{T}^k_{[ts,te]}$ and $\mathcal{T}^k_{[ts',te']}$, they are identical temporal graphs iff $\mathcal{T}^k_{[ts,te]}$.TTI = $\mathcal{T}^k_{[ts',te']}$.TTI.*

Property 3 (Inclusion). *Given two temporal k-cores $\mathcal{T}^k_{[ts,te]}$ and $\mathcal{T}^k_{[ts',te']}$, we have $\mathcal{T}^k_{[ts,te]}$.TTI $\subseteq \mathcal{T}^k_{[ts',te']}$.TTI, if $[ts, te] \subseteq [ts', te']$.*



(a) Without pruning. (b) With pruning.

Figure 4: Examples of subinterval pruning based on tightest time interval.

Figure 4a abstracts Figure 3 as a schedule table of subinterval enumeration, and TCD algorithm will traverse the cells row by row and from left to right. For example, the cell in row 1 and column 6 represents a subinterval $[1, 6]$, in which $[2, 6]$ is the TTI of $\mathcal{T}^2_{[1,6]}$. In particular, the grey cells indicate that the temporal k-cores of the corresponding subintervals do not exist. Figure 4a clearly reveals that TCD algorithm suffers from inducing a number of identical temporal k-cores (with the same TTIs). For example, the TTI $[5, 6]$ repeats six times, which means six cells will induce identical temporal k-cores.

4.2 Pruning Rules

The main idea of optimizing TCD algorithm is to predict the induction of identical temporal k-cores by leveraging TTI, thereby skipping the corresponding subintervals during the enumeration. Specifically, whenever a temporal k-core of $[ts, te]$ is induced, we evaluate its TTI as $[ts', te']$. If $ts' > ts$ or/and $te' < te$, it is triggered that a number of subintervals on the schedule can be pruned in advance. According to different relations between $[ts, te]$ and $[ts', te']$, our pruning technique can be categorized into three rules which are not mutually exclusive. In other words, the three rules may be triggered at the same time, and prune different subintervals respectively.

4.2.1 Rule 1: Pruning-on-the-Right. Consider the schedule illustrated in Figure 4a. For each row, TCD algorithm traverses the cells (namely, subintervals) from left to right. If the TTI $[ts', te']$ in the current cell $[ts, te]$ meets such a condition, namely, $te' < te$, a pruning operation will be triggered, and the following cells in this row from $[ts, te - 1]$ until $[ts, te']$ will be skipped because these subintervals will induce identical temporal k-cores to $\mathcal{T}^k_{[ts,te]}$. Since the pruned cells are on the right of trigger cell, we call this rule Pruning-on-the-Right (PoR). The pseudo code of PoR is given in lines 2-4 of Algorithm 3. The correctness of PoR is guaranteed by the following lemma.

Lemma 2. *Given a temporal k-core $\mathcal{T}^k_{[ts,te]}$ whose TTI is $[ts', te']$, for any time interval $[ts, te'']$ with $te'' \in [te', te]$, $\mathcal{T}^k_{[ts,te'']}$.TTI = $[ts', te']$.*

With Lemma 2, we can predict that the TTIs in the cells $[ts, te-1]$, $\cdots$, $[ts, te']$ are the same as the trigger cell $[ts, te]$, when the PoR rule is satisfied. Thus, the temporal k-cores induced by these subintervals are all identical to $\mathcal{T}^k_{[ts,te]}$ due to Property 2 (Equivalence).

For example, Figure 4b illustrates two instances of PoR (the cells in orange and blue colors with left arrow). When $\mathcal{T}^2_{[3,8]}$ has been induced, we evaluate its TTI as $[3, 6]$, and thus PoR is triggered. PoR immediately excludes the following two cells $[3, 7]$ and $[3, 6]$ from the schedule. As a proof, we can see the TTIs in these two cells are both $[3, 6]$ in Figure 4a.

4.2.2 Rule 2: Pruning-on-the-Underside. We now consider $ts' > ts$, which causes pruning in the following rows but not the current row. So we call this rule Pruning-On-the-Underside (PoU). Specifically, if $ts' > ts$, for each row $r \in [ts + 1, ts']$, the cells $[r, te]$, $[r, te - 1]$, $\cdots$, $[r, r]$ will be skipped. The pseudo code of PoU is given in lines 5-8 of Algorithm 3. The correctness of PoU is guaranteed by the following lemmas.

Lemma 3. *Given a temporal k-core $\mathcal{T}^k_{[ts,te]}$ whose TTI is $[ts', te']$, for any time interval $[ts'', te]$ with $ts'' \in [ts, ts']$, we have the TTI of $\mathcal{T}^k_{[ts'',te]}$ is $[ts', te']$.*

Lemma 4. *Given a temporal k-core $\mathcal{T}^k_{[ts,te]}$ whose TTI is $[ts', te']$, for any time interval $[r, c]$ with $r \in [ts + 1, ts']$ and $c \in [ts, te]$, we have $\mathcal{T}^k_{[r,c]}$ is identical to $\mathcal{T}^k_{[ts,c]}$.*

Lemma 4 indicates that, PoU safely prunes some cells in the following rows, since these cells contain the same TTIs as their upper cells, which even have not been enumerated yet except the trigger cell. For example, Figure 4b illustrates two PoU instances (the cells in yellow and blue colors with up arrow). On enumerating the cell $[1, 6]$, since the contained TTI is $[2, 6]$, the cells $[2, 6]$, $\cdots$, $[2, 2]$ are pruned by PoU, because the TTIs in these cells are the same as the cells $[1, 6]$, $\cdots$, $[1, 2]$ respectively, though the TTIs of cells $[1, 5]$, $\cdots$, $[1, 2]$ have not been evaluated.

