Raphtory: Building Distributed Temporal Graphs From Event Streams

Anonymous Author(s)

ABSTRACT

Temporal graphs capture the development of relationships within data throughout time. This model would fit naturally within a streaming architecture, where new events can be inserted directly into the graph as they arrive from the data source, being compared to related entities or historical state. However, the vast majority of graph processing systems only consider traditional graph analysis on static data, with some outliers supporting either temporal analysis on similarly static data or providing traditional analysis on graphs updated via event streams. In this work we define a temporal graph model for stream updating and discuss the challenges of distribution and graph management. To solve these challenges, we introduce *Raphtory*, a distributed temporal graph platform which maintains the full graph history in-memory, leveraging this to insert streamed events directly into the graph model without batching and with minimal synchronisation.

KEYWORDS

Temporal Graph, Dynamic Graph, Distributed Systems, Event Streams

1 INTRODUCTION

Temporal graphs chronicle the changes in graph state throughout time, unlocking a breadth of possibilities for analysis. These range from expanding upon traditional graph algorithms, such as providing congestion aware GPS navigation via temporal shortest path [14]. To establishing a basis for obtaining novel insights, for example investigating the long term structural changes in cryptocurrency transaction graphs [6].

Current distributed graph systems, however, focus heavily on traditional graph processing e.g. PowerGraph [7], GraphLab [10] and GraphFrames [3]. The temporal graph systems which do focus on these problems complete their execution in an offline batched fashion, normally comparing differences across pairs of graph snapshots e.g. ImmortalGraph[11], Version Traveler [9] and GraphTau[8]. This is flawed for two reasons. Firstly, snapshotting reduces the granularity of temporal data to that of the snapshot window, meaning important updates may be lost. Secondly, because temporal graphs fit naturally within an online/streaming environment where new events can be compared to related entities or historical state.

Kineograph[2] and Weaver[4] come close to this by streaming updates into a mutating in-memory graph model, alongside ongoing analysis. However, their approach still batches changes and analysis is performed on snapshots of the in-memory graph. Only recently have works such as Chronograph [5] began to rethink this architecture, streaming events directly into an in-memory temporal graph.

In this paper we introduce such a temporal graph model, replacing coarse snapshots with fine-grained vertex/edge histories containing all changes to the graph structure and property values. Once defined, we discuss what consists a valid update to this model, establishing protocols for adding, removing and updating graph entities. We explore the challenges of distributing the model across a set of machines, partitioning for high data locality and inserting updates from parallel data sources.

To solve these challenges, we present *Raphtory*, a distributed platform implementing the defined model. *Raphtory* splits the graph over a set of *Partition Managers* which leverage the in-memory history to immediately insert updates without affecting throughput or generating race conditions. Thus, new data may be compared to the full history of related entities in near real-time. Finally, we perform preliminary evaluation of Raphtory, investigating scalability of throughput when increasing the number of *Partition Mangers* and data ingesting *Graph Routers*.

2 TEMPORAL GRAPH MODEL

A temporal graph G consists of a pair $G{=}\langle V, E\rangle$ which chronicle all transpiring changes to the graphs member vertices and edges. V is the set of all unique vertices $V{=}\{v_1, v_2, ..., v_n\}$ which have existed within the graph and E is the set of all unique edges $E{=}\{\langle v_1, v_2\rangle, \langle v_2, v_3\rangle, ..., \langle v_i, v_j\rangle\}$. An edge in this model is defined as an ordered pair of vertices $\langle v_i, v_j\rangle$, depicting directed relationships between vertices in V; thus $\langle v_i, v_j\rangle$ is distinct from $\langle v_j, v_i\rangle$. E may contain self-looping edges where the source and destination are the same ($\langle v_j, v_j\rangle$), but E cannot contain multiple edges with the same source and destination ($\{\langle v_j, v_k\rangle, \langle v_j, v_k\rangle\}$).

Each vertex in V and edge within E is assigned a chronological history $H{=}\big\{\langle t_1, created\rangle, \langle t_2, deleted\rangle, ..., \langle t_n, created\rangle\big\}$ which contains all changes to the state of that entity (either *created* or *deleted*) paired with a timestamp of when the change occurred (illustrated via logical timestamps e.g. $t_1, t_2, ..., t_n$). Thus, each edge and vertex exists for a set time range, or several time ranges if removed and re-added. Each entity (vertex or edge) is additionally assigned a set of properties P, where a property is defined as a key and a value history, specifying all previous values associated with the key and the time at which the change occurred. For example, if an entity has been assigned a property key_i then $P(key_i){=}\big\langle key_i, \{\langle t_1, value_1\rangle, \langle t_2, value_2\rangle, ..., \langle t_n, value_n\rangle\}\big\rangle$. These structural and property histories can then be combined to create the overall history of the graph.

