Raphtory: Modelling, Maintenance and Analysis of Distributed Temporal Graphs

Benjamin Alexander Steer

PhD Thesis
Submitted in partial fulfilment of the
requirements of the Degree of Doctor of Philosophy

School of Electronic Engineering and Computer Science
Queen Mary University of London

2020

Statement of Originality

I, Benjamin Alexander Steer, confirm that the research included within this thesis is my own work or that where it has been carried out in collaboration with, or supported by others, that this is duly acknowledged below and my contribution indicated. Previously published material is also acknowledged below.
I attest that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge break any UK law, infringe any third party's copyright or other Intellectual Property Right, or contain any confidential material.
I accept that the College has the right to use plagiarism detection software to check the electronic version of the thesis.
I confirm that this thesis has not been previously submitted for the award of a degree by this or any other university.
The copyright of this thesis rests with the author and no quotation from it or information derived from it may be published without the prior written consent of the author.

Signature: Benjamin Alexander Steer
Date:23/10/2020

Details of collaboration and publications: The full list of publications in relation to this thesis is discussed in Section . Of these:

- The work presented in Section 6.6 (GraphTides[1]) was completed as a joint effort between researchers from Queen Mary University of London, Imperial College London and The Institute of Distributed Systems at Ulm University.
- The work presented in Section 6.4.2, currently under review at The International Conference on Web and Social Media (ICWSM), was completed as a joint effort between researchers at Queen Mary University of London and Universidad Politécnica de Madrid.

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 majority of graph processing systems only consider traditional graph analysis on static data, whilst those which do expand past this often only support batched updating and delta analysis across graph snapshots. In this work we define a temporal property graph model and the semantics for updating it in both a distributed and non-distributed context. We have built Raphtory, a distributed temporal graph analytics platform which maintains the full graph history in memory, leveraging the defined update semantics to insert streamed events directly into the model without batching or centralised ordering. In parallel with the ingestion, traditional and time-aware analytics may be performed on the most up-to-date version of the graph, as well as any point throughout its history. The depth of history viewed from the perspective of a time point may also be varied to explore both short and long term patterns within the data. Through this we extract novel insights over a variety of use cases, including phenomena never seen before in social networks. Finally, we demonstrate Raphtory's ability to scale both vertically and horizontally, handling consistent throughput in excess of 100,000 updates a second alongside the ingestion and maintenance of graphs built from billions of events.

Acknowledgements

I would like to thank everybody who has helped me during the completion of this thesis. First and foremost, thanks to my primary and secondary supervisors Félix Cuadrado and Richard Clegg. Throughout my time at Queen Mary they have always been fully supportive of my endeavours and fostered a wonderful exploitative environment. Their guidance and feedback has been truly invaluable to my research and I feel deeply lucky to have been able to work alongside them. I would also like to thank my tertiary supervisor, Steve Uhlig, for ensuring my research journey progressed successfully.

Thanks also goes to my wonderful colleagues in the QMUL Networks Research Group, especially Naomi Arnold, Timm Böttger, Dami Ibosiola and, most recently, Imane Hafnaoui who have helped me with innumerable issues and made my time at Queen Mary a joy. In addition I would like to thank my many collaborators, notably Gábor Szárnyas, Jack Waudby, Alhamza Alnami and Haaroon Yousaf who have helped tremendously with both my main research threads and fruitful side ventures.

Finally, a special thanks to my family and friends for their unwavering support and encouragement, particularly my mum, Jacki Steer, who has always been my editor-in-chief, much to her dismay. To everyone else not mentioned here, I express my heartfelt appreciation to you all.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Graphs are a powerful abstraction which 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 and social network analysis. By superseding the base graph model with that of a temporal graph[2], we may additionally capture the evolving interconnectivity of entities within the underlying dataset over time. This unlocks a breadth of analytical possibilities by expanding on standard graph algorithms, such as providing congestion aware GPS navigation via a temporal shortest path algorithm[3]. Furthermore, this model fits naturally within a streaming architecture, where new events may be inserted into the graph upon arrival and compared to related entities or historical state. For instance, the e-commerce site Alibaba ingests all new sales into a graph of the previous week's transactions to monitor for fraud in real-time[4].

With this in mind, there are a number of challenges which must be overcome as graph processing systems mature[5], many of which focus around the manner in which the user interacts with the system. These range from reducing the complexity of deployments and improving the manner in which raw data may be ingested and modelled, to providing intuitive analytical APIs and descriptive visualisations of results. However, scalability is the main barrier faced by graph practitioners and researchers[6], as graph algorithms are very intensive in memory and computation resources. There has, therefore, been substantial development in distributed graph processing systems which can scale both vertically and horizontally in order to enable large-scale graph analytics.

Across these systems there has been an historic focus primarily on the analysis of static graphs built from bounded datasets (e.g Pregel[7], GraphLab[8] & PowerGraph[9]). This is a batch oriented style of processing where bulk data is read in, a graph is created, transformed in some manner and an output garnered. This works well for use cases requiring a large collection of data to be ingested only once and where there is little time constraint. However, in many business sectors (such as the examples above) new information is always arriving, meaning graph snapshots soon lose relevance. This, therefore, requires continuous generation of fresh snapshots, including all updates, alongside timely processing to obtain relevant results.

In response, newer systems have begun to buck this trend, ingesting streams of data whilst maintaining an in-memory graph model (e.g. Kineograph [10] and Weaver [11]). These, however, only focus on the most up-to-date version of the data, and fail to realise the potential insights they are losing be overwriting older property values and not maintaining the order of how the graph came to be. Finally, the temporal graph processing systems which attempt to tackle this issue often come with their own caveats. Many execute on static pre-prepared data (e.g. ImmortalGraph[12] and Version Traveller[13]) where new information cannot be inserted. Whilst others which are online often only perform delta-based analysis across coarse snapshots, losing temporal resolution between these (e.g. LLAMA [14]), or natively do not support 'time-aware' graph algorithms where the history/update order is included (e.g. Chronograph[15]).

1.2 Research Hypothesis and Problem

Taking this all into consideration, it is felt that online temporal graph analytics have been largely overlooked by the systems community and as such its application within most use cases is laborious, requiring graph system experts and bespoke software solutions. It is, therefore, hypothesised that if a platform were created which enabled the inclusion of a graph's history within analysis in an intuitive and scalable manner this would empower the wider data science community to explore and adopt the area, deriving novel insights within their established datasets.

This is, however, a non-trivial task which comes with a host of problems which must be overcome. Firstly, the definitions of temporal graphs vary widely and there are no clear semantics for updating and distribution. Secondly, as stated above, graph processing systems often lack intuitive API's for ingestion and analysis and are data intensive applications. All of these elements are magnified with the addition of time. For ingestion it must be established how to extract graph entities from raw data alongside how/when they change. For analysis this information must be explorable, both in terms of running an algorithm at any point in the history as well as providing access to the history within algorithms. Finally, this historical information must also be stored and indexed for quick modification and query, opening several questions on best practices for partitioning, caching and memory management.

1.3 Aim

The overall aim of this thesis is, therefore, to design and build a system which provides online temporal graph processing which can scale alongside the increasing demands of modern datasets, provide intuitive ingestion/analysis APIs and innovative time-aware analytical functionality. To this end there are derived requirements which the platform must fulfil. Firstly, the system should be online, ingesting new data into the graph, and be distributable across multiple machines, allowing it to deal with high throughput and large data volumes. Secondly, new updates streamed into the system should be able to be inserted without batching or a centralised ordering oracle, eliminating loss of temporal resolution and centralised bottlenecks. Finally, the developed system should be able to encompass all previously discussed processing models, alongside introducing novel analytical paradigms. This means facilitating traditional and temporal graph analysis on the most recent instance of the graph, as well as at any point back through its history.

1.4 Research Contributions

There have been four major areas of research contribution towards obtaining this goal. The first of these is the definition of a distributed temporal property graph model, which includes the semantics for ingesting events from a stream and updating the graph across partitions. This allows events to be delayed and arrive out of order, leveraging the history of entities to ensure the correct graph is always created. This additionally includes the definition of 'graph flattenings' which are a way of viewing the equivalent static graph built from a range of updates in the stream, extracted from the temporal graph model.

The second contribution comes via the first implementation of this model, *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 across partitions, based on the established semantics. Furthermore, Raphtory decouples the ingestion and transformation of data to permit the same sources to be modelled in a variety of ways, as well as simplifying the ingestion of data separated across silos. All elements of ingestion and modelling, as well as the partitions which maintain the graph, may additionally be scaled independently depending on the use case at hand. Finally, Raphtory implements a novel form of watermarking and memory management, ensuring the user is always executing on a complete set of data up to a given time and the in-memory graph is kept within the limitations of the machines housing it.

Complementary to this is the third area of contribution, Raphtory's analytical engine. This operates in parallel with ingestion, executing on extracted graph flattenings. Within this abstraction, edges and vertices are given access to their full structural and property histories. By exposing the graph in this manner, users may develop both traditional and time-aware graph algorithms and apply them easily at any point within the history of the graph. To exploit this further, APIs are provided to allow ranges of flattenings to be executed over periods of interest, as well as applying windows with varying temporal depth, to explore both short and long term patterns. Lastly, these algorithms may be set to execute continuously on the most recent version of the graph, periodically returning the newest result.