4.2.3 Rule 3: Pruning-on-the-Left. Lastly, if both $ts' > ts$ and $te' < te$, for each row $r \in [ts' + 1, te']$, the cells $[r, te]$, $[r, te - 1]$, $\cdots$, $[r, te' + 1]$ will also be skipped, besides the cells pruned by PoR and

Algorithm 3: Pruning operation.

Input: $[ts, te]$ and $\mathcal{T}^k_{[ts,te]}$

```
1  [ts', te'] ← T^k_[ts,te].TTI // Theorem 2
2  if te' < te then                                  // Rule 1: PoR
3      for c ← te - 1 to te' do
4          prune the subinterval [ts, c]
5  if ts' > ts then                                  // Rule 2: PoU
6      for r ← ts + 1 to ts' do
7          for c ← te to r do
8              prune the subinterval [r, c]
9  if ts' > ts and te' < te then                     // Rule 3: PoL
10     for r ← ts'+1 to te' do
11         for c ← te to te'+1 do
12             prune the subinterval [r, c]
```

PoU. Although these cells are in the rows under the current row ts, the temporal k-core of each of them is identical to the temporal k-core of a cell (namely, $[r, te']$) on the right in the same row but not its upper cell like PoU. So we call this rule Pruning-On-the-Left (PoL). The pseudo code of PoL is given in lines 9-12 of Algorithm 3. The correctness of PoL is guaranteed by the following lemma.

Lemma 5. *Given a temporal k-core $\mathcal{T}^k_{[ts,te]}$ whose TTI is $[ts', te']$, for any time interval $[r, c]$ with $r \in [ts'+1, te']$ and $c \in [te'+1, te]$, we have $\mathcal{T}^k_{[r,c]}$ is identical to $\mathcal{T}^k_{[r,te']}$.*

For example, Figure 4b illustrates a PoL instance (the cells in blue color with right arrow). On enumerating the cell $[4, 8]$, PoL is triggered since the contained TTI is $[5, 6]$. Then, the cells $[6, 8]$ and $[6, 7]$ are pruned by PoL because the TTIs contained in them are the same as the cell $[6, 6]$ on the right of them. PoL is more tricky than PoU because the cells are pruned for containing the same TTIs as other cells that are scheduled to traverse after them by TCD algorithm. Note that, the cell $[4, 8]$ triggers all three kinds of pruning. In fact, a cell may trigger PoL only, PoU only, or all three rules.

4.3 Optimized TCD Algorithm

Compared with TCD algorithm, the improvement of Optimized TCD (OTCD) algorithm is simply to conduct a pruning operation whenever a temporal k-core has been induced. Specifically, we evaluate the TTI of this temporal k-core, check each pruning rule to determine if it is triggered, and prune the specific subintervals on the schedule in advance. The pseudo code of pruning operation is given in Algorithm 3. Note that, the "prune" in Algorithm 3 is a logical concept, and can have different physical implementations.

As illustrated in Figure 4b, OTCD algorithm completely eliminates repeated inducing of identical temporal k-cores, namely, each distinct temporal k-core is induced exactly once during the whole procedure. It means, the real computational complexity of OTCD algorithm is the summation of complexity for inducing each distinct temporal k-core but not the temporal k-core of each subinterval of $[Ts, Te]$. Therefore, we say OTCD algorithm is scalable with respect to the query time interval $[Ts, Te]$. For many real-world datasets, the span of $[Ts, Te]$ could be very large, while there exist only a limited number of distinct temporal k-cores over this period, so that OTCD algorithm can still process the query efficiently.

5 IMPLEMENTATION

In this section, we address the physical implementation of proposed algorithm.

5.1 Temporal Edge List (TEL)

We propose a novel data structure called Temporal Edge List (TEL) for representing an arbitrary temporal graph (including temporal k-cores that are also temporal graphs), which is both the input and output of TCD operation. Conceptually, TEL($\mathcal{G}$) preserves a temporal graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ by organizing its edges in a 3-dimension space, each dimension of which is a set of bidirectional linked lists, as illustrated in Figure 5. The first dimension is time, namely, all edges in $\mathcal{E}$ are grouped by their timestamps. Each group is stored as a bidirectional linked list called Time List (TL), and TL(t) denotes the list of edges with a timestamp t. Then, TEL($\mathcal{G}$) uses a bidirectional linked list, in which each node represents a timestamp in $\mathcal{G}$, as a timeline in ascending order to link all TLs, so that some temporal operations can be facilitated. Moreover, the other two dimensions are source vertex and destination vertex respectively. We use a container to store the Source Lists (SL) or Destination Lists (DL) for each vertex $v \in \mathcal{V}$, where SL(v) or DL(v) is a bidirectional linked list that links all edges whose source or destination vertex is v. Actually, an SL or DL is an adjacency list of the graph, by which we can retrieve the neighbor vertices and edges of a given vertex efficiently. Given a temporal graph $\mathcal{G}$, TEL($\mathcal{G}$) is built in memory by adding its edges iteratively. For each edge $(u, v, t) \in \mathcal{E}$, it is only stored once, and TL(t), SL(u) and DL(v) will append its pointer at the tail respectively.

Figure 5 illustrates a partial TEL of our example graph. The SLs and DLs other than SL(v_5) and DL(v_3) are omitted for conciseness. Basically, TL, SL and DL offer the functionality of retrieving edges by timestamp and linked vertex respectively. For example, for removing all neighbor edges of a vertex v with degree less than k in TCD operation, we can locate SL(v) and DL(v) to retrieve these edges. Moreover, the linked list of TL can offer efficient temporal operations. For example, for truncating $\mathcal{G}$ to $\mathcal{G}_{[ts,te]}$ in TCD operation, we can remove TL(t) with $t < ts$ or $t > te$ from the linked list of TL conveniently. To get the TTI of a temporal k-core, we only need to check the head and tail nodes of the linked list of TL in its TEL to get the minimum and maximum timestamps respectively.

