Tegra: Efficient Ad-Hoc Analytics on Time-Evolving Graphs

Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E. Gonzalez, Ion Stoica
UC 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. Existing systems face limitations when used for such tasks. We present Tegra, a system that enables efficient ad-hoc window operations on evolving graphs. Tegra enables 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 datastructures 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 extensively evaluate Tegra against existing evolving graph analysis techniques, and show that it significantly outperforms other 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, 52]. This growth has spurred the development of a large number of graph processing systems [15, 16, 18, 23, 24, 26, 30, 34, 37, 44, 47, 49–51, 58] 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 [19].

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 [27]. Additionally, emerging applications such as connected cars [13], real-time fraud detection [10], and disease analysis [21] all produce such graph data. Analyzing these *time-evolving* graphs can be useful, from scientific and commercial perspectives, and is often desired [52].

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 [26] allows executing queries on a single window, while Differential Dataflow [47] supports 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 time evolving graph workloads: *(1) during ad-hoc analysis graphs change slowly*

[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.

over time relative to their size , and (2) queries are frequently applied to multiple windows relatively close by in time.

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 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 (§ 3.1). 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. These allow Tegra to outperform existing systems by up to 30× 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. (§ 3.1)
- 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 iterative, incremental graph computation model which supports non-monotonic computations. (§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, followed by the workloads in time-evolving graph processing. We then discuss the limitations of existing approaches and describe the challenges in supporting efficient ad-hoc analysis on evolving graphs.

2.1 Graph-Parallel Systems

Most general purpose graph systems provide a **graph-parallel** abstraction for performing computations on graphs. 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. The simplest of the graph-parallel abstraction is a vertex-centric model [40], where every vertex independently runs the user program. Several other forms have been proposed, such as the graph centric model [56], edge centric models [50] and the more recent subgraph centric models [49]. 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 [54] or message passing interface [38]. PowerGraph [24] 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. Tegra focuses on the GAS model. (§6)

2.2 Time-evolving Graph Workloads

Time-evolving graph workloads, an important graph workload [52], can be of 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, and are executed in batch.

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 queries focus on the most recent data, thus streaming systems operate on live graph.

Ad-hoc Queries: In these workloads, an analyst is likely to explore the graph by performing ad-hoc queries on arbitrary windows. For example, consider a network administrator troubleshooting a transient failure that occurred at 09:00AM. To do so, she may ask *"What were the hotspots at 08:00AM?"*

which runs a connected component algorithm on the snapshot. Based on what she sees, she may ask *"What were the hotspots at 10:00AM?"* followed by *"At 10:00AM, what is the shortest path of hotspot X to the controller?"*. She iteratively refines this query by taking many snapshots and running the query.In another example, a financial expert is interested in improving the fraud-detection algorithm[4]. She queries *"Who were the top influencers 1 month around April 1?"* which retrieves the window of 1 month around a known fraud and runs an algorithm (e.g., personalized page rank) on the window, and then launches follow-up queries based on what she sees. She repeats this on different windows to learn new rules that would detect the fraud. She then tests her changes on a different set of windows, possibly also with injecting artificial data.

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 Limitations of Existing Solutions

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

Temporal analysis engines (e.g., Chronos [26], ImmortalGraph [43]) 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 [39]. 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.

Streaming systems (e.g., Kineograph [18], Differential Dataflow [42]) 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. Further, most systems (with the exception of Differential Dataflow, to the best of our knowledge) do not support *non-monotonic* computations in their incremental model, and either assume some properties of the algorithm, or leave it up to the developer to ensure correctness.

Differential Dataflow (DD) allows general, non-monotonic incremental computations using special versions of operators. Each operator stores "differences" to its input and produces the corresponding differences in output (hence 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 it completes and storing these efficiently for reuse across queries is an open question[5].

[4]Typically a combination of machine learning and expert rules.

2.4 Challenges

Meeting the requirements necessary to support efficient ad-hoc window operations in practice is hard. One of the key challenges in building such a system is efficient *in-memory* state management. It is ideal to store the history of the graph as individual *snapshots* for zero overhead ad-hoc retrieval, but the system needs to consider the storage overhead due to duplication with every snapshot stored. Similarly, for the system to accelerate queries across windows, it must not only be able to store intermediate states efficiently, but also be able to leverage them in its computation model. In essence, it must be able to compactly represent and share data and state between queries across multiple windows and users.

3 TEGRA Design

Our solution, TEGRA, consists of three components:

Timelapse Abstraction (§ 3.1): 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. 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.

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. In fact, Timelapse can be seen as "views" on the data stored in DGSI. Such decoupling of state from queries and operators allow TEGRA to share it across queries and users.

3.1 Timelapse Abstraction & API

TEGRA introduces *Timelapse* as a new abstraction for time-evolving graph processing that enables efficient ad-hoc ana-

[5]Our conversations with the author of DD revealed that incorporating the state management techniques we propose in this paper in DD is fundamentally hard and requires modification of its execution engine.

`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.

lytics. 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, 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 [23], 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 graph analysis (e.g., queries on an 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 users to efficiently operate on the timelapses. (§5)

Since timelapse logically represents a sequence of related graph snapshots, it is intuitive to expose the abstraction using

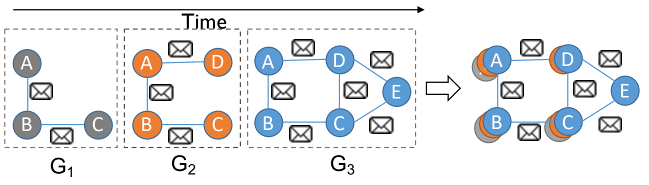

Figure 1: A timelapse of graph G consisting of three snapshots. For temporal analytics, instead of applying graph-parallel operations independently on each snapshot (left), timelapse enables them to be applied to all snapshots in parallel (right).

the same semantics as that of static graph. In TEGRA, users interact with timelapses using a language integrated API. The API 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. For example, a user familiar with querying the vertex degrees of a graph `G` in a static graph processing system as `G.degrees` can query the same at a snapshot `t` by using `G.retrieve(t).degrees`.

