TARIS: Scalable Incremental Processing of Time-Respecting Algorithms on Streaming Graphs

Ruchi Bhoot, Suved Sanjay Ghanmode, and Yogesh Simmhan, *Senior Member, IEEE*

***Abstract*—Temporal graphs change with time and have a lifespan associated with each vertex and edge. These graphs are suitable to process time-respecting algorithms where the traversed edges must have monotonic timestamps. Interval-centric Computing Model (ICM) is a distributed programming abstraction to design such temporal algorithms. There has been little work on supporting time-respecting algorithms at large scales for streaming graphs, which are updated continuously at high rates (Millions/s), such as in financial and social networks. In this article, we extend the windowed-variant of ICM for incremental computing over streaming graph updates. We formalize the properties of temporal graph algorithms and prove that our model of incremental computing over streaming updates is equivalent to batch execution of ICM. We design TARIS, a novel distributed graph platform that implements these incremental computing features. We use efficient data structures to reduce memory access and enhance locality during graph updates. We also propose scheduling strategies to interleave updates with computing, and streaming strategies to adapt the execution window for incremental computing to the variable input rates. Our detailed and rigorous evaluation of temporal algorithms on large-scale graphs with up to 2 B edges show that TARIS out-performs contemporary baselines, Tink and Gradoop, by 3–4 orders of magnitude, and handles a high input rate of $83k$–587 M Mutations/s with latencies in the order of seconds–minutes.**

***Index Terms*—Discrete Mathematics, distributed programming, distributed systems, general, graph algorithms, graph theory, information storage and retrieval, information technology and systems, mathematics of computing, numerical analysis, programming techniques, parallel algorithms, systems and software, software/software engineering.**

I. Introduction

Temporal graphs are ones that exhibit dynamism. Vertices and edges that form the topology can be added and removed over time, as can the labels or properties associated with them [1]. Many real-world dynamic systems can be modeled as a temporal graph, e.g., traffic flows in a road network (properties change, sometimes topology) [2], information diffusion in social networks (both topology and properties change) [3], and contact networks for contagion spreads like COVID-19 (both topology and properties change) [4], [5]. Such graphs can also be large, with millions or billions of vertices and edges, and spanning days or months.

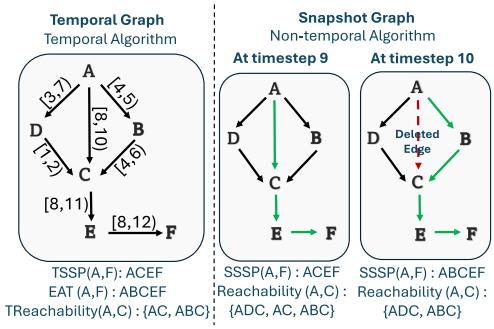


Fig. 1. (Left) Example temporal graph with sample outputs of time respecting algorithms. (Right) Non-temporal SSSP from $A-F$ changes in time 9 and 10, but temporal shortest path stays the same.

Temporal graphs are interval property graphs, where vertices and edges possess key-value pair properties and are associated with time intervals. The validity of vertices, edges, and properties is restricted to these intervals [1], [4], [6], [7], [8]. The *lifespan* of vertices, edges and properties is valid only during these specified intervals. This makes the design of temporal graph algorithms interesting and challenging since they need to consider not just *which* vertices are interacting along the edges but also *when* they interact and the duration of interaction, which can determine how paths are traversed.

We focus on a class of *time-respecting paths* (or temporal paths) has been defined in literature [6], where edges along the topological path need to have monotonically increasing timestamps. E.g., in COVID-19 contact tracing, the disease can spread from person A to person C via path $A \rightarrow B \rightarrow C$ only if B interacts with C *after* A interacts with B [4], [5]. Several such temporal path definitions exist, e.g., Earliest Arrival Time (EAT) [6], Reachability [9], and fastest and shortest paths [6]. These have also been used to find constrained temporal paths [10], enumerate temporal paths between vertices [11], limit the wait time between hops [12], and calculate temporal centrality [13]. So this is a rich space to explore.

For example, in the temporal graph in Fig. 1 where the edge traversals take 1 unit of time, the *Earliest Arrival Time (EAT)* from A to F is along $A(\cdot,4) \rightarrow B(5,5) \rightarrow C(6,8) \rightarrow E(9,9) \rightarrow F(10,\cdot)$, which arrives at F at time 10, while the *Temporal Single Source Shortest (length) Path (TSSSP)* is $A(\cdot,8) \rightarrow C(9,9) \rightarrow E(10,10) \rightarrow F(11,\cdot)$, which takes just 3 hops to reach F; the arrival and departure times for a vertex

Received 9 July 2024; revised 13 September 2024; accepted 15 September 2024. Date of publication 30 September 2024; date of current version 28 October 2024. The work of Ruchi Bhoot was supported in part by a Wells Fargo Women Fellowship at IISc. The work of Yogesh Simmhan was supported in part by a Swarna Jayanti Fellowship from the Department of Science and Technology, Government of India. Recommended for acceptance by F. Zhang. *(Corresponding author: Ruchi Bhoot.)*

The authors are with the Indian Institute of Science, Bangalore 560012, India (e-mail: ruchibhoot@iisc.ac.in; suvedsanjay@iisc.ac.in; simmhan@iisc.ac.in).

This article has supplementary downloadable material available at https://doi.org/10.1109/TPDS.2024.3471574, provided by the authors.

Digital Object Identifier 10.1109/TPDS.2024.3471574

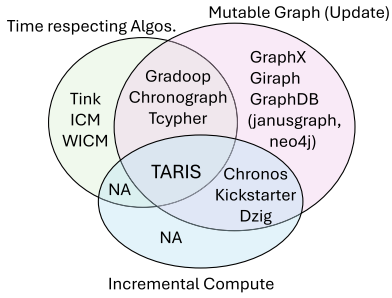


Fig. 2. Categorizing temporal graph analytics frameworks.

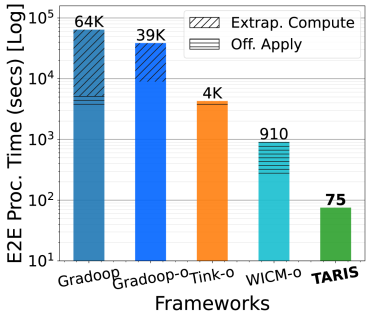


Fig. 3. Time to *apply updates* and *compute EAT* on Reddit graph. Hatched region shows linear extrapolation to fill failed runs.

are in parentheses, e.g., departing A at time 4 to arrive at B at $4 + 1 = 5$, and departing immediately from B at time 5.

Temporal algorithms have different properties than non-temporal algorithms [1], [14]. E.g., *non-temporal reachability* is transitive between vertices, i.e., if $A - D$ is reachable, and $D - C$ is reachable, then $A - C$ is also reachable; but *temporal reachability* is not transitive, as we can see from Fig. 1, where $A(\cdot, 3) \rightarrow D(4, \cdot)$ and $D(\cdot, 1) \rightarrow C(2, \cdot)$ are reachable, but $A - C$ is not reachable (through D) since we arrive at D from A at time 4, after the lifespan [1,2) of the edge $D \rightarrow C$ has expired. *So non-temporal paths cannot be trivially translated to time-respecting paths.* Ensuring that the temporal paths are valid, i.e., connected and time respecting, adds to the challenging nature of graph algorithms with their irregular nature and costly computation. Temporal graph processing platforms (Fig. 2) have been developed to ease the design and scalable execution of time-respecting algorithms, e.g., ChronoGraph [15], Tink [8], ICM [7] and Gradoop [16].

A. Motivation

We focus on temporal graphs that continuously receive a *stream of mutations* to modify its topological structure by updating temporal lifespan while the graph algorithm is executing. These mutations can add or remove a vertex or edge, or change their property from that point in time. The graph algorithm itself becomes a form of "standing query" over the stream of updates and the prior state of the temporal graph. E.g., one may wish to compute the temporal shortest path over a road network between a source and a destination, and keep updating this path as the traffic flow changes while we are transiting this path [17]. Similarly, applications may *continuously* detect the dispersion of contaminants in a city water network [18], fake news within a social network [19], or fraudulent flows within a financial transaction graph [20], in order of seconds-minutes to help take corrective action.

These motivate the need for graphs platforms that can continuously operate on a stream of mutations that update the temporal graph, and include these updates within the ongoing computation of a graph algorithm. The platform should be able to operate on *large graphs* with 1 M–1 B vertices and edges, a *high input rate of mutations*, e.g., $83k$–587 M updates per second for social networks and fintech transaction graphs, and return *updated results with low latency* of seconds–minutes for the online graph algorithms.

B. Gaps

Several existing graph platforms (Fig. 2) like ICM [7] and Tink [8] that support time-respecting algorithms do not support dynamic graph updates. They operate over a materialized temporal graph, where the algorithms execute in batch manner over a historical graph that does not change as the algorithm is executing. Any mutations that arrive need to first be applied offline to the temporal graph before the algorithm runs. In contrast, platforms like Gradoop [16] and Chronograph [15] natively support updates to the in-memory graph data structure as the streaming updates arrive. These address the requirement of continuously executing time-respecting algorithms over mutation streams.

However, in both these classes, the platforms need to *recompute the algorithm* on the entire cumulative temporal graph when the graph is updated with new mutations, i.e., when updates for a time duration δ arrive, the algorithm re-runs over all timesteps of the graph from 0 to $t + \delta$ even when it has previously been executed until t. This is necessary since these platforms do not reuse the state of the graph at the previous timestep to derive the state at the next timestep as we show is possible for these time-respecting algorithms, i.e., they do not perform *incremental computation*.

This recomputation is computationally expensive. E.g., the Reddit social network graph has 9.3 M vertices, 528 M edges and 122 timesteps of mutations ($\approx$ 100 to 50 M updates per timestep). We perform *offline update ("-o")* of the temporal graph for the mutations received at each timestep, and then run the EAT algorithm using Tink [8] on each of these updated temporal graphs. EAT computation cumulatively takes 3635 s due to recomputation after every timestep (Fig. 3, *Tink-o*), with the last recomputation at timestep 122 taking 68 s. This grows monotonically with more updates, and also excludes the $\approx$ 630 s spent in pre-processing of the graph updates offline. Gradoop [16], which supports in-place mutations of the graph, does a full recomputation after the updates are applied and takes over $64,000$ s to apply the updates and do EAT (Fig. 3, *Gradoop*) – interestingly, doing the offline update and reloading it into Gradoop for recomputation is faster, taking about $37,000$ s for EAT compute (*Gradoop-o*). Both Gradoops failed for some timesteps despite retries and causing us to interpolate its execution times. So performing efficient online updates to large graphs is non-trivial. Our prior work, ICM [7], takes a total of 255 s for recomputation of each temporal graph using its window-based variant *WICM* [21]. In contrast, the *TARIS* system we propose here takes a total time

of 75 s for *incrementally applying the update and computing EAT* on all 122 timesteps, and takes just 1.2–33.1 s per window with an incremental window size of 20 timesteps, as we discuss later.

The incremental computation approach of our proposed TARIS system is inspired by prior works on streaming graph for *non-time respecting algorithms* [22], [23], [24], [25]. These platforms (Fig. 2, Incremental Compute) perform in-place mutation of the updates and execute the graph algorithm to return the updated state incrementally. But they *do not* natively support time-respecting algorithms. E.g., in Fig. 1, when the edge $A \to C$ is deleted at time 10, the non-temporal shortest path from $A - F$ changes from $A \to C \to E \to F$ in snapshot 9 to $A \to B \to C \to E \to F$ at snapshot 10. But the temporal shortest path remains $A \to C \to E \to F$ since $A \to C$ was deleted after we arrived at C at time 6. So non-time respecting platforms cannot capture temporal algorithms without fundamentally changing the algorithm or their system.

TABLE I
CAPABILITIES OF TEMPORAL GRAPH PLATFORMS

Framework	Mutable	Streaming	Incremental	Time-Resp.	Distrib.
Giraph [26]	✓				✓
GraphX [28]	✓				✓
Neo4j [29]	✓			✓	
JanusGraph [30]	✓			✓	✓
Incremental Computation					
KickStarter [22]	✓	✓	✓		
RisGraph [23]	✓	✓	✓		
Dzig [25]	✓	✓	✓		
Chronos [24]	✓	✓	✓		✓
Time-respecting Algorithms					
Chronograph [15]	✓			✓	
Tink [8]				✓	✓
ICM [7]				✓	✓
WICM [21]				✓	✓
Gradoop [16]	✓			✓	✓
Tegraph [31]	✓	✓		✓	
Raphtory [32]	✓	✓		✓	✓
Tcypher [33]	✓			✓	✓
TARIS	✓	✓	✓	✓	✓

C. Contributions

In this article, we propose *TARIS*, a distributed platform to design and execute *Temporal AlgoRithms with Incremental processing for Streaming graphs*. TARIS builds upon our prior works, Interval-centric Computing Model (ICM) [7] and its window-based distributed execution (WICM) [21], which allow intuitive designing and scalable *batch* execution of time-respecting algorithms on a *materialized* temporal graph. TARIS substantially expands this in two dimensions to execute those same time-respecting algorithms (1) *incrementally*, and (2) over *streaming temporal graphs*.

