An Interval-centric Model for Distributed Computing over Temporal Graphs

Swapnil Gandhi, Yogesh Simmhan
Indian Institute of Science, Bangalore
gandhis@IISc.ac.in, simmhan@IISc.ac.in

***Abstract*—Algorithms for temporal property graphs may be time-dependent (TD), navigating the structure and time concurrently, or time-independent (TI), operating separately on different snapshots. Currently, there is *no unified and scalable programming abstraction* to design TI and TD algorithms over large temporal graphs. We propose an *interval-centric computing model (ICM)* for distributed and iterative processing of temporal graphs, where a vertex's time-interval is a unit of data-parallel computation. It introduces a unique *time-warp* operator for temporal partitioning and grouping of messages that hides the complexity of designing temporal algorithms, while avoiding redundancy in user logic calls and messages sent. GRAPHITE is our implementation of ICM over Apache Giraph, and we use it to design 12 TI and TD algorithms from literature. We rigorously evaluate its performance for diverse real-world temporal graphs – as large as $131M$ vertices and $5.5B$ edges, and as long as 219 snapshots. Our comparison with 4 baseline platforms on a 10-node commodity cluster shows that ICM shares compute and messaging across intervals to out-perform them by up to $25\times$, and matches them even in worst-case scenarios. GRAPHITE also exhibits weak-scaling with near-perfect efficiency.**

I. INTRODUCTION

Temporal graphs are an emerging class of property graphs with applications in both traditional domains like transit, financial transaction and social networks, and emerging ones like Internet of Things, knowledge graphs and human connectomes. The structure and attributes of such graphs may change over time [8]. These are represented concisely as *interval graphs* where each *entity* in the graph (vertex, edge, their attributes) has a start and an end time-point indicating their interval of existence. Fig. 1(a) shows an interval graph for a *transit network*, where vertices are transit-stops, directed edges indicate a transit option (e.g., bus, train) between them, an interval on the edge identifies the time-period between which the transit option can be initiated, and an edge attribute identifies the travel cost for that transit. In the example, the lifespan of these vertices are perpetual, $[0, \infty)$, for simplicity. Interval graphs can be *multi-graphs*.

Despite their growing availability, there is limited work on temporal graph primitives, platforms and algorithms. Broadly, temporal graphs algorithms can be *time-independent (TI)* or *time-dependent (TD)* [25]. TI algorithms, also called *snapshot-reducible* [23], can discretize a temporal graph into *snapshots*, one per time-point [7], and operate on each snapshot independently. E.g., Fig. 1(c) shows the transit network decomposed into 8 snapshots, $S1$–$S8$, each indicating the vertices, edges and attributes active at that time-point. Algorithms like PageRank (PR), Breadth First Search (BFS) and Connected Components can be modeled as TI to run on each S_i. Existing vertex-centric computing models (VCM) for non-temporal graphs like Google's Pregel [16], or multi-snapshot approaches like SAMS [25] can be used to design and execute such algorithms on temporal graphs. The latter avoids redundant computation across different snapshots to improve performance.

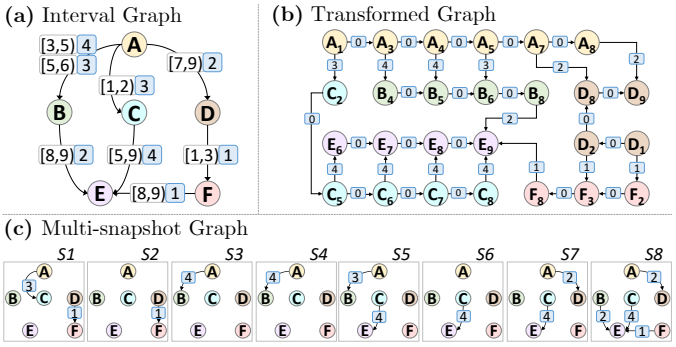


Figure 1: Transit network as a temporal graph.

TD algorithms, also called *extended snapshot-reducible* [23], actively use temporal knowledge to navigate and process the entire graph, or large intervals within them. The need for *time-respecting paths* on a road network is intuitive; it ensures that time-varying factors like traffic density and road-closures are incorporated [26]. *TD centrality measures* are used to estimate information propagation delays in social networks [8]. *Temporal motifs* like feed-forward triangles in transaction networks let us identify monetary routing patterns.

Multi-snapshot approaches applied to TD algorithms can give incorrect results [18], [25], [26]. TD algorithms for earliest/latest arrival time and reachability have been proposed [26]. Other bespoke algorithms [5], [9] and patterns can be extended to similar ones. E.g., the *transformed graph* approach [26] converts an interval graph into an algorithm-specific non-temporal graph. Intervals on vertices and edges map to vertex and edge replicas for time-points in the interval. TD algorithms work on the much larger transformed graph with implicitly-encoded intervals, allowing traversal over time and space. Fig. 1(b) shows a transformed graph for the transit network.

A key gap is the *lack of a unifying abstraction that scales* for constructing both TI and TD algorithms on temporal graphs, which will *ease* algorithm design and *perform well* for diverse, large and long graphs. Platforms and primitives like SAMS [25], Chronos [7] and GraphInc [3] reuse computing or

messaging across snapshots, and some operate in a distributed mode for scalability [3]. But they are limited to TI algorithms. Distributed abstractions for TI and TD algorithms [14], [22] do not scale well due to redundant computing or messaging across time-points and are, arguably, less intuitive. *Ad hoc* patterns like transformed graph are neither intuitive nor scale.

We address this gap through an *interval-centric model of computing (ICM)* for designing TI and TD algorithms over temporal graphs. ICM uses an *interval-vertex* as the data-parallel unit of computing, and executes in a distributed and iterative manner, like popular component-centric abstractions [16], [17]. ICM relies on our novel *time-warp* operator, which automatically partitions a vertex's temporal state, and temporally aligns and groups messages to these states. Warp offers two essential properties. *One,* it implicitly enforces temporal bounds between the time-intervals of vertices, edges and messages for simple and consistent processing by the user logic. *Two,* its maximal partition-size property guarantees that the number of user logic calls and messages generated are minimized. Such *automatic* sharing of compute and messaging within an interval gives ICM its performance and scaling.

TD Example (Temporal SSSP). Say we wish to find a time-respecting path with the *shortest travel cost* [26] in the transit network in Fig. 1(a), from vertex A starting from time 0 to every other vertex. For simplicity, the *travel time* over any edge is assumed to be 1. Multiple solutions can exist for the same source and destination vertices, but which arrive at different points in time and have minimal cost for that point.

This degenerates to running the *single source shortest path (SSSP)* algorithm using VCM on the *transformed graph* in Fig. 1(b). E.g., to reach from A to E, we depart A at time 5 (denoted by A_5), arrive at B at time $5{+}1 = 6$ while incurring a cost (edge attribute) of 3 units, and depart B at time 8 to reach E at time $8+1 = 9$, for a total travel cost of $3+2 = 5$ units. Another solution is from $A_1 \rightarrow C_2 \rightarrow C_5 \rightarrow E_6$ that costs $3+4 = 7$ units, but is valid for the earlier arrival time of 6 at E. Finding the shortest paths from the source to all destination vertices at all valid arrival times takes 21 *vertex visits* and 27 *edge traversals* – the compute and messaging cost.

Our *ICM design* for temporal SSSP, operates on the interval graph in Fig 1(a), navigates across both vertices and edges, by traversing valid overlapping time-intervals, with just 7 *"interval vertex" visits* and 6 *edge traversals*. While we discuss the design for SSSP in Sec. IV, intuitively, we replicate the vertex into the minimal necessary sub-intervals, on-demand, based on the different intervals present in the messages that arrive and the out-edges. This makes designing temporal SSSP (among many other algorithms) similar to its non-temporal VCM variant, while avoiding all redundant compute and messaging.

We *cannot* solve this algorithm on a *multi-snapshot graph* as the partial paths over time is lost across snapshots. ■

Specifically, we make the following contributions:

1) We define the temporal graph *data model* in Sec. III. We introduce and illustrate the novel *ICM programming abstraction* and *time-warp operator* to design distributed TI and TD algorithms on temporal graphs, in Sec. IV.
2) We briefly discuss the use of ICM to *intuitively design* 12 *TI and TD algorithms* from literature in Sec. V.
3) We describe the GRAPHITE *distributed platform*, which implements ICM, in Sec. VI. In Sec. VII, we offer detailed *experiments* to evaluate the performance and scalability of ICM for these 12 algorithms on 6 diverse real-world graphs, as large as $131M$ vertices and $5.5B$ edges, and as long as 219 snapshots. We compare ICM to 4 baselines which we implement from literature.

We offer a review of related work in Sec. II, and present our conclusions and future work in Sec. VIII.

II. Related Work

A. Distributed Graph Processing Primitives

Graph applications tend to be irregular and computationally complex. Graph processing primitives offer a structure to more-easily design and execute graph algorithms. Distributed abstractions such as *Pregel* [16] and *GraphLab* [15] adopt a data-parallel, iterative execution model to horizontally scale across machines, using multiple CPU cores and cumulative memory. Parallelism is exposed at the granularity of *graph components*, and hence called component-centric computing models [17], with VCM the most common [20], [28]. But, existing abstractions focus on large non-temporal graphs. ICM is in the spirit of such intuitive component-centric models, but introduces *time-intervals* and *time-warp* as first-class entities to ease programming and enhance scaling for temporal graphs.

