Doctoral Symposium - Raphtory: Decentralised Streaming for Temporal Graphs

Benjamin A. Steer
Queen Mary University of London
b.a.steer@qmul.ac.uk*

Felix Cuadrado
Queen Mary University of London
felix.cuadrado@qmul.ac.uk

Richard G. Clegg
Queen Mary University of London
r.clegg@qmul.ac.uk

ABSTRACT

Temporal graphs capture the relationships within data as they develop throughout time. Intuition, therefore, suggests that this model would fit naturally within a streaming architecture, where new points of comparison can be inserted directly into the graph as they arrive from the data source. However, the current state of the art has yet to join these two concepts, supporting either temporal analysis on static data or streaming into one-dimensional dynamic graphs. To solve this problem we introduce Raphtory, a temporal graph streaming platform, which maintains a full graph history whilst efficiently inserting new alterations.

ACM Reference format:
Benjamin A. Steer, Felix Cuadrado, and Richard G. Clegg. 2017. Doctoral Symposium - Raphtory: Decentralised Streaming for Temporal Graphs. In *Proceedings of DEBS '17, Barcelona, Spain, June 19-23, 2017,* 3 pages.
DOI: 10.1145/3093742.3096341

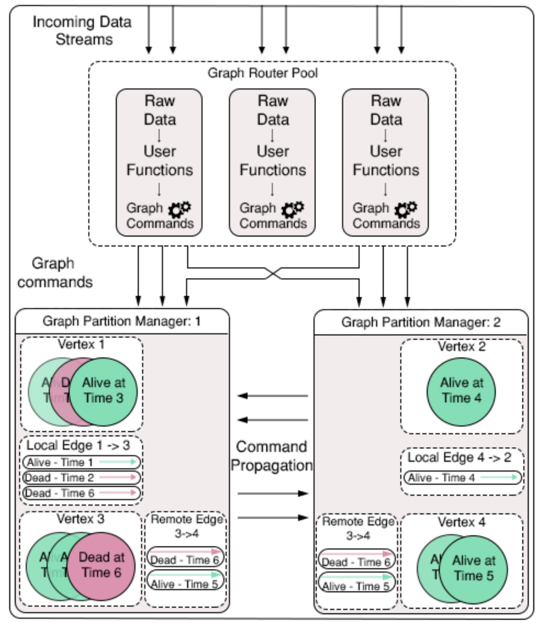

Figure 1: *Raphtory* Architecture Overview

1 INTRODUCTION

Temporal graphs provide a simple framework for exploring the evolving interconnectivity of entities within a dataset. Works such as [4] build explicitly around this structure, but do so on static data sources, failing to explore the rich temporal dimension prevalent in event-driven data streams. If such an input could be ingested it would allow new data to be compared to the full history of related entities in real time, invaluable for business use cases which require swift processing to keep the returned metrics applicable.

Real time graph streaming systems, such as [1] and [2], make use of these data sources, but their approach is to batch changes, processing static snapshots of the in-memory graph. This snapshotting reduces the granularity of temporal data to that of the snapshot window, requiring previous checkpoints to be reloaded into memory if any historical comparison is to be made. Unfortunately, batching is necessary as incoming data can frequently arrive out of sequence, especially when there are multiple information streams or points of ingestion. Deciding upon an execution order and what belongs in the next snapshot can, therefore, often require feedback between the ingestion and storage nodes, alongside some centralised arbiter. Completing this level of synchronisation for each update, or even a small batch, significantly diminishes throughput, meaning large batches are required for a system to remain viable. This tightly couples the updating process with the snapshotting cycle and means that updates may not be reflected within the graph for a long time.

To solve these problems we introduce Raphtory, a graph streaming system which maintains the full history of graph entities within chronological lists, ordered by global timestamps. This model removes the need for centralised command ordering or batched updating, as data can be inserted as soon as it arrives without affecting throughput or generating race conditions. This paper discuses the development of Raphtory's data model and ingestion architecture, followed by the proposed timeline for the processing model and other future developments.

2 ARCHITECTURE

The initial motivation behind Raphtory was to create a platform for real time analysis on temporal graphs, with an emphasis on efficient updating and high scalability. The first step towards this goal was to develop a data model and ingestion pipeline as the foundation for later processing.

