TEGRA: Efficient Ad-Hoc Analytics on Evolving Graphs

Anand Padmanabha Iyer, *Microsoft Research and University of California, Berkeley;*
Qifan Pu, *Google;* Kishan Patel, *Two Sigma;* Joseph E. Gonzalez
and Ion Stoica, *University of California, Berkeley*

https://www.usenix.org/conference/nsdi21/presentation/iyer

This paper is included in the
Proceedings of the 18th USENIX Symposium on
Networked Systems Design and Implementation.

April 12–14, 2021

978-1-939133-21-2

Open access to the Proceedings of the
18th USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

TEGRA: Efficient Ad-Hoc Analytics on Evolving Graphs

Anand Padmanabha Iyer[⋆†], Qifan Pu[◇], Kishan Patel[‡], Joseph E. Gonzalez[†], Ion Stoica[†]

[⋆]*Microsoft Research* [◇]*Google* [‡]*Two Sigma* [†]*University of California, Berkeley*

Abstract

Several emerging *evolving graph* application workloads demand support for efficient ad-hoc analytics—the ability to perform ad-hoc queries on arbitrary time windows of the graph. We present TEGRA, a system that enables efficient ad-hoc window operations on evolving graphs. TEGRA allows efficient access to the state of the graph at arbitrary windows, and significantly accelerates ad-hoc window queries by using a compact in-memory representation for both graph and intermediate computation state. For this, it leverages persistent data structures to build a versioned, distributed graph state store, and couples it with an incremental computation model which can leverage these compact states. For users, it exposes these compact states using Timelapse, a natural abstraction. We evaluate TEGRA against existing evolving graph analysis techniques, and show that it significantly outperforms state-of-the-art systems (by up to 30×) for ad-hoc window operation workloads.

1 Introduction

Graph-structured data is on the rise, in size, complexity and dynamism [1, 61]. This growth has spurred the development of a large number of graph processing systems [16, 17, 19, 26, 27, 30, 33, 39, 42, 51, 54, 57, 59, 60, 68] in both academia and the open-source community. By leveraging specialized abstractions and careful optimizations, these systems have the ability to analyze large, *static* graphs, some even in the order of a trillion edges [20].

However, real-world graphs are seldom static. Consider, the familiar example of social network graphs such as in Facebook and Twitter. In these networks, "friends" relations and tweets with "mentions" are created continuously resulting in the graph's constant evolution. Similarly, each new call in a cellular network connects devices with receivers and can be used in real-time for network monitoring and diagnostics [31]. Additionally, emerging applications such as connected cars [13], real-time fraud detection [9], and disease analysis [23] all produce such graph data. Analyzing these *time-evolving* graphs can be useful, from scientific and commercial perspectives, and is often desired [61].

In this paper, we focus on the problem of *efficient ad-hoc window operations* on evolving graphs—the ability to perform ad-hoc queries on arbitrary time windows (i.e., segments in time) either in the past or in real-time. To illustrate the need for such operations, consider two examples. In the first, a financial expert wishes to improve her fraud-detection algorithm. For this, she retrieves the *complete states of the graph at different segments in time* to train and test variants of her algorithm. In the second example, a network administrator wishes to diagnose a transient failure. To do so, she retrieves a *series of snapshots*[1] *of the graph* before and after the failure, and runs a handful of queries on them. She iteratively refines the queries until she comes up with a hypothesis. In such scenarios, neither the queries nor the windows on which the queries would be run are predetermined.

To efficiently perform ad-hoc window operations, a graph processing system should provide two key capabilities. First, it must be able to quickly retrieve arbitrary size windows starting at arbitrary points in time. There are two approaches to provide this functionality. The first is to store a *snapshot* every time the graph is updated, i.e., a vertex or edge is added or deleted. While this allows one to efficiently retrieve the state of the graph at *any* point in the past, it can result in prohibitive overhead. An alternative is to store only the changes to the graph and reconstruct a snapshot on demand. This approach is space efficient, but can incur high latency, as it needs to re-apply all updates to reconstruct the requested snapshot(s). Thus, there is a fundamental trade-off between in-memory storage and retrieval time.

Second, we must be able to efficiently execute queries (e.g., connected components) not only on a single window, but also across multiple related windows of the graph. Existing systems, such as Chronos [30] allows executing queries on a single window, while Differential Dataflow [54] and GraphBolt [45] support continuously updating queries over sliding windows. However, none of the systems support efficient execution of queries across multiple windows, as they do not have the ability to share the computation state across windows and computations. This fundamental limitation of existing systems arises from their inability to efficiently store intermediate state from within a query for later reuse.

We present TEGRA[2], a system that enables efficient ad-hoc window operations on time-evolving graphs. TEGRA is based on two key insights about such real-world evolving graph workloads: *(1) during ad-hoc analysis graphs change slowly over time relative to their size , and (2) queries are frequently applied to multiple windows relatively close by in time.*

[1] A snapshot is a full copy of the graph, and can be viewed as a window of size zero. Non-zero windows have several snapshots.

[2] for **T**ime **E**volving **GR**aph **A**nalytics.

Leveraging these insights TEGRA is able to significantly accelerate window queries by reusing both storage and computation across queries on related windows. TEGRA solves the storage problem through a highly efficient, distributed, versioned *graph state store* which compactly represents graph snapshots in-memory as logically separate versions that are efficient for arbitrary retrieval. We design this store using persistent (functional) data-structures that lets us heavily share common parts of the graph thereby reducing the storage requirements by several orders of magnitude (§5). Second, to improve the performance of ad-hoc queries, we introduce an efficient in-memory representation of intermediate state that can be stored in our graph state store and enables non-monotonic[3] incremental computations. This technique leverages the computation pattern of the familiar graph-parallel models to create compact intermediate state that can be used to eliminate redundant computations across queries. (§4).

TEGRA exposes these compact persistent snapshots of the graph and computation state using a logical abstraction named *Timelapse*, which hides the intricacies of state management and sharing from the developer. At a high level, a timelapse is formed by a sequence of graph snapshots, starting from the original graph. Viewing the time-evolving graph as consisting of a sequence of independent static *snapshots* of the entire graph makes it easy for the developer to express a variety of computation patterns naturally, while letting the system optimize computations on those snapshots with much more efficient incremental computations (§3). Finally, since Timelapse is backed by our persistent graph store, users and computations always work on independent *versions* of the graph, without having to worry about consistency issues. Using these, TEGRA outperforms state-of-the-art systems significantly on ad-hoc window operation workloads (§7).

In summary, we make the following contributions:

- We present TEGRA, a time-evolving graph processing system that enables efficient ad-hoc window operations on both historic and live data. To achieve this, TEGRA shares storage, computation and communication across queries by compactly representing the evolving graph and intermediate computation state in-memory.
- We propose *Timelapse*, a new abstraction for time-evolving graph processing. TEGRA exposes timelapse to the developer using a simple API that can encompass many time-evolving graph operations. (§3)
- We design *DGSI*, an efficient distributed, versioned property graph store that enables timelapse APIs to perform efficient operations. (§5)
- Leveraging timelapse and DGSI, we present an incremental graph computation model which supports non-monotonic computations across (non-contiguous) windows. (§4)

[3] Allows vertex/edge deletions, additions and modifications on any graph algorithm implemented in a graph-parallel fashion.

2 Background & Challenges

We begin with a brief background on graph-parallel systems (§2.1) and then describe various types of time-evolving graph workloads (§2.2). The limitations of existing systems are discussed (§2.3) before we layout the challenges in enabling efficient ad-hoc analytics on evolving graphs (§2.4).

2.1 Graph-Parallel Systems

Most general purpose graph systems provide a *graph-parallel* abstraction for performing computations. In the graph-parallel abstraction, a user-defined program is run (in parallel) on every entity in the graph, who then change their state depending on the neighborhood. This process is iteratively done until convergence. Thus, the graph-parallel abstraction lets the end-developer view distributed graph computations as simpler entity centric computations, leaving the burden of orchestration to the system. The simplest of the graph-parallel abstraction is a vertex-centric model [46], where every vertex independently runs the user program. Several other forms have been proposed, such as the graph centric models [66], edge centric models [59] and the more recent subgraph centric models [57]. In all these models, the basic form of computation is implemented as message exchanges between the entities and their corresponding state changes. Communication is enabled either via shared memory model [63] or message passing interface [43]. PowerGraph [27] introduced the **Gather-Apply-Scatter (GAS)** model, a popular vertex-centric model adopted by many open-source graph processing systems, where a vertex program is represented as three conceptual phases: **gather** phase that collects information about adjacent vertices and edges and applies a function on them, **apply** phase that uses the function's output to update the vertex, and **scatter** phase that uses the new vertex value to update adjacent edges. To perform a graph algorithm computation, the system iteratively applies these phases until convergence. TEGRA focuses on the GAS model. (§6).

2.2 Time-evolving Graph Workloads

Time-evolving graph workloads, an important graph workload [61], can be classified into three categories:

Temporal Queries: Here, an analyst is querying the graph at different points in the past and evaluates how the result changes over time. Examples are *"How many friends did Alice have in 2017?"* or *"How did Alice's friend circle change in the last three years?"*. Such queries may have time windows of the form $[T-\delta, T]$ and are performed on offline data, executed in a batch fashion.

Streaming/Online Queries: These workloads are aimed at keeping the result of a graph computation up-to-date as new data arrives (i.e., $[Now-\delta, Now]$). For example, the analyst may ask *"What are the trending topics now?"*, or use a moving window (e.g., *"What are the trending topics in the last 10 minutes?"*). These focus on the most recent data, thus streaming systems operate on the live graph.

