On Querying Historical K-Cores

Michael Yu
University of New South Wales
Australia
mryu@cse.unsw.edu.au

Dong Wen*
AAII, University of Technology
Sydney
Australia
dong.wen@uts.edu.au

Lu Qin
AAII, University of Technology
Sydney
Australia
lu.qin@uts.edu.au

Ying Zhang
AAII, University of Technology
Sydney
Australia
ying.zhang@uts.edu.au

Wenjie Zhang
University of New South Wales
Australia
zhangw@cse.unsw.edu.au

Xuemin Lin
University of New South Wales
Australia
lxue@cse.unsw.edu.au

ABSTRACT

Many real-world relationships between entities can be modeled as temporal graphs, where each edge is associated with a timestamp or a time interval representing its occurrence. K-core is a fundamental model used to capture cohesive subgraphs in a simple graph and have drawn much research attention over the last decade. Despite widespread research, none of the existing works support the efficient querying of historical k-cores in temporal graphs. In this paper, given an integer k and a time window, we study the problem of computing all k-cores in the graph snapshot over the time window. We propose an index-based solution and several pruning strategies to reduce the index size. We also design a novel algorithm to construct this index, whose running time is linear to the final index size. Lastly, we conducted extensive experiments on several real-world temporal graphs to show the high effectiveness of our index-based solution.

PVLDB Reference Format:
Michael Yu, Dong Wen, Lu Qin, Ying Zhang, Wenjie Zhang, and Xuemin Lin. On Querying Historical K-Cores. PVLDB, 14(11): 2033 - 2045, 2021.
doi:10.14778/3476249.3476260

1 INTRODUCTION

The k-core is a fundamental cohesive subgraph model and has attracted a great amount of research interest over the last decade [7, 16, 23, 32]. Given a graph G, a k-core is a maximal connected subgraph of G in which every vertex has a degree of at least k. The k-core model has a broad spectrum of real-world applications, such as network visualization [2], community detection [8, 20], and system structure analysis [35]. Given an integer k, all k-cores of a graph G can be computed in $O(m)$ time where m represents the number of edges in G. Due to the linear computability, many works use finding k-cores as a means to prune search space before computing their respective complicated models [3, 5, 6, 31].

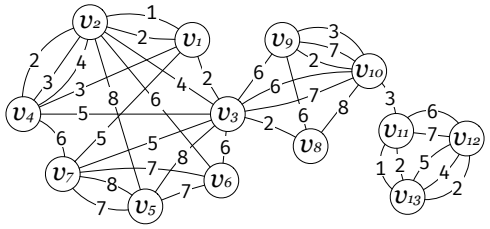


Figure 1: The temporal graph $\mathcal{G}$.

Many real-world graphs evolve over time and are naturally organized as temporal graphs in which each edge is associated with a timestamp. Figure 1 presents an example of a temporal graph. The value on each edge represents the occurring time. Given a time window, we can derive a graph snapshot over the window by taking edges inside the window and merging multiple edges with same endpoints but different timestamps into a single unlabeled edge in the snapshot. Existing works on computing k-cores can be roughly categorized into two types. The first type computes k-cores for all possible values of k (a.k.a. core decomposition) in a simple graph [7, 16, 32], which can be a snapshot of a temporal graph for any specific duration. The second type studies algorithms t update k-cores when the time window changes and the corresponding simple graph updates [21, 26, 36]. Given that graph snapshots may vary considerably from one time window to the other, we naturally wonder *"what was the k-core of the snapshot for a particular period?"*. Unfortunately none of the existing research covers this question. The problem of efficiently answering queries for a graph snapshot of a specific time window is called historical graph queries [17, 18, 25, 28–30]. In this paper, we study the problem of querying historical k-cores. Specifically, given a temporal graph, an arbitrary time window (a start time and an end time) and an integer k, we aim to design an index that allows for the efficient retrieval of all k-cores in the graph snapshot for the query time window.

Application. We provide several applications as follows.

Inherent k-Core Applications. Based on applications of the k-core model, the historical k-core problem can be intrinsically applied to many scenarios. For example, assume that we were to study research communities from different periods of a collaboration network (e.g. *DBLP*, where each edge is associated with a timestamp representing the co-authorship of two researchers). Given that

*DongWen is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476260

the k-core model is often used to capture community structures [10, 19], a straightforward solution is to construct the snapshot and compute k-cores from scratch for every time window concerned. Our indexing techniques in the paper can be used and boost the efficiency of querying historical k-cores.

Building Blocks of Querying Complex Models. There have been several works that integrate the time labels of edges into the k-core model and focus on mining cohesive subgraphs that continuously exist for some time in a temporal graph [11, 20]. In particular, [11] formulates the span-core, which is the k-core in the intersection of snapshots over a set of discrete and contiguous time windows. [20] defines the (θ, τ)-persistent k-core — a temporal graph over a period of τ in which the snapshot over every θ-width window is a k-core. Querying k-cores of a given time window serves as a building block and a foundational operator in the computing process of such temporal variations of the k-core model.

Straightforward Solutions and Challenges. We can easily derive an online solution of historical k-core query based on the existing algorithms for k-cores computation [4, 16]. Specifically, given a time window, we could first locate the edge with start time of the window and derive the snapshot by sequentially scanning the edges till the end time. We then compute k-cores by iteratively removing all vertices with a degree less than k. However, such online solution requires accessing all edges in the window, and the query efficiency is limited, especially when the window is wide.

In this paper, we aim to design an index-based solution to support the historical k-core query for every possible time window and integer k. Existing research normally maintains the k-cores for all k in a simple graph by keeping the core number of each vertex [7, 32]. The core number of a vertex u is the largest integer k such that there exists a k-core containing u. Consequently, the k-core query in a simple graph can be answered by collecting all vertices with core numbers not less than k. We immediately have a straightforward index structure by maintaining the core number of each vertex for every possible time window. However, although the index supports a constant time complexity to identify the core number of each vertex over a specific window, the index takes $O(n \cdot t_{max}^2)$ space where n represents the number of appeared vertices and t_{max} represents the number of distinct times in the graph. The index size is not acceptable given that t_{max} can be very large in practice.

Our Index-based Approach. We propose a novel index called PHC-Index (Pruned Historical Core-Index) to answer historical k-core queries. PHC-Index is designed to support a basic historical k-core containment query, which identifies whether u is in the k-core of the query time window. We answer the historical k-core query by performing the containment query for every vertex u. Section 4.3 also provides several other queries that are extended from the historical k-core containment query.

PHC-Index computes a set of time windows for each possible core number of each vertex. A historical k-core containment query can be answered by comparing the indexed time windows for the integer k with the input query window. We first consider a subproblem where the start time of the query window is given in advance and never changes. The aforementioned naive index takes $O(n \cdot t_{max})$ space for such case, additionally, we observe that the core number of each vertex monotonically decreases when the end time of a window decreases. This observation inspires us to define the *core time* for a vertex u and an integer k, which is the earliest time such that u is contained in a k-core. Given an end time t of a window and an integer k, we know that a vertex u is in the k-core if the corresponding core time of u is earlier than t. Let G denote the snapshot of a whole temporal graph. Based on the concept of core time, we only store $core(u)_G$ core times for each vertex u where $core(u)_G$ is the core number of u in G. Given $core(u)_G$ is bounded by the degree of u in G, we reduce the index size for the subproblem from $O(n \cdot t_{max})$ to $O(m^*)$, where m^* is the number edges in G.

Intuitively, there may exist t_{max} core times for each integer k and vertex u since we have t_{max} possible start times. To reduce the index size, we observe that the core time for a specific k and a vertex monotonically increases when the start time increases. Based on such monotonicity of the core time, we only preserve the core time for a start time that is different from (larger than) that for the previous start time. This strategy significantly reduces the index size. The space complexity is $O(m^* \cdot \bar{t})$, where $\bar{t}$ is the average number of core times for each core number and each vertex. For example, $\bar{t} = 9$ for the real-world dataset *DBLP* in our experiments.

Index Construction. The key step in constructing PHC-Index is to compute the core time. Assume the core number of a vertex u decreases from k of the window $[t_s, t_e]$ to $k-1$ of the window $[t_s, t_e - 1]$. We derive the core time of u for k and t_s is t_e. Therefore, computing core times is equivalent to computing core numbers of all windows. We can invoke the state-of-art algorithm for core number maintenance and derive the core numbers of a window from earlier results. However, even with the help of well-studied core maintenance algorithms, the efficiency of index construction is quite limited due to the vast volume of time windows.

To improve the indexing efficiency, we propose a new algorithm to reduce the t_{max}^2 magnitude of windows scanning to t_{max}. Instead of maintaining core numbers, we iteratively increase the start time and maintain core times of all vertices. Once the start time increases, we design several rules to identify the validity of each core number. When the core time is required to be updated, we also design a new local strategy to compute the core time of a vertex by only scanning all vertices and edges connecting to the vertex. The optimization significantly improve the efficiency of index construction and reduces the time complexity to be linear to the index size if the maximum degree in the graph is regarded as a constant.

Contribution. We summarize main contributions as follows.

- *An index-based solution for historical k-core queries.* We formulate the problem of querying historical k-cores given a temporal graph. As far as we know, the problem has never been studied.
- *An elegant index structure.* We design a novel index structure, called PHC-Index. The index size is bounded by $O(m^* \cdot \bar{t})$, where $\bar{t} \ll t_{max}$ in practice.
- *An optimized index construction algorithm.* We propose a new paradigm to compute and update the core time of each vertex. The running time of the optimized index construction algorithm is bounded by $O(|\mathcal{PHC}| \cdot \mathcal{D}_{max})$, where $|\mathcal{PHC}|$ is the index size and $\mathcal{D}_{max}$ is the maximum vertex degree.
- *Extensive experiments to show the effectiveness of our approach.* We conduct extensive experiments on 8 real-world networks. The

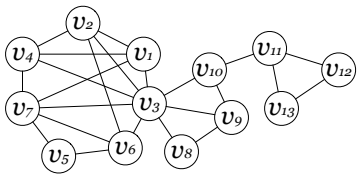


Figure 2: The snapshot of $\mathcal{G}$ over the time window $[2, 7]$

results show the effectiveness of PHC-Index and the efficiency of the index construction algorithm.