3.2 Evolving Graph Analytics Using Timelapse

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. 1 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 [53]. In the earlier example, sequentially applying the query to each snapshot results in 11 messages of which 5 are duplicates (fig. 1). In addition, some vertices compute the same value repeatedly, which is wasteful.

To avoid such inefficiencies, timelapse allows access to the *lineage* of graph entities. That is, it provides efficient 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 efficiently 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. Timelapse enables both local (e.g., developer can discard duplicate messages in each partition) and global optimizations (e.g., TEGRA can discard duplicates across partitions).

4 Computation Model

TEGRA's primary goal is to support efficient ad-hoc analytics on evolving graphs. The key challenge in doing so is in effectively reusing previous query results to reduce or eliminate redundant computations, commonly referred to as *incremental computations*. In this section, 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. By storing this state, the system only needs to perform computations on the *changed* data and can simply reuse the results for the unchanged data. In the context of *some* graph computations, this can be achieved by applying the graph-parallel stages only on the changes to the graph.

Revisiting the example of degree computation (fig. 1), if the analyst has already performed degree computation on G_1 and now wants to do the same on G_2, the system can leverage her last query result by executing the computation only on the changes (vertex D and edge A to D). Here, vertices D and A send messages with value 1 to each other, and vertex A adds this to its previous result to obtain the correct result. Timelapse enables such incremental computations trivially by using **diff**() to find the changes and **merge**() to combine the partial and previous results to obtain the updated results.[6]

```
results = G1.degrees() // previously computed
diffs = G.diff(G2, G1)
changes = diffs.degrees()
new_results = G.merge(results, changes, _+_)
```

However, this technique makes several assumptions—such as single-pass[7] implementation, monotonic computations (retractions are not allowed, e.g., the degree computation requires special handling for deletions and/or updates) and the associative and commutative properties of the user-defined vertex functions involved in the stages—which makes it impractical. We do not make these assumptions (§ 4.3).

4.2 Incremental Iterative Graph Computations

Many graph algorithms are iterative in nature, where the graph-parallel stages are repeatedly applied in sequence until a fixed point (§ 2.1). Here, simply restarting the computations from previous results do not lead to correct answers. To illustrate this, consider a connected components algorithm

[6]The Scala syntax _+_ instructs the system to add new values to old values for vertices present in both snapshots that are being merged.

[7]Here, each graph-parallel stage executes only once.

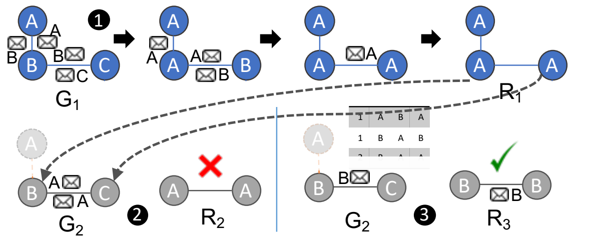

Figure 2: ❶ Connected components by label propagation on snapshot G_1 produces R_1. ❷ Vertex A and edge $A-B$ is deleted in G_2. Using the last result to bootstrap computation results in incorrect answer R_2. ❸ A strawman approach of storing all messages during the initial execution and replaying it produces correct results, but needs to store large amounts of state.

using label propagation on a graph snapshot, G_1 as shown in ❶ in fig. 2 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., functions form an abelian group) and the monotonicity of updates (e.g., the graph only grows).

Supporting general non-monotonic iterative computations require maintaining *intermediate* state. In the previous example, one solution is to store every message exchanged between graph entities during the initial execution of the algorithm. When the query is executed on the updated graph, the system can selectively replay these stored messages to ensure correctness of the results as depicted in ❸. However, this approach requires storing and effectively using large amounts of state[8], which may pose prohibitive overheads when applied to real-world graphs where the number of edges are significantly more compared to vertices [24]. Further, the state is tied to the computation performed and thus doesn't provide opportunities to share it across queries.

TEGRA proposes a general, incremental iterative graph-parallel computation model that significantly reduces the state requirements. We leverage 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. Since timelapse can efficiently store and retrieve these snapshots, we can perform incremental computations without the need to store the message exchanges. We call this model *Incremental Computation by entity Expansion* (ICE).

4.3 ICE Computation Model

At a high level, ICE executes graph-parallel computations only on the subgraph that would be 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.

[8]The state is proportional to the number of edges for every iteration.

