Raphtory : Streaming Analysis Of Distributed Temporal Graphs

Ben Steer, Felix Cuadrado, Richard Clegg

Queen Mary University of London

Abstract

Temporal graphs capture the development of relationships within data throughout time. This model fits naturally within a streaming architecture, where new events can be inserted directly into the graph upon arrival from a data source and be 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 batched updating and temporal analysis across graph snapshots. In this work we define a temporal graph model which can be updated via event streams and discuss the challenges of distribution and graph maintenance. To solve these challenges, we introduce *Raphtory*, a distributed temporal graph management system which maintains the full graph history in memory, leveraging this to insert streamed events directly into the model without batching or centralised ordering. Raphtory additionally provides an API to perform both approximative analysis on the most up-to-date version of the graph, as well as temporal analysis throughout its full history; executed in parallel with ingestion.

Keywords: Temporal Graph, Distributed Computing, Stream Processing, Temporal Analysis

1. Introduction

Graphs are a powerful abstraction that can represent complex interconnectivity between entities within data, as well as model a variety of theoretical and practical problems. Graphs have applications within a multitude of domains, notably finance, epidemiology, telecommunications, social network analysis and more. By superseding the base graph model with that of a temporal graph[1], we may additionally capture the evolving interconnectivity of entities within the underlying dataset over time; a noted desire of many graph practitioners[2]. This unlocks a breadth of analytical possibilities by expanding on standard graph algorithms, such as providing congestion aware GPS navigation via temporal shortest path[3]. Furthermore, this model fits naturally with real-time graph analysis and algorithms (e.g. within a streaming architecture), so that new events are inserted into the graph upon arrival and are compared to related entities or historical state. For instance, the e-commerce site Alibaba ingests all new sales into a graph of the previous weeks transactions to monitor for fraud[4].

Scalability is the main barrier faced by graph practitioners and researchers [5], as graph algorithms are very intensive in memory and computation resources. There has been substantial development in distributed graph processing systems that can scale vertically in order to enable large-scale graph analysis. Current systems focus primarily on static graph analysis (e.g Pregel[6], GraphLab[7] & PowerGraph[8]). However, in many business sectors, such as the example above, graph snapshots soon lose relevance, requiring a fresh snapshot with new updates to be created and processed quickly to obtain relevant results. Newer systems have begun to buck this trend, ingesting streams of data whilst maintaining an in-memory graph model (e.g. Kineograph [9] and Weaver [10]). These, however, batch changes into coarse snapshots and fail to explore the rich temporal dimension prevalent in their event-driven data streams.

In this paper we introduce Raphtory, a distributed system that enables scalable temporal graph analysis from live data streams. Raphtory is based on a temporal graph model which captures the history of changes to each graph vertex and edge, rather than full snapshots. The system can ingest multiple streams of events, creating and updating a distributed temporal graph from this information. We describe in the paper how Raphtory addresses the system challenges to maintain the temporal graph in-memory. The graph is split into a set of partitions; each partition within Raphtory manages the full history of the vertices and edges it is assigned, inserting any new updates into the correct chronological position without centralised synchronisation. The system manages the ever growing history of the graph by compressing older changes and moving them to long term storage until they are requested for later analysis. Graph analysis can be performed on the latest version of the graph or at any point in its history, with Raphtory dynamically retrieving offset data as required in a manner which is transparent from the users perspective.

The rest of this paper is organised as follows: Section 2 analyses established graph processing systems and attempts in other disciplines to define a temporal graph; Section 3 presents our temporal graph model and the semantics for ingesting updates via a stream; Section 4 discusses the challenges of distributing this model, from both a maintenance and processing perspective; Section 5 describes the Raphtory model in full and how these challenges have been addressed; Section 6 contains the preliminary evaluation of Raphtory, investigating scalability of throughput for different workloads when increasing the number of partitions, followed with an extended evaluation plan and several proposed use cases; finally, Section 7 concludes and discusses the future goals of the Raphtory project.

2. Related Work

2.1. Graph Processing Systems

Traditionally, distributed graph processing systems have been based primarily on Pregel's[6] 'think like a vertex' paradigm. This is a bulk synchronous parallel[11] programming model in which each vertex is viewed as an autonomous entity, storing information about itself and its outgoing edges, executing in iterative batches known as supersteps. Vertices may not access each others data, but may send messages/updates which are queued on the receiving vertex and processed in the next iteration, allowing vertices to execute independently and easing distribution. Several variations of this paradigm have also been proposed, notably, PowerGraph's[8] Gather Apply Scatter (GAS) model and edge-centric computing, moving the function off the vertex, allowing high degree nodes to be partitioned. Alternatively, GraphLabs[7] 'datagraph' which replaces superstep restrictions with a dynamic scheduler and stringent consistency constraints, allowing multiple vertex functions to run asynchronously and access each others data. As well as existing independently, these programming models have found prominence as extensions to general purpose distributed processing frameworks, namely GraphX[12] for Apache Spark[13] and Gelly for Apache Flink[14]. However, all of these systems focus on processing of static data, making no concessions for updating the graph without rebuilding or enabling temporal analysis.

Extending from these, Kineograph[9] maintains an in-memory graph, ingesting a stream of updates via a set of 'ingestion nodes' which forward the update to 'graph nodes' storing affected entities. These updates are recorded within a global progress table which periodically requests a graph snapshot to be generated, which can then be analysed offline. This works effectively as micro batching, allowing the user to strike a balance between reasonable timeliness and higher throughput. However, as update acknowledgement requires feedback between the ingestion and storage nodes, alongside a global singleton, large batches are required for the system to remain viable. To minimise this, update order is based on processing time, ignoring event time and meaning the snapshots may differ drastically if the stream were re-ingested. Additionally, snapshotting reduces the granularity of temporal data to that of the snapshot window, meaning the order of updates within this period are lost. Weaver [10] attempts to improve on this architecture by generating ‘happens before’ relations between all updates, utilising coarse ingestion timestamps where possible and refining these via a centralised ordering oracle when a clash occurs. However, whilst the oracle is meant to be lightweight when intervening, it has been shown to bottleneck the system in practice[15].

ImmortalGraph [16] looks at how to efficiently store a graph’s history on disk, presenting ‘snapshot groups’ and the Locality-Aware Batch Scheduling (LABS). Snapshot groups maintain a full snapshot for the start of an assigned time-range and an update log containing all changes up until the end. This allows the graph to be queried at any given point in its history, whilst never exceeding a user set threshold for graph construction. These are replicated on disk to provide both high spacial-locality (placing neighbours within the same snapshot in close proximity) and high temporal-locality (all versions of a vertex stored consecutively), with the LABS engine picking the most appropriate given the desired use case. Alternatively, Version Traveller [17] provides ‘arbitrary local version switching’ between snapshots of a graph, computing the next snapshot from the current by combining it with a delta representing the difference between the two. Deltas are typically much smaller than the full graph and can often be cached in memory, incurring significantly less overhead than loading from disk. Both of these systems, however, are implemented with an offline perspective, executing on static data repositories.

Chronograph [18] combines components of all these systems, providing a dynamic graph model which allows concurrent modifications via an unbound stream of updates. This is enabled by instantiating vertices as actors[19] and performing local event sourcing, where each vertex logs its changes and that of its outgoing edges. A global log is then only required to maintain vertex creation and deletion order, minimising bottlenecks. Various computation models may then execute on top of this, performing online approximations on the live dynamic graph and offline batch processing on consistent snapshots. However, whilst this permits temporal analysis via snapshot comparison, the history is not maintained in memory and snapshots must be recomputed from the stored logs.

2.2. Temporal Networks

As established distributed graph processing systems focus primarily on snapshot comparison for temporal analysis, it appeared appropriate to investigate how temporal graphs have been proposed and formalised, as well as understand the manner in which temporal information is traditionally stored. Having many mutli-disciplinary applications, a range of models are available in the literature under various pseudonyms such as temporal networks [1] and evolving graphs [20]. These are, therefore, explored to discuss all desirable characteristics, noting possible expansions.

In [21] the authors present a directed graph where each edge is labelled to show the initial time of connection between the source and destination vertices. These labels may then be used to find ‘time-respecting paths’ throughout the graph where a hop can only proceed if the time on the next edge is non-decreasing. This assumes that travel along an edge is instantaneous and overlooks the possibility of edge deletion. Graphs in [22] instead label edges with a range of time to include this feature. This could, however, be expanded further to allow multiple ranges if edges are to be re-added. Additionally, whilst not included in the model, vertex deletion is discussed within the context of connectivity problems/reachability. Another possible expansion would, therefore, be to track the time at which vertices are added and removed from the graph to encompass such analysis.