2 PRELIMINARIES & RELATED WORK

We study an undirected temporal graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$. Each edge $e \in \mathcal{E}$ is a triplet (u, v, t) representing an interaction between u and v at time t. Edges are organized as a stream order ordered by the occurrence times of edges. The projected graph (snapshot) of $\mathcal{G}$ over a given time window $[t_s, t_e]$, denoted by $G_{[t_s,t_e]}$, is a unlabeled graph where the vertex set $V_{[t_s,t_e]} = \mathcal{V}$ and the edge set $E_{[t_s,t_e]} = \{(u, v)|(u, v, t) \in \mathcal{E}, t \in [t_s, t_e]\}$. Without loss of generality, we assume the earliest edge time is 1, and the latest edge time is t_{max} in $\mathcal{G}$. We use n and m to denote the number of vertices that appeared and the number of temporal edges in the graph. We use m^* to denote the number of edges in the projected graph over $[1, t_{max}]$. Note that m^* can be much smaller than m in practice due to the repeated interactions at different times. The degree of a vertex u in a projected graph is denoted by $deg(u)$. $\mathcal{E}_{t'}$ denotes all edges at time t', i.e., $\mathcal{E}_{t'} = \{(u, v)|(u, v, t) \in \mathcal{E}, t = t'\}$. Given a vertex u and a window $[t_s, t_e]$, $\mathcal{N}(u)_{[t_s,t_e]}$ denotes the neighbors of u over $[t_s, t_e]$, i.e., $\mathcal{N}_{[t_s,t_e]}(u) = \{\langle v, t\rangle|(u, v, t) \in \mathcal{E}, t \in [t_s, t_e]\}$.

Example 2.1. Figure 1 shows a temporal graph, where $t_{max} = 8, m = 36$, and $m^* = 26$. Given a time window $[2, 7]$, the projected graph of $\mathcal{G}$ is given in Figure 2. The isolated vertex v_5 is omitted.

Definition 2.2. (K-Core) Given a graph G and an integer k, the k-core is a maximal connected induced subgraph of G in which all vertices have degree at least k. [27]

Problem 1. Given a temporal graph $\mathcal{G}$, a time window $[t_s, t_e]$ and an integer k, we aim to identify all vertices contained in k-cores of the projected graph of $\mathcal{G}$ over $[t_s, t_e]$.

Example 2.3. The 3-core in Figure 2 is the induced subgraph of $\{v_1, v_2, v_3, v_4, v_6, v_7\}$ in which every vertex has a degree value at least 3. Given a temporal graph $\mathcal{G}$ in Figure 1, $k = 2$ and a time window $[2, 4]$, the historical k-core query returns $\{v_1, v_2, v_3, v_4\}$.

2.1 Core Decomposition

The k-core structures in a graph are usually maintained as core numbers of all vertices [4, 32].

Definition 2.4. (Core Number) Given a graph G and a vertex u, the core number of u, denoted as $core(u)$, is the largest possible k such that u is in a k-core of G.

Example 2.5. Given the simple graph in Figure 2, the core numbers are $core(v_1) = 3$, $core(v_2) = 3$, $core(v_3) = 3$, $core(v_4) = 3$, $core(v_5) = 2$, $core(v_6) = 3$, $core(v_7) = 3$, $core(v_8) = 2$, $core(v_9) = 2$, $core(v_{10}) = 2$, $core(v_{11}) = 2$, $core(v_{12}) = 2$ and $core(v_{13}) = 2$.

Algorithm 1: *CoreDecomposition*$(G(V, E))$

```
1 c ← 0;
2 while V ≠ ∅ do
3     u ← arg min_{v∈V} deg(v);
4     c ← max(deg(u), c);
5     core(u) ← c, V ← V \ u;
6     foreach v ∈ N(u) do deg(v) ← deg(v) − 1 ;
7 return core(u) for all u;
```

All core numbers can be computed by a greedy method which iteratively removes the vertex with the smallest degree [4]. The pseudocode is given in Algorithm 1. The time complexity is $O(|E|)$ by using bin-sort in line 3. [16] provides an efficient implementation of Algorithm 1 using a flat array structure. The algorithm for k-core computation adopts a similar idea. Instead of removing the vertex with the minimum degree, we only iteratively remove all vertices with degree less than k. The algorithm terminates when the degree of every remaining vertex is not less than k.

Core Maintenance. [26] proposes an algorithm to maintain core numbers when new edges are inserted or old edges are deleted in streaming graphs. A similar work is done by [21]. [36] improves the efficiency of core maintenance by arranging vertices in the degeneracy order. [1] and [32] consider core maintenance in distributed environments and external memory, respectively.

Scalable Core Decomposition. [7] proposes an external-memory algorithm for core decomposition, even though the memory size cannot be well bounded. [16, 32] propose algorithms under the semi-external memory setting where the memory size is bounded by $O(n)$. [9] proposes a parallel algorithm, and [23, 24] propose distributed algorithms for core decomposition.

2.2 Other Related Works

Several variations of k-core have been formulated in temporal graphs, which is closely related to our problem. The span-core [11] and the persistent core model [20] have been mentioned in Section 1. The span-core model differs from our setting in the formulation of the graph snapshot for a given time window. Under the span-core model, an edge is in the snapshot if it appears at every time over the query window. The persistent core computes a subgraph which is the k-core for the snapshot of every θ-length sub-window. The snapshot is derived by considering all covered edges which is same as ours. Therefore, given the same time window and the integer k, both the span-core and the persistent core is a subset of our result. [33] also considers k-cores in temporal graphs. However, they omit the timestamps on edges and essentially compute k-cores in a graph with multiple edges between vertices. When computing k-cores, the degree of a vertex is the number of edges connecting to that vertex. k-core has also been studied in other various types of graphs, such as uncertain graphs [34], directed graphs [14], weighted graphs [13], multi-layer graphs [12] and random graphs [15, 22].

3 STRAIGHTFORWARD METHODS

3.1 Online Query Processing

An online solution can be easily derived based on the k-core computation introduced in Section 2. Given a query time window, we

	Query Time	Index Space	Indexing Time
Online	$O(\log m + m_{[t_s,t_e]})$	$\oslash$	$\oslash$
Baseline	$O(\|R\|)$	$O(n \cdot t_{max}^2)$	$O(m \cdot t_{max}^2)$
Ours	$O(n \cdot \log \bar{t})$	$O(m^* \cdot \bar{t})$	$O(m^* \cdot \bar{t} \cdot \mathcal{D}_{max})$

Table 1: Comparison of different solutions , where $m_{[t_s,t_e]}$ is the number of edges in the query window $[t_s, t_e]$, $|R|$ is the result size, $\bar{t}$ is theoretically bounded by t_{max} but much smaller in practice, and $\mathcal{D}_{max}$ is the maximum degree.

can use a binary search to locate the first edge to fall in the window and construct the corresponding projected graph for k-core computation. The time complexity is $O(\log m + m_{[t_s,t_e]})$, where $m_{[t_s,t_e]}$ is the number of edges in $\mathcal{G}$ from t_s to t_e. The online algorithm works but requires visiting the whole projected graph.

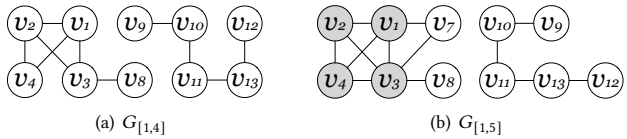
3.2 A Straightforward Index

Baseline. In this paper, we aim to design an index to efficiently answer historical k-core queries for every possible input time window and k. A straightforward index is to maintain core numbers for the projected graph of every window. The index is named HC-Index (Historical Core-Index) and takes $O(nt_{max}^2)$ space, which is extremely large given that t_{max} can be very large in practice. Given a window $[t_s, t_e]$ and an integer k, it takes constant time to identify whether a vertex u is in a k-core by comparing the core number of u over $[t_s, t_e]$ with k. By sorting vertices in non-increasing order of their core numbers, Problem 1 can be answered in optimal time. HC-Index essentially precomputes and stores all possible query results. We provide Table 1 to summarize and compare the online solution, the baseline index-based solution and our final solution.

The Other Potential Idea. Intuitively, another potential idea is to divide edges into a set of discrete and continuous time intervals. We only maintain core numbers for each interval. For example, the real-world dataset *WikiTalk* in our experiments records edges over 2320 days. Assume that we only maintain core numbers of vertices for every calendar month (e.g., 30 days for simplicity), and the index would take about $O(77 \cdot n)$ space where n is the number of vertices in *WikiTalk*. We will show why the idea does not work as follows. First, it is easy to see that we can never use the index to answer the query of an arbitrary window over different indexing intervals, e.g., from the middle of a month to the end of the next month. Second, answering the query of a short-period window in a calendar month may benefit a little from the index since we can first prune all vertices that cannot be a k-core in the graph of the whole month. However, a peeling phase is still required which iteratively removes all other vertices with degrees smaller than k. Third, even if we do not support any fine-granularity intervals and only allow month-to-month queries, the index still does not work for any window over multiple months. For example, assume that we query all k-cores for two consecutive months. Given the core numbers for each month, we only know that a vertex u must be in the result if the core number of u is not smaller than k in any month. This would not help speed up k-core computation since we do not have any vertex that are guaranteed not in the result. Therefore, we still perform the same online algorithm in this case.

4 OUR SOLUTION

In this section, we investigate several characteristics of the historical k-core query and propose a novel index. With a little sacrifice of query efficiency compared with the optimal approach, we reduce the index size from $O(n \cdot t_{max}^2)$ to $O(m^* \cdot \bar{t})$ where $\bar{t}$ is a small integer and $\bar{t} \ll t_{max}$ in practice.



(a) $G_{[1,4]}$ (b) $G_{[1,5]}$

Figure 3: The projected graphs of $[1, 4]$ and $[1, 5]$, where the 3-core is marked in grey.

4.1 Anchoring the Start Time

Our optimized index is called PHC-Index. It records a set of times for each possible k of a vertex. It is designed to support the historical k-core containment query, which is formally defined as follows.

Problem 2. Given a temporal graph $\mathcal{G}$, a time window $[t_s, t_e]$, an integer k and a vertex u, the historical k-core containment query aims to identify whether u is contained in a k-core of the projected graph over $[t_s, t_e]$.

Assuming we answer the historical k-core containment query in $O(f)$ time, then Problem 1 can also be answered in $O(n \cdot f)$ time. We start by considering a sub-problem where the starting time is given in advance and remains unchanged for any query processed, called *start-anchored containment query* for short. Assume that this start time is always t_s. According to the idea of the naive HC-Index, we will maintain the core number of every vertex for the windows $[t_s, t_i]$ from $t_i = t_s$ to $t_i = t_{max}$. However, we observe that the core number of each vertex only monotonically updates and may stay the same over consecutive windows as t_i increases from t_s to t_{max}.

