Distributed Programming Abstraction for Scalable Processing of Temporal Graphs

A THESIS

SUBMITTED FOR THE DEGREE OF

Master of Technology (Research)

IN THE

Faculty of Engineering

BY

Swapnil Gandhi

Department of Computational and Data Sciences
Indian Institute of Science
Bangalore – 560 012 (INDIA)

January, 2020

Declaration of Originality

I, **Swapnil Gandhi**, with SR No. **06-02-00-10-22-17-1-14764** hereby declare that the material presented in the thesis titled

Distributed Programming Abstraction for
Scalable Processing of Temporal Graphs

represents original work carried out by me in the **Department of Computational and Data Sciences** at **Indian Institute of Science** during the years **2017-2020**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Yogesh Simmhan Advisor Signature

DEDICATED TO

Mom and Dad

Acknowledgements

This thesis would not have been possible without the support of many people across various walks of my life.

I am deeply indebted to my advisor, Yogesh Simmhan. It is not an exaggeration to say that my research career changed significantly after I started working with Yogesh. This thesis is a culmination of his careful mentoring and guidance throughout my Masters degree. I am also very grateful to Jayant Haritsa. His teachings, both inside and outside class have helped me develop a different viewpoint on various things, many of which became a natural part of this thesis.

I would be remiss if I did not mention the many selfless hours Aakash Khochare and Anubhav Guleria put in for the numerous discussions we have had over the last two years. I am thankful to Srinivas Karthik, Anupam Sanghi, Sheshadri K R, Shriram R and members of DREAMLab for their valuable and observant feedback, which has helped me crystallize my ideas.

My time at the Indian Institute of Science has been made wonderful by support from many friends and family. Kunal and Neha hosted me on many Saturday evenings (sometimes on Sundays too) and I am thankful for the good times we spent hanging out. Last but not least, I would like to thank Baa, Dada, Mom, Dad, Shilpi, and Hardik for their continuous support, their patience and immeasurable understanding. Thank you for being their.

Abstract

Graph-structured data is on the rise, in size, complexity and dynamism, and underlies many traditional and modern applications in diverse fields of science, engineering and business. While analysis of static graphs is a well-explored field, new emphasis is being placed on understanding and representing the ways in which networks evolve over time through the insertion, deletion and modification of vertices, edges, and associated attributes. Such evolving graphs that have been persisted for offline analysis are called temporal graphs while those that continue to change are called dynamic graphs. For example, one may wish to study the evolution of well-studied static graph properties such as centrality measures, density, conductance, etc., over time. Another need is to search and discover temporal patterns, where the events that constitute the pattern are spread out over time. Despite increasing interest and availability, there is limited work on distributed programming abstractions to design algorithms over temporal graphs and on scalable platforms to execute them. Moreover, existing abstractions and platforms developed for static graphs are either inapplicable, need non-trivial algorithm-specific generalization or are inefficient due to redundant computation and communication. We address both these gaps in this thesis. We propose high-level distributed programming primitives for algorithm designers to concisely express a wide range of common and novel analytics over temporal graphs, while abstracting away the fine-grained orchestration and distribution of computation across a cluster of servers to achieve weak scaling. These primitives and distribute platform are one of the *first of their kind contributions* to temporal graph processing.

Specifically, we focus on *ad hoc* batch processing of fully evolved time-varying graphs, also known as temporal graphs. We propose an Interval-centric Computing Model (ICM) for distributed iterative processing over the entire history of the graph. Users define their computing and communication logic from the perspective of a vertex and its time-interval, and this also forms the unit of data-parallel computation. The cornerstone of our model is a unique transformation operator called *Time-warp*, which enables automatic sharing of computation and communication across adjacent time-points of a vertex. *Graphite* is our open-source distributed implementation of ICM by extending Apache Giraph that enables composability of multi-stage

algorithms and includes optimizations to enhance compute and communication performance. We use it to design 12 common temporal graph algorithms from literature. We have rigorously evaluated its performance for 6 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×, and matches them even in worst-case scenarios.

We also make a preliminary contribution on primitives for incremental processing of dynamic graphs, where a stream of graph updates are applied to an existing temporal graph. Here, the intuition is to recompute the algorithm only on those parts of the graph that have changed and not on the entire graph. We offer initial results on this approach, and leave the generalization of this model to a broader class of streaming graph algorithms to future work.

Publications based on this Thesis

1. **S. Gandhi**, and Y. Simmhan, "An Interval-centric Model for Distributed Computing over Temporal Graphs", 2020 IEEE 36$^{\text{th}}$ International Conference on Data Engineering (ICDE), Dallas, Texas.

2. **S. Gandhi**, "Wave : A Substrate for Distributed Incremental Graph Processing on Commodity Clusters", 2$^{\text{nd}}$ ACM *Student Research Competition* (SRC) at 27$^{\text{th}}$ Symposium on Operating Systems Principles (SOSP), Ontario, Canada. 2019 *(Won Bronze Medal)*

3. **S. Gandhi**, and Y. Simmhan, "Think like an Interval: A Distributed Computing Model for Temporal Graphs", 2020. *(Under preparation as a Journal Article)*

Contents

List of Figures

List of Tables

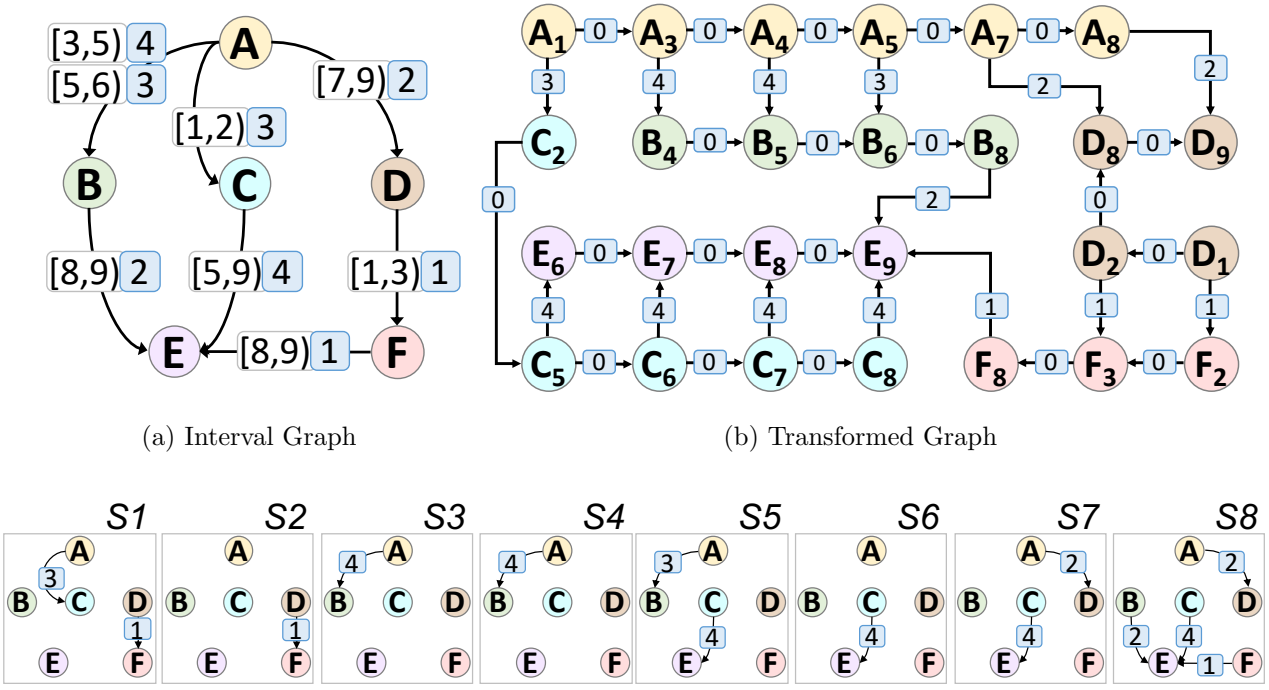
Chapter 1

Introduction

Graph-structured data is on the rise, in size, complexity and dynamism, and underlies many traditional and modern applications in diverse disciplines, spanning from social, transport and communication networks, to molecular biology, neuro-science and epidemiology. Typically, such data is represented as a static graph that describes the concepts (vertices) and the relationship between them (edges), with optional attributes associated with vertices and edges. While analysis of static graphs is a well-explored field [43], new emphasis is being placed on understanding and representing the ways in which networks evolve over time through the insertion, deletion and modification of vertices, edges, and associated attributes [45, 116]. For example, one may wish to study the evolution of well-studied static graph properties such as centrality measures, density, conductance, etc., over time [66]. Another need is to search and discover temporal patterns, where the events that constitute the pattern are spread out over time [55]. Such time-evolving graphs that have been persisted for offline analysis are called *temporal graphs* while those that continue to change via stream of updates are called *dynamic graphs*.

1.1 Abstractions and Platforms for Temporal Graphs

Temporal graphs are an emerging class of fully-evolved property graphs [8] with applications in both traditional domains like transit [25], financial transaction and social networks [115], and emerging ones like Internet of Things, knowledge graphs and human connectomes [110]. The structure and attributes of such graphs may change over time [45, 116]. 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. 1.1(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



(a) Interval Graph

(b) Transformed Graph

(c) Multi-Snapshot Graph

Figure 1.1: Transit network as a temporal graph.

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)* [107]. TI algorithms, also called *snapshot-reducible* [101], can discretize a temporal graph into *snapshots*, one per time-point [40], and operate on each snapshot independently. E.g., Fig. 1.1(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 [74], or multi-snapshot approaches like SAMS [107] can be used to design and execute such algorithms on temporal graphs. The latter avoids redundant computation across different snapshots to improve performance.

TD algorithms, also called *extended snapshot-reducible* [101], 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 [120]. *TD clustering coefficient* helps estimate the rate of spread of a disease over time [104], while *TD centrality measures* are used to estimate information propagation delays in social networks [45]. *Temporal motifs* like feed-forward triangles in transaction networks let us identify monetary routing patterns [61]. Notice that in these applications it is essential that the temporal order is respected.

Multi-snapshot approaches applied to TD algorithms can give incorrect results [78, 107, 120]. TD algorithms for earliest/latest arrival time and reachability have been proposed [120]. Other bespoke algorithms [33, 46] and patterns can be extended to similar ones. E.g., the *transformed graph* approach [120] 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. 1.1(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 [107], Chronos [40] and GraphInc [17] reuse computing or messaging across snapshots, and some operate in a distributed mode for scalability [17]. But they are limited to TI algorithms. Distributed abstractions for TI and TD algorithms [70, 99] 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 [76, 74]. 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.

1.1.1 TD Algorithm Example using Temporal SSSP

Say we wish to find a time-respecting path with the *shortest travel cost* [120] in the transit network in Fig. 1.1(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. 1.1(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 1.1(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. 4, 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. E.g., the shortest path solution $A \rightarrow B \rightarrow E$ does not exist in any single snapshot of Fig. 1.1(c).

1.2 Contributions

We make the following contributions in this thesis:

1. We define the temporal graph *data model* in Chapter 3. We introduce and illustrate the novel *ICM programming abstraction* and *time-warp operator* to design distributed TI and TD algorithms on temporal graphs, in Chapter 4.

2. We discuss the use of ICM to *intuitively design* 12 *TI and TD algorithms* from literature in Chapter 5.

3. We describe the architecture of GRAPHITE *distributed platform*, which implements ICM,

along with its features and optimizations, in Chapter 6.

4. In Chapter 7, 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.

5. In Chapter 8, we present our preliminary results on WAVE, an approach to extending ICM to support incremental graph computation.

1.3 Organization of the thesis

Rest of the thesis is organized as described below. Chapter 2 presents background and review of related work. Chapter 3 defines the temporal graph data model. Chapter 4 describes our interval-centric computing model (ICM) for designing TI and TD algorithms. Chapter 6 presents GRAPHITE, which is our implementation of ICM over Apache Giraph. Chapter 7 evaluates the performance of GRAPHITE for diverse real-world temporal graphs. Chapter 8 presents our preliminary results on WAVE, a substrate for incremental graph computation. Chapter 9 presents conclusions.future work.

Chapter 2

Background and Related Work

In this chapter, we offer a background on component-centric computing models that form the basis for our proposed Interval-centric Computing Model (ICM) temporal graph abstraction. We also discuss related works on temporal and dynamic graph processing primitives and platforms, identify their gaps, and contrast how ICM, our GRAPHITE platform for ICM, and our preliminary work on WAVE for dynamic graph processing address these.

2.1 Background

Pregel [74] is a Vertex-centric Computational Model (VCM) designed for large scale distributed graph processing, inspired by Valiant's *Bulk Synchronous Parallel* [109] programming model. Pregel has proved popular for designing distributed and scalable graph algorithms on large graphs that execute on distributed machines in a commodity cluster. There have also been subtantial research into extending Pregel's programming abstractions [100, 122, 38] and the platform capabilities of its open source implementation, Apache Giraph [90, 57].

The input to a Pregel program is a *directed graph* whose vertices, along with their respective out-edges, are partitioned across machines of a computing cluster. Graphs with undirected edges are expressed as a pair of directed edges. Pregel algorithms are executed as a *sequence of iterations (supersteps)*.

Pregel requires programmers to "think like a vertex" by following a *vertex-centric computing model (VCM)*. Here, users provide the logic to be executed at each vertex independently. During a superstep, this vertex-centric logic is invoked for each vertex, in a data-parallel manner. In each superstep, the logic can *receive messages* sent to it from the previous superstep, *send messages* to other vertices to be delivered in the next superstep, and *modify the state* of the vertex and its outgoing edges, typically by processing its received messages and the prior states.

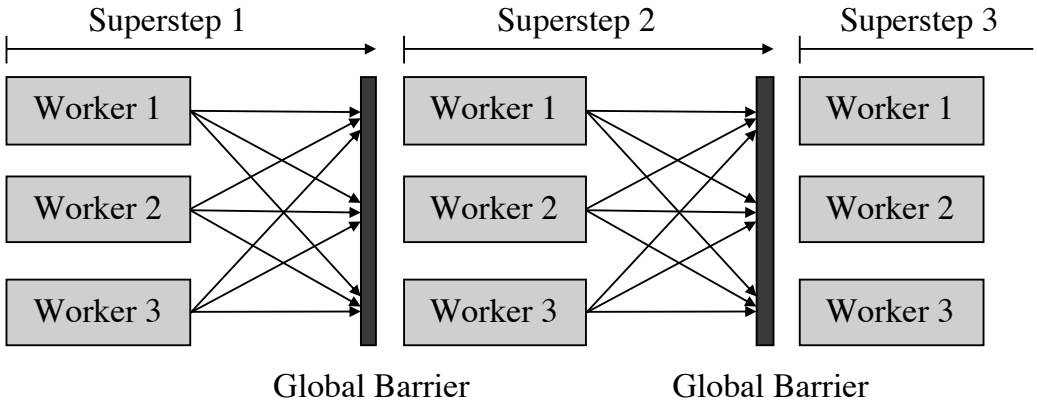


Figure 2.1: Bulk synchronous processing (BSP) computation model of Pregel, illustrated with three supersteps and three workers [57]

A *synchronization barrier* is present between supersteps is used to ensure that all messages are delivered at the beginning of the following superstep. This is illustrated in Figure 2.1.

The supersteps proceed will until there is a global consensus to stop the execution. A vertex starts in the active state. It may *vote to halt* at any superstep and it will get deactivated; it will be reactivated only if it receives a message in a future superstep. The program terminates when all vertices are inactive and there are no messages in transit.

Apache Giraph [1] is a popular open-source implementation of Pregel that uses a Map-only Hadoop job for computation. A number of distributed graph algorithms for static graphs have been expressed using VCM due to its intuitive abstraction, including PageRank [74], Connected Components [91], Coloring [91], Clustering [74], Bipartite Matching [74] and Traversal-Based algorithms [123]. Our Interval-centric Model of Computing (ICM) described later (Refer Chapter 4) is inspired by the component-centric, iterative execution model of Pregel.

2.2 Related Work

2.2.1 Static Graph Processing

2.2.1.1 Programming Abstractions and Primitives

Graph applications tend to be irregular and computationally complex [105]. Graph processing primitives offer a structure to more-easily design and execute graph algorithms. Graph programming abstractions such as *Vertex-Centric Programming* [74], *Gather-Apply-Scatter (GAS)* [72], *Edge-Centric Programming* [88], and *Subgraph-Centric Programming* [100, 84] adopt a data-parallel and iterative execution model were users design graph analytic from the perspective of a

component which could vary from a vertex (or edge) to a partition. Parallelism is exposed at the granularity of *graph components*, and hence these are also called component-centric computing models [76], with VCM the most common [91, 123].

2.2.1.2 Distributed Platforms

Distributed graph programming platforms such as *Pregel* [74], GraphX [35], *GraphLab* [72], *X-Stream* [88], *GoFFish* [100], and *NScale* [84] are designed to horizontally scale high-level graph programming primitives on multiple CPU cores and cumulative memory across machines. These platforms hide the complexity of orchestration, communication, and synchronization from the end users. Giraph is a VCM platform which leverages the task scheduling component (namely YARN) of Hadoop clusters for orchestration. It runs workers as special mappers, which communicate with each other to deliver messages between vertices and makes use of global barriers to synchronize between supersteps. In-addition to graph-specific systems, general-purpose iterative data processing systems such as Hadoop [98], Spark [125] and Flink [18] have been leveraged to program graph analytics.

However, existing abstractions and systems focus on large static 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 programmability for temporal graphs and enhance their scalability.

2.2.2 Temporal Graph Processing

2.2.3 Time Independent Temporal Graph Algorithms

Time Independent (TI) processing of temporal graphs models them as a *series of snapshots* [41]. This allows existing primitives, platforms and algorithms for static graph processing [91, 123] to be applied independently to each snapshot at a distinct time-point. Efficient storage of multiple snapshots on-disk and in-memory, and their hierarchical indexing for fast snapshot retrievals have also been proposed [58, 73, 59, 77, 20]. However, processing snapshots independently causes redundant computation and messaging, limiting scalability. Systems and abstractions [40, 17, 65, 15, 87] have tried to address this inefficiency.

In particular, *SAMS* [107] presents a suite of rewriting rules which enable automatic co-scheduling of common steps during multi-snapshot analysis, in a style similar to SIMD processing. This addresses some of the performance limitations we ourselves observe in our experiments when operating over a large number of snapshots. However, SAMS does not offer mechanisms for distributed execution, which limits scalability. More importantly, the multi-snapshot approach implicitly limits us to the class of snapshot-reducible algorithms, which leaves out many interesting applications.

Chronos [40] offers an efficient in-memory layout for vertices that span multiple snapshots to leverage time-locality. It couples this with a vertex-centric engine that does batched execution over multiple snapshots, with concurrent processing of the vertex states from multiple snapshots. These enhance cache hits. Unlike us, the user logic execution for a vertex is not shared across snapshots. But they do reduce (in-memory) communication costs when pushing common messages that span contiguous snapshots.

However, these platforms are designed for independent snapshot analytics. States from prior snapshots are used to reduce the recompute time for a later snapshot rather than support time-dependent algorithms. ICM supports both TI and TD algorithms, but focus on fully evolved graphs with valid time [62] rather than streaming ones.

2.2.4 Time Dependent Temporal Graph Algorithms

Time Dependent (TD) algorithms actively use 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 [33, 46, 120, 119, 117, 82, 64, 127, 71, 68, 96, 23]. These are not generalizable primitives, though TD algorithms that are similar to each other can reuse a pattern. Of these bespoke design-patterns, the *transformed graph approach* [119] 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 (Chapter 7), it bloats the graph size and suffers from poor scalability.

Like us, *Tink* [70] 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.

GoFFish-TS [99] proposes primitives for designing TD algorithms using a multi-snapshot approach. Here, the state from a prior snapshot can be explicitly passed to the next snapshot by the user logic. Within a snapshot, it uses a subgraph-centric model of execution. It too does not avoid sharing computation, is limited to proceeding one snapshot at a time, and states have to be explicitly passed by the user logic 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.

2.2.5 Dynamic Graph Processing

Unlike static graph processing systems, streaming systems like Kineograph [21], Tornado [97], ReMo [92], GraphBolt [24] and Kickstarter [114] operate on *dynamic graphs*. Contrarily to static graph processing systems, these systems execute graph computation *concurrently* with graph updates. Some of these frameworks even allow computation results to be updated *incrementally*, rather than recomputed from scratch when data is updated.

Kineograph [21] supports incremental processing of real-time graph updates. It constructs consistent snapshots of an evolving graph for streaming computation. It reuses the state of the prior snapshot to rapidly compute an analytic for the new snapshot. However lacks support for deletes, which is non-trivial to achieve in incremental graph algorithms. *GraphInc* [17] is complementary to Kineograph in that it supports real-time updates, but also memoizes incoming messages to avoid redundant vertex-compute if the same message was seen earlier. Both these platforms must complete updates to current snapshot before moving to the next *Tegra* [50] relaxes this by allows 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. It is designed over Apache Spark [125].

Tornado [97] processes streams of graph updates by forking execution to process user-program while the graph structure updates in the main branch, but only supports a subset of algorithms. *Kickstarter* [114] uses a global dependence tree to maintain state dependencies over RDMA. *GraphBolt* [24] uses approximate techniques to trade-off accuracy for performance.

However, these systems only allow maintaining computation results on the latest snapshot and do not support any notion of time. On the other hand, Wave supports incremental maintenance of computational results across time.

2.2.6 Models and Algebra

The need to manage and process temporal data, in the context of graphs or otherwise, has been well-studied in the database research community [32, 93]. Snodgrass's seminal work on defining the temporal data model dates back to the early 1990s [102]. Further, temporal join, coalescing, alignment and aggregation have been studied in the context of relational algebra [11, 28, 32, 131]. Our time-warp operator, which deals with the problem of partitioning overlapping time intervals into disjoint interval groups, is similar to the recently proposed *disjoint interval partitioning* (DIP) [16] for temporal joins and other sort-based operations (e.g., temporal aggregation). Despite the similarity, time-warp is specifically targeted at the unique requirements of temporal graph processing, and allows us to avoid runtime overheads in temporal graph processing.

Temporal data model and querying primitives from relational databases [62] are only gradually translating to modeling temporal features in graphs, and on graph querying languages [9]. *Moffit and Stoyanovich* [78] 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 [95], and indexing for temporal shortest path queries [94, 15]. ICM is imperative and can be used to design general purpose temporal graphs analytics, and is complementary to these.

2.2.7 Graph Storage and Maintenance Systems

Update-optimized graph storage systems like *DeltaGraph* [58, 59] focus on efficiently storing updates and provide access to state of the graph at multiple points in time using differential versioning and delta-encoding. Clustering temporally adjacent snapshots and computing a representative snapshot was also proposed [87]. Others systems like *ImmortalGraph* [77] ensure efficient storage and retrieval by exploring on-disk temporal and structural locality. *LLAMA* [73] applies incoming updates in batches and creates copy-on-write snapshots for graph analysis. Both, LLAMA and GraphOne [63] can handle queries running on the most recent version of the graph while updates are applied concurrently. However, these systems lack support for TD algorithms as the partial paths over time are lost across snapshots. GRAPHITE is well-suited to process graph data stored in such storage systems and we state that efficient storage of temporal graph is beyond the scope of this thesis.

Chapter 3

Temporal Graph Model

Our Interval-centric Computing Model (ICM) is a distributed primitive for composing analytics over *temporal graphs*. These are historic graphs with dynamism in their structure and attributes, but which are *fully evolved* and ready for processing. In this chapter, we define the *temporal graph data model* that our proposed ICM abstraction supports. Such formalism is given to avoid ambiguity in this fast changing domain.

3.1 Preliminaries

3.1.1 Time Domain.

Without loss of generality, 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.

3.1.2 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}, i.e., a half-open notation [7, 62]. The time-points that are part of a time-interval $\tau = [t_{start}, t_{end})$ is the set $\{t \mid \ t \in \Omega \text{ and } t_{start} \leq t < t_{end}\}$.

3.1.3 Interval Relations

Boolean relations between intervals follow *Allen's* conventions [7]. 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.

3.2 Temporal Graph

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 $val \neq val'$.

Example. As a simple example of Temporal Graph, consider the transit network interval graph shown earlier in Figure 1.1(a). Here we have V, E, L, and A_E as vertices, edges, property labels, and edge property values, respectively. In this example, there are no vertex properties.

$$V = \{\langle A, [0, \infty) \rangle, \langle B, [0, \infty) \rangle, \langle C, [0, \infty) \rangle, \langle D, [0, \infty) \rangle, \langle E, [0, \infty) \rangle, \langle F, [0, \infty) \rangle\}$$

$$E = \{\langle AB, A, B, [3, 6) \rangle, \langle AC, A, C, [1, 2) \rangle, \langle AD, A, D, [7, 9) \rangle, \langle BE, B, E, [8, 9) \rangle,$$
$$\langle CE, C, E, [5, 9) \rangle, \langle DF, D, F, [1, 3) \rangle, \langle FE, F, E, [8, 9) \rangle\}$$

$$L = \{W\}$$

$$A_E = \{\langle AB, W, 4, [3, 5) \rangle, \langle AB, W, 3, [5, 6) \rangle, \langle AC, W, 3, [1, 2) \rangle, \langle AD, W, 2, [7, 9) \rangle,$$
$$\langle BE, W, 2, [8, 9) \rangle, \langle CE, W, 4, [5, 9) \rangle, \langle DF, W, 1, [1, 3), \langle FE, W, 1, [8, 9) \rangle \rangle\}$$

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

3.2.1 Constraints

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 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.

3.2.2 Space Complexity

A temporal graph over k time-points, when modeled as a sequence of snapshots (Refer Fig. 1.1(c)) consumes $\mathcal{O}(k \times (|V| + |E|))$ space. Our interval graph model reduces this to $\mathcal{O}(|V| + (\delta \times q \times |E|))$, where δ is the most times an edge in the graph is updated and q is number of properties. Usually, $\delta \ll k$ and q is a small constant.

Chapter 4

Thinking Like an Interval

In this chapter, we describe our novel and intuitive interval-centric distributed programming abstraction as a *unified model* for designing Time Independent (TI) and Time Dependent (TD) algorithms. This simplifies the user logic when designing algorithms over a temporal graph in a distributed environment. 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.

4.1 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. 4.2.1) in a *data-parallel* manner. We use *Bulk Synchronous Parallel (BSP)* execution [74], which alternates a *computation phase*, where the user logic executes, with a *communication phase*, where messages are bulk-transferred between vertices at a global barrier. These continue for several iterations till the application converges. Fig. 4.1 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 this 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 transformation function on the vertex state to create the message and its valid interval.