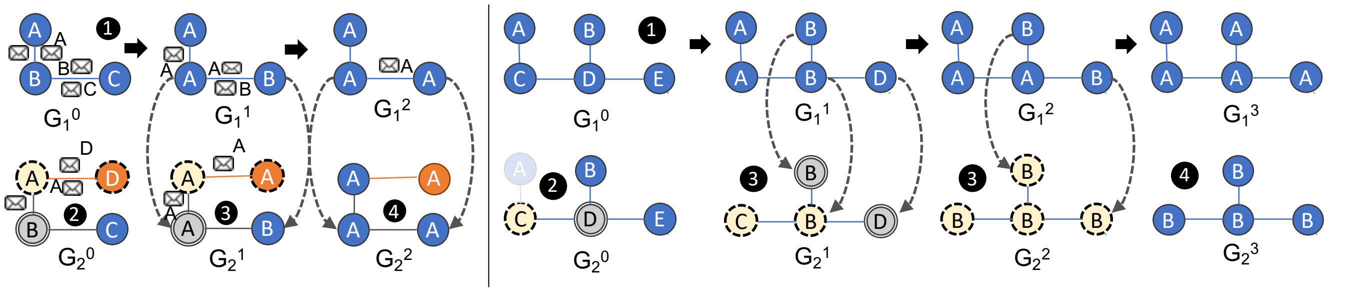

Figure 3: Two examples that depict how ICE works. Dotted circles indicate vertices that recompute, and double circles indicate vertices that need to be present in the subgraph to compute the correct answer, but do not recompute state themselves. ❶ Iterations of initial execution is stored in the timelapse. ❷ ICE bootstraps computation on a new snapshot, by finding the subgraph consisting of affected vertices and their dependencies (neighbors). In the second example, C is affected by the deletion of $A-C$. To recompute state it needs D (yields subgraph $C-D$). ❸ 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 the second example, 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. ❹ ICE terminates when the subgraph converges and no entity in the graph needs the state copied from stored snapshots in the timelapse.

Initial execution: When an algorithm is executed for the first time, ICE stores the state 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 in the examples in fig. 3, 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 changes 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. The first example's ❷ in fig. 3 shows how ICE bootstraps when a new vertex D and a new edge between A and 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 the beginning of 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 and expanding to the one-hop neighborhood of affected entities. For the vertices/edges that did not recompute the state, ICE simply copies the state from the timelapse. In ❸ in fig. 3 for the first example, 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) graph-parallel computation from that point. Thus, ICE converges only when the subgraph converges and no graph entity needs their state to be copied from the stored snapshot in the timelapse. (❹ in fig. 3)

4.4 Improving ICE Model

Incremental computations may not be beneficial 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. 3). To perform incremental computation, a system needs to do book-keeping of state and perform comparisons which costs computation cycles. Due to this, the total work done by the system may exceed that of completely executing the computation from scratch [57]. Since ICE generates the exact same intermediate states at every iteration as a system that executes the algorithm from scratch (i.e., non-incremental computation), TEGRA can address this inefficiency easily. We use a simple cost model, based on the number of entities that would execute computation, to determine, at every iteration, whether to proceed with incremental computation on the subgraph or switch to full execution on the entire graph[9].

ICE provides several desirable properties for ad-hoc and exploratory analytics on evolving graphs. Since the state is immutable and separate from computation, it can easily be shared between multiple queries. Further, since ICE can utilize any state, there is no ordering constraints (e.g., an analyst can run a query on a graph snapshot at 9:00am and use the state to run the query on 8:00am snapshot). The immutability aspect also allow queries to continue running when the underlying graph evolves. This lets TEGRA easily incorporate recently proposed approximate streaming computation models [28] which copies the real-time computation state to a new snapshot when it is available. Though ICE can work on any graph-parallel model, we restrict our scope to the GAS model in this work, and discuss how to implement ICE in it in §6.

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 requirements (1) and (2) by leveraging persistent datastructures 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 Datastructures

In TEGRA, we leverage persistent datastructures [20] to build a distributed, versioned graph state store. The key idea in persistent datastructures is to maintain the previous versions of data when modified, thus allowing access to earlier versions. DGSI uses a persistent version of the Adaptive Radix Tree [33] as its datastructure. ART provides several properties useful for graph storage such as efficient updates and range scans. Persistent Adaptive Radix Tree (PART) [3] 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,

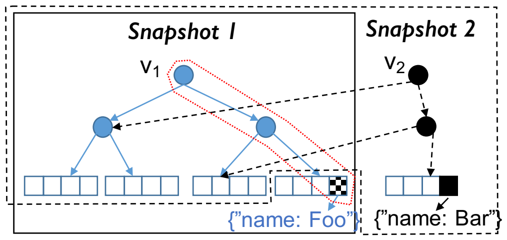

Figure 4: TEGRA's DGSI consists of one pART datastructure for vertices and one for edges on each of the partitions. Here, a vertex pART stores properties in its leaves. Vertex id traverses the tree to the leaf storing its property. Changes generate new versions.

avoiding unnecessary small object creation and optimizing path copying under heavy writes.

5.2 Graph Storage & Partitioning

TEGRA stores graphs using two pART datastructures: 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 primitve types.

TEGRA supports several graph partitioning schemes, similar to GraphX [23], 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. 4). By using local trees, we further amortize the (already low) cost[10] associated with modifying the tree upon graph updates. To consume graph updates, TEGRA needs to send the updates to the right partition. For this, we impose the *same* partitioning as the original graph on the vertices and edges in the update.

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

[9]We use 50% threshold, and are actively exploring better cost models.

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

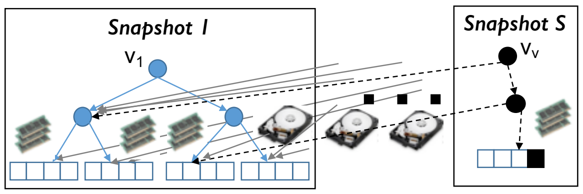

Figure 5: DGSI has fine-grained control over leaves (where data is stored). Here DGSI has 1000s of snapshots. All snapshots except *S* is on disk, their parents just hold pointers to the file. Parents are also dynamically written to disk if all of their children are on disk. Datastructure uses adaptive leaf sizes for efficiency.