Raphtory was shown to scale well both vertically and horizontally, maintaining high throughput over very large graphs. Additionally, across analysis of various datasets, Raphtory was able to extract many surprising insights. Notably within this was the continual collapse and reformation of a giant connected component in the social network Gab, a phenomena never seen before in this domain. Performing this analysis in Raphtory was shown to be 300x faster than when conducted in Spark GraphX, a popular choice for large scale graph analytics. Finally, during the evaluation of Raphtory, it was discovered that a standardised manner for benchmarking streaming graph processing systems was yet to be established, leading to the development of GraphTides. This combined elements of stream processing with graph workloads and included the creation of a testbed and methodology for fair system comparison.

1.5 Thesis Structure

Chapter 2: Background The first chapter following the introduction provides an overview of the related literature. This initially dives into the different types of graph model, how they are materialised and what graph analysis, both traditional and time-aware, consists of. Following this, there is a look at general use big data platforms and why graph processing systems tend to be more bespoke, and finally there is a full exploration of the graph ecosystem.

Chapter 3: Temporal Graph Model Once the surrounding literature is understood and the problem contextualised, Chapter 3 defines a temporal graph model upon which the developed system may be based, expanding on those explored in Chapter 2. The challenges of distributing such a model are then discussed, and the stream semantics for updating the graph are defined for both a distributed and non-distributed context.

Chapter 4: Raphtory Ingestion, Modelling and Maintenance The fourth chapter introduces Raphtory, the developed temporal graph analytics platform, initially summarising how ingestion and analysis ties together. Following the overview, this chapter discusses how the defined model is brought to life within Raphtory. This includes how data is ingested into the system, how a user may model this stream of events as graph updates and how the updates are processed across partitions to build and maintain the graph. Finally Raphtory's watermarking model is introduced, tracking where in the graph's history is safe to analyse.

Chapter 5: Raphtory Analysis The sixth chapter initially provides an overview of the types of analysis possible within Raphtory. This presents the two level API established, one level for the development of new algorithms by the user, interacting with the graph, and a second higher level for selecting time points or ranges upon which to execute. This chapter also discusses how the developed algorithms may execute safely in parallel with update ingestion.

Chapter 6: Evaluation With Raphtory having been fully described, this chapter covers the evaluation of both the ingestion and analytical components. This includes the scale up and out testing of Raphtory's throughput, the exploration of two datasets - the social network Gab and the Ethereum blockchain - a comparison with Spark GraphX and the definition of a streaming graph system benchmark, GraphTides.

Chapter 7: Conclusions and Future Work The final chapter of this thesis concludes with a summary of the contributions brought forward in prior chapters and discusses a number of research directions in which Raphtory will be taken in the future.

1.6 Associated Publications

Segments of the work detailed in this thesis have been presented in the following international scholarly publications (journal publications highlighted in bold):

- An initial version of the work on distributed stream ingestion and graph modelling from Chapter 4 was presented at the International Conference on Distributed and Event-based Systems (DEBS)[16], being awarded best poster[17]. A later version was presented at Advances in Mining Large-Scale Time Dependent Graphs (TD-LSG); a VLDB Workshop [18].
- The final publicised versions of Chapters 3, 4.2, 4, and an early version of 5, were accepted for publication in the journal **Future Generation Computer Systems**[19].
- The work presented in the first section of Chapter 6, establishing a benchmark for streaming graph systems, was presented at the Joint International Workshop on Graph Data Management Experiences & Systems and Network Data Analytics (GRADES-NDA)[1]. This work was additionally integrated into standard graph benchmarks, under the banner of the Linked Data Benchmark Council; similarly published at GRADES-NDA[20].
- Finally, the analysis of the social network Gab, utilising Raphtory, is currently under review at The International Conference on Web and Social Media (ICWSM), but is available to view on archive[21].

Chapter 2

Background

2.1 Introduction

The overall goal of this work is to provide scalable online analytics for temporal graphs, unlocking the insights they provide whilst handling the continuously increasing demands of modern datasets. However, even this succinct summary provides many terms and concepts which may be foreign to the reader. Therefore, this chapter contextualises these concepts by introducing and exploring the surrounding areas of literature. The first of these areas focuses on defining what a graph is, building up from basic elements to more complex models, and how these are often materialised within real systems. These 'static' graph models are then expanded upon, discussing how they may mutate as the underlying dataset changes with time, resulting in dynamic or time-evolving graphs. Finally, temporal graphs are disambiguated from their time-evolving counterparts, exploring models in this space which may be expanded upon in this work.

The second area of focus is the types of algorithm which may be applied to graphs, categorising these based upon their structural scope, i.e. how many of the entities within the data are involved in the analysis. Algorithms across the range of structural scopes are then discussed in relation to what new insights they may extract when provided with the history of the graph entities involved; made available via temporal graphs. The goal here is not to focus on one specific area or manner in which an algorithm has been implemented, but to broadly understand what is possible.

In a similar vein to this, the third area is a broad coverage of general big data tools developed to handle the distribution and analysis of enormous datasets. Within this a distinction is made between batched-based systems and streaming systems, or those which handle bounded and unbounded datasets. As the goal of this work is directed more to the latter, key components of streaming systems which will be important in later chapters are discussed, namely windowing and watermarking. Finally, the ability of general big data platforms to perform graph analytics is discussed, noting where the major flaws lie and why bespoke solutions have been more successful in this regard.

Following this the forth and final area focuses upon these aforementioned graph-specific systems. Again a high level distinction is drawn here, this time between graph databases and graph analytical platforms, due to the differing workloads they prioritise. The latter of these is explored in detail, as this work focuses on analytics, looking at systems which work with static, dynamic and temporal graphs in both a batched and streaming nature. Through this exploration, clear areas of improvement are noted and taken forward within the subsequent chapters.

2.1.1 Chapter Roadmap

Section 2.2: Graph Models and Representations Here we discuss the different types of graph model and their characteristics, providing our definition for dynamic and temporal graphs as well as how these may be materialised.

Section 2.3: Graph Analysis Categorisation This section then explores the different types of algorithms which may be applied to a graph, categorising them based on the portion of the graph they touch and whether they are temporal in nature.

Section 2.4: Big Data Processing Systems Before touching graph-specific processing systems we take a brief look at general big data processing systems, how they work, and how they have been applied to graph problems.

Section 2.5: Graph Processing Systems We then take a deep dive into a range of graph processing systems, focusing on those with distributed, dynamic and temporal features which may be taken forward.

2.2 Graph Models and Representations

When exploring the concept of graph models, we may first consider the simplest form in which a graph $G = (V, E)$ consists of a set of vertices V and edges E. Vertices constitute the entities within the represented network, whilst edges denote the relationships between these entities. There are, however, many variations of this model with distinct use cases and applications. A graph may be directed or undirected, referring to whether the edges between vertices have an associated direction. If this is the case one of the vertices will consider the edge outbound and be the 'source' of the relationship, whilst the other vertex involved will consider the edge inbound and be the 'destination'. In an undirected graph, the edge has no defined start or end and is considered equally within both vertices. For example, in a social network, if two people are friends this is a reciprocal relationship and may be modelled as an undirected edge. Alternatively, if one person subscribes/follows a second to see their content, this may not be reciprocated and would, therefore, be modelled as a directed edge[22]. In addition to direction, a graph may be weighted in which vertices or edges have an associated numerical value, or more generally, labelled to denote some attribute. Continuing with the social network example, edges may be weighted between users to show how many interactions they have with each other, or perhaps denote some 'friendship score'. Labels could be used in the same instance to give context to the graph, for example labelling vertices with their names. Finally, the graph model may be generalised further into a hypergraph[23], in which an edge may involve any number of vertices, all of which may be labelled and/or weighted. Note graphs are defined more thoroughly in Chapter 3.

Two notable models which encompass many of the above features are the Labelled Property Graph (LPG) model[24] and the Resource Description Framework (RDF) model[25]. The Labelled Property Graph model consists of a directed graph within which all vertices and edges have a set of associated labels, referred to as properties or attributes. These properties consist of key value pairs allowing them to be referenced individually. Property graphs often additionally divide their vertices and edges by 'type', with graph components of the same type sharing a similar set of properties; possibly managed by a graph schema[26]. These are, therefore, popular within graph databases[27] where they are used to represent and query complex datasets. The RDF model on the other hand consists of a set of $\langle subject, predicate, object \rangle$ triples, which are used to represent both the directed relations in the underlying data and entity labels. Subjects are vertices denoted by a Uniform Resource Identifier (URI), giving them a unique reference across all data. Predicates are the relationship or attribute label name, also denoted by a URI. Objects are then either a destination vertex, if the predicate is an edge, or *literal* (attribute value) if not. RDF models have been popular in the creation of knowledge graphs[28] and the semantic web[29], providing structure across loosely connected datasets which contain intersecting references to real world entities.

