On More Efficiently and Versatilely Querying Historical k-Cores

Zhi Wang
School of Computer Science, Wuhan University
Wuhan, China
zhiwang@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
Wuhan University
Wuhan, China
qty@whu.edu.cn

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

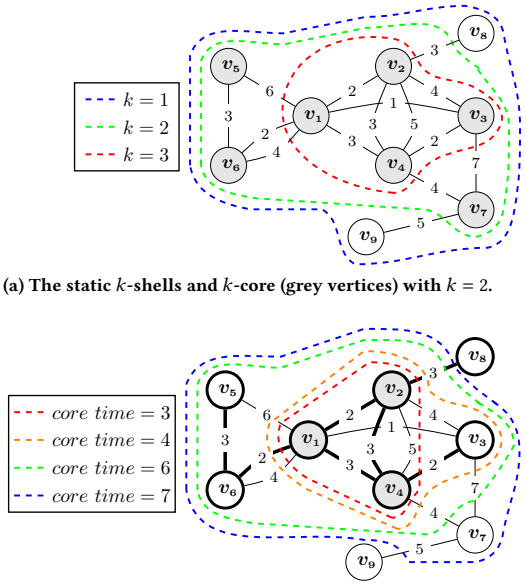
Jeffrey Xu Yu
The Chinese University of Hong Kong
Hong Kong, China
yu@se.cuhk.edu.hk

ABSTRACT

The recently proposed historical k-core query introduces a new paradigm of structure analysis for temporal graphs. However, the query processing based on the existing PHC-index, which preserves the distinct "core time" of each vertex, needs to traverse all vertices for each query, even though the results usually contain only a small subset of vertices. Inspired by the traditional k-shell that ensures the optimal k-core query processing, we propose a novel concept called "core time shell", which reveals the hierarchical structure of vertices with respect to their core time. Based on the core time shell, we design a time-space balanced Merged Core Time Shell index (MCTS-index). It is theoretically guaranteed that, the MCTS-index provides the approximately optimal query performance, and has the approximately same space complexity as the PHC-index. Moreover, we leverage the MCTS-index to efficiently address the brand-new "when" historical k-core queries orthogonal to the current "what" historical k-core queries. Our experimental results on ten real-world temporal graphs demonstrate both the superior efficiency of processing "what" queries and the effectiveness of processing versatile "when" queries for the MCTS-index.

PVLDB Reference Format:
Zhi Wang, Ming Zhong, Yuanyuan Zhu, Tieyun Qian, Mengchi Liu, and Jeffrey Xu Yu. On More Efficiently and Versatilely Querying Historical k-Cores. PVLDB, 18(5): 1335 - 1347, 2025.
doi:10.14778/3718057.3718063

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



(a) The static k-shells and k-core (grey vertices) with $k = 2$.

(b) The core time shells for $k = 2$ and the start time $t_s = 2$, and the historical k-core (grey vertices) until the end time $t_e = 3$.

Figure 1: A running example of temporal graph with two kinds of views: k-shell and core time shell, respectively.

1 INTRODUCTION

Recently, it has been widely recognized that the temporal graphs should gain more research attention than the static graphs [14]. For example, a variety of real-world temporal graphs have been studied, such as communication networks [12], social networks [36], transaction networks [16], and transportation networks [6], though their static versions emerged earlier. Thus, the new paradigms of querying temporal graphs are becoming prevalent research topics.

As a basic query paradigm for temporal graphs, the so-called "historical" graph queries are to find entities such as vertices, paths, subgraphs, or even boolean/statistical values from temporal graphs within a given moment or period. Khurana and Deshpande [9] have given a visionary taxonomy of historical graph queries. Then, many research works [2, 7, 18, 25, 30, 31, 33, 34, 37] have appeared, and a temporal graph analytics system [5] has been proposed to support developer to implement the query algorithms efficiently.

Among the various historical graph queries, the historical k-core query deserves deeper exploration, since the existing state-of-the-art solution called PHC-query [34] is not guaranteed to be optimal. In a nutshell, the historical k-core query aims to find the k-core of

*The corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 5 ISSN 2150-8097.
doi:10.14778/3718057.3718063

Table 1: Examples of when query for the historical k-core. Note that, the words in square brackets can be replaced by other meaningful words in the context, which actually represent user-defined conditions.

Id	Semantics (given the start time t_s and cohesiveness k)	Other Input	Output
q1	When will the core contain the given [vertices]?	A vertex set V	The earliest end time t_e s.t. $V \subseteq C^k_{[t_s,t_e]}$
q2	When will the core grow to the given [size]?	An integer s	The earliest end time t_e s.t. $\lvert C^k_{[t_s,t_e]}\rvert \geq s$
q3	When will the core have the [greatest] [average degree]?	None	The end time t_e s.t. $\overline{\deg}\{v \lvert v \in C^k_{[t_s,t_e]}\}$ is maximized
q4	In what time period will the core grow [most] [rapidly]?	None	The pair (t_e, t'_e) s.t. $\frac{\lvert C^k_{[t_s,t'_e]}\rvert - \lvert C^k_{[t_s,t_e]}\rvert}{t'_e - t_e}$ is maximized

the historical subgraph during a given time interval $[t_s, t_e]$, namely, a set of vertices that induce the maximal subgraph in which 1) each edge contains a timestamp $t \in [t_s, t_e]$ and 2) each vertex has at least k neighbor vertices. By leveraging a precomputed PHC-index, the PHC-query deals with any historical k-core query in $O(|\mathcal{V}| \cdot \log \bar{t})$ time, where $|\mathcal{V}|$ is the total number of vertices and $\bar{t}$ is a bounded parameter that is determined by the specific temporal graph. Thus, the PHC-query is still costly for temporal graphs with great $|\mathcal{V}|$.

To improve the query efficiency, inspired by the concept of k-shell [10] (as illustrated in Figure 1a), we propose a novel index called Merged Core Time Shell index (MCTS-index). Preliminarily, we consider each temporal edge as a fact that will not be denied (namely, removed) over time. Given a start time t_s, since the coreness of a vertex in the historical subgraph of the time interval $[t_s, t_e]$ increases monotonically with the increase of the end time t_e, the earliest end time at which a vertex can have a specific coreness is called "core time". We put the vertices with the same core time for specific k and t_s into a "core time shell". For example, Figure 1b illustrates the core time shells in a temporal graph for $k = 2$ and $t_s = 2$, which comprise a hierarchical structure. Thus, it is optimal to scan the hierarchical structure to find the vertices with the core time no later than the given end time t_e for addressing the historical k-core query. Based on that, for each k, the MCTS-index further compresses the hierarchical structures with different t_s into a logical graph structure, which still preserves the traversal order of vertices in the hierarchical structures for each t_s while avoiding to keep repeated predecessor-successor relationships between vertices in different hierarchical structures. Due to the discreteness of vertex core time evolution in real-world temporal graphs, the MCTS-index reduces the overall space overhead of storing core time shells significantly, and thereby can scale to large graphs.

Theoretically, the MCTS-index has approximately the same space complexity as the PHC-index, but allows the nearly optimal query processing. By introducing another parameter $\bar{l}$ with the upper bound $3\bar{t}$, the MCTS-index based query algorithm called MCTS-query only needs $O(|C^k_{[t_s,t_e]}| \cdot \log \bar{l})$ time to find the result $C^k_{[t_s,t_e]}$, which is usually a small subset of $\mathcal{V}$. Thus, the MCTS-query approximates the optimal time complexity $O(|C^k_{[t_s,t_e]}|)$ by a factor of $\log \bar{l}$, which varies in the range of $[1.56, 8.18]$ in our empirical studies on ten real-world temporal graphs listed in Table 3. In practice, the MCTS-query is guaranteed to be more efficient than the PHC-query, unless the historical k-core is almost as large as the entire graph. Crucially, this advantage on query processing is not gained at the unreasonable expense of space, because the space complexity of the MCTS-index is limited to $\bar{l}/\bar{t}$ (in the emprirical range of $[1.36, 2.48]$) times the space complexity of the PHC-index.

Moreover, the MCTS-index can efficiently address the brand-new "when" historical k-core query with versatile semantics. The current historical graph queries, such as the k-core query [34] studied in this paper, the reachability/connectivity query [25, 33], and the connected component query [30, 32], intrinsically belong to the What-Query model. While, for temporal graphs, another query model orthogonal to the What-Query model, namely, the When-Query model that focuses on the time instants in which a result like a path or a subgraph appeared is grossly ignored [21]. For example, Table 1 shows several examples of the "when" historical k-core query. Given a start time and a particular condition of the historical k-core, the queries aim to find the specific end time at which the condition can be satisfied. As a unified pipeline to deal with such queries, we can incrementally maintain the historical k-core for given k and t_s (namely, traverse the core time shells in the ascending order of core time iteratively) using the MCTS-index. When evaluating the condition, we can stop early like the MCTS-query as long as the condition is monotonic on time.

In summary, we have the following contributions.

1) We propose a new state-of-the-art index approach to address the emerging historical k-core query on temporal graphs. The analysis of the parameterized complexity guarantees that our approach is more efficient than the existing PHC-query if the number of result vertices is less than $1/(\log_{\bar{t}} \bar{l} + 1)$ of the total number of vertices, which are generally satisfied by real-world queries due to $\bar{l} \leq 3\bar{t}$. Meanwhile, our approach requires only $\bar{l}/\bar{t}$ times as much space complexity as the PHC-index.

2) At the heart of our index design, a novel concept called "core time shell" is formulated. Similar to the k-shells of static graphs, the core time shells of temporal graphs will facilitate not only the current "what" historical k-core query but also the potential "when" historical k-core query. To the best of our knowledge, we are the first to present the formal query model for the "when" historical k-core query. Also, we present an algorithm framework that leverages our index to address a variety of "when" query templates efficiently.

3) To construct the index, we develop an algorithm to obtain it from the PHC-index without notable space or time overheads. The algorithm only "moves" the vertices to a new core time shell once for each of its distinct core time, thereby both avoiding to construct any core time shell in advance and reducing the construction time. In addition, we introduce a simple heuristics of ranking the vertices in core time shells, in order to reduce $\bar{l}$.

