1. Problem & Motivation

Graph-structured data is on the rise, in size, complexity and dynamism [1]. This growth has spurred the development of a large number of graph processing systems [5, 6, 8, 10–12, 14, 16–18, 21, 24–27, 29] in both academia and open-source community. By leveraging specialized abstractions and careful optimizations, these systems have the ability to analyze large, static graphs, some in the order of trillion edges [9]. However, real-world graphs are seldom static. Consider, for instance, the familiar example of social network graphs such as in Facebook and Twitter. In such networks, “friends” relations, tweets with “mentions” are created continuously resulting in the graph’s constant evolution. The dynamic aspect of such graphs makes it more difficult to answer questions like, “What are the trending topics at the moment?”, or “Who was Bob’s most active friend in 2016?”. In another example, cellular operators collect massive amounts of real-time data useful for network diagnostics [13]. We can also find evolving graphs in Internet of Things (IoT) applications such as connected cars [3], transaction graph in financial networks, and disease propagation graph. Mining these time-evolving graphs can be useful, from both scientific and commercial perspectives.

Ideally, a system for time-evolving graph processing must be able to provide the same analysis abilities as that on static graphs, but on any arbitrary point or points in the history of the graph. Intuitively, a time-evolving graph can be seen as a series of static graphs, called snapshots. Thus, time-evolving graph processing systems must be able to support interactive ad-hoc, incremental and streaming analysis on any arbitrary snapshot, or snapshot windows. The big challenge in achieving this ability is in efficiently accessing arbitrary snapshots and performing computations on them. There are two main techniques to provide access to snapshots: (1) store each snapshot that the user might ask for, (2) log all changes to the graph (commonly referred as deltas), and then use these logs to compute the requested snapshots. Unfortunately, the former technique becomes quickly infeasible due to the prohibitive storage requirements, while the later technique becomes computationally expensive and slow over time. While recent work in graph systems has made considerable progress to address these needs, they often only address a subset or are fundamentally inefficient.

In this work, we present Tegra, a time-evolving graph processing system that addresses this challenge. The key idea in Tegra is to share storage, computation and communication across snapshots in a time-evolving graph. Tegra hides the intricacies of state management and sharing from the end-user using Timelapse, a new abstraction for time-evolving graph processing. At a high level, a timelapse is formed by a series of snapshots starting from the original graph. This enables users (and computations) to work on independent versions of the graph, without having to worry about changes to the underlying evolving graph. Timelapses can be explicitly created by users, or implicitly by the system during computations. For efficiently sharing state and computation, Tegra implements Timelapse using a persistent indexing datastructure. Tegra exposes the Timelapse abstraction to the end-users using a very simple API that supports a wide-variety of time-evolving graph analysis tasks. For ad-hoc analysis, it allows retrieval of arbitrary snapshots. For windowing operations, it provides API to access all snapshots in a particular window. Finally, to support incremental and streaming analysis, as the graph is being updated, Tegra provides APIs to enable higher level graph computational models to implement this functionality.

2. Background & Related Work

Analytics on Static Graphs: There are a large number of graph processing systems [5, 7, 10, 11, 16, 17, 25–30] that focus on iterative analytics of static graphs. Of these, some [16, 17, 27, 29] are single machine systems, while the rest support distributed processing. While many of the techniques can be applied to Tegra, they do not consider the requirements for dynamic graph processing such as incremental maintainence or ad-hoc analysis on snapshot.

---

1. For Time Evolving Graph.
2. We also support deletions, as they represent a more realistic scenario. Deletion increases the complexity of updating previous computation results and in some cases may be sub-optimal compared to restarting the computation.
Graph Store for Evolving Graphs: The problem of managing time-evolving graph has been studied in the context of graph store [2, 4, 22, 23]. Those systems focus on supporting and optimizing point queries or simple graph queries.

Managing Graph Snapshots: A lot of systems took the idea to manage snapshots for a evolving graphs, so the problem is converted to analytics on a series of static graphs. Kineograph [8] supports constructing consistent snapshots of an evolving graph. However, it does not provide retrieving earlier version. Further, it relies on replicating the updates that are common among snapshots. DeltaGraph [14] 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 [15] fixes this, but it is a specialized framework that does not provide a generalized incremental model. LLAMA [18] uses a multiversion array to support incremental ingestion. However, 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 graph analytics.

Incremental Maintenance on Evolving Graphs: Another important body of work are the streaming systems. CellIQ [13] is a specialized system for cell network analytics that shares some of the same objectives as Tegra. It does not support temporal analytics or snapshot management. GraphInc [6] supports incremental graph processing using memoization of the messages in graph parallel computation, but does not support snapshot generation or maintenance. Chronos [12] and ImmortalGraph [20] optimizes for efficient computation across snapshots. While similar in spirit to Tegra’s Timelapse abstraction, it is not a general purpose graph processing engine nor does it support streaming ingestion of updates or window operations. Differential dataflow [19, 21, 24] enables fully dynamic computations on streaming datasets, but doesn’t support ad-hoc analysis. While technically it can recreate a collection at any given time, this requires reconstruction.

3. Approach

Tegra is the first streaming graph processing system, to our best knowledge, that uniquely shares storage, computation and communication:

Sharing Storage: Tegra needs to store evolving graphs efficiently. It uses a persistent version of the adaptive radix tree to store entire snapshots without duplication (fig. 1). This data structure is designed for real-time ingestion and retrieval of arbitrary snapshots. Tegra stores graphs and intermediate state as versions, which are branches. A version can be branched at any point from any other version. A version can be a single update, or a batch of updates. This is completely up to the user. There is no one total ordering of the versions. But the system can provide ordered access if needed. Version IDs can be composite keys, and can be partially matched. Versions can be cached/uncached and can be written-to/read-from external sources (e.g., Parquet/RocksDB/Neo4j). Tegra exposes these versions to the user using a compact API.

Sharing Computation: Tegra provides generic incremental computations that accommodates edge deletions and gives the same semantics as complete re-execution of the algorithm (fig. 2). It stores intermediate computation state (e.g., iterations) using the same storage engine. When the graph is updated in the future, Tegra shares this saved computation by reusing it for parts of the graph that doesn’t need recomputation. Further, it can branch saved intermediate state.

Sharing Communication: When Tegra has access to all the snapshots in a window, it is able to drastically reduce communication among graph entities across snapshots by sharing messages exchanged. Tegra simultaneously starts computation on all the snapshots, and eliminates duplicate messages before transmitting. It further reduces the message size by leveraging delta encoding.

4. Results

Figure 3 shows the performance of Tegra’s streaming incremental computations by depicting the time taken to update the results of a connected component computation on two graphs when a fraction of the graph is updated. We see consistent performance compared to differential dataflow, whose performance suffers over time. Tegra can sustain an ingestion rate of over 1 million edge updates per second per machine. Finally, we have found that since Tegra only needs two snapshots to be in memory for streaming computations at any time, it can support millions of updates with virtually no degradation in performance.
References