Instead of the explicit time range described in [22], Time-Varying Graphs implemented within [23] provide an edge presence function over a defined graph lifetime, returning true if the edge is present within the graph at a specific point, or false if it is absent, enabling multiple ranges. This work also introduces mutli-value labels, where each vertex and edge may have a set of properties associated with it. These properties allow several edges to be set between a pair of vertices, as long as they have different property values, creating a temporal multi-graph. Unfortunately, because of this, these values must remain constant throughout the lifetime of the graph. A clear area for improvement is, therefore, to allow label values to change throughout time, storing all previous versions. In a similar vein to the edge presence function, [24] provides an approximate view of the temporal graph at a time t, over a set of time labelled edge triples. This is defined as $G(t) = V(t), E(t)$, where $E(t)$ is the set of edges created prior to time t and $V(t)$ the unique vertices they connect.

3. Temporal Graph Model

The first step to providing a system which maintains an updatable, in-memory, temporal graph is to formalise the model. In this section we present such a model, expanding on those seen in Section 2 to provide a full history of structural and property changes within the graph, as well as functions to view the graph at any given point in its lifetime. We additionally define the semantics for modifying graph state via a stream of updates, encompassing the addition/removal of vertices and edges as well as updating their associated properties.

3.1. Proposed Model

A static graph G consists of a pair $G{=}\langle V, E\rangle$ where V is the set of all vertices $V{=}\{v_1, v_2, \dots, v_n\}$ and E is the set of all edges $E{=}\{\langle v_i, v_j\rangle, \langle v_k, v_l\rangle, \dots, \langle v_m, v_n\rangle\}$. 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 $\langle v_i, v_j\rangle \neq \langle v_j, v_i\rangle$. E may contain looping edges where the source and destination are the same ($\langle v_i, v_i\rangle$), but E cannot contain multiple edges with the same source and destination ($\{\langle v_j, v_k\rangle, \langle v_j, v_k\rangle\}$). We refer to both vertices and edges as graph entities $Y = V \cup E$. To store meta data for each vertex and edge we define a set of m keys $K = \{k_1, k_2, \dots, k_m\}$ and define properties of entities using key value pairs (where if not set, the value is null). We define the property for key k_i on entity y as $p_i^y = value_i$ or $p_i^y = \emptyset$ if no value is set.

Moving into a dynamic setting where the graph is no longer static and will be updated over time, the graph $G{=}\langle V, E\rangle$ would instead be defined as *G(t)*$=\langle V(t), E(t)\rangle$; where t is a specific point within the graphs lifetime; $t_0 <= t <= t_n$. This begins with the time of initialisation (t_0) where $V(t_0) = \emptyset$ and $E(t_0) = \emptyset$, and ends at t_n which denotes the time of the most recent change. $G(t_0)$ is, therefore, the earliest version of the graph and $G(t_n)$ the most up-to-date graph, referred to as the 'live graph'. Within this range n updates will have been applied, each at a unique time $t_1, t_2, \dots, t_n$. Whilst t may equal any of these, it is not limited to their discrete values and may specify a time in-between them. In this instance, $G(t)$ will be exactly the graph $G(t_i)$, where t_i is the largest value within the set of update times such that $t_i <= t$; i.e. $G(t)$ is the graph seen at the most recent change just before time t. Within $G(t)$, $V(t)$ contains all vertices within the graph at time t and $E(t)$ all the edges. Furthermore, for key $k_i \in K$ then $p_i^y(t)$ will return the associated value for entity y at time t (if one exists).

A temporal graph, therefore, encompasses all observed graphs $G(t)$ from t_0 (the initial graph) to t_n (the most recently observed graph). It is useful to define V_T, the set of all unique vertices which have existed within the graph $V_T = V(t_0) \cup V(t_1) \ldots \cup V(t_n)$, E_T, the set of all unique edges $E_T = E(t_0) \cup E(t_1) \cup \ldots \cup E(t_n)$ and $G_T = \langle V_T, E_T \rangle$. To record the times at which entities have joined or left the graph each vertex and edge is assigned a history $H^y = \big\{\langle t_i, \mathit{created}\rangle, \langle t_j, \mathit{deleted}\rangle, \ldots, \langle t_l, \mathit{created}\rangle\big\}$, where each modification to the state of an entity is represented as a pair containing the new state (either *created* or *deleted*) alongside a timestamp of when the change occurred, allowing for chronological ordering. As with the dynamic graph, the state of an entity at a given time t is the same as the nearest change point before t, that is $\langle t_m, \mathit{state}_m\rangle$ for the largest value of m such that $t_m \leq t$, e.g. in the above example, $H^y(t) = \mathit{created}$ if $t_i \leq t < t_j$. Therefore, an entity is considered present or absent for a set time range, or several time ranges if removed and re-added. H^y includes all of these ranges, but a subset of the history may also be garnered by specifying a start and end point of interest, e.g. $H^y(t, t')$ where $t_0 \leq t < t' \leq t_n$.

In addition to the structural history stored in H, for an entity y and a key k_i the property p_i^y now contains a value history $p_i^y = \{\langle t_j, \mathit{value}_j\rangle, \langle t_k, \mathit{value}_k\rangle, \ldots, \langle t_l, \mathit{value}_l\rangle\}$, specifying the sequence of values associated with the key and the time at which the change occurred. If the property for k_i has never been set for y then $p_i^y = \emptyset$. As with the changes in entity state, the value for a property at a given time t is equal to the closest update anterior to t, e.g. $p_i^y(t) = \mathit{value}_j$ if $t_j \leq t < t_k$. These structural and property histories can then be combined to create the overall history of the graph. Thus the temporal graph can provide a view of the data at any chosen point in time t, unlike many previous systems which can only provide snapshots at coarse intervals. This is done by recreating the equivalent graph view $G(t) = \langle V(t), E(t)\rangle$ where $V(t)$ and $E(t)$ contain all vertices and edges present at time t. Note, as the combination of source and destination are an edges unique identifier, there is no way to disambiguate between recreating an edge and inserting a new edge between the same two vertices (creating a multi-graph). If the latter was desired this would have to be managed via associated edge properties.

3.2. Update Semantics

Updates to this temporal graph come in the form of an unbound stream of events $\{\langle t_1, a_1\rangle, \langle t_2, a_2\rangle, \ldots, \langle t_n, a_n\rangle\}$, where each event depicts an action (see Table 1) and the time of its occurrence. Actions fall into three categories: *Entity Addition* - creation of a vertex or edge; *Entity Removal* - deletion of a vertex or edge; *Entity Update* - changing the value of entity properties. By applying all updates until a given update time t_i, a graph $G(t_i)$ may be created from the stream $\{\langle t_1, a_1\rangle, \langle t_2, a_2\rangle, \ldots, \langle t_i, a_i\rangle\}$.

As the stream is unbounded, stream ingestion becomes the problem of applying a new action a_{n+1} at time t_{n+1}. We make the formal requirement $t_{n+1} > t_n$ to avoid the ambiguity which may arise if an insertion and deletion of the same entity occurred at the same time. In addition to this restriction, before an action can be applied its preconditions must be satisfied as it may otherwise leave the graph in an inconsistent state. For example, if a_{n+1} requests the removal of an edge, this can only be considered valid if the edge exists at time t_n. The full list of these preconditions can be seen in Table 1.

Entity Addition.

For the addition of a given vertex v, it must first be checked if $v \in V_T$. If v has never been a member of the graph, V_T is updated to include this new member $V_T := V_T \cup v$; a history is then assigned to v specifying the time of its creation $H^v = \{\langle t_{n+1}, created \rangle\}$. If $v \in V_T$ we check if the current status is *deleted* and if so add this new update into the history $H^v := H^v \cup \langle t_{n+1}, created \rangle$. If the current state is already *created* the update is considered invalid and is abandoned.

Edge addition is similar to this, where if an edge $\langle v_i, v_j \rangle$ is to be added we must first check that v_i and v_j are present in V_T and currently *created*. If this is not the case the addition is rejected. If these are present, we may then check if $\langle v_i, v_j \rangle \in E_T$, dictating if the edge requires insertion into E_T or if its history requires appending, in the same fashion as described for vertex addition.

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 *deleted* state at the time at which the update occurred. For example, for an edge e deleted at time t_{n+1}, its history would be updated as follows: $H^e := H^e \cup \langle t_{n+1}, deleted \rangle$. Vertex removal is executed in the same manner, but requires an additional step to remove all present edges within E_T with the vertex as a source or destination, as these are now considered hanging edges. This is completed by appending their history with a *deleted* state at the time of vertex removal.

Entity Properties and Updates.