The superiority of TEL is summarized as follows.

- By TCD operation, a TEL will be trimmed to a smaller TEL, and there is none intermediate TEL produced. Thus, the memory requirement of (O)TCD algorithm only depends on the size of initial TEL($\mathcal{G}_{[Ts,Te]}$).
- TEL consumes $O(|\mathcal{E}|)$ space for storing a temporal graph, which is optimal because at least $O(|\mathcal{E}|)$ space is required for storing a graph (e.g., adjacency lists). Although there are $6|\mathcal{E}| + 2|\mathcal{V}| + 3n$ pointers of TLs, SLs and DLs stored additionally, TEL is still compact compared with PHC-Index, where n is the number of timestamps in the graph.

- TEL supports the basic manipulations listed in Table 1 in constant time, which are cornerstones of implementing our algorithms and optimization techniques.
- For dynamical graphs, when a new edge is coming, TEL simply appends a new node representing the current time at the end of linked list of TL, and then adds this edge as normal. Thus, TEL can also deal with dynamical graphs.

5.2 Implement TCD Operation on TEL

Given a TCQ instance, our algorithm starts to work on a copy of TEL($\mathcal{G}_{[Ts,Te]}$) in memory, which is obtained by truncating TEL($\mathcal{G}$). Then, our algorithm only needs to maintain an instance of TEL($\mathcal{T}^k_{[ts,Te]}$) and another instance of TEL($\mathcal{T}^k_{[ts,te]}$) with $[ts,te] \subseteq [Ts,Te]$ in memory. The first instance is used to induce the first temporal k-core $\mathcal{T}^k_{[ts+1,Te]}$ by TCD for each row in Figure 3. The second instance is used to induce the following temporal k-cores $\mathcal{T}^k_{[ts,te-1]}$ by TCD in each row. Each TCD operation is decomposed to a series of TEL manipulations, and trims the input TEL without producing any intermediate data.

To assist the implementation of TCD operation, our algorithm uses a global data structure $\mathbb{H}_v$ that organizes all vertices in the maintained TEL into a minimum heap ordered by their degree, so that the vertices with less than k neighbors can be retrieved directly. Note that, whenever an edge is deleted from the maintained TEL, $\mathbb{H}_v$ will also be updated due to the possible decrease of vertex degrees. The trivial details of updating $\mathbb{H}_v$ is omitted.

Algorithm 4 gives the implementation of TCD operation on TEL. The algorithm takes as input the TEL of a given graph $\mathcal{G}$, along with the parameters k, ts and te specifying the target temporal k-core $\mathcal{T}^k_{[ts,te]}$. In truncation phase, TEL($\mathcal{G}$) is projected to TEL($\mathcal{G}_{[ts,te]}$) (lines 1-14). Specifically, the linked list of TL is traversed from the head and tail bidirectionally until meeting ts and te respectively. For each node representing the timestamp t traversed, the edges in TL(t) are removed from TEL, and $\mathbb{H}_v$ is updated for each edge removed. In decomposition phase, TEL($\mathcal{G}_{[ts,te]}$) is further transformed to TEL($\mathcal{T}^k_{[ts,te]}$) (lines 15-24). Specifically, the algorithm pops the vertex with the least neighbors from $\mathbb{H}_v$ iteratively until the remaining vertices all have at least k neighbors or the heap is empty. For each popped vertex v, it removes the linked edges of v preserved in SL(v) and DL(v) from TEL respectively and updates $\mathbb{H}_v$ accordingly. In particular, a TL will be removed from the linked list of TL after the last edge in it has been removed (lines 19 and 23).

To clarify the procedure of Algorithm 4, Figure 6 illustrates an example of inducing $\mathcal{T}^2_{[4,5]}$ from $\mathcal{T}^2_{[3,6]}$. The edges are going to be deleted are marked in red color. We can see that, the procedure is actually a stream of edge deletion, while TEL maintains the entries to retrieve the remaining edges.

Lastly, the complexity analysis of TCD and OTCD algorithms can be found in our full technical report [35].

Algorithm 4: TCD operation in Algorithm 2

Input: TEL($\mathcal{G}$), $[ts,te]$, k
Output: TEL($\mathcal{T}^k_{[ts,te]}$)

```
1  TL ← the head of linked list of TL in TEL(G)
2  while TL.timestamp ≠ ts do
3      for edge e in TL do
4          del_edge(e)
5          udpate H_v
6      del_TL(TL)
7      TL ← next_TL(TL)
8  TL ← the tail of linked list of TL in TEL(G)
9  while TL.timestamp ≠ te do
10     for edge e in TL do
11         del_edge(e)
12         udpate H_v
13     del_TL(TL)
14     TL ← prev_TL(TL)
15 while H_v is not empty and H_v.peek < k do
16     vertex v ← H_v.pop()
17     for edge e in SL(v) do
18         del_edge(e)
19         del_TL(TL(e.timestamp)) if the TL is empty
20         update H_v
21     for edge e in DL(v) do
22         del_edge(e)
23         del_TL(TL(e.timestamp)) if the TL is empty
24         update H_v
```

6 EXTENSION

To demonstrate the wide applicability of our approach in practice, we present several typical scenarios that extends the data model or query model of TCQ, and sketch how to address them based on our data structure and algorithm in this section.

6.1 Data Model Extension

Dynamical Graph. Since most real-world graphs are evolving over time, it is significant to fulfill TCQ on dynamical graphs. Benefitted from its design in "timeline" style, our data structure TEL can deal with new edges naturally in memory through two new manipulations add_TL(t) and add_edge(u, v, t). When a new edge (u,v,t) arrived, we firstly create an empty TL(t), and append it at the end of the linked list of TL since t is obviously greater than the existing timestamps. Then, we create a new edge node for (u,v,t) and append it to TL(t), SL(u) and DL(v) respectively. Both manipulations are finished in constant time. The maintenance of a dynamical TEL is actually consistent with the construction of a static TEL. Therefore, our (O)TCD algorithm can run on the dynamical TEL anytime.

