TeGraph+: Scalable Temporal Graph Processing Enabling Flexible Edge Modifications

Chengying Huan, Yongchao Liu, Heng Zhang, Hang Liu, Shiyang Chen, Shuaiwen Leon Song, and Yanjun Wu

***Abstract*—Temporal graphs are widely used for time-critical applications, which enable the extraction of graph structural information with temporal features but cannot be efficiently supported by static graph computing systems. However, the current state-of-the-art solutions for temporal graph problems are not only ad-hoc and suboptimal, but they also exhibit poor scalability, particularly in terms of their inability to scale to evolving graphs with flexible edge modifications (including insertions and deletions) and diverse execution environments. In this article, we present two key observations. First, temporal path problems can be characterized as *topological-optimum* problems, which can be efficiently resolved using a universal single-scan execution model. Second, data redundancy in transformed temporal graphs can be mitigated by merging superfluous vertices. Building upon these fundamental insights, we propose TeGraph+, a versatile temporal graph computing engine that makes the following contributions: (1) a unified optimization strategy and execution model for temporal graph problems; (2) a novel graph transformation model with graph redundancy reduction strategy; (3) a spanning tree decomposition (STD) based distributed execution model which uses an efficient transformed graph decomposition strategy to partition the transformed graph into different spanning trees for distributed execution; (4) an efficient mixed imperative and lazy graph update strategy that offers support for evolving graphs with flexible edge modifications; (5) a general system framework with user-friendly APIs and the support of various execution environments, including in-memory, out-of-core, and distributed execution environments. Our extensive evaluation reveals that TeGraph+ can achieve up to 241× speedups over the state-of-the-art counterparts.**

Manuscript received 6 April 2023; revised 21 January 2024; accepted 15 April 2024. Date of publication 26 April 2024; date of current version 1 July 2024. This work was supported in part by Ant Group through Ant Research Intern Program, in part by the National Key Research & Development Program of China under Grant 2020YFC1522702, in part by the Natural Science Foundation of China under Grant 61877035 and Grant 62141216, in part by the National Natural Science Foundation of China under Grant 62002350, in part by National Science Foundation CRII Award under Grant 2000722, CAREER Award under Grant 2046102, in part by SOAR Fellowship, in part by the University of Sydney Faculty Startup Funding, and in part by Australia Research Council (ARC) Discovery Project under Grant DP210101984. An earlier version of this paper was presented at ICDE 2022 [DOI: 10.1109/ICDE53745.2022.00048]. Recommended for acceptance by S. Pallickara. *(Corresponding author: Chengying Huan.)*

Chengying Huan is with the Institute of Software, Chinese Academy of Sciences, Beijing 100045, China, also with Rutgers University, New Brunswick, NJ 08901 USA, and also with Tsinghua University, Beijing 100190, China (e-mail: huanchengying@iscas.ac.cn).

Yongchao Liu is with Ant Group, Hangzhou 310058, China.

Heng Zhang and Yanjun Wu are with the Institute of Software, Chinese Academy of Sciences, Beijing 100045, China.

Hang Liu and Shiyang Chen are with the Rutgers University, New Brunswick, NJ 08901 USA.

Shuaiwen Leon Song is with University of Sydney, Camperdown, NSW 2050, Australia.

Digital Object Identifier 10.1109/TPDS.2024.3393914

***Index Terms*—Graph algorithm, temporal graphs, parallel and distributed system.**

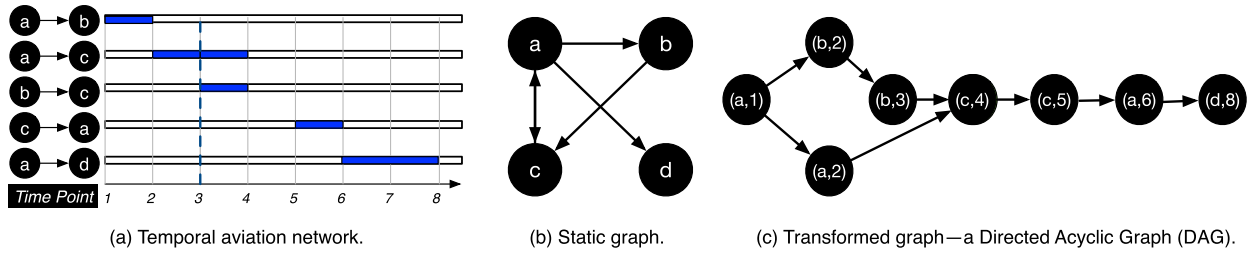
I. Introduction

TEMPORAL graphs, which associate edges with time intervals, offer additional capabilities for representing time-critical applications that traditional static graph computing engines cannot capture [2], [3], [4]. In reality, numerous crucial applications are based on temporal graphs [1], [5], [6], such as aviation networks [7], e-commerce [8], and real-time epidemiological analysis (e.g., Influenza and COVID-19 outbreaks [9]). Social media graphs [6] also incorporate friending periods as edge labels. Despite their importance, existing research has primarily focused on non-temporal static graphs, while some studies have considered evolving graphs as a sequence of updates to non-temporal graphs but without time constraints [10], [11], [12]. Fig. 1(a) depicts a temporal graph of an aviation network, where each edge is assigned a time interval. The time interval $(1, 2)$ connected to the edge from a to b indicates a flight departing from a at time 1 and arriving at b at time 2.

Most temporal graph applications that utilize or are represented by temporal graphs focus on solving the *general temporal path problems*, where a *temporal path* is a valid path subject to time constraints. These path problems serve as fundamental building blocks for numerous essential applications. In the literature, two primary approaches have been proposed to tackle temporal path problems: *static execution* [6], [7], [13] and *transformation-based execution* [7], [14], [15]. Static execution stores the temporal graph in an adjacency list format, with each neighbor also containing its temporal information. During processing, one can directly apply static graph execution, but with additional consideration for time constraints [16]. This approach may suffer from redundant data access and computations. Transformation-based execution is more promising, which transforms the original temporal graph into an equivalent, larger static graph by expanding each vertex according to the timing information. The topological structure of the transformed graph incorporates all necessary timing constraints. The transformed graph is then processed using state-of-the-art static graph processing models. Fig. 1(c) demonstrates an example of such a transformation.

A. Challenges and Motivations

State-of-the-art temporal graph processing techniques encounter four fundamental challenges: *computation, space, bandwidth, and scalability*. We will discuss them as follows.



(a) Temporal aviation network. (b) Static graph. (c) Transformed graph—a Directed Acyclic Graph (DAG).

Fig. 1. An aviation map example.

The computation challenge poses the primary obstacle to realizing the full potential of processing efficiency. Static execution demands high computational complexity and bandwidth overhead to manage the additional time information. In contrast, transformation-based execution greatly simplifies computation by processing transformed graphs (DAGs) but suffers from high latency due to its reliance on traditional iterative processing models. We have demonstrated that there is a lack of *a universal optimization strategy* for achieving optimal performance across different path problems (Section II-C).

For space, the graph transformation in transformation-based execution, although convenient, does suffer from the graph amplification problem which incurs large memory and disk I/O overhead.

For bandwidth, state-of-the-art designs resort to temporal information associated with static graph representations, such as edge lists and adjacency lists, to represent temporal graphs. This results in enormous bandwidth waste for temporal graph analysis which is I/O intensive.

For better scalability, existing methods face two major challenges. First, they are typically optimized for specific execution environments and struggle to adapt to various settings like in-memory, out-of-core, and distributed execution. Our earlier work [1] didn't cover distributed execution, crucial for efficiently processing large, changing temporal graphs. Distributed computing's advantage lies in using multiple resources to speed up processing large-scale data. Second, these methods aren't flexible enough for evolving graphs that require easy edge modifications. In dynamic environments like e-commerce and social networks, temporal graphs constantly change, necessitating updates in both the graph structure and the outcomes of computations. Our previous work [1] focused on static temporal graph snapshots and didn't support graph mutations. With edge changes, static execution methods must rerun algorithms, leading to inefficiency, especially as these changes can affect temporal paths. Even transformation-based methods need to rebuild transformed graphs for new analyses, which isn't optimal. This shows that evolving graph approaches like KickStarter [11] and CommonGraph [17], though good at updating graphs, fall short in handling the temporal constraints of temporal paths. As a result, existing incremental graph update techniques have not developed an efficient strategy for incremental updates on transformed graphs within temporal contexts.

B. Contributions

This paper presents a solution that is fast, space-efficient, bandwidth-efficient, and scalable for handling general temporal path problems and their applications, including dynamic edge modifications like insertions and deletions. Our approach is based on two key insights: (1) We recognize that solving temporal path problems involves addressing what we term *topological-optimum* problems. Here, every subpath of the final path is also a solution to a sub-problem. By transforming a temporal graph into a static Directed Acyclic Graph (DAG), we make these problems topologically optimal. This leads us to propose a *universal single scan execution* strategy, which treats general path problems on temporal graphs as *topological-optimum* problems on transformed DAGs. This method replaces iterative-based graph execution models, reducing the need for continuous edge reads and computations; (2) During the transformation phase, we note that transformed DAGs of temporal graphs often contain redundant data. To tackle this, we introduce a *hyper-method* transformation model to condense the transformed graph efficiently, reducing execution workload and memory footprint. This approach facilitates in-memory processing and minimizes disk I/O in out-of-core execution.

Leveraging these insights, we introduce TeGraph+, a comprehensive engine for temporal graph computing. It includes a spanning tree decomposition (STD) based distributed execution method for scalable distributed execution and a novel time-aware graph format that efficiently processes temporal graphs by treating different time instances as separate vertices. TeGraph+ is versatile, suitable for in-memory, out-of-core, and distributed environments. It also features an efficient mixed imperative and lazy graph update strategy for updating transformed graphs after edge modifications. Our evaluations on real-world graphs show that TeGraph+ achieves exceptional performance, with a throughput of 200 million edges per second, significant disk space savings, and up to 17 times faster graph transformation. Moreover, it outperforms state-of-the-art solutions by up to 241 times in speed.

In comparison to our earlier work, TeGraph [1], this manuscript introduces several technological advancements to address the scalability challenges previously encountered. TeGraph lacked the capability to scale to evolving graph models with support for flexible edge modifications and did not accommodate distributed execution. In this revised version, we have made significant strides in both these areas:

Distributed Execution Strategy: To enable distributed execution, we introduce the Spanning Tree Decomposition (STD)-based distributed execution strategy. This approach involves partitioning the transformed graph (DAG) into various spanning trees, allowing TeGraph+ to extend its reach to distributed environments. As a result, TeGraph+ achieves up to a 3.99× speedup compared to the state-of-the-art distributed graph processing system, GraphScope [18].

Support for Evolving Graph Data Model: Addressing the need to manage evolving graph models, we propose a mixed imperative and lazy graph update strategy. This new approach efficiently accommodates evolving graph models by supporting flexible edge modifications. It efficiently updates the transformed graph and the final results in response to changes, thus enhancing overall system agility and response time.

Enhanced Evaluations: We conduct more extensive evaluations to deeply analyze the advancements of TeGraph+. These evaluations offer a comprehensive insight into the improved performance and scalability of our system.

These contributions collectively represent a significant evolution from TeGraph, addressing key scalability challenges and expanding the scope and applicability of our graph processing framework.

II. Background and Motivation

A. Notations for Temporal Graph

Temporal Graph Format: Unlike static graphs, edges in temporal graphs are associated with time intervals. Consider a temporal graph $\mathbb{G} = (V, E)$, where V represents the set of vertices and E is the set of edges. Each edge $e \in E$ is characterized as (u, v, s, t), with $u, v \in V$. This denotes an edge from vertex u to vertex v, beginning at time s and ending at time t, where $s, t \in \mathbb{R}$. In a graph that includes edge costs, a weight w is assigned to each edge, represented as (u, v, s, t, w). For the sake of simplicity, we assume that all elements are meaningful and valid if $s < t$. It is possible to have multiple edges between the same pair of vertices in such graphs. To denote the number of incoming and outgoing edges of a vertex v, we use $d_{in}[v]$ and $d_{out}[v]$, respectively. The term D is used to represent the maximum in-degree or out-degree found within the graph.

Temporal Path Definition: A temporal path $P = \{v_1, v_2, \ldots, v_{n+1}\}$ in a temporal graph can be represented as $P = e_1 \cdot e_2 \cdot \ldots \cdot e_n$, where each $e_i = (v_i, v_{i+1}, s_i, t_i)$ is a temporal edge. Every temporal path P must adhere to time constraints, specifically $end(e_i) \leq start(e_{i+1})$, or in other words, $t_i \leq s_{i+1}$, for $1 \leq i \leq n$. For example, in Fig. 1(a), a valid temporal path exists from edge $(a, b, 1, 2)$ to $(b, c, 3, 4)$, whereas the path from $(c, a, 5, 6)$ to $(a, b, 1, 2)$ does not constitute a valid temporal path due to the inconsistency in time constraints. A subpath of a path, termed as a prefix, is a subset of consecutive edges from the path. For example, $P' = e_1 \cdot e_2 \cdot \ldots \cdot e_k$ is a prefix of P if $P = P' \cdot e_{k+1} \cdot \ldots \cdot e_n$, which can also be expressed as $P = P' \cup \{v_{k+2}, v_{k+3}, \ldots, v_{n+1}\}$.

Distinction from Traditional Dynamic Graphs: Current dynamic graph (or evolving graph) engines [17], [19], [20] typically handle dynamic graphs characterized by flexible edge insertion and deletion but lack temporal features, meaning they do not contend with the time constraints associated with temporal paths. Consequently, the primary differences between traditional dynamic graphs and temporal graphs lie in the latter's temporal features and the specific time constraints that govern temporal paths.

Dynamic Temporal Graph: Real-world temporal graphs are usually evolving over time. Graph evolving process includes edge insertion operations and edge deletion operations. Specifically, each graph mutation contains a batch of edge modifications $\Delta\mathbb{G}$ with edge (u, v, s, t) be added to previous graph snapshot for edge insertion and edge (u, v, s, t) be removed from previous graph snapshot for edge deletion. The current snapshot of the temporal graph ($\mathbb{G}_k$) can be represented by $\mathbb{G}_k = \mathbb{G}_{k-1} + \Delta\mathbb{G}$. Under graph mutation, temporal graph processing has to update the snapshot $\mathbb{G}_k$ and the final results on the new graph.