4) Lastly, we conduct comprehensive experiments on ten real-world temporal graphs. Compared to the PHC-query, our approach reduces the response time for the "what" queries with most possible parameters by a factor of several to tens of thousands, while our index only expands the space by a factor of 2.16 and spends 0.82% extra construction time on average. Moreover, we demonstrate the versatility of our approach with four case studies of the "when" queries, which reveal interesting insights.

2 PRELIMINARIES

2.1 Problem Definition

The *temporal graph* we study is an undirected multigraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, in which each edge $(u, v, t) \in \mathcal{E}$ means there is a connection between the vertices $u, v \in \mathcal{V}$ at time $t \in \{1, 2, \cdots, t_{max}\}$. Figure 1 illustrates a toy temporal graph as a running example. The integers on edges represent the timestamps. There could be parallel edges with different timestamps between a pair of vertices.

Building upon the above concept, we introduce the *projected graph* as a snapshot of the temporal graph. The projected graph $\mathcal{G}_{[t_s,t_e]}$ from the start time t_s until the end time t_e is defined as the subgraph of the temporal graph induced by all edges with timestamps in the range of $[t_s, t_e]$. For example, given a time interval $[2, 3]$, the projected graph $\mathcal{G}_{[2,3]}$ of our running example is comprised of the vertices and edges marked by bold lines in Figure 1b.

Recently, as a fundamental tool of exploring densely connected subgraphs during the history of temporal graphs, the *historical k-core query* [34] is proposed. It aims to find the k-core, the maximal subgraph in which each vertex has degree (namely, the number of distinct neighbor vertices but not parallel edges) at least k, emerging in a specific time period. The formal definition is as follows.

Definition 2.1 (Historical k-Core Query). For a temporal graph $\mathcal{G}$, given an integer k and a time interval $[t_s, t_e]$, the historical k-core query returns the set of vertices contained by the k-core of the projected graph $\mathcal{G}_{[t_s,t_e]}$, which is denoted by $C^k_{[t_s,t_e]}$.

For example, the historical 2-core of the time interval $[2, 3]$, namely, $C^2_{[2,3]}$ is comprised of the vertices v_1, v_2, and v_4 marked by grey color in Figure 1b. In contrast, the static 2-core (which can be seen as the historical 2-core during $[1, 7]$) is comprised of the vertices marked by grey color in Figure 1a.

2.2 State-of-the-art Approach

In order to address the historical k-core query, the PHC-query [34] is proposed recently. Specifically, the PHC-query decomposes the historical k-core query into $|\mathcal{V}|$ sub-queries called historical k-core containment queries, each of which is to determine whether a vertex $u \in \mathcal{V}$ belongs to the result $C^k_{[t_s,t_e]}$.

Then, to address a historical k-core containment query efficiently, the PHC-index is designed based on the following vital concept.

Definition 2.2 (Core Time). For a temporal graph $\mathcal{G}$, given an integer k and a start time t_s, the core time $\mathcal{CT}(u, k, t_s)$ of a vertex $u \in \mathcal{V}$ is the earliest end time t_e at which the coreness of u (namely, the greatest integer k' such that u belongs to the k'-core) is no less than k in the projected graph $\mathcal{G}_{[t_s,t_e]}$.

Algorithm 1: PHC-query($\mathcal{V}$, k, t_s, t_e)

```
1 C^k_[ts,te] ← ∅ ;                          //prepare an empty result set
2 forall u ∈ V do                            //traverse each vertex
3     obtain CT(u, k, ts) from the PHC-index by a binary search;
4     if CT(u, k, ts) ≤ te then              //check the containment
5         C^k_[ts,te] ← C^k_[ts,te] ∪ {u} ;  //add the contained vertex
6 return C^k_[ts,te]
```

Intuitively, for a vertex, each core time represents a "watershed moment" at which it joins the historical k-core expanding from a certain start time. For example, Figure 2a illustrates the evolving coreness of v_1 for our running example. Each cell in row t_s and column t_e indicates the coreness of v_1 in the projected graph of $[t_s, t_e]$. We can see that, there are continuous boundaries (denoted by dotted lines) between the cells with different coreness, which show the monotonicity of coreness in relation to time. Thus, for preserving the evolving coreness of a given vertex compactly, we only need to record the coordinates of "landmarks" (namely, the cells marked by boxes) for each boundary. Consider the red boundary separating the cells with the coreness 1 and 2. The first "landmark" (namely, red box) is located at $[1, 3]$, which means the core time $\mathcal{CT}(v_1, 2, 1) = 3$. Since the next red box is located at $[3, 6]$, it can be inferred that the core time $\mathcal{CT}(v_1, 2, 2)$ for the start time $t_s = 2$ is also 3. Note that, the last red box with an X mark inside means that, if $k = 2$ and the start time $t_s \geq 4$, the core time of v_1 does not exist (denoted by ∞ in particular).

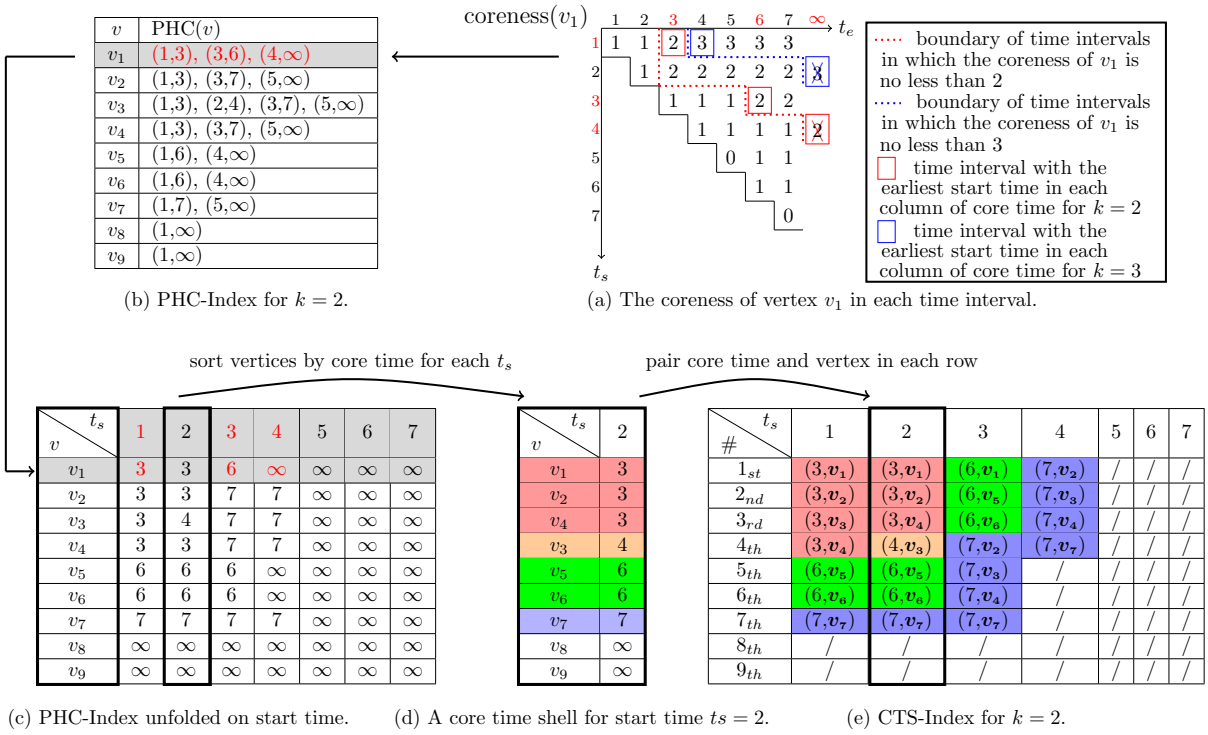
Based on the above observation, the PHC-index precomputes and records the landmark time intervals for each vertex and each reasonable value of k, as illustrated in Figure 2b. Then, a historical k-core containment query can be answered by comparing the retrieved/inferred core time $\mathcal{CT}(u, k, t_s)$ and the given end time t_e. If $\mathcal{CT}(u, k, t_s) \leq t_e$, we have $u \in C^k_{[t_s,t_e]}$.

Thus, the PHC-query addresses the historical k-core containment queries for each vertex and collect the result vertices. Algorithm 1 presents the pseudo code. Its time complexity is $O(|\mathcal{V}| \cdot \log \bar{t})$, where $\bar{t}$ is the average number of time intervals in entries of the PHC-index (which is also the average number of distinct core time of all vertices). Thus, the PHC-query is normally more efficient than the traditional core decomposition algorithm [8] with the time complexity $O(|\mathcal{E}|)$, because $|\mathcal{E}|$ is at least one order of magnitude greater than $|\mathcal{V}|$ for real-world temporal graphs.

Note that, although the index-free OTCD algorithm [31, 37] can process a batch of "temporal k-core queries" within a specific time range more efficiently than the PHC-query by effectively reducing the redundant computation, the PHC-query is still the state-of-the-art index-based approach for processing an individual historical k-core query.

3 INDEX STRUCTURE & QUERY PROCESSING

Since the time complexity of the PHC-query is linear to the total number of vertices $|\mathcal{V}|$ in the entire graph, it may result in low query efficiency on large-scale graphs. For real-world historical k-core queries such as finding influential users or fraud groups, the result set of vertices is often significantly smaller than $\mathcal{V}$, especially



v	PHC(v)
v_1	(1,3), (3,6), (4,∞)
v_2	(1,3), (3,7), (5,∞)
v_3	(1,3), (2,4), (3,7), (5,∞)
v_4	(1,3), (3,7), (5,∞)
v_5	(1,6), (4,∞)
v_6	(1,6), (4,∞)
v_7	(1,7), (5,∞)
v_8	(1,∞)
v_9	(1,∞)

(b) PHC-Index for $k = 2$.

(a) The coreness of vertex v_1 in each time interval.

v \ t_s	1	2	3	4	5	6	7
v_1	3	3	6	∞	∞	∞	∞
v_2	3	3	7	7	∞	∞	∞
v_3	3	4	7	7	∞	∞	∞
v_4	3	3	7	7	∞	∞	∞
v_5	6	6	6	∞	∞	∞	∞
v_6	6	6	6	∞	∞	∞	∞
v_7	7	7	7	7	∞	∞	∞
v_8	∞	∞	∞	∞	∞	∞	∞
v_9	∞	∞	∞	∞	∞	∞	∞

(c) PHC-Index unfolded on start time.

v \ t_s	2
v_1	3
v_2	3
v_4	3
v_3	4
v_5	6
v_6	6
v_7	7
v_8	∞
v_9	∞

(d) A core time shell for start time $ts = 2$.