In contrast, updating PHC-Index is a non-trivial process. Although there are previous work [20, 29] on coreness updating for dynamical graphs, the update is only valid for the whole life time of graph. While, for an arbitrary start time, it is uncertain whether the coreness of a vertex will be changed by a new edge.

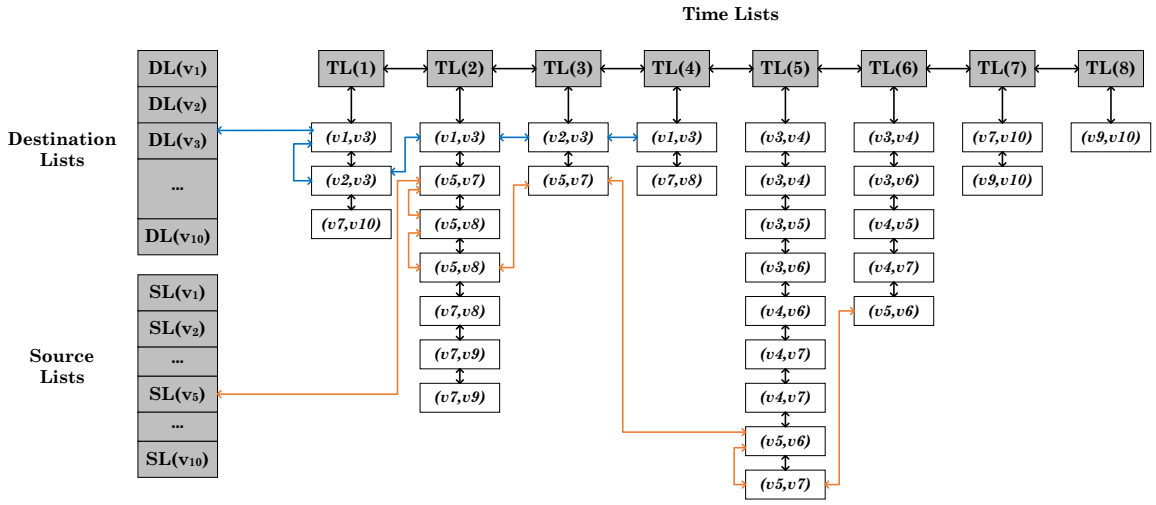


Figure 5: The conceptual illustration of a partial TEL of our running example graph.

Table 1: The basic manipulations of TEL.

Name	Functionality	Complexity
next_TL(TL) / prev_TL(TL)	get the next or previous TL in the linked list of TL	$O(1)$
get_SL(v) / get_DL(v)	get the SL or DL of a given vertex v from a hash map	$O(1)$
del_TL(TL)	remove the given TL node from the linked list of TL	$O(1)$
del_edge(e)	delete a given edge $e = (u, v, t)$ and update TL(t), SL(u) and DL(v) respectively	$O(1)$
get_TTI()	return the timestamps of head and tail nodes of linked list of TL	$O(1)$

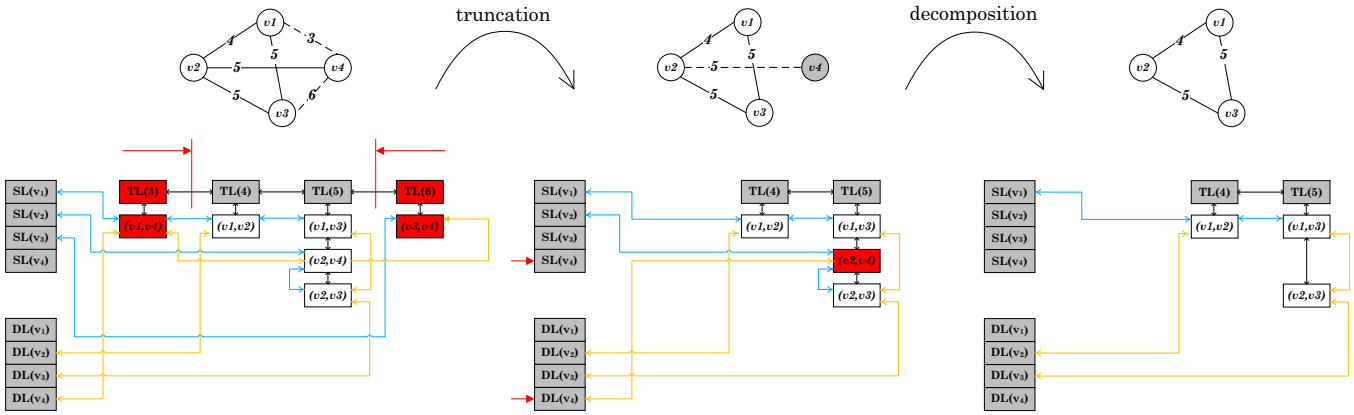


Figure 6: An example of TCD operation on TEL.

6.2 Query Model Extension

The existing graph mining tasks regarding k-core introduce various constraints. For temporal graphs, we only focus on the temporal constraints. In the followings, we present two of them that can be integrated into TCQ model and also be addressed by our algorithm directly, which demonstrate the generality of our model and algorithm.

Link Strength Constraint. In the context of temporal graph, link strength usually refers to the number of parallel edges between a pair of linked vertices. Obviously, the minimum link strength in a temporal k-core represents some important properties like validity, since noise interaction may appear over time and a pair of vertices with low link strength may only have occasional interaction during the time interval. Actually, the previous work [34] has studied this problem without the time interval constraint. Therefore, it is reasonable to extend TCQ to retrieve k-cores with a lower bound of link strength during a given time interval. It can be achieved by trivially modifying the TCD Operation. Specifically, the modified TCD Operation will remove the edges between two vertices once the number of parallel edges between them is decreased to be lower

Table 2: Datasets.