Data Model: In real-world functions, a temporal graph is represented as an *edge stream* [7], [8], [21]: it is simply a sequence of all the edges coming in the order of time that each edge is created or collected. Edges in temporal graphs are normally ordered based on their starting time. The edge stream data representation is a common format for temporal graphs. But the edge modifications do not need to guarantee any order.

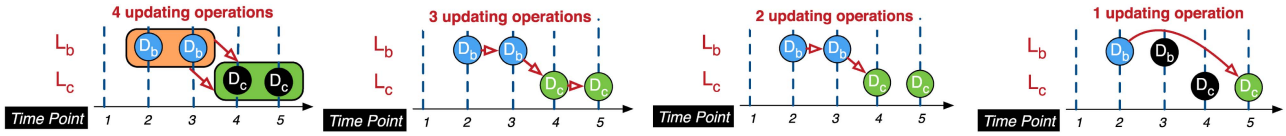
General Temporal Path Problems: Most real-world applications using or represented by temporal graphs focus on solving the *temporal path* problems, which are the essential building blocks for many advanced graph analytics [7], [15], [21], [22], [23]. We list several most representative temporal path problems in the previsou paper TeGraph [1].

B. State-of-the-Art Solutions

Static Execution: Typically, techniques based on static execution directly utilize traditional static graph execution models such as Dijkstra's [24] or Bellman-Ford [25] methods to process temporal graphs [6], [7], [13], [26], [27], [28], [29]. These methods employ a multi-version distance array to record the distance of each vertex at different time instances. This approach results in significant computation complexity and memory access overhead due to the added time dimension and the need for additional procedures to ensure time constraints. For instance, in shortest path processing, the traditional shortest path array $d[u]$ is expanded into a series of $d[u][s]$ to represent the optimal path from the source vertex to vertex u at time s while executing graph algorithms. During each iteration, it is necessary to use $d[u][s']$ ($s' \leq s$) to update $d[v][t']$ ($t \leq t'$). Formally, when updating the edge (u, v, s, t, w), the update function is defined as

$$dis[v][t'] = \min\{dis[v][t'], dis[u][s'] + w\}, s' \leq s, t \leq t'. \quad (1)$$

Fig. 2(a) demonstrates the updating function for the edge $(b, c, 3, 4)$ in Fig. 1(a) under static execution, $(b, 2)$ and $(b, 3)$ will have to individually update vertices $(c, 4)$ and $(c, 5)$ with total four update operations. This leads to redundant computations. Further, since static execution stores neighbors of different



(a) Static Execution. (b) Transformation-based Execution. (c) One-Pass. (d) TeGraph+.

Fig. 2. Updating edge $(b, c, 3, 4)$ on Fig. 1(a) in static execution, transformation-based execution, One-Pass, and TeGraph+.

temporal information together, expired neighbors will be loaded, leading to redundant data access.

A method called *One-Pass* [7] modifies traditional static graph algorithms to process temporal graphs in a single iteration. For every vertex v, One-Pass maintains a list L_v composed of (d_i, t_i) tuples, expressed as $L_v = \{(d_0, t_0), \ldots, (d_n, t_n)\}$. Here, d_i represents the distance from the source vertex to v at time t_i, and these tuples are ordered by t such that $t_{i-1} < t_i$. When processing each edge (u, v, s, t, w), One-Pass should update the tuple (d, t) in L_v. This update is based on all (d_i, s_i) tuples in L_u (where u is the edge's source) that satisfy $s_i \leq s$, i.e., $d = min\{d, d_i + w\}$ for the shortest path. For instance, using edge $(b, c, 3, 4, w)$ as an example, it searches $(d, t) \in L_b$ with $t \leq 3$, e.g., $(d_{(b,2)}, 2)$ and $(d_{(b,3)}, 3)$, to update the destination vertex $(d_{(c,4)}, 4)$ in L_c by following:

$$d_{(c,4)} = min\{d_{(c,4)}, d_{(b,2)} + w, d_{(b,3)} + w\}. \tag{2}$$

This process is repeated for all edges. Since searching tuples takes $log(D)$ time, One-Pass's time complexity is $O(|E|log(D))$. Its space complexity for reachability is $O(|E| + |V|)$, and $O(|E| + 2|V|)$ for other applications. One-Pass needs recalculating for new results if edges are inserted or deleted, as such modifications change the time instances and require rebuilding the temporal paths. Changes in edges and time instances can make some temporal paths invalid due to time constraints, even if the paths are connected.

As we will discuss shortly, although One-Pass is faster than traditional static execution techniques, it (1) is slower than our *single scan* approach, which only requires $O(|E|)$, (2) necessitates ad-hoc optimizations for different temporal path applications, and (3) must re-run the entire algorithm to calculate new results after each edge modification.

Transformation-Based Execution: The transformation-based execution [7], [14], [15] techniques first transform the original temporal graph to an equivalent but larger DAG (Directed Acyclic Graph) with timing information embedded (Fig. 1(c)), and then apply static graph models to the transformed DAG. Compared to the static execution, although introducing an additional transformation phase, the core graph execution is more straightforward without the need to deal with the additional time dimension. For the actual graph transformation, each edge will generate two vertices. For example, edge $u \rightarrow v : (u, v, s, t)$ will generate two vertices: $(u, s)_{out}$ (called an *out-vertex*) and $(v, t)_{in}$ (called an *in-vertex*). Each vertex in the transformed DAG has a label (u, t) where u is the *vertex instance* and t is the *time instance* (Fig. 1(c)). Formally, we define the graph transformation in the state-of-the-art transformation-based execution work *Trans* [15], [21]:

Step 1: Vertex Transformation. Suppose the original graph G is transformed into a new graph G'. Let $T_{in}[v]$ and $T_{out}[v]$ denote the sets of in-vertices and out-vertices in G' corresponding to the same vertex instance v in the original graph. Both sets are ordered by their respective time instances. For example, $T_{in}[v] = \{t_0, t_1, \ldots, t_n\}$ is sorted such that $t_0 < t_1 < \ldots < t_n$. All elements in $T_{in}[v]$ are distinct. The construction of $T_{out}[v]$ follows the same principle as $T_{in}[v]$. Consequently, the vertex set in G' is defined as $V' = \bigcup_{v \in V} T_{in}[v] \cup T_{out}[v]$.

Step 2: Edge Transformation. The transformation of edges is governed by three rules. *Rule 1* is employed to incorporate all edges from the original graph. *Rule 2* and *Rule 3* are devised to create new auxiliary edges. These rules help to eliminate the time constraints along the paths in the transformed graph, while ensuring the correctness of graph queries. Specifically, *Rule 2* generates auxiliary edges between in-vertices or out-vertices that have the same vertex label but different time instances. For example, the edge from $(a, 1)$ to $(a, 2)$ in Fig. 1 (both being out-vertices) is created under this rule. Meanwhile, *Rule 3* is responsible for creating edges that connect in-vertices to out-vertices, which again share the same vertex label but differ in their time instances. An example of this can be seen in Fig. 1 with the edge from $(b, 2)$ to $(b, 3)$ and the edge from $(c, 4)$ to $(c, 5)$.

- Rule 1: All edges in the original temporal graph G are directly included in G' by connecting the corresponding transformed vertices
- Rule 2: For each vertex v in the in-vertices set $T_{in}[v] = \{(v, t_1), (v, t_2), \ldots, (v, t_{k+1})\}$, we create a directed edge with weight 0 from (v, t_i) to (v, t_{i+1}) for each (v, t_i) where $1 \leq i \leq k$. Similarly, edges are created for the vertices in $T_{out}[v]$ following the same approach.
- Rule 3: For each $(v, t) \in T_{in}[v]$, if it can find a tuple $(v, t') \in T_{out}[v]$ satisfying that $t' = min\{t'' \mid (v, t'') \in T_{out}[v]\ t < t''\}$ and there does not exist any in-vertex $(v, t'') \in T_{in}[v]$ satisfying that $t < t'' \leq t'$, it will create a direct edge from (v, t) to (v, t').

Execution on Transformed Graph: With the transformation of the original graph into G', the time constraints of temporal paths become embedded in all paths of the transformed graph. Consequently, traditional static graph execution models, such as the priority-queue-based Dijkstra's algorithm [30], which utilizes a priority queue to maintain distances to vertices, can be

TABLE I
TIME AND SPACE COMPLEXITY COMPARISON

Algo.	One-Pass		Trans		A*		TeGraph+	
	Time	Space	Time	Space	Time	Space	Time	Space
Reach	$O(\|E\|)$	$O(\|E\|+\|V\|)$	$O(\|E\|)$	$O(\|E\|+\|V\|)$	$O(b^d)$	$O(\|E\|+\|V\|)$	$O(\|E\|)$	$O(\|E\|+\|V\|)$
FP	$O(\|E\|log(D))$	$O(\|E\|+2\|V\|)$	$O(\|E\|)$	$O(\|E\|+\|V\|)$	$O(b^d)$	$O(\|E\|+\|V\|)$	$O(\|E\|)$	$O(\|E\|+\|V\|)$
SP	$O(\|E\|log(D))$	$O(\|E\|+2\|V\|)$	$O(\|E\|log(\|E\|))$	$O(\|E\|+2\|V\|)$	$O(b^d)$	$O(\|E\|+\|V\|)$	$O(\|E\|)$	$O(\|E\|+\|V\|)$
KNN	$O(\|E\|log(D))$	$O(\|E\|+2\|V\|)$	$O(\|E\|log(\|E\|))$	$O(\|E\|+2\|V\|)$	$O(b^d)$	$O(\|E\|+\|V\|)$	$O(\|E\|)$	$O(\|E\|+\|V\|)$

directly applied to process G'. The correctness of this approach is established in [7]. Fig. 2(b) shows the updating process of edge $(b, c, 3, 4)$ in *Trans* [15], [21]. Unlike the static execution that requires both (b,2) and (b,3) to individually update (c,4) and (c,5), the update function here follows the path: from (b,2)->(b,3), (b,3)->(c,4), and then (c,4)->(c,5). Since the transformation is about sorting the edges by the temporal information, high-dimensional temporal information will slightly disturb the transformation. That is, we will sort the edges by the first dimension of the temporal vector. If the first dimension information is the same, we will move on to the second dimension and thereafter. As shown in Table I, for the reachability and fastest path, Trans uses Breadth-First Search (BFS) which only needs $O(|E|)$ time and $O(|E|+|V|)$ space. For the shortest path and Top KNN, the time complexity is $O(|E|log(|E|))$ and the space complexity is $O(|E|+2|V|)$. Note that shortest path and Top KNN need more space (e.g., priority queue) for the query.

For graph mutation, current transformation-based execution has to rebuild the transformed graph and then apply the static graph algorithm to it again for new results, resulting in high computational overhead.

A-Based Execution:* Under the transformed execution, some static graph searching algorithms such as A* [31] and A*-like algorithms [32] can be directly applied to the transformed graph for computations. A* uses an additive evaluation function $f(u) = g(u) + h(u)$ to search the transformed graph. $f(u)$ indicates the priority of each vertex u, $g(u)$ records the value of u starting from the source, and $h(u)$ represents the estimated value from u to the destination. During searching, it finds the vertex, which is yet searched and presents the highest priority, in the OPEN list, updates the functions (f, g, and h) of the vertex's neighbors, then moves the vertex into the CLOSED list. The time complexity of A* is $O(b^d)$ as shown in Table I, where b denotes the branching factor and d denotes the depth of the solution. While this time complexity is close to $O(|E|)$, it contains higher constants for involving more updates per computation, i.e., updating $f(u)$, $g(u)$ and $h(u)$, when compared to our design which only needs one update. The space complexity of A* is $O(|E|+|V|)$ for storing the graph and the vertex states [33]. For graph updating, the same as the static execution, A* also has to be re-run for each graph update.

Evolving Graph Processing Systems: Current evolving graph approaches primarily concentrate on dynamic edge modifications in graphs without temporal features, hence not addressing the time constraints inherent in temporal graphs. When it comes to edge modifications, evolving graph engines such as LLAMA [34], Ingress [20], KickStarter [11], and CommonGraph [17] employ incremental technologies for graph updates. However, they lack efficient strategies for incrementally updating transformed graphs, which is a crucial aspect in the context of temporal graph processing. LLAMA [34] excels at managing edge modifications in evolving graphs, but lacks capabilities for handling temporal features. In scenarios involving temporal graphs, LLAMA's multiversioned array system is not effective when compared against our work, i.e., *transformation-based execution* models. The reason is that LLAMA does not offer an efficient method for updating transformed graphs. Theoretically, LLAMA's design involves $O(|E|log(D))$ time complexity, while our TeGraph+ only requires $O(|E|)$. Ingress [20] is an automated system for incremental graph processing, utilizing efficient memoization policies that optimize memory use and select the best-fit policy for various graph algorithms. However, it does not include a strategy for updating the transformed graph. KickStarter [11] employs a dependency tree to track each vertex's value dependence, accelerating incremental graph query updates. CommonGraph [17] builds on KickStarter's approach by transforming edge deletion operations into several edge insertions, as deletions typically have higher time complexity. However, these engines are not equipped to handle the unique challenges of mutating temporal graphs. First, they are unable to address the time constraints associated with temporal paths. Second, in the context of using a transformation-based execution model, these systems lack an effective strategy to ensure the correctness of updating transformed graph which is detailed discussed in Section II-C.

C. Open Problems and Challenges

Although the state-of-the-art techniques have shown some promise for providing ad-hoc solutions to certain temporal path problems, a *fast, space efficient, bandwidth efficient, and scalable* solution still does not exist. We summarize the following four major challenges faced by the current approaches through both empirical results and theoretical analysis.