Raphtory's architecture is built around the Actor Model, ingesting incoming events into a pool of *Graph Routers*. These routers convert the raw input into updates and direct them to the *Graph Partition Manager* handling the affected entity. *Graph Partition*

DEBS '17, Barcelona, Spain
 978-1-4503-5065-5/17/06…$15.00
DOI: 10.1145/3093742.3096341

Managers contain a subsection of the overall graph, maintaining the full history of the vertices and edges they control by inserting updates into the correct position as they arrive via the routing pool. If an update affects multiple partitions, it is propagated by the manager which initially received it; there is no need for centralised supervision as all operations (addition, updating or removal) are both additive and cumulative. The overview for this can be seen in Figure 1.

2.1 Graph Router

Graph Routers are actors which independently attach to an input stream. This model allows the resources allocated for ingestion to scale dynamically according to the level of incoming data. Each ingested event is converted into a graph update via user defined functions. These range from fundamentals such as defining what type of input is converted into a vertex or edge add, to more advanced concepts such as establishing sliding windows of entity decay. Maintaining a larger state for advanced execution may demand more resources per actor, but will never require data to be stored on disk.

When a command is generated, it is allocated a timestamp unique across all Routers, *Graph Partition Managers* can then use this to place the command correctly within the history of all affected entities. Currently this is created via time fields within the raw data, under the assumption that the events were originally in the correct order at the source. However, within future work it is intended to create a method for generating unique orderings when this is not present. *Graph Routers* operate a fire and forget protocol for outgoing commands, routing via a provided partitioning algorithm.

2.2 Graph Partition Manager

Graph Partition Managers maintain a subsection of the in-memory graph in the form of vertex and edge objects. These objects are contained in a key value store and include the history of the entity and its associated properties. An entity can be in one of two states at any given time; 'alive' (present in the graph) or 'dead' (absent from the graph). Adding an entity or updating its properties will insert an 'alive' state within the history at the given timestamp, whilst a removal will insert a 'dead' state. The history itself is constructed in the form of an ordered linked list, giving fast access to the most recent update (the head) and constant insertion time within the tail for delayed or out of sequence commands. This is because, unlike an array based data structure, only the objects either side of an insertion are affected and there are no random waits for memory block reallocation. Furthermore, this structure allows for fast execution of temporal graph mining algorithms, as no previous graph states have to be loaded into memory.

Adding Vertices

When adding or updating a vertex, the key value store is checked to see if an entity object exists for the given vertex ID. If it does, the alteration will be inserted into the history and the properties will be updated as required. If it does not, one will be initialised as 'alive' and placed within the store.

Adding Edges

An edge is managed primarily by the *Graph Partition Manager* storing its source node. If the destination node is also stored on this partition, the edge is considered 'local'; if it is stored in another partition the edge is considered 'remote'. For local edges the entity will be initialised or updated as 'alive', along with its adjoining source and destination vertices (so there are no hanging edges). For a remote edge, the Partition Manager will handle the source vertex and the edge entity, forwarding the command to the destination vertices *Partition Manager* to handle both this and a mirror copy of the edge. This can be seen in the edge between nodes 3 and 4 in Figure 1.

Removing Edges

The removal of an edge follows the same process as adding or updating, the difference being the new state will report the entity as 'dead' at the provided timestamp. An entity can still be initialised as 'dead' when it has yet to be 'alive' within the graph, as the command adding the edge may be delayed and can be slotted into the history when it arrives.

Removing Vertices

A vertex removal requires insertion of a 'dead' state into the vertex and all associated edges. Unfortunately, as only existing objects can be interacted with, there is the possibility here for race conditions. Commands creating relevant edges may be delayed or received after the vertex removal and, therefore, will not contain this information within their history. For example, within Figure 1, if the command which removed vertex 3 arrived before the command which added edge 3⟶4, then only edge 1⟶3 would exist when the remove is executed. 1⟶3 would, therefore, be updated with the new dead state, but 3⟶4 would miss this information, as it is created after the remove is finished.