To view how an equivalent non-temporal graph would appear at a given point in time t, a graph snapshot $G(t)$ may be generated. $G(t)$ is defined as a pair $G(t){=}\langle V(t), E(t)\rangle$ where $V(t)$ and $E(t)$ represent the vertices and edges respectively present in G at time t. For an entity to be considered present, it must be a member of its equivalent

GRADES-NDA '18, June 10, 2018, Houston, USA

ACM ISBN 000-0000-00-000/00/00…$15.00
https://doi.org/doi

set (V or E) and its history must denote a *creation* at time t or at the closest update prior to t. Present entities will then contain a property set $P(t)=\{\langle key_1, value_1\rangle, \langle key_2, value_2\rangle \ldots \langle key_n, value_n\rangle\}$ comprising the values of all associated properties at the closest update anterior to t. A snapshot such as this draws many parallels with the property graph model[12], but lacks type constraints.

2.1 Graph Updates

Updates to this temporal graph model fall into three categories: *Entity Addition* - creation of a vertex or edge; *Entity Removal* - deletion of a vertex or edge; *Entity Update* - appending values for entity properties. For an update to be considered valid and accepted into the graph it must first pass the preconditions set for that update type. For example, a removal can only be considered valid if the entity in question is present within the graph, as it cannot be deleted if it does not currently exist.

Entity Addition. For the addition of a given vertex v_i, it must first be checked if $v_i \in V$. If v_i has never been a member of the graph, V is updated to include this new member $V'=V \cup v_i$; a history is then assigned to v_i specifying the time of its creation $H(v_i)=\{\langle t_i, created\rangle\}$. If $v_i \in V$, its history must be inspected to see if the latest state denotes a *removal*. Given this is the case, the update is considered valid and the history is appended to specify that v_i is now present once again $H(v_i)'=H(v_i) \cup \langle t_i, created\rangle$. If v_i was not previously removed and is currently present within the graph, the update is considered invalid and is abandoned.

Edge addition necessitates similar prerequisites, firstly requiring both the source v_i and destination v_j of an edge $\langle v_i, v_j\rangle$ to be present within V; both are required for a valid update to avoid hanging edges (an edge with only a source or destination). If both are present within the graph, it is checked if $\langle v_i, v_j\rangle \in E$, dictating if the edge requires insertion into E or if its history requires appending, in the same fashion as described for vertex addition.

Furthermore, when adding any entity which has never been a member of G, the property keys it contains must be established; known as the property key-set $P_{ks}=\{key_1, key_2, \ldots, key_n\}$. Each key in P_{ks} must be assigned an initial value to be placed into the history alongside the time of entity initialisation (t_i), thus creating the property set $P=\{\langle key_1, \{\langle t_i, InitialVal_1\rangle\}\rangle, \langle key_2, \{\langle t_i, InitialVal_2\rangle\}\rangle, \ldots, \langle key_n, \{\langle t_i, InitialVal_n\rangle\}\rangle\}$.

Entity Removal. For an edge to be eligible for removal it must first be present within the graph. If so, no information is actually deleted, instead its history is appended with a *remove* state at the time at which the update occurred. For example, for an edge which is removed at time i, its history would be updated as follows: $H' = H \cup \langle t_i, deleted\rangle$. Vertex removal is executed in the same manner, but requires an additional step to remove all present edges within E with the vertex as a source or destination, as these are now considered hanging edges. This is completed by appending their history with a *remove* state at the time of vertex removal.

Entity Update. The process to update a property value is exactly the same for both vertices and edges, firstly checking to see if the entity is present within G, followed by confirming the property key as a member of the entity key-set P_{ks}. For a given key value pair $\langle key_i, value_i\rangle$, if $key_i \in P_{ks}$, then the new value is appended into the property history along with the time the update occurred: $P(key_i)'=P(key_i) \cup \langle t_n, value_i\rangle$. If the entity does not have a property with this key, the update is considered invalid and is discarded.

2.2 Challenges With Distributing The Model

Running this model in a distributed environment poses several challenges for an implementation to be correct. The first issue is how to partition the graph, deciding what consists a partition and the strategy for splitting the overall graph whilst retaining high data locality (e.g. edge-cut or vertex-cut, as described in PowerGraph[7]). However, unlike PowerGraph, partitioning a temporal graph has the additional complexity of managing tradeoffs between structural locality (proximity to neighbours) and temporal locality (proximity to an entities history) [11]. Furthermore, establishing a viable partitioning strategy for a graph built from a stream of updates is difficult as it cannot be prepartitioned and, if not actively managed, data locality will slowly degrade as more entities are added[13].