B. Time Independent Temporal Graph Processing

Time Independent (TI) algorithms can model and process temporal graphs as a *series of snapshots*. This allows existing primitives, platforms and algorithms for graph processing [20], [28] to be applied independently to each snapshot at a distinct time-point. However, processing snapshots independently causes redundant computation and messaging, limiting scalability. Systems and abstractions [3], [7], [13] have tried to address this inefficiency.

In particular, *SAMS* [25] presents rewriting rules for automatic co-scheduling of common steps during multi-snapshot analysis, similar to SIMD processing. This addresses some performance limitations we ourselves observe in our experiments when operating over a large number of snapshots. *Chronos* [7] offers an efficient in-memory layout for vertices that span multiple snapshots to leverage time-locality. It couples this with a vertex-centric engine for batched execution over multiple snapshots. Concurrent processing of the vertex states from across snapshots enhance cache hits. Unlike us, the user logic execution for a vertex is not shared across snapshots but only reduces (in-memory) communication when sending common messages that span contiguous snapshots.

GraphInc [3] incrementally processes real-time graph updates using Giraph's VCM. It reuses the prior snapshot's state to rapidly compute an analytic for the new snapshot. It also memoizes incoming messages to avoid redundant vertex-compute if a message was seen earlier. However, updates to a snapshot must complete before moving to the next. *Tegra* [10]

relaxes this by allowing streaming updates to be folded into an ongoing analytic using a pause-shift-resume model. This reduces the time to apply and process recent updates. But both these platforms are designed for TI analytics. States from prior snapshots are used to reduce the recompute time for a later snapshot rather than support time-dependent algorithms. We support both TI and TD algorithms, but focus on fully evolved graphs with valid time [11] rather than streaming ones.

C. Time Dependent Temporal Graph Processing

Time Dependent (TD) algorithms need the state of the graph at a previous time-point to execute the current one. Given the limited platforms and abstractions for designing such algorithms, custom techniques for individual analytics have been proposed [5], [9], [12], [26]. These are not generalizable primitives. Among bespoke algorithms, the *transformed graph approach* [27] can be adapted for a large class of TD algorithms, albeit with algorithm-specific transformations. It can also be extended for distributed execution using VCM. But, as we demonstrate (Sec. VII), it bloats the graph size and suffers from poor scalability.

Like us, *Tink* [14] supports distributed processing of interval graphs, and offers a library of TD algorithms over Apache Flink. Like Chronos, it avoids sending redundant messages that span an interval but does not share computation across an interval due to time-point based primitives. As we illustrate, this limits scalability. ICM's warp operator maximizes sharing of calls to compute and messages across intervals.

Our prior work, *GoFFish-TS* [22] proposes primitives for TD algorithms using a multi-snapshot approach. Here, the state from a prior snapshot can be explicitly passed as a message to the next snapshot by the user logic. Within a snapshot, it uses a subgraph-centric model of execution. It too does not share computation, is limited to processing one snapshot at a time, and states have to be explicitly passed over time.

None of the reviewed literature provide results for temporal graphs as large and diverse as we report here, nor examine the wide variety of TI and TD algorithms that we consider.

D. Models and Algebra

Temporal data models and querying primitives from relational databases [11] are only gradually translating to modeling temporal features in graphs, and on graph querying languages [2]. *Moffit and Stoyanovich* [18] propose a *Temporal Graph Algebra (TGA)*, which introduces principled temporal generalizations based on temporal relational algebra for conventional graph operators. Others use indexing for temporal reachability queries in strongly connected components at various time points [21]. ICM is imperative and can be used to design general purpose temporal graph analytics, and is complementary to these.

III. Temporal Graph Model

Our distributed primitives focus on composing analytics over historic graphs, with dynamism in their structure and attributes, but which are *fully evolved* and ready for processing. Here, we define the *temporal graph data model* that our proposed abstraction supports; such formalism avoids ambiguity.

Time Domain. WLOG, we assume a linearly ordered discrete time domain Ω whose range is the set of non-negative whole numbers. Each instant in time is a *time-point*, and their linear ordering means that $t_i < t_{i+1} \implies t_i$ *happened before* t_{i+1}. One *time unit* is the atomic increment of time, and corresponds to some user-defined wall-clock time, such as p seconds.

Time-interval. Entities of a temporal graph have an associated *time-interval*. Given $t_{start}, t_{end} \in \Omega$, then $\tau = [t_{start}, t_{end})$ indicates a *time-interval* that starts from and includes t_{start}, and extends to but excludes t_{end}. The time-points that are part of a time-interval $\tau = [t_{start}, t_{end})$ is the set $\{t \mid t \in \Omega$ *and* $t_{start} \leq t < t_{end}\}$.

Interval Relations Boolean relations between intervals follow *Allen's* conventions [1]. The symbol $\sqsubset$ represents *during*, $\sqsubseteq$ represents *during or equals*, $\sqcap$ represents *intersects*, $=$ represents *equals*, and $\dashv$ is the *meets* relation. $\cap$ returns the intersecting interval between two intervals.

Definition 1. *(Temporal Graph) A temporal graph is a directed multi-graph* $\mathcal{G} = (V, E, L, A_V, A_E)$*, where:*

- V is a finite set of *vertices*, where each vertex $v \in V$ is a pair $\langle vid, \tau \rangle$. $vid \in \mathbb{V}$ is a unique and opaque internal identifier and $\tau = [t_s, t_e)$ is the time-interval for which the vertex exists (also called the *lifespan* of the vertex).
- E is a finite set of *edges*, where each directed edge $e = \langle eid, vid_i, vid_j, \tau \rangle \in E$ is a 4-tuple identified by its unique identifier $eid \in \mathbb{E}$, and the edge exists for the interval $\tau = [t_s, t_e)$ (*lifespan* of the edge). The edge connects the source vertex vid_i with the sink vertex vid_j, with $vid_i, vid_j \in \mathbb{V}$.
- L is a finite set of *property* (also called *attribute*) *labels* that can be associated with either vertices or edges.
- A_V (or A_E) is a finite set of vertex (or edge) *property values*, where each 4-tuple $\langle vid, l, val, \tau_a \rangle \in A_V$ represents the value val associated with a label $l \in L$ of the vertex (or edge) identified by vid, for the interval τ_a. A label may have distinct values for non-overlapping intervals during the lifespan of its vertex (or edge). Formally, for all vertex property values [1] $\langle vid, l, val, \tau_a \rangle \in A_V$, there does not exist any $\langle vid, l, val', \tau'_a \rangle \in A_V$ such that $\tau_a \sqcap \tau'_a$ and $val \neq val'$.

We define several *constraints* to guarantee the soundness of the temporal graph.

Constraint 1 (Unique vertices and edges)**.** *Any vertex (or edge) uniquely identified by its* vid *(or* eid*) exists at most once, and only for a contiguous time-interval, and once it ceases to exist, a vertex (or edge) with the same* vid *(or* eid*) can never re-occur at a later time-point. Formally, for all vertices* [1] $\langle vid, \tau \rangle \in V$*, there does not exist another vertex* $\langle vid', \tau' \rangle \in V$ *such that* $vid = vid'$ *and* $\tau \neq \tau'$*.*

Constraint 2 (Referential integrity of edges)**.** *For an edge to exist, the time-intervals associated with its source and its sink vertices must contain the edge's time-interval. Formally, for all*

[1] This can similarly be extended for edges, but is omitted for brevity.

edges $\langle eid, vid_i, vid_j, \tau\rangle \in E$*, there exist vertices* $\langle vid_i, \tau'\rangle \in V$ *and* $\langle vid_j, \tau''\rangle \in V$ *such that* $\tau \sqsubseteq \tau'$ *and* $\tau \sqsubseteq \tau''$.

Constraint 3 (Referential integrity of properties)**.** *For a vertex property value to exist, the interval of the vertex must contain the interval of the vertex property. Formally, for all vertex properties* [1] $\langle vid, l, val, \tau_a\rangle \in A_V$*, there exists a vertex* $\langle vid, \tau\rangle \in V$ *such that* $\tau_a \sqsubseteq \tau$.

Constraint 1 prevents the graph from having multiple copies of a vertex or edge at the same time-point. Forcing a contiguous lifespan simplifies the reasoning about the behavior of our computation model, though this may be trivially relaxed. Users may encode their custom vertex or edge name as a property to indicate logical equivalence of reappearing vertices or edges at disconnected time-intervals. Constraints 2 and 3 prevent an invalid graph by ensuring that edges connecting vertices, or properties for vertex or edges, are concurrent.

IV. Thinking like an Interval

In this section, we describe our novel and intuitive interval-centric distributed programming abstraction as a *unified model* for designing TI and TD algorithms. We also propose an innovative *time-warp* operator that performs efficient temporal alignment and grouping of messages with vertex states. This *eases the temporal reasoning* required by the user logic, and *avoids redundant execution* of user logic and messaging within an interval to provide key performance benefits.

A. Interval-centric Computing Model (ICM)

ICM lets users define their logic from the perspective of *a single vertex, for a particular time-interval*, and this logic is executed on every *active* vertex and its *active* interval(s) (defined in Sec. IV-A2) in a *data-parallel* manner. We use *Bulk Synchronous Parallel (BSP)* execution [16], which alternates a *computation phase*, where the user logic executes, with a *communication phase*, where messages are bulk-transfered between vertices at a global barrier. These continue for several iterations till the application converges. Fig. 2 illustrates this.