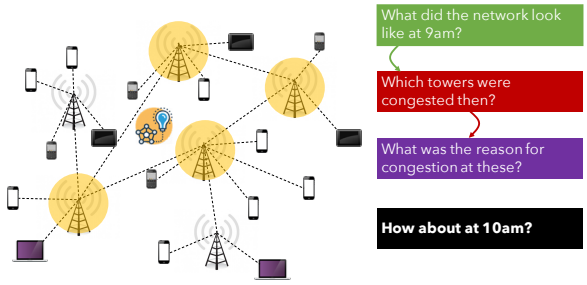

Figure 1: Carol, a network administrator, diagnoses issues by performing ad-hoc queries on *snapshots* of the network at disjoint but *close-by* points in time.

Ad-hoc Queries: In these workloads, an analyst is likely to explore the graph by performing ad-hoc queries on arbitrary windows. Below, we illustrate a real-world use case. We omit some details and use fictitious names for anonymity.

Carol is a network administrator at a large cellular network operator in the United States [58]. Her job is to manage several thousands of wireless base stations, deployed across a large geographic region. When problems occur, Carol is tasked with finding the reason for the issue and fix them. For instance, she may be trying to answer *"What is the reason for poor download throughput for (several) users at 9:00am?"*.

Cellular network operators collect extensive data (several terabytes per day) from their network, and previous studies have shown the benefits of viewing cellular network analytics as a time-evolving graph problem [31]. Carol might start by asking *"What did the network look like at 9am?"*, when the problem was reported. The query returns a *graph view* of the network, depicted in fig. 1, where there are several base stations (towers) serving many users. Carol doubts congestion as the cause for low throughput, so the next query is *"Which towers were congested at this point?"* that returns a subset of towers from the original answer. Based on her extensive domain knowledge, she knows that temporary congestion in some towers do not cause persistent poor throughput, so she is interested in learning if *clusters* of towers were congested. To do so, she runs a *connected component* algorithm on the network graph. Then, to confirm her hypothesis, she asks *"How about at 10am?"* meaning to repeat the entire analysis again, but now on a different subset of the data. By the time Carol finishes her investigation, she has retrieved *100s of different subsets of data*, each depicting a *snapshot of the network* at some *disjoint, random* point in time around the failure, conducted exploratory analysis to test several of her hypothesis including one where she runs connected components.

Thus, in ad-hoc workloads, not only does the analyst need to access arbitrary windows, but also the queries and the windows on which they are executed are determined just-in-time (i.e., not predetermined). Further, the analyst applies the same query to multiple (close-by, discontinuous) windows.

2.3 State of the Art & Limitations

Recent work in graph systems has made considerable progress in the area of evolving graph processing. (§8)

Temporal analysis engines (e.g., **Chronos** [30], **ImmortalGraph** [50]) operate on *offline data* and focus on executing queries on one or a sequence of snapshots in the graph's history. Upon execution of a query, these systems load the relevant history of the graph and utilize a pre-processing step to create an in-memory layout that is efficient for analysis. Such preprocessing can often dominate the algorithm execution time [44]. As a result, these systems are tuned for operating on a large number of snapshots in each query (e.g., temporal changes over months or year), and are efficient in such cases. Fundamentally, the in-memory representation in these systems cannot support updates. Additionally, these systems do not allow updating the results of a query.

Proposals such as GraphOne [35,36] and Aspen [21] focus on providing *efficient storage for streaming computations*. These (typically single-machine) systems allow only storing a few recent versions of the graph and do not support storing intermediate state, or updating the results of a previous query. **GraphOne** [35,36] combines circular edge logs and adjacency store to allow storing a few recent versions of the graph and dual versioning to decouple computation from storage. However, the use of *chaining* (with compaction) in the adjacency store to enable versioning introduces an ordering constraint among the versions, and traversing (and applying operations e.g., deletions) is necessary to retrieve a specific snapshot. For ad-hoc analysis where arbitrary changes maybe applied on a version, this may be expensive and fundamentally difficult. **Aspen** [21] leverages functional/persistent datastructures to preserve previous version of the graph upon mutation and presents C-trees, a highly compressed functional tree that can store graphs efficiently (they do not allow property graphs). This is similar in spirit to our use of persistent datastructures in designing DGSI (§5). However, C-trees are tuned for streaming workloads where there is one (or a few) previous version(s) and thus employ aggressive garbage collection for efficiency. When large number of versions are required, main memory becomes a bottleneck and thus it is necessary to have a persistent storage based hybrid store (e.g., DGSI).

Streaming systems (e.g., Kineograph [19], Differential-Dataflow [49], Kickstarter [67], GraphBolt [45]) operate on *live data* and allow query results to be updated *incrementally* (rather than doing a full computation) when *new* data arrives. These systems only allow queries on the live graph, and do not support ad-hoc retrieval of previous state. Additionally, the incremental computation is tied to the live state of the graph, and cannot be utilized over multiple windows.

Differential Dataflow (DD) [49] is a distributed system that allows general, non-monotonic incremental computations using special versions of operators. Each operator stores "differences" to its input and produces the corresponding dif-

ferences in output (hence the full output is not materialized), automatically incrementalizing algorithms written using them. While this technique is very efficient for real-time streaming queries, incorporating ad-hoc window operations in it is fundamentally hard. Since the computation model is based on the operators maintaining state (*differences* to their input and output) indexed by data (rather than time), accessing a particular snapshot can require a full scan of the maintained state. Further, since every operator needs to maintain state, the system accumulates large state over time which must be compacted (at the expense of forgoing the ability to retrieve the past). Finally, intermediate state of a query is cleared once completed and storing these efficiently for reuse is an open question.

GraphBolt [45], a *single-machine* streaming system, presents a dependency driven "refinement" based computation model for (non-monotonic) incremental computations that tracks dependency information as aggregation values at vertices thus reducing the state requirements to O|V| (in contrast to DD's O|E|). Users can implement incremental algorithms by defining user-defined refinement functions (e.g., `repropagate`, `retract` and `propagate`) whose implementations are algorithm specific (e.g., Algorithm 3 in [45] for PageRank), and iteratively refines aggregation values. GraphBolt only stores the value aggregations for the current snapshot of the graph and does not present a solution for storing multiple versions of aggregations or efficiently using/operating on them. Thus, GraphBolt does not support ad-hoc analysis. Building ad-hoc support requires building a state store, similar to the solution we present (DGSI), tuned for GraphBolt's computation model and exposing the right APIs.

2.4 Challenges

Several challenges stand in the way of enabling efficient ad-hoc analytics on evolving graphs. First is the ability to efficiently *store* and *retrieve* snapshots of the graph at arbitrary time windows. Second, we must be able to *compactly represent large amounts of computation state and use it to accelerate future queries, across multiple windows* using a computation model that can leverage the state efficiently. Finally, the system should be able to provide users a natural and intuitive way to operate on evolving graphs. Based on this, our solution, TEGRA, consists of three components:

Timelapse Abstraction (§3): In TEGRA, users interact with time-evolving graphs using the *timelapse* abstraction, which logically represents the evolving graph as a sequence of *static*, *immutable* graph snapshots (fig. 2). TEGRA exposes this abstraction via a simple API that allows users to save/retrieve/-query the materialized *state* of the graph at any point.

Computational Model (§4): TEGRA proposes a computation model that allows *ad-hoc queries across windows to share computation and communication*. The model stores compact intermediate state as a timelapse, and uses it to perform general, *non-monotonic* incremental computations.

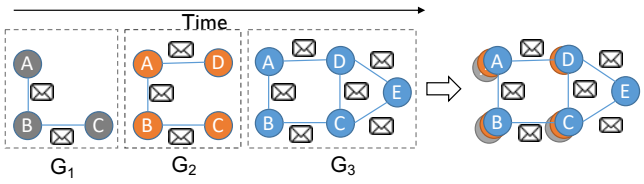

Figure 2: A timelapse of graph *G* consisting of three snapshots. For temporal analytics, instead of applying graph-parallel operations independently on each snapshot (left), timelapse enables them to be applied to all snapshots in parallel (right).

Distributed Graph Snapshot Index (§5): TEGRA stores evolving graphs, intermediate computation state and results in DGSI, an efficient indexed, distributed, versioned property graph store which *shares storage* between versions of the graph. Such decoupling of state from queries and operators allow TEGRA to share it across queries and users.

3 Timelapse Abstraction & API

TEGRA introduces *Timelapse* as a new abstraction for time-evolving graph processing that enables efficient ad-hoc analytics. The goal of timelapse is to provide the end-user with a simple, natural interface to run queries on time-evolving graphs, while giving the system opportunities for efficiently executing those queries. In timelapse, TEGRA *logically* represents a time-evolving graph as a sequence of immutable, static graphs (fig. 2), each of which we refer to as *snapshot* in the rest of this paper. A snapshot depicts a consistent state of the graph at a particular instance in time. TEGRA uses the popular property graph model [26], where vertices and edges in the graph are associated with arbitrary properties, to represent each snapshot in the timelapse. For the end-user, timelapse provides the abstraction of having access to a *materialized* snapshot at any point in the history of the graph. This enables the usage of the familiar static graph processing model in evolving graphs (e.g., queries on arbitrary snapshot).

Timelapses are created in TEGRA in two ways—by the system and by the users. When a new graph is introduced to the system, a timelapse is created for it that contains a single snapshot of the graph. Then, as the graph evolves, more snapshots are added to the timelapse. Similarly, users may create timelapses while performing analytics. Because snapshots in a timelapse are immutable, any operation on them creates new snapshots as a result (e.g., a query on a snapshot results in another snapshot as a result). Such newly created snapshots during an analytics session may be added to an existing timelapse, or create a new one depending on the kind of operations performed. For instance, for an analyst performing what-if analysis by introducing artificial changes to the graph, it is logical to create a new timelapse. Meanwhile, snapshots created as a result of updating a query result should ideally be added to the same timelapse. The system does not impose restrictions on how users want to book-keep timelapses. Instead, it simply tracks their lineage and allows

`save(id): id`	Save the state of the graph as a snapshot in its timelapse. ID can be autogenerated. Returns the id of the saved snapshot.
`retrieve(id): snapshot`	Return one or more snapshots from the timelapse. Allows simple matching on the id.
`diff(snapshot, snapshot): delta`	Difference between two snapshots in the timelapse. (§4)
`expand(candidates): subgraph`	Given a list of candidate vertices, expand the computation scope by marking their 1-hop neighbors. Used for implementing incremental computations (§4)
`merge(snapshot, snapshot,func): snapshot`	Create a new snapshot using the union of vertices and edges of two snapshots. For common vertices, run func to compute their value. Used for implementing incremental computations (§4)

Table 1: TEGRA exposes Timelapse via simple APIs.

users to efficiently operate on them. We describe how TEGRA implements timelapses in §5.3.