Secondly, within a distributed setting, updates are not ingested via a serial stream and may need to be sent to multiple partitions, leading to commands arriving out of order; this must be handled to ensure updates are not processed incorrectly or dropped unnecessarily. For example, if an edge add were to arrive before the addition of its source vertex, the update would be incorrectly abandoned.

Following this, a distributed graph will inadvertently have entities spanning multiple partitions. It must, therefore, be decided how to manage/propagate updates affecting such entities. For example, if an edge-cut partitioning strategy were utilised, edges with the source and destination on different machines would require synchronisation whenever an update to the state or properties occurred. Furthermore, the effect of this requirement is multiplied for the removal of vertices which could potentially have millions of edges spanning the entire cluster, all of which would require notification of the removal.

Finally, as no updates are removed from memory, even with a large cluster of machines, eventually the memory limitations would be reached. A protocol must, therefore, be established to govern what data is retained in memory and what is offset onto more permanent offline storage.

3 RAPHTORY

To address these challenges we introduce Raphtory, a system which maintains temporal graphs over a distributed set of partitions, ingesting and processing parallel updates in real time. Raphtory's architecture is based on the actor model[1], with the core actor types consisting of *Graph Routers* and *Graph Partition Managers*. *Graph Routers* attach to a given input stream and convert raw data into one of the update types established in Section 2.1, forwarding this to the *Graph Partition Manager* handling the affected entity. *Graph Partition Managers* contain a partition of the overall graph, split in an edge-cut fashion. *Managers* maintain the vertices and edges they control by inserting out-of-order updates into the correct chronological position within the history of the entity as they arrive via the pool of *Graph Routers*. This removes the need for centralised synchronisation, as commands may be executed in any given arrival order whilst resulting in the same history. An overview of this can be seen in Figure 1.

Raphtory allows the graph to be partitioned by any given strategy. For initial testing this is a simple hash partition as it requires no

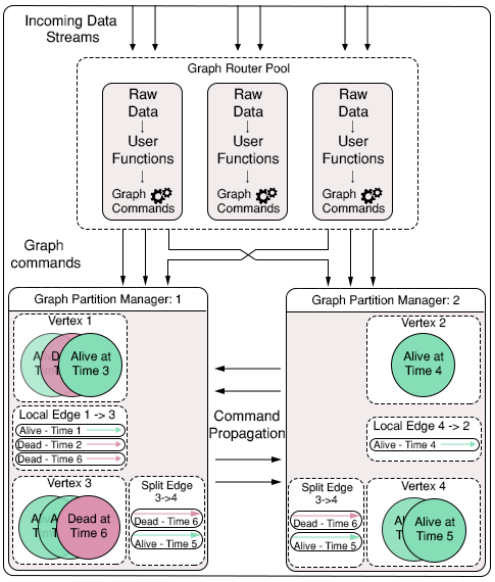

Figure 1: *Raphtory* Architecture Overview

state and can scale along with the number of *Routers* and *Partition Managers*. To deal with memory constraints the user may set a threshold for memory usage for each *Partition Manager*. When this threshold is met, the *Manager* will establish a cutoff point where all updates prior to this time will be transferred to offline storage. This will begin at the time of the oldest update and move forward through time at a speed proportional to the number of incoming messages. This additionally has the benefit of acting as a snapshotting mechanism to allow partitions to be recovered if the *Manager* was to crash. However, this is yet to be fully implemented and is set as future work.

3.1 Graph Router

Graph Routers are the point of ingestion for the new data from which graph updates are derived. Each *Router* upon instantiation is attached to an input stream and converts incoming events into graph updates via user defined functions. These functions may be as complex as the user requires, ranging from fundamentals, such as what type of input is converted into a vertex or edge add, to facilitating more advanced concepts such as sliding windows or entity decay. Maintaining a larger state for advanced execution such as this 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. This timestamp 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* process events fully independently, operating a fire and forget protocol for outgoing commands, routing via the established partitioning algorithm. This allows the resources allocated for ingestion to scale dynamically according to the level of incoming data by adding or removing *Routers* from the pool as required.

3.2 Graph Partition Manager

Graph Partition Managers maintain a sub-section of the in-memory graph in the form of vertex and edge objects. These objects are organised into hashmaps (one for vertices, one for edges) and contain the entity meta data (such as its ID), its structural history and a map of its associated properties. Entity history is constructed in the form of an ordered linked list, giving fast access to the most recent update (the head) and efficient insertion time within the tail for delayed or out of sequence commands, as only the elements either side of an insertion are affected. Property objects within the associated properties map mirror this structure, containing the property key and their own linked list based history of previous values.