The computation phase has two steps: `compute` and `scatter`, which are user-provided logic. *Compute* operates on the vertex, its prior states and the incoming messages, *in the context of a particular interval*, and can update the vertex's current state for that interval. Then, *scatter* operates on the out-edges for a vertex, and plays two roles. It decides if the updated state should be sent as a message to the adjacent vertex the edge connects to, and if so, provides a transform function on the state to create the message and its interval.

Once the *compute* and *scatter* logic execute for all active vertices and their active intervals, the communication phase delivers messages to the destination vertices. The current iteration (*superstep*) is done, and the next iteration can start.

1) Dynamically Partitioned Vertex States: Vertices in ICM inherit *static information* from the temporal graph $\mathcal{G}$, and also maintain *dynamic states* for the user logic. For a vertex vid, the former includes the interval τ of the vertex, its out-edges and their lifespans $\langle eid_j, vid, vid_j, \tau_j\rangle$, and the properties of vertex intervals, $\langle vid, l, val, \tau_a\rangle$, and similarly edge intervals. The dynamic state for a vertex consists of discrete states for a set of *partitioned intervals* that cover the vertex's lifespan. *Compute* and *scatter* can access these states, and *compute* can update them in the context of these partitioned intervals. A state may hold any user-defined content. Formally, if $\tau = [t_s, t_e)$ is the static lifespan of a temporal vertex, then the state for the vertex, partitioned into n intervals, is: $S(\tau) = \{\langle \tau_i, s_i\rangle \mid i \in [1,n] \wedge \tau_i = [t_s^i, t_e^i) \wedge t_s^1 = t_s \wedge t_e^n = t_e \wedge \forall j \in [1,n), t_e^j = t_s^{j+1}\}$, i.e., the partitioned intervals cover the entire lifespan of the vertex, and no two partitioned intervals overlap.

Importantly, states are *dynamically repartitioned* when the state for a sub-interval in the partitioned interval's state is updated. So if we have $\langle \tau_i, s_i\rangle$ as a partitioned state for a vertex, and *compute* updates the state for its initial sub-interval τ_j, where $t_s^j = t_s^i$ and $t_e^j < t_e^i$, with a new value s_j, then we automatically replace the state s_i with two states $\langle [t_s^i, t_e^j), s_j\rangle$ and $\langle [t_e^j, t_e^i), s_i\rangle$. Even without a state update, it is valid to split a partitioned interval into sub-intervals while replicating their state values, i.e., $\{\langle [t_s, t_e), s\rangle\} \equiv \{\langle [t_s, t'), s\rangle, \langle [t', t_e), s\rangle\}$.

In the first iteration of ICM, each vertex starts with a single *initialized state* for its entire lifespan [2]. As the iterations progress and states for sub-intervals for the vertex are updated by the *compute* logic, the number of partitions can grow. In the worst case, we will have as many partitions as the number of time-points in the vertex's lifespan.

2) Active Vertices and Intervals: Compute only executes on active vertices, and on active intervals within them. Vertices that have received a message from the previous iteration are called *active vertices*, and the sub-intervals within them which overlap with the interval of at least one message to that vertex are *active intervals*. The *time-warp operator* (Sec. IV-B) finds the intersections between the partitioned vertex state and the messages it receives, and *compute* is invoked on each intersecting vertex sub-interval, with that state and those messages. Each time-point within the active sub-intervals of a vertex will be part of *exactly one* compute method call.

Unlike Pregel, all our vertices implicitly *vote to halt* and deactivate after each superstep, and get reactivated only if they receive a message in the next or a future iteration. This reflects the design of most VCM algorithms [20], [28]. ICM stops when no vertices are activated by messages in an iteration.

3) Compute and Scatter Logic: Say, for the temporal vertex $v = \langle vid, \tau\rangle$, $\tau_i \sqsubseteq \tau$ is an active sub-interval. The signature of the user-defined interval-centric compute logic is given by:

$$\texttt{compute(vid, } \langle \tau_i, s_i\rangle \texttt{, M[ ])} \rightarrow S(\tau_i)$$

where $\langle \tau_i, s_i\rangle$ is a partitioned state for the vertex inherited from the previous superstep, and $M[\,]$ is the set of messages received by this vertex from the previous superstep whose intervals τ_m are such that $\tau_i \sqsubseteq \tau_m$. The user's logic can access the vertex's and its edges' static attributes (E, A_V and A_E)

[2] In fact, the state of a vertex interval τ_j is pre-partitioned based on all sub-intervals τ_a of its static properties l. So our computing unit is an *interval property vertex*. However, since properties are optional and to keep the discussion concise, we consider states as partitioned only on the vertex interval and not its property intervals.

for any time-interval. These, along with the prior state s_i and the received messages $M[\,]$ for this interval τ_i, are processed to return optionally updated partitioned states for this interval $S(\tau_i) = \{\langle \tau_j, s_j \rangle \mid \tau_j \sqsubseteq \tau_i\}$. *Compute* can be called data-parallelly on the active intervals of the vertex, and the exact invocation is decided by the *warp operator*, discussed next. Since time-points in each active interval are part of *exactly one* compute method execution, these updates can happen on the partitioned states concurrently without interference.

The signature for the user's transformation and message passing logic for an active vertex is:

$$\texttt{scatter}(\texttt{eid},\ \langle \tau'_k, s_k \rangle) \ \rightarrow\ \{\langle \tau_m,\ \texttt{M} \rangle\}$$

Scatter is called for those out-edges *eid* of the active vertex with a time-interval τ_e such that $\tau_k \sqsubseteq \tau_e$. Here, $\langle \tau_k, s_k \rangle \in \bigcup S(\tau_i)$, for all partitioned state intervals τ_i that were updated by *compute*, and $\tau'_k = \tau_k \cap \tau_e$. *Scatter* is called once for each such $\langle \tau'_k, s_k \rangle$. *Scatter* returns one or more message payload(s) M with their associated time-interval τ_m that is to be sent to the sink vertex for that edge. *Scatter* may be called data-parallelly on the partitioned intervals of the out-edges, for each active vertex. Each time-point in an edge's lifespan is part of *no more than one* scatter execution in an iteration, and the exact number of *scatter* calls is decided by *warp*. *Scatter* can access the edge's static attributes (E, A_E) for any interval.

Typically, users implement *scatter* with two concise functions f_t and f_m that perform transformations to give $\tau_m = f_t(\tau_k)$ and $M = f_m(s_k)$. But several variations are possible to balance brevity and flexibility. If the method returns an output message $M = \varnothing$, then no message is sent for this edge and for this state interval. *Scatter* may omit the time-interval from the output, in which case the input state interval is inherited, i.e., $\tau_m = \tau'_k$. If *scatter* itself is not provided, then we send a single message with $\tau_m = \tau'_k$ and $M = s_k$.

Once messages for an active vertex are received in a superstep after the barrier, *warp* decides their grouping and executes *compute* on them for the partitioned vertex states. Similarly, once the *compute* step for a vertex completes, *warp* decides for each of its out-edges, the mapping from the updated partitioned state to the sub-interval of the edge on which to invoke *scatter*. This is discussed in Sec. IV-B.

Temporal SSSP Example. The temporal *single source shortest path (SSSP)* [26] finds a time-respecting path with the shortest travel cost between a single source vertex and every other vertex in a temporal graph. Multiple solutions can exist for the same source to each destination vertex, but which arrive at different points in time; each path will have the least cost for that interval of arrival.

The Java pseudo-code for temporal SSSP using ICM is shown in Alg. 1, and illustrated in Fig. 2 for the interval graph from Fig. 1(a). The partitioned (dynamic) states for a vertex maintain the current known lowest cost from the source to that vertex, for different intervals of arrival. The *init* method is called only before superstep 1, and initializes a vertex's state to ∞ for its entire lifespan. *Compute* is called on all vertices in superstep 1, with no messages and for the entire vertex lifespan. Only the source vertex updates its state to a travel

```
1  void init(Vertex v) {
2     v.setState(v.interval, ∞);
3  }
4  void compute(Vertex v, Interval t, int vstate,
       Message[ ] msgs) {
5     if(getSuperstep() == 1 && isSource(v)) {
6        v.setState(t, 0); return;
7     }
8     minVal = ∞;
9     for(Message m : msgs)
10       minVal  = min(m.value, minVal);
11    if(minVal < vstate) v.setState(t, minVal);
12 }
13 Message scatter(Edge e, Interval t, int vstate){
14    int travelTime = e.getProp("travel-time");
15    int travelCost = e.getProp("travel-cost");
16    return new Message(e, new Interval(t.start +
          travelTime, ∞), vstate + travelCost);
17 }
```

Algorithm 1: Temporal SSSP using ICM

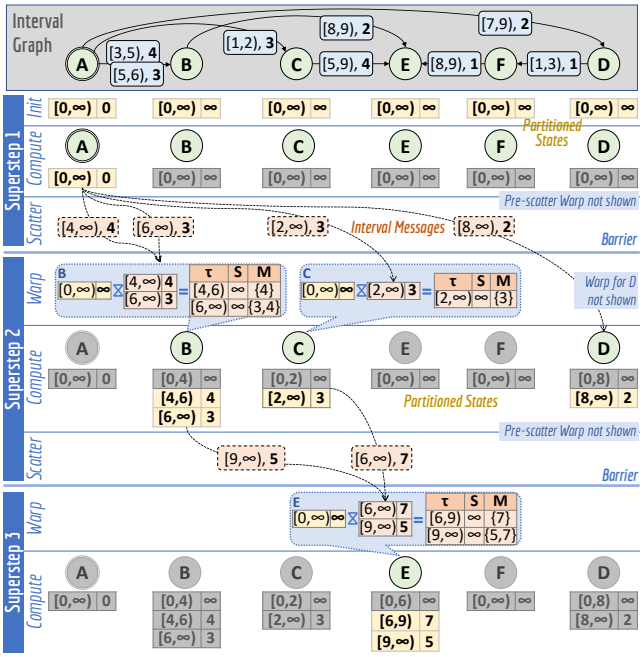


Figure 2: SSSP execution using ICM for the temporal graph from Fig. 1(a). A is the source. Travel time on an edge is 1.

cost of 0 for its lifespan. Since *compute* has changed the state for the source vertex for its entire lifespan, *scatter* is called once for each overlapping interval of its out-edges having a distinct property. Each edge sends a message to its sink vertex with the travel cost to the current vertex (i.e., its updated state; 0 for the source), *plus* the static property *'travel-cost'* on that edge to the sink. The start time of this message is set to the later of the starting interval of the updated state (cost) or the edge's lifespan, *plus* the *'travel-time'* property on the edge. So the *cost message* received at the sink vertex is valid from that arrival time and beyond. This logic lets both the travel time and cost of the edge to be dynamic. This ends superstep 1.