Lemma 4.1. *Given a temporal graph G, a vertex u and two time windows $[t_s, t_e]$, $[t'_s, t'_e]$, we have $core(u)_{[t_s,t_e]} \leq core(u)_{[t'_s,t'_e]}$ if $[t_s, t_e] \subset [t'_s, t'_e]$.*

Proof. The lemma holds since all edges in the projected graph of $[t_s, t_e]$ must exist in that of $[t'_s, t'_e]$. The core number can never decrease when new edges are inserted. □

Based on Lemma 4.1, we reduce the HC-Index by only recording the times at which the core number changes for each vertex.

Definition 4.2. (Core Time) Given a temporal graph $\mathcal{G}$ and a vertex u, the *core time* of u for an integer k and a start time t_s, denoted as $\mathcal{CT}_{t_s}(u)_k$, is the smallest time t_e such that u is in a k-core of the projected graph $G_{[t_s,t_e]}$.

Example 4.3. Given $\mathcal{G}$ in Figure 1, assume that the start time $t_s = 1$ and $k = 3$. We first consider the time window $[1, 5]$ and the corresponding projected graph shown in Figure 3. v_1 is in the 3-core $\{v_1, v_2, v_3, v_4\}$ of $G_{[1,5]}$. However, a 3-core does not exist in the projected graph of $G_{[1,4]}$. Therefore, the core time of v_1 is $\mathcal{CT}_1(v_1)_3 = 5$. Similarly, we have $\mathcal{CT}_1(v_1)_2 = 3$.

Lemma 4.4. *Given a start time t_s, a vertex u and the core time $\mathcal{CT}_{t_s}(u)_k$ of u for the integer k, u is contained in the k-core of the projected graph over $[t_s, t_e]$ if $\mathcal{CT}_{t_s}(u)_k \leq t_e$.*

Given a start time t_s, the core number of u monotonically increases from $core(u)_{[t_s,t_s]}$ to $core(u)_{[t_s,t_{max}]}$ when increasing the end time from t_s to t_{max}. Therefore, we have $O(core(u)_{[t_s,t_{max}]})$ possible core numbers for u and t_s. We record the core time of each vertex for all core numbers. Based on Lemma 4.4, it takes constant time to answer the start-anchored containment query.

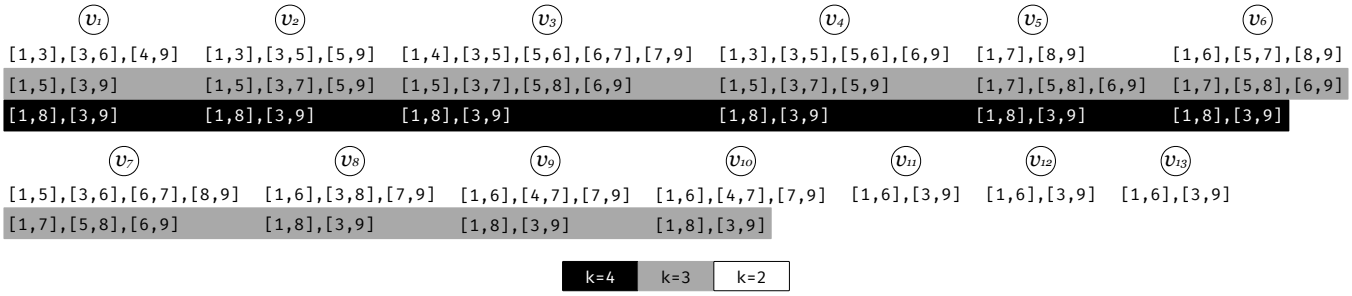


Figure 4: PHC-Index for the temporal graph $\mathcal{G}$

Example 4.5. Given $t_s = 1$ and vertex v_1, we have four possible core times for v_1 since the core number of v_1 in $G_{[1,8]}$ reaches to 4. The core times are $\mathcal{CT}_1(v_1)_1 = 1$, $\mathcal{CT}_1(v_1)_2 = 3$, $\mathcal{CT}_1(v_1)_3 = 5$, and $\mathcal{CT}_1(v_1)_4 = 8$. Given a query time window $[1, 4]$ and $k = 3$, v_1 is not in the 3-core of $G_{[1,4]}$ since $\mathcal{CT}_1(v_1)_3 = 5 > 4$.

Lemma 4.6. *Given a start time t_s, the core times of each vertex u for all possible core numbers take $O(\sum_{u\in V} core(u)_{[t_s,t_{max}]})$ space, where $core(u)_{[t_s,t_{max}]}$ is the core number of u in the projected graph over $[t_s, t_{max}]$.*

Proof. Based on Lemma 4.1, the core number of a vertex u for any window starting form t_s is not larger than $core(u)_{[t_s,t_{max}]}$. Therefore, the lemma holds. □

Given that $core(u) \le deg(u)$, we have $\sum_{u\in V} core(u)_{[t_s,t_{max}]} \le \sum_{u\in V} deg(u)_{[1,t_{max}]} = 2 \cdot m^*$. Therefore, the index space for the start-anchored containment query is also loosely bounded by $O(m^*)$.

4.2 The Index Structure

Given the index for the start-anchored containment query, we have t_{max} possible start times, which intuitively incurs $O(m^* \cdot t_{max})$ index space to support an arbitrary time window $[t_s, t_e]$. To reduce the index size, we further exploit the monotonicity of the core time.

Lemma 4.7. *Given a temporal graph $\mathcal{G}$, an integer k, a vertex u and two start time t_s, t'_s, we have $\mathcal{CT}_{t_s}(u)_k \le \mathcal{CT}_{t'_s}(u)_k$ if $t_s < t'_s$.*

Proof. Given that $t_s < t'_s$, all edges in the projected graph of $[t'_s, \mathcal{CT}_{t_s}(u)_k]$ must exist in that of $[t_s, \mathcal{CT}_{t_s}(u)_k]$. The core number of u over $[t'_s, \mathcal{CT}_{t_s}(u)_k - 1]$ is smaller than k based on Definition 4.2. Therefore, the lemma holds. □

Based on Lemma 4.7, the core time of a vertex u for an integer k never decreases as the start time t_s increases. For each vertex and each possible core number, our index aims to store a set of pairs of a start time and a corresponding core time for query processing. Instead of storing pairs for all start times, we prune unnecessary pairs whose core numbers remains the same when increasing the start time. Our final index is formally defined as follows.

Definition 4.8. Given a temporal graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, for each vertex u and each integer $2 \le k \le core(u)_{[1,t_{max}]}$, the PHC-Index maintains a sequence of windows $\mathcal{PHC}(u)_k$ such that:

(1) for each $[t_s, t_e] \in \mathcal{PHC}(u)_k$, $t_e = \mathcal{CT}_{t_s}(u)_k$;

(2) for each $1 \le t_s \le t_{max}$ and corresponding $\mathcal{CT}_{t_s}(u)_k$, there exists one and only one window $[t'_s, t'_e]$ in $\mathcal{PHC}(u)_k$ such that $t'_e = \mathcal{CT}_{t_s}(u)_k$ and $t'_s \le t_s$.

(3) $t_s^i > t_s^{i-1}$ where t_s^i is the start time of the i-th window;

The first condition guarantees the correctness of each time window. The second condition guarantees the completeness and the minimality for the pairs of start time and core time. The third condition is used to improve the query efficiency, which will be discussed in Section 4.3. Note that we do not need to support the case of $k = 1$ since the historical k-core containment query is equivalent to checking whether there exists an edge containing the query vertex in the time window.

Example 4.9. The PHC-Index for the temporal graph $\mathcal{G}$ in Figure 1 is given in Figure 4. For each vertex, the first line, the second line and the third line show the windows for $k = 2$, $k = 3$ and $k = 4$, respectively. For example, given a vertex v_5 and $k = 2$, the core time for $t_s = 1$ is 7. For the start time $1 < t_s \le 7$, the core times are still 7 and are pruned in the index. When $t_s = 8$, v_5 cannot be any 2-core of the projected graph starting from 8, and we set the core time for 8 as $t_{max} + 1 = 9$.

Theorem 4.10. *The size of PHC-Index is bounded by $O(m^* \cdot \bar{t})$, where $\bar{t}$ is the average number of windows for a specific k of a vertex.*

The theorem can be easily proved based on Lemma 4.6. Theoretically, $\bar{t}$ can be t_{max} in the worst case. Consider a special temporal graph with the same set of edges at every time. We have t_{max} core times for each core number and each vertex in the index. However, in real-world applications, temporal graphs are modeled to capture the change of entities' relationships over time. Such special case is meaningless since it can be stored and analyzed as a simple unlabeled graph. In practice, $\bar{t}$ is a small integer value and $\bar{t} \ll t_{max}$. For example, $\bar{t} = 9$ for the real-world dataset *DBLP* in our experiments, and the largest $\bar{t}$ is 150 for the dataset *Email* in our experiments. The details for all other datasets can be found in Table 2.

4.3 Query Processing

The PHC-Index based query algorithm is named PHC-Query. Given the sorted windows (condition 3 in Definition 4.8) in PHC-Index, PHC-Query is supported by the following lemma.

Lemma 4.11. *Given a temporal graph $\mathcal{G}$, a vertex u, an integer k and a time window $[t_s, t_e]$, let $[t'_s, t'_e]$ be the last window in $\mathcal{PHC}(u)_k$ such that $t'_s \le t_s$. The k-core of $G_{[t_s,t_e]}$ contains u if $t'_e \le t_e$.*

Proof. According to Definition 4.8, t'_e is the core time of u for k and the input t_s. By Definition 4.2, u is in the k-core of $G_{[t_s,t_e]}$ if the input t_e is not smaller than the core time t'_e. □

Example 4.12. Given the index in Figure 4, assume $k = 3$ and the time window is $[2, 6]$. We first consider v_1. $[1, 5]$ is the last window in $\mathcal{PHC}(v_1)_3$ with $1 \le 2$. We have v_1 is in the 3-core of $G_{[2,6]}$ since $5 \le 6$. Similarly, we have other results $\{v_2, v_3, v_4\}$.

Lemma 4.13. *Given a vertex u, an integer k, an arbitrary time window and the PHC-Index, the historical k-core containment query can be answered in $O(\log |\mathcal{PHC}(u)_k|)$ time via a binary search.*

Theorem 4.14. PHC-Query *answers the historical k-core query in $O(n \cdot \log \bar{t})$ time.*

Proof. The historical k-core containment query of u can be answered in $O(\log |\mathcal{PHC}(u)_k|)$ time. Given that $\bar{t}$ is the average number of $|\mathcal{PHC}(u)_k|$ for all vertices u and core number k, the final time complexity holds. □

Other Supported Queries. The historical k-core containment query provides a basic operator, which also supports a series of relevant problems as follows.