Once the *compute* and *scatter* logic execute for all the active vertices and their active intervals, the communication phase delivers messages to the destination vertices. The current

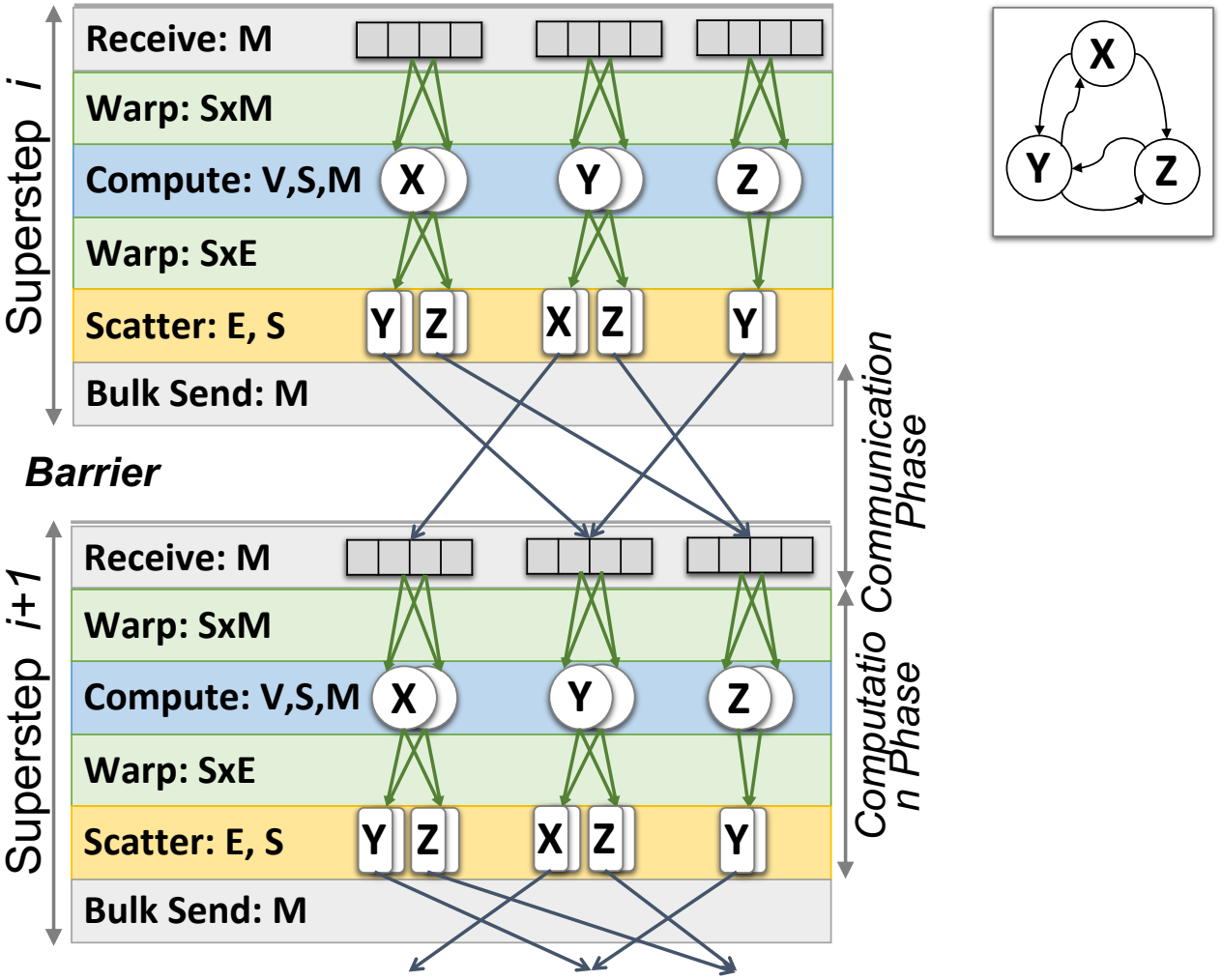


Figure 4.1: Supersteps and steps in ICM shown for the sample graph at the top right.

iteration (also called *superstep*) is finished, and the next iteration can start.

4.2 Dynamically Partitioned Vertex States

Vertices in ICM inherit *static information* from the temporal graph $\mathcal{G}$, and also maintain *dynamic states* for the vertex as part of 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 [1]. 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.

4.2.1 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* (discussed in Sec. 4.3) 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 [91, 123]. ICM stops when no vertices are activated by messages in an iteration.

[1] 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.

4.2.2 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) 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:

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

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 $M = s_k$.

```
1 void init(Vertex v) {
2    v.setState(v.interval, ∞);
3 }
4
5 void compute(Vertex v, Interval t, int vstate, Message[ ] msgs) {
6    if(getSuperstep() == 1 && isSource(v)) {
7       v.setState(v.interval, 0);
8       return;
9    }
10   minVal = ∞;
11   for(Message m : msgs)
12      minVal  = min(m.value, minVal);
13   if(minVal < vstate) v.setState(t, minVal);
14 }
15
16 Message scatter(Edge e, Interval t, int vstate){
17   int travelTime = e.getProp("travel-time");
18   int travelCost = e.getProp("travel-cost");
19   return new Message(e, new Interval(t.start + travelTime, ∞), vstate +
        travelCost);
20 }
```

Algorithm 4.1: Temporal SSSP using ICM

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. 4.3.

4.2.3 Temporal SSSP Example

Finding paths with the shortest travel time, distance or cost is a common problem in temporal graphs. The temporal *single source shortest path (SSSP)* [120] 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. 4.1, and illustrated in Fig. 4.2 for the interval graph from Fig. 1.1(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 cost of 0 for

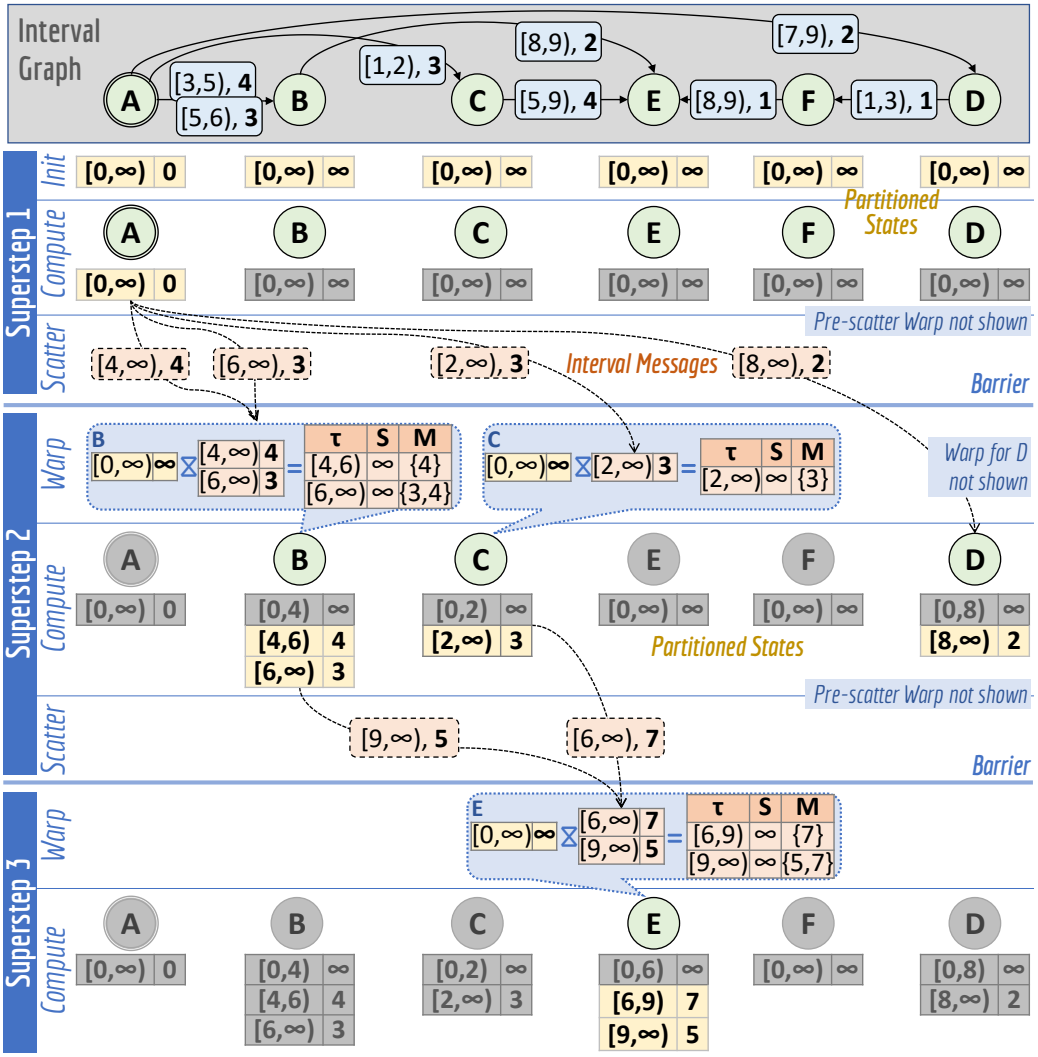


Figure 4.2: SSSP execution using ICM for the temporal graph from Fig. 1.1(a). A is the source. Travel time on an edge is 1.

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. 4.2, 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

```
1  void compute(Vertex v, int[] vState, Message[ ] msgs) {
2     if(getSuperstep() == 1 && isSource(v)) {
3        int[] state ← 0;
4        v.setState(state);
5        return;
6     }
7     int[] minVal ← ∞;
8     for(Message m : msgs) {
9        for(int timePoint :  v.getProp("lifespan")) {
10          if(timePoint >= m.timePoint)
11             minVal[timePoint]  = min(m.travelCost, minVal[timePoint]);
12       }
13    }
14    for(int timePoint :  v.getProp("lifespan")) {
15       if(minValue[timePoint] < vState[timePoint]) {
16          vState[timePoint] = minValue[timePoint] ;
17          for(Edge e : v.getEdges()) {
18             for(int eTimePoint :  e.getProp("lifespan")) {
19                if(eTimePoint >= timePoint) {
20                   int travelTime = e.getProp("travel-time", eTimePoint);
21                   int travelCost = e.getProp("travel-cost", eTimePoint);
22                   sendMessage( e.getTargetVertexId(),
23                      new Tuple2( eTimePoint+travelTime,
24                         vState[timePoint]+travelCost )
25                   );
26                }
27             }
28          }
29       }
30    }
31    v.setState(vState);
32    vertex.voteToHalt();
33 }
```

Algorithm 4.2: Temporal SSSP using VCM

$[3+1, \infty)$ 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 on the temporal graph; 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.

In contrast, we show the pseudo-code for implementing temporal SSSP directly using a vertex-centric computing model (VCM) in Alg. 4.2. As we can see, the lines of code that is required is much more, and can cause the execution time to be longer as well due to unnecessary executions of the vertices' compute function, leading to a longer execution time.

4.3 Time-warp

Adding time-intervals to *compute* and *scatter* is a novel temporal extension to Pregel [74] or GAS [72] models. It enables a unified distributed programming abstraction over temporal graphs. 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* operation 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

S

τ_s	S
[0,5)	s_1
[5,9)	s_2
[9,10)	s_3

M

τ_m	M
[0,4)	m_1
[2,7)	m_2
[5,7)	m_3
[5,9)	m_4
[9,10)	m_5

Time Join $\widetilde{\bowtie}^{t}_{S\times M}$

$\tau_t=\tau_s\cap\tau_m$	S	M
[0,4)	s_1	m_1
[2,5)	s_1	m_2
[5,7)	s_2	m_2
[5,7)	s_2	m_3
[5,9)	s_2	m_4
[9,10)	s_3	m_5

Time Warp $\mathbb{X}_{S\times M}$

TW	τ	S	$\mathbb{M}$
w_1	[0,2)	s_1	m_1
w_2	[2,4)	s_1	m_1, m_2
w_3	[4,5)	s_1	m_2
w_4	[5,7)	s_2	m_2, m_3, m_4
w_5	[7,9)	s_2	m_4
w_6	[9,10)	s_3	m_5

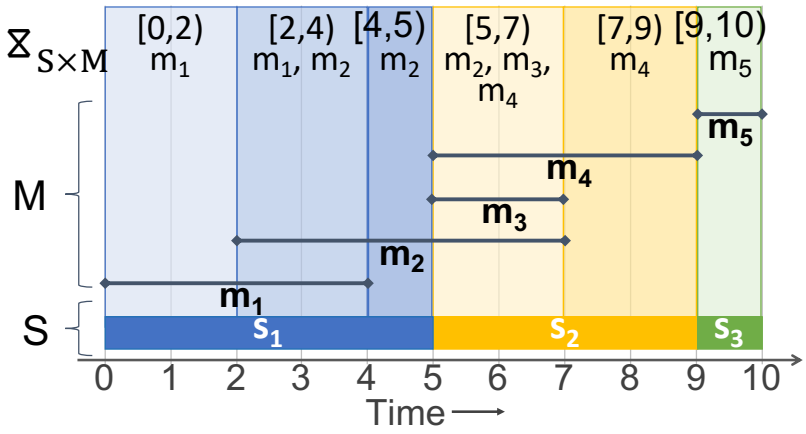


Figure 4.3: Time-warp operating on the partitioned states and input messages for an active vertex.

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 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

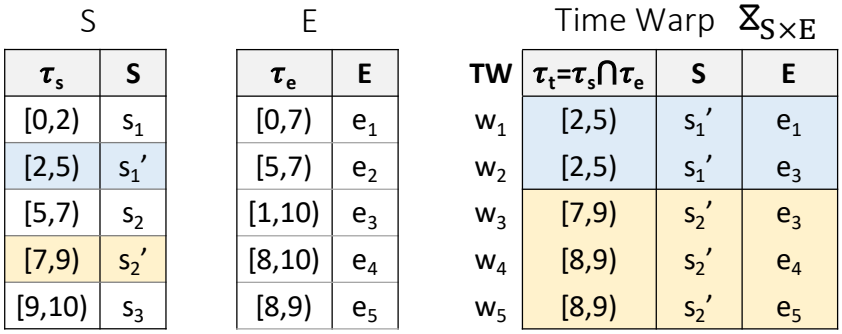


Figure 4.4: Pre-scatter time-warp operating on the *partitioned vertex states* (S) and the *out-edges* of the interval vertex (E). The partitioned states s_1' and s_2' were updated by `compute` in the current superstep and need to be propagated to the relevant out edges. Each row in the Time-warp will trigger a call to `scatter`.

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.

4.3.1 Detailed Warp Example

Fig. 4.3 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 [103] 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.

4.3.2 Formal Definition

Formally, *time-warp* ($\boxtimes_{S\times M}$) operates on two sets of tuples S (outer set) and M (inner set) both having 2-tuples with a time-interval and a value. The outer set S must be temporally partitioned. The *time-join* ($\bar{\bowtie}_{S\times M}$) operator [103] on the two sets is defined as:

$$S = \{\langle \tau_s, s\rangle\}$$
$$M = \{\langle \tau_m, m\rangle\}$$
$$\bar{\bowtie}^t_{S\times M} = \{\langle \tau_t, s_t, m_t\rangle \mid \langle \tau_s, s_t\rangle \in S \ \wedge \ \langle \tau_m, m_t\rangle \in M \ \wedge \ \tau_s \sqcap \tau_m \ \wedge \ \tau_t = \tau_s \cap \tau_m\}$$

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:

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

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.

4.3.3 Properties of Time-warp

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 $\langle \tau, s_j, \mathbb{M} \rangle \in \between_{S \times M}$ 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 $\langle \tau, s_j, \mathbb{M} \rangle \in \between_{S \times M}$, 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 $\langle \tau_j, s_j, \mathbb{M}_j \rangle, \langle \tau_k, s_k, \mathbb{M}_k \rangle \in \between_{S \times M}$ 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 $\langle \tau_j, s_j, \mathbb{M}_j \rangle, \langle \tau_k, s_k, \mathbb{M}_k \rangle \in \between_{S \times M}$ 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*.

4.3.4 Time-warp in Temporal SSSP Example

Continuing the earlier example, warp automatically enforces temporal constraints in the calls to *compute* and *scatter*. This makes the user code concise, correct, and prevents its unnecessary execution. 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 10–13 of Alg. 4.1).

E.g., when superstep 3 starts in Fig. 4.2, 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 comparable 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.*

Chapter 5

Temporal Graph Algorithms

In this chapter, we look at how various Time Independent (TI) and Time Dependent (TD) algorithms from literature can be designed using our unified ICM primitives.

5.1 Time-Independent Algorithms

We formulate ICM variants for 4 TI algorithms: *Breadth First Search (BFS)* [74], *Weakly Connected Component (WCC)* [123], *Strongly Connected Component (SCC)* [91] and *PageRank (PR)* [74]. As discussed before, TI algorithms behave such that the algorithm runs on each time-point or snapshot of the temporal graph independently. The Vertex-centric Computing Model (VCM) logic for these algorithms *can be reused as is* for the `compute` and `scatter` logic of ICM since the default behavior of ICM assigns appropriate intervals to the states and messages. The advantage of ICM is that it compute over adjacent time-points (intervals) in a single pass, and the output returned for different intervals of a vertex in an interval graph is to be interpreted separately for each time-point in that interval and for all vertex outputs of the graph at that time-point. E.g., Weakly Connected Components for vertices in the tiny graph in Fig. 4.1 is returned as A={[1,9):A}, B={[1,3):B, [3,6):A, [6,9): B}, C={[1,2):A, [2,9):C}, D={[1,7):D, [7,9):A}, E={[1,5):E, [5,8):C, [8,9):B}, and , F={[1,3):D, [3,9):F}, then vertices A, B, C, D, E and F at time-point 8 belong to components A, B, C, A, B, and F respectively.

5.1.1 Breath First Search (BFS)

The BFS algorithm is a popular traversal algorithm. It starts from some user-defined source vertex (also referred to as the root) and for each superstep s_i, explores all its s_i hop neighbors before moving on to vertices at increasing hop distance. This may repeat till all vertices in the graph are visited, or for a fixed hops (or depth) from the source vertex. The time-independent variant of this algorithm identifies hop distance for sub-intervals of a vertex from user-defined

```
1 void init(Vertex v) {
2    v.setState(v.interval, ∞);
3 }
4
5 void compute(Vertex v, Interval t, long currentDistance, Message[ ] msgs) {
6    if(getSuperstep() == 1 && isSource(v)) {
7       v.setState(v.interval, 0);
8       return;
9    }
10   long candidateDistance = ∞;
11   for(Message m : msgs) {
12      candidateDistance  = min(m.value, currentDistance);
13   }
14   if(candidateDistance < currentDistance)
15       v.setState(t, candidateDistance);
16 }
17
18 Message[] scatter(Edge e, Interval t, long currentDistance){
19   return new Message(e, t, currentDistance+1);
20 }
```

Algorithm 5.1: Time-Independent Breath First Search using ICM

source vertex. Computed hop distance is applicable to all time-points in that sub-interval.

The ICM algorithm, shown in Alg. 5.1, is identical to the vertex-centric logic for BFS on a static graph [74]. In the 1^{st} superstep, each vertex checks if it is the source (line 3) and if so, sets its hop distance as 0 (line 7) for entire vertex lifetime. All non-source vertices set their hop distance as ∞ (line 2). In supersteps>1, each partitioned interval vertex which receives message, computes the minimum candidate distance from all these messages (line 11-13) and updates its current distance to candidate distance post comparison (line 14-15) for that sub-interval. It additionally, shares this updated hop distance with its temporal out-neighbors (line 19) via interval message. Computation halts when hop distance for no partitioned interval is updated and no messages are in-flight, at which point all sub-intervals during which non-source vertices are reachable from source vertex have been explored and labeled.

5.1.2 Weakly Connected Components (WCC)

The WCC algorithm groups the vertices in the graph into components such that there exists an undirected path between every pair of vertices in the component. All vertices in a WCC are labeled with the component ID they belong to, e.g., the vertex with the smallest ID in that component. The time-independent variant of this algorithm operates on an *undirected* temporal graph, and labels sub-intervals of a vertex with the component ID; this ID applies

```
1 void init(Vertex v) {
2    return;
3 }
4
5 void compute(Vertex v, Interval t, long componentId, Message[ ] msgs) {
6    if(getSuperstep() == 1) {
7       v.setState(v.interval, v.id);
8       return;
9    }
10   minComponentId = ∞;
11   for(Message m : msgs) {
12      minComponentId  = min(m.value, minComponentId);
13   }
14   if(minComponentId < componentId) { v.setState(t, minComponentId); }
15 }
16
17 Message[] scatter(Edge e, Interval t, long componentId){
18   return new Message(e, t, componentId);
19 }
```

Algorithm 5.2: Time-Independent Weakly Connected Components using ICM

to all time-points in that interval. So, for each time point, all vertices that have the same component ID are part of the same WCC.

The ICM algorithm, shown in Alg. 5.2, in 1^{st} superstep, every vertex updates (line 7) its component ID to its vertex ID for its entire lifespan and propagates (line 18) it to its temporal out-neighbors. In future supersteps, each partitioned vertex interval picks the smallest ID from the incoming interval messages (lines 11-14) and propagates it. ICM's warp automatically divides the vertex and edge lifespans into parts before *compute* and *scatter*, ensuring that the relevant minimum vertex ID is correctly passed to temporally overlapping intervals and spatially connected vertices that form a component. ICM lets us find components common to multiple adjacent time-points in one pass, rather than on each snapshot separately. Further, if multiple WCCs exist, all are found in a single pass as well.

5.1.3 Strongly Connected Components (SCC)

The time-independent variant of SCC algorithm operating on a directed temporal graph, groups vertices into distinct components for a sub-interval, such that every vertex for each time-point during that sub-interval is reachable from every other vertex assigned to the same component through a directed path. Each such interval component is uniquely labeled using the smallest vertex identifier associated with a member vertex.

Like it static VCM counterpart [91], ICM's time-independent variant, shown in algorithm 5.3,

also computes SCC using a MasterCompute model having four computation phases (Algorithm 5.3). MasterCompute permits centralized computation prior to every superstep, and its output will be available to all workers before computation is triggered for any vertex.The compute and scatter logic for each of the phases is shown in Algorithm 5.4.

1. The TRANSPOSE phase takes two supersteps. Each vertex first propagates its vertex ID (line 52) to all its out-neighbors. Next, on every vertex, for each received message, an in-edge from the current vertex to the source vertex (line 15-16) is created.

2. The TRIMMING phase takes one superstep. Each vertex with no out-edges and/or no in-edges during some partitioned interval in its entire lifespan (if any), assigns its vertex ID as its component ID for the sub-interval and marked that sub-interval as converged (line 17-18). Converged partition intervals for a vertex ignore all subsequent messages (line 8).

3. The FORWARD phase is identical to WCC. Here, each vertex sets its component ID to its vertex ID for all active sub-intervals and propagates it to its temporal out-neighbors. Each partitioned interval vertex, assigns itself the smallest component ID it has received for the sub-interval until convergence.

4. The BACKWARD phase is split into two sub-phases: Backward-Start and Backward-Rest. The Backward-Start sub-phase takes one superstep. For each vertex, every partitioned interval whose component ID equals to its vertex ID (line 32), propagates its component ID to its temporal in-neighbors computed in the TRANSOPOSE phase, and marks itself as converged (line 34) for that sub-interval. In the Backward-Rest sub-phase, each partitioned interval vertex receiving a component ID that matches its current component ID (line 39), propagates its ID to its temporal in-neighbors and marks itself as converged (line 42) for that sub-interval. The MasterCompute logic (shown in Alg. 5.3) sets the computation phase to Forward Phase. Computation halts when all interval vertices have marked themselves deactivated.

The Alg. 5.4 repeats the Trimming, Forward, Backward-Start and Backward-Rest phases, each time detecting and removing one or more strongly connected components from the graph. It terminates when all vertices have converged.

5.1.4 PageRank (PR)

PageRank (PR) [81] is a classic centrality algorithm for identifying the importance of web pages (vertices) that link to each other (edges) in a web graph. We design a time-independent ICM

```
enum Phases = {"TRANSPOSE", "TRIMMING", "FORWARD", "BACKWARD_START",
    "BACKWARD_REST"};
enum scatterDirection = {"IN", "OUT", "BOTH"};
GLOBAL currPhase , vertexUpdated , scatterDirection;

void MasterCompute() {
   if (getSuperstep() == 1) {
      currPhase == "TRANSPOSE";
      scatterDirection = "OUT";
   } else {
      switch (currPhase) {
         case "TRANSPOSE":
            currPhase == "TRIMMING";
            break;

         case "TRIMMING":
            currPhase == "FORWARD";
            break;

         case "FORWARD":
            if (!vertexUpdated) {
               currPhase == "BACKWARD_START";
               scatterDirection = "IN";
            } break;

         case "BACKWARD_START":
            currPhase == "BACKWARD_REST";
            break;

         case "BACKWARD_REST":
            if (!vertexUpdated) {
               currPhase == "TRIMMING";
               scatterDirection = "OUT";
            } break;
      }
   }
}
```

Algorithm 5.3: MasterCompute for Time-Independent Strongly Connected Components using ICM

```
GLOBAL currPhase , vertexUpdated;
void init(Vertex v) {
   activateInterval(v, t);
}

void compute(Vertex v, Interval t, long componentId, Message[ ] msgs) {
   if(isActive(v, t)) {
      switch(currPhase) {
         case "TRANSPOSE":
            v.setState(v.interval, v.id);
            break;

         case "TRIMMING":
            for(Message m : msgs)
               v.createInEdge(m.interval, m.value);
            if(v.outEdgeCount == 0 || v.inEdgeCount == 0 )
               deactivateInterval(v, t); return;
            v.setState(t, v.id);
            break;

         case "FORWARD":
            minComponentId = ∞;
            for(Message m : msgs)
               minComponentId  = min(m.value, minComponentId);
            if(minComponentId < componentId) {
               v.setState(t, minComponentId);
               vertexUpdated = TRUE;
            } break;

         case "BACKWARD_START":
            if(v.id == componentId) {
               v.setState(t, componentId);
               deactivateInterval(v, t);
            } break;

         case "BACKWARD_REST":
            for(Message m : msgs) {
               if(m.value == componentId) {
                  v.setState(t, componentId);
                  vertexUpdated = TRUE;
                  deactivateInterval(v, t);
                  break;
               }
            }
      }
   }
   return;
}

Message[] scatter(Edge e, Interval t, long componentId){
   return new Message(e, t, componentId);
}
```

Algorithm 5.4: Compute and Scatter for Time-Independent Strongly Connected Components using ICM

algorithm for it where the rank for each vertex at each time-point in its lifespan is calculated independently, based on the state of the graph at that time-point.

In Alg. 5.5's *compute* method, we iterate through each *time-point* in an interval to update the PR for that point based on the partial PR sum from the messages. Similarly, in *scatter*, we send a message to the sink vertex for each edge, with the partial PRs for each time-point in its interval. This repeats iteratively for a fixed number of supersteps (10, in our experiments), or by testing for convergence using a residual threshold. *numVertices* and *numEdges* are built-in helper functions.

Other than iterating through each time-point (lines 8 and 14), the logic is similar to a vertex-centric algorithm on a static graph [74]. ICM *automatically exposes temporal parallelism for each time-point* – after the first superstep, all states have been partitioned to individual points on whom *compute* (or *scatter*) is called concurrently. Using VCM, one has to run the algorithm separately for each snapshot.

```
1  void init(Vertex v) {
2     for(TimePoint p : v.interval) {
3        v.setState(p, 1/numVertices(p))
4     }
5  }
6
7  void compute(Vertex v, Interval t, float vstate, Message[ ] msgs) {
8     for(Message m : msgs)  {
9        sum = sum + m.value;
10    }
11    for(TimePoint p : t) {
12       v.setState(p, 0.15/numVertices(p)+0.85*sum);
13    }
14 }
15
16 Message[] scatter(Edge e, Interval t, float vstate){
17    Message[] msgs;
18    for(TimePoint p : t) {
19       msgs.add(new Message(e, p, vstate/numEdges(p)));
20    } return msgs;
21 }
```

Algorithm 5.5: Time-Independent PageRank using ICM

5.2 Time-Dependent 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)* [120], *Fastest Arrival Time (FAST)* [120], *Latest Departure time (LD)* [120], *Reachability (RH)* [119] and *Time-Minimum Spanning Tree (TMST)* [46], can be solved with minimal changes to the temporal SSSP algorithm we introduced earlier. As discussed before, the behavior of TD algorithms is to span across intervals during execution, and the results returned are valid for the associated time ranges, as was shown for Temporal SSSP in Fig. 4.2.