Since timelapse logically represents a sequence of related graph snapshots, it is intuitive to expose the abstraction using the same semantics as that of static graph. In TEGRA, users interact with timelapses using a language integrated API. It extends the familiar `Graph` interface, common in static graph processing systems, with a simple set of additional operations, listed in table 1. This enables users to continue using existing static graph operations on any snapshot in the timelapse obtained using the `retrieve` API. **(§6.2)**

3.1 Evolving Graph Analytics Using Timelapse

In addition to providing ad-hoc access to any snapshot, timelapse is also useful in enabling efficient time-evolving graph analysis. The natural way to do graph computations over the time dimension is to iterate over a sequence of snapshots. For instance, an analyst interested in executing the `degrees` query on three snapshots, G_1, G_2 and G_3 depicted in fig. 2 can do:

```
for(id <- Array(G1,G2,G3))
  result = G.retrieve(id).degrees
```

However, applying the same operation on multiple snapshots of a time-evolving graph independently is inefficient. In graph-parallel systems (§2), `degrees()` computation is typically implemented using a user-defined program where every vertex sends a message with value 1 to their neighbors, and all vertices adding up their incoming message values. Such message exchange accounts for a non-trivial portion of the analysis time [62]. In the earlier example, sequentially applying the query to each snapshot results in 11 messages of which 5 are duplicates (fig. 2).

To avoid such inefficiencies, timelapse allows access to the *lineage* of graph entities. That is, it provides efficient

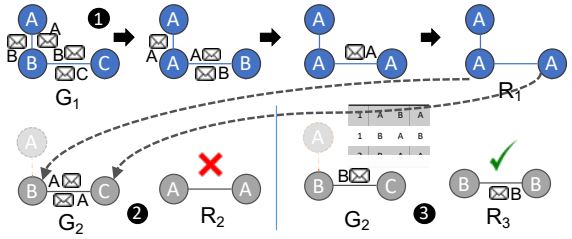

Figure 3: ❶ Connected components by label propagation on snapshot G_1 produces R_1. ❷ Vertex A and edge $A-B$ is deleted in G_2. Using the last result to bootstrap computation results in incorrect answer R_2. ❸ A strawman approach of storing all messages during the initial execution and replaying it produces correct results, but needs to store large amounts of state.

retrieval of the state of graph entities in any snapshot. Using this, graph-parallel phases can operate on the evolution of an entity (vertex or edge) as opposed to a single (at a given snapshot) value. In simple terms, each processing phase is able to see the history of the node's property changes. This allows *temporal* queries (§2.2) involving multiple snapshots, such as the degree computation, to be expressed as:

```
results = G.degrees(Array(G1,G2,G3))
```

where `degrees` implementation takes advantage of timelapse by combining the phases in graph-parallel computation for these snapshots. That is, the user-defined vertex program is provided with state in all the snapshots. Thus, we are able to eliminate redundant messages and computation.

4 Computation Model

To improve interactivity, TEGRA must be able to efficiently execute queries by effectively reusing previous query results to reduce or eliminate redundant computations, commonly referred to as performing *incremental computation*. Here, we describe TEGRA's incremental computation model.

4.1 Incremental Graph Computations

Supporting incremental computation requires the system to manage *state*. The simplest form of state is the previous computation result. However, many graph algorithms are iterative in nature, where the graph-parallel stages are repeatedly applied in sequence until a fixed point. Here, simply restarting the computations from previous results do not lead to correct answers. To illustrate this, consider a connected components algorithm using label propagation on a graph snapshot, G_1 as shown in ❶ in fig. 3 which entails result R_1 after three iterations. When the query is to be repeated on G_2, restarting the computation from R_1 as shown in ❷ computes incorrect result. In general, correctness in such techniques depend on the properties of the algorithm (e.g., abelian group) and the monotonicity of updates (e.g., the graph only grows). Hence, supporting general non-monotonic iterative computations requires maintaining *intermediate* state.

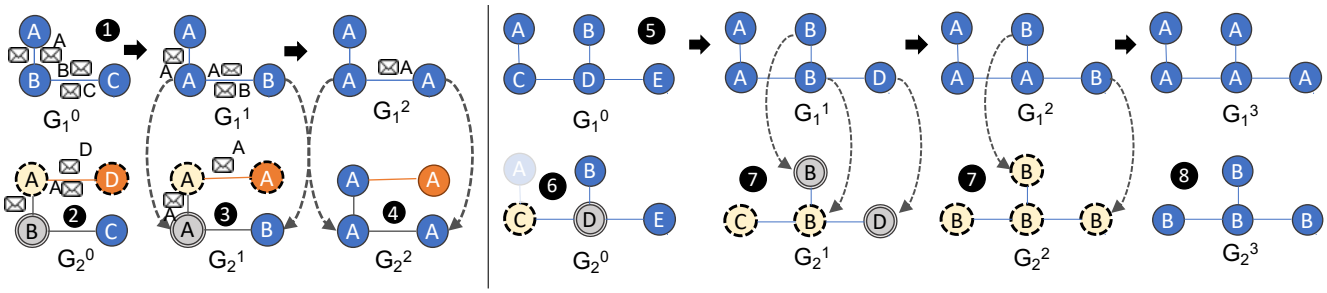

Figure 4: Two examples that depict how ICE works. Dotted circles indicate vertices that recompute, and double circles indicate vertices that need to be present in the subgraph to compute the correct answer, but do not recompute state themselves. ❶❺ Iterations of initial execution is stored in the timelapse. ❷❻ ICE bootstraps computation on a new snapshot, by finding the subgraph consisting of affected vertices and their dependencies (neighbors). In ❻, C is affected by the deletion of $A-C$. To recompute state it needs D (yields subgraph $C-D$). ❸❼ At every iteration, after execution of the computation on the subgraph, ICE copies state for entities that did not recompute. Then finds the new subgraph to compute by comparing the previous subgraph to the timelapse snapshot. In ❼, though C recomputes the same value as in previous iteration, its state is different from the snapshot in timelapse and hence needs to be propagated. ❹❽ ICE terminates when the subgraph converges and no entity in the graph needs the state copied from stored snapshots in the timelapse.

TEGRA proposes a general, incremental iterative graph-parallel computation model that is *algorithm independent*. It leverages the fact that graph-parallel computations proceed by making iterative changes to the original graph. Thus, *iterations of a graph-parallel computation can be seen as a time-evolving graph*, where the snapshots are the materialized state of the graph at the end of each iteration. In each of the snapshots, the intermediate state can be stored as vertex and edge properties. Since timelapse can efficiently store and retrieve these snapshots, we can perform incremental computations in a generic fashion by invoking graph-parallel computations on *affected neighborhood*. We call this model *Incremental Computation by entity Expansion* (ICE).

4.2 ICE Computation Model

ICE executes computations only on the subgraph that is affected by the updates *at each iteration*. To do so, it needs to find the relevant entities that should participate in computation at any given iteration. For this, it uses the state stored as timelapse, and the computation proceeds in four phases:

Initial execution: When an algorithm is executed for the first time, ICE stores the state (using the `save` API) of the vertices (and edges if the algorithm demands it) as properties in the graph. At the end of every iteration, a snapshot of the graph is added to the timelapse. The ID is generated using a combination of the graph's unique ID, an algorithm identifier and the iteration number. As depicted in ❶ and ❺ in fig. 4, the timelapse contains three and four snapshots, respectively.

Bootstrap: When the computation is to be executed on a new snapshot, ICE needs to bootstrap the incremental computation. Intuitively, the subgraph that must participate in the computation at bootstrap consists of the updates to the graph, and the *entities affected by the updates*. For instance, any newly added or changed vertices should be included. Similarly, edge modifications would result in the source and/or destination vertices to be included in the computation. However, the affected entities alone are not sufficient to ensure correctness of the results. This is because in graph-parallel execution, the state of a graph entity is dependent on the collective input from its neighbors. Thus, ICE must also include the one-hop neighbors of *affected entities*, and so the bootstrapped subgraph consists of the affected entities and their one-hop neighbors. ICE uses the `expand` API for this purpose. The graph computation is run on this subgraph. ❷ in fig. 4 shows how ICE bootstraps when a new vertex D and a new edge $A-D$ is added. D and A should recompute state, but for A to compute the correct state, it must involve its one-hop neighbor B, yielding subgraph $D-A-B$.

Iterations: At each iteration, ICE needs to find the right subgraph to perform computations. ICE exploits the fact that the nature of the graph-parallel abstraction restricts the propagation distance of updates in an iteration. Intuitively, the graph entities that might possibly have a different state at any iteration will be contained in the subgraph that ICE has already executed computation on from the last iteration. Thus, after the initial bootstrap, ICE can find the new subgraph at a given iteration by examining the changes to the subgraph from the previous iteration (using `diff`) and expanding to the one-hop neighborhood of affected entities (using `expand`). For the vertices/edges that did not recompute the state, ICE simply copies the state from the timelapse (using `merge`). For instance, in ❸ in fig. 4, though A and D recomputed, only D changed state and needs to be propagated to its neighbor A which needs B.