- *Historical Core decomposition.* Historical core decomposition computes core numbers of all vertices in the projected graph of a query time window $[t_s, t_e]$. Based on Lemma 4.13, the core number of a vertex u of $[t_s, t_e]$ can be answered in $O(\log |core(u)_{[1,t_{max}]}| \cdot \log \bar{t})$ time. Therefore, the problem can be answered in $O(n \cdot \log k_{max} \cdot \log \bar{t})$ time.
- *Historical k-Core Search.* Given a time window $[t_s, t_e]$, an integer k and a vertex u, the problem of historical k-core search aims to compute the k-core containing u in the projected graph of $[t_s, t_e]$. We can start from u and add u to the result set if u is in the k-core of $[t_s, t_e]$. Then, we iteratively explore the neighbors of all unvisited resulting vertices and perform the historical k-core containment query for each neighbor. The search terminates when no other vertices can be added to the result. The time complexity of historical k-core search an be bounded by $O(|R| \log \bar{t} + |R| \log \mathcal{D} + |R|\mathcal{D}_{[t_s,t_e]})$. $|R|$ is the size of resulting vertices. $O(\log \bar{t})$ is the time for historical k-core containment query. $\mathcal{D}$ is the average number of edges containing the vertex in the graph, and it takes $O(\log \mathcal{D})$ time to locate the start edge of a vertex in the query window. $\mathcal{D}_{[t_s,t_e]}$ is the average number of neighbors in the query window of all resulting vertices.
- *Speedup for Other Models.* As discussed in Section 2.2, the result of a historical k-core is a superset of the span-core [11] and the persistent core [20] for the same time window and the integer k. This property enables us to speed up querying them based on our proposed PHC-Index. We take span-core as an example. Given a query window $[t_s, t_e]$, [11] first computes the intersection of edges at all times in the window and then compute k-cores in the snapshot of the intersection. The intersection phase is costly due to a large number of hash join operations. Based on our index, for each edge (u, v, t), we can first test whether both u and v are in the historical k-core of the window. If no, we safely remove the edge since it cannot be in the span-core either. We conduct a case study to show the effectiveness of this method in Section 6.

5 INDEX CONSTRUCTION

We propose algorithms for index construction in this section. We give a basic index algorithm PHC-Construct in Section 5.1. Section 5.2 then proposes an optimized algorithm PHC-Construct*.

5.1 A Non-Trivial Baseline

A key step in PHC-Index construction is to compute the core time. We give the following lemma to associate the core time with the core number and well-studied core decomposition. The lemma is straightforward based on the definition of core time.

Lemma 5.1. *Given a start time t_s and an integer k, the core time of u for t_s and k is t if $core(u)_{[t_s,t]} \geq k$ and $core(u)_{[t_s,t-1]} < k$.*

Based on Lemma 5.1, an immediate idea is to compute the core number of each vertex over all possible time windows and derive the core time of each vertex by comparing the core number over different time windows. Given the existing studies on core number maintenance [21, 26, 32, 36], we dynamically update core numbers from a previous time window instead of computing the core numbers from scratch. For example, given the core numbers of all vertices in $G_{[t_s,t_e]}$, we can derive the core numbers of $G_{[t_s,t_e-1]}$ by performing an updating procedure for deleting all edges at t_e. Note that existing works for core maintenance consider the fully dynamic setting. While in our case, we only need to perform one type of edge update operations (either insertion or deletion). According to[21, 26, 36], updating core numbers for edge deletions is much more efficient than that for edge insertions. Therefore, we always first compute core numbers of relatively wide time windows and derive core numbers of sub-windows in a decremental way.

Core Number Validity. The decremental maintenance of core numbers requires an additional integer value maintained for each vertex, which is defined as follows.

Definition 5.2. (Max-Core Degree) Given a graph G and a vertex u, the max-core degree of u, denoted by $\mathsf{MCD}(u)$, is the number of neighbors v of u such that $core(u) \leq core(v)$. [26, 36]

By the definition of k-core, we have $\mathsf{MCD}(u) \geq core(u)$ for each vertex u. To update core numbers, we decrease $\mathsf{MCD}(u)$ by 1 if an edge (u, v) is deleted and $core(u) \leq core(v)$. Once $\mathsf{MCD}(u) < core(u)$, we decrease $core(u)$ by 1 and compute the up-to-date $\mathsf{MCD}(u)$. The correctness of this step is supported by the observation in [26] that the core number of a vertex can decrease at most 1 when removing one edge. However, the max-core degree is used for a simple graph and cannot be straightforwardly used in our setting. Assume that we update the core number for the time window from $[t_s, t_e]$ to $[t_s, t_e - 1]$. Even though an edge $(u, v) \in \mathcal{E}_{t_e}$ with $core(u) \leq core(v)$, we cannot simply decrease $\mathsf{MCD}(u)$ since (u, v) can be still in the projected graph over $[t_s, t_e - 1]$. An example can be found in the temporal graph of Figure 1 and its snapshot of $[2, 7]$ shown in Figure 2. Assume we aim to update core numbers from $[2, 7]$ to $[2, 6]$. The edge $(v_3, v_{10}, 7)$ cannot be simply removed since (v_3, v_{10}) still exists at 6. Therefore, instead of an integer value, we use a hash table structure to maintain the max-core degree of each vertex in our algorithm, which is defined as follows.

Definition 5.3. (Core Neighbors) Given a temporal graph $\mathcal{G}$, a time window $[t_s, t_e]$ and a vertex u, the core neighbor of u, denoted by $\mathbb{CN}(u)_{[t_s,t_e]}$, is a hash table where each key is a neighbor v of u with $core(v) \geq core(u)$ in $G_{[t_s,t_e]}$, and the value of v is the number of edges (u, v, t) with $t_s \leq t \leq t_e$.

We omit the subscript $[t_s, t_e]$ of the core neighbor if it is clear from context. Let $|\mathbb{CN}(u)|$ be the number of keys. We have $|\mathbb{CN}(u)| = \mathsf{MCD}(u)$ in a projected temporal graph. When the value of a vertex v in $\mathbb{CN}(u)$ becomes 0, we implicitly remove the key v from $\mathbb{CN}(u)$ and decrease $|\mathbb{CN}(u)|$ by one.

The Algorithm. The pseudocode for PHC-Construct is presented in Algorithm 2. We first compute the core numbers in $G_{[1,t_{max}]}$

Algorithm 2: PHC-Construct()

Input: a temporal graph $\mathcal{G}$
Output: the PHC-Index of $\mathcal{G}$

1 $CoreDecomposition(G_{[1,t_{max}]})$;
2 initialize $\mathbb{CN}(u)_{[1,t_{max}]}$ for all $u \in V$;
3 **foreach** $1 \le t_s \le t_{max}$ **do**
4 $\quad$ **foreach** t: *from* t_{max} *to* $t_s + 1$ **do** DelEdges$(t, t_s, t-1)$;
5 $\quad$ reset $\mathbb{CN}(u)$ and $core(u)$ for all u in line 4;
6 $\quad$ **if** $t_s < t_{max}$ **then** DelEdges$(t_s, t_s + 1, t_{max})$;

Algorithm 3: DelEdges(t, t_s, t_e)

1 $Q \leftarrow \emptyset$;
2 **foreach** $(u, v) \in \mathcal{E}_t$ **do**
3 $\quad$ **if** $core(u) \le core(v)$ **then**
4 $\quad\quad$ decrease v in $\mathbb{CN}(u)$;
5 $\quad\quad$ **if** $|\mathbb{CN}(u)| < core(u)$ **then** $Q \leftarrow Q \cup \{u\}$;
6 $\quad$ **if** $core(v) \le core(u)$ **then**
7 $\quad\quad$ decrease u in $\mathbb{CN}(v)$;
8 $\quad\quad$ **if** $|\mathbb{CN}(v)| < core(v)$ **then** $Q \leftarrow Q \cup \{v\}$;
9 **while** $Q \neq \emptyset$ **do**
10 $\quad u \leftarrow Q.pop()$, $oldCore \leftarrow core(u)$;
11 $\quad core(u) \leftarrow$ LocalCore(u, t_s, t_e);
12 $\quad$ compute $\mathbb{CN}(u)_{[t_s,t_e]}$;
13 $\quad$ **foreach** k: *from* $oldCore$ *to* $core(u) + 1$ **do**
14 $\quad\quad [t'_s, t'_e] \leftarrow$ the last item in $\mathcal{PHC}(u)_k$;
15 $\quad\quad$ **if** $\mathcal{PHC}(u)_k = \emptyset \vee t_e + 1 > t'_e$ **then**
16 $\quad\quad\quad$ add $[t_s, t_e + 1]$ to $\mathcal{PHC}(u)_k$;
17 $\quad$ **foreach** *distinct* $v \in \mathcal{N}(u)_{[t_s,t_e]}$ **do**
18 $\quad\quad$ **if** $core(u) < core(v) \le oldCore$ **then**
19 $\quad\quad\quad$ delete v from $\mathbb{CN}(u)$;
20 $\quad\quad\quad$ **if** $|\mathbb{CN}(u)| < core(u)$ **then** $Q \leftarrow Q \cup \{u\}$;

and initialize the corresponding core neighbors of each vertex in line 1 and line 2 respectively. In the rest, we will maintain core numbers and core neighbors for all sub-windows. Specifically, for each $1 \le t_s \le t_{max}$, we iteratively remove edges at t and derive the core numbers for $[t_s, t-1]$ in line 4. In line 6, we update core numbers for the window from $[t_s, t_{max}]$ to $[t_s + 1, t_{max}]$.

DelEdges() in Algorithm 3 accepts three parameters. Parameter t refers to deleting all edges at time t; whereas t_s and t_e refer to the start time and the end time of the updated window, respectively. Other parameters are omitted for simplicity. DelEdges() processes each deleted edge in lines 2–8. Given an edge (u, v), we check the validity of $core(u)$ in lines 3–5. In line 4, to decrease v refers to decreasing the value of key v in $\mathbb{CN}(u)$ by one. Note that $|\mathbb{CN}(u)|$ will drop when the value of v in $\mathbb{CN}(u)$ becomes zero. By Definition 5.3, the core number of u cannot be $core(u)$ if $|\mathbb{CN}(u) < core(u)|$. All vertices whose core numbers require updating are added to the queue Q (lines 5 and 8).