Performance Challenge: As discussed earlier, static execution approaches (e.g., *One-Pass*) incur both computational challenges and memory access overhead when handling the additional time dimension during execution. On the other hand, transformation-based execution significantly reduces computation complexity by processing transformed static graphs (DAGs), but suffers from the high latency of using traditional iterative static graph execution models (Dijkstra's or Bellman-Ford) and additional graph transformation overhead. More importantly, these previous approaches provide individual optimizations for specific path problems without a generic graph

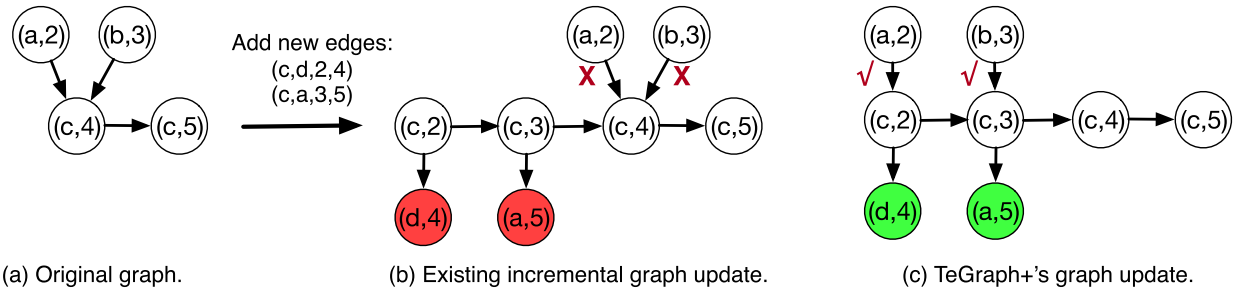


Fig. 3. Example of adding edges $(c, d, 2, 4)$ and $(c, a, 3, 5)$ in Fig. 1, comparing the application of existing incremental graph update technologies with TeGraph+'s graph update strategy.

format for achieving optimal performance across different general path problems. Table I demonstrates the theoretical time complexity across different designs on several path problems.

Space Challenge: Transforming temporal graphs into DAGs for processing is much more convenient without introducing time dimension constraints on execution. However, it suffers from the graph amplification challenge and additional overhead for the transformation process itself. For instance, under the traditional graph transformation strategy (e.g., *Trans*), the size of the vertices set increases from $|V|$ to $|E|$ and the edges set is doubled. Therefore, the space complexity of *Trans* in Table I is up to $O(4|E|)$ while TeGraph+ always enjoys $O(|E| + |V|)$ space complexity. The graph amplification poses significant performance and memory overhead for the execution phase afterward. First, it increases the base workload for graph processing. Second, it incurs large memory overhead which prevents large-scale temporal graphs from being efficiently executed on a single machine with out-of-core execution support. In our experiments, we observe an average of 2.5× and 20× enlargement for edges and vertices under *Trans*, respectively.

Bandwidth Challenge: Recent efforts that simply associate the temporal information to either traditional edge list or adjacency list formats would result in redundant data access. Using a time constraint-associated adjacency list as an example, since we put the neighbors of various time constraints together, at a certain time step, although some neighbors are already expired, this adjacency list format will still load the entire adjacency list of an active vertex into the cores for filtering and further processing. This will exacerbate the I/O intensity of temporal graph analytics.

Scalability Challenge: Current temporal graph research primarily focuses on optimizing strategies for specific execution settings but faces challenges scaling across different environments like in-memory, out-of-core, and distributed systems. This issue arises from the high time complexity of temporal graph algorithms, leading to a focus on algorithm optimization rather than environment adaptability. Additionally, most advanced methods fail to efficiently handle dynamic edge modifications. Both static execution and A* algorithms require re-running the entire algorithm for any updates, lacking an efficient strategy for incremental graph updates due to time constraints. Transformation-based methods also need to rebuild and reprocess the transformed graph for updates. Recent studies

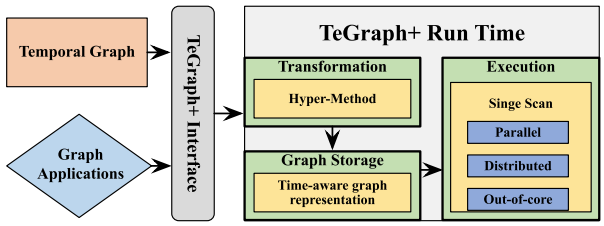


Fig. 4. TeGraph + execution flow diagram.

on evolving graphs [17], [20], [35] lack efficient strategies for incremental updates in transformed graphs of temporal graphs. When their existing incremental update techniques are directly applied to the transformed graphs of temporal graphs, it leads to challenges in maintaining the correctness of the final graph query results. For instance, in Fig. 3, when adding edges $(c, d, 2, 4)$ and $(c, a, 3, 5)$ to vertex c in Fig. 1, current evolving graph works simply apply incremental updates as depicted in Fig. 3(b). This approach results in incorrect query outcomes for vertices $(d, 4)$ and $(a, 5)$. This is because vertex $(d, 4)$ should be updated by vertex $(a, 2)$, and vertex $(a, 5)$ should be updated by both $(a, 2)$ and $(b, 3)$, but in Fig. 3(b), they are not updated by $(a, 2)$ and $(b, 3)$. Conversely, in Fig. 3(c), updated by TeGraph+, the graph update yields the correct final results.

Our Objective: To address these challenges, we introduce TeGraph+, a comprehensive framework designed for effectively handling temporal path problems and their applications in temporal graphs. Key features of TeGraph+ include a unique single-path execution model, a distributed processing model, a transformation model for graph data, an I/O-friendly graph representation, and a strategy for updating dynamic graphs. These features collectively enhance TeGraph+'s performance and ability to manage large graph sizes. Additionally, TeGraph+ is highly scalable, capable of adapting to various computing environments such as in-memory, out-of-core, and distributed systems, and supports updates to evolving graphs.

III. Implementation of TeGraph+

A. System Overview

Fig. 4 shows the overall architecture of TeGraph+, which consists of five major components: the *transformation phase*

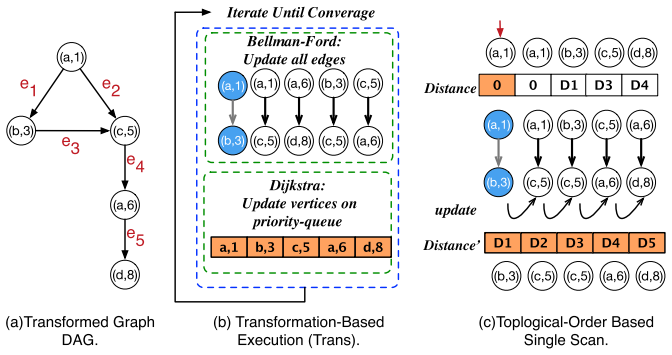


Fig. 5. Workflows of the single scan execution.

Algorithm 1: Single Scan Algorithm.

for all $v \in V$ **then**
  $d[v] = +\infty$ except $v = u$. $d[u] = 0$.
for all $e = (u', v) \in E$ in the topological order **then**
  if $d[v] > update(d[u'], e)$ **then**
    $d[v] = update(d[u'], e)$
return $d[v]$ for all $v \in V$

based on *hyper-method* (Section III-D), the parallel *execution phase* centering around the topological single scan (Section II-I-B) that supports a spanning tree decomposition (STD) based distributed execution strategy (Section III-C), the *time-aware* graph representation with the support of out-of-core single scan (Section IV), and the mixed imperative and lazy graph update strategy (Section V). Inside TeGraph+, a temporal graph is first transformed to a static DAG via our *hyper-method*, with new identifiers assigned to vertices and edges. After the transformation, a novel time-aware graph format is proposed to efficiently store the graph.

B. Execution Model

TeGraph [1] introduces a *universal single scan execution* and its parallelized implementation, based on the concept of *topological-optimum* on the transformed graph (DAGs). In the following, we will briefly describe these two executions. Because the topological order of the DAG is already contained by following the time instance order from the original temporal graph, it requires no additional cost for reordering DAG after transformation. Thus, a temporal path problem can be solved by *only a single round of scan over all the edges following the topological order of the transformed DAG* thanks to the time order of temporal graph. Fig. 5 and Algorithm 1 demonstrates how our single scan accesses the transformed graph in Fig. 5(a) merely once and finishes the entire computation. Fig. 5(b) illustrates how to apply state-of-the-art designs, e.g., Bellman-Ford [25] or priority-queue based Dijkstra's algorithms [30] to the transformed graph. Refer to TeGraph [1] for detailed explanations.

Despite that our single scan sequentially scans the graph and updates each edge following the order of the transformed graph (sorted by starting time of edges), it can support multithreads. Particularly, the parallelism opportunity surfaces when the edges

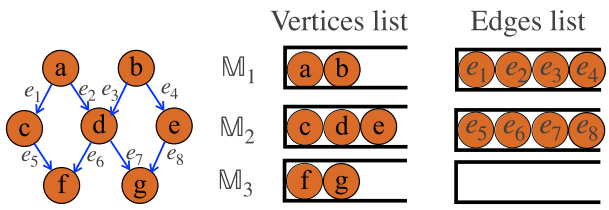


Fig. 6. A parallel single scan example.

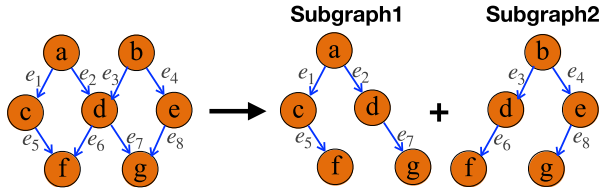


Fig. 7. A STD-based distributed single scan example.

can be updated independently. Formally, we define an order $\prec$ between two edges e and e' as Equation 3 to show that the update of edge e will affect that of edge e'. If $e \prec e'$, e must be updated before e'.

$$e \prec e' \Rightarrow \exists\, P \mid P = e \cdot e_1 \cdot e_2 \cdot \ldots \cdot e'. \tag{3}$$

Therefore, we can get an edges set $\mathbb{M}$ where all the edges in $\mathbb{M}$ are independent. Since our single scan is based on the topological-optimum which follows the topology order of the transformed graph (i.e., DAG), $\mathbb{M}$ can contain edges in the subgraph with their source vertices having no in-edges, i.e., in-degree number of source vertices are zero which can be implemented by creating an in-degree array for each vertex. Using Fig. 6 as an example, since the in-degree of vertex a and b is zero, $\mathbb{M}_1$ contains edges of a and b, that is, $\{e_1, e_2, e_3, e_4\}$. We can hence use four threads to work on these four edges in parallel. During computation, we further get $\mathbb{M}_2$ with $\{e_5, e_6, e_7, e_8\}$. Here, again, we can assign various threads to work on these edges in parallel. In summary, our single scan can be extended to support parallel processing.

C. STD-Based Distributed Single Scan

In this section, we propose our new spanning tree decomposition (STD) based distributed execution strategy with the help of the STD-based distributed single scan. In contrast to the shared-memory parallel single scan, the distributed single scan involves partitioning the transformed graph into distinct subgraphs to minimize network I/O. Decomposing the transformed graphs (DAGs) into spanning trees enables more efficient compression of transitive closures [36], allowing each computing node to work more independently and improving the scalability of distributed computing. Specifically, the STD-based distributed single scan algorithm splits the transformed graph (DAG) into a set of spanning trees based on time order, resulting in n spanning trees, e.g., $\{\mathbb{T}_1, \ldots, \mathbb{T}_n\}$. Each computing node is then assigned a spanning tree $\mathbb{T}_i$. As illustrated in Fig. 7, the DAG from Fig. 6 is broken down into two spanning trees. If the number of spanning

trees exceeds the number of computing nodes, we merge the first and last spanning trees ($\mathbb{T}_1$ and $\mathbb{T}_n$), the second and second to last ($\mathbb{T}_2$ and $\mathbb{T}_{n-1}$), and so on, until the number of trees matches the number of computing nodes. Conversely, if there are fewer spanning trees than computing nodes, we divide the spanning trees into smaller trees. For instance, when executed on a 4-node cluster, each spanning tree in Fig. 7 must be divided into two trees, e.g., the first spanning tree is separated into $\{a, c, f\}$ and $\{a, d, g\}$, while the second is split into $\{b, d, f\}$ and $\{b, e, g\}$. However, in real-world datasets, which are often large, the number of spanning trees usually exceeds the cluster size. To avoid replication computing, different spanning trees are designed not to share the same edges. Due to the topological characteristics of Directed Acyclic Graphs (DAGs), this splitting strategy ensures the correctness of the computation.

During distributed execution, each worker node carries out a single scan on its designated spanning tree. The updating operations within each node parallelize the single scan, as each spanning tree is also a DAG and can be executed concurrently. The primary difference is that after updating each edge, each computing node sends messages to the master node to combine the distance and in-degree values of each vertex. After the global distance is determined and the in-degree value of the current vertex is zero, each computing node proceeds to process the spanning trees within it.

1) Implementation Details: The core of the STD-based distributed execution in TeGraph+ is centered around its graph partitioning and execution processes. For graph partitioning, we have modified the partitioning component from existing distributed graph processing engines, such as Gemini [37]. This modification entails using a disjoint-set data structure to construct spanning trees for each computation node. The specifics of these modifications and their implementation are detailed as follows:

Construction of Spanning Trees: The construction of spanning trees can be managed using the disjoint set union algorithm. Since the edges of the transformed graph are generated in time order, we do not need to sort the edges or build the graph in memory. Instead, we use an array to record the root of each vertex. To achieve high scalability in distributed computing, we need to minimize the variance of different trees' weights, with each tree's weight being the sum of the ranks of its edges' order in the edge stream, which also represents the time instance ranks. This can be implemented by creating a priority queue to record the weight of each spanning tree (sum of normalization time instances), and in each round, inserting the edge into the spanning tree with the smallest weight. The total time complexity of the spanning tree construction is $O(|E|\log(K))$, with K being the number of spanning trees, which is close to a constant. Given that this construction can be employed for various applications, the time consumption can be averaged and considered negligible, as is common in static graph processing systems [3].

Execution Process: In managing the distributed single-scan algorithm, it is essential to include an in-degree array for each vertex. A vertex can update its neighboring edges using the single-scan algorithm only when its in-degree number reaches zero. The in-degree array is updated such that whenever an edge is processed, the in-degree of the destination vertex of that edge is decremented by one. For instance, upon updating the edge (u, v, s, t), the in-degree array of vertex (v, t) will be reduced by one. The process of updating and checking the in-degree array can efficiently utilize *OpenMPI* for message passing, ensuring effective communication and synchronization in the distributed environment.