Entity properties are established and updated during the creation of vertices and edges, as well as standalone update commands. In the former case the *addition* of an entity y will come with a set of one or more key value pairs, specifying the properties to update and their new values. Take the case of a single property update $u_i^y = \langle k_i, value_i \rangle$ arriving at time t_{n+1}. This will be added onto the history of k_i for entity y, that is $p_i^y := p_i^y \cup \langle t_{n+1}, value_i \rangle$. Multiple properties can be updated together in the obvious way. Note, this means entities may both change the values of established properties over time as well as gain new properties (if $p_i^y = \emptyset$ when the update occurs). Property history may, therefore, vary in temporal depth, with their value prior to the point of inception defaulting to $\emptyset$ to remove ambiguity.

Property updates separate from entity addition are analogous to this, but must first confirm if the entity is present within Y_T and set to *created* at time t_n. This is necessary as an entity must be created before its properties can be updated and, in a similar vein, property values should not change if the entity is currently removed from the graph. It should be noted that changes to the entity state (creation or deletion) do not affect its associated properties unless explicitly specified.

4. Challenges of Distribution

4.1. Model Distribution and Maintenance

With the model described in Section 3 in mind, to be able to expand past the limits of vertical scaling and handle the large graphs generated by modern data demands, it must be implemented in a distributed fashion; this, however, comes with several challenges. Firstly, in practice, scalability through distribution is synonymous with graph partitioning, splitting the graph into manageable chunks for each machine. It must, therefore, be decided what a temporal graph partition consists of and the strategy for splitting the overall graph whilst retaining high data locality (e.g. edge-cut or vertex-cut, as described in PowerGraph[8]). However, unlike PowerGraph, partitioning a temporal graph has the additional complexity of managing trade-offs between structural locality (proximity to neighbours) and temporal locality (proximity to an entities history) as assessed in

Table 1: Table of events for an update at time t_{n+1}

Event Type	**Parameters**	**Preconditions**	**Effect**
Add Vertex	v, $\langle k_i, value_i \rangle$	$v \notin V_T$	H^v=$\{\langle t_{n+1}, created \rangle\}$ &
(new vertex)			$p_i^v := p_i^v \cup \{\langle t_{n+1}, value_i \rangle\}$
Add Vertex	v, $\langle k_i, value_i \rangle$	$v \in V_T$ & $H^v(t_n)$ = deleted	$H^v := H^v \cup \langle t_{n+1}, created \rangle$ &
(established vertex)			$p_i^v := p_i^v \cup \{\langle t_{n+1}, value_i \rangle\}$
Add Edge	e=$\langle v_i, v_j \rangle$, $\langle k_l, value_l \rangle$	$e \notin E_T$	H^e=$\{\langle t_{n+1}, created \rangle\}$
(new edge)		$v_i \in V_T$ & $H^{v_i}(t_n)$ = created	$p_l^e := p_l^e \cup \{\langle t_{n+1}, value_l \rangle\}$
		$v_j \in V_T$ & $H^{v_j}(t_n)$ = created	
Add Edge	e=$\langle v_i, v_j \rangle$, $\langle k_l, value_l \rangle$	$e \in E_T$ & $H^e(t_n)$ = deleted	H^e:=$H^e \cup \langle t_{n+1}, created \rangle$
(established edge)		$v_i \in V_T$ & $H^{v_i}(t_n)$ = created	$p_l^e := p_l^e \cup \{\langle t_{n+1}, value_l \rangle\}$
		$v_j \in V_T$ & $H^{v_j}(t_n)$ = created	
Remove Edge	e=$\langle v_i, v_j \rangle$	$e \in E_T$ & $H^e(t_n)$ = created	H^e:=$H^e \cup \langle t_{n+1}, deleted \rangle$
Remove Vertex	v	$v \in V_T$ & $H^v(t_n)$ = created	H^v:=$H^v \cup \langle t_{n+1}, deleted \rangle$
			Remove all edges containing v
Update Property	y, $\langle k_i, value_i \rangle$	$y \in Y_T$ & $H^y(t_n)$ = created	$p_i^y := p_i^y \cup \{\langle t_{n+1}, value_i \rangle\}$

ImmortalGraph [16]. 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[25].

Secondly, unlike single machine systems which can maintain global state, distributed systems must continuously synchronise between machines which share state to minimise inconsistencies. Within a distributed graph this shared state comes in the form of entities which have been cut by the partitioning algorithm and must be copied onto all machines where they are utilised. This can be mitigated with higher data locality, but irrelevant of the chosen partitioning strategy a distributed graph will inadvertently have some 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 their 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.

In conjunction with synchronisation, updates coming into the system may arrive out of order. Whilst this is often due to unavoidable factors such as random network delay between partitions, it is exacerbated by mechanisms for increased throughput, such as concurrent ingestion or multiple data sources. The update semantics above encompass all possible changes, but their prerequisites are based on strict assumptions of serial ingestion and processing. A distributed environment breaks these constraints meaning additional handling of updates is required to ensure they are not processed incorrectly or dropped unnecessarily. For instance, if an edge add arrived before the addition of its source vertex, the update would be incorrectly abandoned. Previous systems have solved this problem by blocking update insertion until they can be correctly ordered, e.g. Kineograph's[9] epoch micro-batching, or via centralised ordering, e.g. Weavers[10] oracle. These are, however, sub-optimal with the former generating a high degree of staleness within the latest version of the graph, not truly representing the live data, and the latter bottlenecking the system, restricting its ability to scale[15]. This should instead be solved in a manner which allows updates to be processed as soon as they arrive, providing the least staleness for any ongoing analysis without relying on a central entity for a ground truth.

Finally, in addition to the problems of distribution, the above model makes no concessions for memory utilisation when placed within real machines. This is an issue for all in-memory systems where, even with a large cluster of servers, as the data grows the memory limitations are eventually reached. However, this is even more of a factor here as all updates and previous property values are maintained in-memory. The model must, therefore, be implemented in a manner which minimises the per-update memory footprint, without compromising readability of an entities history/neighbour list during analysis [16]. Furthermore, protocols should be established to govern what history is retained in memory and what is offset onto more permanent offline storage; this should then be able to be retrieved at a later date if required for processing.

4.2. Analysis

Whilst the distribution and maintenance of the graph is an important milestone, it must also be processable, which comes with its own set of unique challenges. In line with previous discussion on immediate insertion of updates, the processing model should enable continuous analysis of the graph, ensuring updates are included in a timely manner, otherwise returned results could be as stale as if ingestion had taken several minutes. This would also provide an ongoing log of metrics showing how the graph is changing in near real-time. A balance must be struck here, however, as executing too quickly may yield an incorrect result, due to delayed updates or unsynchronised state.

In contrast to timely analysis on new updates, being a temporal graph, it should be possible to perform analysis on the state at any point throughout its history. Given this history may be either in-memory or stored on disk, the retrieval and analysis of offline data must be managed in a way which does not interfere with the maintenance of the in-memory graph. Furthermore, to streamline comparisons between historic points, this process should be transparent from the users perspective, allowing the same algorithms to run on the live graph and history on disk.

Finally, one of the largest overheads for the deployment of distributed analysis is the time taken to re-ingest data[26], which for many systems has to happen every time the code-base is altered as it must be recompiled. This is a major issue during prototyping as it bottlenecks the development cycles, but more importantly it means the graph would be offline for an undetermined period of time if their was a need to change some ongoing analysis. It should, therefore, be possible to submit new tasks without producing downtime for the graph. In a similar vein, in order that the incoming data does not have to be ingested into multiple deployments, concurrent algorithms should be able to execute on the graph, each returning their own results to the user.

5. Raphtory

To address these challenges we introduce *Raphtory*, a system which maintains temporal graphs over a distributed set of partitions. Raphtory is built to ingest and convert streams of events into graph updates, inserting these in real-time into an in-memory temporal graph. The full structural and property history of each vertex and edge is then fully curated, ensuring all changes are correctly ordered and allowing analysis on both the live graph and any point within its history.