It develops novel data structures and strategies to efficiently apply streaming updates to the distributed in-memory graph, maintained using Apache Giraph [26], in a windowed manner that micro-batches several timesteps of updates. It also reuses the state of the traversal completed until the previous timestep to incrementally execute the algorithm over the updated graph, resuming from the prior paths. We formally prove the correctness of the platform, guaranteeing that the result of the incremental execution is equivalent to recomputing from scratch. Further, we adjust the window size dynamically at runtime to support high-throughput with modest latencies. Our detailed experiments show that TARIS adapts to different input stream rate (up to 106 timesteps per sec.), with each timestep having 100–50 M mutations, to execute different time-respecting algorithms incrementally for mutation rates as high as 587 M/s. Our comparative experiments against Gradoop [16] and Tink [8], two state-of-the-art (SOTA) time-respecting graph processing platforms, show TARIS to outperforms them by several orders of magnitude for different graphs and algorithms. We are also $\approx 3\times$ faster than the cumulative execution time of our own WICM [21].

To the best of our knowledge, TARIS is the first platform to support incremental processing of time respecting algorithms on streaming graphs.

We make the following specific contributions:

1) We formalize the properties of temporal graph algorithms and prove the correctness of incremental computation with minimal additional computing for streaming updates (Section III).
2) We design a novel framework, TARIS, extending from WICM that incorporates these incremental computing features using memory-efficient data structures (Section IV). It uses intelligent strategies to interleave graph updates with compute (Section V) and dynamic windowing strategies to adapt to the stream rate (Section VI).
3) We offer detailed evaluations of several temporal algorithms for large realistic graphs and streaming updates (Section VII). TARIS handles input rates of 140 M Mutations/s with 10 s latency, achieves a throughput of 85 M edges/s for Reddit, and easily out-performs Tink and Gradoop.

A preliminary intuition of the scheduling strategies proposed in this article appeared as a short two-page poster [27], where we described three scheduling strategies (AITM, DITM and JITM) to apply updates and validated it for one graph/algorithm (Reddit/EAT). This forms a small part of Section V. We have now substantially formalized the problem and solution, offered proofs of correctness, optimized the processing of graph updates, added new and improved scheduling strategies, and streaming strategies to adapt to dynamic input rates, run extensive experiments on 4 graphs and 5 algorithms, and included comparisons with two other SOTA platforms.

II. RELATED WORK

Scalable graph processing frameworks and databases have been well-studied in literature to execute algorithms, analytics [34] and queries over large graphs. In particular, there has been sustained interest in distributed graph processing using *Google's Pregel* Vertex-centric Computing Model (VCM) [35], and its open-source implementations, *Apache Giraph* [26] and *Spark's GraphX* [28]. These maintain the graph in distributed memory in a cluster of machines and execute VCM algorithms over billion-scale graphs. However, they do not natively support streaming updates to mutate the graph, time-respecting algorithms, or incremental processing of the algorithm (Table I). Next, we review literature on temporal graphs, streaming graph updates and incremental processing, which is our focus.

A. Platforms for Time-Respecting Algorithms

There are two broad approaches to calculate time-respecting paths in temporal graphs: (1) Transform the graph into snapshots, one per timestep, with a snapshot capturing the state of the entire graph at that point in time [7], [8], [21]. We can then run a static graph algorithm on a snapshot, and carry over its state to the next snapshot. This does not scale well due to the explosion in the graph sizes as timesteps increase, since entities are duplicated between adjacent snapshots, causing possible duplication in the compute as well as the next snapshot is processed. (2) The other approach adds temporal properties (e.g., time intervals) to a single graph and/or time constraints to the graph algorithms to enforce time-respecting behavior [8], [15], [16], [33].

Chronograph [15] supports temporal traversals on a property graph using both *point-based*, i.e., when edges are present at only one point in time, and *interval-based* timestamp attributes on vertices and edges. It also offers various temporal traversal patterns based on the Gremlin query language, which can be used to implement time-respecting BFS and shortest path algorithms. While it supports offline mutations to the graph, it does not support streaming updates or incremental computation. Also, it executes on just one machine but with multi-core CPU optimizations. The largest graph they evaluate has just 1.9 M vertices and 39.9 M edges.

Tink [8] is a library built over Apache Flink and its Gelly API to support time-respecting algorithms on temporal graphs with distributed execution. It stores interval properties on the vertices and edges, and uses Signal/Collect model to implement temporal algorithms like EAT, fastest path and centrality measures. It does not support online mutations to the graph or incremental processing. *Gradoop* [16] is designed for scalable analytics over distributed temporal property graphs that store both point-based and interval-based timestamps. Its GRALA language allows time-respecting algorithms to be designed using temporal predicates like *fromTo(t1,t2)*, *precedes(t1,t2)* and *overlaps(t1,t2)*. It internally translates these graph operators to a sequence of Flink operators, which performs distributed execution using Gelly's Pregel-like VCM operators. It is shown to scale to temporal graphs with 283 M vertices and 1.8 B edges. While it allows in-place mutations on the in-memory graph, it does not perform incremental computation over the updated graph. We adapt and use Tink and Gradoop as two SOTA platforms to compare TARIS against.

Tegraph [31] is a single machine platform that efficiently decomposes a temporal graph and its updates into a series of static graph snapshots. It then executes the time-respecting algorithm on a series of these snapshots while using optimizations to remove redundant entities to reduce memory and compute time. *Raphtory* [32] and *Tcypher* [33] both support temporal algorithms and are optimized for updating the in-memory graph based on streaming data. They store both the current state and the history of the graph. However, Tcypher does not store results across queries and recomputes them for every new query, while Raphtory overlaps compute and update, and may return results on a stale version of the graph.

Our prior work introduces a novel *Interval-centric Computing Model (ICM)* [7] for intuitively specifying time-respecting graph algorithms over an interval-graph model by extending the popular Pregel VCM paradigm. It introduces a TimeWarp operator to filter and send messages between vertices only along edges with time-respecting lifespan. This reduces the message transfers, memory overheads and avoids redundant computation. ICM extends Apache Giraph to store the interval graph in distributed memory and execute these primitives over billion-scale graphs. *Windowed ICM (WICM)* [21] further optimizes ICM by splitting the execution of an interval graph into a series of executions over non-overlapping time windows to reduce memory pressure and avoid duplicate messaging and compute across windows. But neither support in-place streaming updates over the graph, or incremental computation over the updated graph. TARIS supports the ICM abstraction and leverages the WICM implementation.

B. Platforms for Incremental Processing

There are several platforms that support incremental computation over streaming graph updates, but without support for time-respecting graph algorithms [36]. These reuse the results on the earlier graph to recompute only the parts which were updated. This involves two broad steps: (1) Apply the update mutations on the prior graph to get the new updated graph, and (2) Compute the incremental result on the updated graph.

Chronos [24] is a distributed, snapshot based, in-memory, iterative graph computation framework for temporal graph analysis. They propose methods to efficiently store the evolving graph snapshots in memory.. It supports incremental processing for non temporal algorithms like SSSP, BFS and DFS, but not for time-respecting algorithms. *Kickstarter* [22] is a single machine framework for incremental graph processing. When a graph changes, it identifies the prior states (e.g., *rank* for PageRank) which become invalid and limits its recomputation to only the new state values for these entities. *GraphBolt* [37] extends this to reduces the number of edges and vertices that needs to be recomputed by using a dependency graph, while *DZig* [25] further optimizes this by passing states to downstream nodes only if there is a change in the state.

Risgraph [23] provides a single machine parallel framework for propagating the updated results of monotonic graph algorithms, like shortest path, for each granular update. They use a hash and list based adjacency list for faster graph updates, and enhance data locality to reduce the update and incremental compute time. *Livegraph* [38] too proposes a novel Transactional Edge Log data structure for faster scans over adjacency lists and edge updates in the graph database, and shows improvements over Neo4j [29].

These frameworks focus on optimization for incremental computation for *non-time respecting* graph algorithms, and need fundamental changes to support such algorithms. TARIS designs and adopts comparable optimizations for time-respecting incremental processing, e.g., reordering update types for efficient mutation updates [39], separating data structures used to accumulate/apply mutations and to execute the computation [40], and sharing vertex states across timestamps to avoid recomputation [24]. Platforms can also be delineated based on whether they support *application time* or *system time* [41]. In the former, the time property arrives with the graph entity (vertex or edge) update and may come out of order to the system [8], [16], [31],

while in the latter, the time at which the entity arrives at the system is considered as its time [32]. TARIS uses application time for processing but realistically assumes that it is close to the system time, i.e., no major delays between update creation and arrival, and that the updates arrives in temporal order.

III. Preliminaries

A. Temporal Graph as an Interval Graph Model

We formally model a temporal graph as an *interval graph* [7], defined as $G\langle V, E\rangle$, where $v \in V$ is *set of interval vertices*, each denoted as $\langle v_{id}, L\rangle$. v_{id} is a unique identifier for the vertex and L is its lifespan, given as $[t_s, t_e)$, where t_s and t_e represent the start (inclusive) and end (exclusive) timesteps, respectively. *Timesteps* are a linearly ordered discrete time points, where $t_i < t_{i+1} \Rightarrow t_i$ occurred before t_{i+1}. Each timestep is an atomic increment of time, and corresponds to some user-defined wall-clock time, such as d secs. Similarly, $e \in E$ is a *set of directed interval edges*, where each edge is given as $\langle e_{id}, u_{id}, v_{id}, L\rangle$, where is e_{id} is the unique identifier for a directed edge from vertex ID u_{id} to v_{id} and the edge has a lifespan L. While not shown, the vertices and edges can also have *interval properties*, e.g., vertex labels, edge distances, etc.

Each vertex and edge conforms to *uniqueness constraints* in space and time, where they have a single continuous lifespan $[t_s, t_e)$, and once a vertex or edge is deleted, no vertex or edge with the same ID is ever added again. All edges e_{id} also follow *temporal referential integrity constraints*, i.e., the lifespan of edge L_e is a subset of the lifespans of both the source L_u and the sink L_v vertices. A vertex or an edge with lifespan $[t_s, t_e)$ is said to *exist* at a timestep t if $t_s \leq t < t_e$; t_e can be ∞. Conversely, a vertex or an edge is *non-existent* at time t if $t < t_s$ or $t \geq t_e$.

B. Streaming Updates to Temporal Graph

Let $G(t_j)$ denote the interval graph for the lifespan $0 \leq t \leq t_j$. At each timestep $t_i < t_j$, the *update set (or mutations)* to the temporal graph since the previous timestep t_{i-1} is given by $\Delta(t_i) = \langle \widehat{V}_{add}, \widehat{V}_{del}, \widehat{E}_{add}, \widehat{E}_{del}\rangle$. These sets contain the v_{id}s of vertex additions ($\widehat{V}_{add}$) and deletions ($\widehat{V}_{del}$), and $e_{id}s$ of edge additions ($\widehat{E}_{add}$) and deletions ($\widehat{E}_{del}$) applied and visible at t_i. Due to the uniqueness constraint, there can never be more than one addition or deletion for a v_{id} or e_{id}.

Applying ($\mathcal{A}$) the update set $\Delta(t_i)$ at timestep t_i to the temporal graph $G(t_{i-1})$ at the previous timestep will mutate and return the temporal graph in the new timestep, $G(t_i) = \mathcal{A}(G(t_{i-1}), \Delta(t_i))$. Vertex (and edge) additions will create a new vertex ID (edge ID) in graph $G(t_i)$ with a lifespan interval $[t_i, \infty)$, i.e., created at current timestep and ending in timestep ∞, some unknown time in the future. Deletions will cause the end time of corresponding vertices and edges in $G(t_i)$ to be truncated at the current timestep, i.e., change the interval from $[t_k, \infty)$ to $[t_k, t_i)$. Note that the update set that arrive are scoped to the current timestep, and the updates received at a particular timestep cannot be applied back in time (e.g., delayed events) or forward in time (e.g., early events).

There is some application-specific mapping from each unit timestep to a wallclock duration (e.g., w secs.). So $\Delta(t_i)$ indicates all updates that happened within that wallclock duration but are considered as instantaneously occurring and visible at the same timestep. This allows us to control the temporal coarseness and precision for the application, and also the performance and throughput of the platform. The rate at which the update sets are received is defined as ρ update-sets per second, which can translate to μ mutations per second, depending on the number of mutations per set, which itself can be constant or variable. In our experiments, we use $\rho = 0.08 - 106$/sec, i.e., one update set every 9 ms–12.5 s, and $\mu = 83$ K–587 M/s.

C. Time-Respecting Algorithms

We define a *time-respecting path* in the interval graph as one where the timesteps on the edges of the path increase (or decrease) *monotonically*. Say a time-respecting (or temporal) path starts at a source vertex v_0 at a timestep t_{src} when it exists. WLOG, let each edge have a static travel time of d duration. For each vertex and edge along the temporal path, $[v_{src}, e_{(src,1)}, v_1, e_{(1,2)}, v_2, \ldots]$, we have the requirement that a traversed edge $e_{(i,i+1)}$ should exist at some timepoint after the arrival time ($AT(v_x)$) at the preceding vertex v_x. The arrival time ($AT(v_{x+1})$) at the succeeding vertex v_{x+1} is, $AT(v_{x+1}) = (\max(AT(v_x), ST(e_{i,x+1})) + d)$, where ST indicates the start time of the edge. This is a monotonically increasing path, and can be modified to a decreasing path for an algorithm (e.g., latest departure time) by using a $\min$ function. Similarly, the value of d can be the edge's interval-property or can be a constant. $P(v_{src}, v_x, t_{ps}, t_{pe})$ denotes a *temporal path* from source vertex v_{src} to sink vertex v_x, which starts no earlier than t_{ps} and ends no later than t_{pe}, i.e., the departure time at v_{src} is $\geq t_{ps}$ and arrival time at v_x is $\leq t_{pe}$. In our notations, we omit v_{src} and t_{ps} when denoting paths subsequently, as they implicitly remain constant for all paths throughout the algorithm.