E.g., in Fig. 2, A's *scatter* is called twice for the edge to B, for the two interval properties $\langle [3,5], 4 \rangle$ and $\langle [5,6), 3 \rangle$. It sends a message with *travel cost* $(0+4)$, valid for the interval

$[3+1,\infty)$ for the first, and $\langle[5+1,\infty), 0+3\rangle$ for the other.

In future supersteps, a vertex may receive messages from its neighbor(s) for one or more of its sub-intervals, with the cost for that interval of arrival. This becomes an *active vertex interval*. After *warp*, *compute* checks if the current cost (partitioned state) for that vertex interval is reduced by any message sent to that interval, and if so, updates it. Any state update causes *scatter* to be called on all edge properties overlapping this interval, and the new candidate lowest cost is propagated to its neighbors with an updated arrival time.

E.g., in superstep 2, *compute* is called twice on vertex B after *warp*, once for the interval $[4,6)$ with message value $\{4\}$ and once for $[6,\infty)$ with messages $\{3,4\}$. The prior states for both these intervals of B is ∞, and *compute* updates these to 4 and 3, respectively. Note that B's state has been dynamically repartitioned into 3 sub-intervals. *Scatter* is called on the edge B to C for its property $\langle[8,9), 2\rangle$ which overlaps with state $\langle[6,\infty), 3\rangle$, causing message $\langle[8+1,\infty), 3+2\rangle$ to be sent.

The algorithm terminates when all vertices and their arrival time intervals have stabilized to the least cost from the source, if feasible – i.e., no states change – and no messages are in flight. E.g., at the final state, vertex F cannot be reached from A; C and D can be reached during 1 contiguous interval each with costs 3 and 2; while B and E can be reached during 2 different intervals, with a different lowest cost for each. ■

B. Time-warp

Adding time-intervals to *compute* and *scatter* is a novel temporal extension to Pregel [16] or GAS [15] models. However, the critical benefit of ICM comes from a unique data transformation we propose: *time-warp* (or *warp*). It is a powerful construct that lets the user logic operate *consistently* over temporal messages and partitioned vertex states, and intuitively design temporal graph algorithms as if for a non-temporal graph. It is analogous to the *shuffle* in MapReduce which transforms the simple Map and Reduce functions into powerful primitives. Also, *warp guarantees automatic sharing of compute and messaging across adjacent time-points*, minimizing the number of calls to *compute* and the *messages sent*. This enhances the performance of ICM algorithms for temporal graphs having non-trivial lifespans on their entities.

The warp step happens between: (1) the message receipt at the start of a superstep and the compute step, and (2) the compute and the scatter steps. It performs temporal alignment, re-partitioning and grouping that decides the number of calls to *compute* and *scatter*, and their parameters.

The warp operator takes two sets: an *outer set* containing partitioned intervals and values, and an *inner set* with intervals and values. It returns a single partitioned set of triples, each containing an interval, a value from the outer set, and a set of values from the inner set. Intuitively, before the *compute step* for an active vertex, warp groups the input messages for a vertex and their intervals (inner set) that overlap with the partitioned states for the vertex (outer set), to form the fewest number of (re)partitioned states that are each a temporal subset of the group of messages. This may repartition the vertex

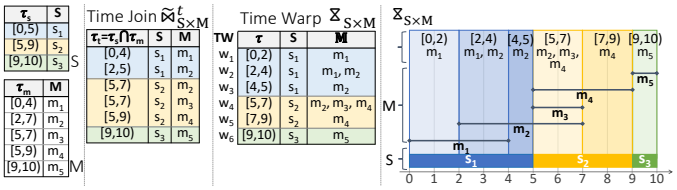


Figure 3: Time-warp operating on the partitioned states and input messages for an active vertex.

states, and duplicate a message to multiple groups that are each a partitioned vertex state. Each partitioned state and its grouped messages forms a single triple in the output from warp, and causes a single invocation to *compute* for that active vertex interval with these as input parameters.

This ensures two things: (1) the user's *compute* logic can leverage this *exact alignment* between the message intervals and the partitioned state in its invocation, and (2) the *compute* itself is called *as few a times* as possible, to avoid redundant computation and hence improve performance.

Similarly, before the *scatter step* for an active vertex, the partitioned *updated states* from the compute step (outer set) is warped with the temporal out-edges for that vertex (outer set) so that each edge is invoked for a sub-interval which has one (re)partitioned state-change that fully overlaps with that interval and also with the edge's lifespan. This too guarantees that the *scatter* for an edge sub-interval receives a state update applicable for that whole interval, and calls to *scatter* (and hence, message generation) is minimized.

Intuitively, longer the intervals of items in the inner and outer sets and greater their overlap, fewer the tuples in the output set and lesser the calls to the user logic.

Detailed Warp Example. Fig. 3 illustrates warp for the 3 partitioned states S of an active vertex that receives 5 messages M. A *time-join* ($\widetilde{\bowtie}_{S\times M}$) operation [24] over these sets finds the intersections between the intervals of a state and a message. E.g., m_2 with an interval of $[2,7)$ overlaps with the intervals of s_1 and s_2, and results in $\langle[2,5), s_1, m_2\rangle$ and $\langle[5,7), s_2, m_2\rangle$. Warp is a form of self-join over the time-join, with temporal semantics that detect the boundaries of the intersections in these time-joins (e.g., $0,2,4,5,7,9,10$). For intervals formed from adjacent pairs of boundaries (e.g., $[0,2), [2,4)$), it groups messages in that interval with the state of the vertex (e.g., $\langle[0,2), s_1, m_1\rangle$, $\langle[2,4), s_1, \{m_1,m_2\}\rangle$). The output tuples are temporally partitioned. Each tuple forms a call to *compute*, with the time-aligned state and the message group passed to it, thus simplifying the user logic. The warp of the updated states after *compute* with the out-edges is similar, and triggers the execution of *scatter*. In practice, a time-join suffices before *scatter* if the edges' properties are time-invariant. ■

Formally, *time-warp* ($\boxtimes_{S\times M}$) operates on two sets S (outer set) and M (inner set) both having 2-tuples with a time-interval and a value. The outer set must be temporally partitioned. The *time-join* ($\widetilde{\bowtie}_{S\times M}$) operator [24] on the two sets is defined as:

$$S = \{\langle\tau_s, s\rangle\}$$
$$M = \{\langle\tau_m, m\rangle\}$$

$$\widetilde{\bowtie}^{t}_{S\times M} = \{\langle \tau_t, s_t, m_t\rangle \mid \langle \tau_s, s_t\rangle \in S \ \wedge \ \langle \tau_m, m_t\rangle \in M \ \wedge \ \tau_s \sqcap \tau_m \ \wedge \ \tau_t = \tau_s \cap \tau_m\}$$

It is a form of natural join over the intervals that identifies sub-intervals of the inner set which are present in the outer, and returns triples in the output set which have the common sub-intervals from both sets and their associated values. Using this, we propose and define the *time-warp* operator as:

$$\begin{aligned}\text{⧖}_{S\times M} = \{\langle \tau_{pq}, s_r, \mathbb{M}_r\rangle \mid \\ \big(\ \forall\, p \in \widetilde{\bowtie}^{p}_{S\times M},\ q \in \widetilde{\bowtie}^{q}_{S\times M} \mid s_p = s_q, \\ \tau_{pq} = [t_s, t_e) \mid t_s \in \{t^p_s, t^p_e\} \ \wedge \ t_e \in \{t^q_s, t^q_e\} \ \big) \wedge \\ \big(\ \forall\, r \in \widetilde{\bowtie}^{r}_{S\times M} \mid s_r = s_p = s_q, \\ (\tau_{pq} \not\sqcap \tau_r \ \vee \ \tau_{pq} \sqsubseteq \tau_r) \wedge \\ \tau_{pq} \sqsubseteq \tau_r \implies m_r \in \mathbb{M}_r \ \big) \wedge \\ \mathbb{M}_r \neq \varnothing\}\end{aligned}$$

The start and end times of each sub-interval in the time-join forms the time-point boundaries at which the tuples from the two sets temporally overlap. The *candidate time-intervals* (τ_{pq}) for the warp are formed from the cross-product of each pair of boundary points of an interval, $\{t^p_s, t^p_e\} \times \{t^q_s, t^q_e\}$, for a given common value $s_p = s_q$ from the outer set S. Implicitly, only valid intervals are considered, i.e., the start time-point of the interval must be smaller than the end time-point.