# \ t_s	1	2	3	4	5	6	7
1_{st}	(3,v_1)	(3,v_1)	(6,v_1)	(7,v_2)	/	/	/
2_{nd}	(3,v_2)	(3,v_2)	(6,v_5)	(7,v_3)	/	/	/
3_{rd}	(3,v_3)	(3,v_4)	(6,v_6)	(7,v_4)	/	/	/
4_{th}	(3,v_4)	(4,v_3)	(7,v_2)	(7,v_7)	/	/	/
5_{th}	(6,v_5)	(6,v_5)	(7,v_3)	/	/	/	/
6_{th}	(6,v_6)	(6,v_6)	(7,v_4)	/	/	/	/
7_{th}	(7,v_7)	(7,v_7)	(7,v_7)	/	/	/	/
8_{th}	/	/	/	/	/	/	/
9_{th}	/	/	/	/	/	/	/

(e) CTS-Index for $k = 2$.

Figure 2: A chain-of-thought demonstrating the design of the CTS-index.

when k is relatively large or the time span $(t_e - t_s)$ is short. This is highly reasonable, because with the increase of k or the decrease of $(t_e - t_s)$, the number of vertices satisfying the k-core condition decreases rapidly in the sparse real-world temporal graphs.

In this section, we design index structures to improve the efficiency of historical k-core query processing. Firstly, we present an intermediate Core Time Shell index that provides the optimal query performance at the expense of high space complexity in Sections 3.1. Then, we present a space-time balanced Merged Core Time Shell index in Section 3.2. Lastly, we compare the complexities of the PHC, CTS, and MCTS indexes in Section 3.3.

3.1 Core Time Shell Index

Inspired by the concept of k-shell [10], we propose a new Core Time Shell index (CTS-index) that enhances the existing PHC-index to achieve the optimal query performance.

3.1.1 Structure. In the domain of network analysis, the *k-shell* derived from the k-core is a pivotal concept for uncovering the hierarchical organization of vertices. A vertex belongs to the k-shell if and only if it is in the k-core but not in the $(k+1)$-core, namely, its coreness is exactly k. For an entire graph, the vertices are assigned to non-overlapping shells with different values of k. Consequently, the k-shell structure consisting of all k-shells ordered by k can solve a k-core query in the optimal $O(|C^k|)$ time, where C^k is the vertex set of the k-core. Figure 1a illustrates an example of the k-shell structure.

Actually, the PHC-index applies a particular (k, u, t_c)-shell that preserves a set of start time $\{t_s\}$ with $\mathcal{CT}(u, k, t_s) = t_c$ but not vertices. Figure 2a illustrates the distribution of the coreness of the vertex v_1 in the projected graphs of all time intervals. Given $k = 2$, for each distinct core time of v_1 (namely, t_e in red color) like 3, there is a $(2, v_1, 3)$-shell $\{1, 2\}$. The PHC-index compresses each (k, u, t_c)-shell by only recording the earliest start time (namely, t_s in red color), because the timestamps in the (k, u, t_c)-shell are continuous. As illustrated in Figure 2b, each entry of the PHC-index is a list of compressed (k, u, t_c)-shells for each distinct t_c.

It naturally raises an interesting question: can we design another kind of shell that still preserves vertices to address historical k-core queries, so that we do not have to traverse all vertices for each query like the PHC-index? The answer is yes. Conceptually, we can fulfill that by swapping the dimensions of u and t_s in the (k, u, t_c)-shell by two steps. Firstly, we unfold the PHC-index in the compressed t_s dimension, as illustrated in Figure 2c. Then, we sort the vertices by their core time for each start time t_s independently, as illustrated in Figure 2d. The result of each sort is a hierarchical structure of vertices, in which each layer (marked by different colors) refers to a distinct core time of these vertices for corresponding k and t_s. For example, the red layer has three vertices v_1, v_2, and v_4, and only they have the core time $t_c = 3$ for $k = 2$ and $t_s = 2$.

Consequently, we have a new (k, t_s, t_c)-shell called core time shell. It is formally defined as follows.

Definition 3.1 (Core Time Shell). For a temporal graph $\mathcal{G}$, given an integer k, a start time t_s, and any time $t_c \geq t_s$, the core time

Algorithm 2: CTS-query(k, t_s, t_e)

```
1 C^k_[ts,te] ← ∅ ;                                  //prepare an empty result set
2 forall (k, ts, tc)-shell in the retrieved (k, ts)-structure do
3     if tc > te then                                //the core time of the next shell is too late
4         break ;                                    //stop early
5     else
6         forall vertex u in the (k, ts, tc)-shell do
7             C^k_[ts,te] ← C^k_[ts,te] ∪ {u} ;      //add vertices in the shell
8 return C^k_[ts,te];
```

shell denoted by (k, t_s, t_c)-shell is a set of vertices $V \subseteq \mathcal{V}$ such that a vertex $u \in V$ if and only if $\mathcal{CT}(u, k, t_s) = t_c$.

The hierarchical structure consisting of core time shells ordered by the core time reveal in which time or order the vertices join the historical k-core since t_s. Figure 1b illustrates the core time shells with $k = 2$ and $t_s = 2$ for our running example. The four shells correspond to the four layers in Figure 2d color by color.

In order to store and retrieve the hierarchical structures, we propose the Core Time Shell index (CTS-index) as follows.

Definition 3.2 (Core Time Shell Index). For a temporal graph $\mathcal{G}$, the CTS-index is comprised of entries for each combination of $k \in [2, k_{max}]$ and $t_s \in [1, t_{max}]$, which are denoted by (k, t_s)-structures. Each (k, t_s)-structure is a core time shell list containing all nonempty (k, t_s, t_c)-shells in the ascending order of t_c.

For example, Figure 2e illustrates the (k, t_s)-structures with $k = 2$ and $t_s = 1, 2, \cdots, 7$ in the CTS-index as a table, in which each column represents a (k, t_s)-structure. The (2,1)-structure in the first column has three core time shells: (2,1,3), (2,1,6), and (2,1,7)-shells distinguished by different colors. For ease of understanding, we record a two-tuple with both core time and vertex in each cell, though we only need to record a core time for an entire core time shell. Note that, for vertices like v_8 and v_9 that are not in the 2-core of $\mathcal{G}_{[2,t_{max}]}$ (whose core time is denoted by ∞), they will not appear in any core time shell of the $(2, 2)$-structure.

3.1.2 Query Processing. Algorithm 2 presents the pseudo code of the CTS-query, which uses the CTS-index to address the historical k-core query. Given k and $[t_s, t_e]$, the CTS-query retrieves the (k, t_s)-structure from the CTS-index and then traverses the nested (k, t_s, t_c)-shells in the ascending order of t_c (lines 2-7). As long as $t_c \leq t_e$, the vertices in the traversed (k, t_s, t_c)-shells are added into the result (lines 6-7). Otherwise, the traversal is terminated immediately (lines 3-4). The correctness of the CTS-query is guaranteed by the definition of core time (Definition 2.2).

For example, given $k = 2$ and $[t_s, t_e] = [2, 5]$, we will traverse the vertices v_1, v_2, v_4, and v_3 sequentially until the core time of next shell is greater than 5, as illustrated in Figure 2d.

3.1.3 Complexity. Let us consider the time complexity of the CTS-query firstly.

Theorem 3.1. *The CTS-query can solve the historical k-core query with the time complexity $O(|C^k_{[t_s,t_e]}|)$, where $|C^k_{[t_s,t_e]}|$ is the size (namely, the number of vertices) of the result historical k-core.*

Proof. The CTS-query is simply traversing the vertices in the (k, t_s)-structure. Due to the order of traversal, it will not meet any vertex not in the result until termination. □

Then, let us consider the space complexity of the CTS-index.

Theorem 3.2. *The space complexity of the CTS-index is bounded by $O(k_{max} \cdot |\mathcal{V}| \cdot t_{max})$, where $|\mathcal{V}|$ is the number of vertices, k_{max} is the maximum coreness, and t_{max} is the maximum timestamp.*

Proof. According to Definition 3.2, for each $k \in [2, k_{max}]$ and each $t_s \in [1, t_{max}]$, a (k, t_s)-structure is stored, so that there are $k_{max} \cdot t_{max}$ (k, t_s)-structures. Within each (k, t_s)-structure, there are a number of (k, t_s, t_c)-shells. Since a vertex has only one or none core time for given k and t_s, it is contained by at most one (k, t_s, t_c)-shell. Thus, the space complexity of a (k, t_s)-structure is bounded by $O(|\mathcal{V}|)$. Consequently, the overall space complexity of the CTS-index is bounded by $O(k_{max} \cdot |\mathcal{V}| \cdot t_{max})$. □

In summary, the CTS-query achieves the theoretically optimal time complexity, and however the CTS-index has a step back from the PHC-index in terms of space complexity because the CTS-index abandons the compression used by the PHC-index.

3.2 Merged Core Time Shell Index

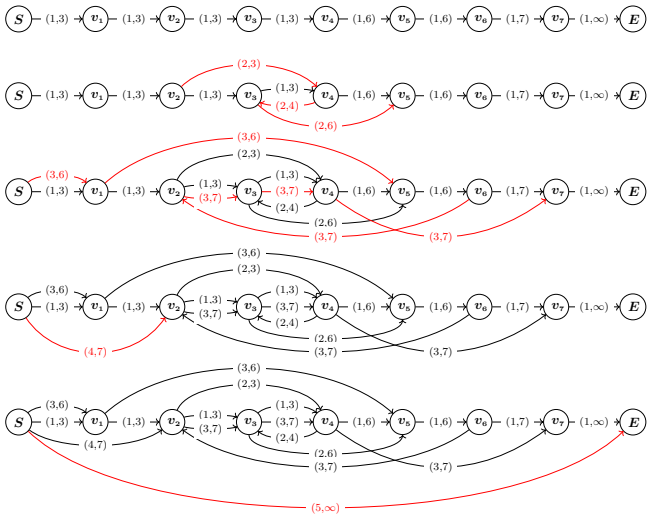
As the CTS-index is too costly for large-scale temporal graphs with massive vertices and timestamps, we propose the Merged Core Time Shell index (MCTS-index) that improves the CTS-index to achieve a better balance between query time and index space.