5.2.1 Earliest Arrival Time (EAT)

Earliest Arrival Time (EAT) (or Foremost Journey [121]) computes the earliest time one can reach a target vertex y from source x using a time-respecting path [120]. Here, we are only interested in the earliest time (unique for each vertex) at which we can reach a vertex and not in subsequent intervals of arrival or cost of travel. ICM program for computing EAT (shown in Alg. 5.6) can be obtained by replacing just the *travel cost* with *vertex arrival time* in Alg. 4.1 (line 17). Note that temporal bounds are automatically enforced by TimeWarp.

```
1 void init(Vertex v) {
2    v.setState(v.interval, ∞);
3 }
4 void compute(Vertex v, Interval t, long currentArrivalTime, Message[ ]
     msgs) {
5    if(getSuperstep() == 1 && isSource(v))
6       v.setState(v.interval, 0); return;
7    long candidateEarliestArrival = ∞;
8    for(Message m : msgs) {
9       if(m.value < candidateEarliestArrival)
10         candidateEarliestArrival = m.value;
11   }
12   if(candidateEarliestArrival < currentArrivalTime)
13      v.setState(t, candidateEarliestArrival);
14 }
15 Message scatter(Edge e, Interval t, long earliestArrivalTime){
16   int travelTime = e.getProp("travel-time");
17   long  arrivalTime = max(t.start, earliestArrivalTime) + travelTime ;
18   return new Message(e, new Interval(arrivalTime, ∞),  arrivalTime);
19 }
```

Algorithm 5.6: Time-Dependent Earliest Arrival Time using ICM

5.2.2 Fastest Travel Time (FAST)

Fastest Travel Time (FAST) computes the minimum time in which one goes from source vertex x to any other reachable vertex y via a time-respecting path [120]. Like EAT, fastest travel time is unique for each source-destination pair across all candidate time-respecting paths, however the fastest path themselves are not unique. E.g., For a given source-destination pair, more than one path departing (or arriving) from source (on destination) at different time-points can result in effectively same elapsed time. In context of a transit network, the goal is to minimize total travel time, which includes time spend on-road and waiting time. Sometimes reducing the time spend on-road can be more economic than arriving at the destination in shortest amount of time, i.e., minimizing on-road time at the cost of increased total travel time [67]. We highlight that, by excluding the *waitingTime* (line 21) from *totalTravelTime* (line 23) computation in Alg. 5.7, we can minimize the on-road time (as computed by MORT [67]) instead of total elapsed time. As with SSSP and EAT, the temporal bounds are automatically enforced by TimeWarp.

```
1 void init(Vertex v) {
2    v.setState(v.interval, new Pair<long, long>(∞, ∞));
3 }
4
5 void compute(Vertex v, Interval t, Pair<long, long> vstate, Message[ ]
      msgs) {
6    if(getSuperstep() == 1 && isSource(v))
7       v.setState(v.interval, new Pair<long, long>(0, 0));
8       return;
9    }
10   long candidateFastestTravelTime = ∞, candidateArrivalTime = ∞;
11   for(Message m : msgs) {
12      if(m.value._2 < candidateFastestTravelTime)
13         candidateArrivalTime = m.value._1;
14         candidateFastestTravelTime = m.value._2;
15   }
16   if(candidateFastestTravelTime < vstate.fastestTravelTime)
17      v.setState(t, new Pair<long, long>(candidateArrivalTime,
           candidateFastestTravelTime));
18 }
19
20 Message scatter(Edge e, Interval t, Pair<long, long> vstate){
21   long waitingTime = t.start - vstate.arrivalTime;
22   if(isSource(e.getSRC())) { waitingTime = 0; }
```

```
23    long totalTravelTime = vstate.fastestTravelTime + waitingTime +
          e.getProp("travel-time");
24    long arrivalTime = t.start + e.getProp("travel-time");
25    return new Message(e, new Interval(arrivalTime, ∞),  new Pair<long,
          long>(arrivalTime, totalTravelTime) );
26 }
```

Algorithm 5.7: Time-Dependent Fastest Travel Time using ICM

5.2.3 Latest Departure (LD)

Latest Departure (LD) lets one compute the largest time-point by which one must leave from vertex y in order to reach destination vertex x by given time (referred to as *Latest_Arrival_Time*) via a time-respecting path. Both, the destination vertex and *Latest_Arrival_Time* are user-specified. Like EAT and FAST, here we are interested in computed the latest departure time (which is unique for each vertex) and not the actual temporal path, however the algorithm can be trivially extended to accommodate it. Unlike Temporal SSSP, LD operates on an inverted graph i.e. source and destination for all edges are swapped. However, edge lifespan remain unaltered. Such a inverted graph allows LD to traverse from sink to source, in space and time. Temporal bounds are automatically enforced by setting message interval to $[-\infty, departureTime)$.

```
1 void init(Vertex v) {
2    v.setState(v.interval, −∞);
3    if(isDestination(v) && LATEST_ARRIVAL_TIME < v.start)
4          haltComputation();
5    }
6 }
7
8 void compute(Vertex v, Interval t, long currentDepartureTime, Message[ ]
      msgs) {
9    if(getSuperstep() == 1 && isDestination(v))
10      long latestDeparture = min(LATEST_ARRIVAL_TIME, v.end-1);
11      v.setState(new Interval(v.start, latestDeparture), latestDeparture);
12      return;
13   }
14   long candidateLatestDeparture= −∞;
15   for(Message m : msgs)
16      candidateLatestDeparture = max(m.value, candidateLatestDeparture);
17   if(candidateLatestDeparture > currentDepartureTime)
18      v.setState(t, candidateLatestDeparture);
19   }
```

```
20
21 Message scatter(Edge e, Interval t, long latestDepartureTime){
22    int travelTime = e.getProp("travel-time");
23    long departureTime = min(t.end-1, latestDepartureTime - travelTime) ;
24    if(departureTime >= t.start)
25       return new Message(e, new Interval(−∞, departureTime),
             departureTime );
26    return null;
27 }
```

Algorithm 5.8: Time-Dependent Latest Departure using ICM

5.2.4 Time-Minimized Spanning Tree (TMST)

Time-Minimized Spanning Tree (TMST) constitutes the computation of a spanning tree which results in the fastest spread from a source vertex to each temporally reachable vertex. In the context of social networks, the spread corresponds to information dissemination, which can be important to social-media marketing campaigns or to study of how fake news spreads [46]. To find the TMST from a given source (Alg. 5.9), we add the *parent vertex ID* to the state and the message value (lines 12 and 17) of the Temporal SSSP algorithm in Alg. 4.1, in addition to replacing *travel cost* with *arrival time*, to rebuild the spanning tree.

```
1 void init(Vertex v) {
2    v.setState(v.interval, new Pair<long, long>(∞, -1));
3 }
4
5 void compute(Vertex v, Interval t, Pair<long, long> vstate, Message[ ]
     msgs) {
6    if(getSuperstep() == 1 && isRoot(v))
7       v.setState(v.interval, new Pair<long, long>(0, v.getId()));
8       return;
9    }
10   long candidateEarliestArrival = ∞, candidateParentID = -1;
11   for(Message m : msgs) {
12      if(m.value._1 < candidateEarliestArrival) {
13         candidateEarliestArrival = m.value._1;
14         candidateParentID = m.value._2;
15      }
16   }
17   if(candidateEarliestArrival < vstate.earliestTime)
18      v.setState(t, new Pair<long, long>(candidateEarliestArrival,
```

```
            candidateParentID));
19 }
20
21 Message scatter(Edge e, Interval t, Pair<long, long> vstate){
22    int travelTime = e.getProp("travel-time");
23    long  arrivalTime = max(t.start, earliestArrivalTime) + travelTime ;
24    return new Message(e, new Interval( arrivalTime,∞), new Pair<long,
         long>(arrivalTime, e.getSRC()));
25 }
```

Algorithm 5.9: Time-Minimized Spanning Tree using ICM

5.2.5 Temporal Reachability (RH)

Temporal Reachability (RH) [119] checks if a valid temporal path exists between a source and a sink vertex. If the shortest temporal path from source to sink vertex is k-hops in length, the ICM algorithm shown in Algorithm 5.10 labels every temporally reachable vertex within a k-hop neighborhood of the source vertex, including the sink vertex as "reachable" and preemptively halts in the k^{th} superstep. If the sink vertex is temporally unreachable, the algorithm labels every temporally reachable vertex and only halts when each of them is marked "reachable". The sink vertex in this case remains marked as "unreachable". `haltComputation()` is a method, if invoked, will halt computation post completion of current superstep, even if there are messages in the system or vertices that have not voted to halt.

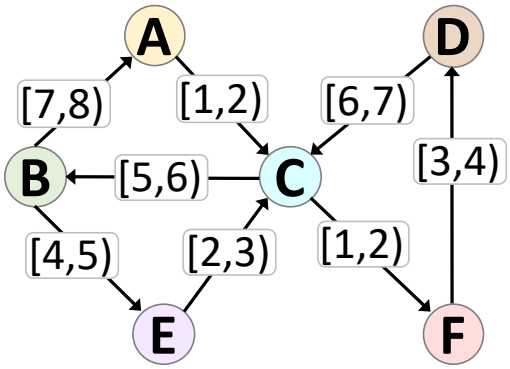
```
1 void init(Vertex v) {
2    v.setState(v.interval, "unreachable");
3 }
4
5 void compute(Vertex v, Interval t, enum vstate, Message[ ] msgs) {
6    if(getSuperstep() == 1 && isSource(v))
7       v.setState(t, "reachable");
8       return;
9    }
10   for(Message m : msgs) {
11      if(m.value == "reachable") {
12         v.setState(t, "reachable");
13         if(isSink(v)) { haltComputation(); }
14         break;
15      }
16   }
17 }
```

```
Message scatter(Edge e, Interval t, enum vstate){
    int travelTime = e.getProp("travel-time");
    return new Message(e, new Interval(t.start + travelTime, ∞),
        "reachable");
}
```

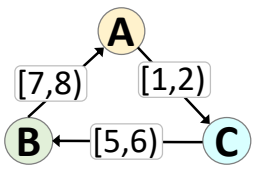
Algorithm 5.10: Temporal Reachability using ICM

5.2.6 Temporal Triangle Count (TC)

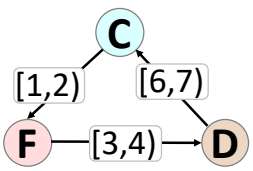
Temporal Cycles indicate the presence of feedback loops and naturally appear in many real-world problems such as stock trading, financial networks, social networks and biological networks. Temporal Triangle Count [64, 82] involves counting temporal cycles (interaction must respect temporal order) of length 3, which starts and ends at the same vertex. Fig. 5.1(b) contains examples of valid temporal cycles A-C-B-A and C-F-D-C from example graph shown in Fig. 5.1(a). However, triangles E-C-B-E shown in Fig 5.1(c) is not an invalid temporal triangle as temporal order is not respected.

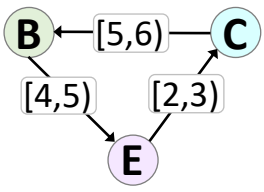


(a) Interval Graph



(b) Valid Temporal Triangle





(c) Invalid Temporal Triangle

Figure 5.1: Example Temporal Graph

For triangle counting, the Algorithm 5.11 computes its strict two-hop temporal neighbor-

```
1 void init(Vertex v) {
2    return;
3 }
4
5 void compute(Vertex v, Interval t,  Long[] nbrs, Message[ ] msgs) {
6    Long[] nbrs;
7    if(getSuperstep() == 1) {
8       nbrs.add(v.id);
9    } else if(getSuperstep() == 2) {
10      for(Message m : msgs) {
11         nbrs.addAll(m.value());
12      }
13   } else if(getSuperstep() == 3) {
14      long count=0;
15      for(Message m : msgs) {
16         for(Long nbr : m.value) {
17            if(v.getIntervalEdge( t , nbr)!=null) { ++count; }
18         }
19      } nbrs.add(count);
20   } v.setState(t, nbrs);
21 }
22
23 Message scatter(Edge e, Interval t, Long[] nbrs){
24    if(getSuperstep()<3)
25       return new Message(e, new Interval(t.start + 1, ∞), nbrs);
26 }
```

Algorithm 5.11: Temporal Triangle Count using ICM

hood and checks if any of its adjacent vertices are part of the neighborhood. If so, it increments the number of triangles identified. Temporal Order among vertices is implicitly ensured while collecting 2-hop neighborhood due to TimeWarp.

5.2.7 Local Clustering Coefficient (LCC)

Local Clustering Coefficient (LCC) of a vertex quantifies how close its neighborhood is to being a temporal clique [45] and is often used in literature for detecting outliers and for role discovery [124]. We define a *temporal wedge* as a pair of time-respecting edges that share exactly one common vertex, and the common vertex is called the *center* of the wedge. The other vertex incident on first and second edge forming the *temporal wedge* is termed as the *head* and *tail* respectively. A temporal wedge is called *closed* if an edge between *tail* and *head* of the wedge exists and results in formation of a *temporal triangle*. The time-dependent clustering coefficient of a center vertex u is then defined as the fraction of *closed temporal wedges* present in the

temporal graph to *temporal wedges* centered at vertex u :

$$LCC(u) = \frac{CW(u)}{W(u)} \tag{5.1}$$

where, $CW(u)$ denotes the number of *closed temporal wedges* centered at vertex u and $W(u)$ denotes the number of *temporal wedges* centered at vertex u .

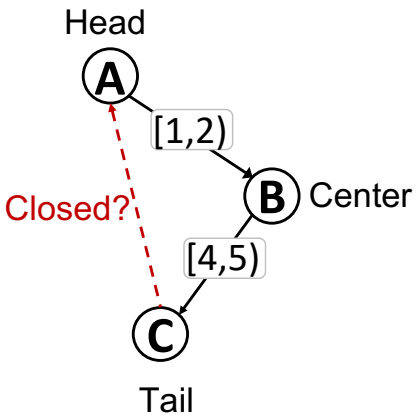


Figure 5.2: Anatomy of Temporal Wedge

As shown in Alg. 5.12, LCC operates in four phases : In phase 1, each vertex accumulates its one-hop neighbors. In phase 2, each vertex computes all *temporal wedges* for which it is the center vertex and forwards each such *temporal wedge* to its respective *tail* vertex. In phase 3, each *tail* vertex receive the wedge from *center* vertex and inspects its out-edges to determine if the candidate *temporal wedge* can be *closed*. If the wedge is indeed closed, it messages value 1 to the center of the wedge otherwise it ignores the wedge. Finally, in Phase 4, the *center* vertex accumulates all values received from respective *tails* and computes its local clustering-coefficient using equation 5.1.

```
1 void init(Vertex v) {
2    v.createProperty("W", 0);
3    v.createProperty("LCC", -1.0);
4    return;
5 }
6
7 /*
8  * Here, Interval Message Payload is of type Tuple4. It encapsulates four
       fields :
9  * 1. Interval
10 * 2. Head Vertex of a Wedge : Type Long
```

```
11 * 3. Center Vertex of a Wedge : Type Long
12 * 4. Tail Vertex of a Wedge : Type Long
13 *
14 * In order to access Head Vertex from message payload, we invoke _2()
     method. Similarly, interval, center and tail can be accessed using
     _1(), _3() and _4() resp.
15 */
16 void compute(Vertex v, Interval t,  Map<Long, Tuple4<Interval, Long,
     Long>[]> vState, Message[ ] msgs) {
17   Map<Long, Tuple4<Interval, Long, Long>[]> wedgeMap;
18   if (getSuperstep() == 1) {
19      v.setState(v.interval, wedgeMap);
20   } else if (getSuperstep() == 2) {
21      for(Message m : msgs) {
22         v.createInEdge(m.value._1, m.value._2);
23         for(Pair<edgeInterval,nbrID >: v.getOutEdges(m.value._1.start)){
24            if(!wedgeMap.contains(nbrID)) {
25               wedgeMap.put(new Tuple4<>(edgeInterval, m.value._2, v.id,
                    nbrID));
26            } else {
27               Tuple4<Interval, Long, Long>[] computedWedge =
                    wedgeMap.getValue(nbrID);
28               computedWedge.add(new Tuple4<>(edgeInterval, m.value._2,
                    v.id, nbrID));
29               wedgeMap.update(nbrID, computedWedge);
30            }
31         }
32      }
33      v.setProperty("W", wedgeMap.size());
34      v.setState(v.interval, wedgeMap);
35   } else if (getSuperstep() == 3) {
36         for(Message m : msgs) {
37               Pair<edgeInterval, nbrID > edge
38                  = v.getOutEdge(t, m.value_2);
39               if(edge!=null) {
40                  if(!wedgeMap.contains(m.value_4)) {
41                     wedgeMap.put(new Tuple4<>(edgeInterval, m.value._2,
                          m.value_3, m.value_4));
42                  } else {
43                     Tuple4<Interval, Long, Long>[] computedWedge =
                          wedgeMap.getValue(m.value_4);
44                     computedWedge.add(new Tuple4<>(edgeInterval,
```

```
                        m.value._2, m.value_3, m.value_4));
45                    wedgeMap.update(nbrID, computedWedge);
46                 }
47              }
48           }
49        } v.setState(v.interval, wedgeMap);
50    }  else if (getSuperstep() == 4) {
51        long closedWedges = 0, TotalWedges= v.getProperty("W");
52        for(Message m : msgs)
53           closedWedges += 1;
54        if(TotalWedges>0)
55           v.setProperty("LCC", closedWedges/TotalWedges);
56    } else {
57       haltComputation();
58    }
59 }
60
61 Message scatter(Edge e, Interval t, Map<Long, Tuple4<Interval, Long,
      Long>[]> vState){
62    if(getSuperstep() == 1) {
63       int travelTime = e.getProp("travel-time");
64       return new Message(e, new Interval(t.start + travelTime, ∞),
65          new Tuple4<>(t, e.getSRC(), -1, -1));
66    } else {
67       Tuple4<Interval, Long, Long, Long>[] computedWedge =
            vState.get(e.getDST());
68       if(wedge!=null) {
69          for(Tuple4<Interval, Long, Long, Long> wedge : computedWedge) {
70             sendMessage(e, wedge._1, wedge);
71          } return null;
72       }
73    }
74 }
```

Algorithm 5.12: Compute and Scatter for Time-Dependent Local Clustering Coefficient using ICM

Chapter 6

The Graphite Platform

GRAPHITE [1] is our implementation of the proposed interval-centric compute model (ICM), built as a layer on top of and with extensions to *Apache Giraph 1.3.0*, a popular Java-based open-source distributed graph processing platform that implements the vertex-centric computing model (VCM).

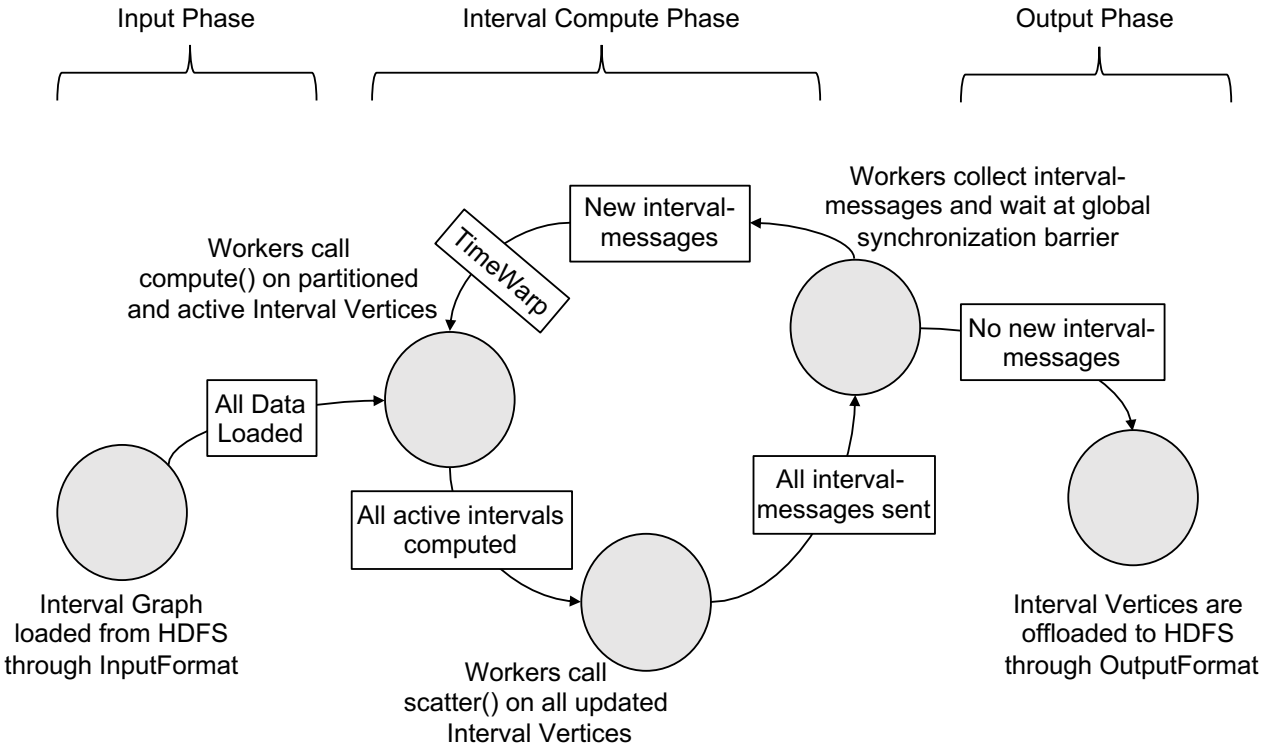


Figure 6.1: GRAPHITE Job Lifecycle, extending from Apache Giraph [75]

[1] https://github.com/dream-lab/graphite

6.1 Architecture

The Giraph Architecture consists of *Workers* that hold the partitioned vertices of the graph, along with their adjacency list, and execute the user logic in a data-parallel and distributed manner with one or more threads per worker. A *Master* is used for synchronizing the BSP barrier at the end of each superstep, decides if the application has terminated, and initiates the next superstep on the workers. This basic design is extended by GRAPHITE and described in more detail below.

6.1.1 Worker Design

Users define their temporal graph algorithm using the ICM primitives, and implement the *compute* and *scatter* logic in Java using interfaces provided by GRAPHITE. The GRAPHITE platform in turn orchestrates the execution of these functions as part of the `compute` method of Giraph. This is illustrated in Fig. 6.1. Each worker determines the active vertices that have received a message, loops through each of them, and invokes the user-defined logic on each active partitioned state interval in it.

The *compute* logic operates on a vertex and its incoming messages, and can update the vertex's current state for an sub-interval contained during its lifespan. To ensure synchronization-free parallel operation, GRAPHITE temporally aligns and groups messages along maximal sub-intervals for each interval-vertex, such that the intervals are disjoint. To find these sub-intervals, GRAPHITE makes use of *time-warp* (Section 4.3), whose implementation is described in the next section. Once these sub-intervals have been identified, GRAPHITE can invoke the compute function on them in a *data-parallel* manner [1]. The prior state for the vertex sub-interval and the grouped incoming messages for this sub-interval are passed as parameters to the function, and the logic can update the state for that sub-interval.

When the compute logic returns, GRAPHITE invokes the *scatter* logic for all sub-intervals for that vertex whose state was updated, once for each out-going edge, in a data-parallel manner (Section 4.2.2). The logic has read-only access to the associated interval state and to the edge properties. For each invocation on a edge's sub-interval, the scatter logic can return zero or more interval messages to be sent to the target vertex. If it returns no messages, then this invocation does not cause any messages to be sent; otherwise we send the messages to the destination as a Giraph's message and made it available at the start of the next superstep to target vertex. Scatter for a sub-interval can be invoked concurrently with the compute for another sub-interval of the same or for a different vertex, and/or with the scatter for other

[1] Parallelism is restricted by the available compute threads on a worker

	Aggregates Handled	Time-Complexity
Basic [108]	All	$\mathcal{O}(n^2)$
Aggregation Tree [60]	All	$\mathcal{O}(n^2)$
Balanced Tree [79]	Count/Sum/Average	$\mathcal{O}(n \log n)$
Sort-Merge Aggregation [79]	All	$\mathcal{O}(n \log n)$
SB-Tree [52]	All	$\mathcal{O}(n \log n)$
Disjoint Interval Partitioning [16]	All	$\mathcal{O}(n \log n)$

Table 6.1: Comparison of Temporal Aggregation Algorithms (GRAPHITE's default algorithm highlighted)

out-edges for the same or a different sub-interval. This allows computation to overlap with communication.

6.1.2 Time-warp

Time-warp is a type of temporal aggregation. We implement it using a merge-sort algorithm [79] (see Appendix A for the pseudo-code and detailed example). 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.

6.1.2.1 Other Aggregation algorithms

A lot of work has been done on temporal aggregation algorithms which allow data to be grouped along the time dimension [12]. The earliest work aimed at efficient processing of temporal aggregates is by Tuma [108]. Key works in this direction include the aggregation tree algorithm [60], SB-Tree [52] and the recently proposed Disjoint-Interval Partitioning [16]. We note that research on temporal aggregation algorithms is orthogonal to our work and to leverage benefits of on-going advancements, GRAPHITE permits users to replace the default sort-merge aggregation algorithm with a custom implementation by using the `graphite.warpOperationClass` property.

6.1.3 Messaging, Global Co-ordination and Termination

During a superstep, interval-messages are serialized, batched and sent asynchronously using communication threads which are always running in the background, concurrently, enabling computation and communication to overlap. At the end of each superstep, workers wait for all outgoing messages to be delivered before blocking on a global barrier. Global synchronization is coordinated by the master using Apache ZooKeeper [47], as provided by Giraph. ZooKeeper

supports high availability via use of quorum, where clients can access information from another peer server if its first call fails. By default, GRAPHITE uses an ensemble of 3 zookeeper servers.

Unlike Giraph, all vertices implicitly mark themselves as inactive at the end of each superstep (see Sec. 6.3.5). The computation halts when all interval vertices are inactive and no messages are in-transit, and otherwise the master instructs all workers to proceed to next superstep. When the computation halts, the master may instruct each worker to save the state for its portion of the interval graph to HDFS.

6.1.4 Master Compute

Just like in Giraph, `MasterCompute` is an optional stage that performs centralized single-threaded computation in GRAPHITE. The logic for this can be registered by the user with the `giraph.masterComputeClass` property. MasterCompute is executed on the master once, at the beginning of each superstep. Users can use this to change the computation classes to be used for different phases during runtime, or perform some global computation whose outcome is made available to all workers before the start of the next superstep. Additionally, GRAPHITE also retains the `preSuperstep()` and `postSuperstep()` functions of Giraph, which can be used for executing user logic once, before or after all the computation completes for any (or all) vertices in a partition.

6.1.5 Composability

Often graph analytics is part of a larger pipeline which consists of extracting graph from raw data, followed and/or preceded by data wrangling, carrying out graph computation, and analyzing the result. In contemporary graph processing systems, these pipelines are composed using a combination of data-processing (e.g. Hadoop [26]) and graph-processing systems (e.g. Giraph [1]), with data transfered between them using a distributed file system like HDFS. However, such pipelines are more complex and leads to inefficient performance due to large-scale disk-based data movement across framework boundaries. Moreover, the phase-based algorithms like Algo. 5.3 and Algo. 5.12, users are burdened with designing bespoke phase-switching mechanism using conditional statements on the supersteps count in MasterCompute. These also restrict chaining of multiple existing temporal graph analytics. Further, users need to take care to ensure compatibility between the output and input message types at phase boundaries.