Each candidate interval must either be fully contained within or fully disjoint with every interval τ_r of the time-join which has the same value as in the outer set. This ensures that the warp's interval does not cross a boundary time-point but rather is *exactly aligned* with them. For each candidate interval that is contained within a time-join interval, we group the values m_r from the inner set into the output $\mathbb{M}_r$; we only include those output triples with a non-empty set of inner values.

The warp operator guarantees the following properties:

1) **Valid Inclusion.** Every value-pair from across the two sets, which both exist at an overlapping time-point, is included for that time-point in an output triple. Formally, for all tuples $\langle \tau_j, s_j\rangle \in S$ and $\langle \tau_k, m_k\rangle \in M$, if $\tau_j \sqcap \tau_k$, then for all time-points $t \in \tau_j \cap \tau_k$, there exists an output tuple $\langle \tau, s_j, \mathbb{M}\rangle \in \text{⧖}_{S\times M}$ such that $t \in \tau$ and $m_k \in \mathbb{M}$.
2) **No Invalid Inclusions.** No value from the two sets are included in the output for a time-point unless they both respectively exist in their sets for that time-point. Formally, for any output tuple $\langle \tau, s_j, \mathbb{M}\rangle \in \text{⧖}_{S\times M}$, there must exist tuples $\langle \tau_j, s_j\rangle \in S$ and $\langle \tau_k, m_k\rangle \in M$ such that $m_k \in \mathbb{M}$, $\tau \sqsubseteq \tau_j$ and $\tau \sqsubseteq \tau_k$.
3) **No Duplication.** A value at a time-point from the outer set appears in no more than one output triple for that time-point. Formally, there are no two output tuples $\langle \tau_j, s_j, \mathbb{M}_j\rangle, \langle \tau_k, s_k, \mathbb{M}_k\rangle \in \text{⧖}_{S\times M}$ such that $\tau_j \sqcap \tau_k$ and $s_j = s_k$.
4) **Maximal.** The number of output triples are temporally grouped into as few as possible. Formally, there are no two output tuples $\langle \tau_j, s_j, \mathbb{M}_j\rangle, \langle \tau_k, s_k, \mathbb{M}_k\rangle \in \text{⧖}_{S\times M}$ with $s_j = s_k$, $\mathbb{M}_j = \mathbb{M}_k$, and either overlapping intervals $\tau_j \sqcap \tau_k$ or adjacent intervals $\tau_j \dashv \tau_k$.

Here, # 1–3 ensure correctness of the grouping, while # 4 limits invocation of the user logic to the *minimally possible*.

Temporal SSSP Example. Continuing the earlier example, warp automatically enforces temporal constraints in the calls to *compute* and *scatter*. Before the compute step, warp ensures that the update messages are aligned and grouped with the (re)partitioned vertex states. So *compute* can rely on the costs in the messages being applicable to the entire sub-interval the logic is called for, and can simply compare the state's cost with the message's cost (lines 9–11 of Alg. 1).

E.g., when superstep 3 starts in Fig. 2, E calls warp on its prior state $\langle [0,\infty), \infty\rangle$, and the messages $\langle [9,\infty), 5\rangle$ from B and $\langle [6,\infty), 7\rangle$ from C. Warp returns the tuples $\langle [6,9), \infty, \{7\}\rangle$ and $\langle [9,\infty), \infty, \{5,7\}\rangle$ that each call *compute*. *Compute* uses a simple *min* logic to change the travel cost (state) to 7 for the interval $[6,9)$, and to 5 for $[9,\infty)$. We also show the pre-compute warp in superstep 2 for B and C.

So the user logic avoids comparing the temporal bounds of each message with each state, and explicitly repartitioning the state before updating its cost. *This makes the logic near-identical to the non-temporal VCM algorithm.* Also, the maximal property of warp ensures that *compute* is called only once for all messages that temporally intersect with a partitioned state, for that interval. *This avoids duplication of calls.* ■

V. Temporal Graph Algorithms

Programming primitives like ICM help rapidly design different temporal graph algorithms from existing ones. Diverse TD path algorithms, such as *Earliest Arrival Time (EAT)* [26], *Fastest Arrival Time (FAST)* [26], *Latest Departure time (LD)* [26], *Reachability (RH)* [27] and *Time-Minimum Spanning Tree (TMST)* [9], can be solved with minimal changes to the temporal SSSP algorithm we introduced earlier.

To find the TMST from a given source, we add the *parent vertex ID* to the state and the message value (lines 12 and 17) in Alg. 1, in addition to replacing *travel cost* with *arrival time*, to rebuild the tree [9]. Just replacing the *travel cost* in the message with the *vertex departure time* instead (line 15) computes EAT from a single source to all destinations. Here, we are only interested in the earliest time at which we can reach a vertex, and not in subsequent intervals of arrival. For RH, we replace the *travel-cost* in the message with a *flag* to help test if a vertex-pair is reachable. The FASTest path reduces the vertex waiting time and the travel time. Its message will include the time at which the *journey started* at the source for each path, and the state maintains the *arrival time* at a vertex interval. *Compute* uses this to minimize the travel duration, and propagates it through *scatter*. LD lets one depart late and reach within a bound. Unlike SSSP, it reverse-traverses from sink to source, in space and time, by setting its message interval to $[-\infty, t.end - travelTime)$. Warp ensures that temporal bounds are not violated.

We also design two TD clustering algorithms: *Local Clustering Coefficient (LCC)* [8] and *Triangle Counting (TC)* [12]. In LCC, each interval vertex quantifies how close its neighbors are to forming a clique. Each vertex messages its neighbors, which then message their neighbors to check the ones adjacent to the initial vertex. This edge-count is sent back to the initial

vertex to compute its LCC. In TC, each vertex messages its two-hop neighbors to see if they are adjacent to the initial vertex. Neighbors for LCC and TC have to be time-respecting.

Besides these, we also formulate ICM variants for 4 TI algorithms: *BFS* [16], *WCC* [28], *Strongly Connected Component (SCC)* [28] and *PageRank (PR)* [16]. The VCM logic for these algorithms can be reused for *compute* since ICM by default assigns appropriate intervals to the states and messages.

The ability to design a variety of TI and TD algorithms attests to the expressivity offered by the unified ICM primitives.

VI. The Graphite Platform

Graphite [3] is our implementation of the interval-compute model, built as a layer on top of *Apache Giraph*, a popular Java-based open-source distributed graph processing platform that offers VCM primitives. Users provide their ICM *compute* and *scatter* logic to Graphite in Java. Our runtime logic, such as *warp*, invocation of the interval-centric user logic, and message handling, are part of the vertex-centric `compute` method exposed by Giraph. We also leverage its Master-Compute pattern for coordination.

Time Warp. We implement warp using a merge-sort aggregation algorithm [19]. It incrementally computes a larger aggregate by merging two smaller aggregates, with the final aggregate at the root. For m input messages, its time-complexity is $\mathcal{O}(m \log m)$ and space-complexity is $\mathcal{O}(m)$. Typically, $m = \mathcal{O}(d \cdot t)$ where d is the in-degree and t is the lifespan of the vertex. For algorithms like TC, the size of each message can itself be d, increasing the space complexity.

Interval Messages. Messages in Graphite includes an interval, with start and end time-points. Given the billions of messages transmitted for large graphs, this affects network costs. Since intervals may have a wide-range of values depending on the temporal graph, we use *variable byte-length* numbers to represent them, and observe that the overall message sizes drop by 59–78%. Also, for unit-length messages, and those that span till ∞, we pass just the start time point and a flag. This saves an 8-byte `long` for the end time point.

Inline Warp Combiner. We allow users to specify *warp combiners* that execute as part of the warp step before *compute*, and applies the combiner logic to the grouped and partitioned messages it generates for each interval. This limits the messages to one per partitioned state when calling *compute*, and can avoid a linear scan through the input messages. This can often be coupled with a receiver-side message combiner that is applied before warp.

Warp Suppression. Interval-centric computing works best when the intervals of entities are long, with large overlap across them. If the lifespan of vertices, edges and properties are small, there is no shared compute and messaging to exploit. Yet, the platform overheads for ICM will apply. Since warp has the most overhead, we selectively disable the warp step if more than a certain fraction of input messages to a vertex have unit lifespans. This avoids the warp costs and degenerates to a time-point centric execution model. While there are more calls to compute, this outstrips the cost of calling warp without its associated benefits. The correctness is not affected.