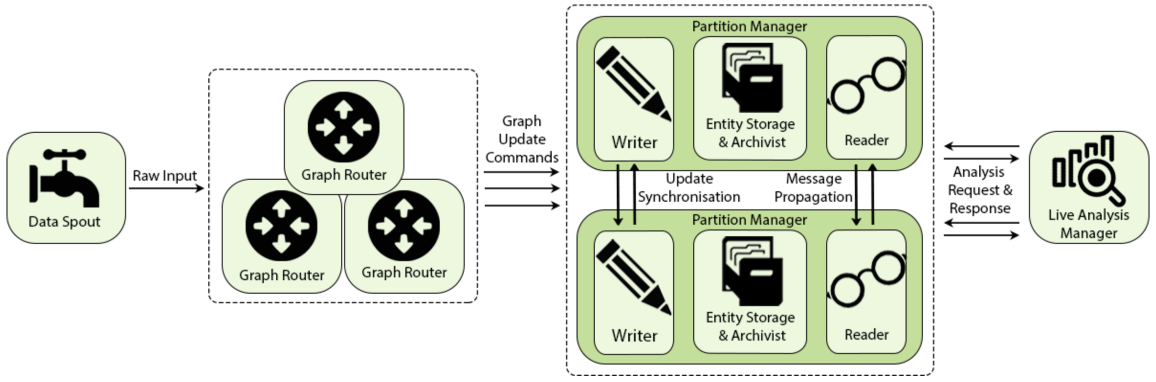


Figure 1: *Raphtory* Architecture Overview

Raphtory's architecture (Figure 1) is based on the actor model[19], a programming paradigm where 'actors' are the primitive unit of computation. Actors have no shared state and communicate via messages which evoke defined control flows, known as 'behaviours', based on the message type. Within these an actor may change its internal state, send messages to other actors or possibly spawn child actors to parallelise a given task. This greatly simplifies concurrent programming and mitigates against traditional multithreading hazards such as dead-locks and stochastic behaviour [27]. This also provides a uniform communication protocol between local and remote actors, enabling straightforward distribution and horizontal scaling. These together improve maintainability of the code-base, with the alternative multithreaded shared-state shown to become extremely cumbersome for large projects [28].

Raphtory's core components consist 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 3.2, 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. As updates arrive via the pool of *Graph Routers* the *Manager* will insert them into the histories of affected entities at the correct chronological position. This removes the need for centralised synchronisation as commands may be executed in any given arrival order whilst resulting in the same history. To deal with memory constraints *Partition Managers* both compress older history and set an absolute threshold for memory usage. If this threshold is met a cutoff point is established, requiring all updates prior to this time to be transferred to offline storage. This additionally has the benefit of acting as a backup mechanism allowing partitions to be recovered if the *Manager* was to crash.

Once established and ingesting the selected input, analysis on the graph is permitted via *Live Analysis Managers*. These connect to the cluster, broadcasting requests to all *Partition Managers* who execute the algorithm in Bulk Synchronous Parallel supersteps, returning partial results after each iteration. Analysis may be completed on the live graph, or at any point back through its history, with *Raphtory* handling the retrieval of data pushed on to disk. Additionally, multiple *Analysis Managers* may operate concurrently on the graph with previously unseen algorithms compiled at run-time, allowing modification of ongoing analysis without re-ingesting the data. In this section we explain each of these components in detail, as well as introducing the Raphtory API which allows users to ingest their own data, parse this into graph updates and compute analysis on the resulting graph.

5.1. ***Graph Router***

Graph Routers are the point of ingestion for raw data/events, converting these into the graph update operations defined in Section 3.2. *Routers* forward each extracted operation to the *Partition Manager* storing the affected entity. All commands generated by *Graph Routers* are assigned a timestamp, which *Graph Partition Managers* use to place the command correctly within the history of all affected entities. By default, 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. This allows *Graph Routers* to operate without synchronising and means events can be read in across a range of time periods in parallel, with the temporal graph ordering them accordingly. If such a field is not present within the data, *Graph Routers* may alternatively utilise their own internal clock to create a timestamp, synchronised via the Precision Time Protocol (PTP) [29]. However, allocating timestamps in this way means events must be ingested in the same order to generate the same temporal graph. To enable message routing, Raphtory maintains a global partitioning algorithm which decides where each vertex is stored, currently a hash partition, as this requires no state and can scale along with the number of *Routers* and *Partition Managers. Graph Routers* process events independently, allowing the resources allocated for ingestion to match the magnitude of data throughput by adding or removing *Routers* from the pool as required.

5.1.1. Temporal Windows and Property Decay

By default, a fire and forget protocol is established for outgoing commands, allowing *Routers* to remain stateless. However, as entity removal is not always prevalent in real world datasets, *Graph Routers* also provide facilities for ‘temporal windowing’. A temporal window specifies a period of time which all vertices and edges within the live graph must have been either added or updated; entities outside this window are no longer considered part of the graph and a removal is inserted into their history. This has several use cases and can drastically change the structure of a graph by varying the window size. For example, if investigating the popularity of users within a social network over time, those with tens of millions of followers may always reach high in the listing. By pruning inactive entities, smaller users rising quickly in popularity will become easier to extract. Furthermore, by varying the window size from minutes to months, an analyst will be able to experiment with both short and long term patterns within their data[30].

To provide this facility, each *Router* may be set to track all entities it has extracted from parsed messages and the last time they were seen. Entities which have not appeared for a defined period of time can then be ‘pushed’ out of the live graph by sending a removal command to its containing *Partition Manager*. As *Routers* operate independently and may receive messages with the same entities, it is possible for a *Router* starved of messages containing a given entity to generate erroneous removals. *Partition Managers*, therefore, sanity check these updates, ignoring those affecting entities which have received event based updates within the defined timeout window.

A softer form of windowing is also provided within Raphtory via ‘Entity Decay’. This operates in a similar manner, but instead of pushing entities out of the graph, the *Router* updates their properties with a decreased value. After an update is sent, the timer can be reset, allowing for numerous rounds of decay to occur. Continuing the example above, the *Router* may decay a weighting on each edge within the graph, allowing newer edges to contribute the majority of a users ranking, but older connections to still have some effect. Removal is possible here if desired; once the values of the defined properties drop below a designated threshold, or the number of decay iterations reaches an upper bound, the *Router* will issue a removal for the entity.

5.1.2. Modelling Data Sources

To streamline data ingestion, event sources in *Raphtory* are modelled as *Data Spouts.* An event source in this context can range from databases and file repositories to streaming API's and message queuing systems. A *Data Spout* will perform the initial connection required to access data within one of these sources, consuming tuples/events and distributing to the *Graph Routers* in a standardised manner. *Data Spouts* are fully decoupled, enabling parallel ingestion from multiple heterogeneous sources, with spouts permitted to join and leave the cluster at run-time. This is useful when reading from repositories which are geographically separate or when joining a new distinct source with one already established within the graph. *Data Spouts* send events to all *Routers*, load balancing via Round-Robin.

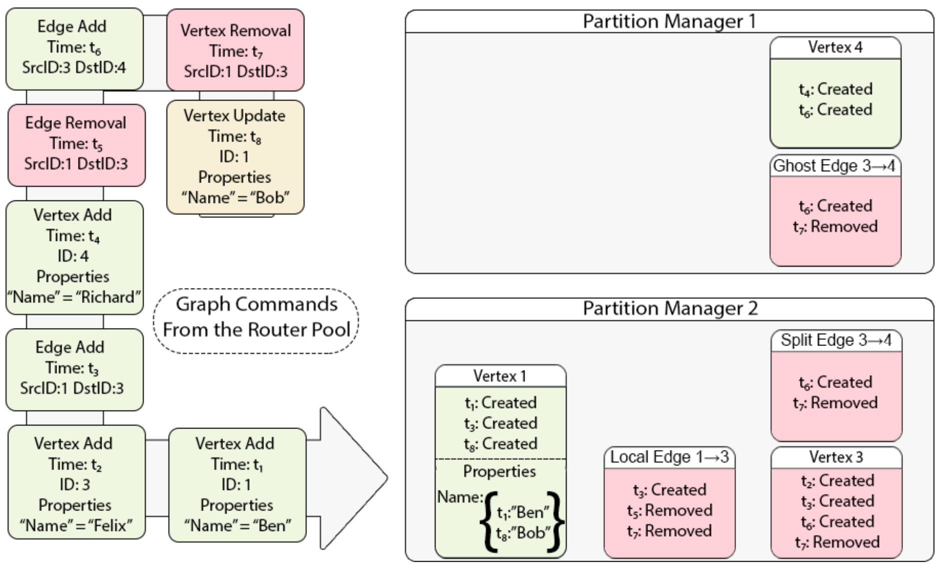
5.2. ***Graph Partition Manager***

Graph Partition Managers are Raphtory's primary component, each storing a partition of the overall in-memory graph. These partitions contain a unique set of vertices and their incoming/outgoing edges, each with their own structural and property histories. *Partition Managers* are responsible for maintaining up-to-date histories, completing analysis requests and performing incremental backups for these entities delegating these tasks to three sub-components, the Writer, Reader and Archivist. Writers are in charge of handing updates to the state of the graph, inserting these into the history of affected entities. Readers handle requests for analysis, executing the provided algorithm and returning the results. Archivists then work in the background, persisting new data to permanent storage and archiving the older entities to alleviate memory constraints.