Materialisation

An abstract graph model may be materialised (i.e. established in computer memory) in a number of ways, depending on the modelled content and goal of the analysis to be executed on top[30]. Fundamental representations include the adjacency matrix, adjacency list, edge list and adjacency array (also known as Compressed Sparse Row or CSR format)[31]. An adjacency matrix representation consists of a matrix $M^{n,n}$ where $n = |V|$ i.e. the number of vertices within the graph. The coordinates within the matrix then reference all possible edges between vertices, with 0 denoting an absence and 1 (or any number for a weighted graph) denoting an edge presence, i.e. $M_{v,u} > 0 \leftrightarrow (v, u) \in E$. As this may be sparse, the adjacency list representation contains the set of vertices instead, providing each with a list of their outgoing neighbours (vertices with whom they share an edge, where they are the source). For example, for a vertex $v \in V$ and its associated adjacency list L_v, $u \in L_v \leftrightarrow (v, u) \in E$. This removes all 0's stored, but increases the lookup time when seeing if two vertices share an edge. Edge lists are similar to this, but provide each vertex with a reference to its incoming and outgoing edges. Finally the CSR format stores all the edges in a contiguous array, with a vertex array containing pointers to where the edges of each vertex begin within the data. This provides very good locality for outgoing edges when scanning across the full data during analysis[32].

For more complex graph models such as the LPG and RDF, the basic graph representations may be used in conjunction with auxiliary data structures storing the vertex/edge attributes. Alternatively the two may be combined, such as in RDF triple stores[33] or in object oriented graph implementations such as Neo4j [34].

2.2.1 Dynamic Graphs

No matter the choice of graph model, it is important to note that in almost all real world use cases the underlying dataset which is being mapped to a graph will change as time progresses. Graph mutations come in the form of topological changes, adding/removing vertices and edges, and attribute changes, modifying labels/weights, if included within the model[35]. Continuing the social media example prior, we can imagine vertex mutations as new users joining the network, or existing users leaving. Likewise edge mutations may consist of new friendships being established or users unfollowing each other. Attribute changes follow similarly where the number of interactions between two users increase over time, reflected in the weighting, or users may update their username, requiring the labels to be modified. The model itself must, therefore, have some way of managing this, transferring from a static graph to a dynamic graph[36]. Dynamic (or time-evolving[37]) graph models will consider the affect of mutation both in terms of how it may be completed and how it will impact analysis running on top.

Depending on the underlying data structure mutations may occur in-place, updating the established graph representation, or require creating new data structures to store the new version. For example, an edge may be added or removed from the adjacency matrix by changing the value within a coordinate, but a CSR may require recreation of the contiguous edge array to make space. The complexity of each mutation type will also vary across data structures. For example, an adjacency matrix edge mutation is $O(1)$ as above, but vertex mutations are $O(n)$ as a whole row and column of the matrix must be created/deleted. An edge list will fare much better here as it may look up which vertices share edges, deleting these references from their list and removing the vertex itself; $O(m)$ for a given vertex v where $m = |L_v|$ and $m << n$. The choice of representation may, therefore, be varied not just based upon access patterns, as with the static graphs, but also upon what sort of mutation workload will be applied on top. This has lead to the creation of hybrid models such as 'Packed CSRs'[38] which leave space between elements in the edge array, allowing much quicker insertions and deletions, but at the expense of traversal speed.

Mutations may be considered to occur at either discrete or continuous points in time. Discrete changes may be established via multi-versioning, with either a sequence of full 'snapshots' or 'snapshot deltas'[39]. In the former, the state of the graph is presented in full at set increments (e.g. an hour, day, etc.) including all mutations up until that point, such as in [40]. Mutations are, therefore, not applied here and snapshots may just be swapped between. In the latter, only a base graph snapshot is maintained in full, with each delta containing the difference between this and the snapshot it represents[13]. This reduces the redundant information, but does mean that the full graph has to be read alongside the delta and the changes applied. Continuous mutations on the other hand come in the form of a list denoting each change individually, often with a timestamp of when this occurred. These may be offline, such as a log, or online, such as a stream where new information is continuously arriving. These may be applied individually, but in many cases mutations from continuous sources are batched together to create the state of the graph at a discrete point and to minimise the per-mutation impact[10].

2.2.2 Temporal Graph Models

A second way to approach the evolution of a network is to model it as a temporal graph. In this instance mutations do not overwrite the current state of a graph component/label, but may instead append the new state alongside, noting the time of change. This way the graph may be queried as to whether it existed at any point in the past, as well as include time restrictions as a component of the query. It is important to note here that having many multi-disciplinary applications, a range of these models are available in the literature under various pseudonyms such as temporal networks [2] and evolving graphs [41]. Within this space dynamic/time-evolving and temporal are frequently used synonymously. To remove confusion, in this work temporal graphs refer to those where the graph components and labels have an associated history of mutations, denoting how that element changed throughout its lifetime. In contrast, dynamic/time evolving graphs do not preserve any history, focusing only on maintaining the most up-to-date graph model. Additionally, temporal graphs may be non-streaming, in the sense they contain all the history they ever will and do not change, or streaming in that new information is still flowing in and being appended. As the goal of this work is to create a system which maintains a streaming temporal graph, we investigate how temporal graphs have been proposed and formalised, as well as understand the manner in which temporal information is traditionally stored, discussing desirable characteristics and noting possible expansions.

Beginning this overview with the most basic temporal graph models, [42] presents a directed graph where each edge is labelled to show the initial time of connection between the source and destination vertices. These labels may 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. In [43], however, graph edges are labelled with a range of time to include this feature. This could 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 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 [43], time-varying graphs implemented within [44] 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 introduces multi-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, so 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, [45] 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. Finally, [46] envisions a static graph consisting of all possible edges and vertices which could exist. These are then labelled with zero or more integers, denoting the points in time they existed, with zero labels meaning they never have. These labels are denoted λ, and the temporal graph is considered to be a set of static graphs $[G_1, G_2 \ldots, G_{max\lambda}]$ where a version of the graph exists for each increment between 1 and the maximum time label, containing all vertices and edges labelled with that time.

Building on top of these basic graph models, [47] defines a temporal property graph model as $G = (V, E, P_V, P_E)$ where vertices and edges are tuples $(vid, \sigma, \tau) \in V$ and $(eid, \sigma, vid_i, vid_j, \tau) \in E$. Within these σ refers to the entity type/schema, where each type has an established set of property keys. τ then refers to the time interval for which the entity exists. These may be used as arguments to P_V and P_E (for vertices and edges respectively), returning the property values at that point in the lifetime of the entity. A different take on the temporal property graph is introduced in [48] and implemented in [49], where vertices and edges represent 'temporally valid graph events' which exist for a time-instant or time-period. For this the tuple $(id, t, P) \in V$ refers to a vertex event which occurs at time t and has a set of properties P for this time.

Materialisation

The materialisation of temporal graphs comes in a variety of forms. Often the data structures introduced above are used for the structural information, denoting all the vertices and edges which have ever existed. The topological and property histories are then stored in auxiliary data structures, or in many cases such as [49], indexed data-stores to handle the extra dimensionality time adds to each entity/property pair. Some works, such as [50] and [35], suggest an alternative to this where the temporal graph is 'flattened' into a standard graph. Here each version of a vertex (including its properties at that time) exists as a separate entity, thus when a vertex is 'updated' a new instance of it appears in the network. Each version of a given edge is then drawn between the temporally closest instance of their source and destination vertices. This allows the full temporal graph to be inserted into the structures defined above without modification, but adds an element of complexity when attempting to read the graph state or compute analysis on top. Finally, it is worth noting that few, if any, of these models refer to the temporal graph in a streaming context (as defined above), with those that do only providing brief reference to how it may be updated in-between analysis.

2.3 Graph Analysis Categorisation

Whilst the manner in which analysis on a graph may be completed varies with use case, model and system implementation (as discussed in section 2.5), it is important to understand the types of analysis which may be undertaken. Across these, algorithms may focus just on the structure of the underlying network, or may make ample use of the attributes/labels associated with each vertex/edge. A sensible way to broadly categorise these is via the structural scope of an algorithm. From this, four categories may be resolved: local queries; neighbourhood queries; graph traversals; and global graph analytics[27].

A local query refers to singling out specific vertices or edges and extracting something about them, for example the value of a property or checking associated labels. Returning to the social media example from Section 2.2, this may be querying the number of interactions between two known users. Neighbourhood queries expand this scope to look at an individual vertex, its edges and those vertices adjacent to it. Within this scope we may ask questions such as the in/out degree of a vertex, i.e. the number of friends/followers a user in a social network has. At this scope there is also graph pattern matching or network motifs[51]. These are statistically significant subgraphs surrounding a given vertex, often providing some deeper context to its activity within the network. These find substantial use across many areas of network science, for example product co-occurrence in e-commerce recommendation systems[52] and enzyme interaction in biological networks[53].

Traversals are when the algorithm is still tethered to a singular vertex, or small set of vertices, but expand past their neighbourhoods to explore a substantial portion of the graph. These normally follow the pattern of stepping from vertex to vertex, i.e. traversing the network, until a specific end goal is met. Notable examples of traversals are: single source shortest path algorithms[54] for finding the quickest route between two points in the network; random walks[55] for probabilistic modelling; and contagion/diffusion algorithms[56] to investigate how an infection may spread throughout a population. Finally, global graph analytics is when all vertices in the graph are involved in the algorithm, calculating some property for the entire network. These may be global versions of the prior algorithms discussed, for example the average in-degree of the network or counting the total triangles present[57]. They may also only exist in this category, for example centrality and ranking algorithms such as PageRank[58] and betweenness centrality[59], for finding the most important entities in the network.