[3]Will be online at https://github.com/dream-lab/graphite

Table 1: Dataset Characteristics

Graph	#Snapshots	Largest Snap		Interval		Transf.		Multi-Snap.		Average Lifespan		
		|V|	|E|	|V|	|E|	|V|	|E|	∑|V|	∑|E|	V	E	Prop.
GPlus[1]	4	17M	225M	28.9M	462M	60M	493M	60M	462M	2.6	1	1
USRN[2,3]	96	24M	58M	24M	58M	1.2B	4.1B	24M	58M	96	96	4.82
Reddit[4]	121	280K	24M	9.1M	523M	60.4M	717M	64.6M	662M	6.6	1.22	1.12
MAG[5]	219	116M	1B	116M	1B	2.6B	11.6B	3.4B	13.1B	20.9	15.8	5.26
Twitter[6]	30	43.5M	2.1B	43.9M	2.1B	519M	26.3B	1.3B	60.1B	29.5	28.4	14.8
WebUK[7]	12	110M	3.9B	131M	5.5B	1.1B	34B	1.3B	45.3B	9.97	9.4	4.7
LDBC10	128	102M	1B	118M	1.4B	--	--	--	--	84	78	12.8

[1] home.engineering.iastate.edu/~neilgong/gplus.html
[2] users.diag.uniroma1.it/challenge9 [3] trafficengland.com
[4] cs.cornell.edu/ jhessel/projectPages/redditHRC.html
[5] openacademic.ai/oag [6] twitter.mpi-sws.org [7] law.di.unimi.it/datasets.php

VII. Experimental Evaluation

We offer a detailed comparative evaluation of the intrinsic benefits of the ICM model, and certain engineering optimizations of Graphite. *No single prior study has examined these number and variety of temporal graphs and algorithms.* For brevity, more details are given in our technical report [4].

A. Setup

1) TI and TD Algorithms: We implement 4 TI algorithms – *BFS* [16], *WCC* [28], *SCC* [28] and *PR*, and 8 TD algorithms *SSSP* [26], *EAT* [26], *FAST* [26], *LD* [26], *TMST* [9], *RH* [27], *LCC* [8] and *TC* [12] discussed earlier. The former do not use any properties, while the TD ones use edge properties.

2) Datasets: We run experiments for a diverse set of 6 real-world graphs (Table 1) to rigorously study the impact of their characteristics on the performance of the algorithms for Graphite and the baselines. These vary in the size, per snapshot and cumulatively *(Small: GPlus, USRN, Reddit; Large: MAG, Twitter, WebUK)*; lifetime of the temporal graph and entities *(Short: GPlus; Long: MAG, Twitter; Mixed: Reddit, USRN, WebUK)*; diameter *(Long: USRN; Short: rest)*; and degree distribution/domain *(Planar/Road: USRN; Powerlaw/-Social: rest)*. One edge property is present and used by the TD algorithms. None of the algorithms use vertex properties and is hence omitted. All graphs are based on real topologies. We introduce structure variations for Twitter using Facebook's LinkBench distribution [4], but the dynamism is real for the others. We use a distribution from a UK road traffic dataset for the properties of USRN and use the LDBC generator for Twitter [5], but the property variations are native for the rest.

3) Comparative Platforms: We compare ICM against four contemporary *baseline approaches* that we implemented over Apache Giraph. This ensures that the primitives are the key distinction and not the programming language or engine.

The *Multi snapshot baseline (MSB)* is used for TI algorithms. It loads and executes on each snapshot independently, using a VCM logic [18], [25]. We implement a variant (clone) of Chronos [7] which we call *Chlonos (CHL)* that enhances

[4]https://github.com/facebookarchive/linkbench
[5]http://ldbcouncil.org

MSB by sharing messages that span multiple adjacent snapshots. It loads a *batch of snapshots* into an in-memory layout that is vectorized into a single structure. Scatter identifies duplicate messages pushed by the compute to adjacent time-points of a sink vertex, and replaces them with one message for the whole interval, saving network time and memory. But, the compute call and state is separate for vertices in each snapshot. Chlonos can operate on incremental batches of snapshots, and each batch fits as many as possible in the distributed memory to run the algorithms. It is limited to expressing TI algorithms.

The *transformed graph baseline (TGB)* [26] converts the snapshots into *transformed graph* where interval vertices are unrolled into vertex replicas, one for the number of incoming and outgoing edges at distinct time-points, and each valid for a single time-point. This transformation is distinct for each algorithm. Edge-weights capture algorithm-specific properties, such as travel cost. Besides user messages and compute calls as part of VCM, shared states between different replicas are exchanged using special messages and applied using compute logic calls. We evaluate TGB only for TD algorithms. While it is possible to use it for TI algorithms, it is much worse than the other two baselines in performance and memory use. E.g., when using TGB for TI algorithms, GPlus was 7–16% slower that MSB, while it ran out of memory for MAG.

GoFFish-TS (GOF) [22] models a temporal graph as a sequence of snapshots. It allows messaging to adjacent snapshots and stateful execution of logic on vertices in each snapshot. An outer loop over the snapshots delivers temporal messages, and an inner loop of supersteps operates on one snapshot using VCM. Our implementation loads stateful snapshots from disk and processes them sequentially. Temporal messages and vertex states from prior snapshots are passed on disk. We limit GoFFish to TD algorithms as it degenerates to MSB for TI.

While we have attempted other platforms like *GraphX* [6] and *Tink* [14] their performance was much worse than ICM or the baselines [4]. E.g., for USRN, Tink took 4.2 × longer compared to TGB and 21.5 × longer than GRAPHITE for FAST, while it ran out of memory for Twitter. We have also evaluated *SAMS* [25] for TI algorithms. But it is written in C++ and for a single machine, and so not comparable directly. It performs 1.6–4.7× faster than our GRAPHITE setup, largely due to C++, but runs out of memory for WebUK. Hence, we exclude these systems from further evaluation.

4) System Setup and Metrics: We run the experiments on a 10-node commodity cluster. Each node has one 8-core Intel Xeon E5-2620 v4 CPU @ 2.1 GHz, 64 GB of RAM, 2 TB of HDD, and 1 Gigabit Ethernet. Each node runs CentOS 7.5 with Java 8, Apache Hadoop 3.1.1 and Apache Giraph 1.3, and is configured with 1 Giraph worker JVM with 14 threads and 60 GB heap space. Except for weak scaling, all other experiments use 8 nodes. Algorithms are run from a cold cache state. Giraph partitions graphs using its hash partitioner, and we disable its check-pointing and out-of-core computation. Graphs are loaded from HDFS.

We report *makespan* as the wall-clock time from the first user superstep, till the end of the last user superstep. This includes the cumulative *compute+ time*, which is the time for the compute (and scatter) calls overlapping with the messaging and barrier synchronization, and the *exclusive messaging time* after compute is done and only messages are being transmitted in a superstep. For fairness, *graph loading time* is reported separately. We also report the total number of calls to the user's *compute logic* and the *messages sent*.

Table 2: Ratio of the makespan of baseline platforms over GRAPHITE, averaged for TI and TD algorithms. 1× means same performance and > 1× means we are better. Italics indicate that some algorithms DNF for that graph and platform.

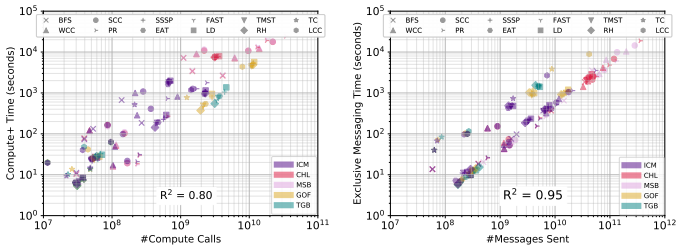
		GPlus	Reddit	USRN	Twitter	MAG	WebUK
TI Alg	MSB	0.95	1.14	0.97	24.79	12.99	5.80
	Chlonos	0.96	1.08	0.98	13.29	10.89	6.27
TD Alg	TGB	0.95	1.13	2.32	*19.90*	DNL	DNL
	GoFFish	0.96	1.05	6.49	6.75	4.60	*3.71*

B. Analysis

Table 2 summarizes the average speedup ($n\times$) GRAPHITE achieves across TI and TD algorithms, relative to other platforms for different graphs. DNL and DNF indicate that a platform *Did Not Load* the graph, or *Finish* the computation due to memory overflow. Fig. 5 plots the *makespan* for each algorithm (left Y axis) running on ICM and the baselines for the different graphs, along with the *number of compute calls* and *messages sent* (right Y axis). The makespan is further split into the total time spent on the *compute calls interleaved with messaging (compute+)* and for the *exclusive messaging* time after all compute calls are done in a superstep. If substantial, the total time spent for the *barrier synchronization between supersteps* or *JVM garbage collection (GC)* is indicated separately from the compute+ time they are usually part of. The TD algorithms run on ICM (indigo bar color), Chlonos (crimson) and MSB (magenta), while the TI algorithms run on ICM (indigo), GoFFish (gold) and TGB (teal); EAT and FAST are omitted in Fig. 5 for brevity. They perform similar to SSSP.

As Table 2 shows, GRAPHITE substantially outperforms all platforms for most graphs by 2.32–24.79×, and is comparable even for graphs that form the worst case for it. *These are based on the inherent characteristics of the ICM primitives rather than engineering artifacts.* We also weakly scale. These outcomes are discussed below.

1) ***All platforms have conceptually equivalent outcomes:*** As expected, all platforms produce *identical results* for all the algorithms and graphs. Further, the programming models produce *conceptually equivalent execution behavior* as well, but with different performance trade-offs. This is apparent when we examine GPlus (Fig. 5, (a)) which has unit-length edge intervals – all platforms degenerate to operating on each snapshot independently as edges do not span across. Here, all platforms have an *identical count of compute calls and messages for an algorithm on a graph*. Also, for each algorithm on a graph, MSB and Chlonos have the same number of compute calls; ICM and Chlonos have the same number of messages if the former can fit all snapshots of



(a) Compute Calls v. Compute+ Time (b) Messages v. Messaging Time

Figure 4: Log-Log Scatter plot of count of compute calls and messages, and their time contribution to the makespan.

the graph in a single batch (GPlus, Reddit, USRN); ICM and GoFFish have identical number of compute calls if properties change with every snapshot; and TGB and GoFFish have identical number of messages and compute calls, if the replica vertex state transfer messages and calls for TGB are ignored.