Name	$\|\mathcal{V}\|$	$\|\mathcal{E}\|$	Span(days)
Youtube	3.2M	9.4M	226
DBLP	1.8M	29.5M	17532
Flickr	2.3M	33M	198
CollegeMsg	1.8K	20K	193
email-Eu-core-temporal	0.9K	332K	803
sx-mathoverflow	24.8K	506K	2350
sx-stackoverflow	2.6M	63.5M	2774

Table 3: Pruning effect.

id	Triggered Times			Pruned Cell Percentage (%)			
	PoR	PoU	PoL	PoR	PoU	PoL	Total
1	54	72	2	0.02	72	23.6	95.62
6	2	4	1	0.01	51.8	32.1	83.91
11	8	10	1	0.04	57.1	24.5	81.64
16	5	9	1	0.04	56.9	33.5	90.44

than the given lower bound of link strength, while the original TCD operation will do this when the number becomes zero. Thus, the modification brings almost none extra time and space consumption.

Time Span Constraint. In many cases, we prefer to retrieve temporal k-cores with a short time span (between their earliest and latest timestamps), which is similar to the previous work on density-bursting subgraphs [5]. Because such a kind of short-term cohesive subgraphs tend to represent the occurrence of some special events. TCQ can be conveniently extended for resolving the problem by specifying a constraint of time span. Since the time span of a temporal k-core is preserved in its TEL, which is actually the length of its TTI, we can abandon the temporal k-cores returned by TCD operation that cannot satisfy the time span constraint on the fly. It brings almost no extra time and space consumption. Moreover, we can also extend TCQ to find the temporal k-core with the shortest or top-n shortest time span.

7 EXPERIMENT

In this section, we conduct experiments to verify both efficiency and effectiveness of the proposed algorithm on a Windows machine with Intel Core i7 2.20GHz CPU and 64GB RAM. The algorithms are implemented through C++ Standard Template Library.

7.1 Dataset

We choose seven temporal graphs with different sizes and domains for our experiments. The first three graphs are obtained from KONECT Project [16], and the other four graphs are obtained from the SNAP [17]. The basic statistics of these graphs are given in Table 2. All timestamps are unified to integers in seconds.

7.2 Efficiency

To evaluate the efficiency of our algorithm, we firstly manually select twenty temporal k-core queries from tested random queries with a time span (namely, $Te - Ts$) of 1-3 days, which have been verified to be valid, namely, there is at least one temporal k-core

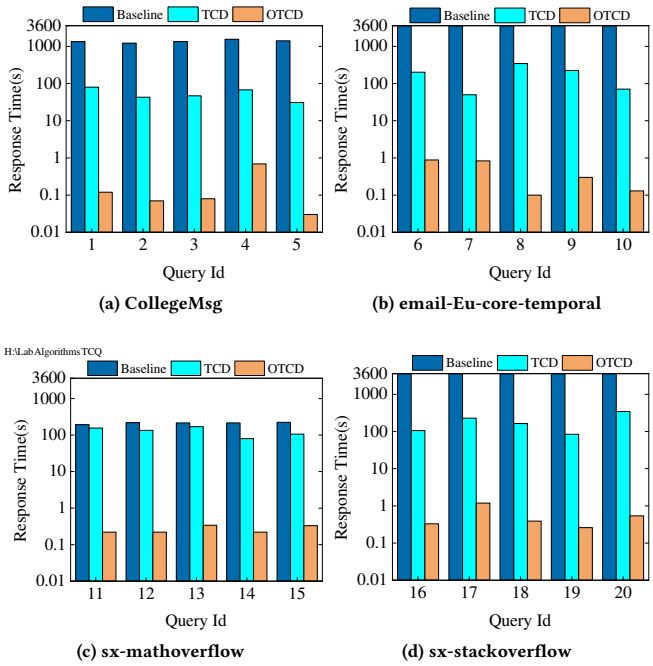


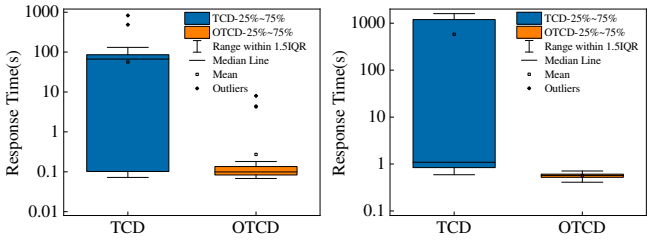
Figure 7: The comparison of response time for selected queries on SNAP graphs.

returned for each query. The time span is moderate, otherwise other algorithms than OTCD can hardly stop successfully. The details of queries can be found in our full technical report [35].

Figure 7 compares the response time of Baseline (iPHC-Query), TCD and OTCD algorithms for each selected query respectively, which clearly demonstrates the efficiency of our algorithm. Firstly, TCD performs better than baseline for all twenty queries due to the physical efficiency of TEL, though they both decrementally or incrementally induce temporal k-cores. Specifically, TCD spends around 100 sec for each query. In contrast, baseline spends more than 1000 sec on CollegeMsg and even cannot finish within an hour on two other graphs, though it uses a precomputed index. Furthermore, OTCD runs two or three orders of magnitude faster than TCD, and only spends about 0.1-1 sec for each query, which verifies the effectiveness of our pruning method based on TTI.

To compare the effect of three pruning rules in OTCD algorithm, Table 3 lists their triggered times and the percentage of subintervals pruned by them for several queries respectively. PoR and PoU are triggered frequently because their conditions are more easily to be satisfied. However, PoR actually contributes pruned subintervals much less than the other two. Because it only prunes subintervals in the same row, and in contrast, PoU and PoL can prune an "area" of subintervals. Overall, the three pruning rules can achieve significant optimization effect together by enabling OTCD algorithm to skip more than 80 percents of subintervals.