Adding Vertices. When adding a vertex, the vertex map is checked to see if an object exists for the given vertex ID. If it does, the objects history will be updated with a *Created* state and its properties will be updated with the new initial values. Note, it is not checked if the vertex is already present within the graph as a remove command may have been delayed, requiring the *Created* state to be present when it arrives. If no object within the map represents the given ID, one is instantiated, beginning the entity history and establishing the property map from the given key-set and initial values.

Adding Edges. An edge is managed primarily by the *Graph Partition Manager* storing its source vertex. If the destination vertex is also stored on this partition, the edge is considered 'local'; if it is stored in another partition the edge is considered 'split'. When adding an edge it is initialised or updated in the same fashion as a vertex. However, receipt of an edge add will also generate vertex add commands for its adjoining source and destination, avoiding possible hanging edges. These will establish placeholder vertices if the 'real' commands are yet to arrive or be ignored if they are already present. For a split edge, the *Partition Manager* will propagate the commands to the destination vertices *Manager*, requesting it to handle both this and a mirror copy of the edge. This can be seen in the edge between vertices 3 and 4 in Figure 1.

Removing Edges. When removing an edge present within the edge map, the representing objects history will be updated with a *Remove* state, again not checking if the edge is already absent as the *Created* command may arrive later. If the map does not contain an object representing this edge a placeholder entity is initialised, beginning the history with the given *Remove* state. The delayed edge add may then be slotted into the history when it arrives, as well as establishing the edges property map. If the edge is split, the remove command will be propagated to the *Partition Manager* handling the destination node, as described in *edge add.*

Removing Vertices. A vertex removal requires insertion of a *Removal* state into the history of 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 *Removal* state, but 3⟶4 would miss this information.

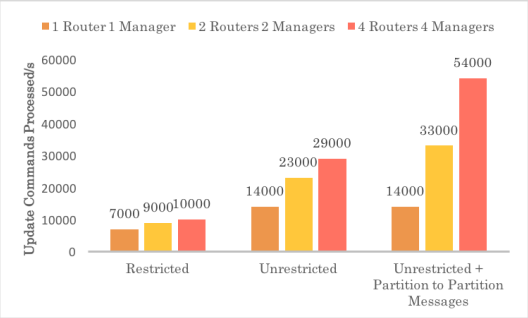

Figure 2: Initial evaluation of the maximum update throughput of *Partition Managers* in various sized clusters.

To prevent this occurring, the *Removals* contained in adjoining vertices must be inserted into the edges history upon creation. This way, if an edge misses the execution of a vertex removal the information is still present. For local edges, the *Removals* can be extracted from the source and destination objects. Split edges, however, require the *Graph Partition Managers* storing the source and destination vertices to exchange information on *Remove* states. 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.

Updating properties. Properties may be updated individually or as a group. If an object representing the affected entity exists within the respective map, the new values will be placed into the history of the property object allocated for each affected key. If this update is for a split edge, it will be propagated as with addition or removal. If a property update arrives before the creation of the affected entity then a placeholder object must be utilised until it arrives.

4 EVALUATION

Raphtory actors are containerised via Docker[1] and execute independently, communicating via the Akka messaging framework [2]. To evaluate this implementation, several secondary actor types were created, providing update generation, benchmarking and live graph analysis; these can be plugged into an established cluster without disturbing ingestion or graph maintenance. Utilising these, a stream of graph updates were generated and ingested into clusters of varying size. Gradually increasing the throughput until the maximum amount of commands each *Partition Manager* could consistently process was reached.

Three different sized clusters were tested, beginning with $Cluster_1$, a 'singleton' implementation with one *Graph Router* and one *Partition Manager*, and then doubling the amount of components each time; two of each ($Cluster_2$), followed by four of each ($Cluster_4$). All clusters were instantiated across two servers, both containing an Intel Xeon E5645 CPU (24 cores @2.4GHz) and 64GB of RAM, with the *Routers* and *Partition Managers* evenly split between them. The updates were then streamed in from a third machine to simulate an external event source. These events were split into 30% vertex adds, 40% edge adds, 10% vertex removals and 20% edge removals over a pool of one million unique vertex ID's. Figure 2 displays the average result for each cluster over five iterations, split into three groups: 'Restricted' where each component was limited to at most one CPU core; 'Unrestricted' where containers could utilise all server resources; and 'Unrestricted + Partition to Partition Messages' which combined the total unique commands (messages from *Graph Routers*) with a count of synchronisation commands sent between *Partition Managers* (messages used to manage split edges, as described in section 3.2).