5.2.1. Entity Storage Singleton

As the Writer, Reader and Archivist may need to access the same entities concurrently, all vertices and edges are contained within the *Partition Managers* storage singleton which ensures safe operational interleaving. Entities within the singleton are stored as objects which contain its meta data (such as its ID), its structural history and a map of its associated properties. Entity history is maintained via a red-black tree, providing fast access when reading the history and efficient insertion time for delayed or out of sequence commands. Property objects within the associated properties map mirror this structure, containing the property key and their own tree based history of previous values. All entity objects are then stored within a TrieMap[31], which provides concurrent, thread-safe and lock-free access.

Raphtory's in-memory graph is split in an edge-cut fashion, with each vertex stored fully within one partition and edges managed primarily by the *Graph Partition Manager* storing its source vertex. If the destination vertex is contained within the same partition, an edge is considered 'local'; if it is in another partition the edge is considered 'split'. In the case of split edges, a 'ghost' copy of the edge is maintained by the *Partition Manager* storing the destination vertex, allowing both to access its state. Split edges are stored in the same fashion as a local edge, but additionally record the location of their master/ghost copy. If a change occurs to a split edge, this information can be utilised to forward the update, synchronising the copy. Edge cut and vertex hashing was chosen as Raphtory's partition strategy as it has good properties in balancing the number of edges per partition and fits well within the decentralised environment. Additionally, as discussed in Section 4, partitioning graphs on both space and time is substantially complex, even on static data, with the algorithm of choice often determining the best strategy[16].



Edge Add Time: t_6 SrcID:3 DstID:4
Vertex Removal Time: t_7 SrcID:1 DstID:3
Edge Removal Time: t_5 SrcID:1 DstID:3
Vertex Update Time: t_8 ID: 1 Properties "Name" = "Bob"
Vertex Add Time: t_4 ID: 4 Properties "Name" = "Richard"
Graph Commands From the Router Pool
Edge Add Time: t_3 SrcID:1 DstID:3
Vertex Add Time: t_2 ID: 3 Properties "Name" = "Felix"
Vertex Add Time: t_1 ID: 1 Properties "Name" = "Ben"
Partition Manager 1
Vertex 4
t_4: Created
t_6: Created
Ghost Edge 3→4
t_6: Created
t_7: Removed
Partition Manager 2
Vertex 1
t_1: Created
t_3: Created
t_8: Created
Properties
Name: { t_1:"Ben" t_8:"Bob" }
Local Edge 1→3
t_3: Created
t_5: Removed
t_7: Removed
Split Edge 3→4
t_6: Created
t_7: Removed
Vertex 3
t_2: Created
t_3: Created
t_6: Created
t_7: Removed

Figure 2: An example stream of graph updates & the equivalent temporal graph once distributed between two Partition Managers. Updates are coloured according to there type: Green for addition, red for removal, yellow for updates. Entities are similarly coloured according to their most recent state: Green for present, red for removed. Note that the properties for vertex 3 and 4 have been omitted for simplicity.

5.2.2. Writer

A *Partition Manager* Writer is responsible for all updates to the state of the in-memory graph. Updates come in the form of graph operations extracted from the raw data by *Graph Routers*, as well as synchronisation messages from other *Partition Managers*. To mitigate spikes in throughput, incoming updates are stored in an ordered queue until they are processed by the appropriate function within the Entity Storage Singleton, passing the entity ID, property key value pairs and update time. These operations are executed in the following manner:

Adding Entities. When adding a vertex, if no object exists for the given ID one is instantiated, beginning its history and establishing the property map. If an object is already present, a new *Created* state will be inserted into its history, even if the latest state denotes a creation. This is done in case a remove command has been delayed, which may then be slotted between these upon arrival. Adjoining key value pairs will then be inserted into the property map. Edges are initialised or updated in the same fashion. However, receipt of an edge add will also generate vertex add commands for its 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 create a ghost copy of the edge and the destination vertex.

Removing Entities. Similarly, for deletions, the entity is not checked to see if it is already absent as a delayed creation may arrive and need to be placed between the two removals. If no object exists for the entity, a placeholder is initialised, beginning the history with the *Remove* state. The delayed creation may then be slotted into the history when it arrives, as well as establishing the edge's property map. In the case of a split edge removal, the command will again be propagated and create placeholders for the source and destination, given these do not exist.

In addition to a change in its own state, removal of a vertex requires insertion of a *Remove* state into the history of 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 2, if the command which removed vertex 3 arrived before the command which added edge 3→4, 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. To prevent this occurring, the *Removals* contained in adjoining vertices must be inserted into the history of the edge upon creation. This way, if an edge misses the execution of a vertex removal the information is still present.

For local edges, *Remove* states can simply be extracted from the history within the source and destination objects. Split edges, however, require the *Managers* storing the source and destination to exchange this information. In this instance, the *Manager* handling the edge creation will send a request for *Removes* present within the destination vertex, along with a set of *Removes* present in the source. The receiving *Manager* will then return the requested data and insert the *Removes* into the new edge. This does generate some overhead for edge initialisation, but it is a one time occurrence and the exchange is non-blocking as the request and response are handled as normal update messages.

5.2.3. Archivist

As one of *Raphtory's* main goals is to maintain its graph in memory, the combined RAM of each *Partition Manager's* host machine sets an initial upper bound on its overall size and temporal depth. The *Archivist* reduces memory load on the system by running checks on the stored entities, compressing old history and offloading entities which are no longer active onto secondary storage. This allows new updates to be inserted into the graph, but also means older history can be retrieved for analysis if the user needs to go back further than the hardware limitations will allow.

As stated in section 5.2.2, new states are added into the history of an entity whenever an update is received, even when this state matches the previous. This verbose chronicling allows delayed commands to be inserted into the correct position, but can quickly generate a large memory footprint which impacts the amount of updates a *Partition Manager* can store. For example, a vertex added to the graph four times would contain the following history: $H{=}\big\{\langle t_5, created\rangle, \langle t_4, created\rangle, \langle t_2, created\rangle, \langle t_1, created\rangle\big\}$, and as such a *Remove* occurring at t_3 can still be inserted upon arrival. This is considered writable history, and states exist in this malleable form for a window of time before it is considered implausible for an update to arrive which occurred before it, at which stage it may be compressed to save memory. Compression is completed when an *Archivist* checks the writable history of an entity, iterating through all states. Consecutive points of the same type which have been present longer than the established write window are pruned from the tree, reducing the example above to: $H{=}\big\{\langle t_4, created\rangle, \langle t_3, removed\rangle, \langle t_1, created\rangle\big\}$. This stores the same information, with the entity still considered present at t_2 and $t_{n>4}$, but uses 60% of the space. Each of the properties associated within an entity will then go through a similar process, removing duplicate values outside the compression window.

As the range of times associated with data ingested by *Raphtory* can vary from seconds to years, the size of the compression window cannot be a static number. Instead, it is set as a percentage of the difference between the time of the oldest in-memory update and the newest, adjusting to the time-range as more updates are ingested. By default this is set to 90%, meaning that given a difference of 10 minutes, any updates which occurred within the last minute would be in a writable form, whilst all within the remaining 9 minutes would be compressed. If by the next compression cycle the difference has increased to ten days, the oldest 9 days would be compressed. This provides a flexible interval for out-of-order updates to be stored correctly, even when bulk ingesting many years of data.

Once compressed, the history is saved by the *Archivist* to secondary storage. This prevents data loss in the case of system errors and simplifies the archiving process. Persistence in *Raphtory* is handled by Cassandra[32]; chosen for its scalability, as well as its high asynchronous write throughput on time-series data[33]. When storing an edge or vertex for the first time, a new row within the database is created, storing the entity meta data and representing the history via a Cassandra Map. Updates to an already saved entity are then placed within this map, minimising the number of rows that must be searched when retrieving an entity for processing. Each row also contains a field for the oldest update, allowing pre-filtering at the database level when pulling a large number of entities back into *Raphtory*. This structure is then mirrored for each entity property.

Finally, once entities have been compressed and persisted, they are safe to be archived i.e. removed from the in-memory graph to free up space. The *Archivist* will attempt to maintain as much history in-memory as possible, monitoring the available memory on the host machine and only archiving if this passes a user defined threshold. Once the threshold has been crossed, the archivist iterates over all entities within the partition, removing events outside a set temporal depth. Similar to the compression window, this depth has to be dynamic and change with the time-range currently stored. By default this is 10% of compressed history, i.e. if 10 minutes of history is compressed, any updates within the oldest minute would be deleted. If all events in the history are older than this depth, the whole entity is archived, leaving only its ID so it can be retrieved at a later date if required. This process is repeated until the available space is back to an acceptable level.