3.2.1 Structure. As indicated by its name, the main idea of the MCTS-index is to merge the (k, t_s)-structures with different t_s for each k, thereby reducing the space overhead. Since each (k, t_s)-structure can be seen as a specific sequence of identical vertices, we can merge the sequences incrementally by delta-encoding, which preserves a new sequence as only a number of "edits" on the previously merged result.

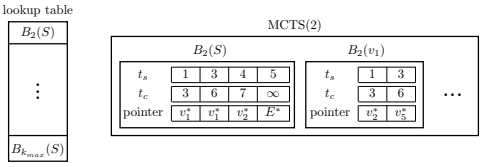
Specifically, given k, the merging procedure of (k, t_s)-structures has the following two steps.

Step 1: representation. We represent each (k, t_s)-structure as a **directed link-labeled path** denoted by P_{k,t_s}, on which there is a node for each vertex in the (k, t_s)-structure. In addition, a path always has a start node and an end node denoted by S and E respectively. Then, different from the edges in the temporal graph, the directed links on the path P_{k,t_s} indicate the predecessor-successor relationships between nodes in the corresponding (k, t_s)-structure, so that the order of traversing nodes from S to E along the path P_{k,t_s} is the same as that of traversing the vertices in the (k, t_s)-structure sequentially. The label of link is a two-tuple (t_s, t_c), which means the successive vertex has the core time t_c for the start time t_s and k. Thus, we denote by (n, n', t_s, t_c) a directed link on the path with $n, n' \in \{S, E\} \cup \mathcal{V}$.

Step 2: merging. We merge the paths P_{k,t_s} into a single **merged graph** for specific k in the ascending order of $t_s \in \{1, 2, \cdots, t_{max}\}$. The simplest merge function is to add all links on a path into the merged graph. However, the space complexity of such a merged graph is the same as that of the CTS-index. Thus, a merge function should add a link into the merged graph only if it is necessary, namely, the merged graph can still preserve the traversal order and



(a) The logical procedure of constructing MCTS(2).



(b) The physical structure of constructed MCTS(2).

Figure 3: An example of the MCTS-index with $k = 2$.

the core time of vertices on each path without this link. For that, we propose a merge function that adds a link (n, n', t_s, t_c) if 1) n and n' do not have the predecessor-successor relationship in the previous path P_{k,t_s-1}, or 2) they do have but the core time of n' in P_{k,t_s-1} is earlier than t_c. Consequently, only a few links will be added if the (k, t_s)-structure is similar to the $(k, t_s - 1)$-structure.

The details of the proposed merge function are given in Algorithm 3. We denote by N_k and L_k the node set and link set of the merged graph for specific k respectively. Firstly, we add all nodes as well as links in $P_{k,1}$ to N_k (lines 1-2), since it can be proved that the vertices in any (k, t_s)-structure are surely contained by the $(k, 1)$-structure. Then, we traverse the links in each path P_{k,t_s} with $t_s > 1$ (lines 4-13). For each link (n, n', t_s, t_c), we determine whether to add it to L_k with respect to certain conditions (lines 7-12), so that the order and core time of vertices in any (k, t_s)-structure can be preserved.

Lemma 3.1. In the merged graph for specific k, given a start time t_s, for any node n in the path P_{k,t_s} except the last one E, its outgoing link $(n, n', t'_s, t_c) \in L_k$ with the maximum $t'_s \le t_s$ reaches another node n' (called the *latest successor* of n) that is the successor of n in the path P_{k,t_s}, and we have $\mathcal{CT}(n', k, t_s) = t_c$.

Proof. Given n and t_s, there are two cases, namely, $t'_s = t_s$ and $t'_s < t_s$ for the link $(n, n', t'_s, t_c) \in L_k$ with the maximum $t'_s \le t_s$. We will discuss both cases. If $t'_s = t_s$, this link is added from the path

Algorithm 3: Path-Merge(N_k, L_k, P_{k,t_s})

```
1  if t_s = 1 then                          //the first path is preserved completely
2      add all nodes and links in P_{k,t_s} to N_k and L_k respectively;
3  else                                     //the other paths are preserved incrementally
4      n ← S ;                              //start the traversal in the path
5      while n ≠ E do
6          (n, n', t_s, t_c) ← the link outgoing from n in P_{k,t_s} ;   //get
             the successor of n in the path
7          (n, n'', t'_s, t'_c) ← the link outgoing from n in L_k with the
             greatest t'_s ;   //get the latest successor of n in the merged
             graph
8          if n'' ≠ n' then                 //the successors are different
9              add (n, n', t_s, t_c) to L_k ;          //record the change
10         else
11             if t'_c < t_c then           //the core time is different
12                 add (n, n', t_s, t_c) to L_k ;      //record the change
13         n ← n' ;                         //traverse the next node in the path
14 return N_k, L_k;
```

P_{k,t_s}, so that n' must be the successor of n and the core time of n' is certainly t_c in P_{k,t_s}. If $t'_s < t_s$, we can prove this by induction. For $t''_s = t'_s + 1$, the link (n, n'', t''_s, t'_c) outgoing from n in the path P_{k,t''_s} is not added to the merged graph only because the successor $n'' = n'$ and its core time $t'_c = t_c$ according to Algorithm 3. Consequently, the conclusion also holds for any other $t''_s \in (t'_s + 1, t_s]$. □

For example, Figure 3a illustrates the procedure of constructing the merged graph for the running example temporal graph with $k = 2$. Each row is the merged graph after gradually merging the path P_{2,t_s} with $t_s = 1$, 2, 3, 4, and 5 (namely, the corresponding column in Figure 2e). The links marked by red color are newly added from the corresponding path. In $P_{2,2}$, the vertex following the vertex v_2 is v_4 with the core time 3 but not v_3, so that a new link $(v_2, v_4, 2, 3)$ is added into the merged graph in the second row. Moreover, in $P_{2,3}$, the vertex v_2 is followed by the vertex v_3 with the core time 7, so that a new link $(v_2, v_3, 3, 7)$ is added into the merged graph in the third row though there already exists a link $(v_2, v_3, 1, 3)$. Lastly, since $P_{2,5}$ is empty, a new link $(S, E, 5, \infty)$ is added into the merged graph in the fifth row in order to declare that there is no historical 2-core if the start time $t_s \ge 5$. By statistics, the final merged graph has 19 links, and in contrast, the original paths have 32 links in total.

Note that, the compression ratio of the merged graph depends on how discretely the core time of vertices evolves with the increasing start time. The more discretely the core time evolves, the more similar the (k, t_s)-structure and the $(k, t_s + 1)$-structure are, and the better the compression ratio is.

Then, we formally define the MCTS-index as follows.

Definition 3.3 (Merged Core Time Shell Index). For a temporal graph $\mathcal{G}$, the MCTS-index is composed of graph structures for each possible k, which are denoted by MCTS(k). For each MCTS(k) = (N_k, L_k), the set N_k is the union of nodes on the paths P_{k,t_s} with $t_s = 1, 2, \cdots, t_{max}$, and the set $L_k = \{(n, n', t_s, t_c)\}$ with $n, n' \in N_k$

Algorithm 4: MCTS-query(k, t_s, t_e)

```
1  C^k_[t_s,t_e] ← ∅, n ← S ;                    //initialization and start from S
2  while n ≠ E do                    //traverse vertices until reaching the end node
3      obtain A_{t_s}, A_{t_c}, and A_n from B_k(n) in MCTS(k);
4      i ← arg max_i {A_{t_s}[i] | A_{t_s}[i] ≤ t_s} ;    //get the latest successor
5      if A_{t_c}[i] ≤ t_e then                              //check the core time
6          C^k_[t_s,t_e] ← C^k_[t_s,t_e] ∪ {A_n[i]} ;                //add the vertex
7          n ← A_n[i] ;                                  //jump to the latest successor
8      else                                  //the core time of the next shell is too late
9          break ;                                                    //stop early
10 return C^k_[t_s,t_e];
```

is a subset of all links on the paths such that we can restore the (k, t_s)-structure for any t_s from the merged graph MCTS(k).

Lastly, we physically store the MCTS-index in a compact and query-friendly format, as illustrated in Figure 3b. Each MCTS(k) is stored as an adjacency list that represents its merged graph. Specifically, each node n is stored as a block $B_k(n)$ that contains three arrays A_{t_s}, A_{t_c}, and A_n of the same length, which record t_s, t_c, and the pointer to the block $B_k(n')$ for its outgoing links $\{(n, n', t_s, t_c)\}$ respectively. All arrays are sorted in the ascending order of the corresponding t_s, so that MCTS(k) supports the retrieval of the latest successor efficiently. Given t_s, we perform a binary search in A_{t_s} to find the maximum start time less than t_s. Then, we use the index of the returned start time in A_{t_s} to get the corresponding core time and successor node in the other two arrays respectively. The pointers to the blocks of start nodes of each MCTS(k) are stored in a lookup table, which serves as the entrance of the MCTS-index.

3.2.2 Query Processing. We propose a query processing algorithm called MCTS-query based on the MCTS-index. It directly search the merged graph and does not need to restore the (k, t_s)-structure from the MCTS-index.

Algorithm 4 presents the pseudo code of the MCTS-query. Given an integer k and a time interval $[t_s, t_e]$ as inputs, we traverse the nodes in the merged graph of MCTS(k) from the node S to return $C^k_{[t_s,t_e]}$. For each traversed node, we choose its latest successor as the next node (lines 3-7), until the core time of the latest successor is later than the given end time t_e (lines 8-9). The result set $C^k_{[t_s,t_e]}$ is comprised of all traversed vertices.

The correctness of the MCTS-query is obvious. Since it traverses the vertices along the links to the latest successor, the order of traversal is the same as that of traversing the corresponding (k, t_s)-structure according to Lemma 3.1. Meanwhile, it also stops if the core time of the next vertex is greater than the given end time. Thus, it finds the same result as the CTS-query.