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 [6], Titan [5] or Weaver [7] 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. Technically, DGSI can ingest each update to the graph as a separate version. However, there are two inefficiencies with it. First, pART needs to copy a leaf and its parents for each update. Second, our execution engine, Spark [65], is inefficient for record-by-record operations. Thus, 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 of the snapshot.

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 the version id.

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. 4, 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. 5, 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

We have implemented Tegra on Apache Spark [65] as a drop-in replacement for GraphX [23], its graph processing library[11], in about 3800 lines of Scala code. We describe details of our ICE implementation specific to the GAS model.

6.1 ICE on GAS Model

As described in § 4.3, the `diff()` API marks the candidates that must perform graph-parallel computation in a given iteration. In GAS decomposition, 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`[12] 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 new 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`

[11] Tegra is a new implementation and does not extend GraphX's codebase.
[12] `EdgeDirection` in GraphX

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

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

`()` 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)` enables retrieval of vertices at a given snapshot id, and `Graph.mapV([ids])` can apply a map function on vertices of the graph on a set of snapshots). Graph-parallel computation is enabled in GraphX using the `Graph.aggregateMessages()` (previously `mrTriplets()`) API. Because this API works on the `Graph` interface, developers can directly apply it to the subgraph found by using Tegra's **`diff`**`()` and **`expand`**`()` calls and then use **`merge`**`()` to materialize the complete result.

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. Tegra provides an incremental version of this Pregel API that can perform ICE from a previously saved computation state provided as an argument. Internally, we invoke **`diff`**`()` and **`expand`**`()` before the `aggregateMessages` call, and replace the `joinVertices` with **`merge`**`()` followed by a **`save`**`()` of the updated state. In general, a developer can write incremental versions of any iterative graph parallel algorithm by using the Tegra APIs along with `aggregateMessages`. Tegra provides a library of incremental versions of commonly used graph algorithms.

7 Evaluation

We have evaluated Tegra through a series of experiments.

Comparisons. We compare Tegra against a streaming engine and a temporal engine. For streaming system, we use the Rust implementation of Differential Dataflow (DD) [4]. Since we were unable to obtain an open source implementation of a temporal engine, we developed a simplified version of Chronos [26] in GraphX [23], which we call Clonos (Clone of Chronos) in this section. This implementation emulates an array based in-memory layout of snapshots and the incremental computation model in Chronos. In particular, for computation on multiple snapshots, we compute the result on the first snapshot and use it to bootstrap the rest. For graph updates, we use a mixture of additions and deletions.

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 v0.5.0 and Apache Spark v2.3.0. We warm up the JVM before measurements.

Caveats. While perusing the evaluation results, we wish to remind the reader a few caveats. Tegra is implemented on a JVM based system and thus it is unfair to compare the performance numbers directly. Further, Spark's execution model differs significantly from Timely Dataflow, the engine that powers DD. Tegra supports edge and vertex properties (and creates a default value) while DD offers no such support which enables it to use highly efficient datastructures.

7.1 DGSI Microbenchmarks

We first evaluate the effectiveness of DGSI.

Graph Update Throughput: We evaluated the ability of Tegra to sustain high update volumes of the underlying graph. We load the Twitter graph on Tegra and DD. We then randomly add and remove 1 million edges in the graph, and note the time to store the updates for each system to compute its update throughput. We do not perform any computation. We then average this over 10 runs and repeat the experiment with varying number of machines. Clonos is not used since its in-memory layout does not allow updates. From fig. 6, we see that Tegra is able to average about 1 million updates per machine. These numbers scale linearly with more machines, which is expected as there is no coordination required for updating the graph. While we do not show here, DD is able to ingest 20 million updates per machine, but it is simply adding them in native arrays. In contrast, Tegra applies the updates to the graph which is stored in a tree datastructure.

Snapshot Retrieval Latency: Next, we repeat the experiment to evaluate the snapshot retrieval latency. After 1000 updates, we retrieve random snapshots from graph. For each retrieval, we note the time for the system to materialize the output. We note the average of 10 retrievals each on different number of machines in fig. 7. DD does periodic state compaction to reduce state overheads. However, since this results in the inability to retrieve the past, we disable compaction.

We see that Tegra is able to return the queried snapshot with no computation at all, since it materializes the snapshot at ingestion time. In contrast, DD needs to reconstruct the graph when it is queried. Since it does not offer the capability to retrieve any version, we retrieve the current version. Reconstructing the graph takes about 20 seconds on a single machine, and reduces linearly with number of machines. However, as more updates are added, the retrieval time degrades since the system needs to accumulate all.

Snapshot Storage Overhead: To evaluate the snapshot storage overhead, we ran an experiment on a single high memory instance in Amazon EC2 to estimate how many

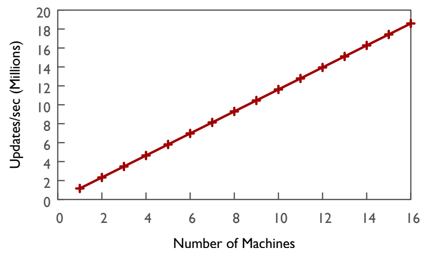

Figure 6: TEGRA can sustain high ingest rates and exhibits linear scaling.

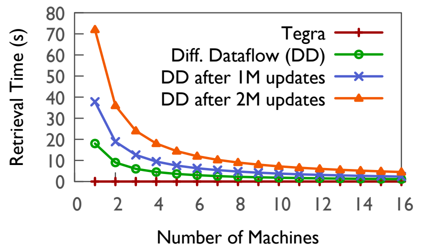

Figure 7: Snapshot retrieval latency in DD incurs cost and degrades with time.