We address these performance and usability short-comings by designing a *pipeline composability framework* in GRAPHITE that builds upon the basic compute and scatter functions of ICM. Code is written as *stages*, which can be composed together to form a *pipeline* (see Figure 6.2). Each *stage* is a ICM program, which may be configured to iterate for a

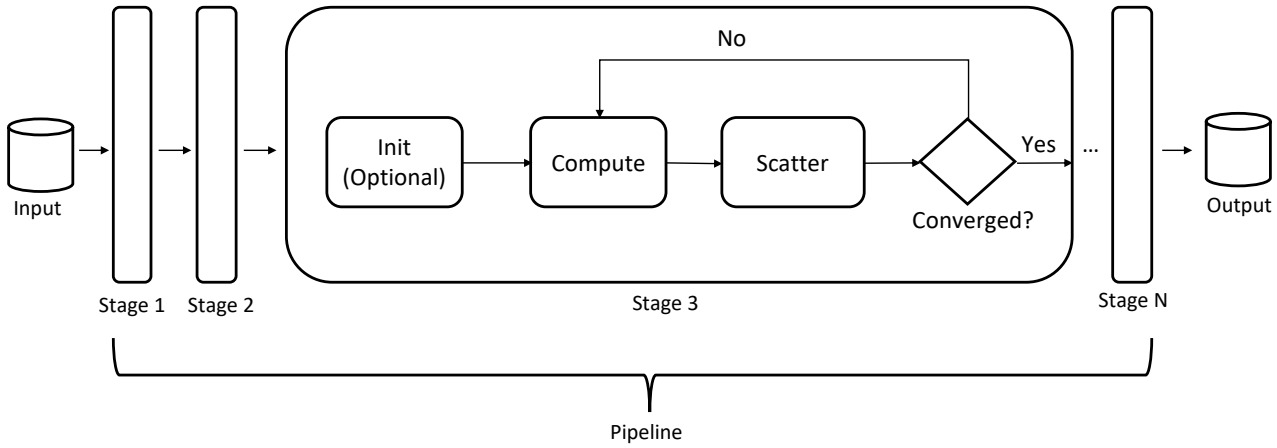


Figure 6.2: Computation Pipeline

fixed number of supersteps (using `Repeat(Stage userStage, int numOfSupersteps)`) or until some user-specified convergence criteria is met (using `RepeatUntil(Stage userStage, BooleanCriteria converged)`). users implement the BooleanCriteria interface that exposes two methods: `get()` and `apply()`. At the start of each superstep, the Master invokes `get()`, which if returns `true` if the stage has completed and can be terminated, otherwise master invokes next superstep of this stage. By default, all vertices are active at the start of a stage and a stage terminates when no new messages are generated in a superstep. Like ICM, all vertices `voteToHalt()` at the end of each superstep. Rather than pass explicit messages from one stage to the next, the vertex state is carried over and acts as an implicit channel between stages. The pipeline framework executes each stage linearly, in the order specified by the user. Currently, we do not allow two stages to run concurrently.

Example Algorithm 6.1 shows the logic for a Pipeline Master that allows us to design SCC in a more intuitive manner compared the the MasterCompute approach shown in Alg. 5.3. We use `Make_Graph_Symmetric` stage to construct a transpose of the input graph, and `Trimming` stage to identify vertices with only incoming or outgoing edges. Both `Make_Graph_Symmetric` and `Trimming` are executed once. Next, in the `Forward_Traversal` stage, each active interval vertex assigns its own vertexID as its componentID for all active intervals and propagates it along its out-edges. Additionally, such active vertices will invoke `apply()` method of the user-specified BooleanCriteria `areAllVerticesDeactivated` to signal they are active. In subsequent superteps, vertices update their own componentID with the smallest candidate componentID they had seen so far for each interval partition. `Forward_Traversal` stage continues propagating the componentID until the vertices converge.

Finally, in `Backward_Traversal` stage, every interval vertex whose componentID equals its

vertexID, propagates its componentID along its in-edges (computed during `Make_Graph_Symmetric` stage) and marks itself deactivated for the interval. All deactivated interval partitions will no longer participate in future computations and will ignore all received messages. In subsequent supersteps, vertices which receive messages test if the received message equals its current componentID and if so propagate it along their in-edges and deactivate themselves for the interval. `Backward_Traversal` stage converges when no new messages are generated. The Pipeline Master now tests the convergence criteria provided by the user by invoking `get()` on `areAllVerticesDeactivated`. If all vertices are marked deactivated, then SCC the algorithm terminates, and otherwise Master schedules `Forward_Traversal` stage for re-execution. Here, the `scatterDirection` (line 17 and 38) variable indicates the direction (IN, OUT or BOTH) along which scatter operates.

6.1.5.1 Summarize

Summarize logically encodes the two essential phases of data parallel applications; map and reduce, and can be composed with other graph computation *stages* in a pipeline. The user-defined map function is applied to each interval vertex object, yielding zero or more key-value pairs, which are then reduced to one scalar per key using the user-defined reduce function. GRAPHITE automatically inserts a group-operation between map and reduce, which takes a list of records as an input and creates a collection of values with same key. For any summarize operation, there can be at-max as many parallel reducers as the number of workers. In Section 7.9, we describe a computation pipeline which, identifies weakly-connected components using an interval program and computes the number of distinct connected components for each timepoint using summarize.

6.2 Implementation using Giraph

Besides these above design elements, GRAPHITE largely reuses the existing capabilities of Giraph, some of which we highlight below. The high level architecture is shown in Figure 6.3.

6.2.1 Resource Acquisition and Graph Loading

GRAPHITE uses YARN [111], a cluster resource management service, to request allocation of the machines for user application. The number of machines can be specified by the user using `workers` property. One of these machines hosts the *Giraph master*. The master is not assigned any part of the interval graph to process, but is responsible for global synchronization, error handling and assigning partitions to the *workers*. By default, each worker is assigned as many partitions as the the number of threads available to it for computation. However this can

```
1
2 class SCCPipeline implements Pipeline {
3    BooleanCriteria areAllVerticesDeactivated;
4
5    SCCPipeline() {
6       return new Pipeline(
7          Repeat(Make_Graph_Symmetric, 1),
8          Repeat(Trimming, 1),
9          RepeatUntil(
10            new Pipeline(
11               RepeatUntilConvergence(Forward_Traversal),
12               RepeatUntilConvergence(Backward_Traversal)
13            ),
14         areAllVerticesDeactivated),
15         RepeatOnce(WriteOutputToDisk)
16      );
17   }
18 }
```

Algorithm 6.1: Pipeline Master Composition for Time-Independent SCC using ICM

altered using `giraph.numComputeThreads` property. If w is the number of workers and t the number of compute threads available, we have the number of partitions as $p = w \times t$, e.g., in our experiments, we have 8 workers and 14 compute threads per worker, resulting in 112 partitions.

Interval vertices (and edges) of the input interval graph map to native vertices (and edges) in Giraph, but include details of their lifespan. Giraph loads the interval vertices and all of their out-going interval edges and associated time-varying properties from an input adjacency list file present in HDFS [98]. Each worker loads HDFS blocks for the input file present in its local machine into memory. GRAPHITE partitions the interval vertices into specific partitions depending on a *partitioning function* (refer sec. 6.2.3). This is performed in a separate pre-processing supertep after the graph is loaded. Based on this, each worker transfers local interval vertices and edges to their respective partitions present on various workers. This function also makes it possible for a worker to later know which partition a given interval vertex belongs to in order to send messages to the relevant worker hosting a vertex. The adjacency list loading and interval vertex partitioning together form the *load time* for the graph.

6.2.2 Input and Output Format

There are many possible file formats for interval graphs, and graphs in-general, such as comma-seperated values (CSV), tab-seperated values (TSV), GEXF [3], GML [4], GDF [2], DIMACS [51], or stored as relations in a database [128, 74]. To avoid imposing a specific choice of input file format, GRAPHITE decouples the task of interpreting an input file from the task of graph

```
class Make_Graph_Symmetric implements Stage {
   void compute(Vertex v, Interval t, long componentId, Message[ ] msgs) {
      for(Message m : msgs)
         v.createInEdge(m.interval, m.value);
   }
   void scatter(Edge e, Interval t, long componentId) {
      return new Message(e, t, componentId);
   }
}
class Trimming implements Stage {
   void compute(Vertex v, Interval t, long componentId, Message[ ] msgs) {
      if(if(isActive(v, t) && (v.outEdgeCount == 0 || v.inEdgeCount == 0) )
         deactivateInterval(v, t);
   }
}
class Forward_Traversal implements Stage {
   scatterDirection="OUT";
   BooleanCriteria areAllVerticesDeactivated;
   void init(Vertex v) {
      if(isActive(v, t)) {
         areAllVerticesDeactivated.apply(FALSE);
         v.setState(v.interval, v.id);
      } else { areAllVerticesDeactivated.apply(TRUE);  }
   }
   void compute(Vertex v, Interval t, long componentId, Message[ ] msgs) {
      if(isActive(v, t)) {
         minComponentId = ∞;
         for(Message m : msgs)
            minComponentId  = min(m.value, minComponentId);
         if(minComponentId < componentId){  v.setState(t, minComponentId);}
      }
   }
   void scatter(Edge e, Interval t, long componentId) {
      return new Message(e, t, componentId);
   }
}
class Backward_Traversal implements Stage {
   scatterDirection="IN";
   void compute(Vertex v, Interval t, long componentId, Message[ ] msgs) {
      if(isActive(v, t)) {
         if(componentId == v.id) {
            deactivateInterval(v, t);
         } else {
            for(Message m : msgs) {
               if(componentId == m.value)
                  deactivateInterval(v, t);
            }
         }
      }

   void scatter(Edge e, Interval t, long componentId) {
      return new Message(e, t, componentId);
   }
```

Algorithm 6.2: Reusable Computation Components/Stages

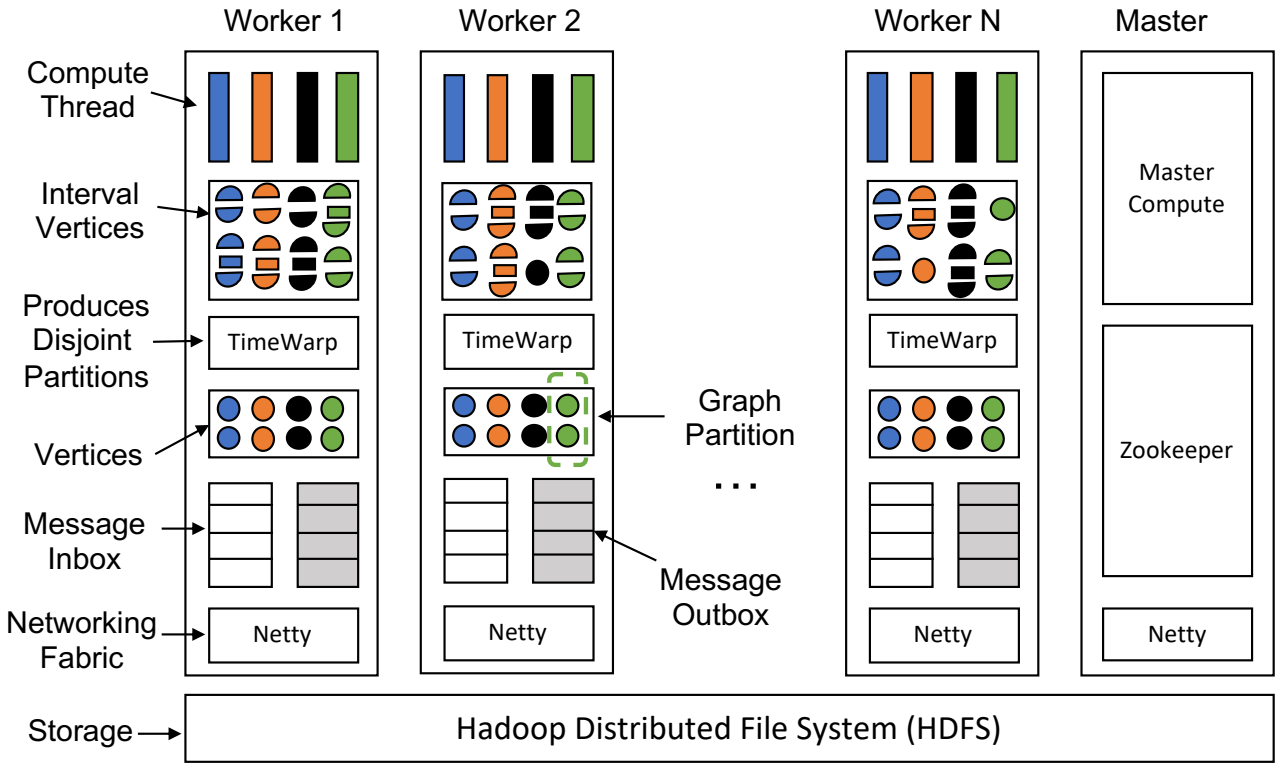


Figure 6.3: Architecture for GRAPHITE using Giraph [75]

computation using various implementations of *readers*. Further, the vertex and edge states after ICM computation can be generated in any arbitrary output format by the user by implementing *writer* interfaces. Like Giraph, GRAPHITE provides readers and writers for many common file formats, with the ability for the users to over-ride them using the properties `giraph.VertexInputFormatClass` and `giraph.VertexOutputFormatClass`.

6.2.3 Graph Partitioner

Graph partitioning is an essential pre-processing step for distributed graph computations, since to perform computation over multiple machines in a cluster, the input graph first needs to be partitioned by assigning vertices to individual machines. This can have a significant impact on the performance and resource usage in the computation stage [113]. Assignment of an interval vertex to a partition depends on a *vertex partitioning function*, and the default in GRAPHITE is the `HashPartitionerFactory` defined as `hash(ID) mod N`, which hashes a vertex ID to one of N partitions. users can replace it by modifying the `giraph.graphPartitionerFactoryClass` property. Although a random hash partitioner generates well-balanced workloads across machines, almost all neighbors of the interval vertices tend to be on remote machines. Hence

most ICM messages are sent across the network rather than through in-memory data transfer that is possible if vertices are collocated on the same worker. In Section 7.6, we show how more sophisticated partitioning algorithms can help reduce the network communication costs and overall execution time. Currently, GRAPHITE only supports static allocation of vertices to partitions – vertices once assigned to a partition cannot be migrated to a different partition. However this can be relaxed in future.

By default, Partitions inside GRAPHITE are stored using a map-based structure, which allows parallel access to vertices, albeit with a higher memory overhead. Users can replace this default data structure `SimplePartition` to a `ByteArrayPartition` class or a custom class using `giraph.partitionClass` property.

6.2.4 Fault tolerance

Fault tolerance in GRAPHITE is achieved through Giraph's existing checkpointing mechanism. Just like Giraph, GRAPHITE can be configured to trigger a *checkpoint* after every n supersteps. GRAPHITE serializes and persists the dynamic state and message stores to durable storage during a checkpoint, and uses it to recover from one or more worker failures. If a worker fails in a superstep, the rollback and recovery will be done for all workers to the nearest prior superstep at which a checkpoint was performed. In case of algorithms with multiple computation phases, GRAPHITE performs checkpointing at the global barriers between computation phases.

Users can modify the frequency of checkpointing by using `giraph.checkpointFrequency` property. By default, this property is set to 0, which disables checkpointing.

6.3 Optimizations

We highlight a few optimizations used to improve GRAPHITE's performance for ICM algorithms.

6.3.1 Interval-Message Combiner

Combiners are a common feature in platforms like Giraph, MapReduce and Spark to reduce network transfers between workers across iterations. We allow users to provide an optional interval message combiner using the `graphite.intervalMessageCombinerClass` property, which is an aggregation function over a set of messages received by a specific vertex at the start of a superstep and having overlapping intervals. It reduces the different message values for a single time-point into a single message value for that time-point. GRAPHITE then coalesces adjacent time-points back into intervals based on value-equality, and causes fewer interval messages to be passed to the compute function of that vertex. This is called a *receiver-side combiner*. There are no guarantees that the combiner will execute on the messages or the particular order in

which the messages will be combine. So it is only suitable for commutative and associative operations over message values, such as summing the rank values in PR or finding the smallest travel-time for TMST.

Unlike Hadoop MapReduce, Giraph only supports a receiver-side combiner and not a sender-side combiner. Such a combiner would execute over all messages generated to a target vertex from all vertices present on a worker (machine) before the end of a superstep. While we did implemented a sender-side combiner feature in GRAPHITE, we did not observe any performance improvements. Further, to combine messages at the sending worker, we need to store all outgoing message in a list for each destination vertex and this increases the memory usage on the sending worker. Also, messages are buffered twice, once in the outgoing messages list for combining, and then in the message buffers before batching and network transmission. This slows the rate at which buffers fill and are flushed. As a result, sender-side combiners are not enabled [53].

6.3.2 Inline Warp Combiner

We also allow users to specify *warp combiners* that execute as part of the warp step before *compute*, and after any receiver-side message combiners have executed. It applies the warp combiner logic to the grouped and partitioned messages it generates for each sub-interval of the vertex. This reduces the number of messages per partitioned vertex state to just 1 when calling the *compute* function, and avoids a linear scan through the input messages. This is coupled with the receiver-side message combiner that is applied before warp.

6.3.3 Warp Suppression

Interval-centric computing works best when the lifespan of entities are long, and with large time overlaps 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 causes the most overhead, we enable a feature to *selectively disable* the warp step if more than a certain fraction of input messages to a vertex have a unit lifespan. This avoids the warp costs and degenerates to a time-point centric execution model. While this causes more calls to the compute function, this outstrips the cost of calling warp without its associated benefits. The correctness is not affected. By default, warp suppression is enabled and can be turned off by setting the `graphite.warpSuppression` property to 0.

6.3.4 Variable-Integer Encoding

While GRAPHITE uses 64-bit data structures for vertex IDs, in most cases the vertex ID have a much smaller range than 2^{64}. To exploit this, we encode vertex IDs using a variable bit-length encoding scheme. For each byte, we use the 7 least significant bits to encode the value and the most significant bit to indicate if we need another byte to encode the residual value. Variable-integer encoding is enabled by default and can be disabled by setting `graphite.variableIntegerEncoding` property to 0.

Messages in GRAPHITE include an interval, with the start and end time-points. Given the billions of messages transmitted for algorithms over large graphs, this adds to the network costs. Since intervals may have a wide-range of durations depending on the temporal graph, we also use variable-integer encoding to represent them. In the graphs used in our our experiments, we observe that this optimization causes the overall message sizes to drop by 59–78%. Also, unit-duration messages and those whose end time spans till ∞ are treated specially – we pass just the start time point and a corresponding flag which is used to compute the 8-byte `long` for the end time at the receiving end.

6.3.5 Implicit Vote-to-Halt at the End of Each Superstep

In Giraph, we need to loop through each and every vertex in the graph at the start of every superstep to determine which vertices are active. Either this active flag should be set, or the vertex should have received a message in order for a vertex's compute function to be called. However, by choosing to halt-to-halt the vertices by default in ICM at the end of every superstep, we can avoid this full scan through the list of vertices in a partition to check if they are active and instead just use the message receipt for the decision. In fact, most Giraph and VCM algorithms over static graphs explicitly vote to halt after every superstep. Optionally, we also allow users to explicitly mark a vertex to remain active by invoking the `voteToRemainActive()` method.

6.4 Advantages of Architecting Graphite over Giraph

Giraph, on account of its popularity and broad adoption across industry and academia has received significant research interest over the years. Researchers have studied run-time characteristics (e.g. messaging and memory access pattern) of Giraph for a variety of algorithms and have identified several performance bottlenecks, which are commonly attributed to imbalanced computation and communication. To this end, several platform enhancements have been proposed and implemented for Giraph. Han and Daudjee [38] identified message staleness and

frequent global synchronization barriers to be a performance overhead and proposed Barrierless Asynchronous Parallel execution. Khayyat et al. [57] observed the need for run-time vertex migration to ensure balanced computation and communication. Their proposed approach, Mizan, monitors run-time characteristics and using these measurements constructs a migration plan to minimize imbalance across workers. Dindokar and Simmhan have designed runtime partition migration using a static analysis of the graph algorithm [29]. Ching et al. [22] identified algorithms having messaging patterns that can exceed available memory on destination vertex and created superstep splitting technique, which splits a message heavy superstep across several iterations. Liakos et al. [69] studied memory usage patterns in Giraph and proposed to replace default in-memory adjacency list with compressed representations to reduce memory footprint. Zhou et al. [129] propose online message computing, where incoming messages are consumed in a streaming manner to reduce memory footprint. A number of VCM algorithms over static graphs have also been implemented using Giraph.

Given this body of work, developing GRAPHITE over Giraph allows us to leverage many of these benefits, several of which can be applied transparently to Giraph without affecting the correctness of GRAPHITE. In Section 7.10 we discuss how two existing techniques, superstep splitting and asynchronous processing, proposed in context of static graph processing can be natively ported to GRAPHITE.

Chapter 7

Experimental Evaluation

In this chapter, we offer a detailed comparative evaluation of the intrinsic benefits of the ICM model and various platform optimizations. No single prior study has examined these number and variety of temporal graphs and algorithms.

7.1 Temporal Graph Datasets

We run experiments for a diverse set of 6 real-world graphs (Table 7.1) to rigorously study the impact of their characteristics on the performance of the algorithms for Graphite and several baseline platforms (Section 7.2). These graphs vary in the size, per snapshot and cumulatively *(Small: GPlus [34], USRN [27], Reddit [44, 71]; Large: MAG [30], Twitter [19], WebUK [13])*; 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 [10] distribution [1], 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 [48] generator for Twitter, but the property variations are native for the rest. These are described in detail below:

7.1.1 Google Plus (GPlus)

Google Plus is a directed social network where vertices are *users* and edges are the *follows* relationship between them. Each snapshot represents the network at the end of a particular

[1] https://github.com/facebookarchive/linkbench

Table 7.1: 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] http://home.engineering.iastate.edu/~neilgong/gplus.html
[2] http://users.diag.uniroma1.it/challenge9/download.shtml [3] http://trafficengland.com
[4] http://cs.cornell.edu/~jhessel/projectPages/redditHRC.html
[5] http://openacademic.ai/oag [6] http://twitter.mpi-sws.org [7] http://law.di.unimi.it/datasets.php

month during *July–October, 2011*. The *edge property* weight (float) for GPlus is generated using the Facebook distribution given in the LDBC Data Generator's configuration[1]. Data generated using LDBC mimics the cardinalities, correlations and distributions of real social networks.

Each snapshot is self-contained and no edge spans across snapshots. This means that each snapshot requires distinct compute and no messages can be shared across snapshots. This forms the best case for all baselines and is the worst case for Graphite.

7.1.2 Reddit

This is a temporal graph constructed from *comments* made by users on the Reddit social news aggregation site during 2005–2015. Vertices are *users* and edges are *comments* made on a user's post by another user. These are aggregated on a *monthly basis* to generate an interval graph. The *time interval* of an edge from one user to another ranges from the time of their first comment to the time of their last comment. The *edge property* represents the number of such interactions between them during a given month.

7.1.3 US Road Network (USRN)

This is the full road network of USA, where intersections and endpoints are represented by vertices and the roads connecting them are undirected edges. Its large diameter (6262) results in a large number of supersteps for traversal algorithms to complete. The topology of the graph does not change. We synthetically sample edge properties that represent coarse-grained travel

[1] https://github.com/ldbc/ldbc_graphalytics/blob/master/config-template/graphs/datagen-8_9-fb.properties

duration on each road from a distribution of real traffic flow from 2500 roads, provided by the UK Highway Agency [1]. We generate 96 snapshots, each representing the traffic during a 15 *min* interval and for a 24-hour duration.

7.1.4 Microsoft Academic Graph (MAG)

This citation graph captures over 166 million *papers published* between the years 1800 and 2018 as vertices, and its directed edges represent the *cites* relationship between them i.e. a directed edge is created from citing paper to cited paper. We have *one snapshot per year*, with each vertex and its out-edges having a *starting time-point* as the year of publication and a *fixed ending time* as 2018 – the last snapshot indicating the "present". This is a monotonically growing graph since all vertices and edges have the same, fixed ending time and are never removed. In each snapshot, weight assigned to an edge is the weighted average of the citing paper's (source vertex's) cumulative citation count till that snapshot.

7.1.5 Twitter

This is a directed social network generated from 30 snapshots crawled during September 2009, where vertices are users and edges indicate the *follows* relationship, . It has a power-law degree distribution with a few vertices having high in-edge degree. This is the only graph where real dynamism in the topology is absent and instead we simulate it using the Facebook distribution from LinkBench Distribution Generator [1]. Starting from the original Twitter graph, LinkBench adds and removes vertices with a probability of 0.72 and 0.28, and similarly, adds and removes edges with a probability of 0.75 and 0.25, for a monthly vertex and edge churn of 4% and 12% respectively. The *edge property* weights for Twitter dataset is generated using a different Facebook distribution LDBC Data Generator Config template[2].

7.1.6 WebUK

This is a temporal graph generated from 12 monthly snapshots of the `.uk` web domain, crawled between May, 2006 and May, 2007. The vertices are *web pages* and a directed edge connecting two vertices represents presence of hyperlink from webpage represented by source to target. For each snapshot, the *weight* property assigned to an edge is the normalized duration of association (i.e. edgeLifespan/graphLifespan) times the in-degree of source vertex for that snapshot. If this weight value does not vary for an edge across contiguous snapshots, the value is used for that entire interval in the interval graph.

[1] http://www.trafficengland.com

[2] https://github.com/ldbc/ldbc_graphalytics/blob/master/config-template/graphs/datagen-9_4-fb.properties

7.1.7 Discussion

These diverse and realistic characteristics of the interval graphs can affect performance of ICM in different ways. High degree skews can cause performance bottlenecks at a handful of worker machines, leading to stragglers. Large diameters can result in slow convergence. The different rates of property changes allows us to study and compare GRAPHITE's performance under those conditions. For Example, MAG allows us to showcase the static overheads of multi-snapshot analysis and highlights the short-coming of the graph transformation baseline, which as seen in Table 7.1 blows up the graph size by a factor of 13×.

7.2 Comparative Baseline Platforms

We compare ICM against four contemporary baseline abstractions that we adapt to temporal graphs, and implement over Apache Giraph. This ensures that the primitives are the key distinction and not the platform programming language or the execution engine.

7.2.1 Multi snapshot baseline (MSB) and Chlonos (CHL)

The *Multi snapshot baseline (MSB)* is used for Time Independent (TI) temporal graph algorithms. Here, Giraph loads each snapshot sequentially from disk and executes the VCM logic for the algorithm on each snapshot independently [78, 107].

Next, we enhance MSB and implement a variant ("clone") of Chronos [40], which we call *Chlonos (CHL)*. This improves upon the simple MSB strategy 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. Its scatter logic identifies duplicate messages pushed by the VCM compute logic to adjacent time-points of a sink vertex, and replaces them with a single interval message, with the whole interval assigned as its validity, saving network time and memory use. But, the compute logic is invoked for each snapshot independently. To ensure lock-free execution on target vertex, based on message validity, payload is replicated for all valid time-points (each time-point corresponds to a snapshot). Chlonos can operate on incremental batches of snapshots, and each batch fits as many snapshots as possible in the distributed memory to run the algorithms. It is also limited to expressing TI algorithms.