For example, given $k = 2$ and $[t_s, t_e] = [2, 3]$, we use the MCTS-query to address the historical k-core query on the merged graph of MCTS(2) illustrated in Figure 3a. For the initial node S, there are four links: $(S, v_1, 1, 3)$, $(S, v_1, 3, 6)$, $(S, v_2, 4, 7)$, and $(S, E, 5, \infty)$. Thus, we traverse along $(S, v_1, 1, 3)$ because its start time 1 is the maximum time no later than 2 and its core time 3 is no later than 3, and v_1 is added into the result set. For the next node v_1, there are two links: $(v_1, v_2, 1, 3)$ and $(v_1, v_5, 3, 6)$. Thus, we traverse along $(v_1, v_2, 1, 3)$ to v_2. Similarly, we will further traverse along $(v_2, v_4, 2, 3)$ to v_4. Then, the next link $(v_4, v_3, 2, 4)$ to the latest successor is not qualified because its core time 4 is later than 3, and thereby the traversal is terminated immediately. The final result set is comprised of v_1, v_2, and v_4, which can be verified in both Figure 1b and Figure 2e.

3.2.3 Complexity. Firstly, let us consider the space complexity of the MCTS-index. Since each MCTS(k) is a graph structure, we can obtain its space complexity by counting its links.

Theorem 3.3. *By introducing a parameter $\bar{l}$ that denotes the average number of links outgoing from a node in MCTS(k) for $k \in [2, k_{max}]$, the space complexity of the MCTS-index is bouned by $O(k_{max} \cdot |\mathcal{V}| \cdot \bar{l})$.*

Proof. Obviously, there are $k_{max} - 1$ merged graphs, each of which has at most $|\mathcal{V}| + 1$ nodes with $\bar{l}$ links. □

More importantly, the correlation of $\bar{l}$ and $\bar{t}$, which denotes the average number of distinct core time of a vertex for $k \in [2, k_{max}]$, can be established, so that the complexities of the MCTS-index and the PHC-index are comparable.

Theorem 3.4. *Given a global order to rank vertices in any core time shell, we have $\bar{l} \leq 3\bar{t}$ if the merge function is Algorithm 3.*

Proof. Given a global order like the ascending vertex id, the positions of vertices in each core time shell are fixed. Thus, for the merged graph that has merged the paths $\{P_{k,t_s}\}$ with $t_s \in [1, t-1]$, a link (n, n', t, t_c) in the path $P_{k,t}$ will be added into the merged graph only if 1) the core time of n for the start time t is increased (namely, n comes from another core time shell), 2) the core time of n' for the start time t is increased, (namely, n' comes from another core time shell) or 3) the core time of the nodes between n and n' in the path $P_{k,t-1}$ but not n and n' themselves is increased (namely, n and n' are still in their old core time shells but the nodes that used to be between them are gone). Thus, when merging a path P_{k,t_s}, we only add the links that satisfy at least one of the three conditions. Conversely, for each vertex whose core time is increased to a new distinct timestamp, we have at most one new link that can satisfy each condition. Consequently, the (average) number of links is at most three times the (average) number of distinct core time. □

Moreover, the time complexity of the MCTS-query (Algorithm 4) can be estimated as follows.

Theorem 3.5. *The MCTS-query can solve the historical k-core query with the time complexity $O(|C^k_{[t_s,t_e]}| \cdot \log \bar{l})$.*

Proof. For a historical k-core query of $[t_s, t_e]$, the MCTS-query will only traverse the vertices in the result $C^k_{[t_s,t_e]}$ like the CTS-query. Moreover, different from the CTS-query, the MCTS-query needs to find the latest successor by a binary search in the start time array A_{t_s}, whose length is $\bar{l}$ on average. Thus, the overall time complexity is $O(|C^k_{[t_s,t_e]}| \cdot \log \bar{l})$. □

Note that, $\log \bar{l}$ can be seen as the approximation ratio of the time complexity of the MCTS-query to the optimal complexity, which varies in the range of $[1.56, 8.18]$ in our empirical studies.

Table 2: Comparison of complexities.

	index space	query time
PHC	$O(k_{max} \cdot \lvert\mathcal{V}\rvert \cdot \bar{t})$	$O(\lvert\mathcal{V}\rvert \cdot \log \bar{t})$
MCTS	$O(k_{max} \cdot \lvert\mathcal{V}\rvert \cdot \bar{l})$	$O(\lvert C^k_{[t_s,t_e]}\rvert \cdot \log \bar{l})$
CTS	$O(k_{max} \cdot \lvert\mathcal{V}\rvert \cdot t_{max})$	$O(\lvert C^k_{[t_s,t_e]}\rvert)$

3.3 Comparison of PHC, CTS, and MCTS

Table 2 shows the parameterized complexities of all three indexes. For real-world data and queries, we generally have $\bar{l} \simeq \bar{t} \ll t_{max}$ and $|C^k_{[t_s,t_e]}| \ll |\mathcal{V}|$ (see the experimental results in Section 6).

3.3.1 Index Space. In terms of space complexity, the MCTS-index and the PHC-index are about the same, and both effectively compress the CTS-index in different ways. The CTS-index could be too costly on large-scale temporal graphs with great $|\mathcal{V}|$ and t_{max} (note that, k_{max} is relatively very small on the sparse real-world graphs). In contrast, the MCTS-index requires only $\bar{l}/\bar{t}$ (at most three) times as much index space as the PHC-index.

3.3.2 Query Time. For querying the historical k-core, the CTS-query is the most efficient, the MCTS-query is almost as efficient as the CTS-query, and the PHC-query is generally slower than them. The CTS-query can achieve the theoretically optimal time complexity, while the MCTS-query trades off limited performance for much less space. In contrast, the PHC-query needs to traverse all vertices no matter how small the result is.

In summary, the MCTS-index achieves a theoretically excellent balance between space overheads and query performance for dealing with historical k-core queries.

4 INDEX CONSTRUCTION

4.1 Naive Algorithm

Straightforwardly, we can obtain the MCTS-index from the constructed CTS-index. For each k, we construct MCTS(k) by merging the (k, t_s)-structures with $t_s = 1, 2, \cdots, t_{max}$ gradually. The details of merge function have been introduced in Section 3.2.

However, due to the high space overheads of the CTS-index, it is impractical to construct the MCTS-index on top of that for large-scale temporal graphs. As discussed in Section 3.3, the space complexity of the PHC-index is much lower than that of the CTS-index. This naturally raises the idea of obtaining the MCTS-index directly from the constructed PHC-index, thereby avoiding to construct the core time shells in advance.

4.2 Vertex Moving Algorithm

We develop an advanced construction algorithm for the MCTS-index, which directly obtain it from the space-efficient PHC-index. For each k, we still merge the (k, t_s)-structures into the merged graph MCTS(k) in the ascending order of t_s. However, due to the absence of the CTS-index, the (k, t_s)-structures are unknown. Thus, we obtain each (k, t_s)-structure from the previous $(k, t_s - 1)$-structure incrementally by "vertex movement". Specifically, we firstly retrieve the vertices whose core time t_c for the start time t_s is increased (with respect to that for $t_s - 1$) from the PHC-index, and then we move each vertex to the core time shell corresponding to its new core time t_c. For example, to obtain the $(2, 2)$-structure from the $(2, 1)$-structure in Figure 2e, it is needed to move v_3 to the $(2, 2, 4)$-shell, because the core time of v_3 for $t_s = 2$ is increased to 4 from 3.

Moreover, we propose an optimization method that can effectively reduce the overheads of merging the (k, t_s)-structures. Since a vertex movement will only change the predecessors and successors of a few of vertices in a temporary (k, t_s)-structure, we only need to add and remove links for each vertex movement to preserve the vertex traversal order for the (k, t_s)-structure, instead of traversing all links in its path P_{k,t_s} as in Algorithm 3.

The addition of links is based on the following observation.

Lemma 4.1. After moving a vertex into a new core time shell, it requires at most three new links to preserve the new predecessor-successor relationships in the merged graph.

Proof. Assume the vertex u has been moved into the (k, t_s, t_c)-shell. There are possibly three new predecessor-successor relationships. 1) For the old latest predecessor n and the old latest successor n' of u, we add a link $(n, n', t_s, \mathcal{CT}(v', k, t_s - 1))$ to connect them. 2) For the new latest predecessor n of u, we add a link (n, u, t_s, t_c) to connect it and u. 3) For the new latest successor n of u, we add a link $(u, n, t_s, \mathcal{CT}(n, k, t_s - 1))$ to connect it and u. In particular, if $t_c = \infty$ (namely, u has no core time for t_s), only one link is needed for the case 1), or if the old and new latest predecessors of u are the same node, only one link is needed for the case 2). □

Meanwhile, for each newly added link (n, n', t_s, t_c), we need to check whether there exists another link from n with the same t_s but different n' or t_c. If so, the old link has to be removed, because the latest successor of n has changed. Consequently, it is maintained a unique vertex traversal order for any given start time.

Algorithm 5 outlines the procedure of construction. We have two further explanations on this algorithm. 1) In order to retrieve the vertices with the increased core time for given t_s, we can easily reorganize the PHC-index to fulfill that in $O(1)$ time. 2) When moving a vertex, we heuristically move it to the last position of the destined core time shell, though any position is actually feasible. Interestingly, compared to maintaining a global order, we observed that the heuristics can leverage the correlations of core time between vertices to reduce the total number of links in the final merged graph.

For example, Figure 4 illustrates the procedure of updating the merged graph of MCTS(2) as t_s is increased to 3. For each movement, the red solid lines are newly added links and the red dashed lines are newly removed links. Firstly, v_1 is moved to the last position of the $(3, 6)$-shell. Thus, we add three links $(S, v_2, 3, 3)$, $(v_6, v_1, 3, 6)$, and $(v_1, v_7, 3, 7)$ to the merged graph with respect to the three cases in the proof of Lemma 4.1 respectively. Then, v_2 is moved to the last position of the $(3, 7)$-shell. We also add three links $(S, v_4, 3, 3)$, $(v_7, v_2, 3, 7)$, and $(v_2, E, 3, \infty)$. Moreover, the previous link $(S, v_2, 3, 3)$ is removed because it is replaced by $(S, v_4, 3, 3)$. Finally, after all four vertices with increased core time have been moved, we search the merged graph from S by traversing the latest successor node, and will obtain the $(2, 3)$-structure $\{\{v_5, v_6, v_1\}, \{v_7, v_2, v_4, v_3\}\}$, which is obviously correct.