Updating Core Number. DelEdges() computes the up-to-date core numbers and core neighbors in line 11 and line 12, respectively.

Algorithm 4: LocalCore(u, t_s, t_e)

1 initialize $cnt[k] = 0$ for each k from 1 to $core(u)$;
2 **foreach** *distinct* $v \in \mathcal{N}(u)_{[t_s,t_e]}$ **do**
3 $\quad cnt[\min(core(u), core(v))] + +$;
4 $cd \leftarrow 0$;
5 **foreach** k: *from* $core(u)$ *to* 1 **do**
6 $\quad cd \leftarrow cd + cnt[k]$;
7 $\quad$ **if** $k \le cd$ **then return** k;

In the case of single edge deletion considered in existing core maintenance algorithms, the core number of each vertex decreases by at most one for each edge deletion. However, in our problem, multiple edges may exist at the same time, and the core number of a vertex may decrease by over one. We adopt the following lemma to locally compute the up-to-date core number.

Lemma 5.4. *The core number of a vertex u is the largest integer k such that there exist at least k neighbors v with $core(v) \ge k$. [23]*

Based on Lemma 5.4, we compute the core number of a vertex by counting the number of neighbors in a non-increasing order of their core numbers. The details are shown in the procedure LocalCore(u, t_s, t_e) in Algorithm 4. Since the new core number must be smaller than the old value $core(u)$, a bucket-sort strategy is used here, and there are $core(u)$ buckets (line 1).

Lines 13-16 of Algorithm 3 add the core time of u to the index based on Lemma 5.1. Given the current window $[t_s, t_e]$, the core times of u for $t_s - 1$ and all possible k have been computed in earlier iterations. In line 15, $\mathcal{PHC}(u)_k = \emptyset$ means $t_s = 1$ and $\mathcal{CT}_1(u)_k$ has not been computed. The condition $t_e + 1 > t'_e$ prunes the cases that the core time is same as that for the previous start time.

After updating $core(u)$, lines 17–20 update $\mathbb{CN}(v)$ for each neighbor v of u of the current time window and add v to Q if $core(v)$ requires to be updated. Since the core numbers never increase in DelEdges(), the neighbor v is influenced if the old core number of u contributes to the max-core degree of v (i.e., $core(v) \le oldCore$), but the new $core(u)$ drops below $core(v)$ [32]. Note that unlike decreasing the value for v in line 4, we directly delete the key v from $\mathbb{CN}(u)$ since v can never be the core neighbor of u even though (u, v) may exist several times over the current window.

Efficient Neighbor Access. A frequent operation in the algorithm PHC-Construct() is to scan the neighbors of each vertex over a specific time window, like line 12, line 17 of Algorithm 3 and line 2 of Algorithm 4. Given a time window $[t_s, t_e]$, a straightforward implementation is performing a binary search to locate the first neighbor since t_s and breaking the iteration of neighbor access once the neighbor time exceeding t_e.

To improve pracitcal efficiency, we maintain an integer value for each vertex u, which represents the offset of the first neighbor of u since the current t_s. The offset for each vertex is initialized as 1, which means the scanning of neighbors will start from the first item. During the algorithm, we always scan neighbors of each vertex starting from the offset since the start time t_s monotonically increases. For each new start time t_s, we update the offset for every vertex as a byproduct when scanning the neighbors of u. With the offset value, we avoid the binary search of neighbors and a great amount of unnecessary access.

Algorithm 5: LocalCT$(u, t_s, k, \mathcal{CT}, \mathcal{G})$

```
1  T ← ∅, ub ← 0;
2  foreach ⟨v, t⟩ ∈ N(u)_[t_s, t_max] do
3      if v is visited then continue;
4      if |T| ≥ k ∧ t ≥ ub then break;
5      mark v as visited;
6      ct ← max(t, CT(v)_k);
7      T ← T ∪ {ct};
8      if |T| ≤ k then ub ← max(ub, ct);
9  if k ≤ |T| then return k-th smallest value in T;
10 core(u) ← min(core(u), k − 1);
11 return ∞;
```

Theorem 5.5. *Given a temporal graph $\mathcal{G}$, the running time of Algorithm 2 is bounded by $O(m \cdot k_{max} \cdot t_{max})$, where k_{max} is the maximum core number in all projected graphs.*

Proof. In DelEdges, lines 2–8 take $O(|\mathcal{E}_t|)$ time. The overall time complexity of lines 2–8 is $O(m \cdot t_{max})$ time since we scan t_{max} times of the graph (line 3 in Algorithm 2). Then we consider lines 10–20 of the procedure DelEdges. Line 11 takes $O(|\mathcal{N}(u)|)$ time, which is the dominant time complexity in lines 10–20. Given a start time t_s and a vertex u, LocalCore is only invoked when the core number of u updates. Given a start time t_s, LocalCore can be invoked at most $O(k_{max} \cdot n)$ times, and the total time complexity by invoking LocalCore is $O(\sum_{u \in \mathcal{V}} |\mathcal{N}(u)| k_{max}) = O(m \cdot k_{max})$. Therefore, the overall time complexity is $O(m \cdot k_{max} \cdot t_{max})$. □

5.2 Optimized Index Construction

Drawbacks of Algorithm 2. Algorithm 2 works correctly but surfers from a relatively poor scalability due to the factor m and t_{max} in Theorem 5.5. Specifically, based on Lemma 4.7, the core time of a vertex u for a specific k can be the same at different start times t_s. However, Algorithm 2 computes the core times of all vertices at every start time from 1 to t_{max}, which incurs the factor t_{max} in the running time complexity. In addition, computing core times of all vertices at a start time t_s takes $O(m \cdot k_{max})$ time. While, possibly, only a small number of core times at t_s can be larger than that at the previous start time (e.g., the second condition in line 15 of Algorithm 3) and are added to the index. In other words, much time is wasted in computing the same core times as those at previous start times. Consider the example graph in Figure 1 with 8 distinct times. We perform 8 iterations in line 3 and a total of 28 iterations in line 4 of Algorithm 2.

To improve the efficiency of index construction, we propose an algorithm that only computes the core times which are different from their counterparts in earlier iterations. The computation of each core time only relies on local information, which avoids the $O(m \cdot k_{max})$ time for all vertices at a start time t_s.

Core Time Validity. Given the core time $\mathcal{CT}_{t_s}(u)_k$ of a vertex u, when moving the start time from t_s to $t_s + 1$ (line 3 of Algorithm 2), all edges in $\mathcal{E}_{t_s}$ are deleted. We define an additional structure to identify whether $\mathcal{CT}_{t_s+1}(u)_k = \mathcal{CT}_{t_s}(u)_k$.

Definition 5.6. (CT Degree) Given a vertex u and its core time $\mathcal{CT}_{t_s}(u)_k = t$, the CT (Core Time) degree of u, denoted by $\mathrm{CTD}_{t_s}(u)_k$, is the number of neighbors in the k-core of $G_{[t_s,t]}$.

We always have $\mathrm{CTD}_{t_s}(u)_k \geq k$ according to the definition of core times. When removing edges, we assume the core time of each vertex u does not change and maintain the CT degree of u. Consequently, we may have $\mathrm{CTD}_{t_s}(u)_k < k$ for some vertices u, which means the core time of u is not valid. To efficiently maintain the CT degree of each vertex, we define the following hash table.

Definition 5.7. (CT Neighbors) Given a vertex u and its core time $\mathcal{CT}_{t_s}(u)_k = t_e$, the CT neighbors of u, denoted by $\mathbb{CTN}_{t_s}(u)_k$, is a hash table where each key is a vertex v with $\mathcal{CT}_{t_s}(u)_k \leq t_e$ and $(u, v) \in G_{[t_s,t_e]}$, and each value of v is the number of edges (u, v, t) with $t_s \leq t \leq t_e$.

We use $|\mathbb{CTN}_{t_s}(u)_k|$ to denote the number of keys (distinct vertices). The following lemmas immediately hold.

Lemma 5.8. $|\mathbb{CTN}_{t_s}(u)_k| = |\mathrm{CTD}_{t_s}(u)_k|$.

Lemma 5.9. $|\mathbb{CTN}_{t_s}(u)_k| \geq k$.

The idea of PHC-Index is to maintain the CT neighbors for each vertex u in the index construction. Given a removed edge, we use constant time to update the CT neighbor by assuming the corresponding core time remains unchanged. Once $|\mathbb{CTN}_{t_s}(u)_k| < k$, we recompute the up-to-date $\mathcal{CT}_{t_s}(u)_k$ and $\mathbb{CTN}_{t_s}(u)_k$.

Local Core Time Computation. We propose a new method to compute the core time of a vertex u instead of that in Algorithm 2, which has a global order dependency.

Lemma 5.10. *Given a start time t_s and an integer k, the core time of a vertex u is the smallest time t such that there exist a set of vertices C satisfying (1) $|C| \geq k$; (2) $\forall v \in C : \mathcal{CT}_{t_s}(v)_k \leq t$ and $\exists (u, v) \in G_{[t_s,t]}$.*

Proof. Based on Definition 4.2, assume the core time of u is t. There are at least k neighbors v of u with core number not less than k in the projected graph of $[t_s, t]$ [23, 32]. There does not exist a value $t' < t$ satisfying the condition. Otherwise, the core time would be t'. For each such neighbor v, the core time of v is not larger than t since $core(v) \geq k$ over $[t_s, t]$, and there must exist an edge (u, v) in the projected graph of $[t_s, t]$. □

Lemma 5.10 provides a method to compute the core time of u by the core times of u's neighbors. The detailed process is given in Algorithm 5, which returns the core time of the input vertex u. The start time t_s, an integer k, core times of other vertices $\mathcal{CT}$ and the graph $\mathcal{G}$ are also given. The algorithm iteratively scans each neighbor over the time window in line 2. Note that the neighbors are naturally arranged in chronological order. The algorithm adopts an early termination strategy in line 4 and line 8. Specifically, after processing k distinct neighbors, the variable ub in line 8 is the largest value in T and an upper bound of $\mathcal{CT}_{t_s}(u)_k$. The iteration is terminated if $t \geq ub$ (line 4) since t never decreases, and the k-th smallest values in T cannot change.