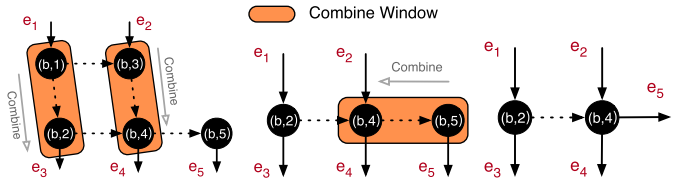


Fig. 8. Hyper-method illustration. Dash arrows indicate the time order, not the actual edges in the DAG.

D. Transformation Model

To tackle the graph amplification challenge, TeGraph+ has proposed a novel transformation model for further accelerating the execution phase (discussed previously) via *effectively condensing the transformed DAG size without any accuracy loss on graph information*. This can dramatically reduce the execution workload and memory footprint to increase the chance for in-memory processing with reduced disk I/O.

However, this transformation will result in many redundant vertices that are unnecessary for information propagation in temporal graph execution. Specifically, we find that *when in-vertices are used to propagate information to out-vertices, it can lead to redundant vertices and edges*. Fig. 8(a) demonstrates such redundancy in the state-of-the-art transformation (Section II-B). Based on this unique redundancy feature of temporal graph transformation, TeGraph+ puts forward the *hyper-method*, which is a new transformation method applied prior to the execution. It is used to reduce the size of the transformed graph by merging unnecessary vertices into the essential ones. Specifically, we define the original temporal graph as G and the transformed graph (DAG) as G'. We only pick the essential vertices for execution, called *hyper-vertices*, and the related vertices that are redundant are merged with them. Our hyper-method guarantees the topological structure of DAG and the hyper-vertices' target values are unchanged.

The term of *hyper-vertex* has been formally defined in TeGraph [1], where a hyper-vertex must be an out-vertex. The entire hyper-vertices set is defined as $T_{hyper}[u] = \{(u,t) : (u,t) \in T_{out}[u]\}$ and satisfies at least one of the following criteria:

- *Criterion 1:* $\forall\ (u, t_1) \in T_{out}[u],\ t \leqslant t_1$
- *Criterion 2:* $\exists (u, t_1) \in T_{in}[u],\ t_1 \leqslant t \Rightarrow \forall\ (u, t_2) \in T_{out}[u],\ t_2 < t_1$ *or* $t_2 \geqslant t$

Using Fig. 8(a) as an example, there are two in-vertices and three out-vertices. Vertex $(b, 2)$ is a hyper-vertex because it has the smallest time instance (Criterion 1). $(b, 4)$ is also a hyper-vertex because there is no out-vertex between $(b, 3)$ and $(b, 4)$ (Criterion 2). However, $(b, 5)$ is not a hyper-vertex. Thus, $T_{hyper}[b] = \{(b, 2), (b, 4)\}$. Furthermore, the merge operation

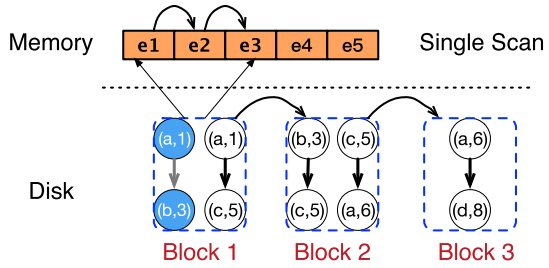


Fig. 9. TeGraph+ out-of-core data organization.

TABLE II
QUANTITATIVE ANALYSIS OF OUT-OF-CORE EXECUTION

Systems	TeGraph+	GridGraph	Wonderland
Complexity	min(|E|/S,|E|/C)	N*β*(|E|/S+|E|/C/T)	N*γ*(|E|/S+|E|/C/T)

for vertex $(u,t) \notin T_{hyper}[u]$ is performed if it satisfies at least one of the following conditions:

- *Merge Cond1:* When $(u,t) \in T_{in}[u]$, it can be merged to vertex $(u,t_1) \in T_{hyper}[u]$ if $t_1 = min\ \{t_i :\ (u,t_i) \in T_{hyper}[u]\ and\ t_i \geq t\}$
- *Merge Cond2:* When $(u,t) \in T_{out}[u]$, it can be merged to vertex $(u,t_1) \in T_{hyper}[u]$ if $t_1 = max\ \{t_i :\ (u,t_i) \in T_{hyper}[u]\ and\ t_i < t\}$

Merge Cond1 and Cond2 describe how an in-vertex and out-vertex is merged into a hyper-vertex, respectively. Fig. 8(b) is reduced from Fig. 8(a) by *Merge Cond1*. Fig. 8(c) is reduced from Fig. 8(b) by *Merge Cond2*. Refer to the previous paper TeGraph [1] for detailed explanations.

IV. TIME-AWARE GRAPH REPRESENTATION

To better support our single scan execution, TeGraph+ introduces a novel *time-aware graph data organization*, which regards identical vertex IDs with dissimilar time instances as different vertices. *This design decouples the connection between vertices with the same vertex label but with different time instances.* With this further transformation, the transformed graph can be viewed as a new static DAG.

Fig. 9 shows how to store Fig. 5(a) in an external-memory setting. Here we target external-memory setting because temporal graphs are often too large to fit in the main memory of the commodity computing systems. Basically, the entire graph is stored according to the increasing time instances. At each time step, identical vertices are grouped together. We further partition the graph into several blocks so that each block can fit into the memory. As shown in Algorithm 2, during out-of-core execution, TeGraph+ applies the parallel single scan to the just loaded block. To further hide the data transfer cost, we overlap the computing of one block with the transfer of another due to the fact that the execution and disk I/O are independent. Furthermore, there are two optimizations proposed to enable time-aware format conversion: *vertex grouping* to optimize the vertex transformation, and *parallel relabeling* to optimize the edge transformation. Refer to TeGraph [1] for more details about our time-aware graph representation.

Algorithm 2: Out-of-Core Execution.

$Blocks = \{B_1, B_2, .., B_k\}$, E is partitioned into k blocks
for all $B_i \in Blocks$ **then**
    Parallel single scan in B_i

A. I/O Complexity Analysis

This section quantitatively analyzes the I/O complexity of TeGraph+. Note that One-Pass and Trans do not provide out-of-core execution support due to the large memory overhead (Section II-C). For baseline, we choose *GridGraph* [4] and *Wonderland* [3] which are suitable for high diameter graphs because the transformed graph (DAG) often has a longer diameter. Specifically, we feed transformed graphs into these out-of-core graph engines and apply static graph algorithms in the transformed graphs. Note that they operate on the same transformed DAG as we do. Also, since other out-of-core engines such as LUMOS [38] are not optimized for high-diameter graphs, we do not analyze them here.

Table II shows the quantitative analysis on out-of-core execution performance of TeGraph+, *GridGraph* and *Wonderland*. $|E|$ represents the number of edges; S is the sequential read bandwidth of the disk; C is the CPU processing speed (edges/sec); N is the iteration number; β is the proportion of the average active graph data of GridGraph and γ for Wonderland; T is the thread number. We can ignore the abstraction computation overhead of Wonderland because the abstraction size is usually small. We observe that the out-of-core performance for all three is bounded by graph size, I/O bandwidth, and CPU speed.

Table II suggests that TeGraph+ always outperforms *GridGraph* and *Wonderland* in theory. Also, the performance of *GridGraph* and *Wonderland* is heavily impacted by the nature of the input graphs and algorithms, which is reflected in the parameters N, β, or γ. However, because of our hyper-method together with only a single round of sequential readings on the transformed graph, TeGraph+ significantly reduces the random memory accesses on edges and iterations (N). Furthermore, TeGraph+ can pipeline the loading of data blocks with the execution phase to hide I/O latency.

V. MIXED IMPERATIVE AND LAZY GRAPH UPDATE

This section introduces TeGraph+'s mixed imperative and lazy graph update strategy, designed to quickly and accurately update the transformed graph.

In TeGraph+, for each hyper-vertex, we create two queues to store the time instances of merged in-vertices and out-vertices, respectively. For each edge modification, we maintain the priority queues of affected vertices. In the following discussion, we will examine how to efficiently update the transformed graph and final results. When updating an edge (u,v,s,t) in the current graph, if the out-vertex (u,s) or in-vertex (v,t) already exists, the insertion or deletion process is skipped. Here, we will focus solely on cases where these vertices do not exist.

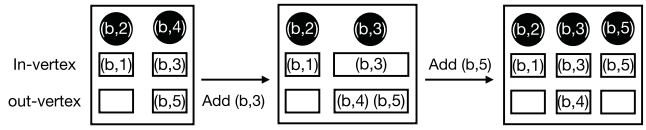


Fig. 10. Edge insertion process.

A. Imperative Edge Insertion

Inserting Out-vertex: Before vertex insertion, we first check whether (u, s) is a hyper-vertex (lines 31-32 in Algorithm 3). If it is, we do not need to modify the transformed vertex label. If not, we then insert the out-vertex (u, s). After vertex insertion, we verify whether (u, s) now meets either *Criterion 1* or *Criterion 2* from Section III-D (line 34). (1) If not, we employ *Merge Cond2* to merge (u, s) into the out-vertex queue of the hyper-vertex (u, s_1), which possesses the maximum time instance yet smaller than s. (2) If true, (u, s) becomes a new hyper-vertex, and we begin merging corresponding in-vertices and out-vertices into it. For in-vertices merging (lines 17–20), *Merge Cond3* merges some in-vertices with smaller time instances than s of the hyper-vertex, which have the minimized time instance larger than s, into the in-vertex queue of (u, s) ($Q^{in}_{(u,s)}$). Similarly, for out-vertices merging (lines 21–24), *Merge Cond4* merges out-vertices with time instances not smaller than s of the hyper-vertex, which has the maximum time instance but smaller than s, into the out-vertex queue of (u, s) ($Q^{out}_{(u,s)}$). Finally, if the insertion process causes another hyper-vertex (u, s_1) to no longer satisfy the criteria for being a hyper-vertex (*Criterion 1* and *Criterion 2*) as per *Merge Cond5*, then (u, s_1) ceases to be a hyper-vertex after the insertion. This determination is based on whether all time instances of in-vertices of (u, s_1) are not greater than s (as indicated in lines 26–27). In such a scenario, we transfer the out-vertex queue of (u, s_1), including the vertex itself, into the out-vertex queue of (u, s) ($Q^{\text{out}}_{(u,s)}$, as detailed in lines 28–29). Employing these three merging strategies, we can efficiently manage out-vertex insertions.

- *Merge Cond3:* For the hyper-vertex (u, s_1) satisfying that $s_1 = min\{s_i : (u, s_i) \in T_{hyper}[u],\ s_i > s\}$, each in-vertex $(u, s') \in Q^{in}_{(u,s_1)}$ with $s' \leq s$ will be merged into $Q^{in}_{(u,s)}$.
- *Merge Cond4:* For the hyper-vertex (u, s_1) satisfying that $s_1 = max\{s_i : (u, s_i) \in T_{hyper}[u],\ s_i < s\}$, each out-vertex $(u, s') \in Q^{out}_{(u,s_1)}$ with $s \leq s'$ will be merged into $Q^{out}_{(u,s)}$.
- *Merge Cond5:* For the hyper-vertex (u, s_1) with $s_1 = min\{s_i : (u, s_i) \in T_{hyper}[u],\ s_i > s\}$, if $\forall (u, s') \in Q^{in}_{(u,s_1)}$ satisfying that $s' \leq s$, then (u, s_1) will not be a hyper-vertex, then $Q^{out}_{(u,s_1)} \cup (u, s_1)$ and will be merged into $Q^{out}_{(u,s)}$.

Consider the example of inserting out-vertex $(b, 3)$. As shown in Fig. 10, it shows the created queues for hyper-vertices including the in-vertex queue and out-vertex queue. Initially, vertex $(b, 3)$ serves as an in-vertex, while $(b, 5)$ is the out-vertex. The in-vertex queue of $(b, 2)$ contains $(b, 1)$, and the in-vertex queue

Algorithm 3: Edge Insertion Procedure-Code.

1: **function** Criterion1Vertex (u,s)
2: **if** $\forall\ (u, t_1)\ \in\ T_{out}[u],\ s \leqslant t_1$ **then**
3: **Return True**
4: **Return False**
5: **function** Criterion2Vertex (u, s)
6: $s_1 = max\{s_i : (u, s_i) \in T_{in}[u],\ s_i \leq s\}$
7: $s_2 = max\{s_i : (u, s_i) \in T_{out}[u],\ s_i \leq s\}$
8: **if** $s_1 > s_2$ **then**
9: **Return True**
10: **Return False**
11: **function Merge1**Vertex (u, s)
12: $s_1 = min\{s_i : (u, s_i) \in T_{hyper}[u],\ s_i \geq s\}$
13: (u, s) is merged into $Q^{in}_{(u,s_1)}$
14: **function Merge2**Vertex (u, s)
15: $s_1 = max\{s_i : (u, s_i) \in T_{hyper}[u],\ s_i < s\}$
16: (u, s) is merged into $Q^{out}_{(u,s_1)}$
17: **function Merge3**Vertex (u, s)
18: $s_1 = min\{s_i : (u, s_i) \in T_{hyper}[u],\ s_i > s\}$
19: **for all** $(u, s') \in Q^{in}_{(u,s_1)}$ & $s' \leq s$ **then**
20: Merge (u, s') into $Q^{in}_{(u,s)}$.
21: **function Merge4**Vertex (u, s)
22: $s_1 = max\{s_i : (u, s_i) \in T_{hyper}[u],\ s_i < s\}$
23: **for all** $(u, s') \in Q^{out}_{(u,s_1)}$ & $s' \geq s$ **then**
24: Merge (u, s') into $Q^{out}_{(u,s)}$.
25: **function Merge5**Vertex (u, s)
26: $s_1 = min\{s_i : (u, s_i) \in T_{hyper}[u],\ s_i > s\}$
27: **if** $\forall (u, s') \in Q^{in}_{(u,s_1)}$ & $s' \leq s$ **then**
28: Delete (u, s_1) from $T_{hyper}[u]$
29: Merge $Q^{out}_{(u,s_1)} \cup (u, s_1)$ into $Q^{out}_{(u,s)}$
30: **function Insert_out_vertex**Vertex (u, s)
31: **if** $(u, s) \in T_{hyper}[u]$ **then**
32: **Return**
33: Add (u, s) to $T_{out}[u]$
34: **if** Criterion1((u, s))=True or Criterion2((u, s))=True **then**
35: Add (u, s) to $T_{hyper}[u]$
36: Merge3((u, s))
37: Merge4((u, s))
38: Merge5((u, s))
39: **else**
40: Merge2((u, s))
41: **function Insert_in_vertex**Vertex (u, s)
42: $s' = max\{s_i : (u, s_i) \in T_{hyper}[u],\ s_i \leq s\}$
43: $s_1 = min\{s_i : (u, s_i) \in Q_{(u,s')}out,\ s_i \geq s\}$
44: **if** $s_1 \neq NULL$ **then**
45: **if** $(u, s_1) \notin T_{hyper}[u]$ **then**
46: Add (u, s_1) to $T_{hyper}[u]$
47: Merge3((u, s_1))
48: Merge4((u, s_1))
49: Merge5((u, s_1))
50: **else**
51: Add (u, s) into $Q^{in}_{(u,s_1)}$
52: **else**
53: Merge1((u, s))
54: **function Insertion**Edge (u, v, s, t)
55: Insert_out_vertex((u, s))
56: Insert_in_vertex((v, t))