Algorithm 5: MCTS-Construct(the PHC-index)

```
for k ← 2 to k_max do                          //start the construction of MCTS (k)
    retrieve the core time of all vertices for k and t_s = 1 from the
      PHC-index and obtain the path P_{k,1};
    initialize the merged graph of MCTS(k) to be P_{k,1};
    for t_s ← 2 to t_max do                    //start the merge of P_{k,t_s}
        retrieve the vertices with the increased core time for t_s
          from the PHC-index ;                 //implemented in O(1) time
        forall the vertex u with the increased core time t_c do
            simulate to move u to the last position of the
              (k, t_s, t_c)-shell and add links to the merged graph for
              recording new predecessor-successor relationships ;
              //Lemma 4.1
            for each new link (n, n', t_s, t'_c), remove the old link
              (n, n'', t_s, t''_c) with n' ≠ n'' or t'_c ≠ t''_c if it exists
```

Lastly, it can be proved that, the merged graph constructed by Algorithm 5 can also retain the property in Lemma 3.1, and the final MCTS-index has the same space complexity as the one constructed by Algorithm 3 due to Lemma 4.1, though the inner order of vertices in core time shells may be different. Meanwhile, since the number of vertices with increased core time is usually much less than the total number of vertices for a (k, t_s)-structure, Algorithm 5 can improve the efficiency of index construction significantly.

5 QUERY EXTENSION

In this section, we extend the semantics of the historical k-core query from "what" to "when", and demonstrate how the MCTS-index addresses this brand-new type of queries efficiently.

5.1 When-Query Model

Firstly, we formally give the following general definition of When-Query model in the context of querying historical k-cores.

Definition 5.1 (When Historical k-Core Query). For a temporal graph, given 1) the start time t_s and 2) another condition on a specific metric $f(\cdot)$ of historical k-cores, find one or more end time $\{t_e\}$ such that the metric values of historical k-cores $\{f(C^k_{[t_s,t_e]})\}$ satisfy the given condition.

The first condition (namely, starting from t_s) can avoid the unnecessary complexity in analysis. If we treat t_s as the output but not the input of queries, an arbitrary time interval will belong to the solution space at t^2_{max} scale. As a result, the query processing will be extremely costly, and the number of qualified time intervals could be large. However, our observation on temporal graphs often focuses on dynamics or evolution with respect to a time line [19], which means there should be a reference time and the observation starts from or ends at there. Consequently, we choose to query t_e for given t_s, which can be implemented by exploiting the core time shell to traverse the (k, t_s)-structure.

The second condition is flexible and can be defined according to application scenarios. On the one hand, the metric $f(\cdot)$ of historical k-cores is a user-defined function. For example, there are traditional non-temporal metrics [3], such as average degree, internal density, conductance, and clustering coefficient, and also temporal metrics

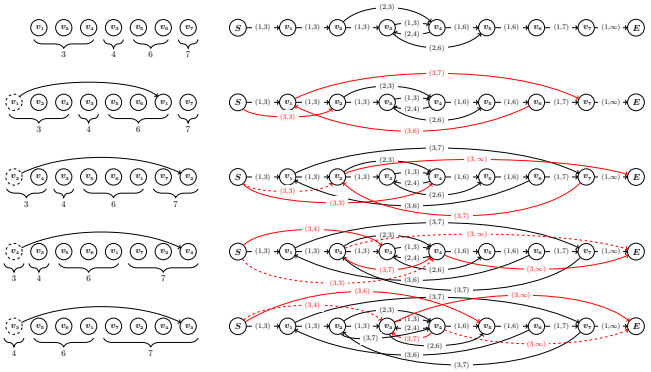


Figure 4: A step-by-step example of updating the merged graph (on the right) according to the heuristic vertex movement (on the left) when t_s is increased to 3.

such as interaction frequency [1, 29], persistence [15], burstiness [4, 22, 35], periodicity [23], continuity [17], and reliability [26, 27]. On the other hand, the condition itself could be either constraint or optimization on the metric. Obviously, such an abstract condition is general enough to express various when queries.

5.2 Typical When-Query Template

With Definition 5.1, we give four typical query templates and their examples respectively, as shown in Table 1.

1) Time of containment. This kind of queries aim to find the time of interested members to join an evolving community. For example, q1 requires the historical k-core to contain a given set of vertices. Moreover, we can compose more advanced examples like requiring it to contain a certain percentage of vertices in the set. Note that, the metric $f(\cdot)$ is an identity function in this case.

2) Time of satisfaction. This kind of queries aim to find the time of the historical k-core to gain an expected property. For example, q2 requires the historical k-core to have the given number of vertices. Other than the size, the metric can be defined arbitrarily as long as its value varies monotonically with t_e. Otherwise, the query processing cannot be stopped early before traversing the entire (k, t_s)-structure.

3) Time of being optimal. This kind of queries aim to find the time of the historical k-core to be optimal for an interested metric. For example, q3 requires the historical k-core to have the greatest average degree of its vertices. The average degree is usually not monotonic for t_e. Thus, q3 can be seen as an optimization task to find the densest moment of the historical k-core.

4) Time period of being optimal. This kind of queries aim to find the time period in which the historical k-core is optimal for a metric of interest. For example, q4 requires the historical k-core to grow most rapidly in the returned time period. Different from the previous queries, this condition involves the comparison of more than one historical k-cores, so that the result is a pair (which could be extended to a list) of timestamps.

The above four typical When-Query templates can be applied in many fields. Take the social network analysis as an example. Since the previous studies [10, 20] have revealed that the most influential

Algorithm 6: When-Query-Example(k, t_s, $|C^k_{[t_s,t_e]}| \geq s$)

```
1  C^k_[ts,tc] ← ∅, n ← S, tc ← 0 ;          //initialization and start from S
2  while n ≠ E do                            //traverse vertices until reaching the end node
3      obtain A_ts, A_tc, and A_n from B_k(n) in MCTS(k);
4      i ← arg max_i {A_ts[i] | A_ts[i] ≤ ts} ;   //get the latest successor
5      if A_tc[i] ≠ tc then                  //meet a new core time shell
6          if tc ≠ 0 and |C^k_[ts,tc]| ≥ s then   //condition is satisfied
7              return tc ;                   // tc is the earliest qualified end time
8          tc ← A_tc[i] ;                    //update the current core time
9      C^k_[ts,tc] ← C^k_[ts,tc] ∪ {A_n[i]} ;   //add a vertex
10     n ← A_n[i] ;                          //jump to the latest successor
11 return ∞ ;                                // ∞ means no qualified end time
```

users reside in the k-cores of social networks with high values of k, the analysers may be interested in asking such questions: (q1) when a known celebrity user became influential from a date; (q2) when the number of influential users increased to one thousand from a date; (q3) when the group of influential users achieved the best conductance from a date; (q4) in which month the number of influential users increased the most from a date.

In addition, there are other examples. [28] queries periods of increased interaction between students from different classes, such as during breaks, to indicate higher risk periods for disease transmission. [24] queries time points in the runtime history of a smart healthcare system for monitoring structural changes.

5.3 When-Query Processing

The when queries derived from Definition 5.1 can be addressed with a unified pipeline. Specifically, we maintain the historical k-core $C^k_{[t_s,t_e]}$ with a given t_s in the ascending order of $t_e \geq t_s$, and check whether $f(C^k_{[t_s,t_e]})$ can meet the query condition.

With the MCTS-index, the maintenance can be implemented efficiently. We incrementally obtain the historical k-cores with the same t_s for each distinct core time t_c but not each end time t_e. In this way, the total cost of maintenance only depends on the size of the largest core but not the total size of all cores. Moreover, the maintenance can stop early when the metric $f(\cdot)$ is a monotonic function on t_e. For example, when processing q1 or q2 in Table 1, it can be terminated as soon as the last vertex in the given vertex set has been contained by the core or there have been the given number of vertices in the core. Therefore, the MCTS-index can accelerate the When-Query processing significantly.

Due to the limitation of paper length, only the query processing algorithm of q2 in Table 1 is presented by Algorithm 6 as an example. In a nutshell, for each enumerated distinct core time, we add all vertices in the corresponding core time shell into the maintained historical k-core, and return the core time if the query condition has been satisfied.

The time complexity of the above uniform query processing approach can be summarized as follows.

Theorem 5.1. *The time complexity of When-Query processing based on the MCTS-index is $O(vnum \cdot \log \bar{l} + tnum \cdot O_f)$, where $vnum$ is the number of traversed vertices bounded by the total number of vertices in the (k, t_s)-structure, $tnum$ is the number of enumerated core time bounded by the number of core time shells in the (k, t_s)-structure, and O_f is the time complexity of $f(\cdot)$.*

Proof. Our approach is simply comprised of core maintenance and metric evaluation. According to Theorem 3.5, the time complexity of maintenance derived from the MCTS-query is $O(vnum \cdot \log \bar{l})$. Moreover, there are $tnum$ times of evaluation of $f(\cdot)$, and thereby the time complexity of evaluation is $O(tnum \cdot O_f)$. □

Lastly, we discuss an interesting issue: how efficient is the PHC-index for addressing when queries. Firstly, the PHC-index cannot maintain the historical k-core in the ascending order of t_e as efficiently as the MCTS-index. Because the time complexity of maintenance based on the PHC-query is always $O(|\mathcal{V}| \cdot \log \bar{t})$. Secondly, since the PHC-index is designed to determine whether or not a vertex is contained by a specific historical k-core, we can develop a dedicated algorithm to address the "time of containment" queries optimally, which uses the PHC-index to obtain only the core time of the given set of vertices and returns the maximum core time. Therefore, the MCTS-index is generally the more efficient tool for when query processing, and the PHC-index gains advantages on those queries with vertex containment conditions.

6 EXPERIMENT

We conduct extensive experiments on a Linux machine with a 2.2 GHz CPU and 120 GB of RAM. All algorithms are implemented using the C++ Standard Template Library.

6.1 Dataset

For fairness, we obtain the ten real-world datasets of [34] from the public sources including SNAP [13] and KONECT [11]. The statistics of the datasets are shown in Table 3. Except the regular graph metrics like $|\mathcal{V}|$ and $|\mathcal{E}|$ that represent the scale in topology, the number of distinct timestamps t_{max} represents the scale in time. By comparing t_{max} and the average number of distinct core time $\bar{t}$ for each dataset respectively, we empirically prove the discreteness of the evolution of vertex coreness (see Figure 2a), which guarantees the compression effectiveness of the MCTS/PHC-index. Moreover, the maximum coreness k_{max} is as small as assumed, so that constructing MCTS(k) for each $k \in [2, k_{max}]$ will not dramatically expand the overall index size.