Compute calls and message counts are intrinsic to the programming model, as opposed to execution times that may depend on the platform and system at runtime. Matching these across billions of calls and messages helps assert that we are comparing the primitives and not just the platforms.

*2) **ICM primitives cause better GRAPHITE performance:*** ICM reduces the count of compute calls and messages sent for different algorithms and graphs, as we show later. These intrinsic improvements due to the primitives leads to better performance by GRAPHITE. All platforms are implemented using Giraph. Since the time spent in the compute calls and messaging form the bulk of the makespan for all platforms, we correlate these counts against the compute+ and messaging times using the scatter-plot in Fig. 4. There are 206 data points in each plot. We see a high correlation for both these factors, with $R^2 = 0.80$ for the compute+ and $R^2 = 0.95$ for messaging – the former is smaller since compute+ includes some interleaved messaging as well. This establishes that the performance of the platforms are consistent with the behavior of their primitives, and benefits seen for GRAPHITE are due to ICM and not better engineering.

*3) **ICM out-performs for graphs with longer lifespans:*** The benefits of ICM come from sharing compute and messages across multiple time-points. This is limited by the lifespan of the graph entities, as only temporally contiguous vertices can share compute calls with partitioned states, and neighboring vertices can share messages along their edge lifespans. The lifespan for the interval graph $\sqsupseteq$ interval vertex $\sqsupseteq$ adjacent edges $\sqsupseteq$ edge properties. So the benefits of ICM are constrained by the smallest of these. Our TI algorithms do not use edge properties and are affected by the edge lifespan. TD algorithms use edge properties and are limited by its lifespan.

Twitter and MAG have the longest average lifespans (Table 1). For Twitter, the edge lifespan is 28.4 and almost spans the entire graph lifespan. GRAPHITE is 24.1–26.3× faster for TI algorithms than MSB. This is equally due to a drop in the number of compute calls by ≈ 27× and in messages by ≈ 28×, compared to MSB. Chlonos calls compute on each time-point like MSB, but can share messages across intervals within a single batch. Due to the large size of Twitter, Chlonos can fit only 6 snapshots in memory and creates 5 batches. GRAPHITE takes 93% less time than Chlonos – largely due to 27× fewer compute calls that reduces makespan by 79%. While Chlonos sends fewer messages than MSB, it still sends ≈ 4.5× more messages than ICM due to the 5 batches.

Twitter's average edge property lifespan is 14.8 – half of its edge lifespan. However, GRAPHITE is 19.1–20.3× faster than TGB, with a 95% smaller makespan, for the TD algorithms. Besides an 8× drop in messages and 10.5× drop in compute calls, there are two other factors at play. One, despite hash-based vertex partitioning, 70% of the messages are for 4 of the 8 graph partitions. This network bottleneck causes a higher messaging time for TGB. Two, the larger size of the Twitter transformed graph causes memory pressure and triggers the JVM GC, causing GRAPHITE to have a 40% lower makespan. This is discussed in Sec. VII-B4. GRAPHITE is 2.98–8.2× faster than GoFFish, mainly due to an 8× drop in the message count, and partly due to a 6× drop in compute calls. Like TGB, GoFFish does not share compute or messages across intervals.

Also, ICM is faster for TI (≈ 12×) and TD (≈ 4.6×) algorithms for MAG due to fewer compute calls and messages, which correlate with its edge (≈ 15.8×) and property (≈ 5.3×) lifespans.

*4) **ICM out-performs for large graphs:*** ICM offers several benefits for temporal graphs with *large sizes* and long lifespans, but due to complementary reasons from above. Its interval graph model that is loaded and retained in distributed memory is more compact than the transformed graph of TGB (Table 1, Fig. 6(a)). E.g., the transformed graph for MAG and WebUK cannot load into 480 GB of distributed memory. They need 604 GB and 684 GB of memory just to load the graph, compared to just 130 GB and 183 GB for our interval graph. Besides memory pressure, this also increases the number of messages and compute calls performed in TGB to share state between replica vertices, e.g., by 50% on Twitter. While these are more light-weight than the application compute calls and messages, they do pose a noticeable overhead.

Large graphs use more memory and create billions of message objects. This triggers the JVM's GC; we use the G1 GC that is efficient for large heap sizes. E.g., for Twitter, TGB calls GC 33 times for SSSP and this takes ≈ 32% of its total makespan, compared to 6 calls to the GC for ICM that account for 5% of its makespan. For WebUK, calls to GC make up ≈ 20% of ICM's makespan for TD algorithms, limiting its improvements over other platforms. GC calls are fewer for GoFFish and MSB that operate on just one snapshot at a time, and it depends on the batch size for Chlonos. E.g., Chlonos is slower than MSB only for WebUK due to GC overheads on batches of 2 snapshots, which outstrips its message sharing benefits. However, often the compute times dominate GC time. E.g., for MAG, ICM spends 27–163 seconds on GC for TI algorithms, which is more than Twitter's 11–42 seconds, but forms just 3–6% of the overall makespan.

While MSB, Chlonos and GoFFish relieve memory pressure

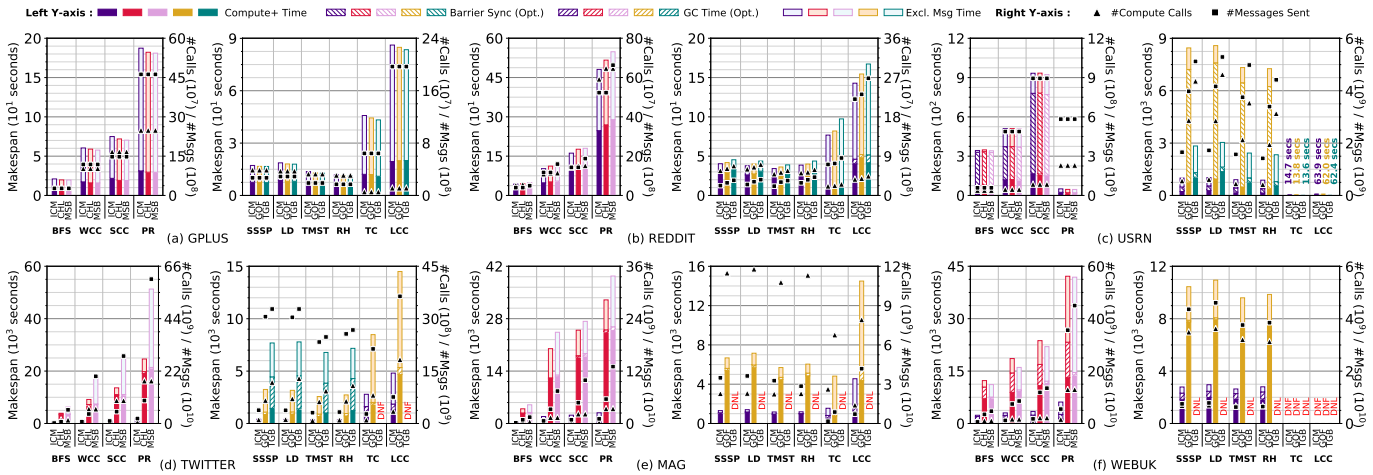
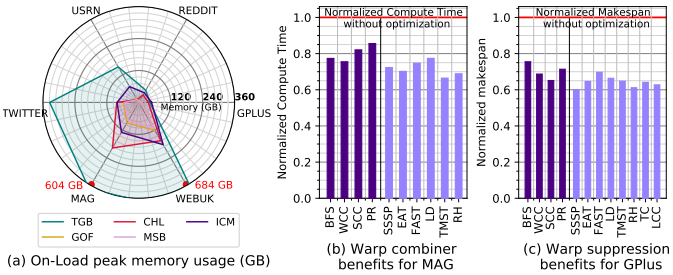


Figure 5: Makespan and the count of compute calls and messages sent for the 4 TI and 6 TD algorithms; EAT/FAST are omitted for brevity. Barrier & GC time splits for makespan are shown only if large. Note the different scaling on the Y axis.



(a) On-Load peak memory usage (GB)

(b) Warp combiner benefits for MAG

(c) Warp suppression benefits for GPlus

Figure 6: GRAPHITE optimizations and memory footprint.

by operating on one or a batch of snapshots, their snapshot data size on disk is larger than ICM. Fig. 6(a) shows the in-memory size of the interval/transformed graph (ICM, TGB) and largest snapshot/batch (MSB, Chlonos, TGB) on loading. TGB has the largest size followed by Chlonos, ICM, GoFFish and MSB. While these result in disk and network I/O load times from HDFS for ICM and TGB, these times *accumulate across different snapshots/batches* for MSB, Chlonos and GoFFish. E.g., for MAG, these cause an additional 24 secs (GRAPHITE), 2682 secs (MSB), 138 secs (Chlonos) and 2931 secs (GoFFish); TGB did not finish, but took 103 secs on a larger cluster. *These times are substantial, but not included when we report the makespan out of fairness to other platforms.*

Lastly, using *warp combiner* reduces a pass by the warp and another by the compute on the input messages into a single pass that does both. All our algorithms except LCC and TC are commutative and associative, and define combiners. This benefits large graphs with many messages received per interval vertex. Fig. 6(b) shows the benefits of using the combiner in GRAPHITE for MAG, relative to disabling it. The *compute time* drops by 17–25% across all algorithms, which lowers makespan by 1.2–1.5×. A 16–27% drop in compute time is seen for WebUK. This feature is enabled for all experiments.