of $(b,4)$ stores $(b,3)$. The out-vertex queue of $(b,4)$ holds $(b,5)$, and $(b,2)$ does not possess any out-vertices. Upon inserting out-vertex $(b,3)$, it transforms into a hyper-vertex. We then apply *Merge Cond3* to remove in-vertex $(b,3)$ from the in-vertex queue of $(b,4)$ ($Q^{in}_{(b,4)}$) and place it into the in-vertex queue of $(b,3)$ ($Q^{in}_{(b,3)}$). Consequently, $(b,4)$ ceases to be a hyper-vertex, satisfying *Merge Cond5*. The out-vertex queue of $(b,4)$, along with $(b,4)$ itself, i.e., $Q^{out}_{(b,4)} \cup (b,4)$, should then be merged into $(b,3)$. Ultimately, the out-vertex queue of $(b,3)$ encompasses both $(b,4)$ and $(b,5)$. Note that the out-vertex queue of $(b,3)$ does not include $(b,3)$ itself, since each hyper-vertex also functions as an out-vertex; thus, merging each hyper-vertex into its out-vertex queue is unnecessary.

Inserting In-vertex: When inserting the in-vertex (v,t), we examine if it can generate a new hyper-vertex (lines 43–45 in Algorithm 3). This is determined by checking if there exists a hyper-vertex (u,s') such that $s' = \max\{s_i : (u,s_i) \in T_{hyper}[v] \ and \ s_i \leq s\}$ and there is an out-vertex (u,s_1) in the out-vertex queue of (u,s') meeting the condition $s_1 = \min\{s_1 : s_1 \geq t\}$ (lines 43–45). (1) If it cannot generate a new hyper-vertex, we merge (v,t) into the hyper-vertex according to *Merge Cond1* (lines 10–13), i.e., merging it into the hyper-vertex which has the smallest time instance but larger than t. (2) If inserting (v,t) can create a new hyper-vertex–meaning there is an out-vertex (u,s_1) which is not initially a hyper-vertex but satisfies *Criterion 2* after the insertion–it becomes a new hyper-vertex, and we merge the corresponding in-vertices and out-vertices into (u,s_1). In this case, we apply *Merge Cond3*, *Merge Cond4*, and *Merge Cond5* for the merging process. For example, when inserting the in-vertex $(b,5)$, as shown in Fig. 10, $(b,5)$ transforms into a new hyper-vertex. We then remove $(b,5)$ from the out-vertex queue of $(b,3)$ and insert it into its in-vertex queue.

Implementation Details: Each search and check operation, such as finding $s_1 = \min\{s_i : (u,s_i) \in T_{hyper}[u], s_i \geq s\}$ in the *Merge1* function and $s_1 = max\{s_i : (u,s_i) \in T_{in}[u],\ s_i \leq s\}$ of *Criterion2* function, can be efficiently handled using binary search with a time complexity of $O(\log(D))$. Additionally, the merging, deleting, and adding operations for in-vertex/out-vertex queues can be readily implemented using C++ STL containers with minimal overhead. Thus, despite the complexity of the insertion process, it can be executed with little time overhead.

B. Lazy Edge Deletion

Deleting Out-vertex: When deleting the out-vertex (u,s), we first ascertain whether (u,s) is a hyper-vertex (line 15 in Algorithm 4). (1) If it is not, we find the hyper-vertex (u,s_1) such that $(u,s) \in Q^{out}_{(u,s_1)}$, and then directly remove (u,s) from $Q^{out}_{(u,s_1)}$ (line 17). (2) If (u,s) is a hyper-vertex and will cease to be one post-deletion, we evaluate the potential creation of a new hyper-vertex by checking whether both the out-vertex and in-vertex queues of (u,s) are non-empty (line 19). If a new hyper-vertex, say (u,s_1), is formed, we apply *Merge Cond6* to merge the in-vertex and out-vertex queues of (u,s) into the new hyper-vertex (u,s_1) (lines 1–5). Conversely, if no new

Algorithm 4: Edge Deletion Procedure-Code.

```
1: function Merge6(Vertex (u, s))
2:     s_1 = min{s_i : (u, s_i) ∈ Q^out_(u,s)}
3:     Add (u, s_1) to T_hyper[u]
4:     Merge Q^in_(u,s) to Q^in_(u,s_1)
5:     Merge Q^out_(u,s) \ (u, s_1) to Q^out_(u,s_1)
6: function Merge7(Vertex (u, s))
7:     if Q^out_(u,s) = ∅ then
8:         s_1 = min{s_i : (u, s_i) ∈ T_hyper[u], s_i > s}
9:         Merge Q^in_(u,s) to Q^in_(u,s_1)
10: function Merge8(Vertex (u, s))
11:    if Q^out_(u,s) = ∅ then
12:        s_1 = max{s_i : (u, s_i) ∈ T_hyper[u], s_i < s}
13:        Merge Q^out_(u,s) ∪ (u, s) to Q^out_(u,s_1)
14: function Delete_out_vertex(Vertex (u, s))
15:    if Criterion1((u, s))=False &
       Criterion2((u, s))=False) then
16:        s_1 = min{s_i : (u, s_i) ∈ T_hyper[u], s_i ≥ s}
17:        Delete (u, s) from Q^out_(u,s_1)
18:    else
19:        if Q^out_(u,s) ≠ ∅& Q^in_(u,s) ≠ ∅ then
20:            Merge6((u, s))
21:        else
22:            Merge7((u, s))
23: function Delete_in_vertex(Vertex (u, s))
24:    s_1 = min{s_i : (u, s_i) ∈ T_hyper[u], s_i ≥ s}
25:    Delete (u, s) from Q^in_(u,s_1)
26:    if Q^in_(u,s_1) = ∅ then
27:        Delete (u, s_1) from T_hyper[u]
28:        Merge8(u, s_1)
29: function Deletion(Edge (u, v, s, t))
30:    Delete_out_vertex((u, s))
31:    Delete_in_vertex((v, t))
```

hyper-vertex emerges, we utilize *Merge Cond7* to combine the in-vertex queue of (u,s) with another hyper-vertex (u,s_1) (lines 6–9).

- *Merge Cond6:* If $Q^{out}_{(u,s)} \neq \emptyset$ and $Q^{in}_{(u,s)} \neq \emptyset$, for the out-vertex (u,s_1) satisfying $s_1 = min\{s_i : (u,s_i) \in Q^{out}_{(u,s)}\}$, (u,s_1) will become a new hyper-vertex, then $Q^{in}_{(u,s)}$ will be merged into $Q^{in}_{(u,s_1)}$ and $Q^{out}_{(u,s)} \setminus (u,s_1)$ will be merged into $Q^{out}_{(u,s_1)}$.
- *Merge Cond7:* If $Q^{in}_{(u,s)} = \emptyset$, for the hyper-vertex (u,s_1) with $s_1 = min\{s_i : (u,s_i) \in T_{hyper}[u],\ s_i > s\}$, $Q^{in}_{(u,s)}$ will be merged into $Q^{in}_{(u,s_1)}$.
- *Merge Cond8:* If $Q^{out}_{(u,s)} = \emptyset$, for the hyper-vertex (u,s_1) with $s_1 = max\{s_i : (u,s_i) \in T_{hyper}[u],\ s_i < s\}$, $Q^{out}_{(u,s)} \cup (u,s)$ will be merged into $Q^{out}_{(u,s_1)}$.

Using Fig. 11 as an example, which is based on Fig. 10, when removing the out-vertex $(b,3)$, its out-vertex queue is not empty, i.e., $Q^{out}_{(b,3)} \neq \emptyset$, satisfying *Merge Cond6*. Consequently, vertex $(b,4)$, the out-vertex with the smallest time instance in $Q^{out}_{(b,3)}$,

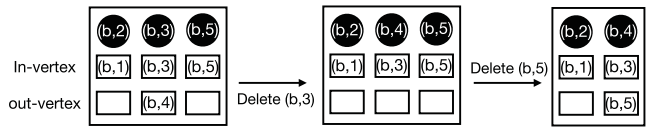


Fig. 11. Edge deletion process.

TABLE III
REAL-WORLD TEMPORAL GRAPH DATASETS (K = 10^3)

Graph	Vertices ($\|V\|$)	Edges ($\|E\|$)	$Avg(D)$	$Max(D)$
Flickr [39]	2.3M	33.1M	28.8	3.42E4
Growth [40]	1.8M	39.9M	42.7	2.27E5
Edit [41]	21.5M	266.8M	21.1	3.27E6
Delicious [42]	33.8M	301.2M	66.7	4.36E6
Twitter [43]	41.7M	1.47B	70.5	6.42E6
UK-Union [44]	134M	5.51B	70.3	3.04E6

becomes the new hyper-vertex. The in-vertex queue of $(b,3)$, denoted as $Q^{in}_{(b,3)}$, is then merged into $Q^{in}_{(b,4)}$. Additionally, the out-vertex queue of $(b,3)$, excluding $(b,4)$, i.e., $Q^{out}_{(b,3)} \setminus (b,4)$, is also merged into $Q^{out}_{(b,4)}$.

Deleting In-vertex: When deleting the in-vertex (v,t), we locate the hyper-vertex (u,s) such that $(v,t) \in Q^{in}_{(u,s)}$ and remove (v,t) from $Q^{in}_{(u,s)}$ (lines 24–25 in Algorithm 4). We then ascertain if $Q^{in}_{(u,s)} = \emptyset$. If it is not empty, (u,s) continues to be a hyper-vertex. Conversely, if it is empty, (u,s) ceases to be a hyper-vertex, and we employ *Merge Cond8* to combine its out-vertex queue with the corresponding hyper-vertex, such as (u,s_1) (lines 10–13), which can be determined by $s_1 = \max\{s_i : (u,s_i) \in T_{\text{hyper}}[u], s_i < s\}$. For example, in Fig. 11, upon removing the in-vertex $(b,5)$, we apply *Merge Cond8* to merge the out-vertex queue of $(b,5)$ into the out-vertex queue of the hyper-vertex $(b,4)$.

Lazy Operations: Since edge deletion operations generally exhibit higher time complexity than edge insertions [17], we adopt a lazy operation strategy to balance workloads. Unlike edge insertions, which immediately influence hyper-vertices, edge deletions can be deferred. This is because within each hyper-vertex, a deleted edge can be simply marked without immediate repercussions. For a batch of edge deletions, it's unnecessary to immediately remove each edge. Instead, if some deleted edges don't affect other edge insertions, we can mark their time instances as $+\infty$, signifying that these edges are no longer relevant for updating others. When deleted edges do impact the insertion process, we process these deletions collectively. This method enhances the overall efficiency of edge modifications. Importantly, the implementation details for edge deletion are akin to those for edge insertion.

C. Vertex Relabeling and Result Updating

After the transformed graph is updated, the updated vertex labels need to be relabeled. For example, in Fig. 8, vertex $(b,3)$ is relabeled as $(b,4)$. However, after the updating progress shown in Fig. 10, vertex $(b,3)$ becomes a hyper-vertex and is relabeled to its original label, $(b,3)$. For the vertex relabelling, we don't need to record all the relabeling logs of each vertex because each vertex only has one label. Instead, we can use an array to record the new label of each vertex and only change the label array of affected vertices.

When updating the final values of each vertex, we only need to incrementally update the affected vertices by Breath First Search (BFS). Specifically, we update the values of each hyper-vertex in affected queues by BFS following the topology order of the transformed graph (DAG) until there exist no hyper-vertices can be updated. Because the value of each hyper-vertex is the same as the value of the vertex in its queues (in-queues or out-queues), we only needs to update the value of hyper-vertices. Due to the topology feature of DAG which does not contain any cycles, the incremental result updating of DAG is similar to the dependence tracking strategy in KickStarter [11]. The only difference is that TeGraph+ does not need the additional overhead for constructing and maintaining a dependency tree.

VI. TeGraph+ Programming Interface

TeGraph+ provides intuitive programming model and APIs for users to easily express general path problems and their applications on temporal graphs via simple target function update as done in TeGraph [1]. Its key interfaces are described below.

In the framework of TeGraph+, each edge is denoted as a 5-tuple (src, dst, start, end, weight), representing the source and destination vertices, start and end time, and the weight for the edge. Each vertex is represented by a 2-tuple (vid, d), indicating its vertex instance and property (i.e., the target value of vid), respectively. The framework inputs sid, did, V and E, and returns the query answer from sid to did, where sid is the source vertex of each query, did is the target vertex, V represents the vertices set and E is the edges set. In general, the framework works by sequentially invoking the following interface functions: **TRANSFORM()**, **VINIT()**, **EMAP()**, **VAGGRE()**, **INSERT()**, and **DELETE()**. The detailed usage of APIs is presented in the previous version of the paper, TeGraph [1].