We next formalize the class of *time-respecting temporal algorithms* that we support. These time-respecting algorithms iteratively expand and identify a time-respecting path from a (user-defined) source vertex v_{src} to every other reachable sink vertices $v_x \in V$ in the graph just by using a *selection function (σ)* [21], [22], which uses the state of the vertices and their edge property to determine the state of the next vertex. Further, the algorithm must ensure that the start and/or end times for the path (t_{ps}, t_{pe}) fall within some user-defined threshold. In other words, the selection function of these algorithms picks one path from the set of all possible paths ($\mathbb{P}$) for each sink vertex v_x from a given source vertex v_{src} that meets some algorithm-specific constraints (e.g., time, state, property, etc.) and returns them as the *result set* $R(t) = \{\sigma(\mathbb{P}(v_x, t)) \rightarrow P(v_x, t)\, \forall v_x \in V\}$ on the graph $G(t)$.

A large class of traversal-based temporal algorithm can be modelled using these primitives in ICM [7], e.g., Temporal Minimum Spanning Tree (TMST), Earliest Arrival Time (EAT), Latest Departure Time (LDT), Fastest travel time, Temporal Reachability (TR), etc., which we evaluate later.

Let the set of time-respecting paths selected by one of these temporal algorithms be the *result set* $R(t_i)$ over the graph $G(t_i)$ at time t_i. So $R(t_i)$ contains an output path from the source vertex to each vertex within timestep t_i, if one exists, and some associated state. E.g., for Temporal Reachability, paths in $R(t_i)$ can be used to derive the tuples $\langle v_x, Yes|No\rangle$,

indicating if the sink vertex v_x can be reached within timestep t_i if we start from source v_{src} at timestep t_{ps}. For TMST, the union of paths in $R(t_i)$ gives the minimum spanning tree of $G(t_i)$ rooted at v_{src}. For EAT, the paths in $R(t_i)$ can be used to obtain the earliest arrival time t_{ea} for each sink vertex v_x from a user-given source vertex, $\{\langle v_x, t_{ea}\rangle \mid t_{ea} \leq t_i, \forall v_x \in V\}$. E.g., in Fig. 1, the result set for EAT from source vertex A is $\{\langle A,0\rangle, \langle B,5\rangle, \langle C,6\rangle, \langle D,4\rangle, \langle E,9\rangle, \langle F,10\rangle\}$ derived from the corresponding paths in R, $\{A, AB, ABC, AD, ABCE, ABCEF\}$. Given as input the initial temporal graph at t_0, $G(t_0)$, the source vertex for the algorithm, v_{src}, the initial starting time when the algorithm is initiated, t_{ps}, and a stream of updates, $\Delta(t_j)$, at each timestep $t_j \mid j > 0$, the goal of executing the temporal graph algorithm is to produce the corresponding output result sets, $R(t_j)$.

D. Properties of Temporal Algorithms With Graph Updates

There are several properties for temporal algorithms that hold for updates made to temporal graphs. We prove these here and reuse them in the design of our increment streaming graph processing to ensure its efficiency and correctness.

1) Zero Recomputation: When mutations to the graph are applied, we need to find the correct results for the new updated graph. One way is to recalculate the results from scratch ($t = 0...n$) on the new graph. Another is to use the results from the previous timestep and perform incremental compute only on the new updates. But this is challenging since the past results may no longer be valid at the current timestep after applying the update set. Hence, the vertex states which become invalid need to be recomputed. However, the characteristic properties of time-respecting paths that we prove below helps omit this recalculation altogether.

Lemma 1: The set of vertices and edges which existed in $G(t_i)$ at any timestep $t, \forall t \leq t_i$, does not change after applying updates at a future timestep, $\Delta(t_{i+1})$.

The deletes in $\Delta(t_{i+1})$ will cause the end time of the affected vertices and edges to be set to t_{i+1} from the default ∞ in $G(t_{i+1})$. Similarly, additions create new vertices or edges, but with a starting time of t_{i+1} and no earlier. So, the graph topology $G(t_{i+1})$ when considering only vertices and edges that existed until the previous timestep t_i is not affected. □

Lemma 2: For all time-respecting paths starting from v_{src} at time t_{ps} to a sink vertex v_x until time $t_i \mid 0 \leq t_{ps} < t_i$, the temporal path $P(v_x, t_i)$ does not change on applying $\Delta(t_{i+1})$.

From Lemma 1, the set of all edges which exist at any timestep $t \leq t_i$ remains the same in $G(t_i)$ and $G(t_{i+1})$. Hence, the prior discovered set of all temporal paths till time t_i also remains the same even after applying the updates at t_{i+1}. □

Lemma 3: If $R(t_i)$ is the result of executing a temporal algorithm on $G(t_i)$, then $R(t_i)$ remains the correct result on $G(t_{i+1})$ as well.

$R(t_i)$ is a function of paths that end within time t_i. From Lemma 2, the set $\mathbb{P}(v_x, t_i)$ of all time-respecting paths from v_{src} to any v_x till timestep t_i remains the same in $G(t_i)$ and in $G(t_{i+1})$. So, the selected path from this path set into the result set by the selection function σ will also remain the same. Hence, $R(t_i)$ will also be the same for $G(t_i)$ and $G(t_{i+1})$. □

Theorem 1: If $R(t_i)$ is the set of result paths computed on $G(t_i)$, then finding the result set $R(t_{i+1})$ on $G(t_{i+1})$ does not require recomputation of prior paths.

From Lemma 3, the output set $R(t_i)$ remains correct in $G(t_{i+1})$ and hence needs no re-computation. It can be directly used to extend the paths in $R(t_i)$ till time t_{i+1} to calculate $R(t_{i+1})$. Fresh computation has to be performed only for the expansion from the previous to the new timestep, i.e., to get $R(t_{i+1})$ from $R(t_i)$, using new information in $G(t_{i+1})$. This makes *incremental computation* highly beneficial. In fact, we can even omit the contents of the temporal graph until the previous timestep since $R(t_i)$ serves as a proxy for it. □

Example. In TR, we answer the question: *Which vertices can be reached within timestep t_{i+1} from a given source vertex?* If we have computed the temporally reachable output path set $R(t_i)$ at timestep t_i, when the graph is updated at time t_{i+1}, we only need to consider edges which are valid for travel at time t_i to t_{i+1} from prior paths in $R(t_i)$. This is because if a vertex was reachable at t_i from the source it will remain reachable at t_{i+1} Any new edges that were added at t_{i+1} can expand from these prior valid paths (or directly from the source) to reach new vertices and update the reachability set $R(t_{i+1})$. This is also a key distinction between time-respecting and non-time respecting paths, as shown in Fig. 1. Non-time respecting paths in the old graph snapshot can become invalid in the new graph snapshot as each snapshot is independently processed, and hence forces recomputation [22], [25].

Corollary 1: The result set $R(t_i)$ remains correct for all future evolution of the temporal graph, $G(t_{i+x})\forall x > 0$.

Lemmas 1, 2 and 3 can be generalized from $G(t_i)$ to $G(t_{i+x})\,\forall x > 0$ through the commutative property over the timesteps. The temporal paths calculated and their vertex states remain valid until that point in time t_i even after the graph evolves in future timesteps. □

2) Incremental Computation: In incremental computing, the idea is to use the prior results until time t_i to calculate the new result at time t_{i+1} with minimal additional computations required only for the newer mutations, i.e., given $R(t_i)$ we need to find $R(t_{i+1})$ on $G(t_{i+1})$. Given the temporal path, $P(v_{src}, v_x, t_{ps}, t_i)$, from v_{src} at timestep t_{ps} to sink vertex v_x within timestep t_i, we can compute the temporal path for the next timestep t_{i+1} as: $P(v_{src}, v_x, t_{ps}, t_{i+1}) = \{P(v_{src}, v_y, t_{ps}, t_j) + P(v_y, v_x, t_j, t_{i+1}) \mid t_j < t_i$ and $v_y \in V \setminus \{v_{src}, v_x\}\}$. Since we assume that the state of vertices change monotonically in the algorithm, by using Theorem 1, the most recent value of $R(t_i)$ is adequate to compute $R(t_{i+1})$. This is proved in prior literature [14].

We use these important properties that we have established for time-respecting temporal algorithms to design the TARIS platform, extending from WICM, for efficient incremental computation while guaranteeing the correctness of the result. We next discuss this architecture and further optimizations.

IV. TARIS Architecture

A. Background

1) Vertex-Centric Computing Model (VCM): Google's Pregel [35] introduced *VCM* to design and execute iterative

Algorithm 1: Template for Temporal Path Traversal Using ICM.

1: **function** ICMCOMPUTEVERTEX (v, Interval t, Messages[] $msgs$)
2: $vState \leftarrow s_v(t)$
3: **for** each Message m in $msgs$ **do**
4: $vState \leftarrow vState \oplus m \triangleright \oplus$ *is User Compute function*
5: ***end for***
6: ***if*** $(vState \neq s_v(t))$ ***then***
7: $s_v(t) \leftarrow vState$
8: ***end if***
9: ***end Function***
10: ***function*** ICMSCATTERVERTEX (u, Interval t, Edge $v \rightarrow u$)
11: $t_r \leftarrow [t + d(v \rightarrow u, t), \infty) \triangleright$ *d is edge travel time*
12: $\mu \leftarrow \sigma(s_v(t), \pi(v \rightarrow u, t)) \triangleright \sigma$ *is User Scatter function*
13: SENDMESSAGE($\mu_n(t) \rightarrow u(t_r)$)
14: ***end Function***

graph algorithms over graphs partitioned across distributed workers (machines). VCM operates in a series of *supersteps* with a barrier synchronization across workers after each superstep. In each superstep, a user-defined `compute` method is executed on all "active" vertices, where the state of the vertex and its out edges can be accessed and updated based on the user-logic, and *messages* can be communicated (scattered) to neighboring vertices. Vertices receiving messages in the next barrier-synchronized superstep become active while, typically, other vertices become inactive. The supersteps repeat till the algorithm converges and no vertices are active. Apache Giraph [26] is a popular open source Java implementation.

2) Interval-Centric Computing Model (ICM): Our prior work on *ICM* [7] extends VCM to design time-respecting graph algorithms over an interval graph for distributed iterative execution. Users define a `compute` method to access and update the local *interval state* of a vertex and its out edges, and a `scatter` method to decide *interval messages* to send to neighbors. The ICM runtime enforces the temporal constraints of time-respecting algorithms using the *TimeWarp* operator to ease the development: any incoming interval message is only sent to the sub-interval of a vertex that overlaps with the message's lifespan; the vertex state update is also limited to this interval potentially partitioning an existing state interval; and `scatter` sends messages only to edges whose lifespan overlaps with the generated interval message. ICM is implemented by extending Apache Giraph.

A generic temporal traversal algorithm using ICM is shown in Algorithm 1. The user compute ($\oplus$) is called by ICM COMPUTE on each partitioned interval state of a vertex ($s_v(t)$) and its incoming messages ($msgs$), which have been aligned by TimeWarp, and update the vertex state. ICM SCATTER sends messages to temporally overlapping neighbors along the out-edges by calling user scatter (σ), which uses the updated interval states and interval edge properties ($\pi(t)$). E.g., in EAT, the $\oplus$ is the $\min()$ function to calculate earliest time and σ is $(t + d(v \rightarrow u, t))$ i.e., sum of the time when we leave vertex v and the travel time d along the edge $v \rightarrow u$ at that time. Similarly, for Reachability, $\oplus$ is the $\max()$ function and σ is the reachability state (*true*) of vertex S_v.

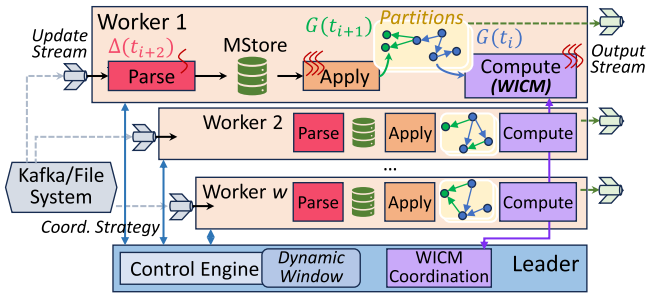


Fig. 4. TARIS architecture.

Windowed ICM (WICM) [21] optimizes the performance of ICM by reducing the time complexity of TimeWarp and temporally partitions the interval graph into disjoint time windows, executing ICM within each time window and then progressing to the next window. Messages spanning windows are accumulated and combined, and made available at the next window's start. This reduces the number of messages generated, conserving memory and network use.

The current work, TARIS, builds upon WICM. WICM meets the requirements stated for time-respecting algorithms in Section III-C. The user's `compute` method ($\oplus$) is a selection function, which is commutative, associative and distributive, and selects from one of the incoming messages that indicate the prior traversed path. The `scatter` method (σ) is monotonic over vertex states. It can be used to design numerous path-based algorithms like EAT, TMST, Reachability, etc., but not non-traversal algorithms like triangle counting and Page Rank. WICM also builds upon Apache Giraph.

B. TARIS System Overview

The architecture of TARIS is shown in Fig. 4, which builds upon WICM and Apache Giraph distributed graph processing framework. It has one *leader* machine for coordination and w *worker* machines holding the temporal graph partitions. A partition is the unit of single-threaded work in a worker, and contains the temporal vertices, their out edges and their properties. The initial temporal graph (if non-empty) is hash-partitioned by Giraph using the vertex IDs into $w \times p$ partitions. Each worker is assigned p partitions, with one compute thread assigned to it; usually, p is the # of CPU cores on the worker. The workers receive, parse and apply the mutations, and perform WICM compute on them, while the *control engine* running on the leader coordinates mutation updates and WICM compute supersteps across workers uses Giraph's aggregators.