To evaluate the stability of our approach, we conduct statistical analysis of one hundred valid random queries on two new graphs, namely, Youtube and Flickr. We visualize the distribution of response time of TCD and OTCD algorithms for these random



(a) Youtube (b) Flickr

Figure 8: The statistical distribution of response time for random queries on KONECT graphs.

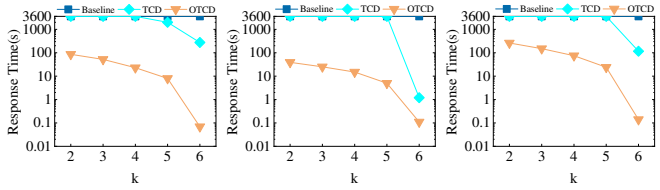
queries as boxplots, which are shown by Figure 8. The boxplots demonstrate that the response time of OTCD varies in a very limited range, which indicates that the OTCD indeed performs stable in practice. The outliers represent some queries that have exceptionally more results, which can be seen as a normal phenomenon in reality. They may reveal that many communities of the social networks are more active during the period.

To verify the scalability of our method with respect to the query parameters, we test the three algorithms with varying minimum degree k and time span (namely, $Te - Ts$) respectively.

Impact of k. We select a typical query with span fixed and k ranging from 2 to 6 for different graphs. The response time of tested algorithms are presented in Figure 9, from which we have an important observation against common sense. That is, different from core decomposition on non-temporal graphs, when the value of k increases, the response time of TCD and OTCD algorithms decreases gradually. For OTCD, the behind rationale is clear, namely, its time cost is only bounded by the scale of results, which decreases sharply with the increase of k. To support the claim, Figure 10 shows the trend of the amount of result cores changing with k. Intuitively, a greater value of k means a stricter constraint and thereby filters out some less cohesive cores. We can see the trend of runtime decrease for OTCD in Figure 9 is almost the same as the trend of core amount decrease in Figure 10, which also confirms the scalability of OTCD algorithm. For TCD, the behind rationale is complicated, since it enumerates all subintervals and each single decomposition is more costly with a greater value of k. It is just like peeling an onion layer by layer, which has less layers with a greater value of k, so that the maintenance between layers become less.

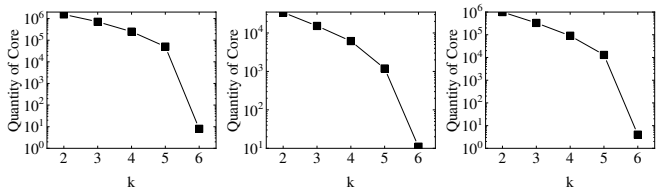
Impact of span. Similarly to the test of k, we also evaluate the scalability of different algorithms when the query time span increases. The results are presented in Figure 11. Although the number of subintervals increases quadratically, the response time of OTCD still increases moderately following the scale of query results. In contrast, TCD runs dramatically slower when the query time span becomes longer.

The above results demonstrate that the efficiency of OTCD is not sensitive to the change of query parameters. We also give a simple and rational criteria for selecting the proper k value on different graphs in our full technical report [35].



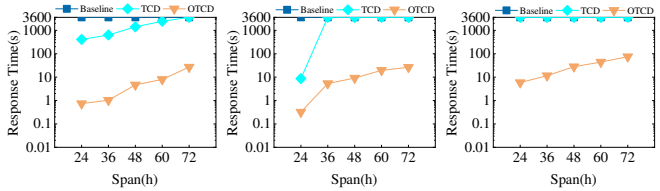
(a) CollegeMsg (b) sx-mathoverflow (c) sx-stackoverflow

Figure 9: Trend of response time under a range of k.



(a) CollegeMsg (b) sx-mathoverflow (c) sx-stackoverflow

Figure 10: Trend of amount of distinct temporal k-cores under a range of k.



(a) CollegeMsg (b) sx-mathoverflow (c) sx-stackoverflow

Figure 11: Trend of response time under a range of span.

Lastly, for a large graph with a long time span like Youtube, we test OTCD algorithm by querying temporal 10-cores over the whole time span. The result is, to find all 19,146 temporal 10-cores within 226 days, the OTCD algorithm spent about 55 minutes, which is acceptable for such a "full graph scan" task.

7.3 Effectiveness

The effectiveness of TCQ is two-fold. Firstly, by given a flexible time interval, we can find many temporal k-cores of different subintervals, each of which represents a community emerged in a specific period. Consider the above test on Youtube. Although it is not feasible to exhibit all 19,146 cores, Figure 12 shows their distribution by time span. The number of cores generally decreases with the increase of time span, which makes sense because there are always a lot of small communities emerged during short periods and then they will interact with each other and be merged to larger communities within a longer time span.

Secondly, we can continue to filter and analyse the result cores to gain insights. For example, we record the date in GMT time for nine of the result cores with a time span less than one day in

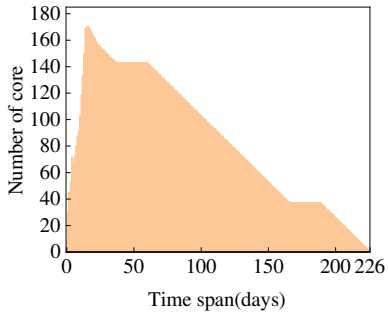


Figure 12: Distribution of all temporal 10-cores in Youtube by time span.

Table 4: The date and size of nine temporal 10-cores emerged within one day in Youtube.