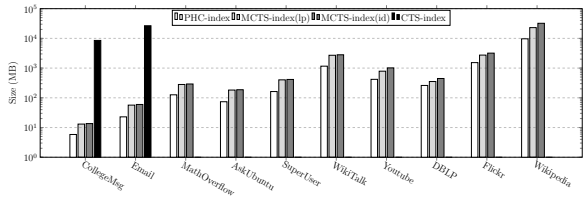
6.2 Index Overhead

We construct four indexes including the PHC-index [34], the CTS-index, the MCTS-index (id), and the MCTS-index (lp) to compare index overheads. The difference between the two versions of the MCTS-index is that, the (id) version ranks the vertices in each core time shell by their ids, and the (lp) version heuristically moves a vertex to the last position of its new core time shell (Algorithm 5).

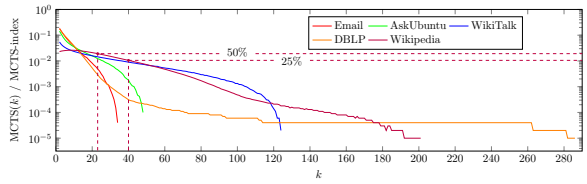
Figure 5a shows the space overheads on all datasets. We have the following observations. 1) Compared to the PHC-index, our MCTS-index is at least 1.36 times, at most 2.48 times, and on average 2.16 times larger. Thus, the MCTS-index is almost as space-efficient as the PHC-index. 2) In practice, not every MCTS(k) needs to be stored. The smaller the value of k, the larger the size of MCTS(k), and the

Table 3: The statistics of public real-world temporal graphs used in our experiments.

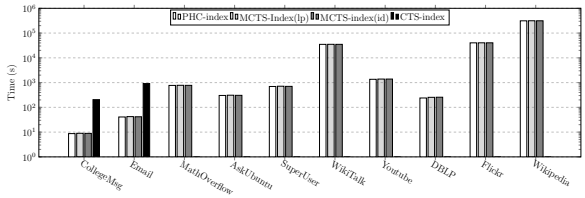
	CollegeMsg	Email	MathOverflow	AskUbuntu	SuperUser	WikiTalk	Youtube	DBLP	Flickr	Wikipedia
$\|\mathcal{V}\|$	1,899	986	24,818	159,316	194,085	1,140,149	3,223,585	1,824,701	2,302,925	1,870,709
$\|\mathcal{E}\|$	59,835	332,334	506,550	964,437	1,443,339	7,833,140	9,375,374	29,487,744	33,140,017	39,953,145
t_{max}	58,911	207,880	505,784	960,866	1,437,199	7,375,042	203	77	134	2,198
$\bar{t}$	51.82	174.72	85.38	20.73	29.28	54.13	5.72	3.46	8.48	34.46
k_{max}	20	34	78	48	61	124	88	286	600	206



(a) Space overheads.



(b) Distribution of individual MCTS(k) size.



(c) Time overheads.

Figure 5: Comparison of index construction overheads.

less cohesive the historical k-core, as illustrated in Figure 5b. Thus, the MCTS-Index could include MCTS(k) only if k is greater than a threshold, thereby reducing the space overhead while preserving the most cohesive results. 3) Compared to the MCTS-index (id) with a global order of vertex, the MCTS-index (lp) is always smaller. It proves the effectiveness of heuristics that exploits the correlation of core time between vertices in Algorithm 5. 4) As expected, the sizes of the CTS-indexes are too large, so that we only construct the CTS-indexes for the two smallest datasets. It confirms the necessity of developing Algorithm 5.

Figure 5c shows the time overheads on all datasets. Note that, the time overhead of the MCTS-index is composed of two parts: constructing the PHC-index and obtaining the MCTS-index from that (see Algorithm 5). The construction time of the MCTS-index is almost equal to that of the PHC-index, which means our vertex moving algorithm (no matter which position) costs only a small percentage of the total construction time. In detail, the percentage is between 0.3% and 1.9%, and the mean is 0.8%.

6.3 What-Query Efficiency

We compare the efficiency of the PHC-query [34], TCD-single (only dealing with a single query though the TCD algorithm is designed for batch processing) [31], and the MCTS-query (using the MCTS-index (lp)) with random queries. For a specific dataset and each combination of $k = 10\%, 30\%, 50\%, 70\%, 90\%$ of k_{max} and $(t_e - t_s) = 10\%, 30\%, 50\%, 70\%, 90\%$ of t_{max}, we generate a group of 10,000 queries with random span-fixed time interval $[t_s, t_e]$. The reported response time for each group is the mean.

Figure 6 shows the results on three datasets including Wikipedia, Flickr, and WikiTalk. They are the three largest datasets except DBLP and Youtube, which we do not adopt because [34] does not obtain them from public sources. Moreover, WikiTalk has the most timestamps, and thereby is suitable to test the sensitivity to time span. The results demonstrate that, the MCTS-query is generally much more efficient than the PHC-query, and the index-free TCD-single is the slowest as expected, which means our main research objective is achieved. Moreover, unlike the PHC-query, the efficiency of the MCTS-query increases significantly with the increase of k or the decrease of $(t_e - t_s)$, because the MCTS-query only traverses the vertices in the historical k-cores. Especially for the groups with the time span $(t_e - t_s) \leq 30\% t_{max}$ in which most ordinary queries fall, the MCTS-query outperforms the PHC-query by 1~3 orders of magnitude.

Note that, in the only exceptional group for which the MCTS-query is slower than the PHC-query (see Figure 6e), the historical k-cores have almost the same size of the entire graph.

6.4 When-Query Effectiveness

To evaluate the effectiveness of When-Query, we implement the algorithms to address the when historical k-core queries in Table 1 based on the MCTS-index (lp).

For q1, we run four queries with $k = 40$, $t_s = 2001.10$ (month), and V comprised of the vertices whose static coreness is in the range of [40,59], [60,79], [80,99], or [100,119] respectively on WikiTalk. Figure 7a shows the distribution of the earliest end time t_e to join the historical k-core for each vertex. The result reveals an interesting fact: the vertices that are more influential (namely, have the greater coreness) in the whole history would also join a historical k-core earlier from a specific start time in the sense of statistics.

For q2, we run a set of queries with $k = 30$, $t_s = 2009.09$ (month), and $s = 100, 101, \cdots, 500$ on three datasets including AskUbuntu, MathOverflow, and SuperUser, respectively. Figure 7b shows the result t_e. Generally, the historical k-cores gradually grow larger with the expanding of time interval. Moreover, the historical k-core of AskUbuntu grows more quickly and frequently than others.

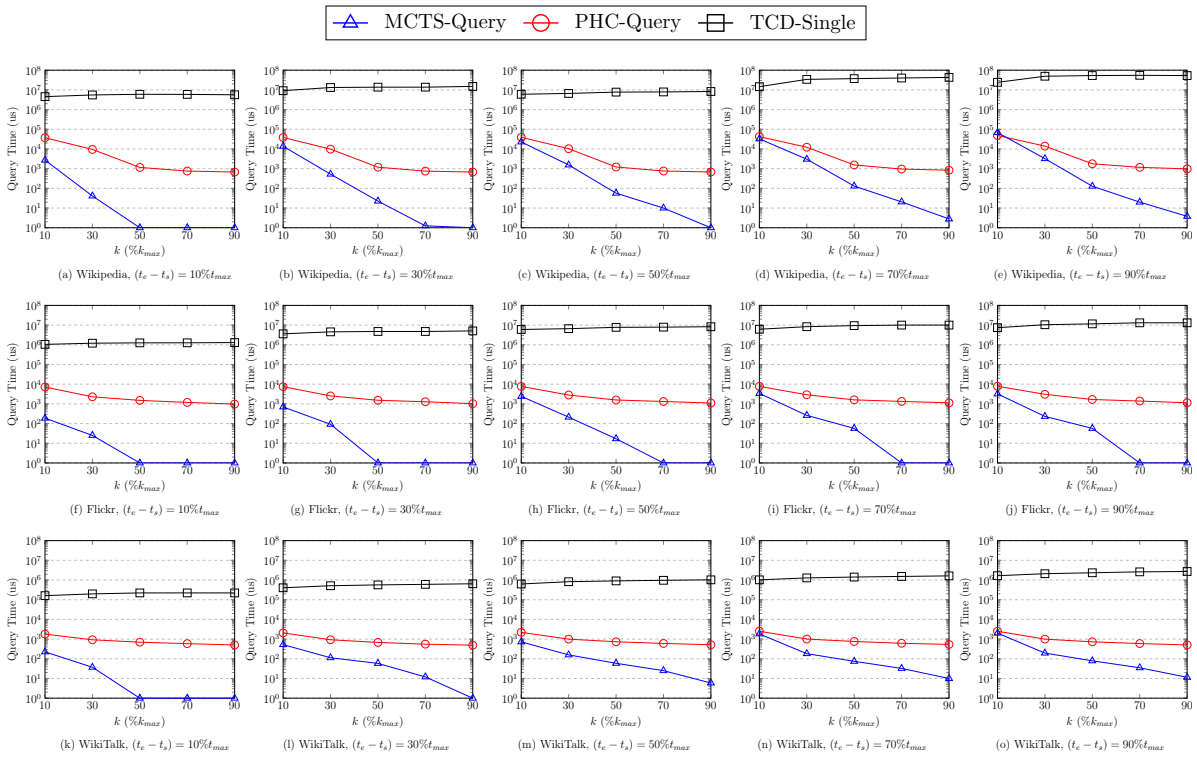
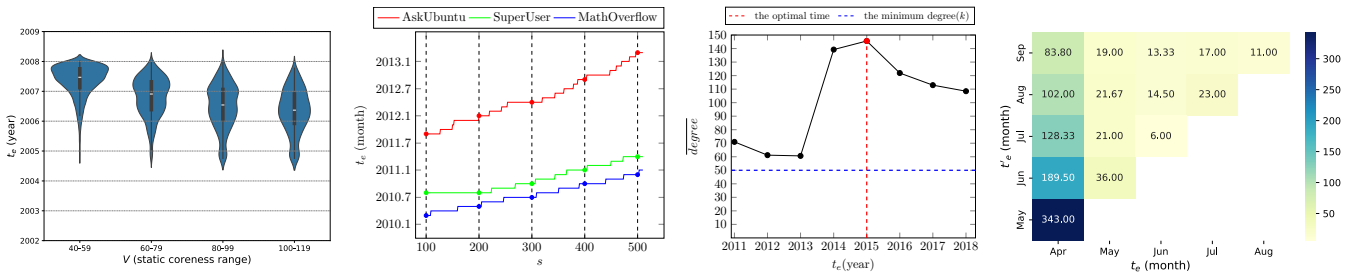


Figure 6: The query time for random historical k-core queries with varied k and $(t_e - t_s)$ on the three largest datasets.