In each superstep, workers process mutation update sets for future timesteps that stream to each worker (Section IV-C), and incrementally perform WICM compute on the timesteps that are applied to the interval graph (Section IV-D). The mutation streams are also hash-partitioned and the relevant updates are routed to the workers using a pub-sub system like Apache Kafka or as a file stream. Once a set of timestep updates are applied, the WICM engine can execute the user-logic on the next

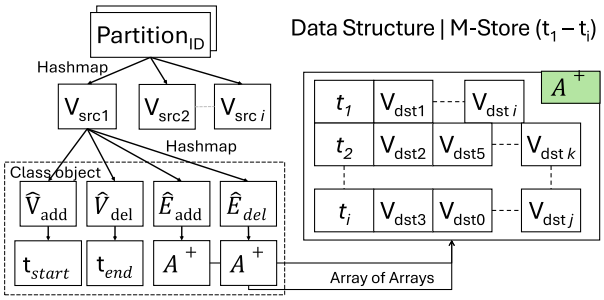


Fig. 5. MStore to hold graph mutations at each worker.

window of the updated graph $G(t_i)$ while streaming updates are concurrently applied for future timesteps $G(t_{i+\epsilon})$. The state of the vertex at time t_i after WICM execution is the algorithm's output, accessible to the user in-memory for further processing, exposed through an output Kafka topic or, by default, written to HDFS.

Parsing and applying these updates to the graph is memory and compute intensive, given the millions of mutations per second and millions–billions of graph vertices and edges in the graph. We propose multiple *scheduling strategies* for efficient pipelined execution for a *mini-batch* of timesteps (Section V), and strategies to dynamically select the WICM *compute window size* to reduce latency and increase throughput (Section VI).

C. Processing Streaming Updates

Each worker (Worker 1 in Fig. 4) performs three phases: *parsing* the received updates to extract mutations and efficiently storing them in our Mutation Store (MStore), *applying* the updates to the temporal graph, and performing *WICM compute* on these updated timesteps. We describe these below.

1) MStore Graph Mutation Store: Mutating the temporal graph in Giraph is memory and compute intensive, and these overheads preclude piecemeal processing of each update arriving at a worker. Our strategies process mutations in mini-batches to limit these overheads and help scale. We develop the MStore data structure to cache the parsed mutations at each worker before it is applied to the graph partitions. It is carefully designed to reduce Java object allocation, limit memory accesses and enhance locality, exploit data parallelism and reduce the time complexity to apply updates.

The MStore data structure (Fig. 5) groups the mutations in a hierarchy, first by the partition, then vertex object, then mutation type and then timestep to achieve these optimizations. It stores the updates for one or more timesteps, $\Delta(t_i)..\Delta(t_{i+\epsilon})$, separately for each partition in the worker. This allows the partitions to be updated concurrently by independent compute threads. For a partition, it maintains a *HashMap* from a vertex ID being updated to all the mutations for that vertex in the time range $t_i..t_{i+\epsilon}$. This allows quick access to the vertex updates when newer update sets are inserted. Each vertex update class maintains the four types of mutations for it. This allows the apply phase to later perform all updates to a vertex in one go, reducing the object access and adjacency list de/serialization and scan costs for the Giraph vertex object. For each edge update type, we maintain an array of timesteps with an array of edge updates in each timestep for that vertex – using arrays reduces indirection, enhances cache locality and Java garbage collection overheads. We avoid any duplication of timesteps, etc. to minimize the memory footprint.

2) Parse Phase: The parser receives the binary-encoded streaming update sets for all partitions on that worker from a file or Kafka topic, with one set per timestep. It routes the mutations in the set to the relevant partition MStore, and parses and inserts them into it. Update sets are assumed to arrive in temporal order of application time. The parser is executed by one of the partition compute threads prior to initiating its compute. Having it be single-threaded reduces cache misses and limits contention with other compute threads concurrently executing the graph algorithm. The mutations for each vertex are parsed and inserted in a single shot, ensuring that the MStore vertex object access is limited to $\mathcal{O}(V)$ for V vertices in an update set, rather than $\mathcal{O}(M)$ for M mutations; $M \gg V$.

3) Apply Phase: The apply phase updates the interval graph using mutations in the MStore. Apply is done in parallel by each partition thread from the MStore mutations for its partition, during the pre-compute phase of a WICM compute. All mutations for a vertex, spanning all available timesteps, are applied as a single mini-batch to minimize accesses to the Giraph vertex object, which involves a costly deserialization process to extract the vertex's adjacency list. Apply uses *double-buffering*, creating a new empty MStore for the parser to write new stream updates to while it exclusively reads from the old MStore object and applies prior updates. To maintain *correctness* and *referential integrity* across timesteps, the update order is: *add vertex*, *add edge*, *delete edge* and *delete vertex*. We scan the edge list for a vertex only once, since it may have 1000s of neighbors, and perform all edge operations in this single scan. For a temporal vertex with adjacency list size d, MStore with d' edge addition and d'' deletion for a timestep, and ts timesteps present in the MStore, the apply time complexity is $\mathcal{O}(\frac{d+d'}{ts \cdot d''})$.

D. Incremental Computing With WICM

TARIS uses WICM to execute the ICM-based graph algorithm over windows of the interval graph, for timesteps whose mutations have been applied to the graph. Compute uses the full graph $G(t_n)$ and the intermediate vertex states of previous windows (t_0, t_m) to incrementally compute results on the current window by calling WICM compute over the window $[t_m, t_n)$, having a *window size* of $(ws = m - n)$. The window width ws over which compute executes is configurable, statically by the user or dynamically chosen by TARIS for each window (Section VI).

Windowed ICM stores the states of a vertex at all past timesteps using interval states. This helps it achieve Zero Recomputation (Theorem 1) by storing all past results as vertex states. We need to use the states of the vertex at timesteps $t_x \forall x < i$ to send messages that expand the prior paths till timestep t_i (Section III-D2). WICM's interval messages help achieve this easily. The states of vertices at current timestep can be calculated using previous states by finding the smallest of all valid extended paths by the user's `compute` logic using the appropriate interval state and messages.

Algorithms like SSSP and FAST can have multiple computed values at different timesteps, and hence will have multiple interval states. When passing a message to an outgoing edge

we send the states of all these intervals to compute the correct answer. Even though the states of the vertices in these algorithms monotonically decrease, it does not suffice to store just the latest state. So, the interval states provided by WICM are essential to compute such temporal path algorithms, which platforms like Gradoop lack. On the other hand, for algorithms like EAT, TR and TMST, once we have a non-default state at a timestep, it never changes even if the graph mutates. So we will have at most *two* interval states per vertex for these.

E. Correctness of Incremental Computing

The output results of incremental processing using WICM after applying graph updates is equivalent to computing the ICM algorithm from scratch on the updated interval graph. In a window $[t_m, t_n)$, WICM only traverse paths till timestep t_n. This is ensured by filtering of interval messages passed to WICM compute where their lifespan is truncated to the sub-interval which overlaps with the current window. This guarantees correctness even when the graph changes in future timesteps, and reduces unnecessary messaging. Since TARIS applies updates to the graph $G(t_n)$ before starting compute till t_n, all the edges needed to send messages within this window are present. Any edge added at $t_i, \forall i > t_n$ would have $t_s > t_i$ and been filtered out even if the future timesteps were present in the graph. Also, WICM retains the partitioned temporal state at all previous timesteps and does not change past states ($R(t_j), \forall t_j < t_n$), allowing us to reuse them for incremental computation when expanding to the new timesteps.

We make two observations on incremental computation by WICM that helps with the empirical analysis.

Theorem 2: Amount of incremental computation performed is not strictly proportional to the number of mutations.

We omit a formal proof for this for brevity, but offer this intuition. Even with many mutations, only vertices which have not yet converged or have a viable expansion due to the mutations will participate in the computation. E.g., in TR, a vertex that is unreachable cannot be used in future paths even if new out-edges are added. Conversely, paths can also extend from existing edges to which no new mutations arrived.

Theorem 3: Deletions do not cause any new computation.

Here again, the intuition is that mutations that delete a vertex or edge truncates it's end time from ∞ to the current timestep, i.e., it cannot be used to extend any paths and requires no computation.

V. Scheduling Strategies

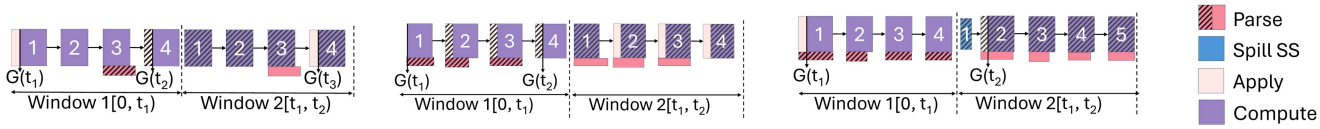
Parsing is done timestep at a time while apply and compute are done in mini-batches of timesteps to enhance throughput. The control engine uses one of several scheduling strategies to intelligently decide when to perform the parse and apply, and for the range of timesteps to perform them on. These are notified to the workers as part of the interaction with them in each superstep. Parse, apply and compute must happen sequentially for any given timestep, i.e., for a compute window of size ws to execute, all those timesteps must have been parsed and applied. But parse and apply of future timestep(s) can overlap with the compute of previous timestep(s) – this does not affect the correctness (proof given in Appendix Section A, available online).

We propose five different *scheduling strategies* for processing streaming updates, which combine various parsing and apply strategies for correct and efficient pipelined execution (Fig. 6). While JITM, AITM and DITM were proposed earlier in our short paper [27], the rest are novel. Here, we assume that the input update rate is unbounded and the parser does not have to wait if it is scheduled to parse the next timestep. We also assume that the WICM compute has a pre-defined execution window size of ws. These are both relaxed in Section VI where we adaptively select the window size for WICM compute.

A. Parsing Strategies

We have several options for when to parse ws timestep updates to prepare them for subsequent apply and compute.

1) *Parse once in current window (***P-cw***):* All ws timestep updates are parsed at the start of a new compute window and placed in the MStore, in the first and an exclusive superstep of that window. This strategy is simple, but increases the number of supersteps required for WICM compute of the window by 1, and introduces a delay between the end of the previous compute superstep and start of the next compute superstep since we do not overlap parsing with compute.
2) *Parse once in the previous window (***P-pw***):* All ws timestep updates are parsed at the penultimate compute superstep of the previous compute window. The parse thread runs concurrently with other compute threads of that superstep, overlapping the parse for the next window with compute of the current window and hiding costs. The parsed mutations stored in MStore do not affect compute over the graph. However, deciding which superstep is the penultimate one to trigger the parse requires us to know the number of supersteps needed for a compute window. We assume an "oracle" provides it.
3) *Parse a fixed number of timesteps in each superstep of previous window (***P-fs***):* In *P-pw*, parse time may exceed compute time in the penultimate window, depending on number of mutations in the update set. To hide this further, we spread equally the timesteps parsed across all compute supersteps of the current window. If the current compute window is expected to have s supersteps, we parse $\lceil \frac{ws}{(s-1)} \rceil$ timesteps in each; the last superstep is reserved for apply. While this reduces chances of parsing delays seen in *P-pw*, we may still encounter them if mutations are not uniformly distributed across all timesteps or all graph partitions. Here too we assume s is known.
4) *Parse a variable number of timesteps in each superstep of previous window (***P-vs***):* We try to fully hide the parsing time within the compute times of the previous window. At each compute superstep, the parse thread greedily reads as many timesteps as possible while any compute thread is active. So no extra time is spent exclusively for parsing at any superstep. This also avoids load imbalances across partitions or timesteps. Any timesteps of ws left unparsed at the penultimate superstep after completion of the compute threads are parsed in a separate superstep of the next compute window, like *P-cw*.



(a) AITM *(P-pw + A-pw)* (b) DITM *(P-fs + A-i)* (c) SpillRead *(P-vs + P-cw + A-cw)*

Fig. 6. Illustration of three of the five scheduling strategies, covering different Parse and Apply strategies. All hatched (or solid) refer to timesteps in the same window being processed by different phases.

B. Apply Strategies

Once mutations are parsed into the MStore, they can be applied concurrently by the compute threads onto the partition they operate on, in the pre-compute phase of a superstep.

1) *Apply once in current window (***A-cw***):* All update timesteps present in the MStore at the start of a new compute window are applied in the first compute superstep of the window by the compute thread for the partition, before compute for the window starts in the superstep. Since mutations are applied only once per window, it minimizes the number of memory accesses from MStore and to the graph partitions.
2) *Apply once in previous window (***A-pw***):* All update timesteps present in MStore at the last compute superstep of the previous window are applied to the interval graph partition. This happens as part of the pre-compute phase of the compute thread in the last superstep to ensure that the updated interval graph is ready for processing in the next compute window.
3) *Apply immediately after parsing in previous window (***A-i***):* Here, we perform the apply in the superstep immediately after one or more timesteps have been parsed in a previous superstep. When the parsing is distributed across supersteps, then the apply will also occur multiple times, in each succeeding superstep. This reduces the *per-superstep* apply time. Mutations are applied soon after they are parsed, reducing the peak MStore size and enabling a larger number of mutations per update set. However, each call to apply causes all vertices in the MStore and their adjacency lists in the Giraph object to be accessed, which can increase the *cumulative* apply time.

C. Scheduling Strategy Combinations

We formulate various scheduling strategies (Fig. 6) to update the interval graph by combining these parse and apply strategies. We list these below with a brief comment on the trade-offs, and evaluate them in our experiments in Section VII-B. These are further illustrated in Fig. 1 in Appendix, available online.