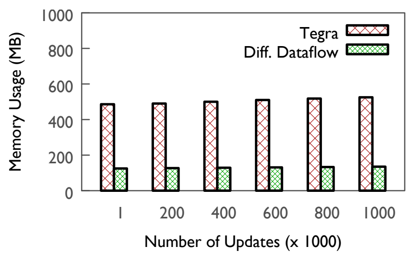

Figure 8: TEGRA can store a large number of snapshots with minimal overheads.

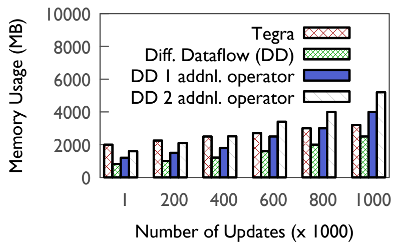

Figure 9: Differential dataflow's memory usage increases with more operators, and surpasses TEGRA's.

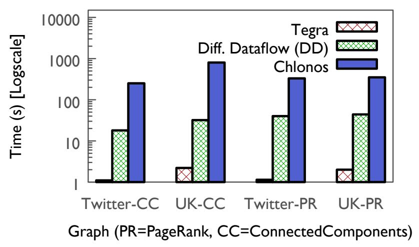

Figure 10: On ad-hoc queries on snapshots, TEGRA is able to significantly outperform due to state reuse.

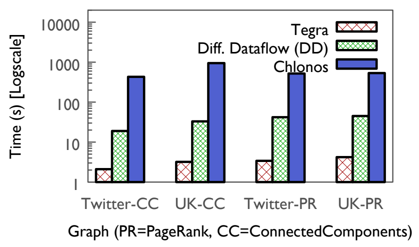

Figure 11: TEGRA's performance is superior on ad-hoc window operations even with materialization of results.

full snapshots can be stored without DGSI in a system like Chlonos. We generated a synthetic graph with 1 million vertices and 10 million edges to allow storing many snapshots. Then we randomly add and remove 1000 edges to create a snapshot. We measure the memory usage by each system after intervals of 200 batches (hence, a total of 1 million updates at the end of the experiment). Figure 8 shows the results.

While not shown, a full snapshot uses 370 MB each, and thus only around 500 snapshots can be stored. TEGRA requires 486MB to store the first snapshot, a 1.3× increase in memory usage compared to using Chlonos. This overhead is due to TEGRA's use of tree structure compared to columnar arrays, and is a small price to pay which is quickly amortized as more snapshots are stored in the system. With increasing number of snapshots, we see only a small increase in memory usage. DD uses only about 125MB to store the graph and exhibits linear increase with updates. The comparatively higher memory usage for TEGRA is because it stores default properties for vertices and edges, and some auxiliary structures.

Computation State Storage Overhead: Finally we measured the memory overhead due to computation state. We use the same setting as the last experiment, and run Breadth First Search (BFS) in an incremental fashion, and note the memory usage by each system after every 200 updates until 1000 updates (for a total of 1 million edge updates). Figure 9 shows this experiment's results. TEGRA uses around 2GB of memory once the first graph update has been used to compute the results. This is approximately 4× the memory usage when no computations were run (fig. 8). DD uses about 821MB, about 7× that of the memory usage when it was not doing any computations. Both TEGRA and DD use more memory as updates are applied and the systems compute results, but DD exhibits steeper increase over time due to per-operator state. Further, DD's state depends on the number of operators. To show this, we wrote the DD program using more operators. The memory usage increases steeply, and with two additional operators, surpasses the memory requirements of TEGRA.

7.2 Ad-hoc Window Operations

Here, we present evaluations that focus on our computation model ICE. In these experiments, we emulate an analyst. We load the graph, and apply a large number of sequential updates to the graph, where each update modifies 0.1% of the edges. We then retrieve 100 random windows of the graph that are close-by, and apply queries in each. We use page rank and connected components as the algorithms. Page rank is either run until a specific tolerance, or 20 iterations, whichever is smaller. We assume that earlier results are available so that the system could do incremental computations. We do not consider the window retrieval time in this experiment for DD and Chlonos. We present the average time taken to compute the query result once the window is retrieved.

Single Snapshot Operations: In the first experiment, we set the window size to zero so every window retrieval returns a single snapshot. The results are depicted in fig. 10. DD 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

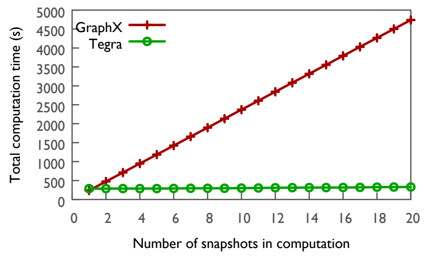

Figure 12: Timelapse can be used to optimize graph-parallel stage.