Termination: It is possible that modifications to the graph may result in more (or less) number of iterations compared to the initial execution. Unlike normal graph-parallel computations, ICE does not necessarily stop when the subgraph converges. If there are more iterations stored in the timelapse for the initial execution, ICE needs to check if the unchanged

parts of the graph must be copied over. Conversely, if the subgraph has not converged and there are no more corresponding iterations, ICE needs to continue. To do so, it simply switches to normal (non-incremental) computation from that point. Thus, ICE converges only when the subgraph converges and no entity needs their state to be copied from the stored snapshot in the timelapse. (❹ and ❽ in fig. 4)

By construction, ICE generates the exact same intermediate states for all edges and vertices at all iterations, as compared to running a full execution on the entire graph. Thus, not only does ICE guarantee correctness of the incremental execution, but also enables *any* algorithm implemented in a graph-parallel fashion to be made incremental.

4.3 Improving ICE Model

Sharing State Across Different Queries Many graph algorithms consist of several stages of computations, some of which are common across different algorithms. For example, variants of connected components and pagerank algorithms both require the computation of vertex degree as one of the steps. Since ICE decouples state, such common computations can be stored as separate state that is shared across different queries. Thus, ICE enables developers to generate and *compose* modular states. This reduces the need to duplicate common state across queries which results in reduced memory consumption and better performance.

Incremental Computations Can Be Inefficient Incremental computation is not useful in all cases. For instance, in graphs with high degree vertices, a small change may result in a domino effect in terms of computation—that is, during later iterations, a large number of graph entities might need to participate in computation (e.g., Example 2 in fig. 4). To perform incremental computation, ICE needs to spend computations cycles to identify the set of vertices that should recompute (using `diff`) and copy the state of vertices that did not do computations (using `merge`). As a result, the total work done by the system may exceed that of completely executing the computation from scratch [24, 67]. Fortunately, the design of ICE lets us overcome this inefficiency. Since ICE generates the same intermediate states at every iteration as full re-execution, it can switch to full re-execution at any point.

A Simple Learning Based Model for Switching A key requirement for avoiding the inefficiencies with incremental execution detailed previously is to determine *when* to switch to full re-execution. This can be done at two places: at the start of the incremental execution, or at iteration boundaries (i.e., at the beginning of an iteration during the execution). In TEGRA, we picked the latter. A strawman approach is to use a simple threshold—for instance, *active* vertices in an iteration—to determine when to switch. Unfortunately, such approaches did not perform well in our evaluation, as we found that the optimal point for the switch depends on a number of factors, including the query, the properties of the graph and the nature of the modifications. Thus, we use a simple learning based approach to determining when TEGRA makes the switch.

In our approach, we train a simple random forest classifier [14] to predict, at the beginning of an iteration, if switching to full re-execution from that point would be faster compared to continuing with incremental execution. We do the training in an offline phase, where we use several runs of queries both in a full incremental and full re-execution mode as the input, ensuring enough runs in both cases to avoid class sensitivity. For each run, in every iteration, we record the following fields that we use as *features* for the learning: number of vertices that participate in the computation, the average degree of the active vertices, the number of partitions active, the number of messages generated per vertex, the number of messages received per vertex, the amount of data transferred over the network and the time taken for the iteration to complete. To make the learning general, we also use a few graph-specific characteristics such as the average degree of vertices, the average diameter and clustering coefficient. The label indicates whether switching to full recomputation in the next iteration resulted in faster execution.

While simple, we found that this approach works well as we show in fig. 11. Examination of the model revealed that it tries to learn the significance of vertices that participate in the computation (in terms of average degrees), the layout (how they are partitioned) and graph characteristics (in terms of diameter and clustering coefficient) in relation to the execution time. We plan to explore ways to improve our technique (e.g., better/robust models) as part of our future work.

5 Distributed Graph Snapshot Index (DGSI)

To make timelapse abstraction and ICE computation model practical, TEGRA needs to back them with a storage that satisfies the following three requirements: (1) enable ingestion of updates in real-time, and make it available for analysis in the minimum time possible, (2) support space-efficient storage of snapshots and intermediate computation state in a timelapse, and (3) enable fast retrieval and efficient operations on stored timelapses. These requirements, crucial for efficiently supporting ad-hoc analytics on time-evolving graphs, pose several challenges. For instance, they prohibit the use of preprocessing, typically employed by many graph processing systems, to compactly represent graphs and to make computations efficient. In this section, we describe how TEGRA achieves this by building DGSI. It addresses (1) and (2) by leveraging persistent data structures to build a graph store (§5.1, §5.2) that enables efficient operations (§5.3) while managing memory over time (§5.4).

5.1 Leveraging Persistent Data Structures

In TEGRA, we leverage persistent data structures [22] to build a distributed, versioned graph state store. The key idea in persistent data structures is to maintain the previous versions of data when modified, thus allowing access to earlier ver-

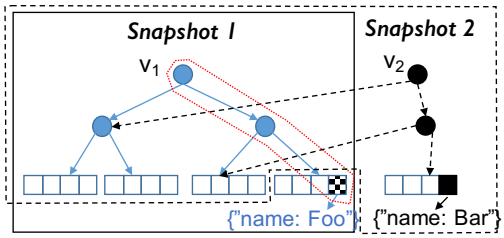

Figure 5: DGSI has one pART for vertices and one for edges in each partition. Version 1 (v_1) of a vertex pART stores properties of vertices in its leaves. Vertex id traverses the tree to its properties (e.g., *name*). Changes to vertices (e.g., property *name* changed from *Foo* to *Bar*) generates a new version v_2. The snapshot of the vertex pART before (Snapshot 1, shown in solid box) and after (Snapshot 2, shown in dotted box) the change share common leaves. (§5.2)

sions. DGSI uses a persistent version of the Adaptive Radix Tree [38] as its data structure. ART provides several properties useful for graph storage such as efficient updates and range scans. Persistent Adaptive Radix Tree (PART) [5] adds persistence to ART by simple path-copying. For the purpose of building DGSI, we reimplemented PART (hereafter pART) in Scala and made several modifications to optimize it for graph state storage. We also heavily engineered our implementation to avoid performance issues, such as providing fast iterators, avoiding unnecessary small object creation and optimizing path copying under heavy writes.

5.2 Graph Storage & Partitioning

TEGRA stores graphs using two pART data structures: a *vertex* tree and an *edge* tree. The vertices are identified by a 64-bit integer key. For edges, we allow arbitrary keys stored as byte arrays. By default, the edge keys are generated from their source and destination vertices and an additional short field for supporting multiple edges between vertex pairs. pART supports prefix matching, so using matching on this key enables retreiving all the destination edges of a given vertex. The leaves in the tree store pointers to arbitrary properties.We create specialized versions of pART to avoid (un)boxing costs when properties are primitive types.

TEGRA supports several graph partitioning schemes, similar to GraphX [26], to balance load and reduce communication. To distribute the graph across machines in the cluster, vertices are hash partitioned and edges are partitioned using one of many schemes (e.g., 2D partitioning). We do not partition the pART structures, instead TEGRA partitions the graph and creates *separate* pART structures locally in each partition. Hence logically, in each partition, the vertex and edge trees store a subgraph (fig. 5). By using local trees, we further amortize the (already low) cost[4] associated with modifying the tree upon graph updates.

To consume updates, TEGRA needs to send the updates to the right partition. Here, we impose the *same* partitioning as the original graph on the vertices/edges in the update.

[4]Modifications to nodes in ART trees only affect the $O(\log_{256} n)$ ancestors

5.3 Version Management

DGSI is a versioned graph state store. Every "version" corresponds to a root in the vertex and edge tree in the partitions—traversing the trees from the root pair materializes the graph snapshot. For version management, DGSI stores a mapping between a root and the corresponding "version id" in every partition. The version id is simply a byte array.

For operating on versions, DGSI exposes two low level primitives inspired by existing version management systems: `branch` and `commit`. A `branch` operation creates a new working version of the graph by creating a new (transient) root that points to the original root's children. Users operate on this newly created graph without worrying about conflicts because the root is exclusive to them and not visible in the system. Upon completing operations, a `commit` finalizes the version by adding the new root to version management and makes the new version available for other users in the system. Once a `commit` is done on a version, modifications to it can only be done by "branching" that version. Any timelapse based modifications cause `branch` to be called, and the timelapse `save` API invokes `commit`.

TEGRA can interface with external graph stores, such as Neo4J [4] or Titan [6] for importing and exporting graphs. While importing new graphs, DGSI automatically assigns an integer id (if not provided) and commits the version when the loading is complete. We create a version by batching updates. The batch size is user-defined. In order to be able to retrieve the state of the graph in between snapshots, TEGRA stores the updates between snapshots in a simple log file, and adds a pointer to this file to the root.

The simplest retrieval is by using its id. In every partition, DGSI then gets a handle to the root element mapped to this id, thus enabling operations on the version (e.g., branching, materialization). By design, versions in DGSI have no global ordering because branches can be created from any version at any time. However, in some operations, it may be desirable to have ordered access to versions, such as in incremental computations where the system needs access to the consecutive iterations. For this purpose, we enable suffix, prefix and simple ranges matching primitives on version id.