7.2.2 Transformed Graph Baseline (TGB)

The *transformed graph baseline (TGB)* converts the snapshots into a *transformed graph* (also called a "static graph") where the interval vertices are unrolled into vertex replicas, one each for the number of incoming and outgoing edges at distinct time-points, and each being valid for a single time-point [118]. This is discussed in more detail below. This can operate on both TI

and Time Dependent (TD) temporal graph algorithms using a VCM logic that is similar to a static graphs. However, its transformation logic is unique for each algorithm, which negates its ability to serve as a unifying model. The edge-weights are used to capture algorithm-specific properties, such as travel cost. The number of vertices and edges in the transformed graph is bounded by $\mathcal{O}(k \times |E|)$.

As discussed in [120], a Temporal Graph $\mathcal{G} = (V, E)$ can be transformed to a static graph $\tilde{G} = (\tilde{V}, \tilde{E})$ using a transformation process which consists of two phases, vertex creation and edge creation. the example shown in Fig. 7.1(b) and Fig. 7.2 illustrates this graph transformation approach for the temporal graph shown in Fig. 7.1(a) (same as Fig. 1.1(a)).

In the *Vertex creation phase* [1], each temporal vertex $\langle vid, [t_s, t_e)\rangle \in V$, creates static vertices in $\tilde{V}$ as follows:

1. Let $T_{in}(vid)$ be the set of k distinct time-points at which edges from any in-neighbor of temporal vertex vid are incident on vid. Formally stated, $T_{in}(vid) = \{t' \mid t' = t + \lambda,\ t \in [t_s, t_e)\ and\ \exists \langle eid, vid', vid, [t_s, t_e)\rangle \in E\}$ and $\mid T_{in}(vid) \mid = k$. We create k static vertices, one for each distinct time-point $t \in T_{in}(vid)$, uniquely identified using a composite identifier $\langle vid, t\rangle \in \tilde{V}$. We denote the ordered list of these k static vertices as $V_{in}(vid) = \{\langle vid, t\rangle \mid t \in T_{in}(vid)\}$. Elements in $V_{in}(vid)$ are ordered in descending order of their time-point, i.e., $\forall \langle vid, t_i\rangle, \langle vid, t_i\rangle \in V_{in}(vid)$, $\langle vid, t_i\rangle$ is ordered before $\langle vid, t_j\rangle$ if and only if $t_i > t_j$.

2. Let $T_{out}(vid)$ be the set of $\hat{k}$ distinct time-points at which a temporal edge originates from a temporal vertex vid. Formally, $T_{out}(vid) = \{t',\ t \in [t_s, t_e)\ and\ \exists \langle eid, vid, vid', [t_s, t_e)\rangle \in E\}$ and $\mid T_{out}(vid) \mid = \hat{k}$. We create $\hat{k}$ static vertices, one for each distinct time-point $t \in T_{out}(vid)$, uniquely identified using a composite identifier $\langle vid, t\rangle \in \tilde{V}$. We denote the ordered list of these $\hat{k}$ static vertices as $V_{out}(vid) = \{\langle vid, t\rangle \mid t \in T_{out}(vid)\}$. Like $V_{in}(vid)$, elements in $V_{out}(vid)$ are also ordered in descending order of their time-point.

In the *Edge creation phase* [1], for each temporal vertex $\langle vid, [t_s, t_e)\rangle \in V$, we create static edges in $\tilde{E}$ as follows:

1. Using $V_{in}(vid)$ and $V_{out}(vid)$ computed in the vertex creation phase, we now create a directed static edge from the static vertex $\langle vid, t_{in}\rangle \in V_{in}(vid)$ to $\langle vid, t_{out}\rangle \in V_{out}(vid)$, where $t_{out} = min(\{t : \langle vid, t\rangle \in V_{out}(vid), t > t_{in}\})$, and no other static edge from any other static vertex $\langle vid, t'_{in}\rangle \in V_{in}(vid)$ to $\langle vid, t_{out}\rangle \in V_{out}(vid)$ has been created. Such a

[1] γ and λ are algorithm-specific parameters which may be user-defined constants or derived from graph.

static edge $\langle vid_{t_{in}}, \langle vid, t_{in}\rangle, \langle vid, t_{out}\rangle\rangle \in \tilde{E}$ is uniquely identified by $vid_{t_{in}}$ and its weight is set to γ.

2. Let $V_{in}(vid) = \{\langle vid, t_1\rangle, \langle vid, t_2\rangle, ..., \langle vid, t_k\rangle\}$, $k < t_e$. For all $1 \leq i < k$, we create a directed static edge from vertex $\langle vid, t_{i+1}\rangle \in V_{in}(vid)$ to $\langle vid, t_i\rangle \in V_{in}(vid)$. Each such static edge $\langle vid_{t_{i+1}}, \langle vid, t_{i+1}\rangle, \langle vid, t_i\rangle\rangle \in \tilde{E}$ is uniquely identified by $vid_{t_{i+1}}$ and its weight is set to γ. Static Edges for $V_{out}(vid)$ are created similarly, but is omitted from discussion for brevity.

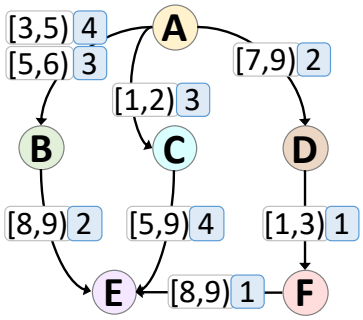
3. For each temporal edge $\langle eid, vid_i, vid_j, [t_s, t_e)\rangle \in E$, a directed static edge is created $\forall t \in [t_s, t_e)$ from vertex $\langle vid_i, t\rangle \in \tilde{V}$ to $\langle vid_j, t + \lambda\rangle \in \tilde{V}$. Each such static edge $\langle eid_t, \langle vid_i, t\rangle, \langle vid_j, t + \lambda\rangle\rangle \in \tilde{E}$ is uniquely identified by eid_t and all property labels and values associated with time-point t for temporal edge eid is copied over to static edge eid_t.

Once the transformed (static) graph is created, design the temporal algorithm just by by executing the non-temporal VCM logic on it. For Example, to compute the *time-dependent single source shortest path* on a transformed graph G shown in Fig. 7.2c from the source vertex A to all temporally reachable vertices in interval graph $\mathcal{G}$, we augment the transformed graph G to $\tilde{G}$ (shown Fig. 7.2d) by creating a vertex $A' \in \tilde{G}$ and a directed edge from A' to each vertex $\langle A, t\rangle \in V_{out}(A), V_{out}(A) \in G$ with weight 0. Then, we run static single-source shortest path algorithm on augmented graph $\tilde{G}$ from source vertex A'. The path with the least distance among the computed shortest path from A' to each $\langle A, t\rangle \in V_{in}(A)$ is the time-dependent shortest path from A to vertex v in interval graph $\mathcal{G}$.

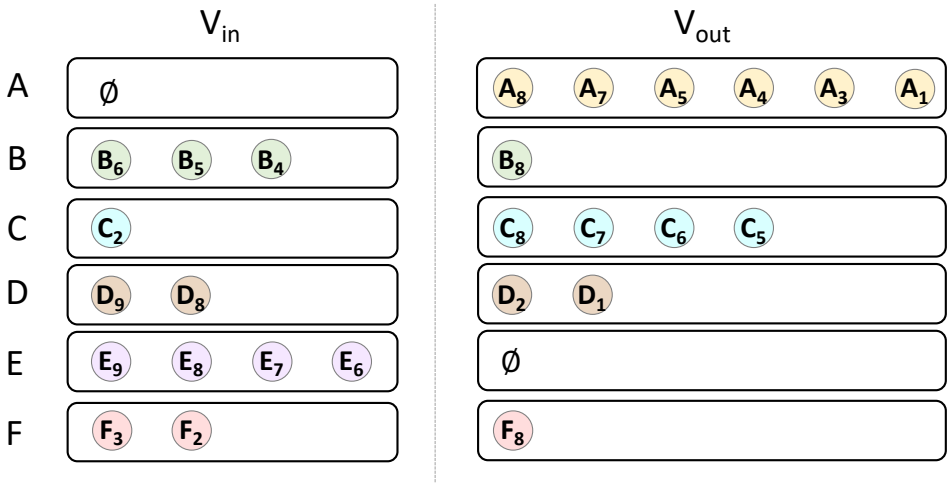
We evaluate TGB only for TD algorithms. While it is possible to use it for TI algorithms, its performance and memory use is much worse than the MSB and CHL baselines discussed above for the TI algorithms, as shown in Figure 7.3 for two graphs and the four TI algorithms. E.g., when using TGB, GPlus was 7–16% slower that MSB, while it ran out of memory for MAG. The higher memory footprint for TGB can be accounted for by the much larger transformed graph and the associated state it operates on.

7.2.3 GoFFish-TS (GOF)

GoFFish-TS (GOF) [99] models a temporal graph as a sequence of snapshots. Besides support for sending messages to neighboring vertices in the same snapshot, GOF allows sending messages from a vertex in a snapshot to itself at a future snapshot by writing them to disk. However, the execution of the VCM logic processes vertices of a single snapshot at a time. An outer loop (referred as a time-step in [99]) over the snapshots delivers (and retrieves) temporal messages to



(a) Input Temporal Graph



(b) Output of Phase-1(a) *(left) and Phase-1(b)* (right)

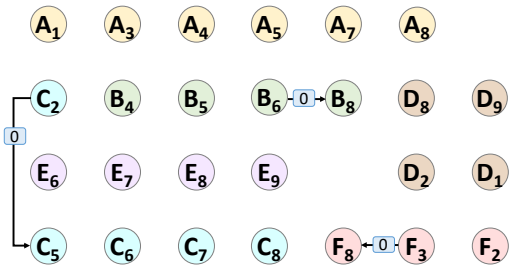
Figure 7.1: Phase-1 of Graph Transformation ($\gamma = 0$ and $\lambda = 1$)

(and for) future snapshots from disk, and an inner loop of supersteps operates on one snapshot using VCM.

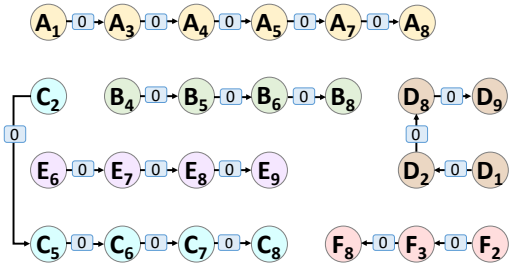
Our implementation uses the output state produced by snapshot s_i, as an input to the computation of next snapshot s_{i+1} , following the sequentially time-dependent pattern. Additionally, all temporal messages received for snapshot s_{i+1} from any prior snapshots are made available to target vertices in the first superstep. This is illustrated in Fig. 7.4. We limit an evaluation of GOF to just TD algorithms as it degenerates to MSB for TI algorithms.

7.2.4 Other Baseline Platforms Considered

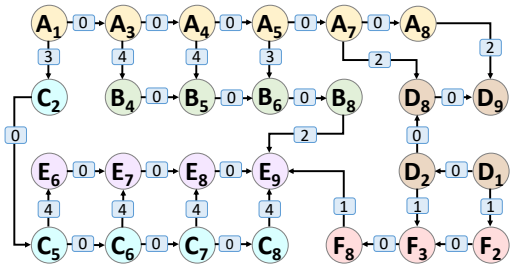
We have also evaluated other baseline approaches like Apache Spark's GraphX [35] for TI and TD algorithms, and Tink [70] for TD algorithms only. They are based on alternative execution platforms, GraphX on Apache Spark [125] and Tink on Apache Flink [18]. However, their



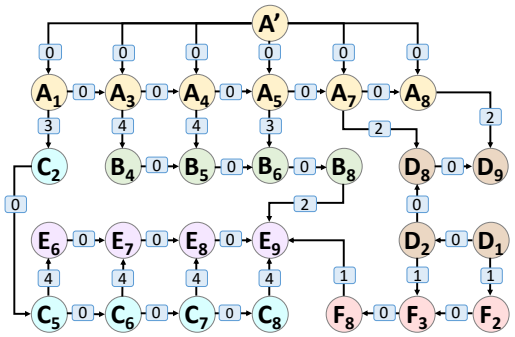
(a) Output of Phase-2(a)



(b) Output of Phase-2(b)



(c) Output of Phase-2(c)



(d) Final Tranformed Graph

Figure 7.2: Phase-2 of Graph Transformation ($\gamma = 0$ and $\lambda = 1$)

performance was much worse than ICM or the other baselines we have implemented over Giraph, as shown in Fig. 7.5 for GraphX on TI and TD algorithms for 3 graphs. E.g., for USRN, Tink

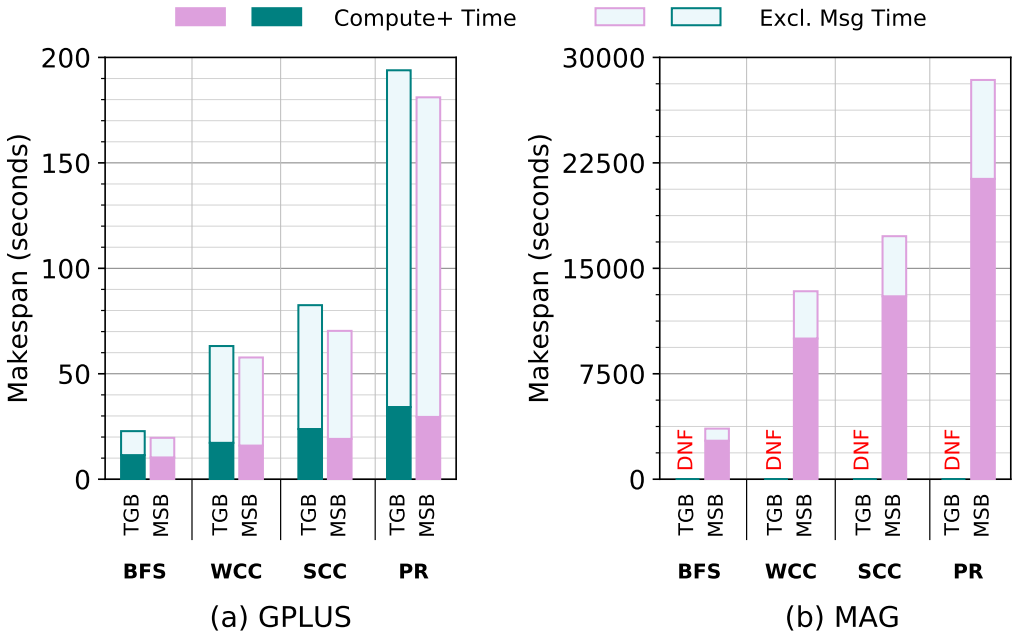


Figure 7.3: Makespan for 4 TI algorithms using TGB and MSB baseline implemented in Giraph for GPlus (*left*) and MAG (*right*)

took 4.2× (not shown) longer compared to TGB and 21.5× longer than Graphite for FAST, while GraphX took 3× longer compared to TGB and 9.5× longer compared to Graphite. For Twitter, Tink ran out of memory, while GraphX took 2.3× and 43× longer compared to TGB and Graphite respectively. For MAG, both Tink and GraphX run out of memory. Further, using Giraph as the common execution platforms for ICM and the baselines allows us to focus on the performance of the programming primitives and conceptual approach rather the software implementation or execution engine. Hence, we exclude these other less performant systems from further evaluation.

7.3 System Setup

We run the experiments on a 10-node commodity cluster. Each node has one 8-core (16 Hyper-Thread) Intel Xeon E5-2620 v4 CPU @ 2.1 GHz, 64 GB of RAM, 2 TB of HDD, and connected through 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 each and 60 GB heap space. 2 HyperThreads and 4 GB of RAM is reserved for opersting system. Except for weak scaling, we use 8 nodes for all other experiments. The algorithms are run from a cold cache state. We use the default hash partitioner of Giraph to partition the graphs, and

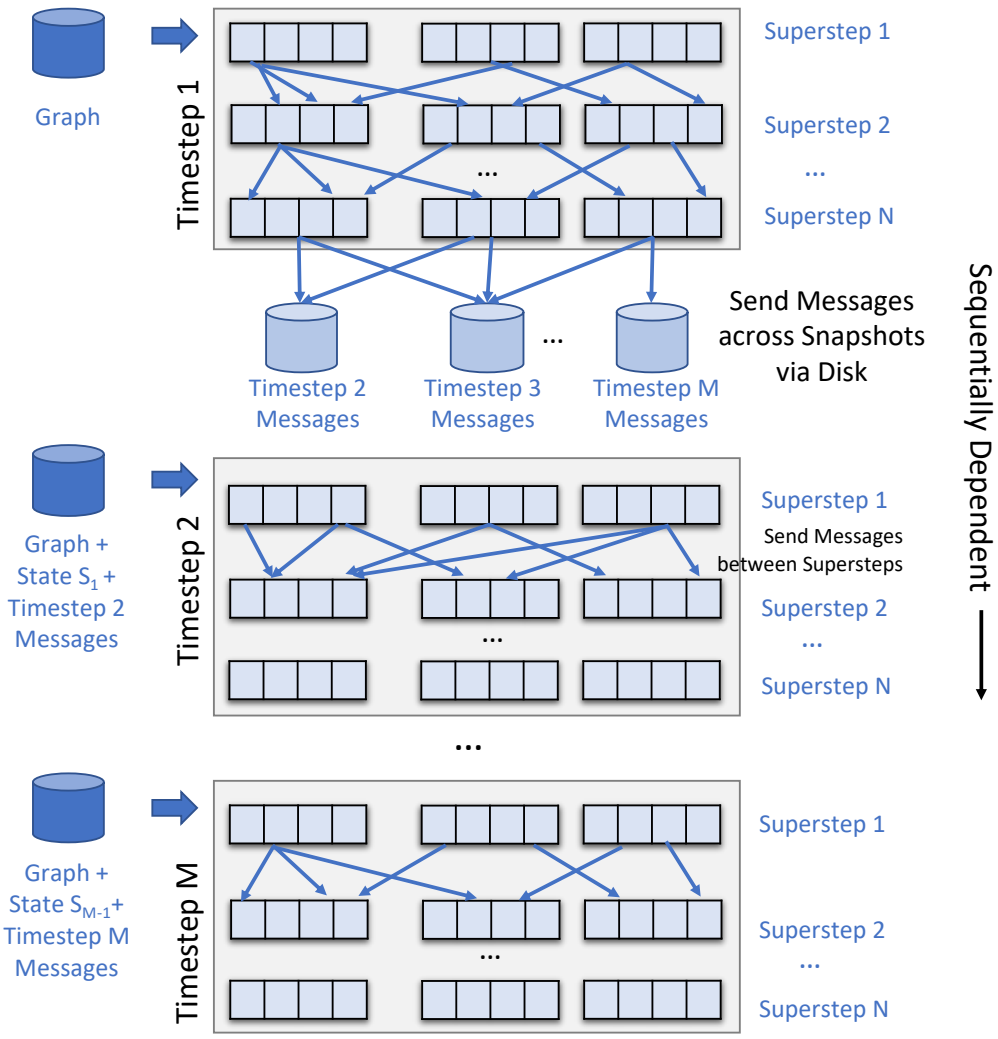


Figure 7.4: GoFFish-TS: Each Timestep operates upon a single snapshot, and is decomposed into multiple supersteps as part of vertex-centric model.

we disable its check-pointing and out-of-core computation. Graphs are loaded from HDFS.

7.4 Metrics Reported

We report the *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 to the baselines, *graph loading time* is reported separately. We also report the total number of calls to the user's *compute logic* and the *messages sent*.

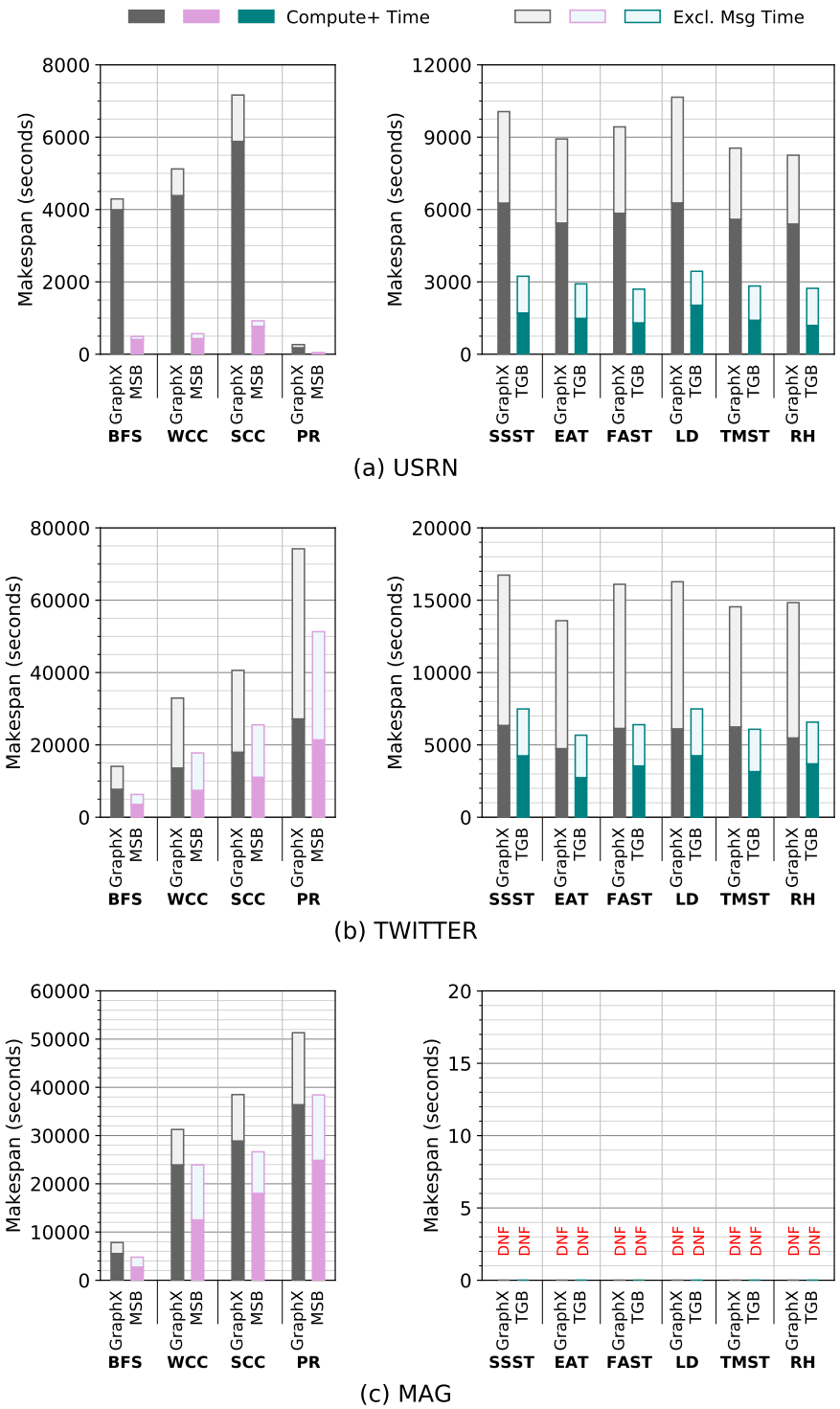


Figure 7.5: TI (*left*) and TD (*right*) algorithm baselines implemented using Apache Spark's GraphX [35] API and Apache Giraph [1]

Table 7.2: 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 Did Not Finish (DNF) for that graph and platform. DNL indicates that the input graph Did Not Load due to memory pressure.

		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*

7.5 Analysis

Table 7.2 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. 7.6 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. 7.6 for brevity. They perform similar to SSSP.

As Table 7.2 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.