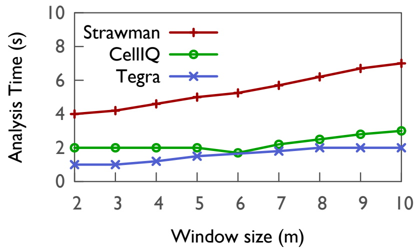

Figure 13: APIs can reimplement specialized systems with good performance

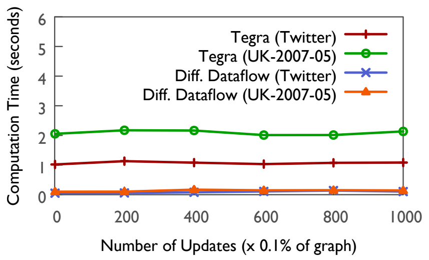

Figure 14: DD significantly outperforms TEGRA for purely streaming analysis.

in its DGSI from earlier queries for much faster computation. In this case, most of the snapshots incurs no computation overhead because of the extremely small amount of changes between them, and TEGRA is able to produce an answer within a few seconds. DD takes a few 10s of seconds, while Chlonos requires 100s of seconds. TEGRA's benefits range from 18-30× compared to DD.

Window Operations: Here we set the window size to be 10 snapshots. Chlonos and DD are able to apply incremental computations once the query has been computed on the first snapshot. Figure 11 shows the results. We see that DD is very fast once the first result has been computed, and incurs minimal overheads after that. In contrast, Chlonos incurs a penalty because it uses the first result to bootstrap the rest. Because TEGRA needs to materializes the result and also store it separately for each snapshot, and due to scheduling overheads in Spark it incurs a slight penalty compared to a single snapshot. TEGRA is still 9-17× faster compared to DD.

Large Graphs: Finally, we evaluate the ability of TEGRA to support ad-hoc window analysis on very large graphs. For this, we use synthetic graphs provided by Facebook [2] modeled using the social network's properties. There are 3 graphs, with 5B, 10B and 50B edges respectively. For comparison, we provide the time for Giraph and GraphX from [2].

Graph	5B		10B		50B	
	PR	CC	PR	CC	PR	CC
Giraph	2m	1m	4m	1.5m	22.5m	4.5m
GraphX	6m	3.5m	18m	12m	-	-
TEGRA	**10s**	5s	**19s**	**7s**	**1.5m**	**18s**

Table 3: Ad-hoc analytics on big graphs. A '-' indicates the system failed to run the workload.

Here, we load the graph and execute the queries once. Then we modify the graph by a tiny percentage, 0.01% randomly 1000 times to create 1000 snapshots. We then randomly pick a snapshot, and run the queries on it. We provide the average time over 100 such runs. The results are shown in table 3. We were unable to get differential dataflow work on these graphs, as it failed due to excessive memory usage during initial execution. In contrast, TEGRA is not only able to keep the state compact and scale to large graphs, but also provide significant benefits by using previous computation state. TEGRA carries the performance to larger graphs, the runtime increases linearly, which is intuitive for PageRank. Note that Giraph and GraphX recompute the entire results as they do not support incremental computations.

7.3 Usefulness of Timelapse

One of the goals of Timelapse abstraction is to provide efficient ways for graph-parallel phases to perform operations (§ 3.2) and make it easy to write new evolving graph analysis tasks. We evaluate this aspect.

Parallel computations. We develop a simple parallel computation model (§ 3.2) in which a query applied to a sequence of snapshots can be run in parallel on all the snapshots. That is, instead of running the query snapshot-by-snapshot, we use timelapse to combine graph-parallel stage across snapshots.

We create 20 snapshots of the Twitter graph by starting with 80% and adding 1% to it repeatedly. We then apply the connected components algorithm on these snapshots where we vary the number of snapshots on which the algorithm runs. Thus, the value in the X-axis of this plot indicates the number of snapshots included in the computation. In each run, we measure the time take 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. 12. We see that the computation time on a single snapshot is a little worse compared to GraphX. This is due to the use of tree datastructures compared to GraphX's arrays. Even then, we see a linear trend with increasing number of snapshots. By sharing computation and communication, TEGRA is able to achieve up to 14× speedup.

New applications. *"Can TEGRA's APIs be used to implement specialized systems?"*. CellIQ [27] has shown that incremental connected component can be very useful in domain specific analytics and proposes a specialized system for this purpose. We implemented the persistent hotspot detection algorithm in it. We compare this against a strawman implementation of the algorithm that simply buffers the graph in the window, combines it and runs connected components. Figure 13 shows that Tegra provides significant benefits over the strawman. Our results are better due to the use of DGSI.

7.4 Tegra Shortcomings

Finally, we ask the question "What does Tegra **not** do well?". Tegra's design goal is ad-hoc window analytics, so it may not be optimal in two settings:

Purely Streaming Analysis: For this experiment, we consider an online query (§2) of connected components. To emulate a streaming graph, we first load the graph and continuously change 0.1% of the graph by adding and deleting edges.We assume that earlier results are available so that the system could perform incremental computation. We do this as follows: after a fixed number of updates, we stop and do a complete computation to create previous state. Then we do incremental computation from the next update. The average runtime of 10 runs is shown in fig. 14. We see that DD is significantly better than Tegra for such workloads, providing 20-30X improvements. This is due to a combination of DD optimized for online queries and its Rust implementation. In contrast, Tegra materializes the result after each computation, and incurs scheduling overheads due to Spark.

Purely Temporal Analysis: We also consider a purely temporal query. Here, we assume that the system knows the queries and the window before the start, and thus it has optimized the data layout for the query. We run a simple query on a window size of 10 and compare Tegra and Clonos. Excluding processing time, we noticed that Tegra takes around a 15% performance hit due to its use of tree structures in DGSI.

COST Analysis: The COST metric [41] is not designed for incremental systems, but we note that Tegra is able to run pagerank on Twitter graph in 16 machines, each with 8 cores in about 100 seconds and has a COST of 128 cores, mainly due to its use of Spark for execution.