5.3.1 Implementing Timelapses

TEGRA implements timelapses using DGSI with its version ids and the matching primitives it provides. Recall that each timelapse logically represents a sequence of graph snapshots. Hence, every snapshot stored in DGSI is part of one or more timelapses. As a simple example, a user intending to track the Twitter graph by time could create snapshots by appending UNIX epoch to a unique ID for the graph (e.g., `TWTR`). In this scheme, a snapshot created at 9:00AM on 01/01/2020 may be given ID `TWTR_1577869200`. Prefix matching `TWTR` provides the entire timelapse for this graph. When this snapshot is chosen for a query, say PageRank, TEGRA can automatically append an algorithm ID (e.g., `PR`) and an iteration number to gener-

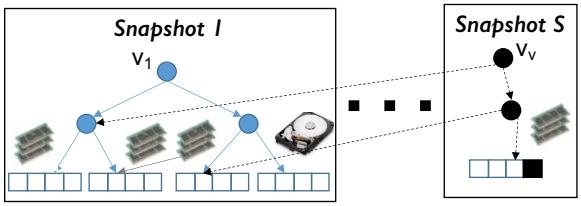

Figure 6: DGSI has fine-grained control over leaves (where data is stored). Here DGSI has 1000s of snapshots. All snapshots except *S* are on disk, their parents just hold pointers to files on disk. Parents are also dynamically written to disk if all of their children are on disk. Data structure uses adaptive leaf sizes for efficiency.

ate version IDs such as `TWTR_1577869200_PR_1`. Here, prefix matching on `TWTR_1577869200_PR` provides the timelapse of the execution of the page rank algorithm.

Currently, TEGRA only manages automatic ID generation for time-based snapshots and iterations of query execution. For instance, creating snapshots every hour is automated by doing a **`branch`** on the last snapshot in the `TWTR` timelapse and **`commit`**ing the new changes with new timestamp appended (e.g., `TWTR_1577872800`). Similarly, saving iterations of a query is as simple as keeping track and **`branch`**ing the last snapshot in the corresponding timelapse, and saving the new iteration with incremented iteration number (e.g., `TWTR_1577869200_PR_2`). However, since TEGRA stores IDs as byte arrays, users are free to choose any ID generation schemes; for instance it is possible to come up with complex hierarchical IDs that enable sophisticated retrievals using the matching capabilities on IDs provided by DGSI.

5.4 Memory Management

Over time, DGSI stores several versions of a graph, and hence TEGRA needs to manage these versions efficiently. We employ several ways to do this. Between **`branch`** and **`commit`** operations, it is likely that many transient child nodes are formed. We aggressively remove them during the **`commit`** operation. In addition, we enable in-place updates when the operations are local, such as after a branch and before a commit. Further, during ad-hoc analysis, analysts are likely to create versions that are never committed. We periodically mark such orphans and adjust the reference counting in our trees to make sure that they are garbage collected.

For managing stored versions, we leverage a simple Least Recently Used (LRU) eviction policy. Each time a version is accessed, we annotate the version and all its children with a timestamp. The system then employs a thread for periodically removing versions that were not accessed in a long time. The eviction is done by saving the version to local disk (or distributed file system). We do this in the following way. Since every version in DGSI is a branch, we write each subtree in that branch to a separate file and then point its root to the file identifier (e.g., in fig. 5, we can store v_2's leaf that is different from v_1 in disk as a file and point the parent node to this file). By writing subtrees to separate files, we ensure that different versions sharing tree nodes in memory can share tree nodes written to files. Due to this technique, we can ensure that leaf nodes (which are most memory consuming) that are specific to a version (not shared with any other version) are always written to disk if the version is evicted. As depicted in fig. 6, a large number of versions can be flushed to disk over time while still being retrievable when necessary. Thus, only active snapshots are fully materialized in memory, thereby allowing TEGRA to store several snapshots.

6 Implementation

TEGRA is a drop-in replacement for GraphX [26]. It uses the popular Gather-Apply-Scatter (GAS) [27] graph parallel model. We utilize the barrier execution mode to implement direct communication between tasks to avoid most Spark overheads. Spark provides fault tolerance by checkpointing inputs and operations for reconstructing the state. TEGRA provides coarse-grained fault tolerance by leveraging Spark's `rdd.checkpoint` semantics. Users can explicitly run checkpoint operation, upon which TEGRA flushes the contents in DGSI to persistent storage. We currently do not support fine-grained lineage-based fault tolerance provided by Spark.

6.1 ICE on GAS Model

As described in §4.2, the **`diff`**`()` API marks the candidates that must perform graph-parallel computation in a given iteration. In GAS decomposition [27], the `scatter()` function, invoked on `scatter_nbrs`, determines the set of active vertices which must perform computation. Starting with an initial candidate set (e.g., at bootstrap the changes to the graph, and at any iteration the candidates from the previous iteration) the **`diff`**`()` API uses `scatter_nbrs` (`EdgeDirection` in GraphX) in the user-defined vertex program to mark all necessary vertices for computation. We mark all `scatter_nbrs` of a vertex if its state differs from the previous iteration, or from the previous execution stored in the timelapse. For instance, a vertex addition must inspect all its neighbors (as defined by `scatter_nbrs`) and include them for computation.

The vertices in GAS parallel model perform computation using the user defined `gather()`, `sum()` and `apply()` functions, where `gather_nbrs` determine the set of neighbors to gather state from. The **`expand`** API enables correct `gather()` operations on the candidates marked for recomputation by also marking the `gather_nbrs` of the candidates. After the **`diff`** and **`expand`**, TEGRA has the complete subgraph on which the graph-parallel computation can be performed.

6.2 Using TEGRA as a Developer

TEGRA provides feature compatibility with GraphX, and expands the existing APIs in GraphX to provide ad-hoc analysis support on evolving graphs. It extends all the operators to operate on user-specified snapshot(s) (e.g., `Graph.vertices(id)` retrieves vertices at a given snapshot id, and `Graph.mapV([ids])` can apply a map function on vertices of

```
def IncPregel(g: Graph[V, E],
     prevResult: Graph[V, E],
     vprog: (Id, V, M) => V,
     sendMsg: (Triplet) => M,
     gather: (M, M) => M): Graph[V, E] = {
 iter = 0
 // Loop until no active vertices and nothing to copy
 // from previous results in timelapse.
 while (!converged) {
   // Restrict to vertices that should recompute
   val msgs: Collection[(Id, M)] =
     g.expand(g.diff(prevResult.retrieve(iter))).
      .aggregateMessages(sendMsg, gather)
   iter += 1
   // Receive messages and copy previous results
   g = g.leftJoinV(msgs).mapV(vprog)
        .merge(prevResult.retrieve(iter)).save(iter) }
 return g }
```

Listing 1: Implementation of incremental Pregel using TEGRA APIs.

the graph on a set of snapshots). Graph-parallel computation is enabled in GraphX using the `Graph.aggregateMessages()` (previously `mrTriplets()`) API. To use TEGRA, users incorporate the TEGRA API in table 1 in their normal, static (non-incremental) versions of the algorithm at places where graph's state is mutated. These are places where GraphX's `Graph.aggregateMessages()` is used.

GraphX further offers iterative graph-parallel computation support through a Pregel API which captures the GAS decomposition using repeated invocation of the `aggregateMessages` and `joinVertices` until a fixed point. Listing 1 shows how a user might use TEGRA APIs to implement an incremental version of Pregel. The code is reproduced from GraphX [26], with minimal changes to incorporate TEGRA APIs to store and retrieve state. In general, a developer can write incremental versions of any iterative graph parallel algorithm by using the TEGRA APIs along with `aggregateMessages`.

7 Evaluation

We have evaluated TEGRA through a series of experiments.

Comparisons: We compare TEGRA against many state-of-the-art systems (§2.3). For streaming system, we use GraphBolt [45] and the Rust implementation of Differential Dataflow (DD) [3]. Since we were unable to obtain an open source implementation of a temporal engine, we developed our version of Chronos [30] in GraphX [26], which we call Chlonos (Clone of Chronos) in this section. This implementation emulates the array based in-memory layout of snapshots and the incremental computation model in Chronos. We note that while Chronos supports updates to graphs by storing the temporal changes on disk, it uses a pre-processing step to create an in-memory layout which is used for every query. This in-memory layout does not support updates (§2.3) and needs to be recreated every time. We compare DGSI against GraphOne [35] and Aspen [21].

Dataset	Vertices / Edges
twitter [11]	41.6 M / 1.47 B
uk-2007 [12]	105.9 M / 3.74 B
Facebook Synthetic Data [2]	Varies / 5, 10, 50 B

Table 2: Datasets in our evaluation. M = Millions, B = Billions.

Evaluation Setup: All of our experiments were conducted on 16 commodity machines available as Amazon EC2 instances, each with 8 virtual CPU cores, 61GB memory, and 160GB SSDs. The cluster runs a recent 64-bit version of Linux. We use Differential Dataflow v0.10.0 and Apache Spark v2.4.4. We warm up the JVM before measurements. For single machine systems, we use a x1.32xlarge instance with 128 virtual CPUs and 2 TB memory to be comparable with our cluster.

Dataset & Workloads: We evaluate TEGRA on a number of real-world graphs depicted in table 2, with up to 50 billion edges. TEGRA creates default properties at vertices and edges to allow queries that compute on them (our comparisons do not support *arbitrary* properties). We use three standard, well understood, iterative graph algorithms with varying computation and state requirements, commonly used to evaluate graph processing systems as queries: Connected Components (CC), Page Rank (PR) [55] and Belief Propagation (BP) [74]. We run PR until a specific convergence or 20 iterations, whichever is lower. Note that while the queries in this section do not access the vertex and edge properties explicitly (i.e., queries do not ask for them), TEGRA depends on them extensively to store intermediate state (§4).

Caveats. While perusing the evaluation results, we wish to remind the reader a few caveats. Though many of the graphs we use fit in the memory of a modern machine, TEGRA is focused on ad-hoc analytics which requires storage of multiple snapshots (and computation state) of the graph. Further, ad-hoc analytics requires the use of property graphs. TEGRA supports edge and vertex properties and creates a default value, which increases the graph size by several magnitudes and also affects performance. Finally, DD's connected component uses the union-find based implementation (hard to fit in a vertex centric model) which is superior to TEGRA's label propagation based implementation.

7.1 Microbenchmarks

We first present experiments that highlight the effectiveness of DGSI. GraphBolt is excluded as it doesn't allow storing multiple versions or intermediate state. (§2.3).