2.3.1 Temporal Graph Analytics

The above algorithms all work on the assumption that the underlying graph is static, not pertaining to any evolving or temporal properties. For this work it is important to understand how graph algorithms may be expanded and applied when a temporal graph model is utilised. Snapshot based analysis and time-respecting traversal algorithms[50] are the two main camps for this. The first of these is the simplest, whereby the state of the network at a specific time is extracted from the temporal graph model, generating a traditional static graph, upon which any of the algorithms above may then be applied without modification. This may be done throughout the lifetime of the network, incrementing at set intervals, with deltas calculated between result sets to see how the network evolved in relation to the chosen metric[60].

This type of analysis is low effort and may produce some interesting insights not available on the aggregate graph, but it does not make full use of the temporal information within in the model. This is the purpose of time-respecting graph algorithms which expand the scope of those above, including the temporal dimension as part of the analysis. Each of the categories discussed may, therefore, be explored through this new lens. At a local scope we may look at how a vertex's attributes have changed throughout time, noting periods of activity/in-activity or anomalous behaviour[61]. For a vertex's neighbourhood we may consider temporal motifs[62] or subgraph patterns. These are pertinent when the order or temporal intervals[63] are important within the pattern. For example, in money laundering and financial crime, illicit transactions often cycle money back round to the original sender (i.e. have a temporal order) and occur at systematic intervals[64].

Traversals find many applications with this new scope. For instance a shortest path algorithm may now consider the amount of hops, the weight on edges and how this changes throughout time, as well as the intervals at which edges/vertices exist[65]. This is useful in the context of GPS navigation where the weight on edges may denote congestion on roads and change throughout the day. We may then consider problems such as the earliest arrival path when leaving at a given time, the latest departure path after which point the destination becomes unreachable or the optimum departure time to minimise travel[50]. Finally for global analytics we may again expand on those from other categories, for example defining temporal centrality rankings utilising time-respecting shortest paths[66]. It is also now possible at this scope to cluster or group vertices in the graph, not just on their structural locality, but also temporal locality[67]. This is useful for seeing how communities form and shift, as well as noting key members.

An expansion of this section and an exploration of the different types of temporal graph algorithms and their use cases enabled by this work can be found in Section 5.2.

2.4 Big Data Processing Systems

As datasets have become larger than can be managed by a single machine, even with massive vertical scaling, 'big data' platforms have become standard practice for analytics across all sectors[68]. These drastically simplify the process of distributed computing, providing simple APIs to allow users to describe their chosen algorithm, then fully orchestrate the execution of this task across any number of available machines. Graph analytics is no exception here. It makes sense, however, to initially cover more general big data systems to understand how these operate and why graph solutions tend to be more bespoke. These general big data platforms fall into two categories, batched/offline analysis and streaming/online analysis. Similar to the discussion of static vs dynamic above, batched analysis refers to when the data already exists/is 'bounded' and may be processed in full in one batch; streaming based analysis refers to analysis run on a continuous live feed of data (unbounded), possibly maintaining minimal state or having some real-time requirement.

2.4.1 Batched Systems

A multitude of batched processing systems exist in the big data ecosystem. However, the two enduring systems upon which many others are based are Hadoop MapReduce[69] and Apache Spark[70]. MapReduce provides a simple two step API (which most algorithms can be adapted into) consisting of *mapping*, where the raw data may be parsed into key value pairs, and *reducing*, where aggregation may take place. The data on which the algorithm is set to run over is stored in the Hadoop distributed file system (HDFS)[71], split into blocks and replicated across the available cluster of machines. One mapping task is allocated per block of data, with each being set to run in parallel on one of the machines storing the chosen blocks, 'moving computation to the data'. This is fully orchestrated by YARN[72] which manages the cluster resources, allocating 'containers' to submitted jobs. Once the mapping phase is complete the output tuples are grouped by their key and sent to the reduce tasks. These run the defined reduce function once per allocated key, outputting the final aggregates back to HDFS. Key allocation is managed in a decentralised manner where the mappers hash the output key and modulo this by the number of reducers, sending the tuple to the resulting reducer task ID. As long as all mappers have the same hashing function this allows Hadoop to scale to thousands of machines/tasks without having a central bottleneck.

MapReduce encapsulates a surprising number of analytical problems, including many graph algorithms. However, as graph analysis tends to have an iterative component to it (i.e. in traversals, label propagation or score convergence) they are not the best fit in Hadoop. This is because each MapReduce job must save the output to HDFS and then reingest it into the mappers for the next iteration. Whilst this will be managed by Hadoop/YARN, the overhead from constantly writing to and reading from disk massively increases the overall execution time.

In contrast to this, Spark is an in-memory processing system where data is maintained within 'Resilient Distributed Datasets' (RDDs) [73]. Upon these the user may perform functional 'transforms', such as mapping and filtering the rows, and actions which materialise the output, such as a reduceByKey (mimicking Hadoop) or saving the final transform to disk. Transforms are lazily evaluated until an action is reached, at which point an optimiser will work alongside YARN to establish the optimum execution order and which machines each stage should be performed in. RDDs build up a lineage from the transforms executed, keeping these in-memory if possible. This provides fault tolerance, but also simplifies iterative algorithms as the function may be called repeatedly on the same RDD until it converges, at which point the output may be written to disk. Spark may operate in the same cluster stack as Hadoop (with jobs run in-tandem if both are managed by YARN), reading and writing from HDFS.

Spark, therefore, allows complex problems to be set out across a number of transforms. It also solves the issue discussed within Hadoop, but in turn suffers with problems of its own for graph algorithms. The main challenge here is that the elements of an RDD are completely independent (so they may be executed upon in parallel) which makes it difficult to model a graph which is inherently interconnected. Graph algorithms, therefore, require the execution of many group-by/reduce-by steps to mimic vertex communication. These generate high network traffic as the data repartitions around the cluster and cannot be optimised via lazy execution as they are actions. Extensions of Spark have been created to address this, namely GraphX[74] and GraphFrames[75], but were not successful in this endeavour as discussed below.

2.4.2 Streaming Systems

Streaming systems provide a wide variety of approaches to processing data. In the most general term these are 'processing engines designed with infinite datasets in mind'[76], where data tuples may be handled as soon as they arrive or periodically in a 'micro-batching'/windowing format[77]. Within this scope many, if not all, of the algorithms/analysis which can be computed in a batched system may also be done within a streaming system. However, these often branch out with different flavours, notably low-latency/real-time requirements, minimal state maintenance or approximate analysis. The possible data sources are also expanded upon here, whereby a streaming system may ingest data off of disk/HDFS, connect to an online API/endpoint, such as the Twitter API[78], or subscribe to topics of interest via a message queueing system such as Kafka[79].

Windowing and Micro-batches

Before discussing notable streaming systems it is important to first understand the concept of windowing over streams. This is when a data source is split along temporal boundaries into a number of batches which may then have an algorithm applied[76]. Windows may be fixed, where the temporal size of the window is the same as the period between batches, or sliding where these two are independent. Fixed windows have no overlap in the tuples they contain, whilst sliding windows may have an overlap if the period is less than the window size, i.e. to include the last hour of data every minute.

Within these the time upon which the windows are based may refer to 'processing' time or 'event' time. Processing time refers to when the tuple was initially received by the system, whereas event time refers to when it actually happened within the real world. Ideally event time and processing time should be equal, but due to factors such as network delay, tuples often arrive much later or out of order leading to these desynchronising, sometimes by large amounts.

Processing time windows are appropriate if the underlying data has no concept of event time or the intention is to derive a result about the data as observed on arrival at the system. They are also much simpler to handle, the system just buffering results for the period and then executing the algorithm; this result then never changes. Event based windows provide a more accurate view of what happened within the underlying data source at any given point. These are, however, more complicated to produce as when a window is triggered for analysis it may be 'incomplete', i.e. not all of the tuples within the period have arrived in the system. To improve correctness the reported result for older windows may be incrementally refined as new tuples arrive[80].

Stream Semantics and Watermarks

A point of concern for streaming systems in general, but definitely in the case of window refinement, is how many times a record is processed and the side-effects of non-idempotent analysis. Alongside late arrival, tuples may be dropped or lost and as such streaming systems may consider different semantics for handling this. At a high level these fall under: *At-least-once*, where processed tuples must be acknowledged and, if not completed within a set timeout, are then replayed; *At-most-once*, where duplication is not used as it negatively affects results, instead raising an error and possibly accepting that some messages are lost; *Exactly-once*, where the data source and streaming system work together to ensure that any replayed messages are only processed once, with duplicates ignored, returning the correct result. The last of these is obviously the desired choice, but in practice can be difficult to guarantee.

Even if exactly-once semantics are obtained, the process of window refinement will still require the tuples to be reprocessed. To minimise the need for refinement, streaming systems may make use of watermarking, which helps it to decide the best time to trigger an event window based on the likelihood that all tuples for that window have arrived[76]. One type of watermark utilises the timestamps of the oldest known tuple yet to be processed at the current step in the pipeline. This can be propagated downstream letting later stages know to hold off on their analysis until the timestamp has crossed the end of the window they are planning to trigger. A heuristic watermark may then be created in-tandem with the first, calculating the probability that new tuples with an event time prior to the window will appear in the system. This may be based on prior delays experienced within the stream and help decide if it is better to wait longer or if it is fairly safe to trigger the window. Of course these heuristics are in no way perfect so the underlying algorithm should still be designed in a manner where re-execution has little or no effect. Note, watermarking is discussed further in Section 4.7.