5.3. Reader

Readers are the processing engine within *Raphtory*, executing user defined functions on the entities within their partition. Readers are completely stateless and operate upon 'Analysers' sent from *Live Analysis Managers*. Analysis is based on a vertex centric model, similar to Pregel[6]. When a reader receives an *Analyser*, they will complete one superstep, informing the *Analysis Manager* of the completion and returning some partial results. This allows new entities to be quickly inserted into any ongoing analysis and provides continuous feedback on the state of the in-memory graph.

To access graph entities inside of a partition, the *Analyser* must utilise the *Graph Retrieval Proxy*. This proxy masks the complexity of retrieving entities which have been archived and ensures only the Writer is able to modify graph state. If analysis is to be performed on the live graph, the *Proxy* will return the most recent state of all present vertices and any messages received within the previous superstep. This set may then be iterated over, completing the algorithm specified within the *Analyser*. During this algorithm, associated edge state can be requested and messages may be sent to other vertices within the graph. If the analysis is to be performed on a snapshot, initially it is seen if the requested time t is in-memory or has been pushed onto secondary storage. If it is still in-memory, the partition's vertex/edge map is cloned and iterated through, filtering out entities' created after or removed from the graph at point t. The remaining entities properties are then screened to select the value for t, as described in Section 3. Alternatively, if t has been pushed to disk, Cassandra is contacted and all relevant entities are brought back into *Raphtory*. These then go through the same filtration processing as those in-memory, generating the required snapshot.

Taking this one step further, as some machines during a deployment of Raphtory may have different memory constraints, it is reasonable to assume that different temporal depths may exist on each machine. Given a point in time is chosen which is in-memory on one machine and on disk on another, each *Reader* will generate their portion of the snapshot in the required manner. Whilst this may be noticed via the difference in setup time between *Partition Managers*, it is otherwise transparent to the user, necessitating no additional input. The only major caveat with storing cloned snapshots is the physical space they require, which may have to be gained via a reduction of temporal-depth on the live graph. This is, however, handled automatically by the Archivist and the memory will be relinquished once the analysis is completed.

5.4. Live Analysis Manager

To allow the user to interact with the graph, create snapshots and oversee ongoing analysis, *Raphtory* provides the *Live Analysis Manger*. Each *Live Analysis Manager* is responsible for the full execution of one *Analyser*, tracking which superstep the algorithm is on, what functions the *Reader* must run and any termination conditions, broadly split into three stages: setup, analyse and finish. Analysis begins with a setup request broadcast to all *Readers*, performing any prestart necessities such as sending initial messages. Once all readers have reported back to the *Analysis Manager*, the first superstep of the analyse stage is broadcast, consisting of the user defined function to be executed on each vertex. Once all Readers have again notified the *Live Analysis Manger* they have completed the superstep, it may perform any final aggregation of returned results and decide if another superstep is to be performed or if the analysis is complete. If the latter is chosen, the finish stage will be executed, removing any generated snapshots.

In addition to this base life-cycle, *Live Analysis managers* support perpetual analysis on the live graph. This will have no finalising stage and will continuously run supersteps, including any newly ingested updates in the analysis. This gives the user the least stale version of any metrics of interest, but as the analysis is occurring in parallel with graph mutations, and updates may be delayed, the returned results must only be considered approximative. To check the degree to which approximate results are correct, *Live Analysis Managers* may be requested to recompute each result after graph compression has occurred for the time the results were returned. The result pairs may then be compared to get a rolling average for accuracy within the system, which can be taken into consideration for both business decisions and future algorithmic choice. In line with this, it is quite possible that use cases/algorithms evolve throughout the life of a Raphtory Deployment. *Live Analysis Managers* may, therefore, join and leave the cluster at run time. To facilitate this, *Readers* which do not possess a new *Analyser* version will request the source and compile this at run-time, allowing prototyping/algorithmic changes without re-ingesting the raw data on a fresh deployment.

5.5. API

To maximise accessibility, *Raphtory* provides a simple API for ingesting new data sources and performing analysis on the graphs generated. For data ingestion, users must create their own Spout and *Router*, overriding a couple of key functions. For the Spout this is an *ingest* function which attaches to the chosen data source, passing incoming data to the *sendToRouter()* function handling communication and load balancing between individual *Routers*. *Routers* are handled in a similar manner, requiring the user to override *ParseRawInput*, which runs on each message received from the Spout. To simplify this conversion, the *Router* provides a set of functions to create each update type, only requiring essential data such as the time of update, entity ID and property key value pairs (if applicable). These functions then handle the routing of the created command to the relevant *Partition Manager*.

Once the input is established, a user may then implement their own *Live Analysis Manager/Analyser* pair. The Analyser will require the user to override the *setup*, *analyse* & *finish* functions described above, making use of the *Graph Retrieval Proxy* API. The *Analysis Manager* then requires the completion of two functions, one for processing the partial results received from each *Reader* and one to decide if the algorithm has converged after each superstep. Both of these, however, have helper methods for collecting the data, starting another superstep and sending the final results back to the user. Algorithms 1 and 2 below give a flavour of this API, implementing the Setup and Analyse Functions for a simple PageRank[34].

Algorithm 1 Example Setup Function for PageRank

```
GraphRepoProxy.setTime(t)
size = GraphRepoProxy.VertexCount()
for each $vertex \in GraphRepoProxy.getVertices()$ do
    value=1/size
    vertex.setProperty("pageRank",value)
    for each neighbour $\in$ vertex.getOutgoingNeighbours() do
        GraphRepoProxy.sendMessage(neighbour,value)
    end for
end for
```

Algorithm 2 Example Analyse Function for PageRank

```
GraphRepoProxy.setTime(t)
size = GraphRepoProxy.VertexCount()
for each $vertex \in GraphRepoProxy.getVertices()$ do
    messageTotal=sum(vertex.getMessages())
    value=((1/size)*messageTotal)
    vertex.setProperty("pageRank",value)
    for each neighbour $\in$ vertex.getOutgoingNeighbours() do
        GraphRepoProxy.sendMessage(neighbour,value)
    end for
end for
```

6. Evaluation

The evaluation of *Raphtory* is based on the principals established in GraphTides[15]. This work discusses how distributed stream-based graph processing systems differ from their batched/offline counterparts, key focal points when testing such systems and the importance of reproducibility. To best provide the desired reproducibility and simplify deployment, Raphtory's Base implementation has been compiled into a Docker Image[1], allowing deployment as a set of containers via Docker

[1]https://hub.docker.com/r/miratepuffin/raphtory/

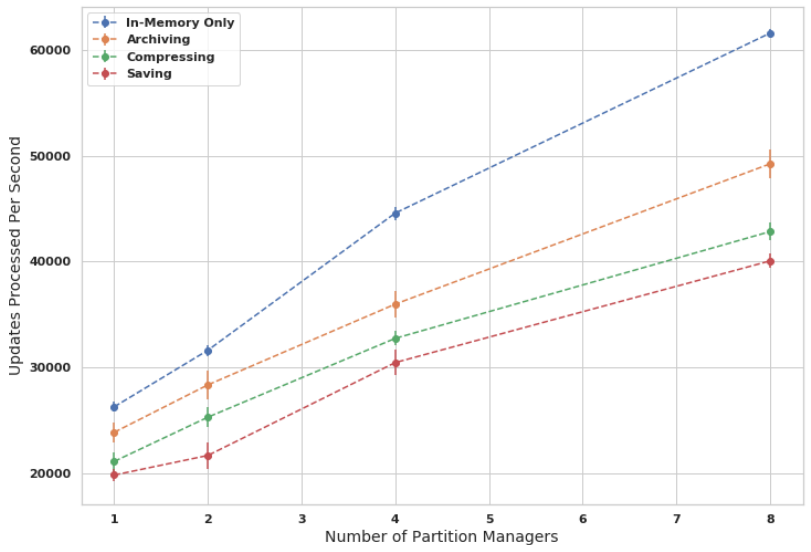


Figure 3: Maximum update throughput of *Partition Managers* when ingesting only vertex and edge additions.

Swarm. Once deployed, Docker Swarm establishes a virtual network for these containers, allowing them to contact the 'cluster seed' (a known actor) and any other connecting components, forming the overall actor system. This means the number of *Spouts*, *Routers* and *Partition Managers* can be set within a configuration file and will automatically connect with each other and set about their given roles. Furthermore, the classes for these actors (implemented by the user or otherwise) may also be set within configuration, minimising the need to recompile the image. Utilising all of these attributes, all tests within this evaluation have been automated as part of a Popper Pipeline[35], as requested in GraphTides. The scripts for these can be found within the Raphtory Repository[2].