(a) q1: $k = 30$, $t_s = 2001.10$, **WikiTalk.** **(b) q2:** $k = 30$, $t_s = 2009.09$, **multiple.** **(c) q3:** $k = 50$, $t_s = 2011$, **DBLP.** **(d) q4:** $k = 15$, $t_s = 2004.04$, **CollegeMsg.**

Figure 7: Case studies of when historical k-core query.

For q3, we run a query with $k = 50$ and $t_s = 2011$ (year) on DBLP. Figure 7c shows the average degree of the historical k-core with $t_e = 2011, 2012, \cdots, 2018$. The answer to the query is 2015, in which year the average degree reaches the peak. Then, the average degree starts to fall, which means the new vertices of the historical k-core have the relatively lower degree (but still no less than 50).

For q4, we run a query with $k = 15$ and $t_s = 2004.04$ (month) on CollegeMsg. Figure 7d shows the growth speed (# vertex/month) of the historical k-core in each period from t_e until t'_e. Starting from April, the historical k-core grows most rapidly between April (t_e) and May (t'_e). That is because the distribution of the edge number in each month is skewed and 45.4% of the edges appear in May.

7 CONCLUSION

We propose a novel index called MCTS-index for temporal graphs. By organizing the vertices in a list of "shells" in the ascending order of core time, the MCTS-index facilitates nearly optimal processing of both "what" and "when" historical k-core queries. Meanwhile, it is also space-efficient due to the compression technique that leverages the discreteness of the evolution of vertex coreness.

ACKNOWLEDGMENTS

This work was supported by the Natural Science Foundation of Hubei Province (No. JCZRYB202500322), the NSFC (No. 61202036, 62272353, and 62276193), and the RGC of Hong Kong (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. https://doi.org/10.1016/j.ins.2019.11.003

[2] Kaiyu Chen, Dong Wen, Wenjie Zhang, Ying Zhang, Xiaoyang Wang, and Xuemin Lin. 2024. Querying Structural Diversity in Streaming Graphs. *Proceedings of the VLDB Endowment* 17, 5 (2024), 1034–1046.

[3] Deming Chu, Fan Zhang, Xuemin Lin, Wenjie Zhang, Ying Zhang, Yinglong Xia, and Chenyi Zhang. 2020. Finding the Best k in Core Decomposition: A Time and Space Optimal Solution. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*.

[4] 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. https://doi.org/10.14778/3358701.3358704

[5] Joana MF da Trindade, Julian Shun, Samuel Madden, and Nesime Tatbul. 2024. Kairos: Efficient Temporal Graph Analytics on a Single Machine. *arXiv preprint arXiv:2401.02563* (2024).

[6] Monika Filipovska and Hani S. Mahmassani. 2023. Spatio-Temporal Characterization of Stochastic Dynamic Transportation Networks. *IEEE Transactions on Intelligent Transportation Systems* 24, 9 (2023), 9929–9939. https://doi.org/10.1109/TITS.2023.3276190

[7] Xiangyang Gou, Xinyi Ye, Lei Zou, and Jeffrey Xu Yu. 2024. LM-SRPQ: Efficiently Answering Regular Path Query in Streaming Graphs. *Proceedings of the VLDB Endowment* 17, 5 (2024), 1047–1059.

[8] Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, and Alex Thomo. 2015. K-core decomposition of large networks on a single PC. *Proceedings of the VLDB Endowment* 9, 1 (2015), 13–23.

[9] Udayan Khurana and Amol Deshpande. 2019. *Historical Graph Management*. Springer.

[10] Maksim Kitsak, Lazaros K. Gallos, Shlomo Havlin, Fredrik Liljeros, Lev Muchnik, H. Eugene Stanley, and Hernán A. Makse. 2010. Identification of influential spreaders in complex networks. *Nature Physics* 6, 11 (2010), 888–893.

[11] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd international conference on world wide web*. Association for Computing Machinery, 1343–1350. https://doi.org/10.1145/2487788.2488173

[12] Sune Lehmann. 2019. *Fundamental Structures in Temporal Communication Networks*. Springer International Publishing, 25–48. https://doi.org/10.1007/978-3-030-23495-9_2

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

[14] A. Li, S. P. Cornelius, Y.-Y. Liu, L. Wang, and A.-L. Barabási. 2017. The fundamental advantages of temporal networks. *Science* 358, 6366 (2017), 1042–1046. https://doi.org/10.1126/science.aai7488

[15] 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. https://doi.org/10.1109/ICDE.2018.00077

[16] Sijia Li, Gaopeng Gou, Chang Liu, Chengshang Hou, Zhenzhen Li, and Gang Xiong. 2022. TTAGN: Temporal Transaction Aggregation Graph Network for Ethereum Phishing Scams Detection. In *Proceedings of the ACM Web Conference 2022*. Association for Computing Machinery, 661–669. https://doi.org/10.1145/3485447.3512226

[17] 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. https://doi.org/10.1007/s11280-021-00917-z

[18] Longlong Lin, Pingpeng Yuan, Rong-Hua Li, Chunxue Zhu, Hongchao Qin, Hai Jin, and Tao Jia. 2024. QTCS: Efficient Query-Centered Temporal Community Search. *Proceedings of the VLDB Endowment* 17, 6 (2024), 1187–1199.

[19] Naoki Masuda and Renaud Lambiotte. 2021. *A Guide to Temporal Networks*. World Scientific.

[20] Sen Pei, Lev Muchnik, Jr. José S. Andrade, Zhiming Zheng, and Hernán A. Makse. 2014. Searching for superspreaders of information in real-world social media. *Scientific Reports* (2014).

[21] Evaggelia Pitoura. 2017. Historical Graphs: Models, Storage, Processing. In *European Business Intelligence and Big Data*.

[22] Hongchao Qin, Rong-Hua Li, Ye Yuan, Guoren Wang, Lu Qin, and Zhiwei Zhang. 2022. Mining Bursting Core in Large Temporal Graphs. *Proceedings of the VLDB Endowment* 15, 13 (2022), 3911–3923. https://doi.org/10.14778/3565838.3565845

[23] Hongchao Qin, Rong-Hua 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* 34, 8 (2020), 3927–3945. https://doi.org/10.1007/978-3-031-25158-0_38

[24] Lucas Sakizloglou, Sona Ghahremani, Matthias Barkowsky, and Holger Giese. 2022. Incremental execution of temporal graph queries over runtime models with history and its applications. *Softw. Syst. Model.* 21, 5 (Oct. 2022), 1789–1829. https://doi.org/10.1007/s10270-021-00950-6

[25] Jingyi Song, Dong Wen, Lantian Xu, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2024. On Querying Historical Connectivity in Temporal Graphs. 2, 3, Article 157 (2024), 25 pages.

[26] Yifu Tang, Jianxin Li, Nur Al Hasan Haldar, Ziyu Guan, Jiajie Xu, and Chengfei Liu. 2022. Reliable Community Search in Dynamic Networks. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2826–2838. https://doi.org/10.14778/3551793.3551834

[27] Yifu Tang, Jianxin Li, Nur Al Hasan Haldar, Ziyu Guan, Jiajie Xu, and Chengfei Liu. 2023. Reliability-driven local community search in dynamic networks. *IEEE Transactions on Knowledge and Data Engineering* (2023).

[28] Evangelia Tsoukanara, Georgia Koloniari, Evaggelia Pitoura, and Peter Triantafillou. 2024. The GraphTempo Framework for Exploring the Evolution of a Graph Through Pattern Aggregation. *IEEE Transactions on Knowledge and Data Engineering* 36, 11 (2024), 7143–7156. https://doi.org/10.1109/TKDE.2024.3410647

[29] 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. https://doi.org/10.1109/BigData.2015.7363809

[30] Haoxuan Xie, Yixiang Fang, Yuyang Xia, Wensheng Luo, and Chenhao Ma. 2023. On Querying Connected Components in Large Temporal Graphs. *Proc. ACM Manag. Data* 1, 2, Article 170 (2023), 27 pages.

[31] Junyong Yang, Ming Zhong, Yuanyuan Zhu, Tieyun Qian, Mengchi Liu, and Jeffrey Xu Yu. 2023. Scalable time-range k-core query on temporal graphs. *arXiv preprint arXiv:2301.03770* (2023).

[32] Junyong Yang, Ming Zhong, Yuanyuan Zhu, Tieyun Qian, Mengchi Liu, and Jeffrey Xu Yu. 2024. Evolution Forest Index: Towards Optimal Temporal k-Core Component Search via Time-Topology Isomorphic Computation. *Proc. VLDB Endow.* 17, 11 (Aug. 2024), 2840–2853. https://doi.org/10.14778/3681954.3681967

[33] Yajun Yang, Hanxiao Li, Xiangju Zhu, Junhu Wang, Xin Wang, and Hong Gao. 2023. HR-Index: An Effective Index Method for Historical Reachability Queries over Evolving Graphs. *Proceedings of the ACM on Management of Data* 1, 2 (2023).

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

[35] Zheng Zhang, Jun Wan, Mingyang Zhou, Zhihui Lai, Claudio J Tessone, Guoliang Chen, and Hao Liao. 2023. Temporal burstiness and collaborative camouflage aware fraud detection. *Information Processing & Management* 60, 2 (2023), 103170.

[36] Kun Zhao, Márton Karsai, and Ginestra Bianconi. 2013. *Models, Entropy and Information of Temporal Social Networks*. Springer Berlin Heidelberg, 95–117. https://doi.org/10.1007/978-3-642-36461-7_5

[37] Ming Zhong, Junyong Yang, Yuanyuan Zhu, Tieyun Qian, Mengchi Liu, and Jeffrey X. Yu. 2024. A Unified and Scalable Algorithm Framework of User-Defined Temporal (k, X)-Core Query. *IEEE Transactions on Knowledge and Data Engineering* (2024).