VII. Evaluation

A. Experiment Settings

Testbed: We use an 8-node cluster, each node using Duo Intel(R) Xeon(R) Processors E5-2640 v2 @ 2.00 GHz (8-cores/processor, total 16 threads) with 20 MB L3 Cache, 256 GB DRAM, and a 1 TB SSD drive with a throughput around 650 MB/s for sequential reads. The cluster is connected by 40 Gbps IB inter-connection. For in-memory processing, we employ OpenMP for multithreading (16 threads) on an individual node. In distributed mode, OpenMPI is used for inter-node message passing. For the out-of-core mode, we store the graph data in the disk and apply $cgroup$ to set various memory limits for execution on a single node.

Input Graphs: We selected six popular benchmarks in Table III for our evaluation. Edges in the temporal graph edge stream are sorted by their starting time. Note that we select to use large static graphs *Twitter* and *U.K.-Union* because publicly available temporal graphs are not large enough for evaluation.

For large static graphs which have not been attributed with time instances, we randomly allocate time instances to them according to the temporal graph format that is consistent with the state-of-the-art works. Main attributes of these graphs are listed in Table III. Note that $Avg(D)$ and $Max(D)$ represent the average and maximum vertex degree, respectively. They are important attributes for applications whose processing overhead can be significantly affected by the vertex degrees.

Applications: We evaluate four typical temporal path applications: reachability, fastest path, shortest path, and top-k nearest neighbors. For the top-k nearest neighbors, we use the shortest path to calculate the top k nearest neighbors and choose k as $|V|/10$. We randomly select 100 vertices as input and report the average execution time for each application (single-source query).

Evaluation Methodology: Traditional static graph computation models such as vertex- or edge-centric approaches will lead to high overhead because they require additional efforts to process time constraints recorded on the topological structure. When applied to time-aware format in Section IV, we find TeGraph+ can work with both approaches. We evaluate the performance of TeGraph+ under three modes:

1) *The full memory mode* puts all the graph data inside the memory. The full memory mode demonstrates the efficiency of TeGraph+ compared to three temporal graph execution models including static execution (*One-Pass* [7]), transformation-based execution (*Trans* [21]), and *A** with the transformed graph of *Trans* as the input. We evaluate the performance improvement from each component in TeGraph+ including parallel single scan, hyper-method-based transformation, and time-aware graph transformation. Not only this, but we also test the performance of edge modifications with the input of a batch of edge modification operations.

2) *The distributed mode* partitions the DAG into several spanning trees and each computing node holds several spanning trees according to Section V. Under this mode, we compare TeGraph+ with *GraphScope* [18] which is the state-of-the-art distributed static graph processing system. The input of GraphScope is the transformed graph of TeGraph+ and we report the evaluations under a different number of nodes.

3) *The out-of-core mode* uses one SSD drive to hold the graph data. Under this mode, we compare TeGraph+ with *GridGraph* [4], *Wonderland* [3], *Blaze* [45], *FlashGraph* [46], and *Graphene* [47]. We use their most current official versions. For fairness, all engines are fed with the same transformed graphs constructed by hyper-method. We report the execution time breakdown for both graph partition and execution.

B. Overall Performance: Full Memory Mode

In this section, we analyze and compare the performance of various designs in full memory mode. Our execution time comparison includes several datasets listed in Table III, except for the *U.K.-Union* dataset, which is too large (88 GB) for in-memory testing. Using 16 threads for these tests, TeGraph+ consistently outperforms other methods like *One-Pass*, *Trans*, and *A** across all datasets. The speedup ranges from 1.69 to a notable 241.18 times. For a more in-depth look at in-memory execution, see the TeGraph study [1].

Reachability: Since the algorithm for reachability used in Trans and A* is the same as the fastest path, we only compare TeGraph+ and *One-Pass* here. Table IV shows that TeGraph+ can achieve a speedup up to $16.74\times$ over One-Pass. Although both designs have the same theoretical time complexity (Table I), TeGraph+ outperforms One-Pass in three aspects. First, due to the topological-optimum nature, TeGraph+ only needs to use bitwise operations to update an edge, i.e., $d[v] \mid = d[u]$ for (u, v, s, t), while One-Pass requires more operations for updating, i.e., $if\ d[u] \leq s,\ d[v] = min\{d[v],\ t\}$. Second, in reachability, TeGraph+ only requires processing two variables for each edge (the source and destination vertex labels) while One-Pass requires four (i.e., start and end time additionally) because of the extra time dimension. Third, although TeGraph+ benefits from our multithreading design (e.g., $3\times$ to $4\times$ under 16 threads), we observe that the parallel versions for One-Pass and Trans [21] both perform worse than their single-thread versions due to the large message-passing overhead.

Fastest Path: TeGraph+ processes tasks significantly faster than One-Pass, with speeds ranging from 26.71 to 112.61 times faster across various workloads. While One-Pass has a time complexity of $O(|E|log(D))$, TeGraph+ operates at $O(|E|)$, gaining more advantage with larger graphs due to the increasing value of $log(D)$. Against Trans, a BFS-based algorithm for the fastest path, TeGraph+ achieves a speedup of 7.65 to 16.30 times. This is because Trans spends a considerable amount of time (up to 36%) maintaining each vertex's degree, and its transformed graph is 1.84 to 2.56 times larger than TeGraph+'s, leading to more computation and cache misses. Additionally, TeGraph+'s parallel scan operation further enhances its speed by 2.69 to 4.11 times compared to single-threaded processing. Compared to A*, TeGraph+ is 3.75 to 13.04 times faster. A* requires updating three functions for each neighbor during updates, while TeGraph+ needs only one operation per edge update. Though both have nearly $O(|E|)$ time complexity, A* has larger constants, and its transformed graph size is the same as Trans, which is on average 2.09 times larger than TeGraph+'s. These factors contribute to TeGraph+'s superior performance.

Shortest Path: TeGraph+ achieves significant performance gains over both *One-Pass* and *Trans:* up to $90.48\times$ and $241.18\times$, respectively. The performance gap comes from TeGraph+'s lower time complexity which reduces from $O(|E|log(D))$ (*One-Pass*) and $O(|E|log(|E|))$ (*Trans*) to $O(|E|)$ as shown in Table I. When compared to *A**, TeGraph+ can get $4.05 \sim 17.36\times$ speedup. The improvement comes from the smaller computation constants, transformed graph of TeGraph+ over A*, and the parallel single scan of TeGraph+. It needs to be stressed that even using the same transformed graph as TeGraph+, A* still takes $8.74\times$ longer time than TeGraph+.

Top-K Nearest Neighbors: Its computation consists of two portions: calculating the shortest path for all vertices and using a priority queue to find k vertices with the smallest shortest path. Table IV shows that compared to *One-Pass*, *Trans*, and *A**, TeGraph+ achieves up to a $39.74\times$, $104.99\times$, and $8.08\times$ speedup,

TABLE IV
OVERALL EXECUTION TIME IN SECONDS

dataset	reachability		fastest path				shortest path				top k nearest neighbors			
	TeGraph+	One-Pass	TeGraph+	One-Pass	Trans	A*	TeGraph+	One-Pass	Trans	A*	TeGraph+	One-Pass	Trans	A*
Flickr	**0.0086**	0.0907	**0.0987**	2.6368	0.7546	0.3697	**0.0986**	1.2522	5.1928	0.3998	**0.4324**	1.5859	5.5265	0.7336
Growth	**0.0162**	0.1794	**0.1446**	4.5404	1.1879	1.0572	**0.1458**	2.8376	20.610	1.3602	**0.4985**	3.1903	20.962	1.7130
Edit	**0.1131**	1.6978	**0.6184**	64.574	6.1619	5.6689	**0.6213**	42.793	133.67	9.8426	**4.2825**	46.455	137.33	13.504
Delicious	**0.1942**	3.3769	**1.1095**	117.60	15.798	13.428	**1.1140**	89.362	254.06	15.243	**7.0614**	95.310	260.01	21.191
Twitter	**1.3613**	22.794	**6.1952**	697.64	100.98	80.785	**6.2266**	563.41	1501.7	108.11	**14.380**	571.56	1509.9	116.27

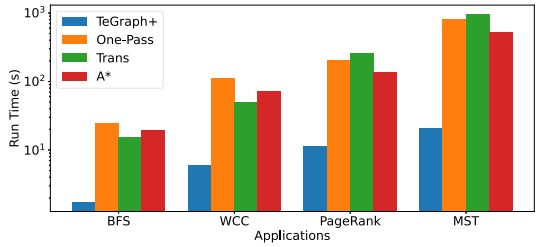


Fig. 12. More applications evaluation.

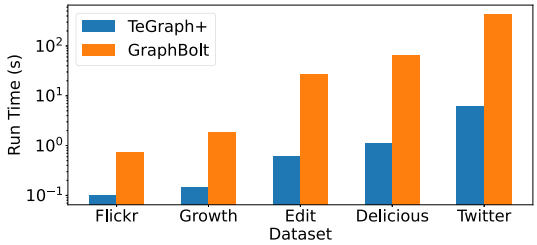


Fig. 13. Comparison with streaming graph engine.

respectively. The reasons for these speedups are analogous to the reasons in the shortest path.

End-to-end Comparison: Although graph transformation can be used for all applications to amortize the graph transformation time, under the end-to-end comparison, TeGraph+ can still get up to $1.17 \sim 7.18\times$ speedup than One-Pass which does not need the preprocessing. This is because the end-to-end execution only needs $O(|E|)$ time complexity, whereas One-Pass needs $O(|E|log(D))$ time complexity.

More Applications: To further assess the efficiency of TeGraph+, we have included four more applications: BFS (Breadth-First Search) [14], WCC (Weakly Connected Component) [48], PageRank [49], and MST (Minimum Spanning Trees) [50]. Each of these applications is tailored for temporal graphs and must adhere to time constraints on temporal paths. Given their reliance on graph path propagation, these applications can benefit from our transformed graph model and single scan algorithm. As illustrated in Fig. 12, we present the performance evaluation of TeGraph+ against baselines for these applications using the Twitter dataset. The results demonstrate that TeGraph+ outperforms the baselines on these additional applications, thanks to its efficient graph transformation model and algorithms that effectively address temporal path-related challenges. Overall, TeGraph+ achieves a speedup of up to $45.2\times$ on these applications.

Comparison with Streaming Graph Engine: Here, we conduct an evaluation against the streaming graph engine GraphBolt [10]. To enable a comparison with GraphBolt, we modify the edge updating function to adhere to time constraints. This is achieved by employing a muti-version array that records the distance of each vertex at different time instances, akin to the One-Pass approach. Fig. 13 presents a performance comparison between TeGraph+ and GraphBolt, using various graph datasets with the shortest path as the application. It is evident that TeGraph+ achieves a speedup ranging from 7.34 to 69.69 times compared to GraphBolt. This significant improvement stems from TeGraph+'s graph transformation-based execution model, coupled with its efficient temporal graph algorithms. In contrast, GraphBolt lacks optimized algorithms to effectively handle time constraints along temporal paths.

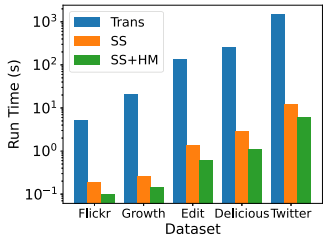


Fig. 14. Breakdown.

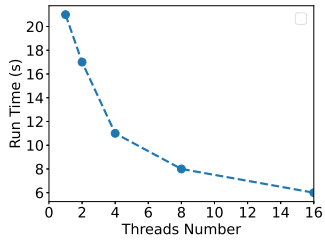


Fig. 15. Scalability.

C. Piecewise Breakdown

To evaluate the performance improvement gained through our optimizations, topological single scan (multithreading) and hyper-method, we compare the execution phase of TeGraph+ with *Trans* under different optimizations, i.e., only single scan (SS) and single scan together with hyper-method (SS+HM) shown in Fig. 14. Then we will show the scalability of our parallel single scan method in Fig. 15.

(I) Impact of Topological Single Scan: As depicted in Fig. 14, TeGraph+'s topological single scan offers a remarkable speedup, reaching up to 124 times faster than the priority-queue based

TABLE V
TRANSFORMED GRAPH SIZE (K = 10^3)

	TeGraph+		Trans	
Dataset	|V|	|E|	|V|	|E|
Flickr	5556k	36732k	38718k	69555k
Growth	5789k	47980k	50351k	88433k
Edit	31319k	277170k	374866k	620193k
Delicious	42036k	310319k	529894k	797300k
Twitter	75792k	1502505k	1508208k	2934921k
UK-Union	295037k	5805037k	6339373k	11849373k

Dijkstra's algorithm used in Trans, particularly evident on the *Twitter* dataset. This performance enhancement is also notable on smaller datasets like *Flickr*, with a speedup of approximately 27.8 times. Such significant improvements primarily result from reducing the time complexity from $O(|E|log(|E|))$, typical of priority-queue based Dijkstra's algorithm, to $O(|E|)$ in TeGraph+'s single scan. This approach also effectively transforms random memory accesses into sequential ones, thereby reducing cache misses. The scalability impact of multithreading on our single scan is demonstrated using the *Twitter* dataset in Fig. 15. Despite additional requirements for edge updates and maintaining vertex degrees (as outlined in Section III-B), resulting in more operations including atomic updates for correctness, our approach still shows a substantial speedup, ranging between 2.69 to 4.11 times.

(II) Impact of Hyper-Method: The Hyper-method is designed to minimize redundancy, thereby reducing the transformed graph's size and memory usage, and enhancing the core execution's efficiency with a lighter workload. According to Table V, TeGraph+, when compared to Trans, decreases the total count of edges and vertices by up to 61% and 94.9%, respectively. This reduction in the number of edges directly impacts processing time, given TeGraph+'s $O(|E|)$ time complexity. Simultaneously, the reduction in vertex count aids out-of-core execution, as it increases the likelihood of holding vertices in main memory. Illustrating the hyper-method's effect on core execution, specifically for the shortest path problem, Fig. 14 reveals that the hyper-method contributes to a performance enhancement of up to 2.6 times compared to the version of TeGraph+ without it.