System Examples

Continuing from Section 2.4.1, Spark-Streaming[81] is an extension of the Spark framework which converts it into a micro-batch based streaming system. Within this Spark creates a 'streaming context' allowing it to connect to continuous data sources represented via a discretized stream (DStream). DStreams are an abstraction for an ever growing list of RDDs, where each RDD is a processing window micro-batch containing all tuples arriving within its allocated period. All of the transforms which may be called on standard RDDs may also be called on DStreams, with the function run on each contained batch independently. These, however, may be processed together via an additional layer of windowing on-top, which itself may be sliding or fixed. DStreams may also be joined using standard SQL join semantics (right, left, full-outer etc.) to create new DStreams. Finally, Spark-Streaming is fully compatible with other Spark extensions, notably Mllib[82], allowing machine learning algorithms to be run across the batches.

Apache Storm[83] takes a different approach to streaming, implementing a custom pipeline for each task, known as a topology. Topologies operate on top of a set of worker units spread across a cluster, running allocated logic/assignments contained within a set of 'Spouts' and 'Bolts'. A spout is Storm's unit for connecting to external data sources, starting the topology and pushing tuples to the first set of bolts. Bolts in turn are the general worker unit and may run any code on tuples they receive. This includes the transforms, aggregations and joins seen in Spark, but these may also run custom functions, maintain local state and can read and write from external sources if required. Topologies are then defined by specifying which streams should receive as input for each bolt (from the spouts or other bolt types). Bolts of the same type may be scaled horizontally to handle greater throughput, with different stream partitioning strategies ('groupings') available to split the work between them. Notably 'shuffle grouping' (round robin) to guarantee an equal workload and 'fields grouping' (key hashing) to ensure tuples with the same key are processed by the same bolt instance. Finally, bolts may be implemented using a windowing subclass, fully supporting event and processing time windows. Watermarks are automatically calculated from timestamps within the data and exactly-once semantics may be implemented via the Trident[84] API which sits on top of Storm.

Apache Flink[85] attempts to unify streaming and batch analysis, providing both over continuous data sources. At Flink's core is a distributed dataflow[80] engine which executes programmes built from stateful operators connected via data streams. These programmes are represented as a Directed Acyclic Graph (DAG), similar to the Storm topology, and are created by both the DataSet API for batch processing, and the DataStream API for stream processing. These stateful operators may consist of transforms, joins and windows as above, but can also be much more complex encompassing full algorithms. As their name suggests, state is explicitly incorporated into the API allowing local variables to be registered, checkpointed and updated with exactly-once semantics. The data streams connecting these may also distribute the data between producer and consumer in various patterns, for example broadcasts, merges and repartitioning. In both the batched and streaming context, Flink additionally allows iterative algorithms (directed cyclic graphs), managed by a Bulk Synchronous Parallel (BSP)[86] execution model. These may also be incremental[87], where the results of analysis on the prior version of the data are used as the starting point for the next execution, speeding up convergence. This is the basis for both the machine learning and graph libraries. Finally, once defined, DAGs are submitted to a 'JobManager' which interacts with a cluster controller such as YARN to distribute and paralyse the work across the cluster.

Streaming Graphs

Across these types of streaming system, graph algorithms may be computed, with implementations existing in both the real-time and micro-batching camps. The first of these implementation types are edge-centric stream[88] or semi-stream[89] algorithms which maintain minimal state/do not build a full graph, possibly due to memory constraints. In this instance the stream of tuples ingested each represent an edge in the graph and via one, or possibly several, passes over the data the metric of interest is extracted. These algorithms have many applications, such as motif mining[90], but are often approximate and slightly out of scope for this work.

The second manner in which graph algorithms may be applied is by building a full graph from the tuples ingested within a window and performing the algorithm upon it. From an implementation perspective, these algorithms are similar to those seen in the batched systems above, hence suffering from similar issues, but port over well. From an output perspective their results may be drastically different. In this instance, instead of incrementally performing analysis on the full aggregate graph as it changes throughout time, the user now has the ability to vary the size of the window and hence how far back in time the graph looks. The deltas between results are, therefore, able to extract new insights by comparing across time and window sizes. For example, [91] uses this technique to extract core groups and patterns in social interaction networks by looking at minute scale windows in comparison to the aggregate where these were obscured. Similarly, [92] computes windowed graphs over mobile telephone communication networks. Through smaller windows they were able to extract daily/weekly patterns, notably circadian rhythms in the populations and drastically different call patterns between weekdays and weekends. Through longer windows (months/years) the formation and evolution of communities could be seen instead.

Similar to deletions being included in the dataset, one unfortunate element of this analysis is that incremental algorithms are often not possible. This is because prior vertices and edges may no longer be present in the graph and for many metrics (notably label propagation and paths) the starting state may now be impossible to reach, likely leading to an incorrect result being returned. To enable analysis of the aggregate dynamic graph built from all tuples, the logical next step is to create the full graph within the streaming system of choice, possibly via an ever increasing window. However, streaming systems on the whole are not designed to work with largely interconnected data models such as a graph, as typical streaming tasks are a lot more independent. Nor are they designed to deal with ever increasing in-memory state (into the gigabytes). Hence, instead of utilising these as a base, alternative bespoke streaming graph processing systems have been developed to process large dynamic graphs, as discussed below in Section 2.5.

Whilst this is a general rule there are a few notable exceptions which are important to raise here. The first of these is Flink, as described above, which has this as a core feature. Secondly, one can imagine hybrid deployments where a platform like Kafka Streams[93] handles the ingestion and storage of graph data (either keeping it within broker logs or a distributed storage like HDFS) whilst another more appropriate platform (such as Hadoop or Spark) handles analysis. Thirdly, the work of Fernandez et al[94] attempts to tackle this issue for updating distributed data structures within machine learning algorithms, which similarly are often heavily interconnected and require large in-memory state.

In [94] the authors push back against the new programming paradigms required by the likes of Spark and Hadoop, instead opting to keep the imperative algorithmic design most developers are accustomed to. Within this, however, the user is required to annotate their data structures to specify what must be globally accessible, what may be partitioned and, if required, how partial state should be merged for the final results. These programs are then translated automatically into a 'stateful dataflow graph' which combines 'task elements', representing the analytical steps to be executed, with 'state elements' which encapsulate the state of the computation. Task elements may be assigned to multiple physical nodes for parallel execution, with each allocated a local state element which may be read and updated. Depending on the access patterns of these task elements the state may be cleanly partitioned requiring no cross machine synchronisation. Unfortunately in many instances access is arbitrary, meaning the state element must be 'fully' represented on each node. These may still be freely updated, but when a task element performs a read the nodes must synchronise to return the correct result. This is optimised via partial calculations/merges and addresses the issue of growing intertwined state well, but does ask a lot from the user to consider what is going on under the hood, which is otherwise hidden by other distributed programming paradigms.

2.5 Graph Processing Systems

Due to the issues graph models and algorithms bring in more generalised systems, an array of bespoke graph focused platforms have appeared in recent years, both from academia and industry. In this section these are split between graph databases and graph analytics platforms, with the major focus being on the latter. This split, whilst in no way a hard and fast rule, is based on a number of factors which can categorise systems into either camp. First and foremost, graph databases will store the data in a permanent format on disk ready for querying. A graph analytical tool on the other hand will read the data (bounded or unbounded) from an external source, build this into a graph to analyse and then relinquish the memory (destroying the model) once the algorithm has converged.

The second category of comparison is the workloads on which they focus. Graph databases tend to focus on Online Transactional Processing (OLTP) workloads with thousands of parallel requests looking at small portions of the data. Analytical platforms on the other hand have traditionally focused on singular offline analysis of the whole graph. Both, however, have recently been exploring Online Analytical Processing (OLAP) workloads, overlapping in the middle. These mirror the categorisation of algorithms by scope in Section 2.3. This difference is reflected in the chosen graph models, with databases having more complex models (LPGs and RDF), whilst analytical engines are often more bare bones i.e. weighted graphs.

A final notable difference is the manner in which the systems interact with their data or model. Databases often provide a declarative query language, abstracting away the underlying data structures and allowing the database engine to handle retrieval. Conversely, analytical platforms will provide an imperative API with few abstractions, allowing the user to work directly with the data structures for optimum performance, although sacrificing usability.

2.5.1 Graph Databases

Whilst graphs may be modelled in relational databases[95] this provides numerous challenges. Firstly, graphs do not always map directly to tables, although when they do very different access patterns are produced in comparison to traditional relational workloads, often containing irregular jumps instead of sequential reads. Secondly, traversals along edges are computed by repeatedly joining the table storing the vertices on itself, which is both inefficient and, for many-hop traversals, can require a query to be thousands of lines long, with each hop represented by a repeated block of code. To this end there are many graph concepts which are very difficult, if not impossible, to represent in relational languages such as SQL[96].