Snapshot Retrieval Latency: We generate 1000 snapshots of the Twitter and UK graphs by randomly modifying (adding and removing equal number) 1% of the edges (no computations are performed) to emulate the evolution of the graph. Table 3 shows the average latency for 10 random retrievals with varying number of snapshots in the system.

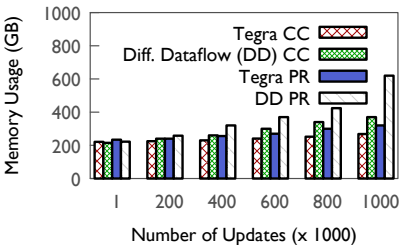

Figure 7: Differential dataflow generates state at every operator, while TEGRA's state is proportional to the number of vertices.

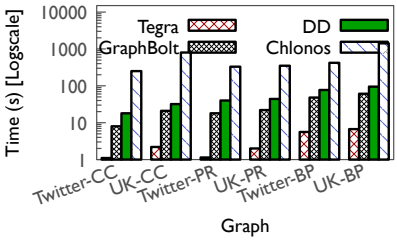

Figure 8: On ad-hoc queries on snapshots, TEGRA is able to significantly outperform due to state reuse.

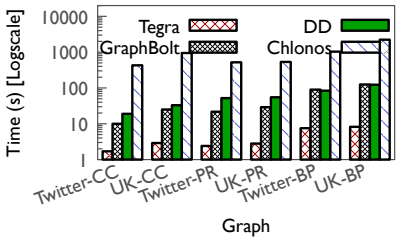

Figure 9: TEGRA's performance is superior on ad-hoc window operations even with materialization of results.

Graph	System	# Snapshots in memory				
		200	400	600	800	1000
Twitter	DD	30.2	44.8	92.4	131.6	180.2
	GraphOne	32.3	50.1	105.5	141.2	201.2
	Aspen	0.43	0.54	0.53	0.57	0.65
	TEGRA	1.34	1.5	1.25	1.78	2.1
UK	DD	63.1	86.4	220	271.5	283.2
	GraphOne	74.3	102.2	263.5	321.4	332.5
	Aspen	0.86	1.04	1.12	1.25	1.31
	TEGRA	2.2	2.5	2.6	2.25	2.3

Table 3: Snapshot retrieval latency (in seconds). DD and GraphOne require reconstruction, while TEGRA and Aspen can simply traverse the data structure from a version's root.

We see that TEGRA is able to return the queried snapshot within seconds due to DGSI which stores and retrieves materialized snapshots efficiently. In contrast, DD needs to reconstruct the graph from indexed differences and takes several minutes. It also exhibits high variance in retrieval time based on the amount of reconstruction required. GraphOne faces similar challenge with its `get-prior-edges()` API which needs to reconstruct the historic view from the durable edge log. While not shown, Chlonos too exhibits similar characteristics as DD but is significantly slower, in some cases up to an order of magnitude. This is because it stores updates on disk and needs an intensive preprocessing step to create the in-memory layout. DD exhausts the memory available in our cluster (≈1TB) in this experiment which limited the number of snapshots we could store to ≈1000. One solution is to store the updates in a persistent storage, but this incurs significant performance degradation while retrieving (like Chlonos). Aspen performs similar to TEGRA (slightly faster since it is able to compress the graph significantly better) due to its use of persistent data structures, hence it only needs to traverse the tree from a specific root to retrieve a version (like DGSI). However, it neither supports intermediate state storage nor includes an incremental computation model.(§2.3)

Computation State Storage Overhead: To measure the memory overhead due to computation state, we perform PR and CC computation on the Twitter graph in an incremental fashion, where we add and delete 1000 edges to create a snapshot. We note the memory usage by each system after every 200 such computations until 1000 computations (for a total of 1 million edge updates). Figure 7 shows this experiment's results. When the number of updates is small, both TEGRA and DD use comparable amount of memory to store the state, even with DD's highly compact layout (native arrays compared to TEGRA's property graph). However, DD's state size increases rapidly as it does more computation and takes up to 2× that of TEGRA. TEGRA's memory requirement also increases over time, but much more gracefully. This is due to the combined effect of TEGRA's compact state representation (proportional to the number of vertices) and the ability of DGSI to manage memory efficiently (§5.4), while DD needs to keep state (proportional to the number of edges) at every operator. The amount of increase also depends on the algorithm. For instance, page rank generates the same amount of state in every iteration while connected component's state requirement reduces over iterations. Note that DD uses compaction in this experiment which is automatically done by the system. While GraphBolt also reduces the state requirement to be proportional to the number of vertices, it does not allow storing computational state for later reuse.

7.2 Ad-hoc Window Operations

Here, we present evaluations that focus on TEGRA's main goal. In these experiments, we emulate an analyst performing ad-hoc analytics. We load the graph, and apply a large number of sequential updates to it, where each update modifies 0.1% of the edges (adds and removes equal number) to adhere to our assumption that during ad-hoc analysis the graph doesn't change much and it is possible to leverage incremental computation (we show results with large changes in fig. 11). We then retrieve 100 random windows of the graph that are close-by, and apply queries in each. We assume that *some* earlier results are available so that the system could do incremental computations. We do not consider the window retrieval time in this experiment for any system. We present the average time taken to compute the query result.

Single Snapshot Operations: In the first experiment, we set the window size to zero so that every window retrieval returns a single snapshot. The results are depicted in fig. 8. DD,

Graph	5B			10B			50B		
	PR	CC	BP	PR	CC	BP	PR	CC	BP
DD	1m	8s	1.5m	2m	34s	-	-	-	-
GraphBolt	29s	21s	1.1m	1.2m	28s	2.2m	5.3m	54s	12m
TEGRA	10s	5s	6.5s	19s	7s	9.3s	1.5m	18s	2.4m

Table 4: Ad-hoc analytics on big graphs, with 5 billion, 10 billion and 50 billion edges. A '-' indicates the system failed to run the workload. TEGRA can handle big graphs and large amounts of state due to its efficient memory management (§5.4)

GraphBolt and Chlonos do not allow reusing computation across queries, so they compute from scratch for every retrieval. In contrast, TEGRA is able to leverage the compact computation state stored in its DGSI from earlier queries to do incremental computation. In this case, most of the snapshots incur no computation overhead because of the small amount of changes between them, and TEGRA is able to produce an answer within a few seconds. DD and GraphBolt take a few 10s of seconds, while Chlonos requires 100s of seconds. TEGRA's benefits range from 18-30× compared to DD and 8-18× compared to GraphBolt.

Window Operations: Here we set the window size to be 10 snapshots. GraphBolt, Chlonos and DD are able to apply incremental computations once the query has been computed on the first snapshot. Figure 9 shows the results. We see that DD is fast once the first result has been computed. This is due to the combination of its extremely efficient streaming computation model (no materialization) and recent optimizations such as shared arrangements [48]. Chlonos incurs a penalty initially because it uses the first result to bootstrap the rest using its LABS model. TEGRA's performance remains consistent. This is due to two reasons. First, since TEGRA separates state from computation, it can reuse the state across multiple snapshots. Second, due to the use of persistent data structures, snapshot can be independently and concurrently operated on (§3.1). Since GraphBolt does not support concurrent processing, it does sequential computation (in an incremental fashion) on the snapshots. For simple queries (e.g., CC), the penalty is unnoticeable. It becomes pronounced in BP which is more computationally heavy. TEGRA is still 9-17× (5-23×) faster compared to DD (GraphBolt).

Large Graphs & Large Amounts of State: Here we answer two questions: (1) can TEGRA support ad-hoc analysis on large graphs, and (2) can TEGRA efficiently manage memory when large amounts of state need to be stored? We use synthetic graphs provided by Facebook [2] modeled using the social network's properties. We execute the queries once on the original graph, then modify the graph by a tiny percentage (0.01%) randomly 1000 times to create 1000 snapshots. We then pick a snapshot, run the queries on it and provide the average of 100 such runs in table 4. DD works well when both the graph and the updates (and the generated state) are small. However, as the graph becomes larger, DD needs to push a large number of updates through the computation, and state becomes a bottleneck in its performance. On the largest graph, we were unable to get DD to work as it failed due to excessive memory usage during initial execution. GraphBolt doesn't store any previous state, and hence is unable to do incremental computations. It also required several optimizations to support the largest graph. In contrast, TEGRA is not only able to efficiently use memory and disk (§5.4) and scale to large graphs and snapshots, but also provide significant benefits by using previous computation state.

		Twitter			UK		
		1K	10K	100K	1K	10K	100K
CF	GraphBolt	15.1	15.3	15.0	21.5	21.8	21.7
	TEGRA	1.2	1.3	1.5	1.4	1.4	1.6
CoEM	GraphBolt	17.1	17.6	17.8	28.1	28.6	28.9
	TEGRA	1.7	1.8	2.1	1.8	1.8	2.3
LP	GraphBolt	22.1	22.4	22.6	30.1	30.2	32.4
	TEGRA	1.8	1.9	1.9	2.0	2.2	2.3
TC	GraphBolt	68.1	68.5	69.2	5.2	5.4	5.7
	TEGRA	0.10	0.12	0.14	0.11	0.15	0.25
BFS	GraphBolt	0.6	0.7	0.8	0.8	0.8	0.9
	TEGRA	0.15	0.16	0.16	0.21	0.21	0.25
k-hop (4-hop)	GraphBolt	1.1	1.1	1.2	1.2	1.2	1.3
	TEGRA	0.6	0.6	0.55	0.7	0.7	0.8

Table 5: Running time (in seconds) for TEGRA and GraphBolt when doing ad-hoc analysis with different batch sizes and algorithms.

Effect of Batch Size & Additional Algorithms: In this experiment, we evaluate the effect of batch size on the ad-hoc analysis capability of TEGRA. For this, we fix the batch size to a specific number in each run, and use several other algorithms. Specifically, we use Label Propagation (LP), Collaborative Filtering (CF) and Triangle Count (TC) and Co-Training Expectation Maximization (CoEM) as used in GraphBolt [45]. For CoEM, we use the Latent Dirichlet Allocation (LDA) implementation in GraphX which uses EM. We also provide results on k-hop, which computes the set of vertices that are k hops away, and Breadth First Search (BFS). For the k-hop algorithm, we set k to 4 for all batch sizes.