5) ***ICM limits downsides, and is competitive even for short-lifespan graphs:*** There is limited or no benefit from ICM for graphs with unit or small lifespan of entities, like GPlus and Reddit, since we cannot share compute or messaging. However, ICM and warp introduce overheads to the GRAPHITE platform relative to the stock Giraph used by the baselines. Our *automatic warp suppression* mitigates this. Here, messages do not pass through the warp if the number of unit-length messages to an interval-vertex is above a threshold (default 70%) in a superstep. Its benefits are evident in Fig. 6(c) for GPlus, which has unit-length edges and is the worst-case for ICM. The makespan reduces by 25–40% with this feature, and we are only marginally slower by ≈ 7% (excluding load times) compared to the other baselines (Fig. 5(a)). This is both due to avoiding warp and reduced messaging. These benefits are also seen for Reddit, where 96% of edges have unit lifespans and yet GRAPHITE manages to out-perform the other platforms by ≈ 14%.

Another optimization for short-lifespan graphs replaces the pair of start and end time-points for a unit-length interval with just one value. This saves 8 bytes per message, which adds up for ≈ $5B$ peak messages sent for GPlus and Reddit.

6) ***ICM benefits graphs with large diameters, and is competitive for non-temporal structures:*** Graphs like USRN have no structural changes, and only properties change. As a manual optimization, developers may instruct MSB and Chlonos to just *operate on a single snapshot and reuse its results* for the TI algorithms. ICM operates on the interval graph, with vertex and edge lifespans matching the graph's lifespan. It naturally sets the message intervals to match this, and automatically garners similar benefits for the TI algorithms. So GRAPHITE's makespan is comparable to these platforms (despite omitting load times). MSB and Chlonos cannot benefit even if there is a small change in the topology, such as for Reddit. TD algorithms use edge properties, and do not benefit from the static topology of USRN as its edge properties vary.

ICM offers some benefits due to the large diameter of 6262 for USRN. The superstep count is proportional to the diameter for traversal algorithms, while PR, TC, and LCC have fixed superstep counts of 10, 3, and 4 respectively. The total

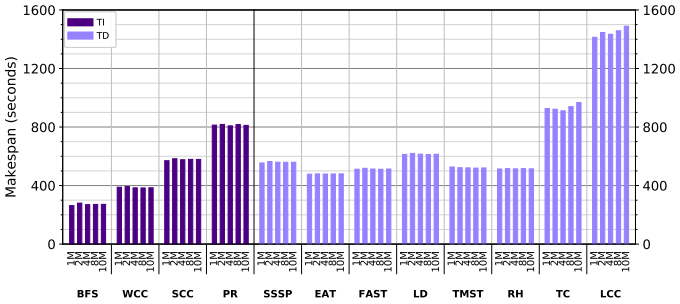


Figure 7: Weak Scaling of GRAPHITE for all algorithms on synthetic graphs, using $1, 2, 4, 8$ and 10 machines ('xM' on X axis). Each machine holds $\approx 10M$ vertices, $\approx 100M$ edges.

barrier synchronization time is separately shown for USRN (Fig. 5(c)). While Giraph spends $\approx 40ms$ on a barrier, this adds up to dominate the makespan for all platforms. This is worse for TD algorithms as they multiply over snapshots for GoFFish. The diameter of the transformed graph is also greater than or equal to the interval graph. TGB takes slightly more barrier time than ICM.

7) ***ICM exhibits weak scaling:*** Weak scaling is a common scalability metric where, ideally, the makespan stays constant as the input and the resources increase proportionally. We perform weak scaling experiments for GRAPHITE by increasing the interval graph size and the number of machines. We generate a synthetic graph using LDBC's Facebook degree distribution [5] , and perturb its structure over 128 time-points using Facebook's LinkBench distributions [4]. The largest snapshot for a graph has $m \times 10M$ vertices and $m \times 100M$ edges, for $m = \{1, 2, 4, 8, 10\}$ machines (Table 1). In Fig. 7, GRAPHITE exhibits *near ideal weak scaling*, with the makespan staying almost constant as the machine count increases, with a fixed load per machine. The scaling efficiency is 95–106%, and indicates that we can scale well to even larger graphs.

8) ***ICM algorithms are concise:*** The lines of user logic code (LoC) for GRAPHITE is 15–47% fewer compared to Chlonos, 19–44% fewer than GoFFish, and 46–152% fewer than TGB. Our LoC is marginally higher than MSB, by 3–19% (exactly 3 lines). These 3 additional lines in TI algorithms are ICM API calls. The 4 TI algorithms take 19–114 LoC using ICM, while the 8 TD algorithms take 27–80 LoC.

VIII. CONCLUSION

In this paper we propose an Interval-centric Computing Model (ICM), a novel and unifying abstraction for designing distributed TI and TD algorithms over temporal graphs. Our warp operator enhances usability and improves performance by sharing compute and messaging across intervals, where possible. Our experiments extensively validate these intrinsic performance and scalability benefits. Our ability to express 12 TD and TI algorithms attests to its intuitiveness. ICM plugs a key gap in current literature for generic and scalable temporal graphs primitives. In future, we plan to extend ICM to process real-time temporal graphs of a streaming nature, offer query capabilities over temporal property graphs and explore storage and partitioning strategies.

REFERENCES

[1] J. F. Allen. Maintaining knowledge about temporal intervals. *CACM*, 1983.

[2] R. Angles et al. G-CORE: A core for future graph query languages. In *ACM SIGMOD*, 2018.

[3] Z. Cai, D. Logothetis, and G. Siganos. Facilitating real-time graph mining. In *CloudDB*, 2012.

[4] S. Gandhi and Y. Simmhan. Graphite: An interval-centric model for distributed computing over temporal graphs. Technical report, Indian Institute of Science, 2019. http://cds.iisc.ac.in/faculty/simmhan/icm.pdf.

[5] J. Gao, P. K. Agarwal, and J. Yang. Durable top-k queries on temporal data. *PVLDB*, 2018.

[6] 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 *OSDI*, 2014.

[7] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen. Chronos: a graph engine for temporal graph analysis. In *EuroSys*, 2014.

[8] P. Holme and J. Saramäki. Temporal networks. *Physics Reports*, 2012.

[9] S. Huang, A. W.-C. Fu, and R. Liu. Minimum spanning trees in temporal graphs. In *ACM SIGMOD*, 2015.

[10] A. P. Iyer, Q. Pu, K. Patel, J. E. Gonzalez, and I. Stoica. TEGRA: Efficient ad-hoc analytics on time-evolving graphs. Technical report, 2019.

[11] K. Kulkarni and J.-E. Michels. Temporal features in SQL:2011. *SIGMOD Rec.*, 2012.

[12] R. Kumar and T. Calders. 2SCENT: an efficient algorithm for enumerating all simple temporal cycles. *PVLDB*, 2018.

[13] A. G. Labouseur, J. Birnbaum, P. W. Olsen, S. R. Spillane, J. Vijayan, J. Hwang, and W. Han. The G* graph database: efficiently managing large distributed dynamic graphs. *Distr. and Parallel Datab.*, 2015.

[14] W. Lightenberg, Y. Pei, G. Fletcher, and M. Pechenizkiy. Tink: A temporal graph analytics library for apache flink. In *WWW Comp.*, 2018.

[15] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *PVLDB*, 2012.

[16] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *ACM SIGMOD*, 2010.

[17] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 2015.

[18] V. Z. Moffitt and J. Stoyanovich. Temporal graph algebra. In *DBPL*, 2017.

[19] B. Moon, I. F. V. Lopez, and V. Immanuel. Scalable algorithms for large temporal aggregation. In *IEEE ICDE*, 2000.

[20] S. Salihoglu and J. Widom. Optimizing graph algorithms on pregel-like systems. *PVLDB*, 2014.

[21] K. Semertzidis, E. Pitoura, and K. Lillis. Timereach: Historical reachability queries on evolving graphs. In *EDBT*, 2015.

[22] Y. Simmhan, N. Choudhury, C. Wickramaarachchi, A. Kumbhare, M. Frincu, C. Raghavendra, and V. Prasanna. Distributed programming over time-series graphs. In *IPDPS*, 2015.

[23] G. Slivinskas, C. S. Jensen, and R. T. Snodgrass. Query plans for conventional and temporal queries involving duplicates and ordering. In *IEEE ICDE*, 2000.

[24] M. D. Soo, R. T. Snodgrass, and C. S. Jensen. Efficient evaluation of the valid-time natural join. In *IEEE ICDE*, 1994.

[25] M. Then, T. Kersten, S. Günnemann, A. Kemper, and T. Neumann. Automatic algorithm transformation for efficient multi-snapshot analytics on temporal graphs. *PVLDB*, 2017.

[26] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu. Path problems in temporal graphs. *PVLDB*, 2014.

[27] H. Wu, Y. Huang, J. Cheng, J. Li, and Y. Ke. Reachability and time-based path queries in temporal graphs. In *IEEE ICDE*, 2016.

[28] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *PVLDB*, 2014.