7.5.1 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. 7.6, (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*

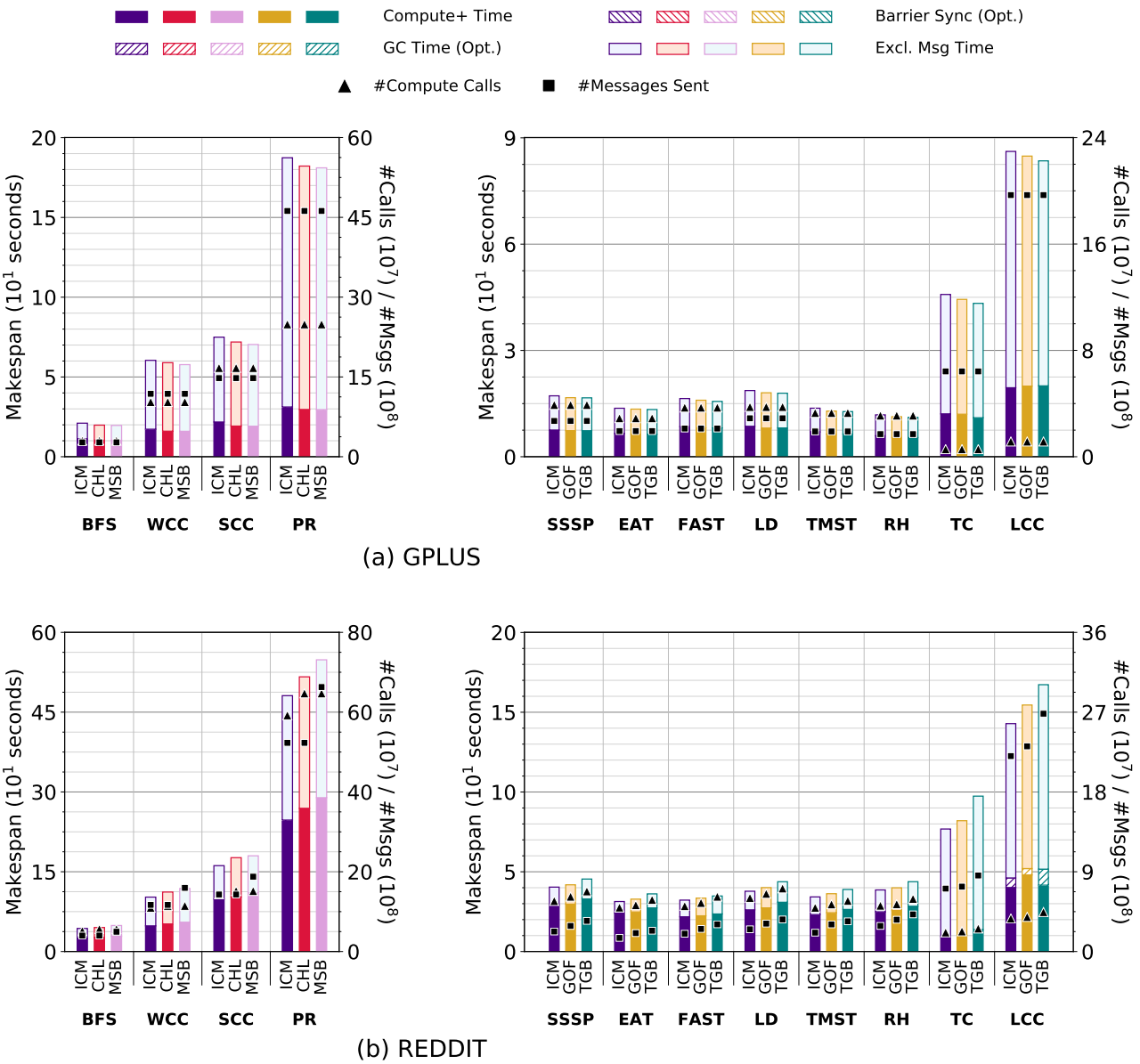


Figure 7.6: Makespan and the count of compute calls and messages sent for the 4 TI and 8 TD algorithms. Barrier & GC time splits for makespan are shown only if large. Note the different scaling on the Y axis. *Continued...*

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 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

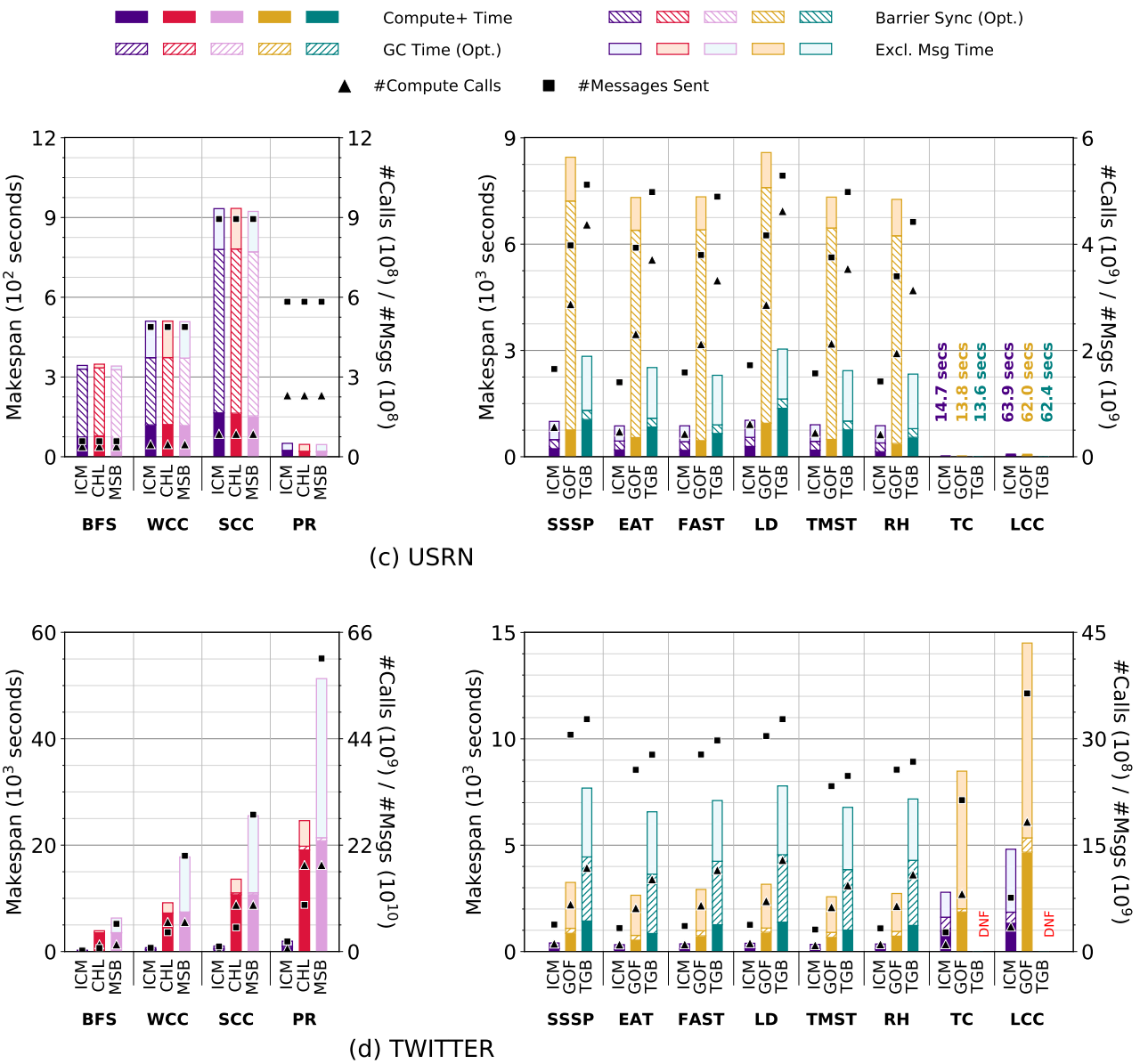


Figure 7.6: *continued...*

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.

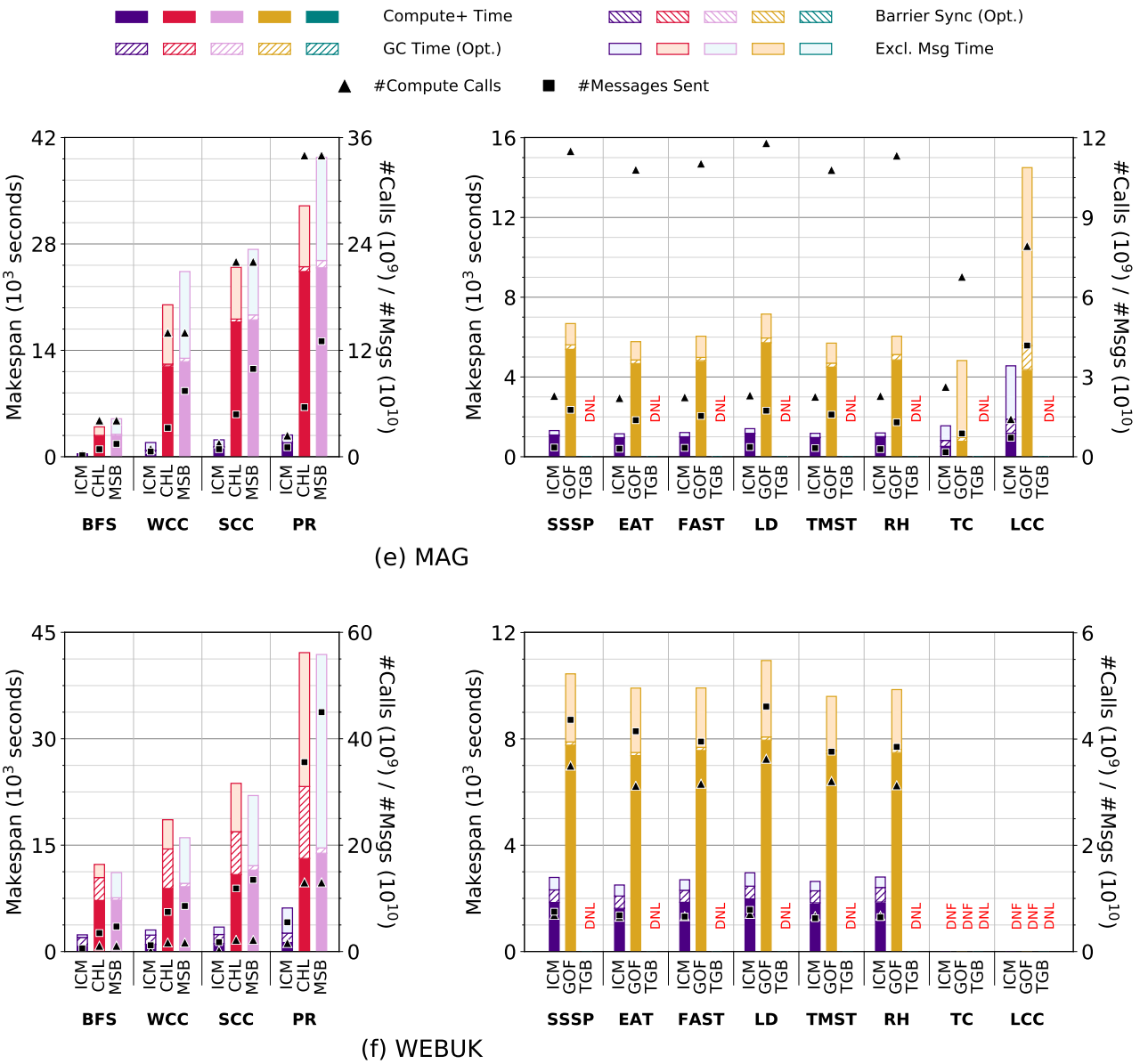
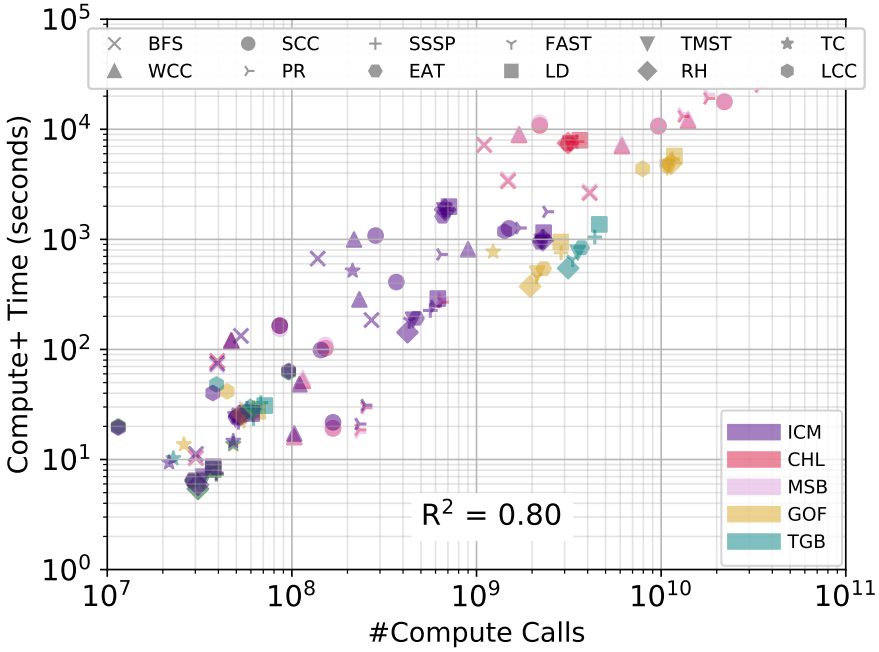


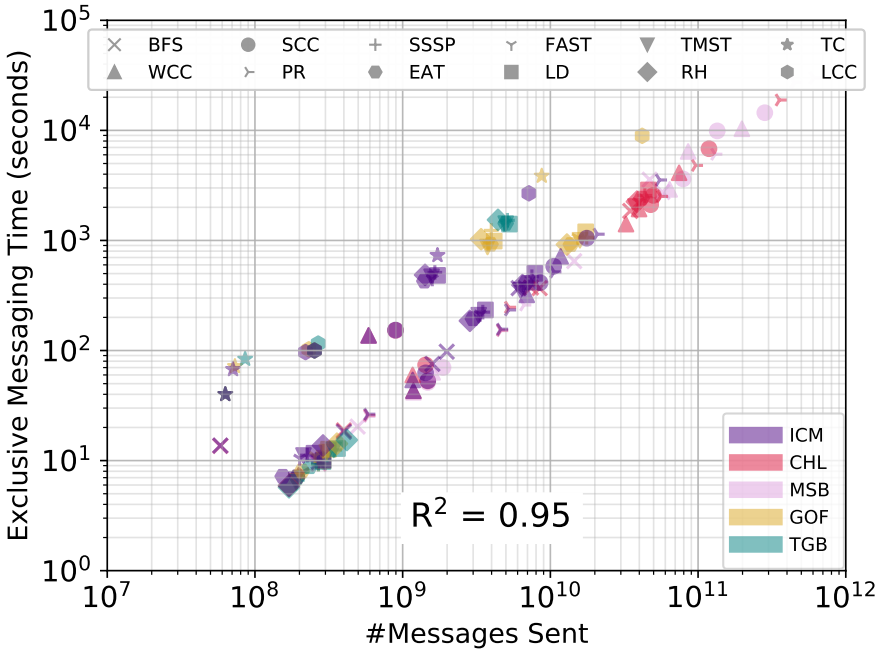
Figure 7.6: *continued...*

7.5.2 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. 7.7. There are



(a) Compute Calls v. Compute+ Time



(b) Messages v. Messaging Time

Figure 7.7: Log-Log Scatter plot of count of compute calls and messages, and their time contribution to the makespan.

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.

7.5.3 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 7.1). 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. 7.5.4. 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.

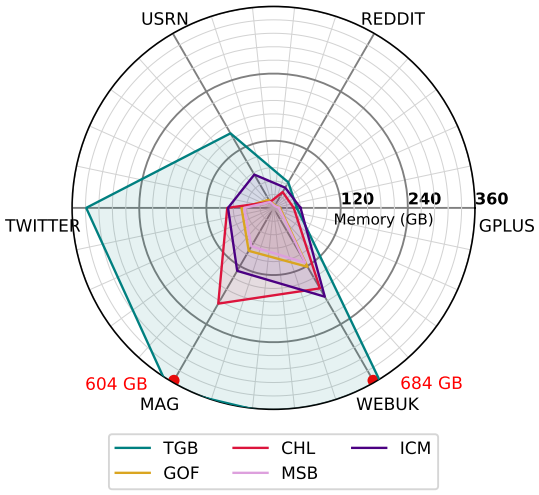
7.5.4 ICM out-performs for large graphs

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 7.1, Fig. 7.8(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.

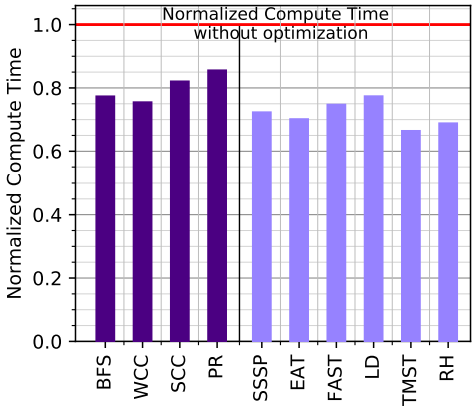
Large graphs use more memory and create billions of message objects. This triggers the JVM's GC; we use the G1 GC that is efficient for large heap sizes. E.g., for Twitter, TGB calls GC 33 times for SSSP and this takes $\approx 32\%$ of its total makespan, compared to 6 calls to the GC for ICM that account for 5% of its makespan. For WebUK, calls to GC make up $\approx 20\%$ 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 by operating on one or a batch of snapshots, their snapshot data size on disk is larger than ICM. Fig. 7.8(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.*

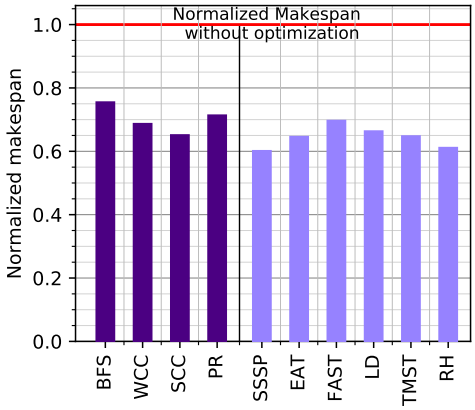
Lastly, using *warp combiner* reduces a pass by the warp and another by the compute on the input messages into a single pass that does both. All our algorithms except LCC and TC are commutative and associative, and define combiners. This benefits large graphs with many



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



(b) Warp combiner benefits for MAG



(c) Warp suppression benefits for GPlus

Figure 7.8: GRAPHITE's memory footprint on graph load, and benefits from Warp optimizations.

messages received per interval vertex. Fig. 7.8(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.

7.5.5 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. 7.8(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 $\approx 7\%$ (excluding load times) compared to the other baselines (Fig. 7.6(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 $\approx 14\%$.

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 $\approx 5B$ peak messages sent for GPlus and Reddit.

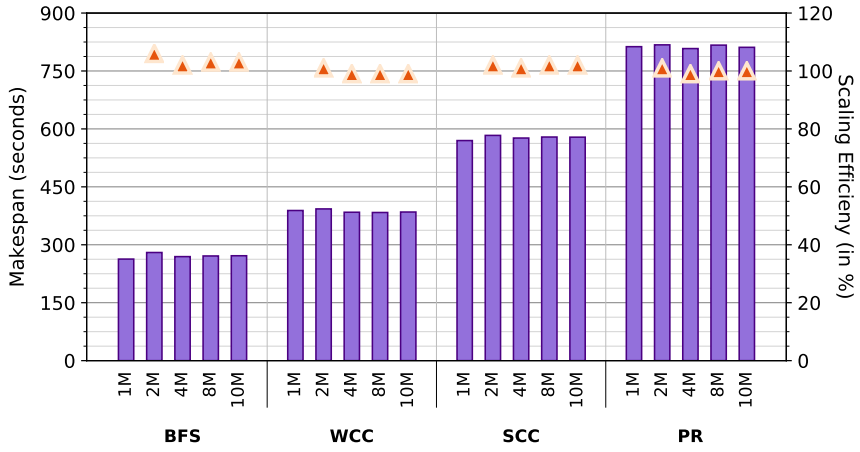
7.5.6 ICM benefits graphs with large diameters, and is competitive for non-temporal structures

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.

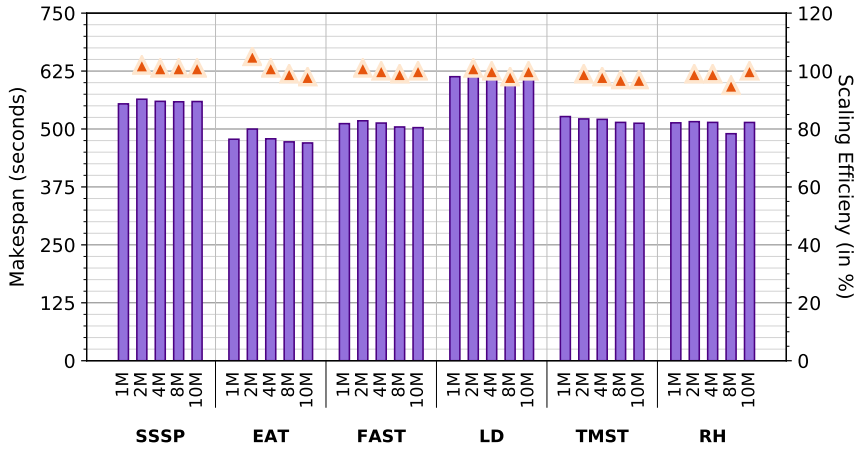
ICM offers some benefits due to the large diameter of 6262 for USRN. The superstep count is proportional to the diameter for traversal algorithms, while PR, TC, and LCC have fixed superstep counts of 10, 3, and 4 respectively. The total barrier synchronization time is separately shown for USRN (Fig. 7.6(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.

7.5.7 ICM exhibits weak scaling



(a) Time-Independent Algorithms



(b) Time-Dependent Algorithms

Figure 7.9: Weak Scaling of GRAPHITE for all algorithms on synthetic graphs, using $n = 1, 2, 4, 8$ and 10 machines ('nM' shown on X axis). Each machine holds $\approx 10M$ vertices, $\approx 100M$ edges. Left Y axis reports the makespan (bars), while right Y axis shows the scaling efficiency relative to a single machine (triangles) – 100% indicates perfect linear scaling.

Weak scaling is a common scalability metric for Big Data platforms and distributed systems, which follows Gustafson's Law [37]. An ideal weak scaling means that the time taken for n items with m machines is the same as $x \cdot n$ items with $x \cdot m$ machines, i.e., 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

TI Algo.	ICM	CHL	MSB
BFS	24	34	21
WCC	19	28	16
SCC	114	131	111
PR	26	36	23

			TGB		
TD Algo.	ICM	GOF	Pre-Proc	Algo	Total
SSST	29	40	*44*	*27*	71
EAT	27	37	*40*	*25*	65
FAST	30	39	*42*	*25*	67
LD	33	45	*45*	*27*	72
TMST	27	39	*42*	*24*	66
RH	25	35	*40*	*23*	63
TC	41	56	*44*	*36*	80
LCC	80	95	*44*	*73*	117

Figure 7.10: Number of lines of Java user code for all algorithms using ICM and the baselines

a synthetic graph using LDBC's Facebook degree distribution [48] , and perturb its structure over 128 time-points using Facebook's LinkBench distributions [10]. 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 7.1). In Fig. 7.9, 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.

7.5.8 ICM algorithms are concise

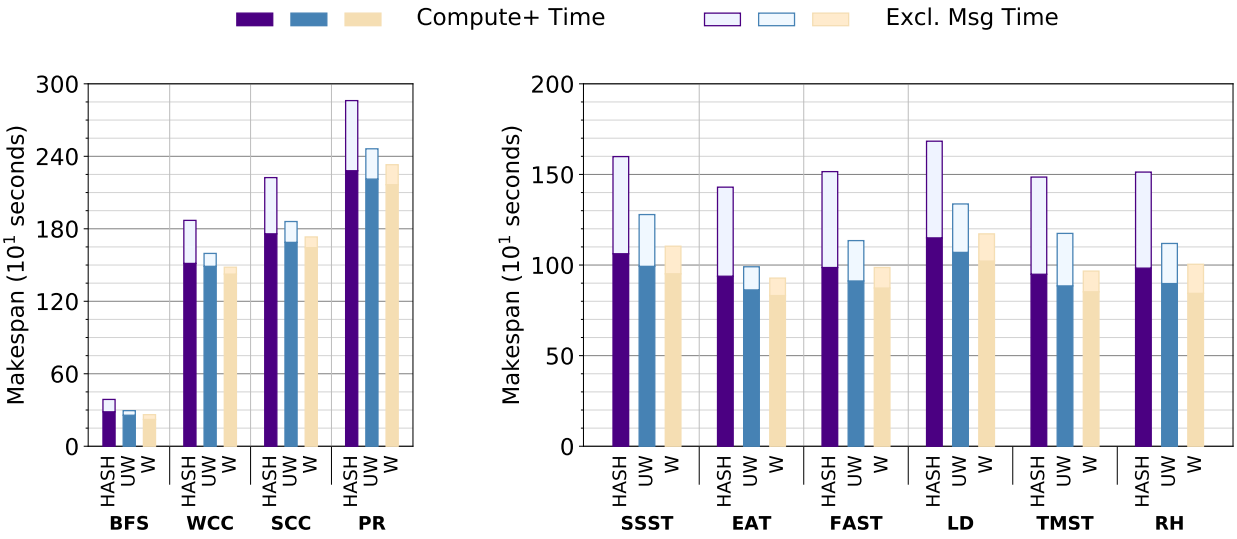
Figure 7.10 reports the number of lines of code (LoC) written by the algorithm designer, for the 10 algorithms using ICM and the four baselines. For TGB, there is substantial pre-processing involved and so we separate it out from the core algorithm.

The 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.

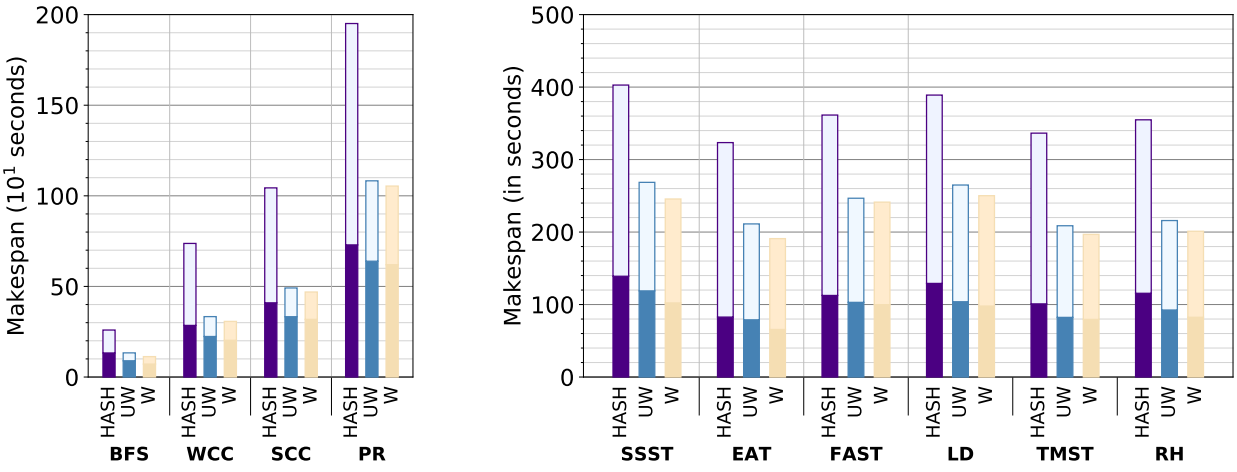
7.6 Effect of Partitioning Quality

We now investigate the effect of partitioning on performance of Graphite. We compare three simple partitioning strategies: Random Hash Partitioning (HASH), which is the default, and partitions generated by METIS for condensed graph[1] with unweighted edges (UW), and condensed graph with weighted edges (W) For weighted graph, edge lifespan is used as edge weight, while for unweighted graph, the weight is always set to 1. Note that condensed graph is

[1]All unique vertices and edges from all snapshot are folded into a single static-graph. Edge properties (if any) are ignored.



(a) MS (left) and TD (right) Algos. for Graph MAG



(b) MS (left) and TD (right) Algos. for Graph Twitter

Figure 7.11: Comparing the performance of ICM using different partitioning strategies: Hash, METIS with Un-Weighted Edges (UW) and METIS with Weighted Edges (W)

used only for generating partition mapping and not for actual graph computation. METIS [54] attempts to reduce the total weight of edge cuts between different partitions, and balance the number of vertices in each partition. The former helps reduce communication costs between machines, while the latter help balance the compute load on each machine. This partitioning is done offline, before graph loading.

As we see from Figure 7.11, the choice of partitioning strategy has a significant effect on communication cost. METIS-based partitioning has uniformly lower communication costs and

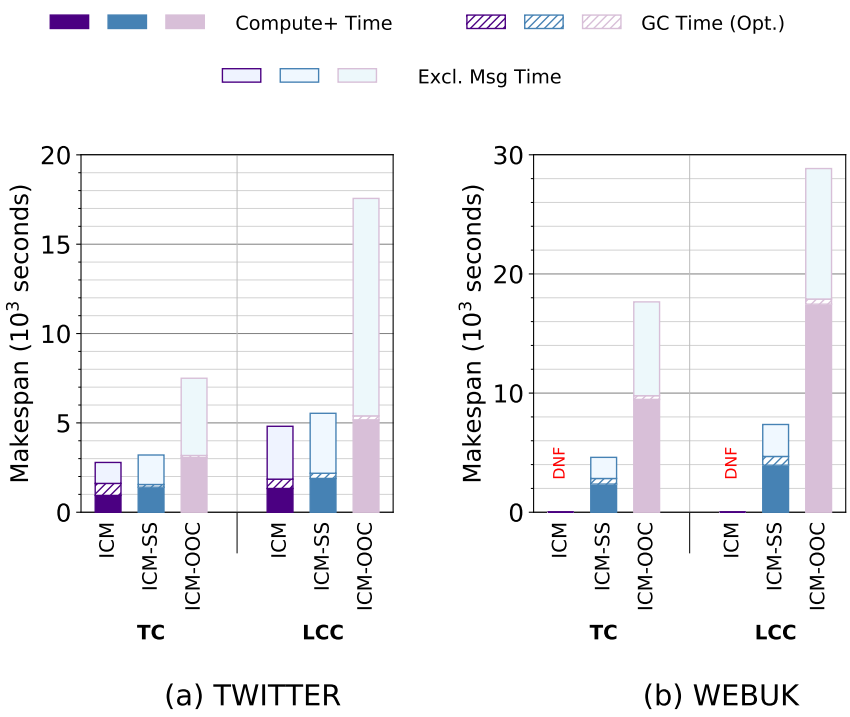


Figure 7.12: Makespan time for TC and LCC algorithms with *superstep splitting* enabled. (ICM-SS : ICM-Superstep-Splitting and ICM-OOC : ICM-Out-of-Core)

overall makespan compare to the default hash partitioning. The weighted edge partitioning performs better than unweighted edges. The benefits of weighing are higher for for graphs with non-uniform vertex and edge lifespans (e.g., MAG), but diminished for graphs with uniform lifespans (e.g. TWITTER).