Lemma 5.11. *Algorithm 5 correctly computes the core time of the input vertex u for t_s and k.*

Proof. The correctness is supported by Lemma 5.10. In the algorithm, ct in line 5 is the earliest time that the neighbor can contribute to the k-core of u. When $|T| = k$ in line 8, the algorithm has visited k distinct neighbors. Now, the set of processed neighbors has satisfied the two conditions in Lemma 5.10 where t in the lemma is assigned by ub in the algorithm. The following iterations will try to find the smallest time in the lemma. Given that ub has been an upper bound of the core time, the algorithm terminates if $t \geq ub$ in line 4 or all neighbors have been processed. □

Algorithm 6: PHC-Construct*()

Input: a temporal graph G
Output: the PHC-Index of G

1 $CoreDecomposition(G_{[1,t_{max}]})$;
2 **foreach** t: *from* t_{max} *to* 2 **do** DelEdges$(t, 1, t-1)$;
3 **foreach** $u \in V$ **do** initialize $\mathbb{CTN}(u)_k$ for $1 \le k \le core(u)$;
4 **foreach** $2 \le t_s \le t_{max}$ **do**
5 $Q \leftarrow \emptyset$;
6 **foreach** $(u, v) \in E_{t_s-1}$ **do**
7 **foreach** $1 \le k \le \min(core(u), core(v))$ **do**
8 DelCTN$(u, v, t_s - 1, k, Q)$;
9 DelCTN$(v, u, t_s - 1, k, Q)$;
10 **while** $Q \neq \emptyset$ **do**
11 $\langle u, k\rangle \leftarrow Q.pop(), oldCT \leftarrow \mathcal{CT}(u)_k$;
12 $\mathcal{CT}(u)_k \leftarrow$ LocalCT$(u, t_s, k, \mathcal{CT}, \mathcal{G})$;
13 add $[t_s, \mathcal{CT}(u)_k]$ to $\mathcal{PHC}(u)_k$;
14 compute $\mathbb{CTN}(u)_k$ for t_s;
15 **foreach** $\langle v, t\rangle \in \mathcal{N}(u)_{[t_s,t_{max}]}$ **do**
16 **if** v *is visited* **then continue**;
17 **if** $\mathcal{CT}(u)_k \le t$ **then break**;
18 mark v as visited;
19 **if** $\max(oldCT, t) \le \mathcal{CT}(v)_k < \mathcal{CT}(u)_k$ **then**
20 delete u from $\mathbb{CTN}(v)_k$;
21 **if** $|\mathbb{CTN}(v)_k| < k$ **then** $Q \leftarrow Q \cup \{\langle v, k\rangle\}$;
22 reset all vertices as unvisited;

Algorithm 7: DelCTN(u, v, t, k, Q)

1 **if** $\max(\mathcal{CT}(v)_k, t) \le \mathcal{CT}(u)_k$ **then**
2 decrease v in $\mathbb{CTN}(u)_k$;
3 **if** $|\mathbb{CTN}(u)_k| < k$ **then** $Q \leftarrow Q \cup \{\langle u, k\rangle\}$;

Example 5.12. Consider the graph in Figure 1. All temporal edges connecting to v_3 are in Figure 5. The gray area shows the core time of each neighbor for $k = 3$ and $t_s = 2$. According to Lemma 5.10, the core time of v_3 is 5 since we have three neighbors $\{v_1, v_2, v_4\}$ with core time not later than 5 and connecting to v_3 not later than 5. We cannot find any earlier time.

Lemma 5.13. *The time complexity of Algorithm 5 is* $O(|\mathcal{N}(u)|)$.

The Final Algorithm for Index Construction. The pseudocode of the optimized algorithm for index construction is presented in Algorithm 6. Line 2 initializes the core times of each vertex for all possible core numbers at the start time 1. Lines 4–22 increase the start time t_s and update the corresponding core times if necessary. Lines 6–9 process each removed edge from t_s to $t_s + 1$.

The procedure DelCTN() shown in Algorithm 7 is used to update the CT neighbors. By Lemma 5.9, we add u to the queue in line 3 of DelCTN() if the core time of u for k requires to be updated.

In line 7 of Algorithm 6, for any integer $k > \min(core(u), core(v))$, neither u nor v can be in the CT neighbor of the other according to Definition 5.7. Line 12 recomputes the core time of u for k and t_s. Unlike Algorithm 2, the computed core time in line 12 is never the same as that for previous start times, and can be safely added to the index in line 13. Line 14 recomputes the CT neighbors of u for k. Lines 15–21 update the status of neighbors. The CT neighbor of a vertex v removes the key vertex u if the core time of u increases and never be in the CT neighbor of v according to Definition 5.7. In line 19, $\max(oldCT, t) \le \mathcal{CT}(v)_k$ is equivalent to $u \in \mathbb{CTN}(v)_k$. The vertex v is added to Q to update the core time if necessary. Similar to Algorithm 2, we also maintain an offset to indicate the start position for the start time of the window. Consequently, it takes $O(1)$ amortized time complexity to access the first neighbor of a vertex for each start time t_s.

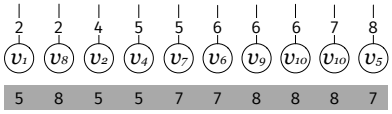


Figure 5: Neighbors of v_3, and the core time of each neighbor for $k = 3$ and $t_s = 2$ in the temporal graph $\mathcal{G}$

Theorem 5.14. *The running time of Algorithm 6 is bounded by* $O(|\mathcal{PHC}| \cdot \mathcal{D}_{max})$, *where* $\mathcal{D}_{max}$ *is the maximum degree.*

Proof. In calculating the time complexity of lines 11-22 in Algorithm 6, Algorithm 5 is the dominant step. Algorithm 5 is invoked when the core time of the vertex u changes. In other words, Algorithm 5 is invoked for each time window in PHC-Index according to the definition of PHC-Index. Therefore, the total time complexity of Algorithm 6 is $O(|\mathcal{PHC}| \cdot \mathcal{D}_{max})$, which can be also rewritten as $O(m^* \cdot \bar{t} \cdot \mathcal{D}_{max})$. □

5.3 Supporting Time Intervals on Edges.

In real-world temporal graphs, e.g. social networks and communication networks, edges may be associated with a time interval $[t, t']$ (i.e., inserted at t and removed at t') instead of a single timestamp. We call such dataset an interval-based temporal graph.

The Snapshot Model. An immediate question for interval-based temporal graphs is which edges are considered in the snapshot of the query window. We focus on a straightforward intersection model in this paper. Specifically, given a query time window $[t_s, t_e]$, we derive a snapshot $G_{[t_s,t_e]}$ by merging all edges whose time intervals overlap with the window between any two vertices into a single unlabeled edge, i.e., $G_{[t_s,t_e]} = (\mathcal{V}, \{(u, v)|(u, v, t, t') \in \mathcal{E}, t \le t_e \wedge t' \ge t_s\})$. We will show later that the idea of PHC-Index can naturally handle this situation.

An alternative model to formulate the snapshot is to take all edges inserted during $[t_s, t_e]$ but not removed until t_e, i.e., $G_{[t_s,t_e]} = (\mathcal{V}, \{(u, v)|(u, v, t, t') \in \mathcal{E}, t_s \le t \le t_e \wedge t' > t_e\})$. Indexing historical k-cores under this model is hard since the monotonicity property (e.g., Lemma 4.1) does not hold. We leave it for future works. A potential idea is to precompute the core numbers of each possible time window for a set of carefully selected vertices. In query processing, we use the core numbers of these indexed landmark vertices to speed up the k-core computation.

Several monotonicity properties (e.g. Lemma 4.1 and Lemma 4.7) in the historical k-core problem still hold for the interval-based temporal graph. Therefore, the index structure still works. To construct the index for the interval-based temporal graph, we preprocess the graph and derive two sorted list. The first list and the second list arrange all start times and all end times of edge in chronological order, respectively. Then, we modify Algorithm 6 based on the two

	CollegeMsg	*Email*	*MathOverflow*	*AskUbuntu*	*SuperUser*	*WikiTalk*	*Youtube*	*DBLP*	*Flickr*	*Wikipedia*
n	1,899	986	24,818	159,316	194,085	1,140,149	3,223,589	1,824,701	2,302,926	1,870,710
m	59,835	332,334	506,550	964,437	1,443,339	7,833,140	9,375,374	29,487,744	33,140,017	39,953,145
k_{max}	20	34	78	48	61	124	88	286	600	206
t_{max}	58,911	207,880	390,157	725,568	1,106,768	6,088,535	5,201,409	2,612,156	134	2,198
$\bar{t}$	50.03	150.59	104.93	30.17	41.94	97.09	31.27	9.1	8.89	35.33
m^*	13,838	16,064	187,986	455,691	714,570	2,787,967	9,375,374	8,344,615	22,838,276	36,532,531
$\mathcal{D}_{max}$	1,546	10,571	7,421	10,122	27,183	233,954	91,751	7,276	34,174	226,577

Table 2: Statistics of datasets.

lists. In line 2, we replace the single time of each vertex with the start time of each vertex. Specifically, we start from the latest start time and iteratively remove the corresponding edge when decreasing the start times. Visiting a start time of an edge represents that such edge cannot be considered in the current time window. After performing line 2, we similarly derive the core time of each vertex for the earliest start time. Recall that lines 4–22 of Algorithm 6 remove edges when the start time increases and update core times if necessary. For the interval-based temporal graph, we only consider the end times in line 4. We iteratively remove edges according to the end time list in line 6 because the core time of a vertex updates only if there exists an edge expiring at the current time.

6 EXPERIMENTS

We conduct extensive experiments to evaluate the performance of our proposed algorithms. All algorithms are implemented in C++ and compiled using the g++ compiler at -O3 optimization level. We run experiments on a Linux machine with an Intel Xeon 3.7GHz CPU and 64GB RAM. All hash tables in our proposed algorithms are implemented with C++ STL.

Datasets. We evaluate algorithms on eight real-world temporal graphs. The detailed statistics are presented in Table 2, where k_{max} is the largest possible core number of all vertices in the graph, t_{max} is the number of distinct timestamps in the graph, and $\bar{t}$ is the average number of time windows for each k of a vertex in our PHC-Index. The first six datasets are derived from SNAP[1], and the last two datasets are derived from the KONECT project[2].