8 Related Work

Analytics on Static Graphs: A large number of graph processing systems [8, 9, 15, 17, 22–25, 32, 34–36, 48–51, 55, 58–63, 66–71] 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-evovling graph has been studied in the context of graph stores [7, 14, 45, 46, 48]. 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: A lot of systems took the idea to manage snapshots for evolving graphs, so the problem is converted to analytics on a series of static graphs. DeltaGraph [30] 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 [31] fixes this, but it is a specialized framework that does not provide a generalized incremental model or ad-hoc operations. LLAMA [37] uses a multiversion array to support incremental ingestion. It is a single machine system, and it is unclear how the multiversion array can be extended to support data parallel operations required for iterative analytics. Version Traveler [29] achieves swift switching between snapshots of a graph by loading the common subgraph in the compressed-sparse-row format and extending it with deltas. However, it does not support incremental computation. Chronos [26] and ImmortalGraph [43] 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. Further, their incremental computation model does not support non-monotonic computations. None of these systems allow compactly representing computation state.

Incremental Maintenence on Evolving Graphs: Another important body of work are the streaming systems. CellIQ [27] is a specialized system for cellular network analytics, but it does not support ad-hoc analysis or compactly storing graph and state. Kineograph [18] supports constructing consistent snapshots of an evolving graph for streaming computations but does not allow ad-hoc analysis. WSP [64] focuses on streaming RDF queries. GraphInc [16] supports incremental graph processing using memoization of the messages in graph parallel computation, but does not support snapshot generation or maintenance. Kickstarter [57] supports edge deletions, but only for some algorithms. It does not support ad-hoc analysis. Differential Dataflow [42, 44, 47] 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.

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 datastructures 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 analysis systems significantly on ad-hoc window operations.

References

[1] Graph dbms increased their popularity by 500http://db-engines.com/en/blog_post//43, 2015 (accessed August 2018).

[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 August 2018).

[3] Persistent adaptive radix tree. https://github.com/ankurdave/part, (accessed April 2018).

[4] Differential dataflow rust implementation. https://github.com/frankmcsherry/differential-dataflow, (accessed August, 2018).

[5] Titan distributed graph database. http://thinkaurelius.github.io/titan/, (accessed August, 2018).

[6] Neo4j. http://www.neo4j.com, (accessed September 2018).

[7] Weaver: A scalable, fast, consistent graph store. http://weaver.systems, (accessed September 2018).

[8] Ai, Z., Zhang, M., Wu, Y., Qian, X., Chen, K., and Zheng, W. 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)* (Santa Clara, CA, 2017), USENIX Association, pp. 125–137.

[9] Ai, Z., Zhang, M., Wu, Y., Qian, X., Chen, K., and Zheng, W. 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)* (Santa Clara, CA, 2017), USENIX Association, pp. 125–137.

[10] Akoglu, L., Tong, H., and Koutra, D. Graph based anomaly detection and description: a survey. *Data Mining and Knowledge Discovery 29*, 3 (2015), 626–688.

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

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

[13] Bonomi, F., Milito, R., Zhu, J., and Addepalli, S. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing* (New York, NY, USA, 2012), MCC '12, ACM, pp. 13–16.

[14] Bronson, N., Amsden, Z., Cabrera, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Kulkarni, S., Li, H., Marchukov, M., Petrov, D., Puzar, L., Song, Y. J., and Venkataramani, V. Tao: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (San Jose, CA, 2013), USENIX, pp. 49–60.

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

[16] Cai, Z., Logothetis, D., and Siganos, G. Facilitating real-time graph mining. In *Proceedings of the Fourth International Workshop on Cloud Data Management* (New York, NY, USA, 2012), CloudDB '12, ACM, pp. 1–8.

[17] Chen, R., Shi, J., Chen, Y., and Chen, H. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA, 2015), EuroSys '15, ACM, pp. 1:1–1:15.

[18] Cheng, R., Hong, J., Kyrola, A., Miao, Y., Weng, X., Wu, M., Yang, F., Zhou, L., Zhao, F., and Chen, E. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 85–98.

[19] Ching, A., Edunov, S., Kabiljo, M., Logothetis, D., and Muthukrishnan, S. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow. 8*, 12 (Aug. 2015), 1804–1815.

[20] Driscoll, J. R., Sarnak, N., Sleator, D. D., and Tarjan, R. E. Making data structures persistent. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing* (1986), ACM, pp. 109–121.

[21] Eubank, S., Guclu, H., Kumar, V. A., Marathe, M. V., et al. Modelling disease outbreaks in realistic urban social networks. *Nature 429*, 6988 (2004), 180.

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

[23] Gonzalez, J., Xin, R., Dave, A., Crankshaw, D., and Franklin, Stoica, I. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association.

[24] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 17–30.

[25] Grossman, S., Litz, H., and Kozyrakis, C. Making pull-based graph processing performant. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2018), PPoPP '18, ACM, pp. 246–260.

[26] Han, W., Miao, Y., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, W., and Chen, E. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 1:1–1:14.

[27] Iyer, A., Li, L. E., and Stoica, I. Celliq : Real-time cellular network analytics at scale. In *Proceedings of the 12th USENIX conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2015), NSDI'15, USENIX Association.

[28] Iyer, A. P., Li, L. E., Das, T., and Stoica, I. Time-evolving graph processing at scale. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems* (New York, NY, USA, 2016), GRADES '16, ACM, pp. 5:1–5:6.

[29] Ju, X., Williams, D., Jamjoom, H., and Shin, K. G. Version traveler: Fast and memory-efficient version switching in graph processing systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, 2016), USENIX Association, pp. 523–536.