1) *Just in Time Mutation processing, JITM(P-cw + A-cw):* Processing of mutation updates required for a WICM compute window is done only just when it is necessary. Parsing happens in the first superstep of the a compute window while apply followed by compute start in the second superstep. JITM takes an extra superstep per window and does not optimize reads.
2) *Ahead in Time Mutation processing, AITM(P-pw + A-pw):* All parse and apply of mutations for the current window's compute are completed in the last two supersteps of the previous window's compute. It can perform better than JITM by saving an extra superstep, and the compute starts in the first superstep of current window.
3) *Distributed in Time Mutation processing DITM(P-fs + A-i):* Processing of mutations for a window happens across all supersteps of the previous compute window, with equal number (ϵ) of timesteps processed in each superstep. The parsing of timestep(s) $t_i..t_{i+\epsilon-1}$ in some superstep s_k happens concurrently with the apply of timestep(s) $t_{i-\epsilon-1}..t_{i-1}$ (which were parsed earlier) in the same superstep.
4) *DITM + AITM(P-fs + A-cw):* This combines the two above, by distributing the parsing of fixed number of timesteps across all supersteps in the previous window (DITM) but applying them all once in the first superstep of the next compute window, just before compute starts. It benefits both from parsing across distributed supersteps and one-time apply.
5) *Spill Read(P-vs + P-cw + A-cw):* It parses timesteps for a window at each superstep of the previous window as long as its compute threads are executing. Any remaining timesteps are parsed at the start of the next window in an exclusive initial *spill-read superstep*, prior to the window's apply and WICM compute initiation in the second superstep. All updates present in MStore are applied in this spill-read superstep.

VI. Dynamic Windowing Strategies

In this section, we handle mutations streaming in (rather than always available), which affects the availability of future update timesteps during the parse phase. We propose *windowing strategies* that dynamically decide the window size ws for the upcoming WICM compute window. They adapt to the input mutation rate to reduce the processing latency per timestep and increase the throughput per execution window.

A. Performance Metrics

The *input timestep rate* is the number of update timesteps that arrive in each unit of wallclock time. The *input mutation rate* is the number of mutations arriving in each unit of wallclock time, i.e., by adding up the mutation counts present in the timestep updates. The *timestep latency* for t_i is the duration between the wallclock arrival time of its update set $\Delta(t_i)$ at TARIS, and the time when TARIS returns its result set $R(t_i)$ on $G(t_i)$. This includes the *parse, apply and WICM compute latencies*, and other delays before it participates in a window execution (Fig. 7). Since WICM compute executes on a window, the *window latency* for $t_i..t_{(i+ws-1)}$ of window size ws is the time between the WICM

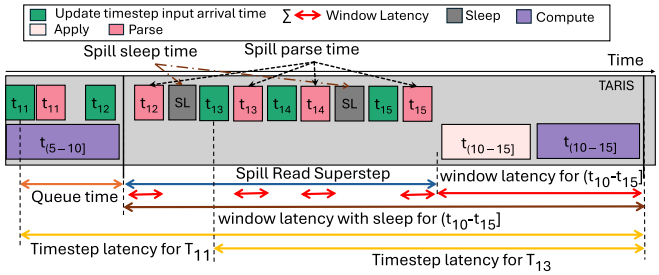


Fig. 7. Performance metrics for dynamic window strategies.

compute for the previous window, $t_{(i-ws)}..t_{(i-1)}$, ending and the compute for the current window ending, less the "sleep time" if present, as discussed next. The *window throughput* is the average number of *temporal entities* processed per second. This is reported after each window's execution as the sum of all vertices and edges present in the temporal snapshots of the window divided by the window latency.

B. Updated Scheduling Strategy: Spill Read With Sleep

As we report later, *SpillRead* performs the best among the scheduling strategy. However, it needs to be modified to respond to the externally determined timestep input rate. Specifically, in the spill-read superstep, if the parser wishes to process the next timestep update to help it reach ws timesteps for the next window's compute but if the update has not yet arrived, then we *sleep* until the next update timestep is received and then perform the parse (Fig. 7). Such a sleep and parse happen iteratively till all ws timesteps have been received and parsed. A spill-read superstep occurs only after the prior compute window has finished, and it only does sleep and parse. It's latency contains *sleep latency*, the sum of the sleep times; and *spill-parse latency*, the time spent in parsing. We omit sleep latency when computing the window latency (and hence throughput) since it is caused by the application's input rate.

C. Dynamic Selection of Window Sizes

Sleeps occur if the application's input timestep rate is slower than TARIS's processing rate. But, if the input rate is faster than TARIS's speed, then a *queuing latency* occurs. This is caused by the previous window still being processed even as the updates for future windows have arrived. This accumulates for all timestep updates that have arrived for future windows.

Since parse, apply and compute are necessary steps, we focus on reducing (or removing) the sleep and queuing latency to achieve a *low timestep latency* and a *high window throughput*. The choice of window size affects these and using a *fixed window size (FixedWS)* is inadequate – if the input rate is low, then waiting for a large ws timesteps to arrive will increase the sleep latency; if the input rate is high, then processing many small ws windows will increase the queuing latency.

We design several strategies to dynamically select the window size, i.e., decide when we have adequate timesteps to trigger the next window's apply and compute. The *Greedy Window Strategy (GRD)* fully avoids the sleep latency. It eliminates the spill-read superstep (hence sleep latency) and only considers update timesteps that have been received and parsed at the end of the WICM compute of the previous window. The next window's apply and compute occur on the parsed timesteps present in the MStore and this becomes its window size. GRD naturally adapts to mutation rate variability, which occurs even for a static input timestep rate. If there are fewer mutations per timestep, then the time to parse them will be low and more parsed timesteps will be ready for the next compute window. If the mutation rate is high, fewer timesteps will be parsed and the next window's size will reduce.

But GRD can cause high queuing latency for high input rates and low window throughput for low input rates. We curtail this in the *MINWS strategy* by setting a *minimum window size* ($ws_{\min}$) that must have arrived and be parsed (in a spill-read superstep, if required) before we initiate a new window's apply and compute. This will increase the throughput for low input rates and reduce the queuing latency for higher rates. We further extend this in the *MAXSR strategy* by setting a *maximum spill-read latency* ($sr_{\max}$) for which the spill-read superstep will operate, beyond which the next window's execution to start even if $ws_{\min}$ timesteps are not ready. This avoids a high sleep latency for low input rates. Lastly, we modify this further in the *MAXSL strategy* where we set a *maximum sleep latency* ($sl_{\max}$) in the spill-read superstep, allowing parsing of updates that have arrived to proceed to completion. This further increases the throughput for slow input rates while bounding the sleep time and ensuring the spill-read time is used to prioritize updates that already arrived.

These dynamic strategies focus on the streaming aspects of TARIS and complement heuristics proposed earlier in WICM [21], which assumed full knowledge of the entire temporal graph to choose the window sizes and used analytical models to estimate the WICM compute time.

VII. Experiments and Results

We evaluate the scheduling strategies of TARIS for diverse interval graphs, temporal graph algorithms and static window sizes, and compare them with the SOTA platforms, Tink [8] and Gradoop [16]. We further evaluate the dynamic windowing strategies of TARIS for multiple streaming configurations to analyze the timestep latency and throughput achieved.

A. Experiment Setup

We evaluate five *time-respecting traversal algorithms* widely used in literature and having different computing needs: Temporal Reachability (TR) [9], [21], [31], Earliest Arrival Time (EAT) [6], [7], [21], [33], Temporal Minimum Spanning Tree (TMST) [7], [21], [45], temporal fastest (time) path (FAST) [6], [7], [21], [33] and Temporal Single Source Shortest Path (TSSSP) [6], [8], [15], [21].

We evaluate the algorithms on four realistic and large-sized *graphs and streaming workloads* (Table II). *Reddit* [42] is a social network graph of comments from users; *LDBC-365* is a synthetic graph with 365 daily snapshots generated using Facebook's community structure on which we introduce temporal variations using the by the Linked Data Benchmark Council (LDBC) Datagen tool [43]; *LDBC-Static* imposes a static mutation rate of 2 M per timestep on LDBC-365; and the *Twitter* [44]

TABLE II
DATASETS USED IN EXPERIMENTS

Graph	#Vertices	#Edges	#Timesteps	#Mutations/ts
Reddit [42]	9.3M	528.2M	122	100–50M
LDBC-365 [43]	10.5M	848.6M	365	1K–14.8M
LDBC-S [43]	10.5M	848.6M	425	2M
Twitter [44]	52.5M	1.9B	90	10M

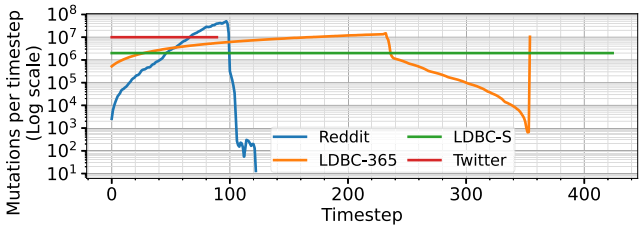


Fig. 8. Mutations per timestep for streaming graph updates.

TABLE III
NUMBER OF VALID TEMPORAL PATHS FROM 3 DIFFERENT SOURCE VERTICES TO SINK VERTICES FOR THE GRAPHS EVALUATED

Src. Vert.	Reddit	LDBC-365	LDBC-S	Twitter
$v1$	3,980,720	8,353,578	8,355,998	49,107,784
$v2$	3,980,248	4,283,481	4,282,628	47,149,722
$v3$	3,979,191	9,987,651	9,989,981	48,844,268

social network graph has mutations synthetically introduced at a static rate of 10 M updates per timestep for 90 timesteps. While LDBC-S and Twitter have a static number of mutations per timestep, and Reddit and LDBC-365 have a dynamic rate ranging from 100–50 M mutations/ts (Fig. 8).

We confirm the empirical correctness of TARIS by verifying that its result set is identical to that of batch computation for all runs, i.e., for each sink vertex and at each timestep, we have the exact same values of arrival time for EAT, shortest distance for TSSSP, parent vertex for TMST, fastest time for FAST and true/false for Reachability. Table III reports the count of sink vertices with a valid temporal path (non-default time/distance) from three different source vertices for all graphs. These match exactly for TARIS, batch execution and the SOTA baselines.

For the scheduling strategies (Section VII-B) and initial part of SOTA comparison (Section VII-C), we assume that the timesteps are accessible on-demand at the rate at which the platforms consume them. We report the end-to-end processing time for the full temporal graph for these algorithms. For the second part of SOTA evaluation and for the windowing strategies (Section VII-E), we replay the graphs at different input timestep rates, ranging from 5 ts/min to 6400 ts/min, depending on the graph, to stress the TARIS system. These translate to a peak mutation rate of up to 587 M mutations/sec.

Experiments for smaller graphs (Reddit, LDBC-365, LDBC-S) are run on a commodity cluster *(polaris)* with compute nodes having an Intel Xeon E5-2620 v4 CPU with 8 cores@2.10 GHz, 64 GB RAM connected over a 1 Gbps Ethernet. For the larger graph (Twitter) that ran Out of Memory (OOM), we use another commodity cluster *(sirius)* with compute nodes having an Intel Xeon Gold 6226R CPU with 16 cores@2.90 GHz, 128GB RAM and 10 Gbps Ethernet. We also use *sirius* for SOTA comparisons since Tink and Gradoop needed large-memory nodes even for smaller graphs. All nodes run CentOS v7, Java v8 and Hadoop v3.1.1. Apache Giraph v1.4.0, on which TARIS is built, runs on 8 worker nodes with 8 compute threads each and 1 leader node.

B. Comparison of Scheduling Strategies

We evaluate the scheduling strategies (Section V-C) on the 4 graphs using all 5 algorithms. The strategies use a static window size, which is configured as $ws = 10, 15, 20, 25$. For brevity, we report the effect of various window sizes using TMST on LDBC-365 (Fig. 9(a)), the effect of different algorithms using Reddit for $ws = 20$ (Fig. 9(b)), and the effect of diverse graphs running EAT using $ws = 20$ (Fig. 9(c)); additional plots are included in Fig. 16 in the Appendix, available online.

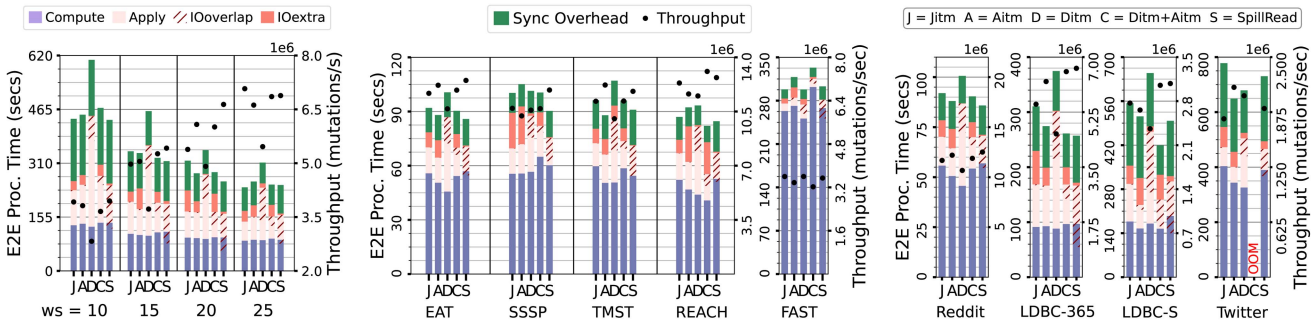
We report the *end-to-end (e2e) processing time* for each graph as the cumulative time (left Y axis) across all windows and split the bars into component times for *compute* (purple), *apply* (peach), and *parse*; since parse runs concurrently with apply and compute, it is split into *IOextra* (orange) time exclusively spent only for parsing, and *IOoverlap* (red hatch) where parse overlaps with the apply and compute, and overlays their bars. The time remaining is the *overheads* for superstep synchronization and writing the output to HDFS (green). The right Y axis markers show *throughput*, the average of mutations processed per second in each window. All plots show averages over 3–5 runs from different source vertices.