To prevent this occurring, the 'dead' states contained in adjoining vertices must be inserted into the edges history upon creation, so if an edge misses the execution of a vertex removal the information is still present. For local edges this information can be extracted from the source and destination objects. Remote edges, however, require the *Graph Partition Manager* storing the destination vertex to return its 'dead' states via an update command after handling its half of the edge creation. Whilst this may seem a heavy bottleneck for the system, this is a one time occurrence at the initialisation of the edge. Furthermore, the response requires no locking or waiting; it can be processed asynchronously at any point in the future.

3 CONCLUSION AND FUTURE DISCUSSION

In this paper we present a model for ingesting event based data streams into a distributed temporal graph, without the need for batched updates or centralised execution ordering. As a continuation of this work we plan to investigate several avenues of interest, set out in the following road map:

Firstly, the model must be benchmarked with real data sets, such as the Twitter stream, as well as realistic graph generators, such as Forest Fire[5] and LDBC-SNB[6], to investigate how different graph distributions affect ingestion. After initial results have been gathered, the implementation will be refined with the initial goal of matching the 100,000 tweets a second ingestion established in [1].

As Raphtory does not currently delete any ingested information, we intend to follow this by developing a method of memory management, as otherwise the graphs history will eventually exceed the available memory of the containing cluster. It is intended to implement this as a background process, checking the history of graph entities on each partition and erasing alterations which occurred before a specified point in time. The erased history can then be transferred to disk, or other long term storage, from where it could be retrieved at a later date if required. The set cut off point could then be controlled by a user defined function, advance more or less frequently based on the volume of incoming data, or perhaps be managed solely by the *Graph Routers*, where entities could decay uniquely according to their type and properties. In a similar vein to this, the sliding windows of entity decay described in 2.1 could be controlled in an equivalent manner for datasets which do not naturally contain entity removal.

Once we are confident the data model and ingestion has proved to be robust and efficient, the project will move on to the development of the processing model. The intention here is to efficiently compute temporal graph algorithms on the current in-memory graph in parallel with ingestion. The proposed model will be based on the Bulk Synchronous Parallel execution often found in distributed graph systems, but instead of allowing the algorithm to converge before new updates are inserted, these will be added at the start of every superstep, similar to [7]. The algorithm will then run in perpetuum, outputting results for new data in a more timely fashion. This will have the obvious trade-off of being less accurate than a full convergence. Therefore, the next step will be to build a test bed which converges the data in each superstep to investigate how much accuracy was sacrificed, and attempt to discover which updates degraded this the most.

The final planned objective is to explore and implement techniques for adaptive graph partitioning. Naturally occurring graphs are difficult to partition due to several of their properties, notably the inclination for powers law distributions [3]. Even if a graph initially has high data locality, the flow of incoming information will cause this to slowly degrade over time, by establishing new vertices and creating relationships between those previously unconnected. To maintain high data locality, vertices will have to be swapped to the *Graph Partition Manager* with the highest number of its neighbours. This is obviously an expensive task to execute regularly, raising many questions as to how it may be completed efficiently. For example: what percentage of neighbours in another manager warrants transfer?; as this is a temporal graph, should recently connected/active neighbours have more effect on this decision?; and as both *Graph Partition Managers* and *Graph Routers* need to be aware of the location of all edges and vertices, how can they be efficiently informed of transfers, and how should they store this information.

REFERENCES

[1] R. Cheng et al. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings Eurosys*, pages 85–98. ACM, 2012.

[2] A. Dubey et al. Weaver: a high-performance, transactional graph database based on refinable timestamps. *Proceedings of the VLDB Endowment*, 9(11):852–863, 2016.

[3] J. E. Gonzalez et al. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

[4] Y. Miao et al. Immortalgraph: A system for storage and analysis of temporal graphs. *Transactions on Storage (TOS)*, 11(3):14, 2015.

[5] Snap-Stanford. Forestfire graph generator, 2017. Available at https://github.com/snap-stanford/snap/tree/master/examples/forestfire.

[6] The Linked Data Benchmark Council. Ldbc social network benchmark, 2017. Available at http://ldbcouncil.org/benchmarks/snb.

[7] L. M. Vaquero et al. Adaptive partitioning for large-scale dynamic graphs. In *ICDCS 2014*, pages 144–153. IEEE, 2014.