6.1. Initial Testing

To assess if the fundamental ingestion and maintenance functionality of Raphtory was scalable, initial stress testing was carried out on clusters of increasing size. This consisted of a singleton implementation containing one *Partition Manager*, referred to as $Cluster_1$, followed by three distributed deployments doubling the number of *Partition Managers* each time; referred to as $Cluster_2$, $Cluster_4$ and $Cluster_8$ respectively. For each cluster size, four different configurations were deployed, investigating pure ingestion throughput and the effect of archiving, compression and saving. These consisted of: *In-memory only*, where the Archivist was disabled; *Archiving*, enabling the Archivist, but only to drop the oldest history; *Compression*, enabling compression, but without saving to Cassandra; and *Saving*, where the full archiving process was executed.

All deployments were instantiated on a docker swarm cluster of 15 servers each containing an Intel Xeon E3-1284L (8 cores @ 1.8GHz) and 32GB of RAM. Eight of the physical machines were reserved for *Partition Managers*, four were assigned to be *Routers* and two were setup as a distributed Cassandra ring. The final node was then allocated to a Prometheus container which automated

[2]https://github.com/miratepuffin/Raphtory-Deployment

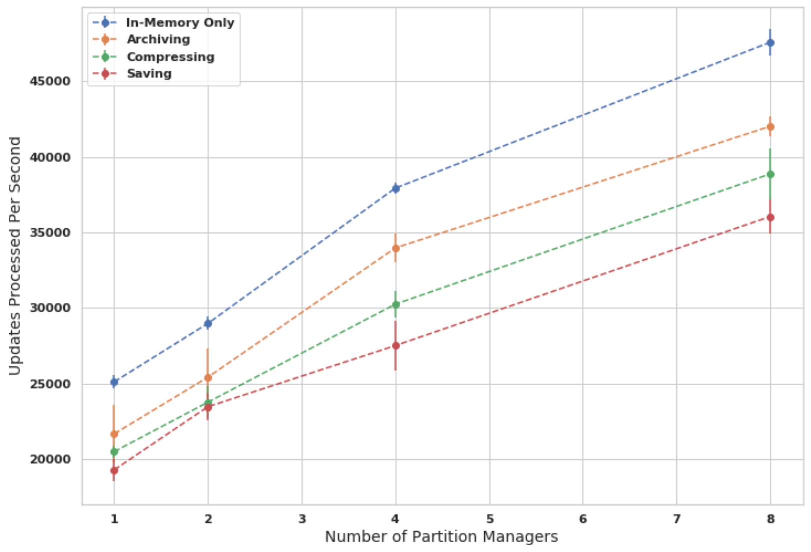


Figure 4: Maximum update throughput of *Partition Managers* when ingesting a worst case workload of graph updates.

log aggregation alongside a watchdog which monitored the cluster for failed components and 'dead letters' (messages dropped due to queue overflow). Once a deployment had been established a Spout was then deployed in a separate cluster, simulating an external event source. This would begin sending updates to the *Routers*, increasing its output every minute (+ 1000 messages/s) until the watchdog reported any issues, at which point the deployment would be decommissioned and the maximum throughput recorded.

Every configuration was tested on two different workload types, implemented within a spout which would output random updates based on which was chosen. The first of these was an 'add only' workload consisting of 30% vertex adds and 70% edge adds over a pool of one million unique vertex ID's, with an equal probability of each ID being chosen. Each of these additions also contained two properties to be updated, allocated from a pool of 20 possible keys and values. The second workload contained all update types split into 30% vertex adds, 40% edge adds, 10% vertex removals and 20% edge removals over the same pool of ID's and property key/values (in line with stream generation in GraphTides). This was designed as a worse case scenario workload with a high amount of churn and deletions often arriving much earlier than their equivalent addition. This was chosen to ensure that Raphtory could handle all update types discussed equally and that interleaving additions and removals correctly adhere to the established update semantics, with all corner cases covered.

Figure 3 displays the mean throughput achieved by five iterations of each configuration ingesting the add only workload. There are several interesting points to note here. Firstly, $Cluster_1$ performed well, ingesting up to 27,000 updates a second when *In-memory Only* and 20,000 when the full archiving process was enabled. Secondly, even with this high baseline, overall scalability was good, with throughput in $Cluster_8$ increasing to 62,000 (129%) and 40,000 (100%) respectively. Thirdly, as may be expected, as more of the archiving process was enabled the maximum throughput was reduced. Interestingly, however, these reductions are noticeably asymmetrical. The largest decrease seen (between *In-memory Only* and *Archiving*) appears as the worker threads must transfer from only inserting state changes and new property values to periodically checking all entities and curating their history. This is more prominent in larger deployments as the time taken for the *Spout* to reach the highest throughput's leads to a much larger graph for the *Archivists* to curate. The next reduction (between *Archiving* and *Compressing*) is less prominent, primarily because the graph entities are already being inspected. Finally, enabling saving to Cassandra caused the least disruption because the Cassandra call is handled asynchronously, allowing the manager to continue with new updates whilst the data is transferred between compression cycles.

In comparison to Figure 3, Figure 4 displays the corresponding results for the deletion workload. Here it can be seen that the throughput for all deployments are lower, with greater variance and less spread between the different configurations. This is because the increasing throughput quickly saturates the pool of vertex ID's, meaning the average vertex degree grows rapidly as more edges are added. Whilst the overhead from this is capped in the add only workload, with a maximum two extra messages generated by new edge synchronisation, here each vertex removal necessitates synchronisation messages for all associated split edges. As more edges are split with the increasing number of partitions and the amount of edges grows with time, the number of messages sent and processed also increases, impeding the *In-Memory Only* deployments from reaching their previous heights. This brings them closer to the *Archiving* runs as the impact from archiving is unaffected by the composition of history as it only drops the oldest part. Removals do however affect compression, requiring more updates in the compressed state to correctly record each entities history. This lower compression ratio means the history takes up a greater portion of memory, demanding a higher frequency of archiving in addition to more data being transferred to Cassandra. This creates the variance seen within both *Compressing* and *Saving* as the stream composition and, therefore, compression ratio differs for each of their runs. Vertex removals are clearly drastic operations. Fortunately though, the level of churn in this workload is seldom plausible in real world datasets, especially on very high degree nodes.

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 required, utilising a larger array of machines, real datasets, and exploring the effectiveness of the Raphtory analysis model.

6.2. Future Test Plan

As discussed in [5], one of the major problems of current graph processing systems is their lack of scalability. The first tests will, therefore, expand on the initial scalability investigation above, increasing the total number of *Partition Mangers* within the swarm cluster to 64. Once each cluster size has a known maximum throughput, a second test will be conducted on each investigating Raphtory's ability to handle more bursty traffic. This will begin by initially ingesting updates at 60% of the maximum throughput, but then varying this between 1% and 500%. These will be longer running deployments than the previous test, to ensure that the system can fully recover from this sort of input instead of crashing at some point during the future.

Following this, the general analysis model will be evaluated, performing a user ranking on a social graph generated via the LDBC Social Network Benchmark [36]. This will consist of ingesting the same size graph (10GB in size) into different deployments, increasing the number of *Partition Managers* as with the scalability test. Analysis will be executed both during ingestion and once the graph has been fully built, comparing the time taken for the algorithms to converge and thus evaluating the Partition Manager's ability to share resources between the Reader and Writer. Similar analysis will also be carried out on snapshots, both in-memory and on disk, investigating the memory/throughput impact of building snapshots, the ability of the Archivist to alleviate memory constraints and the difference in setup time. Finally, the accuracy of approximate analysis will be tested by continuously running the algorithm on the live graph and comparing the returned values to a ‘golden standard’ generated during snapshot execution. This will be additionally supplemented via watermarks injected into the stream, with each actor logging the watermark ID and time seen. These will give a good estimate of the ‘true throughput’ of the system, i.e. the time taken for an update to reach the analysis stage, not just be ingested.

As all previous tests have only been carried out on generated graphs, Raphtory will also be deployed on several real world datasets. The first of these is a full scrape of the social network Gab.AI[37], consisting of 26 million posts spanning a 4 year period, metadata on the users who made each post and any tagged ‘topics’ (similar to a hashtag). This data will be ingested into the largest Raphtory deployment with the live graph continuously analysed utilising a ranking algorithm modified to take into account the difference between posts, users and topics. As the popularity of topics/users can change rapidly within Gab, and there is no data which could be interpreted as removals, it is a perfect dataset to test Temporal Windowing. The *Routers* for the deployment will, therefore, have temporal windowing enabled with a window size appropriate for the rate of ingestion. The fluctuation in most popular topic/user can then be tracked over time, with the initial approximate results once again compared to snapshots to see how accuracy varies when deletions are included within the graph.