However, METIS requires that the number of partitions be known in advance and the partitioning is done offline, while hash partitioning is done online, during graph loading time. Also, others have shown that no single partitioning strategy is likely to be the best fit for all situations for non-temporal graphs [112] and high-performance temporal graph analytics systems should support multiple partitioning strategies. We need to validate this in detail for temporal graphs as part of future work.

7.7 Superstep Splitting

One of the benefits of designing GRAPHITE using Giraph is our ability to leverage other advances to VCM and Giraph. One such technique is called Superstep Splitting, as discussed in [22]. This is beneficial for large graphs that generate a lot of messages in a superstep and can overwhelm the available distributed memory capacity of the workers that have to receive and buffer all

these messages before the next superstep starts. Instead, each superstep is split into multiple sub-iterations, with messages being sent to only a specific subset of vertices participating in that sub-iteration. The vertices receiving a message will apply their receiver-side combiner to reduce these messages into one after that sub-iteration. Once all messages for the superstep have been delivered and incrementally combined, the compute is called. For superstep splitting to be applicable, a receiver-side combiner needs to be defined, i.e., the compute logic must be commutative and associative.

We extend ICM to permit superstep splitting. Here, that the master computation will run the same superstep logic for a fixed number of sub-iterations. During a sub-iteration, every vertex generating a message uses a hash function that decides if the destination vertex ID for the message is participating in this sub-iteration, and only sends it the message if it is. This way, $\mathcal{O}(M)$ cumulative messages that were previously buffered at the receiving vertices now reduces to $\mathcal{O}(\frac{M}{I})$ messages being buffered over I iterations, and being incrementally combined into 1 message after each sub-iteration. However, this also increases the number of times the compute function is called, from $\mathcal{O}(C)$ to $\mathcal{O}(C \times I)$.

Figure 7.12 compares the performance of the default ICM, against superstep splitting (ICM-SS) and also the *out of core* feature of Giraph (ICM-OOC). In the latter, messages that are received and overflow the memory are pushed to disk, and then incrementally loaded when the compute function executes in the next superstep. As we see from the plots, ICM-SS takes marginally more time than ICM, and this is due to the larger computation time from the extra compute function calls. However, ICM-SS is able to reduce the garbage collection time substantially since the number of objects allocated and deallocated for the millions of messages per superstep has now reduced substantially per sub-iteration.

More so, using ICM-SS allows us to compute TC and LCC algorithms for the large WEBUK graph without resulting in out-of-memory as was seen in ICM that did not finish (DNF). This ability to scale to larger graphs by reducing the runtime memory pressure from buffering messages is the key benefit of superstep splitting. Further, we also see that ICM-SS is $\approx 4\times$ faster than ICM-OOC by avoiding the overheads of object de/serialization and disk I/O.

7.8 Relaxing Synchronous Barrier

Vertex centric programs can be executed using two types of iterative models of computation. The synchronous (or BSP) model guarantees that processing in current superstep is only based on the vertex state computed in and the messages received from the previous superstep. This allows algorithms to be easily implemented and reasoned about. Pregel, Giraph, Blogel, GoFF-ish and Graphite are examples of frameworks which make use of the synchronous model. On

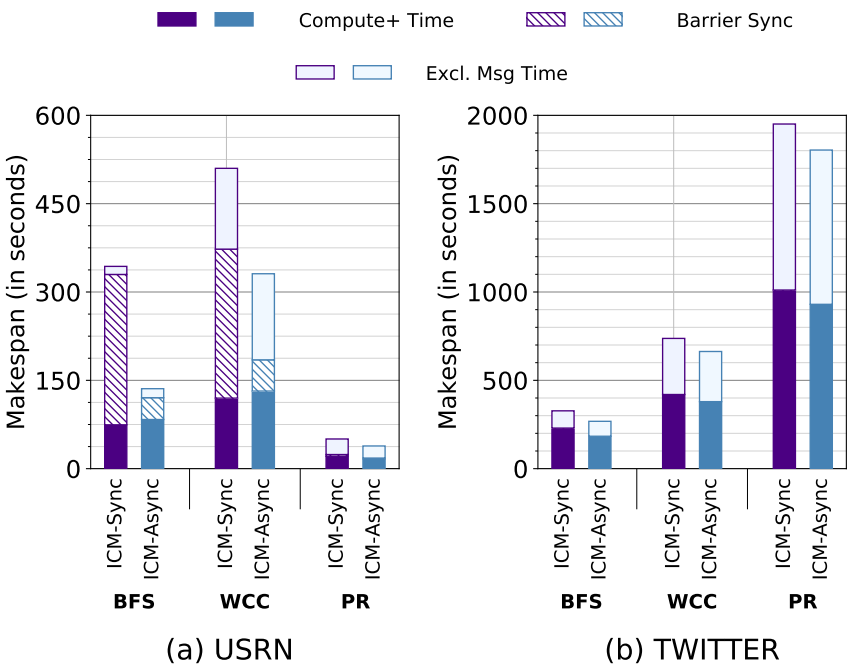


Figure 7.13: Makespan time for Asynchronous Computation Model

the other hand, the asynchronous model [38] permits processing in a superstep to be based upon the vertex state and messages from the previous as well as the current superstep, i.e., vertices that receive a message from other vertices in the current superstep need not wait for a barrier synchronization before they can start processing those received messages. This allows vertices to make faster progress by consuming the most recent messages received, and reduces the penalties imposed by straggler workers during global synchronization.

We adopt the Barrierless Asynchronous Processing (BAP) model, as described for Giraph [38], for GRAPHITE and compare the performance of the BSP (ICM-Sync) and BAP (ICM-ASync) models of computing for ICM. Figure 7.13 compares these performances of three algorithms for USRN, which has a large diameter of 6262 and hence takes that many supersteps for traversal algorithms, and for Twitter.We see that ICM-Async offers substantial benefits for the traversal algorithms (BFS, WCC) on USRN, by sharply reducing the cost of barrier synchronization. The benefits are muted for the Twitter graph with a smaller diameter, and for non-traversal algorithms like PR. In these cases, the algorithms may effectively complete execution within a single superstep by consuming messages as they receive it. However, ICM-Async does have a higher memory footprint ($\approx 1.3\times$ for USRN and $\approx 1.6\times$ for Twitter Graph) due to buffering more messages in a single superstep.

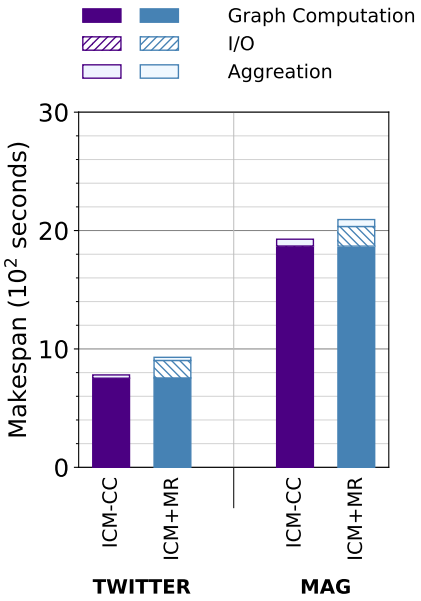


Figure 7.14: Makespan time for WCC algorithm + Global Aggregation

7.9 Composability

We compose a computation pipeline using Graphite's composability feature to implement a WCC application to count the number of weakly-connected components. It first runs the *time-independent WCC* on an input temporal graph as the first stage using the ICM model to identify the component IDs for each interval vertex (Algorithm 7.1). This is followed by a *summarize* stage that finds the number of distinct connected components for each time-point defined as a Map and a Reduce function. The *Map* operator emits the component ID for each time-point in a vertex's lifetime and the *reduce* function counts the number of distinct component IDs for each emitted time-point.

Figure 7.14 shows the makespan time this composed Algorithm 7.1 in Graphite (ICM-CC), and compares it with running WCC in Graphite followed by an explicit MapReduce job using Spark to perform the count of the number of components (ICM+MR) with data being exchanged over HDFS. The latter is $\approx 20\%$ slower than the unified composition within Graphite, caused by the additional disk I/O and replication overheads of writing/reading the intermediate output between Graphite and Spark in HDFS.

In summary, Composability enables us to stay within a single framework throughout the analytics process, eliminating the need to write connectors to move data between frameworks (e.g. Graphite and Hadoop) and reducing expensive data movement.

```
class WCCPipeline implements Pipeline {
   WCCPipeline() {
      return new Pipeline(
         RepeatUntilConvergence(WCC),
         countDistinctComponents,
         RepeatOnce(WriteOutputToDisk)
      );
   }
}

class WCC implements Stage {
   void compute(Vertex v, Interval t, long componentId, Message[ ] msgs) {
      if(getSuperstep() == 1) {
         v.setState(v.interval, v.id);
         return;
      }
      minComponentId = ∞;
      for(Message m : msgs) {
         minComponentId  = min(m.value, minComponentId);
      }
      if(minComponentId < componentId) { v.setState(t, minComponentId); }
   }

   Message[] scatter(Edge e, Interval t, long componentId){
      return new Message(e, t, componentId);
   }
}

class countDistinctComponents implements Summarize {
   Iterator<Pair<Long, Long>> map(Vertex v) {
      Collection<Pair<Long, Long>> tuples;
      for(long timepoint : v.interval) {
         tuples.add(new Pair<Long, Long>(timePoint,
            vertex.getState(timePoint)));
      }
      return tuples;
   }

   Pair<Long, Long> reduce(Long key, Long[ ] values) {
      return new Pair<Long, Long>(key,
         graphiteUtils.countDistinct(values));
}
```

Algorithm 7.1: Computational Composability Example

7.10 Discussion

An important question is whether all kinds of graph analytics algorithms can be expressed efficiently at interval level. Like its vertex-centric variant, ICM can scale linearly with the number of vertices on 300 machines [22]. But it is not well-suited for graph analytics that require a subgraph-centric view around interval-vertices, e.g., local clustering coefficient, triangle and motifs counting [56, 106, 130]. This is due to the communication overhead, network traffic, and the large amount of memory required to construct multi-hop neighborhood in each vertex's local state [84]. The communication overheads are even greater for the baseline approaches on account of graph blow-up and redundant communication. Nevertheless, the interval-centric model permits re-using existing techniques [76] which were proposed in the purview of vertex-centric model to address such overheads, e.g. superstep splitting [22], source vertex batching [86], out-of-core computation [126] and relaxing the synchronous barrier [38], some of which we have explored.

Chapter 8

Toward Incremental Graph Processing

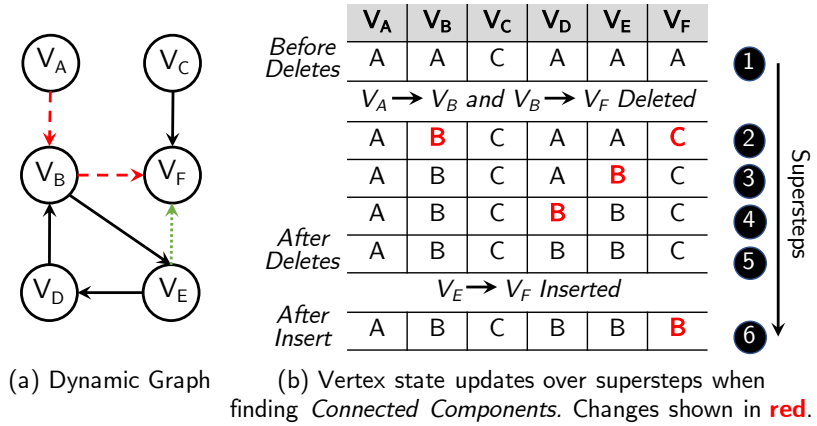
In this chapter, we introduce on the problem of *incrementally processing* of dynamic graphs which are updated in real-time. In order to scale to large graphs and fast rates, we need the computation of the graph algorithm to be updated incrementally, rather then re-computing the entire algorithm from scratch on the updated graph. We offer some preliminary thoughts on how to address this using some of the basic concepts of ICM.

A *dynamic graph* [5] is a graph on which a stream of *updates* (or mutations) are applied, which causes the structure and/or the properties of the graph to change. The rate of updates may be rapid, $\mathcal{O}(10^3/sec) - \mathcal{O}(10^5/sec)$. The updates may either *monotonic*, where the updates cause the structure of the graph strictly grows, i.e., only addition of vertices/edges), or *non-monotonic* which allow both additions and deletions of vertices and edges.

While temporal algorithms can be designed using ICM over the entire updated graph, after one or more updates have been applied, this will not scale to large graphs or high update rates since the latency for computation may exceed the rate of updates. As a result, we need to examine incremental computation of such algorithms so that they localize the effect of the updates and consistently maintains insights on the dynamic graph as it is changing, with low latency. E.g., fraud detection analyzes and recognizes patterns between customers (vertices) and financial transactions (edges, properties) in real-time to preempt losses [6].

8.1 Challenges

While such *large dynamic graphs* are ubiquitous, there are few abstractions and distributed platforms for analyzing them at scale. Some [39, 99, 49] discretize the dynamic graph into a sequence of snapshots and recompute them from scratch. These leverage existing offline graph platforms [74, 35, 88, 80, 89] and algorithms [123], but cause substantial redundancy in compu-



	V_A	V_B	V_C	V_D	V_E	V_F
Before Deletes	A	A	C	A	A	A
	$V_A \rightarrow V_B$ and $V_B \rightarrow V_F$ Deleted					
	A	**B**	C	A	A	**C**
	A	B	C	A	**B**	C
	A	B	C	**B**	B	C
After Deletes	A	B	C	B	B	C
	$V_E \rightarrow V_F$ Inserted					
After Insert	A	B	C	B	B	**B**

(a) Dynamic Graph

(b) Vertex state updates over supersteps when finding *Connected Components*. Changes shown in **red**.

Figure 8.1: Incremental processing using Wave

tation and distributed communication which limits the update rate that can be maintained [31]. ICM itself operates only on the entire materialized graph and is not designed for online processing. Recomputing over the entire graph can result in severe redundancies in the computation if the updates only affect a small portion of the entire graph for a given algorithm. E.g., the clustering coefficient for vertices in one part of the graph may not change if the updates happen to a different part of the graph. But other algorithms like PageRank may require full recomputation of all vertices, but still benefit from faster convergence if we start with prior values that were computed.

A fundamental approach [42, 36, 83] to minimize such redundancy is to perform *incremental processing* [85]. Prior works [92] have examined incremental computation for monotonic graph updates. However, naïvely resuming computation from an initial state or the neighborhood state of the modified vertices/edges may cause the algorithm to produce incorrect results under non-monotonic updates [114]. E.g., in Fig. 8.1(a), we label vertices with the smallest vertex ID of the *connected component* they are part of. Initially, all these vertices are part of the component A. When deleting edge V_A–V_B and V_B–V_F, the vertices V_B and V_F are affected. But updating their component labels just based on their neighbors causes V_B to be incorrectly labeled with A by V_D, while V_F is correctly labeled as C by V_C. This is due to V_B and V_D being part of a cycle. Such inaccuracies will also propagate as future updates arrive.

8.2 Incremental Graph Processing using Wave

We present Wave, a preliminary approach for incremental distributed graph processing for the class of *selective graph algorithms* that extends from the ICM abstraction we have introduced. Users design temporal graph algorithms using ICM, but Wave avoids redundant computation

and communication by *dynamically tracking* the state dependencies among vertices to decide if incremental computation on specific vertices is required. If so, it *transparently schedules* vertex execution and state inheritance at an appropriate superstep. The results provided by Wave are identical to recomputing the algorithm on the new graph using ICM, but *orders of magnitude faster*.

Selective graph algorithms are a sub-class of graph algorithms for which vertex program is a *selection function* that compares messages received via in-edges using min, max or other comparative operations, and uses one of the messages to update vertex state either directly or by performing some computation using it. For such algorithms, the state of vertex depends on a single in-neighbor vertex. e.g., Breath First Search, Weakly Connected Components, Single Source Shortest Path, Reachability, etc.

8.2.1 Approach

Two key challenges in incrementally processing of dynamic graphs are to identify (1) which vertices of the graph are affected by updates and require recomputation, and (2) what prior states should be reused in the recomputation. We make two observations that help address these:

1. *We say a vertex v is dependent on a vertex u if there is a directed path from u to v.* e.g. In Fig 8.1, vertex V_B is dependent on vertex V_A. Similarly, vertex V_D is dependent on V_E, V_B and V_A, however not on V_F or V_C as no directed path from either V_F or V_C exists. Here, if the state of a vertex or its adjacent edge changes, or they are mutated, then that vertex has to be recomputed. We term such a vertex as *affected*. This *may* cascade a recomputation to all its dependent vertices. e.g., In Fig 8.1, when edge $V_B \rightarrow V_F$ is deleted, state of vertex V_F must be re-computed. For doing so, vertex V_F pulls state from its in-neighbor V_C.

2. *For vertices with cyclical dependencies – in-neighbor is dependent on vertex which initiates the pull, directly re-using state from in-neighbor may result in incorrect result.* E.g., for the deletes in Fig. 8.1(a), the cycle V_B–V_F–V_D means V_B cannot directly use V_D's state A.

To eliminate the affect of cycles, we can track all transitive dependencies in the graph. However, naïvely tracking the transitive dependencies between affected vertices is expensive, takes $\mathcal{O}(V^2)$ space and requires maintainence of a global structure across distributed machines. Instead, we maintain a *level* information for each vertex that is the path-length from the "source" vertex that its current state is causally dependent on. Intuitively, if $level(v) < level(u)$, then v is not dependent on u; else, v may be dependent on u. Such levels are useful since it

tells us how many supersteps an update to the source will take to propagate to the dependent vertex, for traversal based algorithms.

Levels also help detect cycles in vertex-centric computation, where a message updates traverse one edge per superstep. If a vertex is not dependent on an updated vertex, it is not part of its cycle. Else, any update on a vertex should propagate and wait for $s = level(u) - level(v)$ supersteps to see if it is returned back to itself. If so, there is a cycle and a recomputation of all vertices in the cycle may be needed. In the worst case, $2s - 1$ supersteps are required to converge. E.g., in Fig. 8.1(b), deleting the edges causes Wave to wait for 3 supersteps for V_B's updates to propagate through the 3-cycle, and subsequently converge in the 4^{th} superstep. But the edge delete and add that affect V_F can converge in 1 superstep as it is not part of a cycle.

Besides levels, vertices also maintain the last received update message from each of its adjacent vertices. This is a form of *memoization* which ensures that causally dependent vertices that are not part of a cycle can immediately reuse an alternative recent message from a neighbor to converge to a solution in one superstep. This trades off memory but saves communication and synchronization costs. The number of vertices for which messages are memoized can be dynamically tuned.

This is a *conservative* version of Wave (Wave-C) that uses just the level information to detect cycles, causing up to $2s - 1$ supersteps. As an optimization, we maintain a fixed-length *Bloom filter* at each vertex that tracks vertices that are part of a cycle it is part of. Since Bloom filters have no false negatives, we know that the absence of an updated vertex at a given vertex's filter means it is not dependent or part of its cycle. This can save s supersteps, and we refer to this as Wave-B.

8.3 Experimental Evaluation

8.3.1 Setup

We evaluate the performance of our preliminary design of Wave for Breadth First Search (BFS), Connected Components (CC) and PageRank (PR) algorithms, on the Twitter graph [14]. The graph is initially populated with 41M vertices and 1.4B edges, and subsequently, updates consisting of an equal number of edge additions and deletions are streamed in to the graph. These updates are batched into 10M, 50M and 100M and are applied before the incremental computation is performed by Wave [97, 114, 49].

We compare Wave against two baselines. *Recompute* is naïve and reruns the full user algorithm on the updated graph. *KS-Lite* is a vertex-centric version of the incremental Kickstarter approach [114], but like the original, does not support the non-monotonic PR algorithm. Be-

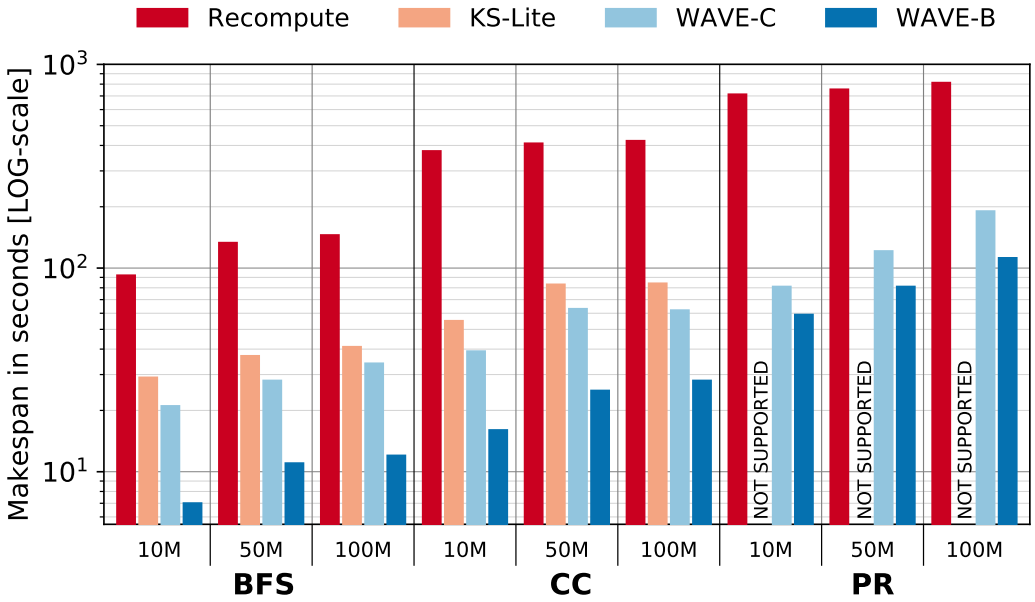


Figure 8.2: Incremental Graph Processing on Twitter [14] Dataset. Size of update batch is shown on inner X axis.

sides the conservative Wave-C that defers compute supersteps based on levels, we also evaluate the efficient Wave-B that uses Bloom filters to identify if vertices are not in a cycle before awaiting supersteps.

8.3.2 Results

Fig. 8.2 shows the time taken by these 4 strategies on 8 machines. Wave-C is 7–23× faster than Recompute and 3–4× faster than KS-Lite. Wave-B is even faster than Wave-C by 1.37–3×. These benefits correlate with fewer (≈ 40%) vertex recomputes. Larger batch sizes increases the throughput (average updates/sec) for the incremental approaches, with Wave-B supporting a peak of 8.3M updates/sec for BFS, i.e., close to a 1 million updates/sec per machine.

8.4 Discussion

These initial results are promising and offer a strong motivation for further investigation of incremental processing for a larger class of graph algorithms, with ICM and Graphite forming the core. We also need to examine formal guarantees of the correctness of such incremental processing for the supported class of graph algorithms. Lastly, additional platform enhancements and validation are required. These are left to future work.

Chapter 9

Conclusions

In this thesis, we present the Interval-centric Computing Model (ICM), a novel and unifying abstraction for enabling analytics over large temporal graphs by exposing time-intervals as a first-class entity. The cornerstone of our model is a unique transformation operator called *Time-warp*, which enables automatic sharing of computation and communication across adjacent time-points of a vertex. Warp offers two essential properties. 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 the number of messages generated are minimal, giving ICM its performance. We rigorously evaluate ICM's performance and scalability for 6 diverse real-world temporal graphs – as large as 131M vertices and 5.5B edges, and as long as 219 snapshots, in one of the largest such studies. Our ability to express 12 TD and TI algorithms attests to its intuitiveness and 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×. Graphite also exhibits weak-scaling with near-perfect efficiency.

In summary, ICM plugs a key gap in current literature for generic and scalable temporal graphs primitives. We sought to develop a thin extension on top of existing parallel graph computation models with the goal to identify the essential data model and core operators needed to support efficient temporal graph computation. We believe that ICM can be adopted in other static graph processing systems, including GraphX [35], GraphLab [72], GoFFish [100], and Gelly [18], and we are hopeful that the proposed abstraction will enable further development of temporal analytics.

As future work, we plan to extend ICM to process real-time temporal graphs of a streaming nature, offer querying capabilities over temporal property graphs and explore possible storage and partitioning strategies. Support for formal temporal graph algebra is also a possibility.