D. Time-Aware Graph Transformation

We tested the effectiveness of our approach to transform temporal graphs into a time-aware format, comparing it to the transformation phase in Trans. Specifically, we focused on the overall execution time, including both vertex and edge transformation steps. Our method uses vertex grouping and parallel relabeling to speed up these steps. Results show that our vertex grouping method, which optimizes vertex transformation, can be up to 18 times faster than the sorting method used in Trans. This improvement is more pronounced with larger datasets, as our vertex grouping method has a near $O(|E|)$ time complexity, compared to Trans's $O(|E|log(D))$ complexity. For edge transformation, our parallel relabeling approach, implemented with `OpenMP` in `C++`, shows excellent scalability, achieving nearly 15 times speedup with 16 threads. Overall, our method,

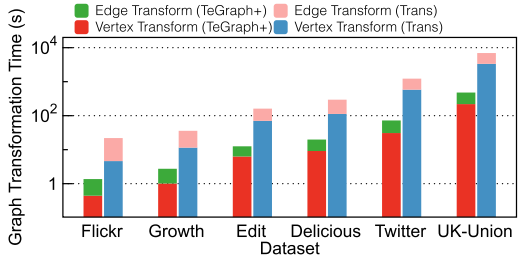


Fig. 16. Graph transformation time breakdown.

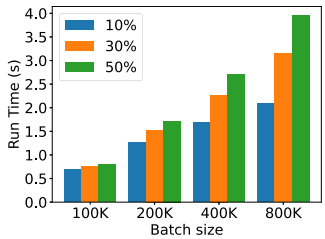


Fig. 17. Graph mutation cost.

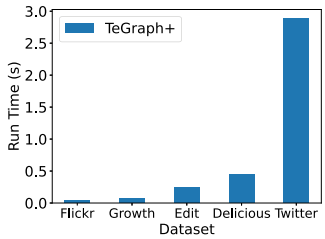


Fig. 18. Result update.

TeGraph+, achieves up to a 16.97 times speedup in graph transformation compared to Trans.

E. Graph Update Performance

In this section, we present the evaluation of mixed imperative and lazy graph update, which includes edge insertion and edge deletion, as depicted in Fig. 17. The input consists of a batch of edge modifications on the Twitter graph, with edge deletions accounting for 10%, 30%, and 50% of the total edge modifications. We choose batch sizes of 100 K, 200 K, 400 K, and 800 K, with the application focusing on the shortest path.

As demonstrated in Fig. 17 and in line with traditional evolving graph research [11], edge deletion in temporal graphs exhibits higher time complexity, as deleting edges can lead to an increased number of merge operations in vertex queues. Compared to 10% edge deletion, 30% and 50% edge deletions take up to 1.51× and 1.79× longer, respectively. As the batch size increases, the time cost also increases, but not linearly with respect to the batch size, as some edges connected to the same hyper-vertex can be updated simultaneously.

Fig. 18 displays the evaluation of incremental result updating after 100 K edge modifications. It reveals that, compared to the rebuild algorithm strategy (a single source shortest path query),

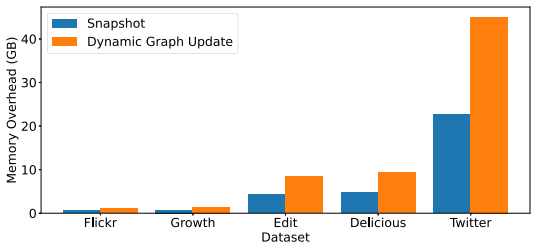


Fig. 19. Memory usage comparison.

incremental computation requires significantly less time, saving up to 2.49× in computation time. This time saving is attributed to the fact that the updated graph part is a sub-DAG of the original graph, which can be updated incrementally more quickly.

Finally, in Fig. 19, we present the memory usage for both dynamic graph update and static graph execution (the latter does not require any graph update, as in the previous paper version titled TeGraph). It is observed that the dynamic graph update stage incurs minimal additional memory overhead, as the total in-vertex and out-vertex queues together occupy about $O(|E|)$ memory size. Consequently, the overall memory overhead is linear with respect to $|E|$ and has small constant factors. This results in the memory usage for dynamic graph update being nearly twice as large as that for static graph execution. The reason is that static graph execution only requires storage for the transformed graph, unlike the dynamic graph update stage, which also needs to store the vertex queues.

F. STD-Based Distributed Execution

To demonstrate the efficiency of TeGraph+'s STD-based distributed execution, we compare its overall performance, scalability, network I/O time, and computation time with TeGraph+'s random-based distributed model (T-Random) and the state-of-the-art graph processing engine GraphScope [18] in Fig. 20. The random-based distributed model refers to TeGraph+ partitioning the transformed graph randomly into N subgraphs for distributed computation across N nodes. We will discuss these comparisons in detail.

First, spanning from Fig. 20(a) to (e), we evaluate the performance of TeGraph+'s STD-based and Random-based distributions in comparison to GraphScope. This assessment utilizes various graph datasets and is conducted under an 8-node distributed computation setting for all applications. The STD-based method of TeGraph+ achieves a speedup of $1.10 \sim 1.57\times$ over GraphScope and $1.31 \sim 1.72\times$ over the random-based method. This is due to TeGraph+'s efficient graph algorithm, which only requires $O(|E|)$ time complexity. In contrast, the random-based method incurs significant network communication overhead, resulting in slower performance than both STD-based and GraphScope. The specifics of network I/O will be analyzed later.

Second, Fig. 20(f) tests the scalability of TeGraph+ and baselines under varying node counts. For TeGraph+, increasing node numbers leads to reduced parallelism due to heightened network communication for propagating vertex values and degrees. Despite this, TeGraph+ still outperforms configurations with fewer

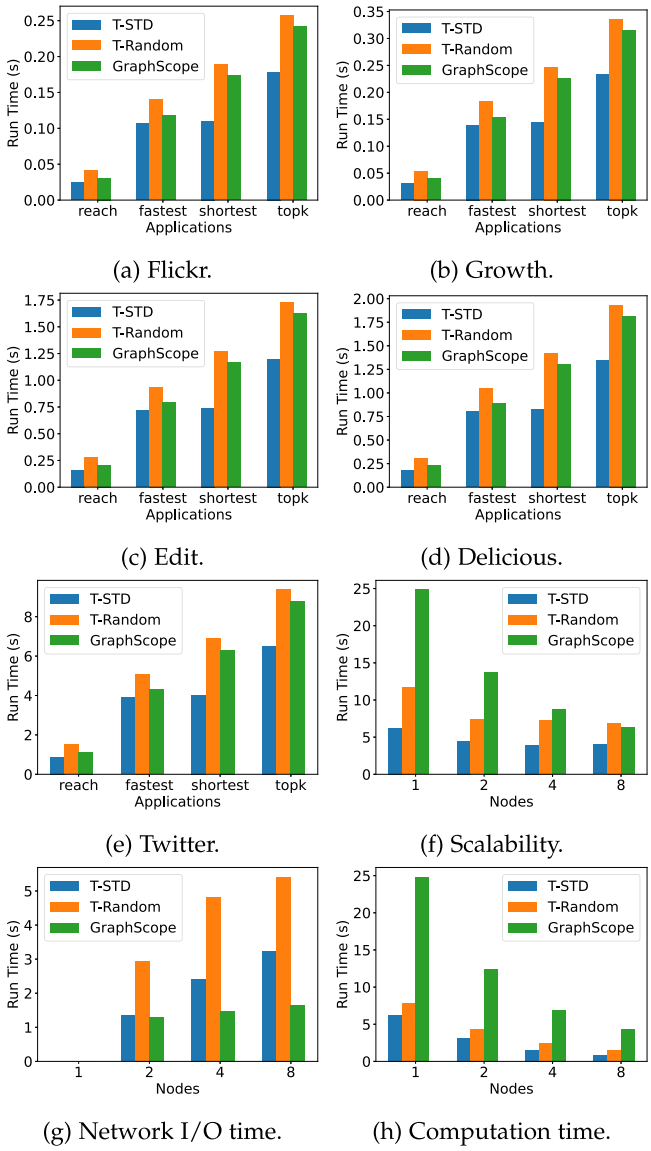
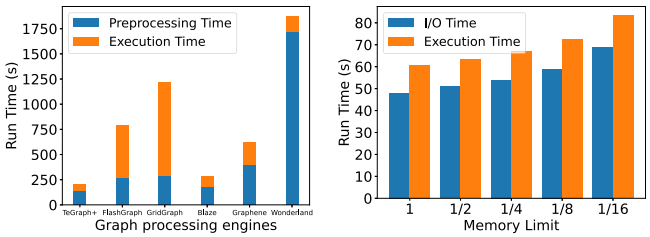


Fig. 20. Distributed execution evaluation.

nodes. When compared to GraphScope, TeGraph+ achieves speedups from 1.57 × to 3.99× in different node configurations, credited to its superior graph algorithm and scalable distributed execution model.

Finally, we break down the distributed execution time into network I/O time (Fig. 20(g)) and computation time (Fig. 20(h)) under different computation nodes. TeGraph+ experiences higher network communication as each node communicates to maintain the in-degree array of each vertex. The communication overhead is up to $O(|E|)$. In computation time, TeGraph+, with both STD-based and random-based methods, incurs less overhead than GraphScope due to its efficient single-scan algorithm.

Limitation Discussion: As per our evaluation, TeGraph+'s bottleneck in distributed execution lies in network I/O, particularly for a transformed graph, which is a DAG. TeGraph+ must maintain an in-degree array for each vertex, updating edges only when certain vertices' in-degrees drop to zero. This process



(a) Breakdown. (b) Memory size sensitivity.

Fig. 21. Out-of-core evaluation.

resembles distributed topology sorting on DAGs, highlighting a crucial area for further optimization.

G. Out-of-Core Execution

Here, we experimentally evaluate TeGraph+ against five state-of-the-art out-of-core engines on the *U.K.-Union* dataset, *GridGraph* [4], *Wonderland* [3], *Blaze* [45], *FlashGraph* [46], and *Graphene* [47].

For a fair evaluation, we provide all engines with the same transformed graph from TeGraph+ and record both preprocessing and execution times. The preprocessing time here refers to organizing the original graph data into a new format for storage on disk. This includes breaking the graph data into multiple blocks by 2D partition for GridGraph, Graphene, and FlashGraph; 2D graph partitioning plus abstraction finding (for iteration reduction) for Wonderland; page-interleaved based graph format for Blaze; and 1D graph partitioning for TeGraph+. As depicted in Fig. 21(a), the performance comparison includes a two-phase breakdown. Overall, TeGraph+ surpasses other engines in end-to-end time.

In terms of preprocessing, except for Wonderland, engines only require linear time complexity, resulting in fairly similar preprocessing times. However, Wonderland demands significantly longer time for graph partitioning, involving both 2D partitioning and abstraction generation with log time complexity due to sorting graph edges. Compared to Wonderland, TeGraph+ achieves an $11.78\times$ speedup in preprocessing time.

During execution, unlike TeGraph+ which reads edges and modifies vertex properties only once, other engines iteratively read and modify until convergence. This iterative approach is less efficient for long diameter graphs like the transformed DAGs used here, as more iterations increase disk I/O access. In contrast, TeGraph+'s single-read approach has disk I/O linearly proportional to $|E|$. Consequently, these engines underperform TeGraph+ in graph execution. TeGraph+ outperforms these engines with a speedup ranging from 1.73 to $15.11\times$.

To delve deeper into the out-of-core execution, we analyze both the disk I/O and the performance of TeGraph+ under varying memory sizes in Fig. 21(b). We use '$cgroup$' to restrict memory usage across all three engines, evaluating them at memory capacities of S, $S/2$, $S/4$, $S/8$, and $S/16$, where S represents the size of the transformed graph. Disk I/O accounts for 79% of TeGraph+'s total execution time, compared to only $47.9\% \sim 74.5\%$ for other engines. This higher percentage in TeGraph+ is attributed to its faster execution algorithm. The I/O overhead for TeGraph+, at $O(|E|)$, is minimally influenced by graph structure, as its single-scan approach requires just one pass through the graph data, whereas other engines necessitate multiple iterations until convergence, leading to repeated disk I/O.

TeGraph+'s performance is only slightly impacted by memory size variations. The differences in performance across different memory limits are primarily due to disk I/O caused by the vertex array, which records the distance value of each vertex. As for loading graph data, the I/O operations are quite similar across the engines since they all load graph data once, constituting the majority of disk I/O.

VIII. Related Work

The existing temporal path problems are solved by using ad-hoc solutions or application-specific optimizations. There are some other applications that adopt temporal path problems as their core component including betweenness centrality [51], minimum spanning tree problem [50], and random walk [52]. These applications are all based on path problems, thus they can all benefit from our TeGraph+.

For temporal graph processing, several static-execution based engines [6], [28], [29] have been proposed, but they are slower than our static-execution baseline One-Pass due to high time complexity of $O(|E|Dlog(|V|D))$. Some distributed works have been proposed such as GRAPHITE [53] which optimizes temporal partitioning and message passing, but they mainly focus on the system-level optimizations without fundamentally reducing algorithms' time complexity. Some temporal graph database systems [26] proposed to mainly focus on traversal language which provides the temporal syntax for temporal graph otherwise algorithm and system-level optimizations of temporal graph applications.

For domain-specific hardware accelerator-based temporal graph processing like SaGraph [54], it coordinates the graph traversals of different instances and adaptively caches frequently accessed vertices' states in on-chip memory, thereby reducing off-chip communication and enhancing data locality. However, SaGraph primarily focuses on system-level and architecture-level optimizations and lacks algorithm-level enhancements. Specifically, it does not feature efficient temporal graph algorithms for rapidly addressing temporal path problems. Conversely, TeGraph+ emphasizes algorithm-level optimizations, such as introducing the hyper method for graph transformation and a single scan approach for execution. Building upon these algorithmic advancements, TeGraph+ further develops system-level optimizations to create a comprehensive framework for temporal graph processing.

In the realm of evolving graph processing, earlier works like Tornado [55] and Naiad [56] only support incremental computation for edge insertion. More recent approaches, such as KickStarter [11] and GraphBolt [35], have expanded their capabilities to handle both edge insertion and edge deletion. CommonGraph [17] improves upon KickStarter by converting

edge deletion into edge insertion. Additionally, there are works like SGraph [19] that intend to enhance point-to-point query performance over evolving graphs with the help of trimming strategies such as bound estimation. However, all these works focus on graphs without temporal features and can not handle the time constraints on temporal paths efficiently.

We also notice that several studies have proposed specific optimizations on static graph reduction [57]. However, they cannot replace our hyper-method because hyper-method guarantees accuracy and correctness without losing any information, while almost all these reduction methods can only perform approximation.