The second real dataset used for testing will be the full Bitcoin[38] blockchain. This is by far the largest dataset to be tested (over 400GB) and will be used to examine Raphtory’s ability to handle input much larger than the available memory; another key feature discussed in [5] where several state of the art systems suffer from OOM errors in this situation. This will be built into a transaction graph[39] with the aim of performing taint analysis[40] on newly ingested blocks. This will be split over two Live Analysis Managers working in tandem, one to attach new transactions to the source of their coins (rebuilding the chain as this is obscured from an individual block’s perspective) and one to track tainted coins. A secondary objective here will be to compare latency of analysis within Raphtory against state of the art blockchain analysis tools such as BlockSci[41], which currently take several minutes to ingest and display statistics about a new block once it is published. Finally, as several datasets are to be ingested, the effectiveness of history and property compression will be tracked to see how this varies both between datasets and throughout time.

7. Conclusion and Future Work

In this work we have discussed the importance of making full use of the wealth of information in the history of an evolving graph, ingesting updates via a stream and distributing the workload to handle modern data demands. Through analysis of the related work it was shown that current graph processing systems individually incorporate a subset of these facilities, but seldom combine them all. To remedy this, we formalised a temporal graph model along with the semantics for updating its structure and property values. We examined the challenges of distributing this model and processing in parallel with update ingestion, presenting the Raphtory system as the solution. Raphtory splits the graph over a set of *Partition Managers* which maintain the full history of each vertex and edge in-memory, coordinating the ingestion of new updates, analysis requests and data archiving. Raphtory provides easily extensible components to allow users to define their own data sources, graph generation and analysis, as well as permitting new *Spouts* and *Analysis Managers* to join at run-time. Finally, Raphtory's scalability was demonstrated via initial throughput testing with further analysis and use cases proposed.

This is, however, only the beginning of the Raphtory project, with several important additions planned for future investigation. The first component of this will be developing an appropriate adaptive partitioning strategy[25] to maintain high data locality within the graph. This is a multi-faceted problem with several interesting questions. For example, if it is optimal to transfer nodes to different *Partition Mangers* to maintain locality is it best to group several transfers together? Should recently connected/active neighbours have more effect when deciding on optimal partitioning strategies? There are also many practical decisions which need to be explored, for example, how can both *Graph Partition Managers* and *Graph Routers* keep track of which partition each node resides on?

The second expansion is then to enable temporal analysis. Raphtory's current API provides comparison of graph views throughout its history, using common time intervals such as hours or days. However, by expanding this to work directly on an entity's history a larger array of analysis will become available, as well as reducing the overhead of executing on a large set of views. There is currently little work in this area so there is ample space to explore, both in implementing various use cases, but more importantly providing the language and syntax for analysts to truly express what they mean when querying history.

8. References

[1] P. Holme, J. Saramki, Temporal networks, Physics Reports 519 (2012) 97 – 125. Temporal Networks.

[2] S. Sahu, et al., The ubiquity of large graphs and surprising challenges of graph processing, Proceedings of the VLDB Endowment 11 (2017) 420–431.

[3] H. Wu, et al., Path problems in temporal graphs, in: VLDB Endowment, volume 7, pp. 721–732.

[4] X. Qiu, et al., Real-time constrained cycle detection in large dynamic graphs, Proceedings of the VLDB Endowment 11 (2018) 1876–1888.

[5] K. Ammar, T. Ozsu, Experimental analysis of distributed graph systems, arXiv preprint arXiv:1806.08082 (2018).

[6] G. Malewicz, et al., Pregel: A system for large-scale graph processing, in: ACM SIGMOD, ACM, pp. 135–146.

[7] Y. Low, et al., Graphlab: A new framework for parallel machine learning, in: arXiv preprint arXiv:1408.2041.

[8] J. E. Gonzalez, et al., Powergraph: Distributed graph-parallel computation on natural graphs, in: USENIX OSDI, volume 12.

[9] R. Cheng, et al., Kineograph: taking the pulse of a fast-changing and connected world, in: ACM EuroSys, pp. 85–98.

[10] A. Dubey, et al., Weaver: a high-performance, transactional graph database based on refinable timestamps, in: VLDB Endowment, volume 9, pp. 852–863.

[11] L. G. Valiant, A bridging model for parallel computation, Communications of the ACM 33 (1990) 103–111.

[12] R. S. Xin, et al., Graphx: A resilient distributed graph system on spark, in: ACM GRADES, ACM.

[13] M. Zaharia, et al., Spark: Cluster computing with working sets, HotCloud 10 (2010).

[14] P. Carbone, et al., Apache flink: Stream and batch processing in a single engine, Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 36 (2015).

[15] B. Erb, et al., Graphtides: a framework for evaluating stream-based graph processing platforms, in: Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), ACM, p. 3.

[16] Y. Miao, et al., Immortalgraph: A system for storage and analysis of temporal graphs, in: ACM TOS, volume 11.

[17] X. Ju, et al., Version traveler: Fast and memory-efficient version switching in graph processing systems, in: USENIX Annual Technical Conference, pp. 523–536.

[18] B. Erb, et al., Chronograph: A distributed processing platform for online and batch computations on event-sourced graphs, in: ACM DEBS, pp. 78–87.

[19] G. A. Agha, Actors: A model of concurrent computation in distributed systems., Technical Report, MIT Cambridge Artificial Intelligence Lab, 1985.

[20] F. Kuhn, R. Oshman, Dynamic networks: Models and algorithms, SIGACT News 42 (2011) 82–96.

[21] D. Kempe, et al., Connectivity and inference problems for temporal networks, Journal of Computer and System Sciences 64 (2002) 820 – 842.

[22] J. Moody, The importance of relationship timing for diffusion, Social Forces - SOC FORCES 81 (2002) 25–56.

[23] N. Santoro, et al., Time-varying graphs and social network analysis: Temporal indicators and metrics, AISB 2011: Social Networks and Multiagent Systems (2011).

[24] P. Holme, Network dynamics of ongoing social relationships, EPL (Europhysics Letters) 64 (2003).

[25] L. M. Vaquero, et al., Adaptive partitioning for large-scale dynamic graphs, in: IEEE ICDCS, pp. 144–153.

[26] L. M. Vaquero, et al., Deploying large-scale datasets on-demand in the cloud: treats and tricks on data distribution, IEEE Transactions on Cloud Computing 3 (2015) 132–144.

[27] S. Rehfeld, et al., An actor-based distribution model for realtime interactive systems, in: 2013 6th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), IEEE, pp. 9–16.

[28] E. A. Lee, The problem with threads, Computer 39 (2006) 33–42.

[29] IEEE, Ieee standard for a precision clock synchronization protocol for networked measurement and control systems, Available at `https://standards.ieee.org/findstds/standard/1588-2008.html`, 2008. Accessed: 03-05-2018.

[30] J. Saramäki, E. Moro, From seconds to months: an overview of multi-scale dynamics of mobile telephone calls, The European Physical Journal B 88 (2015) 164.

[31] A. Prokopec, et al., Concurrent tries with efficient non-blocking snapshots, in: ACM Sigplan Notices, volume 47, pp. 151–160.

[32] A. Lakshman, P. Malik, Cassandra: a decentralized structured storage system, ACM SIGOPS Operating Systems Review 44 (2010) 35–40.

[33] S. K. Jensen, et al., Modelardb: Modular model-based time series management with spark and cassandra, Proceedings of the VLDB Endowment 11 (2018).

[34] L. Page, et al., The PageRank citation ranking: Bringing order to the web., Technical Report, Stanford InfoLab, 1999.

[35] I. Jimenez, et al., The popper convention: Making reproducible systems evaluation practical, in: Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International, IEEE, pp. 1561–1570.

[36] O. Erling, et al., The ldbc social network benchmark: Interactive workload, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, ACM, pp. 619–630.

[37] S. Zannettou, et al., What is gab? a bastion of free speech or an alt-right echo chamber?, arXiv preprint arXiv:1802.05287 (2018).

[38] S. Nakamoto, et al., Bitcoin: A peer-to-peer electronic cash system (2008).

[39] D. Ron, A. Shamir, Quantitative analysis of the full bitcoin transaction graph, in: International Conference on Financial Cryptography and Data Security, Springer, pp. 6–24.

[40] R. Anderson, et al., Making bitcoin legal, in: Security Protocols Workshop.

[41] H. Kalodner, et al., Blocksci: Design and applications of a blockchain analysis platform, arXiv preprint arXiv:1709.02489 (2017).