1) Effect of Parse Strategies: Fig. 9(a) shows the impact of parse strategies for TMST on LDBC-365. The parse times are seen in *IOextra* (orange bar), which is the cost for only parsing and must be reduced, and *IOoverlap* (red), which hides the parse time within apply/compute and lowers the e2e time.

Across all window sizes ($ws = 10, 15, 20, 25$), JITM, AITM and DITM+AITM (C) have a higher IOExtra time, 30–60 s (5–20% of e2e time), as compared to DITM (D) and SpillRead (S), which take just 2–10s ($\leq 2\%$ of e2e). JITM and AITM use *P-cw* and *P-pw*, respectively, to read and parse *all* timesteps needed for the next window in a *single* superstep. P-cw runs in a separate superstep and hence has zero IOoverlap (red hatch). So JITM's parse is seen only in IOextra, taking, e.g., 60 s for $ws = 20$. While P-pw runs concurrently with the compute of the penultimate superstep of the prior window, the compute time in this superstep is short at ≈ 0.01 s, causing the IOoverlap of AITM to be negligible and IOextra to be higher, e.g., 30 s for $ws = 20$.

DITM+AITM uses *P-fs* that parses a fixed number of timesteps at each superstep. We see that compute is heavy in the first superstep of each window and light in all others (≈ 0.01–0.03 s). So parsing a static timestep count exceeds the compute time of these latter supersteps, resulting in higher IOextra time (52 s for $ws = 20$) and lower IOoverlap (38 s). DITM also uses P-fs but can hide the IOextra time as parsing happens in parallel to both apply and compute for every superstep. Since apply time is higher in DITM, it has low IOextra (2 s for $ws = 20$) and a higher IOoverlap (68 s).

SpillRead using *P-vs* largely overlaps parsing with compute. It has the maximum IOoverlap with apply and compute, e.g., 38 s for $ws = 20$. It almost eliminates IOextra time (5 s) by parsing timesteps in a superstep only until compute runs, but completes



(a) Window sizes vary for TMST algo. on LDBC-365 graph

(b) Different algorithms on Reddit graph, $ws = 20$.

(c) Different graphs running EAT algo., $ws = 20$.

Fig. 9. Comparing scheduling strategies for different graphs, algorithms and window sizes.

parsing a timestep it had started even if compute has finished. This results in some IOextra time for timesteps with a spike in the number of mutations, but is minimized if the number of mutations is consistently high since their compute time will also be high. This maximum IOoverlap and minimum IOextra time holds for all window sizes of LDBC-365 on TMST (Fig. 9(a)). This is also consistent across other graph algorithms running on Reddit (Fig. 9(b); $< 1\%$ of e2e time for IOextra, 10–20% for IOoverlap), and across other graphs for EAT (Fig. 9(c); $< 3\%$ for IOextra, 18–43% for IOoverlap).

2) Effect of Apply Strategies: All apply strategies benefit from the MStore data-structure optimizations of storing and grouping mutations that results in fewer vertex fetch and adjacency list scans, e.g., lowering the e2e time by 62%, from 245 s [27] to 92 s, for EAT on Reddit graph using JITM with $ws = 20$. Further, apply time is affected more by the number of vertices with mutations rather than the number of mutations, e.g., taking the same time of 1 s to apply $\approx$ 36 M mutations (18 M add edges, 18 M delete edges and 0.3 M add vertices) on 2.5 M vertices of LDBC-365, and to apply $\approx$ 61 M mutations (33 M add edges, 26 M delete edges, 2 M add vertices and 0.7 M delete vertices) on 2.5 M vertices for Twitter.

In the apply time (peach bar) for TMST on LDBC-365in Fig. 9(a), we observe that all strategies take a similar duration for a given window size, e.g., 68–75 s for $ws = 20$, except for DITM which takes longer (184 s). JITM, DITM+AITM and SpillRead all use *A-cw* while AITM uses *A-pw*. Both these apply strategies apply all timesteps for next window in a single superstep. This batching results in a faster apply time. In contrast, *A-i* distributes apply across supersteps with fewer timesteps per superstep. The static overhead of scanning the adjacency list of updated vertices from the MStore is seen to be twice as many for A-i as for A-cw or A-pw, e.g., 36 M v. 15 M for EAT on Reddit. This affects DITM that uses *A-i* and takes 2–3$\times$ longer for apply, which is also seen across algorithms and graphs, e.g., taking 37.6 s for REACH on Reddit compared to 14.4–15.5 s for others (Fig. 9(b)), and 178.8 s for EAT on Twitter compared to 57.2–76.2 s for others (Fig. 9(c)) Other strategies have a similar apply time but with minor variations due to garbage collection and memory allocation overheads.

3) Other Observations: The *compute* time for all strategies is similar for a given window size, e.g., taking 130–138 s for $ws = 10$ and 87–92 s for $ws = 25$ (Fig. 9(a)). As for apply, there are minor variations due to batching of network messages and memory access overheads. Larger window sizes tend to have lesser apply time initially and also a smaller compute time due to fewer compute window executions, fewer cumulative supersteps and a lower window synchronizations overheads (green bar), and this tends to increase the throughput.

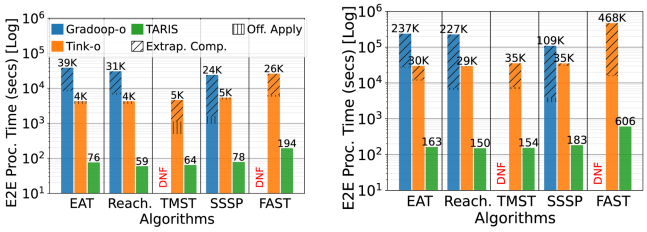
Across graphs, those with a higher vertex degrees tend to be compute heavy as they result in more edge-level processing and number of messages. So for Reddit, whose high-degree vertices have $\approx 9k$ edges, the compute time forms 64% of the e2e time compared to LDBC-365, with $\approx 1.5k$ peak vertex degree, whose compute is 38% of e2e time (Fig. 9(c)). In algorithms like FAST, the compute time dominates, being 4–5$\times$ longer for other algorithms (Fig. 9(b)). Here, the distinction between scheduling strategies is more muted.

In summary, *SpillRead is the best scheduling strategy* since it uses both *P-vs* and *A-cw* which reduce the e2e time. It is consistently better than (or occasionally comparable to) all other strategies, across algorithms, graphs and window sizes. It also has 3–45% higher throughput. Hence, it serves as our default scheduling strategies in future experiments.

C. Comparison With SOTA Baselines

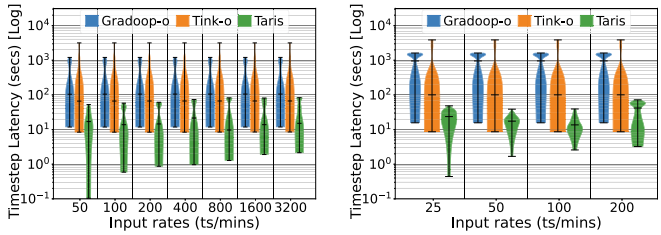
We compare TARIS with two contemporary platforms, *Temporal-Gradoop v0.6.0* [16] and *Tink v1.0.0* [8]. These use the Gelly graph library of Apache Flink, using *v1.7.2* and *v1.1.3*, respectively and run all on the *sirius* cluster. We also design and implement the five temporal algorithms using these frameworks since these were lacking natively. However, we omit FAST and TMST for Gradoop since they were prohibitively slow ($> 10h$ per run). Similarly, we evaluate these only for Reddit and LDBC-365 whose mutation rate vary, and omit Twitter (OOM) and LDBC-S (for brevity).

As discussed in Section I-B, we generate the updated snapshot graphs offline after each timestep update and then process them using *Tink-o* and *Gradoop-o*; we omit online updates to Gradoop since it is 2$\times$ slower than using the offline update script. This pre-processing time took $\approx$ 632 s for all timesteps of the graph for Reddit and $\approx$ 939 s for LDBC-365 graph, and is included in the end-to-end processing time in Fig. 10. This is equivalent to the parse and apply done online by TARIS. The hatch regions



(a) Reddit (b) LDBC-365

Fig. 10. E2E Processing Time for TARIS, Tink and Gradoop, for different graphs and algorithms.



(a) Reddit (b) LDBC-365

Fig. 11. Timestep latency for TARIS, Tink and Gradoop for EAT on streaming graphs with different timestep updates rates.

in some bars indicate repeated failures for some timesteps of Gradoop-o, which we skip and extrapolate.

In the first set of experiments, we assume that the update timesteps are always available when required (like in Section VII-B). We compare TARIS using SpillRead and $ws = 20$ with Tink-o and Gradoop-o. In Fig. 10 (log scale), the time taken by TARIS (green) is faster than Tink-o (orange) and Gradoop-o (blue) by 170–1500× for all five algorithms on both graphs.

To explain this wide variation, use the execution of SSSP on Reddit, where Gradoop-o, Tink-o and TARIS took $24026s, 5428$ s and 78 s to process all 122 timesteps (Fig. 10), and focus on a representative timestep update, t_{92}, which takes 10 m 30 s, 1 m 29 s and 30 s on these platforms, with the TARIS time being the e2e time for the *entire compute window* with 20 timesteps and including t_{92}. We drill this down into the *load, compute and write times*.

Load and Parse: For Gradoop and Tink, we form an interval graph that includes updates from timesteps 1–92 offline. This takes 575 s to generate the graph, and 878 s and 135 s and to write it to their native file formats of size $50.2GB$ and $8.3GB$; Gradoop uses a more complex format than the simple CSV file used by Tink. In contrast, TARIS only loads the update mutations for t_{92} from file, which is 203 MB in size, during its parse phase, which takes just 2.9 s.

Apply: Next, Gradoop and Tink read the interval graph to form the distributed in-memory graph structure file at t_{92}, and take 132 s and 11 s. Gradoop takes longer both due to a larger input file size and its complex logical graph structure. The equivalent in TARIS is the apply of a *window* of mutations from MStore to the in-memory interval graph, which takes just 5.9 s for the apply window containing t_{92} having a total of 20 timesteps. The update-efficient MStore data structure and update patterns that minimize memory access help.

Compute: The compute time for Gradoop and Tink compute are 6 m 32 s and 1 m 14 s for t_{92} while the time to execute the compute window with 20 timesteps, including t_{92}, by TARIS just is 14.5 s. TARIS incrementally extends the states from previous timestep results by one hop in each traversal to avoid recomputation, while Tink and Gradoop compute the paths starting from the source vertices in each timestep, which increases the load and compute overheads. Further, ICM on which TARIS is built upon stores interval states as range maps and uses TimeWarp, which reduces the time to align and update temporal messages with the temporal states; Tink and Gradoop perform a linear search over all states of a vertex when processing messages and updating states. Also, Gradoop's compute time is worse than Tink due to its complex Flink execution DAG which and custom operators, e.g., has 11 stages and 33 tasks compared to 5 stages and 9 tasks for Tink. Like TARIS, both Gradoop and Tink overlap load, compute and write phases, though only we perform online apply while the other two perform this offline.

Write: Lastly, Gradoop's complex graph structure also increases the time to write its output to file at 6 m 59 s for t_{92} compared to 1 s for Tink. In TARIS, this is part of each vertex-compute and forms part of the platform sync overhead, which is 10 s for the compute window with 20 timesteps.

Next, we evaluate a streaming scenario where for the EAT algorithm on the Reddit graph, the timestep update arrive at a given input rate. For TARIS, we use the simple *FixedWS strategy* with a fixed window size of $ws = 15$. We vary the input timestep rates between 50–3200 ts/ min for Reddit and 25–200 ts/ min for LDBC-365. We report the *timestep latency* taken for graph update and EAT compute for each timestep update as a violin plot distribution in Fig. 11. The time taken by Tink-o and Gradoop-o are nearly an order of magnitude higher than TARIS for both graphs on EAT, e.g., taking a median of 100 s and 1000 s, respectively, for an input rate of 100 ts/ min (mutation rate of $83k$–587 M/s) for the LDBC-365 graph compared to just 15 s for TARIS. This is despite our windowed execution that buffers timestep updates and may introduce sleep times. These results underscore TARIS's effectiveness in reducing the graph update and incremental computing times, positioning it as a superior framework for large-scale processing of streaming graph.

D. COST and Strong Scaling Evaluation

The scalability of distributed graph platforms have been shown to be inefficient, with improvements that are relative to sub-optimal single-machine performance [46]. We attempt a *simple COST evaluation* (Configuration that Outperforms a Single Thread) of TARIS against a custom single-threaded *non-incremental* implementation of the EAT time-respecting algorithm in Java (ST-EAT). We extend Dijkstra's shortest path algorithm with constraints to enforce time-respecting paths over a timestep of the interval graph. This is then executed for each timestep of the graph, similar to other baselines. The result-set of this algorithm matches the others.