The goal of graph databases is, therefore, to provide access to the rich data associated with their entities, much like their relational counterparts, whilst also enabling fast traversals throughout the graph. This is done by taking a graph first approach, replacing the tables with vertices and creating new query languages which accommodate key graph concepts such as edges, paths and traversals. This solves the issues above as instead of performing table joins, vertices in an adjacency list may just follow pointers to their neighbours[97] and multi-hop traversals may simply be specified by a minimum and maximum hop number[24].

With this goal in mind, graph databases utilise more complex models to store both the structure of the network and the various properties of the entities within. RDF was originally the more popular choice for this because of its use for knowledge graphs like Stardog[98]. However, more modern and commercially successful graph databases have built around label property graphs, with companies such as Neo4j[34] and TigerGraph[99] spearheading the movement. Along with the more mature nature of knowledge graph/RDF systems these have a standard query language in SPARQL[100], whereas LPG systems tend to develop their own query language, for instance Neo4j's Cypher[24] and the Linked Data Benchmark Councils (LDBC) G-Core[101]. These are, however, beginning to converge, with an ISO standard graph query language (GQL) proposed by the LDBC, alongside collaborating companies and academics[102].

A tertiary mention should also be made here to the Tinkerpop stack[103] and its language, Gremlin, which is an open source framework upon which graph databases/systems may be built. Gremlin straddles the line between declarative and imperative language as it is fairly high level, but is used in conjunction with Java/Groovy in any practical application. This has been the base of numerous community projects such as JanusGraph[104], but is also popular for prototyping new ideas in this area. Excitingly in recent years this has been used for creating temporal graph databases, notably Chronograph[105] and Chronograph[49].

2.5.2 Linear Algebra Based Graph Analytical Platforms

Graph analytical platforms take a number of different forms depending on their intended use case and the resulting data-structure/programming paradigm. This form depends on the intended environment of deployment, with variations covering single nodes, distributed clusters, GPUs and, more recently, hardware designed specifically for graphs[106]. As a general rule, across all of these, analytical systems are less focused on querying vertex/edge attributes and more focused on properties of the overall graph structure. These global analytics may be online/dynamic where the user is interested in seeing how these properties change as new data arrives, as well as being time-aware (as discussed in Section 2.3.1) and are the key points of focus to be taken forward.

The first programming paradigm to look at in this area is the application of linear algebra for graph analytics, whereby graph traversals can be represented as a combination of matrix-vector multiplications. For example, paths of a given length l can be readily extracted from the lth power of a graph adjacency matrix. This is fully defined in the GraphBLAS standard[107] where BLAS (Basic Linear Algebra Subprograms) are the specifications for linear algebra subroutines implemented in many popular programming languages. GraphBLAS focuses on a subset of these, which are effective building blocks for implementing a wide range of graph algorithms when represented via a matrix. As explored in Section 2.2, these are simple models being at most directed and weighted, but can often drastically outperform standard algorithmic implementations[108]; a perfect fit for use cases where only the structure is of interest. This paradigm also allows for large datasets to be processed on modest hardware, updates to the graph including deletions and optimisations for handling both sparse and dense graphs. As GraphBLAS is only a standard, concrete examples of this are provided via the reference implementation SuiteSparse[109].

In conjunction with single threaded executions, much work has gone into parallelising these matrix operations to extract additional speed-up via multi-core CPUs[110] as well as distributed environments[111]. Building on top of this, GraphBLAS based algorithms have found even greater success within GPUs, with systems such as GraphBlast[112] porting these over and obtaining orders of magnitude faster convergence due to the level of parallelism which may be achieved. This does, however, come at a cost in that the barrier to entry is very high from a user's perspective. Not only can it be difficult to understand the underlying mathematics, but the onus is on the individual to ensure their algorithm is efficient. This is improving, but has meant other paradigms have been more successful in their commercial and academic adoption, most notably the vertex centric approach.

2.5.3 Vertex and Edge Centric Analytical Platforms

The vertex centric approach was introduced in Pregel[7], one of the cornerstones of distributed graph processing, introducing many of the key concepts and techniques recurrent throughout later systems[113]. It is built around the bulk synchronous parallel model [86] and champions the idea of 'thinking like a vertex', where each vertex within the graph is viewed as an autonomous entity, storing information about itself, its outgoing edges and executing user defined functions in iterative batches known as supersteps. During these supersteps, vertices may alter their stored state as well as communicating with each other via a messaging system, sending updates or requesting information. At the end of the superstep, these messages are collected at the specified destination vertices and processed in the next iteration. This manner of message sharing is intrinsic for parallel processing within Pregel, as multiple nodes cannot attempt to access data at the same time making it inherently free of deadlocks and race conditions. This has the additional benefit of removing the need for remote data reads, which can be exceedingly high latency operations and would need to occur after the vertices have finished making alterations.

By default, a Pregel graph is partitioned via a hash of the vertex IDs, splitting vertices amongst the workers, although a function can be provided to make use of data locality. Once the workers possess the data associated with their defined partitions they may initiate the first superstep. If a vertex receives no messages from the previous superstep (or its values do not change) it will vote to halt the process, no longer running calculations unless awoken by a message. When all vertices vote to halt, the process self-terminates and the final output is returned. This prevents unnecessary supersteps and reduces processing costs to a minimum as, if only 30% of the graph's vertices require updating, the remaining 70% will stay dormant. Pregel itself was never released for use, but an open source implementation Giraph [114] was built on top of Hadoop. Whilst this came with the iterative caveats of Hadoop, noted in Section 2.4.1, it was shown to compute PageRank[58] on a one trillion edge graph in under three minutes per iteration [115]. An impressive feat even with the strict conditions of the experiment.

Following Pregel, PowerGraph [9] introduces the GAS (Gather Apply Scatter) graph computation model to better define vertex centric operations. In the GAS model each iteration of a vertex task is split across three stages: gathering information from adjacent vertices; applying this information to the executing vertex; and scattering the new value to all neighbours. It notes that in many real world networks vertex degree follows a Powers Law distribution where a small number of 'star nodes' are connected to a large portion of the graph, e.g. a celebrity on Twitter who has millions of followers. Star nodes will often take much longer to compute their GAS functions, stalling the completion of supersteps in Pregel. To enable parallelism within these functions, PowerGraph proposes a variation to the GAS model whereby a 'vertex-programme' is split into four functions: gather; sum; apply and scatter. The execution of the gather and scatter phases are pushed from the vertex onto the edges, allowing each edge task to be distributed amongst the cluster and removing the linear scaling caused by the vertex degree. The gather and sum functions in this new model act as a MapReduce job, with the sum aggregating the partial results of the gather.

A common approach to achieve high data locality is to distribute vertices among the cluster's nodes, so there are as few edges spanning machines as possible, known as edge cut partitioning. In this situation, when two machines share an edge both will maintain a copy of its information. When an update happens this must, therefore, be synchronised across the graph generating network traffic. PowerGraph's distribution of individual vertex-programmes allows it to instead split vertices (vertex cut partitioning) and span them across nodes in the cluster. In this model each machine assigned to a vertex is given an even share of its edges to store and process and, therefore, updates to these do not need to be synchronised. Synchronisation is shifted from edges to vertices where a much clearer IO/parallelism trade-off can be drawn, the amount of machines assigned to a vertex depending on its degree. Once assigned to the vertex a master node will be elected, receiving partial results from the 'mirrors', running the sum/apply functions and propagating the changes.

GraphLab[8], the extension to PowerGraph, explores asynchronous alternatives to the BSP model. Two design patterns are discussed, namely DAGs and the systolic abstraction [116]. A systolic system is a network of independently executing worker nodes, extending upon the pipeline model to allow iterative and cyclic computation. This is synonymous with the Pregel architecture, allowing for rich computational dependencies, but forces computation to be decomposed into small atomic components with limited communication. GraphLab expands further on this with its shared memory 'data graph', which encodes the computational structure in a similar way to the systolic model, but allows the user to allocate data blocks to vertices which can be accessed concurrently without superstep synchronisation. This permits vertices to execute independently of each other, reading and editing any data within their 'scope', where the scope is defined as the data within themselves, their neighbours and the edges in-between Vertices also have access to a 'shared data table', supporting a global state.

This introduces the possibility of race conditions if the scope of two executing vertices were to overlap. To prevent this occurring, GraphLab maintains sequential consistency via a choice of execution models: full consistency, which ensures the full scope of an executing vertex is protected, thus only vertices with no common neighbours may execute in parallel; edge consistency, removing the protection on information stored within neighbours, allowing non-adjacent vertices to execute in parallel; and vertex consistency, protecting only the information within an executing vertex, allowing all vertices to execute in parallel. The choice of model depends on the level of information required by the algorithm and can heavily affect the throughput of the system. To implement an algorithm, the user must create an update function to run on the vertices. These can edit the data in the vertices scope and have read only access to the shared data table. To enable complex algorithms to be executed, any number of these functions may be running at a given time and are dynamically controlled by the GraphLab's scheduler. The scheduler maintains a distributed ordered list of tasks (function plus vertex) which require execution. These can be handled in a simplistic manner utilising static schedulers, such as synchronous or round-robin, or alternatively can be fully dynamic where the update functions can add to and reorder the task list at run time. Furthermore, the user may even create multiple sets of tasks to execute in a specified sequence.