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

[31] Khurana, U., and Deshpande, A. Storing and analyzing historical graph data at scale. *CoRR abs/1509.08960* (2015).

[32] Kyrola, A., Blelloch, G., and Guestrin, C. 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)* (Hollywood, CA, 2012), USENIX, pp. 31–46.

[33] Leis, V., Kemper, A., and Neumann, T. The adaptive radix tree: Artful indexing for main-memory databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)* (Washington, DC, USA, 2013), ICDE '13, IEEE Computer Society, pp. 38–49.

[34] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M. Graphlab: A new framework for parallel machine learning. In *UAI* (2010), P. Grünwald and P. Spirtes, Eds., AUAI Press, pp. 340–349.

[35] Maass, S., Min, C., Kashyap, S., Kang, W., Kumar, M., and Kim, T. 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* (2017), pp. 527–543.

[36] Maass, S., Min, C., Kashyap, S., Kang, W., Kumar, M., and Kim, T. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 527–543.

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

[38] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 135–146.

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

[40] McCune, R. R., Weninger, T., and Madey, G. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv. 48*, 2 (Oct. 2015), 25:1–25:39.

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

[42] McSherry, F., Murray, D. G., Isaacs, R., and Isard, M. Differential dataflow. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings* (2013).

[43] Miao, Y., Han, W., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, E., and Chen, W. Immortalgraph: A system for storage and analysis of temporal graphs. *Trans. Storage 11*, 3 (July 2015), 14:1–14:34.

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

[45] Mondal, J., and Deshpande, A. Eagr: Supporting continuous ego-centric aggregate queries over large dynamic graphs. *CoRR abs/1404.6570* (2014).

[46] Mondal, J., and Deshpande, A. Stream querying and reasoning on social data. In *Encyclopedia of Social Network Analysis and Mining* (2014), pp. 2063–2075.

[47] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 439–455.

[48] Prabhakaran, V., Wu, M., Weng, X., McSherry, F., Zhou, L., and Haradasan, M. Managing large graphs on multi-cores with graph awareness. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)* (Boston, MA, 2012), USENIX, pp. 41–52.

[49] Quamar, A., Deshpande, A., and Lin, J. Nscale: Neighborhood-centric large-scale graph analytics in the cloud. *The VLDB Journal 25*, 2 (Apr. 2016), 125–150.

[50] Roy, A., Bindschaedler, L., Malicevic, J., and Zwaenepoel, W. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 410–424.

[51] Roy, A., Mihailovic, I., and Zwaenepoel, W. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 472–488.

[52] Sahu, S., Mhedhbi, A., Salihoglu, S., Lin, J., and Özsu, M. T. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow. 11*, 4 (Dec. 2017), 420–431.

[53] Satish, N., Sundaram, N., Patwary, M. M. A., Seo, J., Park, J., Hassaan, M. A., Sengupta, S., Yin, Z., and Dubey, P. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, ACM, pp. 979–990.

[54] Shao, B., Wang, H., and Li, Y. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, ACM, pp. 505–516.

[55] Teixeira, C. H. C., Fonseca, A. J., Serafini, M., Siganos, G., Zaki, M. J., and Aboulnaga, A. Arabesque: A system for distributed graph mining - extended version. *CoRR abs/1510.04233* (2015).

[56] Tian, Y., Balmin, A., Corsten, S. A., Tatikonda, S., and McPherson, J. From "think like a vertex" to "think like a graph". *Proc. VLDB Endow. 7*, 3 (Nov. 2013), 193–204.

[57] Vora, K., Gupta, R., and Xu, G. 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* (New York, NY, USA, 2017), ASPLOS '17, ACM, pp. 237–251.

[58] Wang, G., Xie, W., Demers, A. J., and Gehrke, J. Asynchronous large-scale graph processing made easy. In *CIDR* (2013).

[59] Wang, G., Xie, W., Demers, A. J., and Gehrke, J. Asynchronous large-scale graph processing made easy. In *CIDR* (2013), www.cidrdb.org.

[60] Wu, M., and Jin, R. 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* (New York, NY, USA, 2006), SIGIR '06, ACM, pp. 717–718.

[61] Wu, M., Yang, F., Xue, J., Xiao, W., Miao, Y., Wei, L., Lin, H., Dai, Y., and Zhou, L. Gram: Scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (New York, NY, USA, 2015), SoCC '15, ACM, pp. 408–421.

[62] Xiao, W., Xue, J., Miao, Y., Li, Z., Chen, C., Wu, M., Li, W., and Zhou, L. Tux²: Distributed graph computation for machine learning. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association, pp. 669–682.

[63] Xie, W., Wang, G., Bindel, D., Demers, A., and Gehrke, J. Fast iterative graph computation with block updates. *Proc. VLDB Endow. 6*, 14 (Sept. 2013), 2014–2025.

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

[65] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 2–2.

[66] Zhang, M., Wu, Y., Chen, K., Qian, X., Li, X., and Zheng, W. Exploring the hidden dimension in graph processing. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 285–300.

[67] Zhang, M., Wu, Y., Chen, K., Qian, X., Li, X., and Zheng, W. Exploring the hidden dimension in graph processing. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, 2016), USENIX Association, pp. 285–300.

[68] Zhang, M., Wu, Y., Zhuo, Y., Qian, X., Huan, C., and Chen, K. 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* (New York, NY, USA, 2018), ASPLOS '18, ACM, pp. 608–621.

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

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

[71] Zhu, X., Chen, W., Zheng, W., and Ma, X. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 301–316.