In Fig. 12, we compare ST-EAT (pink) against *single machine/single threaded versions* of TARIS (green) and the Tink and Gradoop baselines, for EAT on Reddit run on a sirius compute node (1W-1T). The black hatches show the time to load and apply the graph. TARIS out-performs Tink and Gradoop

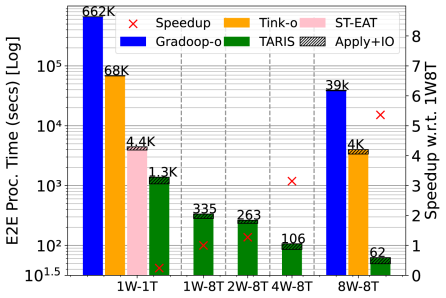


Fig. 12. Performance of TARIS and baseline distributed platforms on a single-threaded worker *(1W-1T)*, along with single-threaded EAT (ST EAT). Strong scaling of TARIS for 1 to 8 workers *(1W–8W)*, with speedup relative to 1W-8T on right Y axis. Results are for EAT on Reddit graph.

as expected. It is also better than ST-EAT, taking $1,379$ secs of end-to-end time to incrementally process the 122 timesteps of Reddit compared to $4,447$ secs for ST-EAT. This is largely due to the incremental processing optimization in TARIS that avoids recomputing at each timestep, unlike ST-EAT. We also have modest benefits from an efficient MStore and apply for graph updates. *This confirms that TARIS is a Configuration that Outperforms a Single Thread (COST).*

TARIS also is able to exploit *multiple threads* on a single worker, reducing its E2E time from $1,379$ secs to 335 secs when we increase from 1 to 8 threads on a single worker. Further, it shows a reasonably good *strong scaling* as we increase 1 to 8 workers, with 8 threads each, with the E2E time reducing to 62 secs with 8 workers. The peak scaling efficiency of 79% is with 4 workers ($3.1\times$ speedup). This drops 68% efficiency but a better speedup of $5.4\times$ with 8 workers. So TARIS offers scalability at a reasonable cost.

Lastly, while the large Twitter graph is unable to load in-memory on a single node with 120 GB heap space by either TARIS or ST-EAT, TARIS can load and run it on 8 nodes. This too is a benefit of distributed memory processing.

E. Streaming Strategies

We evaluate TARIS in more detail for streaming scenarios. Here, updates arrive at different input timestep rates, similar to real world systems, with different mutation counts per timestep. We compare our windowing strategies (Section VI) and ascertain their ability to dynamically adapt to mutation rates.

We run experiments using the EAT algorithm, as it is representative of other traversal variants like TMST and reachability, on the four temporal graphs with various timestep input rate of 5–6400 ts/ min. This translates to mutation rates of 83 K–587 M $mutations/s$. The default SpillRead scheduling strategy is used. We compare the *four dynamic windowing strategies* that adapt the window size against the baseline with *fixed window size*. This window size is set the same as the minimum window size threshold ($ws_{\min}$): 15 for LDBC-S and LDBC-365, 10 for Twitter and 20 for Reddit. We set both the maximum spill-read latency ($sr_{\max}$) and maximum sleep latency ($sl_{\max}$) to 10 s for the dynamic strategies, based on representative performance on different graphs.

For our analysis, we categorize the input timestep rate as *slow* or *fast*, depending on whether this is slower or faster than EAT's WICM compute time to process them using a fixed window, as this is a key factor in adapting to dynamism. For Reddit, LDBC-365, LDBC-S and Twitter, the compute time ratios from Fig. 9(c) are $\approx 1:2:3:8$, whose inverse translates to timestep rates of around $< 1120, 560, 400, 140$ ts/ min, respectively, to qualify as a slow rate.

The two key performance metrics we report are the *timestep latency* to process each update since its arrival, and the *window throughput*, which is the average number of temporal vertices and edge in all timesteps processed in each window divided by the window latency ($entities/s$). These are shown as box plots across timestep rates along with a distribution of the dynamic window sizes selected in Fig. 13, for each windowing strategy on the different streaming graphs, for slow and fast rates. The Appendix, available online, gives a line plot across rates (Fig. 3) and splits the latencies into queue and spill times (Fig. 4). Fig. 14 zooms in on a timeline plot for LDBC-S with a slow (100 ts/ min) and a fast (400 ts/ min) rate. We highlight several key insights.

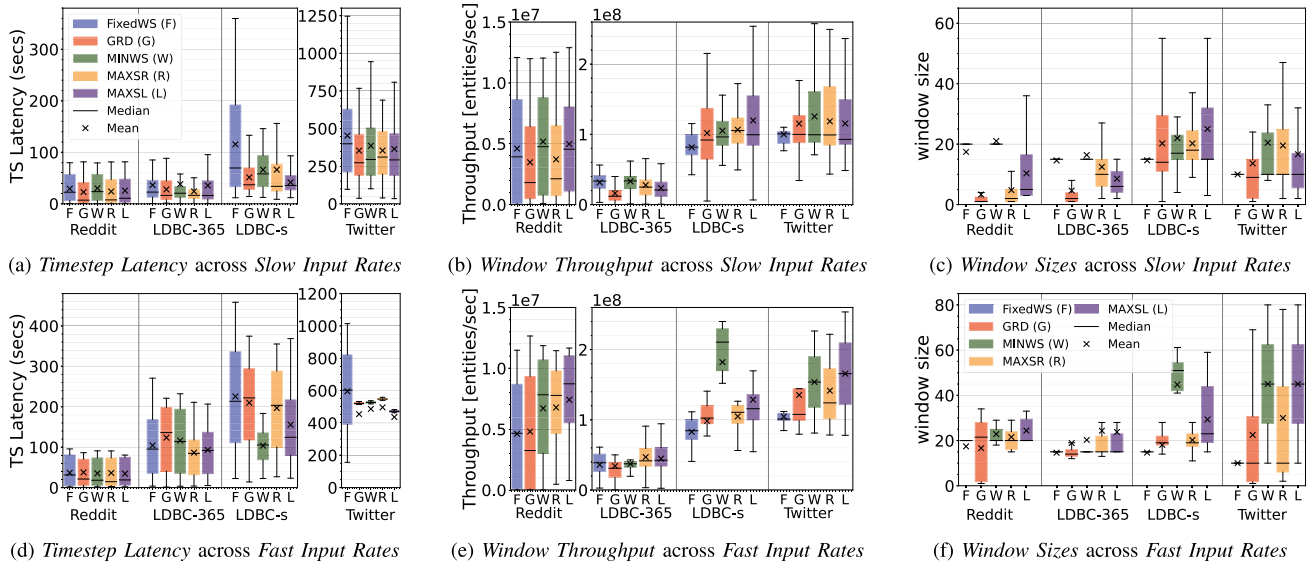
1) A Fixed Window Size is not Efficient, Either for Latency or for Throughput: We see that the average latency for *FixedWS* strategy (Fig. 13(a) and (d), blue box) is consistently higher than all other strategies for all four graphs for slow input rates, and is comparable to others for fast rates. On average, it has up to 177% higher mean latency across all rates compared to the rest (Fig. 5(a) in Appendix, available online).

At lower rates, there is a higher *sleep time* for FixedWS (and MinWS) (Fig. 4(b) in Appendix, available online), as they wait and accumulate $ws_{\min}$ (10–20) timesteps before a compute, as seen in Fig. 13(c), where their window sizes are usually higher than others. This sleep time drops as the timestep rate increases, from 5 s to 1 s per window from 50 ts/ min to 400 ts/ min for EAT on LDBC-S.

On the other hand, it will have higher *queue times* when the input rate is high since it will not compute more than $ws_{\min}$ timesteps even if available (Fig. 13(e)), forcing them to wait (queue). This is seen for LDBC-S and Twitter (Fig. 4(c) in Appendix, available online), where fixedWS is 17–297% higher than MAXSL, the nominally best performing one. This also accumulates over time since delays in earlier windows are also included in future ones, dominating the timestep latency in Fig. 14 where for LDBC-S, the queue time grows from 60 s to 260 s between timestep 100 to 400 for 100 ts/ min, and from 80 s to 420 s for 400 ts/ min. This lower window size with a higher latency causes *FixedWS* to have a lower throughput, e.g., by 15–33% for for LDBC-S and Twitter (Fig. 13(b) and (e)).

Next, we examine the dynamic strategies – GRD, MINWS, MAXSR and MAXSL – which adapt the next window's size.

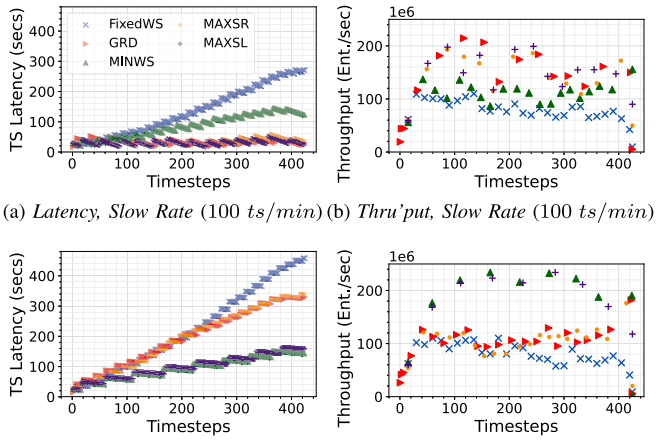
2) GRD has a Lower Latency for Slow Input Rates Compared to MINWS, While MINWS has Lower Latency for Fast Rates: We first contrast GRD and MINWS for a *slow input rate*. GRD computes over available parsed timesteps and does not wait for new timesteps to arrive. So TARIS does not have any *sleep time*. But MINWS waits for $ws_{\min}$ timesteps to arrive, leading to additional sleep time and spill parse during the spill-read superstep. In Fig. 13(a), MINWS (green) has 9–36% higher mean latency compared to GRD (red), of which 3–13 s per timestep is spent in sleeping (Fig. 6 in Appendix, available online).



(a) *Timestep Latency* across *Slow Input Rates* (b) *Window Throughput* across *Slow Input Rates* (c) *Window Sizes* across *Slow Input Rates*

(d) *Timestep Latency* across *Fast Input Rates* (e) *Window Throughput* across *Fast Input Rates* (f) *Window Sizes* across *Fast Input Rates*

Fig. 13. Comparing performance of all windowing strategies for EAT on all graphs for slow and fast input rates.



(a) *Latency, Slow Rate* (100 ts/min) (b) *Thru'put, Slow Rate* (100 ts/min)

(c) *Latency, Fast Rate* (400 ts/min) (d) *Thru'put, Fast Rate* (400 ts/min)

Fig. 14. *Timestep Latency* and *Window Throughput* across *arrival timesteps* for various strategies performing EAT over slow and fast LDBC-S input rates.

The *queue time* component of latency occurs due to sleeps that accumulate, and/or longer computes that process, prior windows. GRD has no sleep and prevents any contribution to the queue time at lower rates while the spill-read waiting time of MINWS adds to its queue time. In Fig. 14(a), for LDBC-S arriving at a slower rate of 100 ts/ min GRD has a lower latency of ≈ 34 s with a modest queue time of < 20 s caused by prior window's compute, while for MINWS the latency keeps increasing due to queue time increasing from 20 s to 110 s between timestep 100 to 400 (Fig. 6(a) in Appendix, available online).

GRD also has *smaller compute window sizes* at slow rates since it accumulates fewer timesteps, while MINWS accumulates at least ws_{mins} timesteps. This is clear in Fig. 13(c), where GRD has a mean window size of 3, 4, 20 and 13 for Reddit, LDBC-365, LDBC-S and Twitter while MINWS has window sizes close to $ws_{\min}$ of 21, 16, 22 and 20. Larger windows can lead to a higher throughput, as seen in Fig. 9(a) – as window size increases, the amount of computation also increases but the number of mutations and temporal entities in the window increases at a greater rate, thus increasing the overall throughput. While MINWS has a higher latency than GRD, its even higher window sizes (50–350% larger) results in throughputs that are on average 3–108% higher (Fig. 13(b)).

Next, we contrast these two strategies for *faster input rates* where their behavior inverts. MINWS has a smaller *sleep time* at faster rates, under 1 s on average, since the faster input rates fill its window quickly, avoiding the need for spill reads. So it approaches the zero sleep time of GRD. On the other hand, the mean latency of GRD is 7%, 6% and 102% higher than MinWS for Reddit, LDBC-365 and LDBC-S, respectively, due to 6%, 7% and 136% higher *queue time* (Fig. 13(d)). For GRD, a fast input rate means that more timesteps arrive while computing the previous windows but only a fraction of them are parsed, resulting in future timesteps waiting for multiple prior compute windows to execute. E.g., for LDBC-S at 400 ts/ min fast rate (Fig. 14(c)), MINWS parses more timesteps in the initial windows during a longer window execution and hence has lower queue time subsequent windows (Fig. 6(b) in Appendix, available online), compared to GRD where the initial window sizes and hence compute times are small, leading to fewer parsed timesteps and smaller compute windows in future too, and a queuing time that keeps growing over timesteps.

The smaller window sizes of GRD also results in a lower throughput compared to MINWS. In Fig. 13(f), GRD has 27–59% smaller window sizes, which leads to 8–46% lower throughput (Fig. 13(e)) relative to MINWS.

3) MAXSL Achieves a Low Latency for Both Slow and Fast Input Rates: MAXSR and MAXSL perform similarly at a *slower input rate*, with a better latency than GRD and also a higher throughput. MAXSR limits the sleep and parse during spill-read to within $sr_{\max} = 10$ s while MAXSL imposes the $sl_{\max} = 10$ s threshold only for sleeping but not parsing. At

slow rates, bounding the sleep cause both to perform better than MINWS by 7–58% for all graphs (Fig. 13(a)). Like GRD, both also reduce the queue time for future windows and have a comparable timestep latency. E.g., in Fig. 14(a), for 100 ts/ min rate in LDBC-S, MAXSR (orange) and MAXSL (purple) have low latency of 34.5 s and 31.2 s, similar to GRD.