Moving away from the distributed environment, Graphchi[117] looks at how the asynchronous GraphLab style of processing may be computed on very large graphs, whilst only utilising a single personal computer. This is a disk-based system where the only requirement is each vertex and its edges must be individually able to fit into memory. To process these large graphs, Graphchi introduces the Parallel Sliding Window (PSW) model. This stores the graph in a CSR format on disk, which is then split up into 'intervals' or sequential sets of vertices which are a shard of the full graph. PSW operates by sliding across this CSR, loading one interval at a time fully into memory, denoted as the memory shard. Once an interval has been loaded the user defined update functions (as specified in GraphLab) may be run on each vertex in parallel. However, to remove race conditions, vertices which have edges with both the source and destination in the same shard are labelled critical and updated sequentially. Vertices where this is not the case may operate freely. Once the update functions have concluded, the new values for each edge and vertex are written back to disk so they may be available for the next execution. As the update functions directly modify the data blocks storing the edges, these are simply written back to disk, fully overwriting the previous instance of the memory shard. PSW then slides further through the CSR to load the next interval. PSW fits well with the asynchronous model as it is effectively the same as only the vertices within the loaded subgraph being scheduled to run, if compared to GraphLab, and will produce the same result.

Graphchi also allows the graph to be updated as it is processing, adding and removing edges. This is done by maintaining an edge-buffer for each logical part of a shard. New ingested edges are stored here until their interval is next loaded from disk, at which point the new edges will be included. If, however, a buffer becomes too big it will be written to disk, merging it with the edges present in that shard. If either of these operations make a shard too big to fit into memory it will be split in half, creating two new intervals. For removals, edges are flagged to be ignored during the next execution of their interval and then simply not written back to disk when the next memory shard is loaded. Finally, updates are hidden from the perspective of the user, with new edges only appearing in the next execution of an interval. This is done to remove the complexity of having to write functions for vertices which are changing in parallel with execution.

Vertex Centric Industry Take up

As well as existing independently, these programming paradigms have had a lasting impact on the Graph Analytics ecosystem, most notably being used as the basis for graph computation within Apache Spark and Apache Flink. Within Spark this began with the GraphX[74] library which allowed the user to create a labelled graph built out of a 'vertexRDD' and 'edgeRDD'. A Pregel engine was then established allowing users to specify their own GAS functions which run as transforms and actions on the RDDs. This, however, quickly runs out of memory for larger datasets, because of the many joins/groupbys and did not get the same support from the developers as other Spark libraries. GraphX was, therefore, phased out in favour of a new graph processing extension for Spark, GraphFrames [75]. GraphFrames was built on top of Spark SQL [118] utilising DataFrames in the same manner GraphX employs RDDs. Two DataFrames, one for vertices and one for edges, are combined to make a GraphFrame with both permitted to contain zero or more attributes akin to the property graph model. This supported GAS functionality, a query language similar to Cypher, and was integrated with Mllib as DataFrames are its base data structure. Unfortunately this has recently been discontinued as Spark has moved away from graph processing.

Gelly, the equivalent library for Apache Flink, has fared better being a base module of the framework and still supported. This similarly utilises a GAS based Pregel model, allowing users to split on vertices, edges or a hybrid approach. Algorithms may be designed with both the DataSet API and DataStream API, meaning Gelly can perform graph analytics on bounded data as well as incrementally update the results if it is unbounded. Work has recently been carried out to extend Gelly to perform temporal graph analytics by labelling the edges with creation/deletion times and adding time-aware functionality. This was progressing under the Tink[119] library, but development seems to have paused since initial publication.

2.5.4 Streaming Graph Analytics

As with Gelly, many academic graph platforms have begun to provide analytics of time-evolving graphs over unbounded data sources. GraphTau[37], built on-top of Apache Spark, blends the ideas of GraphX and Spark Streaming to provide 'Discretized Graph Streams'; treating dynamic graphs as a stream of graph snapshots at regular intervals. Updates to the graph (additions, deletions and label updates) are batched in a 'DeltaDStream' which may be converted into a 'GraphStream', combining all prior updates to build the graph at each time increment. BSP algorithms may be set running as micro-batches from the initial snapshot, but the algorithm may additionally be moved onto new snapshots as they become available, even if it has yet to converge. This is introduced as Pause-Shift-Resume (PSR) where the supersteps are paused on the current snapshot, the metadata shifted to the new one and the algorithm resumed. This works on the basis that the user is only interested on the latest state/reducing the staleness of the result and that, for many algorithms (such as PageRank), the converged answer will not be very different from the result archived from running the algorithm from scratch on the newer snapshot. For those that do not fit this, or are affected by deletions (as discussed in Section 2.4.2) GraphTau provides online rectification, which attempts to roll back the state to a point where the deleted entity was yet to have an affect. This does, however, require vertices to maintain their computational state for all snapshots.

Moving to fully bespoke systems, LLAMA[14] presents a multi-version CSR graph representation storing dynamism as a series of snapshots. Within this the graph exists as a combination of a ‘Large Multiversioned Array’ (LAMA), storing the vertices for all versions of the graph, alongside multiple delta edge arrays, one for each snapshot. As the edge arrays are deltas, a vertex’s full adjacency list is spread across snapshots in ‘adjacency list fragments’ which must be combined to form the full graph for analysis. This does, however, periodically trigger without analysis, building a new base snapshot and bounding the overhead. Each property associated with either edges or vertices is independently stored as a separate LAMA, containing all versions of the associated value for each of the entities. As LLAMA is a non-distributed platform, it allows implemented algorithms to simply loop through the vertices and their edges, which will occur in parallel, or directly access vertices by ID. It also provides a GAS based API for when the graph is larger than the available memory. Finally, as all prior graph versions are saved, it is noted that the snapshots may be analysed ‘temporally’, although this is from the perspective of the evolution of a metric, not time-aware analysis.

In contrast to LLAMAs snapshot approach, Stinger[120] views graphs as an infinite stream of edge insertions, deletions, and updates, providing a dynamic CSR based data structure to store these changes as they arrive. Instead of a contiguous edge array, edges are split into equal sized blocks which are joined together in a linked list, growing and shrinking as new updates arrive. This is a weighted graph where edge tuples consist of the neighbour ID, type, weight and a timestamp of when the last update occurred. Several indexing optimisations are then made, such as having only edges of the same type in an edge block to increase sequential reads. Updates are handled individually for scale-free (Powers law) graphs to avoid star-nodes reducing throughput and in batches when this is not the case. Interestingly, for batches Stinger runs all deletions first to try to make room in existing blocks, compressing the data as best as possible. However, this seems to create an artificial ordering and may end up creating different graphs depending on how the batches are grouped across the stream. For analysis, Stinger provides macros for looping through vertices and their edges, accessing weights and type labels. All vertices may be accessed in parallel through these providing fast traversals, although it is not mentioned if this may happen alongside ingestion.

Following Stinger, a subsequent version DStinger[121] was released, moving from a single machine to a distributed environment. DStinger consists of a primary/replica architecture where the primary nodes delegate computational tasks to the replicas. The replicas are then responsible for maintaining a subset (partition) of the vertex array and the associated edge blocks, updating these and completing analysis. The graph is edge cut with the vertices partitioned based upon a hashing algorithm, claiming this is the fastest approach with the simplest logic, whilst complex partitioning to minimise edge-cuts is often fruitless[122]. Finally, update synchronisation and computation messages between replicas are handled autonomously via DStinger’s Message Passing Interface (MPI). All messages between pairs of replicas are aggregated together into one message, minimising network traffic, and processed within ongoing update batches for optimum throughput.

Kineograph[10] is a distributed graph management system consisting of four components:

graph nodes; ingest nodes; a global progress table; and a ‘snapshooter’. Graph nodes act as a distributed key value store, indexing on vertex ID. The value for these keys is split between the structural metadata for the vertex (edges/associated information) and application data for the algorithms incrementally run on the graph. Updates read into Kineograph are managed by one of several ingest nodes, converting each tuple into a set of graph alterations known as a transaction. This is assigned a sequence number and sent to all graph nodes storing an affected vertex. Once these all confirm the update has been received, the ingest node reports a successful transaction to the global progress table. This progress table stores a vector clock where each number represents the latest fully committed transaction from each ingest node. Periodically the ‘snapshooter’ will request the graph nodes to commit their stored transactions and create a snapshot which may be executed upon. The graph nodes use the global vector clock to decide the end of the ‘epoch’ and what transactions are within it. Transactions that have occurred before the end of the epoch are committed in a deterministic order, whilst the remaining are ignored until the next snapshot is taken. This allows the graph to be consistent across partitions, but the order in which it implements is completely artificial, as with Stinger. Event time and casual relationships are not taken into account, instead establishing an order by running sequentially through the list of ingest nodes. During this time new transactions are still being processed by the ingest nodes and sent to graph nodes in preparation for the following snapshots.

Running in parallel with ingestion, algorithmic execution in Kineograph is a more relaxed version of GraphLab, such that neighbours may access each others information in parallel. Writing to neighbours is restricted, but this alone does not guarantee sequential consistency, although it is claimed from their experiments to produce acceptable results. Instead of the GAS model, Kineograph functions implement either the push or pull model of communication. The push model has the user function calculate its new value and send this out along its outgoing edges. As Kineograph supports incremental algorithms, vertices may also send their incremental change allowing the receiving vertex to decide if the change is large enough to trigger a response. Alternatively in the pull model, an awoken vertex function requests the information it requires from its neighbours to calculate its new value and propagate. This reduces overall network traffic, but is no longer asynchronous IO.