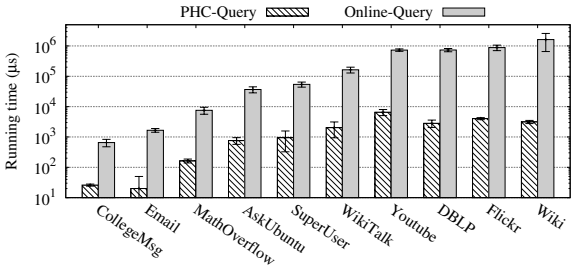


Figure 6: Running time of query algorithms

6.1 Query Processing

Parameters. We compare the proposed query processing algorithm PHC-Query for historical k-core problem with the online algorithm, called Online-Query. Regarding the input parameters, we vary the size of the query time window as 5%, 10%, 20%, 40%, 60% and 80% of t_{max} for each dataset with 60% as default. We vary the integer k as 20%, 40%, 60% and 80% of k_{max} for each dataset with 60% as default. Given the window size and k, we randomly pick 1000 time windows over $[1, t_{max}]$ with the same size and record the average running time of 1000 queries.

Overall Query Efficiency. The running times of both two algorithms under the default parameters are reported in Figure 6. We can see that the speedup of PHC-Query is obvious. PHC-Query is at least one order of magnitude faster than Online-Query and is over two orders of magnitude faster in several large datasets. For example, in *WikiTalk*, PHC-Query takes average 2ms for each query, while Online-Query takes up to 165ms.

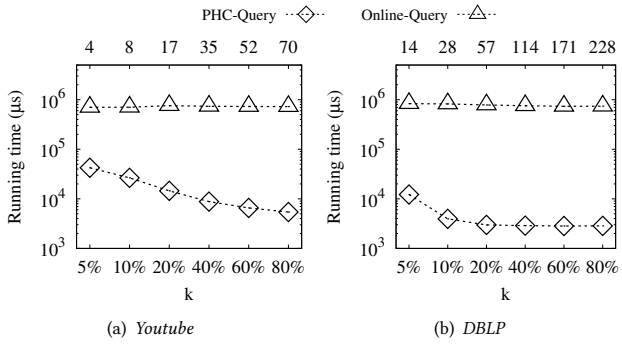
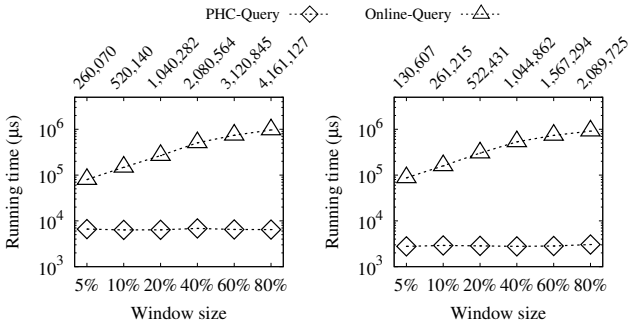


Figure 7: Query time by varying k

Varying Parameters. The running times of algorithms PHC-Query and Online-Query by varying the parameter k are reported in Figure 7. We report the results of two representative large datasets — *Youtube* and *DBLP*. The results of other datasets show the similar trends. The results show that PHC-Query considerably outperforms Online-Query in all settings. The time of PHC-Query performs a slightly downward trend, where is visible from the *Youtube* dataset. The time decreases from 14ms at $k = 17$ ($k_{max} \times 20\%$) to 5ms at $k = 70$ ($k_{max} \times 80\%$), the reason is when k is relatively large, the core numbers of many vertices are smaller than k even in the projected graph of $[1, t_{max}]$. Therefore, it takes constant time to identify that such vertices cannot be in the result instead of performing the binary search. The decrease of PHC-Query for *DBLP* is not obvious since the decrease of the result size is not considerable when k increases. By contrast, the running time of Online-Query is almost stable, since the algorithm needs to scan the whole snapshot given the fixed window size.

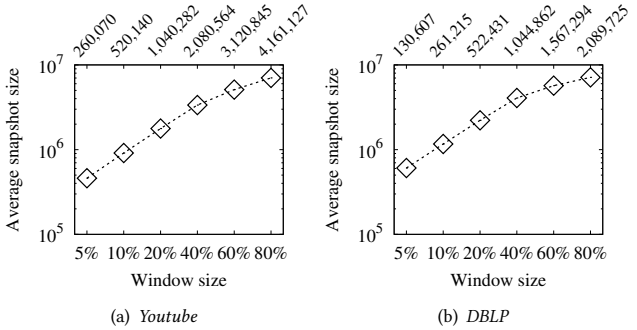
The running times of algorithms PHC-Query and Online-Query by varying the window size are reported in Figure 8. We also show the average snapshot size (number of edges in the corresponding projected graph) of each query in Figure 9 when the window size increases. In contrast to the results from varying k, we can see a remarkable increase of the running time of Online-Query. For example, in *DBLP*, the running time of Online-Query increases

[1] http://snap.stanford.edu/
[2] http://konect.cc/



(a) *Youtube* (b) *DBLP*

Figure 8: Query time by varying window size



(a) *Youtube* (b) *DBLP*

Figure 9: Average snapshot size by varying window size

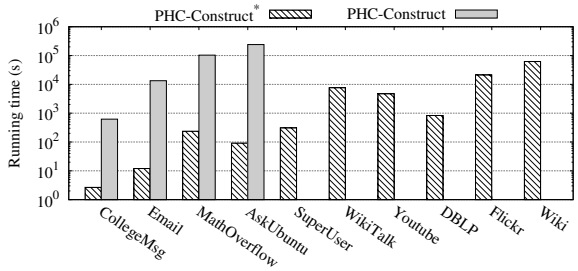
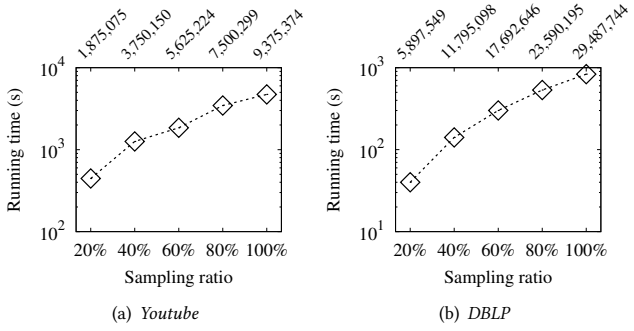


Figure 10: Running time of index construction algorithms

from 87ms at 5% to 0.9s at 80%, the reason for this is because the running time of Online-Query depends on the size of the graph snapshot. However, the time of PHC-Query remains stable, since PHC-Query only performs a binary search for every vertex, and the running time is not sensitive to the size of graph snapshots.

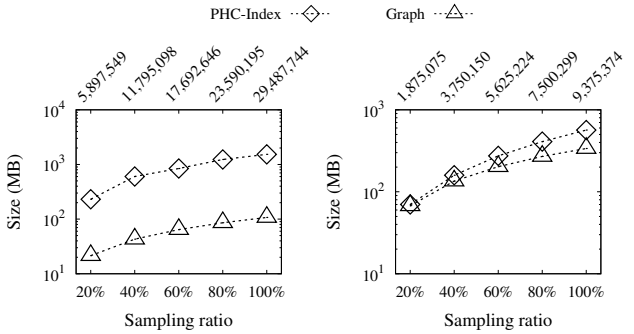
6.2 Index Construction

We report the running time of our final index construction algorithm PHC-Construct* in Figure 10 with PHC-Construct as a comparison. Given that our PHC-Construct essentially computes core numbers for all time windows, we omit tests on the naive HC-Index construction algorithm, which is almost the same as PHC-Construct. The results of PHC-Construct are not reported for several large datasets since the algorithm cannot finish within 12hrs. We can see that PHC-Construct* is several orders of magnitude faster than PHC-Construct thanks to our new strategy for core time computation and maintenance. In the smallest dataset *CollegeMsg*, the algorithm PHC-Construct* takes about 2.5s while PHC-Construct takes up to 620s. For datasets *DBLP* and *WikiTalk*, PHC-Construct* finishes in 14mins and in about 2hrs respectively.



(a) *Youtube* (b) *DBLP*

Figure 11: Running time of index construction algorithms for different sampling ratios



(a) *Youtube* (b) *DBLP*

Figure 12: Index size for different sampling ratios

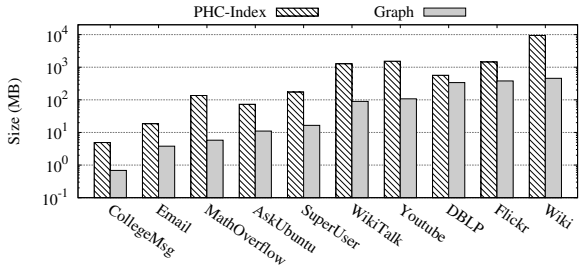
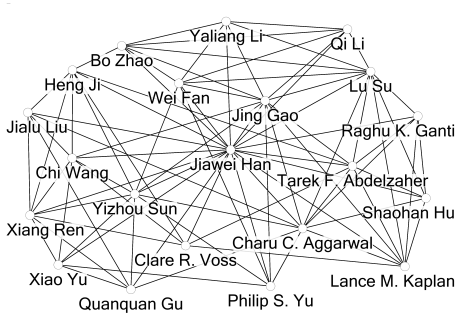


Figure 13: Index size for all datasets.

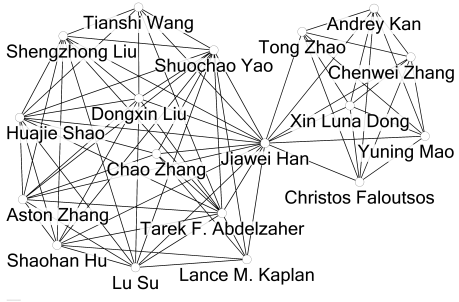
Scalability Testing. We evaluate the scalability of our index construction algorithm on two representative datasets *Youtube* and *DBLP*. For each dataset, all edges are arranged in chronological order. We pick the first 20%, 40%, 60%, 80% of the edges from the original graph to perform the algorithm. The running time of PHC-Construct* by varying the sampling ratio is reported in Figure 11, the results of PHC-Construct are not reported since it does not finish within 12hrs even when sampling 20% of the original graph. The index size and graph size by varying the sampling ratio are also given in Figure 12. For *DBLP*, PHC-Construct* takes about 40s at 20% and reaches to 140s, 301s, and 534s at 40%, 60% and 80% respectively. PHC-Index for *DBLP* takes 70MB, 159MB, 274MB and 409MB at 40%, 60% and 80%, respectively.