Date	$\|\mathcal{V}\|$	$\|\mathcal{E}\|$
Dec 10 2006	46499	885128
Feb 08 2007	1268	12054
Mar 25 2007	21	139
Jun 15 2007	98	713
Jun 18 2007	20	100
Jun 20 2007	124	1012
Jun 30 2007	21	110
Jul 02 2007	21	110
Jul 06 2007	12	66

Youtube, and try to figure out if they emerged for some special reasons. Table 4 lists the date and size of the nine cores. We can see that there is a large core emerged on Dec 10, 2006, which means more than 40,000 accounts had nearly one million interactions with each other in just a day. That is definitely caused by a special event. While, most of the rest cores emerged during summer vacation, which may mean people have more interactions on Youtube in the period. Two more interesting case studies that reveal the evolution of communities and quickly expanding communities respectively can be found in our full technical report [35].

8 RELATED WORK

Recently, a variety of k-core query problems have been studied on temporal graphs, which involve different temporal objectives or constraints in addition to cohesiveness. The most relevant work to ours is historical k-core query [37], which gives a fixed time interval as query condition. In contrast, our temporal k-core query flexibly find cores of all subintervals. Moreover, Galimberti et al [12] proposed the span-core query, which also gives a time interval as query condition. However, the span-core requires all edges to appear in every moment within the query interval, which is too strict in practice. Actually, historical k-core relaxes span-core, and temporal k-core further relaxes historical k-core.

Besides, there are the following related work. Wu et al [34] proposed (k, h)-core and studied its decomposition algorithm, which gives an additional constraint on the number of parallel edges between each pair of linked vertices in the k-core, namely, they should have at least h parallel edges. Li et al [19] proposed the persistent community search problem and gives a complicated instance called (θ, τ)-persistent k-core, which is a k-core persists over a time interval whose span is decided by the parameters. Similarly, Li et al [21] proposed the continual cohesive subgraph search problem. Chu et al [5] studied the problem of finding the subgraphs whose density accumulates at the fastest speed, namely, the subgraphs with bursting density. Qin et al [27, 28] proposed the periodic community problem to reveal frequently happening patterns of social interactions, such as periodic k-core. Wen et al [1] relaxed the constraints of (k, h)-core and proposed quasi-(k, h)-core for better support of maintenance. Lastly, Ma et al [25] studied the problem of finding dense subgraph on weighted temporal graph. These works all focus on some specific patterns of cohesive substructure on temporal graphs, and propose sophisticated models and methods. Compared with them, our work addresses a fundamental querying problem, which finds the most general k-cores on temporal graphs with respect to two basic conditions, namely, k and time interval. As discussed in Section 6.2, we can extend TCQ to find the more specific k-cores by importing the constraints defined by them, because most of the definitions are special cases of temporal k-core, but not vice versa.

Lastly, many research work on cohesive subgraph query for non-temporal graphs also inspire our work. We categorize them by the types of graphs as follows: undirected graph [3, 9, 13, 23, 36, 38], directed graph [4, 24, 30], labeled graph [6, 18, 31], attributed graph [7, 14, 15, 26], spatial graph [8, 10, 40], heterogeneous information network [11]. Besides, many work specific to bipartite graph [22, 32, 33, 39] also contain valuable insights.

9 CONCLUSION

For querying communities like k-cores on temporal graphs, specifying a time interval in which the communities emerge is the most fundamental query condition. To the best knowledge we have, we are the first to study a temporal k-core query that allows the users to give a flexible interval and returns all distinct k-cores emerging in any subintervals. Dealing with such a query in brute force is obviously costly due to the possibly large number of subintervals. Thus, we propose a novel decremental k-core inducing algorithm and the auxiliary optimization and implementation methods. Our algorithm only enumerates the necessary subintervals that can induce a final result and reduces redundant computation between subintervals significantly. Moreover, the algorithm is physically decomposed to a series of efficient data structure manipulations. As a result, although our algorithm does not use any precomputed index, it still outperforms an incremental version of the latest index-based approach by a remarkable margin. In conclusion, our algorithm is scalable with respect to the span of given time interval.

ACKNOWLEDGMENTS

This work was supported by the grants of the National Natural Science Foundation of China (No. 61202036, 62272353 and 62276193), the Guangzhou Key Laboratory of Big Data and Intelligent Education (No. 201905010009), and the Research Grants Council of Hong Kong (No. 14203618, No. 14202919 and No. 14205520).

REFERENCES

[1] Wen Bai, Yadi Chen, and Di Wu. 2020. Efficient temporal core maintenance of massive graphs. *Information Sciences* 513 (2020), 324–340.

[2] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O (m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049* (2003).

[3] Francesco Bonchi, Arijit Khan, and Lorenzo Severini. 2019. Distance-generalized core decomposition. In *Proceedings of the 2019 International Conference on Management of Data*. 1006–1023.

[4] Yankai Chen, Jie Zhang, Yixiang Fang, Xin Cao, and Irwin King. 2021. Efficient community search over large directed graphs: An augmented index-based approach. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*. 3544–3550.

[5] Lingyang Chu, Yanyan Zhang, Yu Yang, Lanjun Wang, and Jian Pei. 2019. Online density bursting subgraph detection from temporal graphs. *Proceedings of the VLDB Endowment* 12, 13 (2019), 2353–2365.

[6] Zheng Dong, Xin Huang, Guorui Yuan, Hengshu Zhu, and Hui Xiong. 2021. Butterfly-core community search over labeled graphs. *arXiv preprint arXiv:2105.08628* (2021).

[7] Yixiang Fang, Reynold Cheng, Yankai Chen, Siqiang Luo, and Jiafeng Hu. 2017. Effective and efficient attributed community search. *The VLDB Journal* 26, 6 (2017), 803–828.

[8] Yixiang Fang, Reynold Cheng, Xiaodong Li, Siqiang Luo, and Jiafeng Hu. 2017. Effective community search over large spatial graphs. *Proceedings of the VLDB Endowment* 10, 6 (2017), 709–720.