In each run, we execute the algorithm first. We then generate several snapshots using varying batches of equal edge additions and deletions. We choose three fixed numbers: 1K, 10K and 100K. We pick a random snapshot and repeat the same algorithm on it. The results are shown in table 5.TEGRA is able to perform incremental computation using previous results, while GraphBolt does not support ad-hoc analysis and hence need to execute the algorithm fully. We also notice that varying batch size doesn't affect TEGRA much, and that it is able to provide results efficiently.

We note a few caveats here. In TC, the incremental computations are simple (edge additions and deletions do not result in multiple iterations) and involves just updating a count based on the edges added or deleted. Similarly, BFS and 4-hop algorithms are light weight and result in only a very small part of the graph to be active, especially during incremental computation. Due to this reason, for these algorithms, we only measure the actual computation time and ignore the scheduling overhead in TEGRA. Hence, the times we report for these

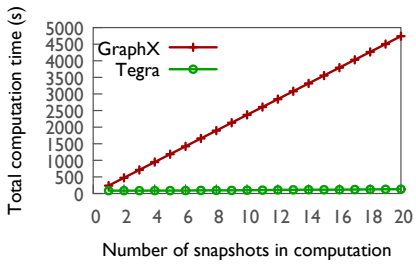

Figure 10: Timelapse lets queries to be executed simultaneously on a sequence of snapshots, enabling efficient temporal analysis.

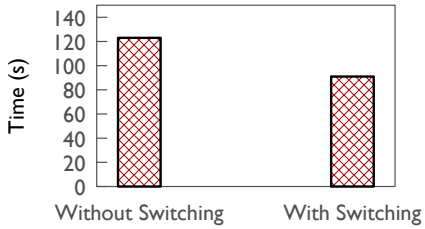

Figure 11: TEGRA can switch to full re-execution when incremental computations are not useful.

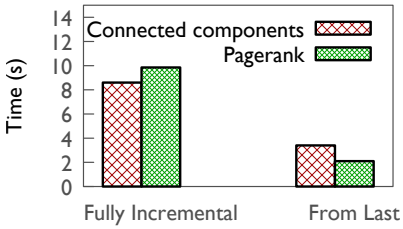

Figure 12: Monotonicity of updates (additions only) can be leveraged to speed up computations by starting from the last answer.

three algorithms are not end-to-end, but just the time it takes for the computation task to complete.

7.3 Timelapse & ICE

Parallel computations. Here we evaluate the ability of Timelapse to do temporal queries, where a query is applied to a sequence of snapshots in parallel (§3.1). We create 20 snapshots of the Twitter graph by starting with 80% of the edges and adding 1% to it repeatedly. We run the connected components algorithm, each time varying the number of snapshots on which the computation is run. In each run, we measure the time taken to obtain the results on all the snapshots considered. For comparison, we use GraphX and apply the algorithm to each snapshot in a serial fashion. The results are depicted in fig. 10. We see that TEGRA significantly outperforms GraphX. The improvement for a single snapshot is due to TEGRA's optimizations and use of barrier execution mode in Spark. Further, we see a linear trend with increasing number of snapshots. By sharing computation and communication, TEGRA is able to achieve up to 36× speedup.

ICE's switching capability: To test ICE's switching capability when incremental computations are not useful (§4.3), we run the CC algorithm on the Twitter graph. Next, we introduce a batch of deletions in the largest components so that incremental computation executes on a large portion of the graph. We then make TEGRA recompute with and without the switching enabled and average the results over 10 such runs. The results are shown in fig. 11. We see that without the switching, TEGRA incurs a penalty—the incremental execution takes more time than complete re-execution of the algorithm. With switching, TEGRA is easily able to identify that it needs to switch and hence does not incur this penalty.

ICE's versatility: Since ICE differs from streaming engines, it can also provide flexibility in how it uses state. For instance, if updates are monotonic (only additions), then ICE can simply restart from the last answer instead of using full incremental computations. Figure 12 shows this on two algorithms on the UK graph. PR and CC can benefit, but PR is faster since it only needs to converge within a given tolerance.

Sharing state across queries: To evaluate how much benefits sharing state between different queries provides, we run an experiment with CC and PR. For these queries, the degree computation can be shared. We run the algorithms with and without sharing enabled on the Twitter graph, and average the results of 10 runs of incremental computations on random snapshots. The results in fig. 13 show 20% and 30% reduction in memory usage and runtime.

7.4 TEGRA Shortcomings

Finally, we ask "What does TEGRA **not** do well?".

Purely Streaming Analysis: We consider an online query (§2) of CC. To emulate a streaming graph, we first perform CC computation on the graph. Then we continuously change 0.01% of the graph by adding and deleting equal number of edges. After fixed number of changes (every 200), we show the average runtime of 10 runs in fig. 14. We see that DD and GraphBolt are significantly better than TEGRA for such workloads. This is due to a combination of DD and GraphBolt optimized for online queries (pushing small updates really fast through computation) and their Rust/C++ implementation. We remind the reader of two caveats here. First, DD uses a much superior union-find approach to CC while TEGRA and GraphBolt use an iterative approach. Second, TEGRA only executes queries when it is asked to, whereas DD and GraphBolt executes queries for every batch of updates (thus TEGRA accumulates more updates when executing queries). While TEGRA can theoretically process each small update separately, the computation engine it builds on (Spark) is tuned for batched updates.

Purely Temporal Analysis: We assume that the queries and the window are known, and the system has optimized the data layout. We run a query on a window size of 10 and compare TEGRA and Chlonos on the incremental processing time (we discard the time for full execution). Excluding processing time, fig. 15 shows that TEGRA incurs a 15% performance hit due to its use of tree structure.

COST Analysis: The COST metric [47] is not designed for incremental systems, but we note that TEGRA is able to match the performance of an optimized single threaded implementation using 4 machines, each with 8 cores and has a COST of 32 cores. However, TEGRA uses property graphs while the optimized implementation does not.

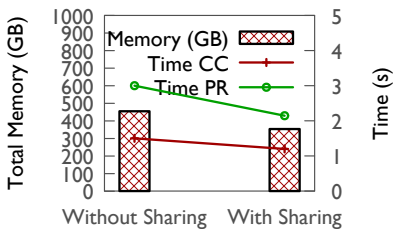

Figure 13: Sharing state across queries leads to reduction in memory usage and improvement in performance.

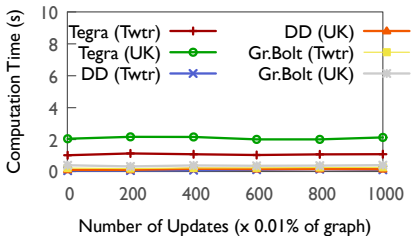

Figure 14: Both DD and GraphBolt significantly outperform TEGRA for purely streaming analysis.

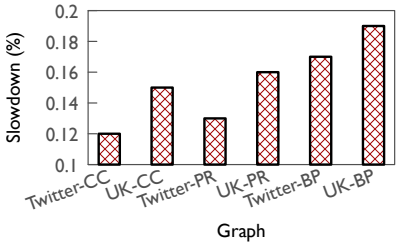

Figure 15: Tree based data structures in TEGRA result in a 15% slowdown compared to Chlonos for purely temporal analysis.

8 Related Work

Analytics on Static Graphs: A large number of graph processing systems [7, 8, 16, 18, 25–28, 37, 39–41, 56, 57, 59, 60, 65, 68–73, 76–81] focus on static graph processing, some of which are single machine systems and some are distributed. These systems do not consider evolving graph workloads.

(Transactional) Graph Stores: The problem of managing time-evolving graph has been studied in the context of graph stores [15, 52, 53, 56]. These focus on optimizing point queries which retrieves graph entities and do not support storing multiple snapshots. This yields a different set of challenges compared to iterative graph analytics.

Managing Graph Snapshots: DeltaGraph [33] proposes a hierarchical index that can manage multiple snapshots of a graph using deltas and event lists for efficient retrievals, but lacks the ability to do windowed iterative analytics. TAF [34] fixes this, but it is a specialized framework that does not provide a generalized incremental model or ad-hoc operations. LLAMA [42] uses a multi-version array to support incremental ingestion. It is a single machine system, and it is unclear how the multi-version array can be extended to support data parallel operations required for iterative analytics. Version Traveler [32] achieves switching between snapshots of a graph by loading the common subgraph in the compressed-sparse-row format and extending it with deltas. It does not support incremental computation. GraphOne [35, 36] uses dual-versioning to provide access to recent snapshots. It doesn't support ad-hoc analysis or efficient retrieval of arbitrary snapshots. Aspen [21] leverages functional data structures to build a compressed streaming graph engine, but doesn't support incremental computations. Chronos [30] and ImmortalGraph [50] optimizes for efficient computation across a series of snapshots. They propose an efficient model for processing temporal queries, and support snapshot storage of the graph on-disk using a hybrid model. While their technique reduces redundant computations in a given query, they cannot store and reuse intermediate computation results. Their in-memory layout of snapshots requires preprocessing and cannot support updates. None of these systems support storing computation state for later reuse.

Incremental Maintenance on Evolving Graphs: Kineograph [19] supports constructing consistent snapshots of an evolving graph for streaming computations but does not allow ad-hoc analysis. WSP [75] focuses on streaming RDF queries. GraphInc [17] supports incremental graph processing using memoization of the messages in graph parallel computation, but does not support snapshot generation or maintenance. Kickstarter [67] and GraphBolt [45] support non-monotonic computations, but do not support ad-hoc analysis or compactly storing graph and state. Differential Dataflow [48, 49, 51, 54] leverages indexed differences of data in its computation model to do non-monotonic incremental computations. However, it is challenging to do ad-hoc window operations using indexed differences (§2.3). As we demonstrate in our evaluation, compactly representing graph and computation state is the key to efficient ad-hoc window operations on evolving graphs.