IX. Conclusion

In this paper, we propose TeGraph+, the first general-purpose temporal graph computing engine that offers support for flexible edge insertion and deletion, to enable efficient processing of temporal path problems and their applications. The core of TeGraph+ is its temporal information-aware graph format, novel single-pass execution workflow, information redundancy removing transformation models, scalable distributed execution model, and efficient graph update strategy. Taken together, TeGraph+ can achieve up to 61% disk space-saving, up to 17× graph transformation speedup, and up to two orders of magnitude performance speedup over the state-of-the-art designs.

References

[1] C. Huan et al., "TeGraph: A novel general-purpose temporal graph computing engine," in *Proc. 38th IEEE Int. Conf. Data Eng.*, 2022, pp. 578–592.

[2] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *Proc. 10th USENIX Symp. Operating Syst. Des. Implementation*, 2012, pp. 31–46.

[3] M. Zhang et al., "Wonderland: A novel abstraction-based out-of-core graph processing system," in *Proc. 23rd Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2018, pp. 608–621.

[4] X. Zhu, W. Han, and W. Chen, "GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 375–386.

[5] P. Holme and J. Saramäki, "Temporal networks," *Phys. Rep.*, vol. 519, no. 3, pp. 97–125, 2012.

[6] W. Han et al., "Chronos: A graph engine for temporal graph analysis," in *Proc. 9th Eurosys Conf.*, 2014, pp. 1:1–1:14.

[7] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu, "Path problems in temporal graphs," in *Proc. VLDB Endowment*, vol. 7, no. 9, pp. 721–732, 2014.

[8] S. Kumar, X. Zhang, and J. Leskovec, "Predicting dynamic embedding trajectory in temporal interaction networks," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, A. Teredesai, V. Kumar, Y. Li, R. Rosales, E. Terzi, and G. Karypis, Eds., 2019, pp. 1269–1278.

[9] Y. Zheng, Z. Li, J. Xin, and G. Zhou, "A spatial-temporal graph based hybrid infectious disease model with application to COVID-19," in *Proc. 10th Int. Conf. Pattern Recognit. Appl. Methods*, 2021, pp. 357–364.

[10] M. Mariappan and K. Vora, "GraphBolt: Dependency-driven synchronous processing of streaming graphs," in *Proc. 14th EuroSys Conf.*, G. Candea, R. van Renesse, and C. Fetzer, Eds., 2019, pp. 25:1–25:16.

[11] K. Vora, R. Gupta, and G. Xu, "KickStarter: Fast and accurate computations on streaming graphs via trimmed approximations," in *Proc. 22nd Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2017, pp. 237–251.

[12] K. Vora, R. Gupta, and G. H. Xu, "Synergistic analysis of evolving graphs," *ACM Trans. Architecture Code Optim.*, vol. 13, no. 4, pp. 32:1–32:27, 2016.

[13] B. B. Xuan, A. Ferreira, and A. Jarry, "Computing shortest, fastest, and foremost journeys in dynamic networks," *Int. J. Found. Comput. Sci.*, vol. 14, no. 02, pp. 267–285, 2003.

[14] S. Huang, J. Cheng, and H. Wu, "Temporal graph traversals: Definitions, algorithms, and applications," 2014, *arXiv:1401.1919*.

[15] H. Wu, Y. Huang, J. Cheng, J. Li, and Y. Ke, "Reachability and time-based path queries in temporal graphs," in *Proc. IEEE 32nd Int. Conf. Data Eng.*, 2016, pp. 145–156.

[16] P. Kumar and H. H. Huang, "GraphOne: A data store for real-time analytics on evolving graphs," in *Proc. 17th USENIX Conf. File Storage Technol.*, A. Merchant and H. Weatherspoon, Eds., 2019, pp. 249–263.

[17] M. Afarin, C. Gao, S. Rahman, N. B. Abu-Ghazaleh, and R. Gupta, "CommonGraph: Graph analytics on evolving data," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, T. M. Aamodt, N. D. E. Jerger, and M. M. Swift, Eds., 2023, pp. 133–145.

[18] W. Fan et al., "GraphScope: A unified engine for big graph processing," in *Proc. VLDB Endowment*, vol. 14, no. 12, pp. 2879–2892, 2021.

[19] H. Chen et al., "Achieving sub-second pairwise query over evolving graphs," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2023, pp. 1–15.

[20] S. Gong et al., "Automating incremental graph processing with flexible memoization," in *Proc. VLDB Endowment*, vol. 14, no. 9, pp. 1613–1625, 2021.

[21] H. Wu, J. Cheng, Y. Ke, S. Huang, Y. Huang, and H. Wu, "Efficient algorithms for temporal path computation," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 11, pp. 2927–2942, Nov. 2016.

[22] Y. Zhao, X. Wang, H. Yang, L. Song, and J. Tang, "Large scale evolving graphs with burst detection," in *Proc. 28th Int. Joint Conf. Artif. Intell.*, 2019, pp. 4412–4418.

[23] G. H. Nguyen, J. B. Lee, R. A. Rossi, N. K. Ahmed, E. Koh, and S. Kim, "Continuous-time dynamic network embeddings," in *Proc. Companion Web Conf.*, 2018, pp. 969–976.

[24] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[25] R. Bellman, "On a routing problem," *Quart. Appl. Math.*, vol. 16, no. 1, pp. 87–90, 1958.

[26] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu, "Graphflow: An active graph database," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 1695–1698.

[27] W. Han, K. Li, S. Chen, and W. Chen, "Auxo: A temporal graph management system," *Big Data Mining Analytics*, vol. 2, no. 1, pp. 58–71, 2018.

[28] M. Steinbauer and G. Anderst-Kotsis, "DynamoGraph: Extending the pregel paradigm for large-scale temporal graph processing," *Int. J. Grid Utility Comput.*, vol. 7, no. 2, pp. 141–151, 2016.

[29] B. Erb et al., "Graphtides: A framework for evaluating stream-based graph processing platforms," in *Proc. 1st ACM SIGMOD Joint Int. Workshop Graph Data Manage. Exp. Syst. Netw. Data Analytics*, A. Arora, A. Bhattacharya, G. H. L. Fletcher, J. L. Larriba-Pey, S. Roy, and R. West, Eds., 2018, pp. 3:1–3:10.

[30] A. Andreiana, C. Badica, and E. Ganea, "An experimental comparison of implementations of dijkstra's single source shortest path algorithm using different priority queues data structures," in *Proc. 24th Int. Conf. Syst. Theory Control Comput.*, L. Barbulescu, Ed., 2020, pp. 124–129.

[31] R. Dechter and J. Pearl, "Generalized best-first search strategies and the optimality of A*," *J. ACM*, vol. 32, no. 3, pp. 505–536, 1985.

[32] D. Ferguson and A. Stentz, "Field D*: An interpolation-based path planner and replanner," in *Proc. 12th Int. Symp. Robot. Res.*, S. Thrun, R. A. Brooks, and H. F. Durrant-Whyte, Eds., 2005, pp. 239–253.

[33] A. Martelli, "On the complexity of admissible search algorithms," *Artif. Intell.*, vol. 8, no. 1, pp. 1–13, 1977.

[34] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "LLAMA: Efficient graph analytics using large multiversioned arrays," in *Proc. IEEE 31st Int. Conf. Data Eng.*, 2015, pp. 363–374.

[35] M. Mariappan and K. Vora, "GraphBolt: Dependency-driven synchronous processing of streaming graphs," in *Proc. 14th EuroSys Conf.*, 2019, pp. 1–16.

[36] Y. Chen and Y. Chen, "Decomposing dags into spanning trees: A new way to compress transitive closures," in *Proc. IEEE 27th Int. Conf. Data Eng.*, 2011, pp. 1007–1018.

[37] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 301–316.

[38] K. Vora, "LUMOS: Dependency-driven disk-based graph processing," in *Proc. USENIX Annu. Tech. Conf.*, D. Malkhi and D. Tsafrir, Eds., 2019, pp. 429–442.

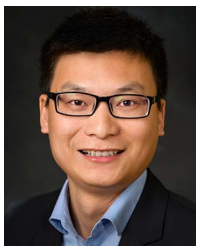
[39] A. Mislove, H. S. Koppula, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Growth of the flickr social network," in *Proc. 1st Workshop Online Social Netw.*, C. Faloutsos, T. Karagiannis, and P. Rodriguez, Eds., 2008, pp. 25–30.

[40] A. E. Mislove and K. P. Gummadi, "Online social networks: Measurement, analysis, and applications to distributed information systems," Rice University, 2009.
[41] J. Kunegis, "KONECT: The koblenz network collection," in *Proc. 22nd Int. World Wide Web Conf.*, L. Carr, A. H. F. Laender, B. F. Lóscio, I. King, M. Fontoura, D. Vrandecic, L. Aroyo, J. P. M. de Oliveira, F. Lima, and E. Wilde, Eds., 2013, pp. 1343–1350.
[42] R. Wetzker, C. Zimmermann, and C. Bauckhage, "Analyzing social bookmarking systems: A del.icio.us cookbook," in *Proc. ECAI Mining Social Data Workshop*, 2008, pp. 26–30.
[43] H. Kwak, C. Lee, H. Park, and S. B. Moon, "What is Twitter, A social network or a news media?," in *Proc. 19th Int. Conf. World Wide Web*, M. Rappa, P. Jones, J. Freire, and S. Chakrabarti, Eds., 2010, pp. 591–600.
[44] P. Boldi, M. Santini, and S. Vigna, "A large time-aware web graph," *SIGIR Forum*, vol. 42, no. 2, pp. 33–38, 2008.
[45] J. Kim and S. Swanson, "Blaze: Fast graph processing on fast SSDs," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2022, pp. 1–15.
[46] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "FlashGraph: Processing billion-node graphs on an array of commodity SSDs," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 45–58.
[47] H. Liu and H. H. Huang, "Graphene: Fine-grained IO management for graph computing," in *Proc. 15th USENIX Conf. File Storage Technol.*, G. Kuenning and C. A. Waldspurger, Eds., 2017, pp. 285–300.
[48] V. Nicosia, J. Tang, M. Musolesi, G. Russo, C. Mascolo, and V. Latora, "Components in time-varying graphs," *Chaos: An Interdiscipl. J. Nonlinear Sci.*, vol. 22, no. 2, 2012, Art. no. 023101.
[49] P. Rozenshtein and A. Gionis, "Temporal pagerank," in *Proc. Eur. Conf. Mach. Learn. Knowl. Discov. Databases*, 2016, pp. 674–689.
[50] S. Huang, A. W.-C. Fu, and R. Liu, "Minimum spanning trees in temporal graphs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 419–430.
[51] R. Pfitzner, I. Scholtes, A. Garas, C. J. Tessone, and F. Schweitzer, "Betweenness preference: Quantifying correlations in the topological dynamics of temporal networks," *Phys. Rev. Lett.*, vol. 110, no. 19, 2013, Art. no. 198701.
[52] S. Huang, Z. Bao, G. Li, Y. Zhou, and J. S. Culpepper, "Temporal network representation learning via historical neighborhoods aggregation," in *Proc. 36th IEEE Int. Conf. Data Eng.*, 2020, pp. 1117–1128.
[53] S. Gandhi and Y. Simmhan, "An interval-centric model for distributed computing over temporal graphs," in *Proc. 36th IEEE Int. Conf. Data Eng.*, 2020, pp. 1129–1140.
[54] J. Zhao et al., "SaGraph: A similarity-aware hardware accelerator for temporal graph processing," in *Proc. 60th ACM/IEEE Des. Automat. Conf.*, 2023, pp. 1–6.
[55] X. Shi, B. Cui, Y. Shao, and Y. Tong, "Tornado: A system for real-time iterative analysis over evolving data," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 417–430.
[56] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 439–455.
[57] A. Kusum, K. Vora, R. Gupta, and I. Neamtiu, "Efficient processing of large graphs via input reduction," in *Proc. 25th ACM Int. Symp. High- Perform. Parallel Distrib. Comput.*, H. Nakashima, K. Taura, and J. Lange, Eds., 2016, pp. 245–257.

Chengying Huan received the PhD degree in computer science and technology from Tsinghua University, Beijing, China, in 2022. Currently, he is a postdoctoral researcher with the Institute of Software, Chinese Academy of Sciences. His research interests include parallel and distributed systems.

Yongchao Liu received the PhD degree in computer engineering from Nanyang Technological University (Singapore), in 2012. He is currently a staff engineer with Ant Group (China), leading the development of distributed graph deep learning systems. He won two Best Paper Awards from IEEE ASAP, in 2009 and 2015, respectively. His research interests include parallel computing, machine learning, bioinformatics, and algorithms.

Heng Zhang received the PhD degree in computer software and theory from the Institute of Software Chinese Academy of Sciences (ISCAS), in 2018. He is currently an associate research professor in computer science and technology with ISCAS. His research interests include high-performance system, operation system, and Big Data technology.



Hang Liu received the PhD degree from George Washington University, in 2017. He is an assistant professor in electrical and computer engineering with Rutgers University. He is on the editorial board for *Journal of BigData*, a program committee member for SC, HPDC, and IPDPS, and a regular reviewer for *IEEE Transactions on Parallel and Distributed Systems* and *IEEE Transactions on Computers*. He is the Champion of the MIT/Amazon GraphChallenge 2018 and 2019, and one of the best papers awardees in VLDB '20.

Shiyang Chen is currently working toward the PhD degree in electrical and computer engineering with Rutgers University. His study focuses on high-performance computation in the area of linear algebra and machine learning. He is particularly interested in using emerging hardware and developing systems for solving large-scale problems.

Shuaiwen Leon Song is an associate professor in NA with the School of Computer Science and the director for Future System Architecture Lab (FSA), University of Sydney. His research interest is with the boundary of system software and hardware, breaking down abstraction barriers, and rethinking the hardwaresoftware interface. He received multiple awards, including the 2020 Australia's most innovative engineer awardand 2017 IEEE TCHPC early career award.

Yanjun Wu received the PhD degree in computer software and theory from the Chinese Academy of Sciences, in 2006. He is currently a professor and the chief engineer of the Institute of Software, Chinese Academy of Sciences. His current research interests include operating systems and system security.