When run in unrestricted mode, $Cluster_1$ averaged a maximum throughput of 14,000 msg/sec, increasing by 64% to 23,000 in $Cluster_2$. A similar increase, however, was not seen when changing to $Cluster_4$ (only 26% to 29,000). This appears to be for two reasons. Firstly, Akka heavily utilises threading to manage the message queues of each actor, for example, one *Router* in $Cluster_2$ peaked at 1700% CPU utilisation. In this instance, this is promising as the four components appear to make efficient use of the 48 cores available. However, for $Cluster_4$, the extra parallelism within actor tasks is diminished by the frequent conflict for a greater share of the available resources. This demand for resources is further demonstrated by the restricted runs where the average throughput was much lower and doubling the amount of components had little effect.

Secondly, $Cluster_4$ divides the graph into four partitions, increasing the number of split edges and, therefore, the amount of synchronisation between *Partition Managers*. This is demonstrated within Figure 2, where almost 50% of messages within $Cluster_4$ were inter-manager communications, compared to 30% in $Cluster_2$. This is, however, not a permanent problem and could be improved by better partitioning strategies as discussed in Section 5.

Overall we believe this preliminary testing demonstrates Raphtory is able to efficiently ingest and store graph events, and scale with the provided resources. However, more testing is clearly required as the project develops, with a larger array of machines utilised.

5 CONCLUSION

In this paper we tackle the lack of graph processing systems which perform online temporal analysis whilst ingesting new data from event streams. To address this, we firstly introduce a temporal graph model and define what may be changed in terms of graph structure and meta data. We discuss the challenges of distributing this model and provide our solution, *Raphtory*, which manages the graph history via a set of *Partition Managers*, ingesting new events into the graph via a pool of *Graph Routers*. Our preliminary experimentation shows that Raphtory scales well when provided with abundant resources, but further testing is required to ascertain performance on a larger cluster of servers.

As a continuation to this work, we first plan to implement the snapshotting/data offsetting described in Section 3, allowing for longer running tests and development of fault tolerance. Following this we intend to replace the current hash partitioning with the adaptive partition strategy described in [13], where vertices may decide to migrate to a partition with a higher number of neighbours to minimise edges spanning machines. Finally, we aim to develop 'live graph analysis' actors which will probe the live graph in parallel with updates, providing fast approximate results. These will be paired with 'snapshot analysis' actors performing the same algorithm offline to confirm the initial approximation.

[1] https://www.docker.com/
[2] https://akka.io/

REFERENCES

[1] Gul A Agha. 1985. *Actors: A model of concurrent computation in distributed systems.* Technical Report. MIT Cambridge Artificial Intelligence Lab.

[2] Raymond Cheng et al. 2012. Kineograph: taking the pulse of a fast-changing and connected world. In *ACM EuroSys*. 85–98.

[3] Ankur Dave et al. 2016. Graphframes: an integrated API for mixing graph and relational queries. In *ACM GRADES*.

[4] Ayush Dubey et al. 2016. Weaver: a high-performance, transactional graph database based on refinable timestamps. *VLDB Endowment* 9, 11, 852–863.

[5] Benjamin Erb et al. 2017. Chronograph: A Distributed Processing Platform for Online and Batch Computations on Event-sourced Graphs. In *ACM DEBS*. 78–87.

[6] Erwin Filtz et al. 2017. Evolution of the Bitcoin Address Graph. In *Springer Data Science–Analytics and Applications*. 77–82.

[7] Joseph E Gonzalez et al. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, Vol. 12.

[8] Anand Padmanabha Iyer et al. 2016. Time-evolving graph processing at scale. In *ACM GRADES*.

[9] Xiaoen Ju et al. 2016. Version Traveler: Fast and Memory-Efficient Version Switching in Graph Processing Systems. In *USENIX Annual Technical Conference*. 523–536.

[10] Yucheng Low et al. 2014. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*.

[11] Youshan Miao et al. 2015. Immortalgraph: A system for storage and analysis of temporal graphs. *ACM TOS* 11, 3.

[12] Neo4J. 2017. What is a Property Graph. Available at https://neo4j.com/developer/graph-database/#property-graph. (2017). Accessed: 15-01-2018.

[13] Luis M Vaquero et al. 2014. Adaptive partitioning for large-scale dynamic graphs. In *IEEE ICDCS*. 144–153.

[14] Huanhuan Wu et al. 2014. Path problems in temporal graphs. *VLDB Endowment* 7, 9, 721–732.