Appendices

Appendix A

Time-warp using Temporal Sort-Merge Aggregation

S

τ_m	S
[0,10)	S_1

M

τ_m	M
[0,4)	18
[2,7)	20
[5,7)	22
[5,9)	27
[9,10)	5

Time Join

τ_m	S	M
[0,4)	S_1	18
[2,7)	S_1	20
[5,7)	S_1	22
[5,9)	S_1	27
[9,10)	S_1	5

Time Warp

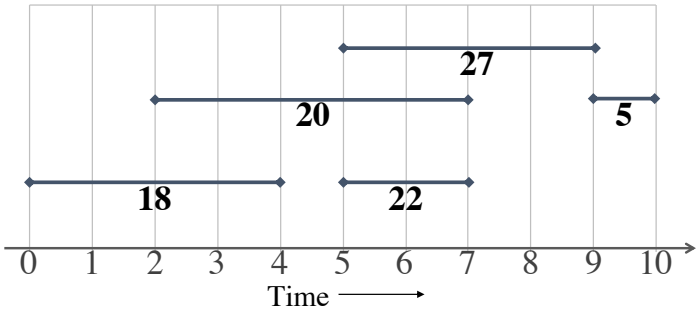
τ_m	S	M
[0,2)	S_1	18
[2,5)	S_1	20
[5,9)	S_1	27
[9,10)	S_1	5

Figure A.1: Example : TimeWarp operating on partitioned state and input message for an active vertex. (Aggregation : MIN)

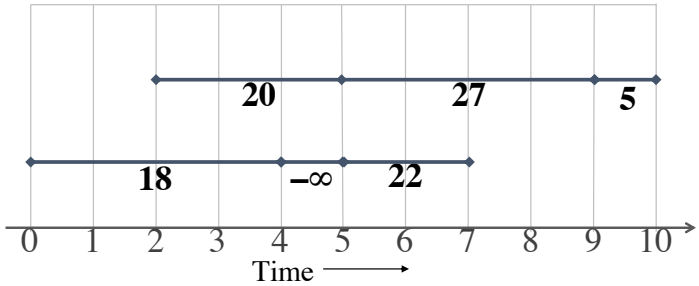
Sort-Merge Aggregation algorithm computes an aggregation result of larger interval by merging aggregate result for two smaller intervals. Figure A.2 illustrates working of sort-merge aggregation (shown in Algorithm A.1) for example shown in Figure A.1. Figure A.2(b) depicts the intermediate MIN aggregate result after the first merging step. In this step, intervals $[0, 4)$ and $[5, 7)$, $[2, 7)$, $[5, 9)$ and $[9, 10)$ are merged. Without loss of generality, $-\infty$ is used for intervals where aggregate value is not known, such as interval $[4, 5)$. Finally, Figure A.2(c) represents the interval after the second merge, $[0, 7)$ and $[2, 10)$ are merged.

```
1 void sortMergeAggregation(int[] timePoints, Message[] msgs[], int left,
    int right, Message identity) implements TimeWarp {
2
3    if (left+1 < right) {
4       int mid = left + (right - left)/2;
5       mid += (mid&1) - 1;
6
7       sortMergeAggregation(timePoints, msgs, left, mid, identity);
8       sortMergeAggregation(timePoints, msgs , mid+1, right, identity);
9       merge(timePoints, msgs, left, mid, right, identity);
10   }
11
12   Message aggregate(Message[] messages, Message identity) {
13   Message minMsg = identity;
14   for(Message msg : messages) {
15      minMsg = msg.getValue() < min.getValue() ? msg : minMsg;
16   } return minMsg;
17 }
```

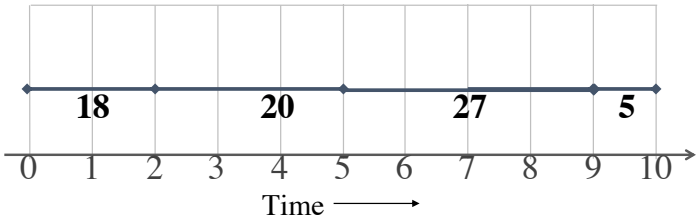
Algorithm A.1: Pseudo Code for Temporal Sort-Merge Aggregation



(a) Input Interval Messages for partitioned state S_1



(b) After first merging step



(c) After second merging step

Figure A.2: Example of merging for MIN aggregation. After each merging step, the values assigned to an interval is the MIN of the merged interval messages.

```
void merge(int[] timePoints, Message[] msgs, int left, int mid, int right,
    Message identity) {
   int n1 = mid - left + 1, L[] = new int [n1];
   int n2 = right - mid, R[] = new int [n2];
   Message L_M[] = new Message [n1], R_M[] = new Message [n2];

   for (int i=0; i<n1; ++i) {
      L[i] = timePoints[left + i];
      L_M[i] = msgs[left + i];
   }

   for (int j=0; j<n2; ++j) {
      R[j] = timePoints[mid + 1 + j];
      R_M[j] = msgs[mid + 1 + j];
   }

   int i = 0, j = 0, k = left;
   Multiset<Message> cache, include, exclude;

   while (i < n1 && j < n2 && !( L[i] < 0 || R[j] < 0 ) ) {
      if (L[i] <= R[j]) {
         timePoints[k] = L[i];
         if(i < n1-1) { include.add(L_M[i+1]); }
         exclude.add(L_M[i]);
         if(L[i] == R[j]) {
            if(j < n2-1) { include.add(R_M[j+1]); }
            exclude.add(R_M[j]);
            j++;
         } i++;
      } else {
         timePoints[k] = R[j];
         if(j < n2-1) { include.add(R_M[j+1]); }
         exclude.add(R_M[j]);
         j++;
      }
      msgs[k] = aggregate(cache.toArray(), identity);
      cache.add(include); cache.removeAll(exclude);
      k++;
   }
   while (i < n1 && !L[i]<0) {
      timePoints[k] = L[i];
```

```
41        msgs[k] = L_M[i];
42        i++; k++;
43    }
44    while (j < n2 && !R[j]<0) {
45        timePoints[k] = R[j];
46        msgs[k] = R_M[j];
47        j++;  k++;
48    }
49    while (k <= right) {
50        timePoints[k] = -∞;
51        msgs[k] = identity;
52        k++;
53    }
54 }
```

Algorithm A.2: Pseudo Code for Merge operation

Bibliography

[1] Apache Giraph. http://giraph.apache.org/. xi, 7, 48, 68

[2] The Graph Definition File Format. https://www.cs.nmsu.edu/~joemsong/software/ChiNet/GDF.pdf, 2019. [Accessed: 2019-12-28]. 51

[3] GEXF File Format. https://gephi.org/gexf/format/, 2019. [Accessed: 2019-12-28]. 51

[4] GML: A portable graph file format. https://www.fim.uni-passau.de/fileadmin/files/lehrstuhl/brandenburg/projekte/gml/gml-technical-report.pdf, 2019. [Accessed: 2019-12-28]. 51

[5] Charu Aggarwal and Karthik Subbian. Evolutionary network analysis: A survey. *ACM Comput. Surv.*, 47(1):10:1–10:36, May 2014. ISSN 0360-0300. doi: 10.1145/2601412. URL http://doi.acm.org/10.1145/2601412. 87

[6] Alaa Mahmoud, 2017. URL https://developer.ibm.com/dwblog/2017/detecting-complex-fraud-real-time-graph-databases/. [Accessed online on 12 August 2019]. 87

[7] James F. Allen. Maintaining knowledge about temporal intervals. *CACM*, 1983. 12

[8] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5):68:1–68:40, September 2017. ISSN 0360-0300. URL http://doi.acm.org/10.1145/3104031. 1

[9] Renzo Angles et al. G-CORE: A core for future graph query languages. In *ACM SIGMOD*, 2018. 11

[10] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. Linkbench: A database benchmark based on the facebook social graph. In *Proceedings of the 2013 International Conference on Management of Data*, SIGMOD '13, pages 1185–1196. ACM, 2013. ISBN 978-1-4503-2037-5. doi: 10.1145/2463676.2465296. URL http://doi.acm.org/10.1145/2463676.2465296. 58, 79

[11] Michael H. Böhlen, Richard Thomas Snodgrass, and Michael D. Soo. Coalescing in temporal databases. In *PVLDB*, 1996. 10

[12] Michael H. Böhlen, Anton Dignös, Johann Gamper, and Christian S. Jensen. Temporal data management – an overview. In Esteban Zimányi, editor, *Business Intelligence and Big Data*, pages 51–83, Cham, 2018. Springer International Publishing. ISBN 978-3-319-96655-7. 47

[13] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. A large time-aware graph. *SIGIR Forum*, 2008. 58

[14] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011. xi, 90, 91

[15] J. Byun, S. Woo, and D. Kim. Chronograph: Enabling temporal graph traversals for efficient information diffusion analysis over time. *IEEE TKDE*, 2019. 8, 11

[16] Francesco Cafagna and Michael H. Böhlen. Disjoint interval partitioning. *The VLDB Journal*, 26(3):447–466, June 2017. ISSN 1066-8888. doi: 10.1007/s00778-017-0456-7. URL https://doi.org/10.1007/s00778-017-0456-7. 10, 47

[17] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. Facilitating real-time graph mining. In *CloudDB*, 2012. 3, 8, 10

[18] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.*, 2015. 8, 64, 92

[19] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna P. Gummadi. Measuring User Influence in Twitter: The Million Follower Fallacy. In *In Proceedings of the 4th International AAAI Conference on Web and Social Media (ICWSM)*, 2010. 58

[20] Amit Chavan and Amol Deshpande. Dex: Query execution in a delta-based storage system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 171–186, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4197-4. doi: 10.1145/3035918.3064056. URL http://doi.acm.org/10.1145/3035918.3064056. 8

[21] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 85–98, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1223-3. doi: 10.1145/2168836.2168846. URL http://doi.acm.org/10.1145/2168836.2168846. 10

[22] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One Trillion Edges: Graph Processing at Facebook-scale. *PVLDB*, 2015. 57, 81, 86

[23] Lingyang Chu, Yanyan Zhang, Yu Yang, Lanjun Wang, and Jian Pei. Online density bursting subgraph detection from temporal graphs. *Proc. VLDB Endow.*, 12(13): 2353–2365, September 2019. ISSN 2150-8097. doi: 10.14778/3358701.3358704. URL https://doi.org/10.14778/3358701.3358704. 9

[24] Miguel E. Coimbra, Renato Rosa, Sérgio Esteves, Alexandre P. Francisco, and Luís Veiga. GraphBolt: Streaming Graph Approximations on Big Data. *arXiv e-prints*, art. arXiv:1810.02781, Oct 2018. 10

[25] Kenneth L Cooke and Eric Halsey. The shortest route through a network with time-dependent internodal transit times. *Journal of Mathematical Analysis and Applications*, 14(3):493 – 498, 1966. ISSN 0022-247X. doi: https://doi.org/10.1016/0022-247X(66)90009-6. URL http://www.sciencedirect.com/science/article/pii/0022247X66900096. 1

[26] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004. 48

[27] Camil Demetrescu, Andrew Goldberg, and Johnson David. 9th DIMACS implementation challenge. http://users.diag.uniroma1.it/challenge9/download.shtml, 2019. [Accessed: 2019-12-28]. 58

[28] Anton Dignös, Michael H. Böhlen, and Johann Gamper. Temporal alignment. In *SIGMOD*, 2012. 10

[29] R. Dindokar and Y. Simmhan. Adaptive partition migration for irregular graph algorithms on elastic resources. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 281–290, July 2019. doi: 10.1109/CLOUD.2019.00-28. 57

[30] Suhendry Effendy and Roland H.C. Yap. Investigations on rating computer sciences conferences: An experiment with the microsoft academic graph dataset. In *WWW Companion*, 2016. 58

[31] Wenfei Fan, Jianzhong Li, Jizhou Luo, Zijing Tan, Xin Wang, and Yinghui Wu. Incremental graph pattern matching. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 925–936, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0661-4. doi: 10.1145/1989323.1989420. URL http://doi.acm.org/10.1145/1989323.1989420. 88

[32] Dengfeng Gao, Christian S. Jensen, Richard T. Snodgrass, and Michael D. Soo. Join operations in temporal databases. *The VLDB Journal*, 2005. 10

[33] Junyang Gao, Pankaj K. Agarwal, and Jun Yang. Durable top-k queries on temporal data. *PVLDB*, 2018. 3, 9

[34] Neil Zhenqiang Gong, Wenchang Xu, Ling Huang, Prateek Mittal, Emil Stefanov, Vyas Sekar, and Dawn Song. Evolution of social-attribute networks: Measurements, modeling, and implications using google+. In *IMC*, 2012. 58

[35] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014. xi, 8, 64, 68, 87, 92

[36] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic management of data and computation in datacenters. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 75–88, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1924943.1924949. 88

[37] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 1988. 78

[38] Minyang Han and Khuzaima Daudjee. Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems. *PVLDB*, 8(9):950–961, 2015. 6, 56, 83, 86

[39] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 1:1–1:14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2704-6. doi: 10.1145/2592798.2592799. URL http://doi.acm.org/10.1145/2592798.2592799. 87

[40] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: a graph engine for temporal graph analysis. In *EuroSys*, 2014. 2, 3, 8, 9, 61

[41] F. Harary and G. Gupta. Dynamic graph models. *Math. Comput. Model.*, 1997. 8

[42] Bingsheng He, Mao Yang, Zhenyu Guo, Rishan Chen, Bing Su, Wei Lin, and Lidong Zhou. Comet: Batched stream processing for data intensive distributed computing. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 63–74, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0036-0. doi: 10.1145/1807128.1807139. URL http://doi.acm.org/10.1145/1807128.1807139. 88

[43] Safiollah Heidari, Yogesh Simmhan, Rodrigo N. Calheiros, and Rajkumar Buyya. Scalable graph processing frameworks: A taxonomy and open challenges. *ACM Comput. Surv.*, 51(3), June 2018. ISSN 0360-0300. doi: 10.1145/3199523. URL https://doi.org/10.1145/3199523. 1

[44] Jack Hessel, Chenhao Tan, and Lillian Lee. Science, AskScience, and BadScience: On the Coexistence of Highly Related Communities. In *In Proceedings of the 10th International AAAI Conference on Web and Social Media (ICWSM)*, 2016. 58

[45] P. Holme and J. Saramäki. Temporal networks. *Physics Reports*, 2012. 1, 3, 41

[46] Silu Huang, Ada Wai-Chee Fu, and Ruifeng Liu. Minimum spanning trees in temporal graphs. In *ACM SIGMOD*, 2015. 3, 9, 35, 38

[47] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. USENIX ATC, 2010. 47

[48] Alexandru Iosup et al. Ldbc graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *PVLDB*, 2016. 58, 79

[49] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. Time-evolving graph processing at scale. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, pages 5:1–5:6, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4780-8. doi: 10.1145/2960414.2960419. URL http://doi.acm.org/10.1145/2960414.2960419. 87, 90

[50] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E. Gonzalez, and Ion Stoica. TEGRA: Efficient ad-hoc analytics on time-evolving graphs. Technical report, UC Berkley, 2019. URL https://www.anand-iyer.com/papers/tegra.pdf. 10

[51] David S. Johnson and Catherine C. McGeoch, editors. *Network Flows and Matching: First DIMACS Implementation Challenge*. American Mathematical Society, 1993. 51

[52] Jun Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. In *Proceedings 17th International Conference on Data Engineering*, pages 51–60, April 2001. doi: 10.1109/ICDE.2001.914813. 47

[53] Maja Kabiljo. Improve the way we keep outgoing messages. http://issues.apache.org/jira/browse/GIRAPH-388, 2012. 55

[54] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998. ISSN 1064-8275. doi: 10.1137/S1064827595287997. URL http://dx.doi.org/10.1137/S1064827595287997. 80

[55] David Kempe, Jon Kleinberg, and Amit Kumar. Connectivity and inference problems for temporal networks. *Journal of Computer and System Sciences*, 64(4):820 – 842, 2002. ISSN 0022-0000. doi: https://doi.org/10.1006/jcss.2002.1829. URL http://www.sciencedirect.com/science/article/pii/S0022000002918295. 1

[56] Arijit Khan. Vertex-centric graph processing: Good, bad, and the ugly. In *EDBT*, 2017. 86

[57] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys

'13, page 169–182, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450319942. doi: 10.1145/2465351.2465369. URL https://doi.org/10.1145/2465351.2465369. x, 6, 7, 57

[58] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *IEEE ICDE*, 2013. 8, 11

[59] Udayan Khurana and Amol Deshpande. Storing and analyzing historical graph data at scale. In *EDBT*, 2016. 8, 11

[60] Nick Kline and Richard Thomas Snodgrass. Computing temporal aggregates. In *Proceedings of the Eleventh International Conference on Data Engineering*, ICDE '95, pages 222–231, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-6910-1. URL http://dl.acm.org/citation.cfm?id=645480.757696. 47

[61] Lauri Kovanen, Márton Karsai, Kimmo Kaski, János Kertész, and Jari Saramäki. Temporal motifs in time-dependent networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2011. 3

[62] Krishna Kulkarni and Jan-Eike Michels. Temporal features in SQL:2011. *SIGMOD Rec.*, 2012. 9, 11, 12

[63] Pradeep Kumar and H. Howie Huang. Graphone: A data store for real-time analytics on evolving graphs. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 249–263, Boston, MA, February 2019. USENIX Association. ISBN 978-1-939133-09-0. URL https://www.usenix.org/conference/fast19/presentation/kumar. 11

[64] Rohit Kumar and Toon Calders. 2SCENT: An efficient algorithm for enumerating all simple temporal cycles. *PVLDB*, 2018. 9, 40

[65] Alan G. Labouseur, Jeremy Birnbaum, Paul W. Olsen, Sean R. Spillane, Jayadevan Vijayan, Jeong-Hyon Hwang, and Wook-Shin Han. The G* graph database: efficiently managing large distributed dynamic graphs. *Distr. and Parallel Datab.*, 2015. 8

[66] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, KDD '05, page 177–187, New York, NY, USA, 2005. Association for Computing Machinery. ISBN

159593135X. doi: 10.1145/1081870.1081893. URL https://doi.org/10.1145/1081870.1081893. 1

[67] Lei Li, Wen Hua, Xingzhong Du, and Xiaofang Zhou. Minimal on-road time route scheduling on time-dependent graphs. *Proc. VLDB Endow.*, 10(11):1274–1285, August 2017. ISSN 2150-8097. doi: 10.14778/3137628.3137638. URL https://doi.org/10.14778/3137628.3137638. 36

[68] Mo Li, Junchang Xin, Zhiqiong Wang, and Huilin Liu. Accelerating minimum temporal paths query based on dynamic programming. In Jianxin Li, Sen Wang, Shaowen Qin, Xue Li, and Shuliang Wang, editors, *Advanced Data Mining and Applications*, pages 48–62. Springer International Publishing, 2019. 9

[69] P. Liakos, K. Papakonstantinopoulou, and A. Delis. Realizing memory-optimized distributed graph processing. *IEEE Transactions on Knowledge and Data Engineering*, 30 (4):743–756, April 2018. ISSN 2326-3865. doi: 10.1109/TKDE.2017.2779797. 57

[70] Wouter Lightenberg, Yulong Pei, George Fletcher, and Mykola Pechenizkiy. Tink: A temporal graph analytics library for apache flink. In *WWW Comp.*, 2018. 3, 9, 64

[71] Paul Liu, Austin R. Benson, and Moses Charikar. Sampling methods for counting temporal motifs. In *WSDM*, 2019. 9, 58

[72] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *PVLDB*, 2012. 7, 8, 22, 92

[73] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. LLAMA: Efficient graph analytics using large multiversioned arrays. In *IEEE ICDE*, 2015. 8, 11

[74] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *ACM SIGMOD*, 2010. 2, 3, 6, 7, 8, 15, 22, 28, 29, 34, 51, 87

[75] Claudio Martella, Roman Shaposhnik, and Dionysios Logothetis. *Giraph in Action*. Manning, 2015. x, 45, 53

[76] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 2015. 3, 8, 86

[77] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. Immortalgraph: A system for storage and analysis of temporal graphs. *Trans. Storage*, 2015. 8, 11

[78] Vera Zaychik Moffitt and Julia Stoyanovich. Temporal graph algebra. In *DBPL*, 2017. 3, 11, 61

[79] B. Moon, I. F. V. Lopez, and V. Immanuel. Scalable algorithms for large temporal aggregation. In *IEEE ICDE*, 2000. 47

[80] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522739. URL http://doi.acm.org/10.1145/2517349.2522739. 87

[81] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. In *WWW*, 1998. 31

[82] Ashwin Paranjape, Austin Benson, and Leskovec. Motifs in temporal networks. In *WSDM*, 2017. 9, 40

[83] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 251–264, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1924943.1924961. 88

[84] Abdul Quamar, Amol Deshpande, and Jimmy Lin. Nscale: Neighborhood-centric large-scale graph analytics in the cloud. *The VLDB Journal*, 25(2):125–150, April 2016. ISSN 1066-8888. doi: 10.1007/s00778-015-0405-2. URL https://doi.org/10.1007/s00778-015-0405-2. 7, 8, 86

[85] G. Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 502–510, New York, NY, USA, 1993. ACM. ISBN 0-89791-560-7. doi: 10.1145/158511.158710. URL http://doi.acm.org/10.1145/158511.158710. 88

[86] Mark Redekopp, Yogesh Simmhan, and Viktor K. Prasanna. Optimizations and analysis of bsp graph processing models on public clouds. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, page 203–214, USA, 2013. IEEE Computer Society. ISBN 9780769549712. doi: 10.1109/IPDPS.2013.76. URL https://doi.org/10.1109/IPDPS.2013.76. 86

[87] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. On querying historical evolving graph sequences. *PVLDB*, 2011. 8, 11

[88] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522740. URL http://doi.acm.org/10.1145/2517349.2522740. 7, 8, 87

[89] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 410–424, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3834-9. doi: 10.1145/2815400.2815408. URL http://doi.acm.org/10.1145/2815400.2815408. 87

[90] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 22:1–22:12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1921-8. doi: 10.1145/2484838.2484843. URL http://doi.acm.org/10.1145/2484838.2484843. 6

[91] Semih Salihoglu and Jennifer Widom. Optimizing graph algorithms on pregel-like systems. *PVLDB*, 2014. 7, 8, 17, 28, 30

[92] S. Sallinen, R. Pearce, and M. Ripeanu. Incremental graph processing for on-line analytics. In *2019 IEEE International Parallel and Distributed Processing Symposium*, 2019. 10, 88

[93] Betty Salzberg and Vassilis J. Tsotras. Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, 1999. 10

[94] K. Semertzidis and E. Pitoura. Top-k Durable Graph Pattern Queries on Temporal Graphs. *IEEE TKDE*, 2019. 11

[95] Konstantinos Semertzidis, Evaggelia Pitoura, and Kostas Lillis. Timereach: Historical reachability queries on evolving graphs. In *EDBT*, 2015. 11

[96] N. Sengupta, A. Bagchi, M. Ramanath, and S. Bedathur. Arrow: Approximating reachability using random walks over web-scale graphs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 470–481, April 2019. doi: 10.1109/ICDE.2019.00049. 9

[97] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. Tornado: A system for real-time iterative analysis over evolving data. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 417–430, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3531-7. doi: 10.1145/2882903.2882950. URL http://doi.acm.org/10.1145/2882903.2882950. 10, 90

[98] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, May 2010. doi: 10.1109/MSST.2010.5496972. 8, 51

[99] Y. Simmhan, N. Choudhury, C. Wickramaarachchi, A. Kumbhare, M. Frincu, C. Raghavendra, and V. Prasanna. Distributed programming over time-series graphs. In *IPDPS*, 2015. 3, 9, 63, 87

[100] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. Goffish: A sub-graph centric framework for large-scale graph analytics. In *European Conference on Parallel Processing*, pages 451–462. Springer, 2014. URL https://doi.org/10.1007/978-3-319-09873-9_38. 6, 7, 8, 92

[101] G. Slivinskas, C. S. Jensen, and R. T. Snodgrass. Query plans for conventional and temporal queries involving duplicates and ordering. In *IEEE ICDE*, 2000. 2

[102] Richard Thomas Snodgrass, Ilsoo Ahn, Gadi Ariav, Don Batory, James Clifford, Curtis E. Dyreson, Ramez Elmasri, Fabio Grandi, Christian S. Jensen, Wolfgang Käfer, Nick Kline, Krishna Kulkarni, T. Y. Cliff Leung, Nikos Lorentzos, John F. Roddick, Arie Segev, Michael D. Soo, and Suryanarayana M. Sripada. TSQL2 Language Specification. *SIGMOD Rec.*, 1994. 10

[103] M. D. Soo, R. T. Snodgrass, and C. S. Jensen. Efficient evaluation of the valid-time natural join. In *IEEE ICDE*, 1994. 24, 25

[104] Balázs Szendroi and Gábor Csányi. Polynomial epidemics and clustering in contact networks. *Proceedings of the Royal Society of London. Series B: Biological Sciences*, 2004. 3

[105] Stephen Taylor, Jerrell R Watts, Marc A Rieffel, and Michael E Palmer. The concurrent graph: basic technology for irregular problems. *IEEE Parallel & Distributed Technology: Systems & Applications*, 4(2):15–25, 1996. 7

[106] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 425–440, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338349. doi: 10.1145/2815400.2815410. URL https://doi.org/10.1145/2815400.2815410. 86

[107] Manuel Then, Timo Kersten, Stephan Günnemann, Alfons Kemper, and Thomas Neumann. Automatic algorithm transformation for efficient multi-snapshot analytics on temporal graphs. *PVLDB*, 2017. 2, 3, 8, 61

[108] Paul Anye Tuma. *Implementing historical aggregates in TempIS*. PhD thesis, Wayne State University, 1993. 47

[109] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 1990. 6

[110] David C Van Essen, Stephen M Smith, Deanna M Barch, Timothy EJ Behrens, Essa Yacoub, Kamil Ugurbil, Wu-Minn HCP Consortium, et al. The wu-minn human connectome project: an overview. *Neuroimage*, 80:62–79, 2013. 1

[111] Vinod Kumar Vavilapalli et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SOCC*, 2013. 50

[112] Shiv Verma, Luke M. Leslie, Yosub Shin, and Indranil Gupta. An experimental comparison of partitioning strategies in distributed graph processing. *Proc. VLDB Endow.*, 10(5):493–504, January 2017. ISSN 2150-8097. doi: 10.14778/3055540.3055543. URL https://doi.org/10.14778/3055540.3055543. 81

[113] Shiv Verma, Luke M. Leslie, Yosub Shin, and Indranil Gupta. An experimental comparison of partitioning strategies in distributed graph processing. *Proc. VLDB Endow.*, 10(5):493–504, January 2017. ISSN 2150-8097. doi: 10.14778/3055540.3055543. URL https://doi.org/10.14778/3055540.3055543. 53

[114] Keval Vora, Rajiv Gupta, and Guoqing Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 237–251, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4465-4. doi: 10.1145/3037697.3037748. URL http://doi.acm.org/10.1145/3037697.3037748. 10, 88, 90

[115] C. Wang, J. Tang, J. Sun, and J. Han. Dynamic social influence analysis through time-dependent factor graphs. In *2011 International Conference on Advances in Social Networks Analysis and Mining*, pages 239–246, July 2011. doi: 10.1109/ASONAM.2011.116. 1

[116] Yishu Wang, Ye Yuan, Yuliang Ma, and Guoren Wang. Time-dependent graphs: Definitions, applications, and algorithms. *Data Science and Engineering*, 4(4):352–366, Dec 2019. ISSN 2364-1541. doi: 10.1007/s41019-019-00105-0. URL https://doi.org/10.1007/s41019-019-00105-0. 1

[117] H. Wu, J. Cheng, Y. Lu, Y. Ke, Y. Huang, D. Yan, and H. Wu. Core decomposition in large temporal graphs. In *2015 IEEE International Conference on Big Data (Big Data)*, 2015. 9

[118] H. Wu, J. Cheng, Y. Ke, S. Huang, Y. Huang, and H. Wu. Efficient algorithms for temporal path computation. *IEEE Transactions on Knowledge and Data Engineering*, 28 (11):2927–2942, Nov 2016. ISSN 2326-3865. doi: 10.1109/TKDE.2016.2594065. 61

[119] H. Wu, Y. Huang, J. Cheng, J. Li, and Y. Ke. Reachability and time-based path queries in temporal graphs. In *IEEE ICDE*, 2016. 9, 35, 39

[120] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. Path problems in temporal graphs. *PVLDB*, 2014. 3, 4, 9, 19, 35, 36, 62

[121] B. BUI XUAN, A. FERREIRA, and A. JARRY. Computing shortest, fastest, and foremost journeys in dynamic networks. *International Journal of Foundations of Computer Science*, 14(02):267–285, 2003. doi: 10.1142/S0129054103001728. URL https://doi.org/10.1142/S0129054103001728. 35

[122] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, October 2014.

ISSN 2150-8097. doi: 10.14778/2733085.2733103. URL http://dx.doi.org/10.14778/2733085.2733103. 6

[123] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *PVLDB*, 2014. 7, 8, 17, 28, 87

[124] Hao Yin, Austin R. Benson, and Jure Leskovec. The local closure coefficient: A new perspective on network clustering. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, WSDM '19, page 303–311, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450359405. doi: 10.1145/3289600.3290991. URL https://doi.org/10.1145/3289600.3290991. 41

[125] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010. 8, 10, 64

[126] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. Wonderland: A novel abstraction-based out-of-core graph processing system. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 608–621, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450349116. doi: 10.1145/3173162.3173208. URL https://doi.org/10.1145/3173162.3173208. 86

[127] Tianming Zhang, Yunjun Gao, Linshan Qiu, Lu Chen, Qingyuan Linghu, and Shiliang Pu. Distributed time-respecting flow graph pattern matching on temporal graphs. *World Wide Web*, 2019. 9

[128] Kangfei Zhao and Jeffrey Xu Yu. All-in-one: Graph processing in rdbmss revisited. In *SIGMOD*, 2017. 51

[129] Chang Zhou, Jun Gao, Binbin Sun, and Jeffrey Xu Yu. Mocgraph: Scalable distributed graph processing using message online computing. *Proc. VLDB Endow.*, 8(4):377–388, December 2014. ISSN 2150-8097. doi: 10.14778/2735496.2735501. URL https://doi.org/10.14778/2735496.2735501. 57

[130] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium*

on Operating Systems Design and Implementation (OSDI 16), pages 301–316, Savannah, GA, November 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhu. 86

[131] Esteban Zimányi. Temporal Aggregates and Temporal Universal Quantification in Standard SQL. *SIGMOD Rec.*, 2006. 10