6.3 Index Size

We report the size of our PHC-Index in Figure 13 with the graph size as a comparison, the peak memory usage of PHC-Construct* is also



(a) *DBLP* (2011–2015)



(b) *DBLP* (2016–2020)

Figure 15: DBLP case study (6-core)

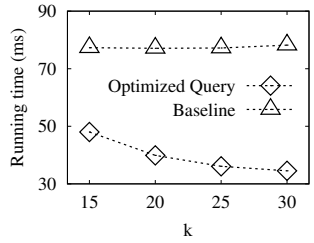


Figure 16: Performance of span-core computation

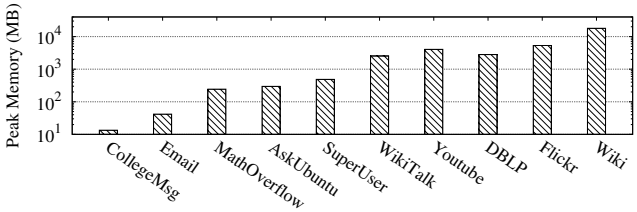


Figure 14: Peak memory usage of index construction

provided in Figure 14 as a reference. Among the datasets, the index size for *Wikipedia* is about 9.1GB, which is the largest one among all datasets. We can see that the pruning effectiveness of our index structure depends on the practical graph structure. Compared with *Youtube* and *Wikipedia*, whose index sizes are over ten times larger than teh graph, the index sizes of *DBLP* and *Flickr* perform well because a large of core times are pruned, and the average number of time windows $\bar{t}$ in Theorem 4.10 is small. Note that in our experiments, PHC-Index is constructed to support all possible start and end times by default based on the original Unix timestamps of edges in each dataset. In some real-world scenarios, the number of possible query windows may be reduced. For example, we may only need to support time windows with clear semantics, like always from the start of one calendar month to the end of the other. Or the temporal graph has been preprocessed into a series of graph snapshots. Our index still works in such cases, and the size can also be significantly reduced. We do not show the space of HC-Index since it is extremely large. Given t_{max} and n in Table 2, the size of HC-Index can be easily computed. For example, even for the smallest dataset *CollegeMsg*, HC-Index takes up to 12TB space.

6.4 Case Studies

In case studies, we generate a temporal graph by collecting the publication data of DBLP from 2011 to 2020. Each vertex is a researcher. For each year, we create an edge with the year number between two vertices if they have at least one common paper. As a result, we have $t_{max} = 10$ in the generated graph.

Historical k-Cores in DBLP. We show the 6-cores of Prof. Jiawei Han's ego network on the DBLP snapshots of 2011–2015 and 2016–2020, respectively. Note that given the large number of publications, we only keep edges between authors if they have at least three common publications for clearness. Figure 15 presents the resulting subgraphs, which shows a great difference between two times.

Accelerating Span-Core Computation. We evaluate the performance of span-core computation [11] based on our PHC-Index on the real-world dataset DBLP. Given a time window and the integer k, the baseline algorithm first creates a graph snapshot by computing the intersection of edges at all times with in the window. Then, the algorithm derives the final k-core by iteratively removing each vertices with a degree smaller than k in the snapshot. The optimized algorithm uses PHC-Index to prune all edges containing a vertex that cannot be in the historical k-core of the query time window before computing the intersection.

We set the window size as 2 and vary k from 15 to 30. For each k, we compute the k-core of every size-2 window and record the average running time of all windows with a non-empty resulting subgraph. The results are shown in Figure 16. In addition to the significant speedup, we can find that the running time of the baseline is stable when increasing k since the dominating cost is in the intersection operation.

7 CONCLUSION

In this paper, we study the problem of efficiently querying historical k-cores in a large temporal graph. Given a time window and an integer k, the problem aims to compute the k-core in the graph snapshot of the query time window. We propose an index-based solution for the problem. We propose an optimized algorithm to efficiently construct the index. Several experiments are conducted to show the efficiency of our solution. Several potential problems are also opened. For example, given the highly dynamic graph, algorithms can be designed to update the PHC-Index. A parallel implementation of PHC-Construct* may also be done to further improve the efficiency of index construction.

ACKNOWLEDGEMENT.

Ying Zhang is supported by ARC FT170100128 and ARC DP210101393. Lu Qin is supported by ARC FT200100787 and DP210101347. Wenjie Zhang is supported by ARC DP180103096 and ARC DP200101116. Xuemin Lin is supported by ARC DP180103096 and ARC DP170101628.

REFERENCES

[1] Hidayet Aksu, Mustafa Canim, Yuan-Chi Chang, Ibrahim Korpeoglu, and Özgür Ulusoy. 2014. Distributed k-Core View Materialization and Maintenance for Large Dynamic Graphs. *TKDE* 26, 10 (2014), 2439–2452.

[2] J. Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. 2005. Large scale networks fingerprinting and visualization using the k-core decomposition. In *NIPS*. 41–50.

[3] Reid Andersen and Kumar Chellapilla. 2009. Finding Dense Subgraphs with Size Bounds. In *WAW*, Konstantin Avrachenkov, Debora Donato, and Nelly Litvak (Eds.), Vol. 5427. 25–37.

[4] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O(m) Algorithm for Cores Decomposition of Networks. *CoRR* cs.DS/0310049 (2003).

[5] Lijun Chang. 2019. Efficient Maximum Clique Computation over Large Sparse Graphs. In *KDD*, Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis (Eds.). 529–538.

[6] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Xuemin Lin, Chengfei Liu, and Weifa Liang. 2013. Efficiently computing k-edge connected components via graph decomposition. In *SIGMOD*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). 205–216.

[7] James Cheng, Yiping Ke, Shumo Chu, and M. Tamer Özsu. 2011. Efficient core decomposition in massive networks. In *ICDE*, Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan (Eds.). 51–62.

[8] Wanyun Cui, Yanghua Xiao, Haixun Wang, and Wei Wang. 2014. Local search of communities in large graphs. In *SIGMOD*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). 991–1002.

[9] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2017. Julienne: A Framework for Parallel Graph Algorithms using Work-efficient Bucketing. In *SPAA*. 293–304.

[10] Yixiang Fang, Reynold Cheng, Siqiang Luo, and Jiafeng Hu. 2016. Effective Community Search for Large Attributed Graphs. *PVLDB* 9, 12 (2016), 1233–1244.

[11] Edoardo Galimberti, Alain Barrat, Francesco Bonchi, Ciro Cattuto, and Francesco Gullo. 2018. Mining (maximal) Span-cores from Temporal Networks. In *CIKM*. 107–116.

[12] Edoardo Galimberti, Francesco Bonchi, and Francesco Gullo. 2017. Core Decomposition and Densest Subgraph in Multilayer Networks. In *CIKM*. 1807–1816.

[13] Antonios Garas, Frank Schweitzer, and Shlomo Havlin. 2012. A k-shell decomposition method for weighted networks. *New Journal of Physics* 14, 8 (2012), 083030.

[14] Christos Giatsidis, Dimitrios M. Thilikos, and Michalis Vazirgiannis. 2011. D-cores: Measuring Collaboration of Directed Graphs Based on Degeneracy. In *ICDM*. 201–210.

[15] Svante Janson and Malwina J. Luczak. 2007. A simple solution to the k-core problem. *Random Struct. Algorithms* 30, 1-2 (2007), 50–62.

[16] Wissam Khaouid, Marina Barsky, S. Venkatesh, and Alex Thomo. 2015. K-Core Decomposition of Large Networks on a Single PC. *PVLDB* 9, 1 (2015), 13–23.

[17] Udayan Khurana and Amol Deshpande. 2013. Efficient Snapshot Retrieval over Historical Graph Data. In *ICDE*. 997–1008.

[18] Alan G. Labouseur, Paul W. Olsen, and Jeong-Hyon Hwang. 2013. Scalable and Robust Management of Dynamic Graph Data. In *BD3 VLDB 2013*, Vol. 1018. 43–48.

[19] Rong-Hua Li, Lu Qin, Fanghua Ye, Jeffrey Xu Yu, Xiaokui Xiao, Nong Xiao, and Zibin Zheng. 2018. Skyline Community Search in Multi-valued Networks. In *SIGMOD*. 457–472.

[20] Rong-Hua Li, Jiao Su, Lu Qin, Jeffrey Xu Yu, and Qiangqiang Dai. 2018. Persistent Community Search in Temporal Networks. In *ICDE*. 797–808.

[21] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. 2014. Efficient Core Maintenance in Large Dynamic Graphs. *TKDE* 26, 10 (2014), 2453–2465.

[22] Tomasz Luczak. 1991. Size and connectivity of the k-core of a random graph. *Discret. Math.* 91, 1 (1991), 61–68.

[23] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. 2013. Distributed k-Core Decomposition. *TPDS* 24, 2 (2013), 288–300.

[24] Katerina Pechlivanidou, Dimitrios Katsaros, and Leandros Tassiulas. 2014. MapReduce-Based Distributed K-Shell Decomposition for Online Social Networks. In *SERVICES*. 30–37.

[25] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. 2011. On Querying Historical Evolving Graph Sequences. *PVLDB* 4, 11 (2011), 726–737.

[26] Ahmet Erdem Sariyüce, Bugra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. 2013. Streaming Algorithms for k-core Decomposition. *PVLDB* 6, 6 (2013), 433–444.

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

[28] Konstantinos Semertzidis and Evaggelia Pitoura. 2016. Durable graph pattern queries on historical graphs. In *ICDE*. 541–552.

[29] Konstantinos Semertzidis, Evaggelia Pitoura, and Kostas Lillis. 2015. TimeReach: Historical Reachability Queries on Evolving Graphs. In *EDBT*. 121–132.

[30] Dong Wen, Yilun Huang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2020. Efficiently Answering Span-Reachability Queries in Large Temporal Graphs. In *ICDE*. 1153–1164.

[31] Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Ling Chen. 2019. Enumerating k-Vertex Connected Components in Large Graphs. In *ICDE*. 52–63.

[32] Dong Wen, Lu Qin, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2016. I/O efficient Core Graph Decomposition at web scale. In *ICDE*. 133–144.

[33] Huanhuan Wu, James Cheng, Yi Lu, Yiping Ke, Yuzhen Huang, Da Yan, and Hejun Wu. 2015. Core decomposition in large temporal graphs. In *BigData*. 649–658.

[34] Bohua Yang, Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Rong-Hua Li. 2019. Index-Based Optimal Algorithm for Computing K-Cores in Large Uncertain Graphs. In *ICDE*. 64–75.

[35] Haohua Zhang, Hai Zhao, Wei Cai, Jie Liu, and Wanlei Zhou. 2010. Using the k-core decomposition to analyze the static structure of large-scale software systems. *J. Supercomput.* 53, 2 (2010), 352–369.

[36] Yikai Zhang, Jeffrey Xu Yu, Ying Zhang, and Lu Qin. 2017. A Fast Order-Based Approach for Core Maintenance. In *ICDE*. 337–348.