Incremental View Maintenance (IVM): In databases, IVM algorithms [10, 29] maintain a consistent view of the database by reuse of computed results. However, they are tuned for different kinds of queries and not iterative graph computations. Further, they generate large intermediate state and hence require significant storage and computation cost [49].

Versioned File Systems (e.g., [64]) allow several versions of a file to exist at a time. However, they are focused on disk based files in contrast to in-memory efficiency.

9 Conclusion

In this paper, we present TEGRA, a system that enables efficient ad-hoc window operations on evolving graphs. The key to TEGRA's superior performance in such workloads is a compact, in-memory representation of both graph and intermediate computation state, and a computation model that can utilize it efficiently. For this, TEGRA leverages persistent data structures and builds DGSI, a versioned, distributed graph state store. It further proposes ICE, a general, non-monotonic iterative incremental computation model for graph algorithms. Finally, it enables users to access these states via a natural abstraction called Timelapse. Our evaluation shows that TEGRA is able to outperform existing temporal and streaming graph systems significantly on ad-hoc window operations.

Acknowledgements

We thank all our reviewers and our shepherd, Christopher Rossbach for the valuable feedback. In addition to NSF CISE Expeditions Award CCF-1730628, this research is supported by gifts from Amazon Web Services, Ant Group, CapitalOne, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk and VMware.

References

[1] Graph dbms increased their popularity by 500http://db-engines.com/en/blog_post//43, 2015 (accessed March 2021).

[2] A comparison of state-of-the-art graph processing systems. https://code.facebook.com/posts/319004238457019/a-comparison-of-state-of-the-art-graph-processing-systems/, 2016 (accessed March 2021).

[3] Differential dataflow rust implementation. https://github.com/TimelyDataflow/differential-dataflow, (accessed March 2021).

[4] Neo4j. http://www.neo4j.com, (accessed March 2021).

[5] Persistent adaptive radix tree. https://github.com/ankurdave/part, (accessed March 2021).

[6] Titan distributed graph database. http://thinkaurelius.github.io/titan/, (accessed March 2021).

[7] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 125–137, Santa Clara, CA, 2017. USENIX Association.

[8] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 125–137, Santa Clara, CA, 2017. USENIX Association.

[9] Leman Akoglu, Hanghang Tong, and Danai Koutra. Graph based anomaly detection and description: a survey. *Data Mining and Knowledge Discovery*, 29(3):626–688, 2015.

[10] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently updating materialized views. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, pages 61–71, New York, NY, USA, 1986. ACM.

[11] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*. ACM Press, 2011.

[12] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[13] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, New York, NY, USA, 2012. ACM.

[14] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[15] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. Tao: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, San Jose, CA, 2013. USENIX.

[16] Aydin Buluç and John R. Gilbert. The combinatorial BLAS: design, implementation, and applications. *IJHPCA*, 25(4):496–509, 2011.

[17] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. Facilitating real-time graph mining. In *Proceedings of the Fourth International Workshop on Cloud Data Management*, CloudDB '12, pages 1–8, New York, NY, USA, 2012. ACM.

[18] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 1:1–1:15, New York, NY, USA, 2015. ACM.

[19] 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.

[20] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.*, 8(12):1804–1815, August 2015.

[21] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 918–934, New York, NY, USA, 2019. Association for Computing Machinery.

[22] James R Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 109–121. ACM, 1986.

[23] Stephen Eubank, Hasan Guclu, VS Anil Kumar, Madhav V Marathe, et al. Modelling disease outbreaks in realistic urban social networks. *Nature*, 429(6988):180, 2004.

[24] Wenfei Fan, Chunming Hu, and Chao Tian. Incremental graph computations: Doable and undoable. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 155–169, New York, NY, USA, 2017. ACM.

[25] P. Gao, M. Zhang, K. Chen, Y. Wu, and W. Zheng. High performance graph processing with locality oriented design. *IEEE Transactions on Computers*, 66(7):1261–1267, July 2017.

[26] Joseph Gonzalez, Reynold Xin, Ankur Dave, Daniel Crankshaw, and Ion Franklin, Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO, October 2014. USENIX Association.

[27] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[28] Samuel Grossman, Heiner Litz, and Christos Kozyrakis. Making pull-based graph processing performant. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 246–260, New York, NY, USA, 2018. ACM.

[29] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 157–166, New York, NY, USA, 1993. ACM.

[30] 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.

[31] Anand Iyer, Li Erran Li, and Ion Stoica. Celliq : Real-time cellular network analytics at scale. In *Proceedings of the 12th USENIX conference on Networked Systems Design and Implementation*, NSDI'15, Berkeley, CA, USA, 2015. USENIX Association.

[32] Xiaoen Ju, Dan Williams, Hani Jamjoom, and Kang G. Shin. Version traveler: Fast and memory-efficient version switching in graph processing systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 523–536, Denver, CO, 2016. USENIX Association.

[33] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 997–1008, April 2013.

[34] Udayan Khurana and Amol Deshpande. Storing and analyzing historical graph data at scale. *CoRR*, abs/1509.08960, 2015.

[35] 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.

[36] Pradeep Kumar and H. Howie Huang. Graphone: A data store for real-time analytics on evolving graphs. *ACM Trans. Storage*, 15(4), January 2020.

[37] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, Hollywood, CA, 2012. USENIX.

[38] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 38–49, Washington, DC, USA, 2013. IEEE Computer Society.

[39] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. In Peter Grünwald and Peter Spirtes, editors, *UAI*, pages 340–349. AUAI Press, 2010.

[40] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woon-Hak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 527–543, 2017.

[41] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 527–543, New York, NY, USA, 2017. ACM.

[42] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. Llama: Efficient graph analytics using large multiversioned arrays. In *2015 IEEE 31st International Conference on Data Engineering*, pages 363–374, April 2015.

[43] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[44] Jasmina Malicevic, Baptiste Lepers, and Willy Zwaenepoel. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 631–643, Santa Clara, CA, 2017. USENIX Association.

[45] Mugilan Mariappan and Keval Vora. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 25:1–25:16, New York, NY, USA, 2019. ACM.

[46] 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 Comput. Surv.*, 48(2):25:1–25:39, October 2015.

[47] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[48] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. Shared arrangements: practical inter-query sharing for streaming dataflows. VLDB, 2020.

[49] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, 2013.

[50] 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*, 11(3):14:1–14:34, July 2015.

[51] Microsoft Naiad Team. GraphLINQ: A graph library for naiad. http://bigdataatsvc.wordpress.com/2014/05/08/graphlinq-a-graph-library-for-naiad/, 2014.

[52] Jayanta Mondal and Amol Deshpande. Eagr: Supporting continuous ego-centric aggregate queries over large dynamic graphs. *CoRR*, abs/1404.6570, 2014.

[53] Jayanta Mondal and Amol Deshpande. Stream querying and reasoning on social data. In *Encyclopedia of Social Network Analysis and Mining*, pages 2063–2075, 2014.

[54] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[55] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[56] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haradasan. Managing large graphs on multi-cores with graph awareness. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 41–52, Boston, MA, 2012. USENIX.

[57] 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.

[58] Real use case at Redacted, tier-1 cellular provider in USA. Company name redacted due to authors NDA with the company.

[59] 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.

[60] 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.

[61] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow.*, 11(4):420–431, December 2017.

[62] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 979–990, New York, NY, USA, 2014. ACM.

[63] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 505–516, New York, NY, USA, 2013. ACM.

[64] Craig AN Soules, Garth R Goodson, John D Strunk, and Gregory R Ganger. Metadata efficiency in a comprehensive versioning file system. 2002.

[65] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A system for distributed graph mining - extended version. *CoRR*, abs/1510.04233, 2015.

[66] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From "think like a vertex" to "think like a graph". *Proc. VLDB Endow.*, 7(3):193–204, November 2013.

[67] 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.

[68] Guozhang Wang, Wenlei Xie, Alan J Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.

[69] Guozhang Wang, Wenlei Xie, Alan J. Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*. www.cidrdb.org, 2013.

[70] Ming Wu and Rong Jin. A graph-based framework for relation propagation and its application to multi-label learning. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '06, pages 717–718, New York, NY, USA, 2006. ACM.

[71] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. Gram: Scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 408–421, New York, NY, USA, 2015. ACM.

[72] Wencong Xiao, Jilong Xue, Youshan Miao, Zhen Li, Cheng Chen, Ming Wu, Wei Li, and Lidong Zhou. Tux²: Distributed graph computation for machine learning. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 669–682, Boston, MA, 2017. USENIX Association.

[73] Wenlei Xie, Guozhang Wang, David Bindel, Alan Demers, and Johannes Gehrke. Fast iterative graph computation with block updates. *Proc. VLDB Endow.*, 6(14):2014–2025, September 2013.

[74] Jonathan S Yedidia, William T Freeman, and Yair Weiss. Generalized belief propagation. In *Advances in neural information processing systems*, pages 689–695, 2001.

[75] Haibo Chen Yunhao Zhang, Rong Chen. Sub-millisecond stateful stream querying over fast-evolving linked data. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17. ACM, 2017.

[76] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian. Graphp: Reducing communication for pim-based graph processing with efficient data partition. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 544–557, Feb 2018.

[77] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. Exploring the hidden dimension in graph processing. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 285–300, GA, 2016. USENIX Association.

[78] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. Exploring the hidden dimension in graph processing. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 285–300, Savannah, GA, 2016. USENIX Association.

[79] 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, pages 608–621, New York, NY, USA, 2018. ACM.

[80] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 293–302, Dec 2017.

[81] 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, GA, 2016. USENIX Association.