[9] 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, 1 (2020), 353–392.

[10] Yixiang Fang, Zheng Wang, Reynold Cheng, Xiaodong Li, Siqiang Luo, Jiafeng Hu, and Xiaojun Chen. 2018. On spatial-aware community search. *IEEE Transactions on Knowledge and Data Engineering* 31, 4 (2018), 783–798.

[11] Yixiang Fang, Yixing Yang, Wenjie Zhang, Xuemin Lin, and Xin Cao. 2020. Effective and efficient community search over large heterogeneous information networks. *Proceedings of the VLDB Endowment* 13, 6 (2020), 854–867.

[12] 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*. 107–116.

[13] 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.

[14] Xin Huang and Laks VS Lakshmanan. 2017. Attribute-driven community search. *Proceedings of the VLDB Endowment* 10, 9 (2017), 949–960.

[15] Md Saiful Islam, Mohammed Eunus Ali, Yong-Bin Kang, Timos Sellis, Farhana M Choudhury, and Shamik Roy. 2022. Keyword aware influential community search in large attributed graphs. *Information Systems* 104 (2022), 101914.

[16] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd international conference on world wide web*. 1343–1350.

[17] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[18] Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao. 2015. Influential community search in large networks. *Proceedings of the VLDB Endowment* 8, 5 (2015), 509–520.

[19] Rong-Hua Li, Jiao Su, Lu Qin, Jeffrey Xu Yu, and Qiangqiang Dai. 2018. Persistent community search in temporal networks. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 797–808.

[20] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. 2014. Efficient core maintenance in large dynamic graphs. *IEEE Transactions on Knowledge and Data Engineering* 26, 10 (2014), 2453–2465.

[21] Yuan Li, Jinsheng Liu, Huiqun Zhao, Jing Sun, Yuhai Zhao, and Guoren Wang. 2021. Efficient continual cohesive subgraph search in large temporal graphs. *World Wide Web* 24, 5 (2021), 1483–1509.

[22] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. 2019. Efficient (α, β)-core computation: An index-based approach. In *The World Wide Web Conference*. 1130–1141.

[23] Qing Liu, Xuliang Zhu, Xin Huang, and Jianliang Xu. 2021. Local algorithms for distance-generalized core decomposition over large dynamic graphs. *Proceedings of the VLDB Endowment* 14, 9 (2021), 1531–1543.

[24] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks VS Lakshmanan, Wenjie Zhang, and Xuemin Lin. 2020. Efficient algorithms for densest subgraph discovery on large directed graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1051–1066.

[25] Shuai Ma, Renjun Hu, Luoshu Wang, Xuelian Lin, and Jinpeng Huai. 2019. An efficient approach to finding dense temporal subgraphs. *IEEE Transactions on Knowledge and Data Engineering* 32, 4 (2019), 645–658.

[26] Shohei Matsugu, Hiroaki Shiokawa, and Hiroyuki Kitagawa. 2019. Flexible community search algorithm on attributed graphs. In *Proceedings of the 21st International Conference on Information Integration and Web-based Applications & Services*. 103–109.

[27] Hongchao Qin, Ronghua Li, Ye Yuan, Guoren Wang, Weihua Yang, and Lu Qin. 2020. Periodic communities mining in temporal networks: Concepts and algorithms. *IEEE Transactions on Knowledge and Data Engineering* (2020).

[28] Hongchao Qin, Rong-Hua Li, Guoren Wang, Lu Qin, Yurong Cheng, and Ye Yuan. 2019. Mining periodic cliques in temporal networks. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1130–1141.

[29] 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 (2016), 425–447.

[30] Mauro Sozio and Aristides Gionis. 2010. The community-search problem and how to plan a successful cocktail party. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. 939–948.

[31] Renjie Sun, Chen Chen, Xiaoyang Wang, Ying Zhang, and Xun Wang. 2020. Stable community detection in signed social networks. *IEEE Transactions on Knowledge and Data Engineering* (2020).

[32] Kai Wang, Wenjie Zhang, Xuemin Lin, Ying Zhang, Lu Qin, and Yuting Zhang. 2021. Efficient and effective community search on large-scale bipartite graphs. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 85–96.

[33] Kai Wang, Wenjie Zhang, Ying Zhang, Lu Qin, and Yuting Zhang. 2021. Discovering significant communities on bipartite graphs: An index-based approach. *IEEE Transactions on Knowledge and Data Engineering* (2021).

[34] 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.

[35] Junyong Yang, Ming Zhong, Yuanyuan Zhu, Tieyun Qian, Mengchi Liu, and Jeffery Xu Yu. 2023. Scalable Time-Range k-Core Query on Temporal Graphs. https://arxiv.org/pdf/2301.03770.pdf.

[36] Kai Yao and Lijun Chang. 2021. Efficient size-bounded community search over large networks. *Proceedings of the VLDB Endowment* 14, 8 (2021), 1441–1453.

[37] Michael Yu, Dong Wen, Lu Qin, Ying Zhang, Wenjie Zhang, and Xuemin Lin. 2021. On querying historical k-cores. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2033–2045.

[38] Chen Zhang, Fan Zhang, Wenjie Zhang, Boge Liu, Ying Zhang, Lu Qin, and Xuemin Lin. 2020. Exploring finer granularity within the cores: Efficient (k, p)-core computation. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 181–192.

[39] Yuting Zhang, Kai Wang, Wenjie Zhang, Xuemin Lin, and Ying Zhang. 2021. Pareto-optimal community search on large bipartite graphs. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 2647–2656.

[40] Qijun Zhu, Haibo Hu, Cheng Xu, Jianliang Xu, and Wang-Chien Lee. 2017. Geo-social group queries with minimum acquaintance constraints. *The VLDB Journal* 26, 5 (2017), 709–727.