MAXSR and MAXSL also allow larger window sizes to accumulate than GRD due to a bounded (rather than non-existent) spill-read, leading to 40–177% larger window sizes (Fig. 13(c)). A larger window size with similar latency gives 40–70% higher average throughput (Fig. 13(b)). E.g., MAXSR and MAXSL process 140.6 M and 148.4 M $entities/s$ (Fig. 14(a)) compared to 127.6 M $entities/s$ for GRD.

At *fast input rates*, MAXSL outperforms MAXSR. The $sr_{\max}$ bound of MAXSR causes parsing to be limited during spill-read and reduces the windows sizes by 13–50% compared to MAXSL, except for LDBC-365. So queue times increase as even available timesteps must wait till the next window for parsing (Fig. 14(c)), similar to GRD. Since MAXSL imposes this threshold only for sleep and not parse, it has high window sizes, reducing queue latency similar to MINWS. This gives a 4–20% lower latency for MAXSL than MAXSR for all graphs but LDBC-365 (Fig. 13). Due to a higher window size and lower latency, MAXSL also has 9–22% higher throughput.

In summary, MAXSL performs better for all input rates, leveraging best characteristics of GRD by having bounded sleep at slow rates(Fig. 14(a)) and of MINWS with proactive parsing during spill-read giving larger windows and smaller queue times(Fig. 14(c)). These result in a lower latency at all rates, on average by $21\%, 9\%, 7\%$ and 2% (Fig. 5(a) in Appendix, available online), compared to FixedWS, GRD, MinWS and MAXSR.

4) Discussion on the Streaming Graphs: LDBC-S has a constant number of mutations in all timesteps and hence less fluctuations. It clearly highlights the effects of all windowing strategies and forms the basis for our analyses. Reddit and LDBC-365 have a highly varying mutation rate, from 100–50 M at each timestep. Hence, there more noise in the compute time and the distinctions in performance of the windowing strategies is not as clear. Also, Reddit is comparatively smaller than others and hence has fewer vertices and edges to be processed in each window. So the quanta by which latency can be reduced is smaller. Twitter is a huge with ≈ 1.9 B edges. It is compute heavy and hence all dynamic windowing strategies finish parsing all timesteps of the graph just after the first two windows for all input rates. Also, apply and parse form only $\approx 10\%$ of the latency compared to 25% for LDBC-S. So the contrast between the dynamic strategies is muted. FixedWS has a high latency as it limits the timesteps per window.

VIII. Conclusion

In this article, we have motivated the need for incremental and scalable processing of time-respecting algorithms over streaming graphs based on a clear gap in literature. We have designed TARIS, a novel distributed framework based on an Interval-centric Computing Model, which incorporates incremental computation with strong guarantees of correctness. It uses efficient data structures and proposes scheduling and windowing strategies to stagger parse and apply of the streaming updates, and performs windowed compute intelligently.

Our detailed experiments with realistic large-scale graphs and several temporal algorithms show that TARIS outperforms state-of-the-art frameworks like Tink and Gradoop by a considerable margin. TARIS handles hundreds of millions of mutations per sec. While all dynamic streaming strategies outperform the fixed window strategy, the MAXSL streaming strategy is able to adapt to both slow and high update arrival rates to reduce the timestep latency that affects responsiveness and maximize throughput to achieve scalability. As future work, it is worth exploring single-machine multi-threaded implementations of ICM with the incremental processing and MStore apply optimizations proposed in TARIS, which are generalizable.

Acknowledgment

The authors would like to thank members of the DREAM:Lab who helped with this article, including Animesh Baranwal and Varad Kulkarni. We also thank Prashanthi Kadambi and Suman Raj for their valuable feedback.

References

[1] P. Holme and J. Saramäki, "Temporal networks," *Phys. Rep.*, vol. 519, pp. 97–125, 2012.
[2] C. M. Queens and C. J. Albers, "Forecasting traffic flows in road networks: A graphical dynamic model approach," Tech. Rep., The Open University, UK, 2008. [Online]. Available: https://www.casperalbers.nl/files/forecastingISF2008.pdf
[3] M. E. Newman, "The structure and function of complex networks," *SIAM Rev.*, vol. 45, pp. 167–256, 2003.
[4] S. Ramesh, A. Baranawal, and Y. Simmhan, "Granite: A distributed engine for scalable path queries over temporal property graphs," *J. Parallel Distrib. Comput.*, vol. 151, pp. 94–111, 2021.
[5] X. M. Rong, L. Yang, H. D. Chu, and M. Fan, "Effect of delay in diagnosis on transmission of COVID-19," *Math. Biosci. Eng.*, vol. 17, pp. 2725–2740, 2020.
[6] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu, "Path problems in temporal graphs," in *Proc. VLDB Endowment*, vol. 7, pp. 721–732, 2014.
[7] S. Gandhi and Y. Simmhan, "An interval-centric model for distributed computing over temporal graphs," in *Proc. IEEE Int. Conf. Data Eng.*, 2020, pp. 1129–1140.
[8] W. Ligtenberg, Y. Pei, G. H. L. Fletcher, and M. Pechenizkiy, "Tink: A temporal graph analytics library for apache flink," in *Proc. World Wide Web Conf.*, 2018, pp. 71–72.
[9] H. Wu, Y. Huang, J. Cheng, J. Li, and Y. Ke, "Reachability and time-based path queries in temporal graphs," in *Proc. IEEE Int. Conf. Data Eng.*, 2016, pp. 145–156.
[10] A. Zhao, G. Liu, B. Zheng, Y. Zhao, and K. Zheng, "Temporal paths discovery with multiple constraints in attributed dynamic graphs," *World Wide Web*, vol. 23, pp. 313–336, 2020.
[11] J. Enright, K. Meeks, and H. Molter, "Counting temporal paths," in *Proc. Int. Symp. Theor. Aspects Comput. Sci.*, 2023, Art. no. 30.
[12] A. Casteigts, A.-S. Himmel, H. Molter, and P. Zschoche, "Finding temporal paths under waiting time constraints," in *Proc. Int. Symp. Algorithms Comput.*, 2020, Art. no. 30.
[13] I. Tsalouchidou et al., "Temporal betweenness centrality in dynamic graphs," *Int. J. Data Sci. Analytics*, vol. 9, pp. 257–272, 2020.
[14] 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.
[15] J. Byun, S. Woo, and D. Kim, "ChronoGraph: Enabling temporal graph traversals for efficient information diffusion analysis over time," *IEEE Trans. Knowl. Data Eng.*, vol. 32, no. 3, pp. 424–437, Mar. 2020.
[16] C. Rost et al., "Distributed temporal graph analytics with gradoop," *VLDB J.*, vol. 31, pp. 375–401, 2022.

[17] J. Xu, Y. Gao, C. Liu, L. Zhao, and Z. Ding, "Efficient route search on hierarchical dynamic road networks," *Distrib. Parallel Databases*, vol. 33, pp. 227–252, 2015.
[18] S. N. Zulkifli, H. A. Rahim, and W. J. Lau, "Detection of contaminants in water supply: A review on state-of-the-art monitoring technologies and their applications," *Sensors Actuators B, Chem.*, vol. 255, pp. 2657–2689, 2018.
[19] S. Rastogi and D. Bansal, "A review on fake news detection 3T's: Typology, time of detection, taxonomies," *Int. J. Inf. Secur.*, vol. 22, pp. 177–212, 2023.
[20] M. Starnini et al., "Smurf-based anti-money laundering in time-evolving transaction networks," in *Proc. Mach. Learn. Knowl. Discov. Databases*, 2021, pp. 171–186.
[21] A. Baranawal and Y. Simmhan, "Optimizing the interval-centric distributed computing model for temporal graph algorithms," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2022, pp. 541–558.
[22] K. Vora, R. Gupta, and G. Xu, "KickStarter: Fast and accurate computations on streaming graphs via trimmed approximations," in *Proc. ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2017, pp. 237–251.
[23] G. Feng et al., "RisGraph: A real-time streaming system for evolving graphs to support sub-millisecond per-update analysis at millions ops/s," in *Proc. ACM Int. Conf. Manage. Data*, 2021, pp. 513–527.
[24] W. Han et al., "Chronos: A graph engine for temporal graph analysis," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2014, Art. no. 1.
[25] M. Mariappan, J. Che, and K. Vora, "DZiG: Sparsity-aware incremental processing of streaming graphs," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2021, pp. 83–98.
[26] Apache, "Apache giraph," 2020. [Online]. Available: https://giraph.apache.org/
[27] V. Kulkarni, R. Bhoot, A. Baranwal, and Y. Simmhan, "Enhancing interval-centric distributed computing to support incremental graphs," in *Proc. IEEE/ACM Int. Symp. Cluster Cloud Internet Comput. Workshops*, 2023, pp. 322–324.
[28] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 599–613.
[29] N. Technology, "Neo4j home," 2007. [Online]. Available: https://neo4j.com/
[30] J. Team, "Janusgraph," 2017. [Online]. Available: https://janusgraph.org
[31] C. Huan et al., "TeGraph: A novel general-purpose temporal graph computing engine," in *Proc. IEEE Int. Conf. Data Eng.*, 2022, pp. 578–592.
[32] B. Steer, F. Cuadrado, and R. Clegg, "Raphtory: Streaming analysis of distributed temporal graphs," *Future Gener. Comput. Syst.*, vol. 102, pp. 453–464, 2020.
[33] A. Debrouvier, E. Parodi, M. Perazzo, V. Soliani, and A. A. Vaisman, "A model and query language for temporal graph databases," *VLDB J.*, vol. 30, pp. 825–858, 2021.
[34] S. Heidari, Y. Simmhan, R. N. Calheiros, and R. Buyya, "Scalable graph processing frameworks: A taxonomy and open challenges," *ACM Comput. Surv.*, vol. 51, 2018, Art. no. 60.
[35] G. Malewicz et al., "Pregel: A system for large-scale graph processing," in *Proc. ACM Int. Conf. Manage. Data*, 2010, pp. 135–146.
[36] M. Besta, M. Fischer, V. Kalavri, M. Kapralov, and T. Hoefler, "Practice of streaming processing of dynamic graphs: Concepts, models, and systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 6, pp. 1860–1876, Jun. 2023.
[37] M. Mariappan and K. Vora, "GraphBolt: Dependency-driven synchronous processing of streaming graphs," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2019, Art. no. 25.
[38] X. Zhu et al., "LiveGraph: A transactional graph storage system with purely sequential adjacency list scans," in *Proc. VLDB Endowment*, vol. 13, pp. 1020–1034, 2020.
[39] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "STINGER: High performance data structure for streaming graphs," in *Proc. IEEE Conf. High Perform. Extreme Comput.*, 2012, pp. 1–5.
[40] D. Sengupta et al., "GraphIn: An online high performance incremental graph processing framework," in *Proc. Int. Eur. Conf. Parallel Distrib. Comput.*, 2016, pp. 319–333.
[41] M. Kaufmann, P. M. Fischer, N. May, C. Ge, A. K. Goel, and D. Kossmann, "Bi-temporal timeline index: A data structure for processing queries on bi-temporal data," in *Proc. IEEE Int. Conf. Data Eng.*, 2015, pp. 471–482.
[42] P. Liu, A. R. Benson, and M. Charikar, "Sampling methods for counting temporal motifs," in *Proc. ACM Int. Conf. Web Search Data Mining*, 2019, pp. 294–302.
[43] J. Waudby, B. A. Steer, A. Prat-Pérez, and G. Szárnyas, "Supporting dynamic graphs and temporal entity deletions in the LDBC social network benchmark's data generator," in *Proc. ACM Joint Int. Workshop Graph Data Manage. Exp. Syst. Netw. Data Analytics*, 2020, Art. no. 8.
[44] M. Cha, H. Haddadi, F. Benevenuto, and K. Gummadi, "Measuring user influence in twitter: The million follower fallacy," in *Proc. Int. AAAI Conf. Web Social Media*, 2010, pp. 10–17.
[45] S. Huang, A. W.-C. Fu, and R. Liu, "Minimum spanning trees in temporal graphs," in *Proc. ACM Int. Conf. Manage. Data*, 2015, pp. 419–430.
[46] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what cost?," in *Proc. 15th USENIX Conf. Hot Topics Operating Syst.*, 2015, Art. no. 14.

Ruchi Bhoot is currently working toward the master's (Research) degree with the DREAM:Lab, Department of Computational and Data Sciences, Indian Institute of Science (IISc), Bangalore. Her research focuses on distributed systems and algorithms for large scale graph analytics.

Suved Sanjay Ghanmode is currently working toward the master's (Research) degree with the DREAM:Lab, Department of Computational and Data Sciences, Indian Institute of Science (IISc), Bangalore. His research interest include real time scalable software systems and applications, and large scale graphs analytics.

Yogesh Simmhan (Senior Member, IEEE) received the PhD degree in computer science from Indiana University, and was previously a research assistant professor with the University of Southern California (USC), Los Angeles, and a postdoc with Microsoft Research, San Francisco. He is an associate professor with the Department of Computational and Data Sciences and a Swarna Jayanti fellow with the Indian Institute of Science, Bangalore. His research explores scalable software platforms, algorithms and applications on distributed systems. He is the recipient of the IEEE TCSC Award for Excellence in Scalable Computing (Mid Career Researcher), in 2020. He is an associate editor-in-chief of the *Journal of Parallel and Distributed Systems (JPDC)*, an associate editor of *Future Generation Computing System (FGCS)*, and earlier served as an associate editor of IEEE TRANSACTIONS ON CLOUD COMPUTING. He is a distinguished member of ACM, and a distinguished contributor of the IEEE Computer Society.