Instead of only building coarse snapshots, where the update order between is lost, Weaver[11] introduces a fully online graph model which may be queried and updated in parallel. This is split between ‘shard servers’ and the ‘timeline coordinator’. Shard servers are analogous to graph nodes within Kineograph, being assigned partitions (shards) of the overall graph, executing updates and performing the requested graph analysis. The timeline coordinator provides ‘refinable timestamps’ via a set of ‘gatekeeper’ servers and the ‘timeline oracle’. Gatekeepers contain a vector clock array, incrementing their own counter when a new transaction is received and sending it to the affected shards. Gatekeepers periodically exchange their vector clocks, establishing a ‘happened-before’ partial order for updates. Overlapping updates received concurrently at different gatekeepers within the defined period then require the timeline oracle to ‘refine’ the order and decide which happened first. Unfortunately, whilst the oracle is meant to be lightweight when intervening, in practice it has been shown to bottleneck the system[1].

To perform analysis on the graph, Weaver provides read-only 'node programmes' which traverse the graph via GAS. If a node programme wishes to alter the properties of a vertex or edge it must specify this in the form of a transaction and resubmit it into the system. As the graph is always updating, node programmes must somehow execute on a consistent snapshot. In order for this to be enabled without blocking subsequent updates Weaver maintains a multi-version (temporal) graph, storing all previous incarnations of graph entities along with the times at which alterations occurred. For example, if a vertex is deleted by a transaction, the deletion is inserted into the object representing the vertex and this 'history' is retained in-memory. Node programmes are then assigned timestamps which the processing shard servers use to generate a 'logically consistent' snapshot; a view of the graph at the point of programme submission. This is done by comparing the timestamp of node programme with the history of the vertices it wishes to process on, requesting the oracle to establish an order if one does not exist. As node programmes may be distributed across multiple shard servers, and there may be some variance in the time at which they arrive, node programmes are blocked until all preceding and concurrent transactions have finished committing to ensure all sections execute on the same snapshot. Interestingly, this history is not used for time-aware queries and is only kept up until the timestamp of the oldest ongoing programme, at which point it is cleared.

Unlike the hash partitioning used by all previous systems, Weaver also discusses (although not implements) the idea of a streaming graph partitioning algorithm[123] which dynamically collocates vertices with the majority of its neighbours, minimising edge cuts. This is fully fleshed out in Vaquero et al[124] which provides a distributed graph processing engine allowing vertices to move around the cluster as new updates arrive. As with DStinger, this is split into a master/-worker architecture where the master provides the analysis API and the workers store partitions of the graph, performing updates and analysis. Within this, once the initial graph is loaded and partitioned (based on any strategy) new vertices and edges will begin to arrive. A background label propagation job is then periodically set running, with the vertices adopting the label of the majority of their neighbours. Once this has converged, the vertices may decide if they should stay in their current partition or move to another. This is a greedy migration heuristic which requires no global state, allowing it to scale freely with the number of partitions. Some limits are, however, put in place where the number of vertices in any partition is capped and vertices will have a preference to stay put to minimise overhead.

Analysis in Vaquero et al is completed in a Pregel BSP style with vertices allowed to migrate between supersteps. Interestingly, similar to the Pause-Shift-Resume model, new updates coming into the system are included in the next superstep if the algorithm is resilient to this i.e. PageRank. Alternatively, updates are buffered until the algorithm has converged to stop the wrong result from being returned. To allow workers to communicate for vertex messaging, edge synchronisation and system metadata (capacity etc.), a messaging queueing system is used

(RabbitMQ[125]). This provides an asynchronous publish subscribe model allowing workers to ingest messages without becoming overwhelmed. Finally, as vertices are moving around and there is no global look-up table each worker maintains a 'Vertex Locator' which tracks the location of vertices allowing messages to be routed to them.

2.5.5 Temporal Graph Analytics

Whilst it is important to keep up-to-date with the current state of a graph, ImmortalGraph [12] focuses on how to process the evolution as a whole, although not online itself, executing on static repositories. ImmortalGraph introduces three contributions, the first of which being how to efficiently store a graph's full history on disk without diminishing query performance. Snapshots cannot be used here as the update order between these is lost and, whilst a full update log does not lose any information, queries take longer the more of the log has to be ingested. To compromise between these ImmortalGraph introduces 'snapshot groups'. A snapshot group is assigned a time range, maintaining a full snapshot for the start of the window and an update log containing all changes up until the end. If an incoming query requires the graph at a point within the specified range, the update log can be read into the base snapshot until this point is reached. Secondly, ImmortalGraph elects four quadrants which queries may fall under with local/global in the spacial dimension and time point/time range in the temporal dimension. It then discusses how the memory layout of a snapshot can heavily affect the performance of a query depending on where it falls within these quadrants. Snapshot groups may be stored with high spacial-locality, placing neighbours within the same snapshot in close proximity, or with high temporal-locality, where all versions of a vertex will be stored consecutively, providing access to the full history of a vertex with one sequential read. As neither option is optimal in all cases, ImmortalGraph creates multiple copies of stored snapshots, optimising each version for different access patterns.

ImmortalGraph uses an iterative GAS model for analysis, allowing snapshots to be processed sequentially or independently in parallel. However, for temporal queries the 'Locality-Aware Batch Scheduling (LABS)' model is introduced, batching the execution of each vertex together across all snapshots. In this model, for N snapshots, the desired algorithm is first executed to completion on the earliest snapshot in the graph's history ($Snapshot_0$). The output of this is then utilised as the starting point for processing the remaining snapshots in the history ($Snapshot_1$ through $Snapshot_{N-1}$). This is completed by each vertex sequentially accessing its different states within $Snapshot_1 \rightarrow Snapshot_{N-1}$ and computing the assigned algorithm on each. These can either be fully parallelised or organised into supersteps for synchronous algorithms. The user can additionally choose between an edge-centric or vertex-centric processing model and push or pull communications. Finally, a later version of ImmortalGraph was released under the name Chronos[126]. This provided a distributed version of LABS, but based on whole snapshots being placed on different machines, not on an actual partitioned graph.

Version Traveller [13] is a graph processing engine which tackles a similar problem to ImmortalGraph, extracting temporal analytics on multi-version graphs. Instead of the LABS approach, Version Traveller focuses on efficient 'arbitrary local version switching' between snapshots of a graph. This is 'local' as sequentially processed snapshots are often similar in structure and values, but also 'arbitrary' as there may be no predetermined order for snapshot execution, i.e. they do not have to be chronological. Previous systems have no awareness of the next graph version, often dropping the current graph from memory and fully loading the next graph from disk. Version Traveller computes the next snapshot from the current by combining it with a delta. This is not an entirely new approach, but prior multi-version processing systems (such as LLAMA) struggle with arbitrary version switching as their deltas are set up to be sequential. To improve on this Version Traveller introduced a hybrid graph model, combining CSRs with a Vector of Vectors (VoVs), which allows neighbour lists to be edited independently. In this model a 'root' CSR neighbour array is created when the first version of the graph is loaded and this then remains constant. When a new version is to be loaded, it is stored as a set of 'vertex delta entries' containing the neighbourhood modifications for each vertex. The hybrid CSRs pointer array then indicates if the neighbourhood of a vertex is stored in the neighbour array or the delta for this snapshot, with Version Traveller reading from either during analysis. When another arbitrary version is to be loaded, the CSR reverts to the base neighbourhood and the process restarts.

Chronograph [15], similar to Weaver, provides a dynamic graph model which allows concurrent local modifications whilst maintaining a consistent global view. Various programming paradigms may then execute on top of this, performing online approximations on the live dynamic graph and offline batch processing on consistent snapshots. Chronograph also tracks the evolution of the graph, allowing retroactive snapshots to be generated and temporal analysis to be performed. To provide asynchronous and scalable processing, Chronograph builds on top of the actor model [127]. Within this model each vertex is considered an 'actor' which executes actions upon its local state based on the information of incoming messages. Actors are completely reactive, computing only when they receive a message and have no access to data outside their own internal state. To get information into the system a special class of I/O vertices are utilised. These are part of the graph topology, but sit at the boundary between the internal regular vertices and external IO operators, feeding in data via the messaging system. Upon receiving this data the regular vertices may perform some user defined function, update their local state, communicate with neighbours or alter the graph topology by creating new vertices or outgoing edges. Eventually this will yield some approximate results which may be sent to 'output vertices', feeding back to the external data source.

Chronograph maintains the history of the graph via event sourcing[128], appending all incoming events and the graph alterations they generate into an event log. All previous incarnations of the graph may then be rebuilt by reapplying the stored events from the beginning up until the desired point. To remove the bottlenecks of global ordering Chronograph allows each vertex to store its own event log and that of its outgoing edges. A global log is then only required to maintain vertex creation and deletion order for when the full graph is recreated. However, whilst this may scale better, maintaining logs in this manner only provides eventual consistency, not a globally consistent view. To provide such a view, each vertex additionally contains a vector clock which is incremented whenever a message is processed. The vector clock is attached to all outgoing messages, allowing the receiving vertex to track the state of its neighbours at their last point of contact. By persisting this information alongside each logged composite event 'casually consistent' snapshots may be created. This may not produce the exact same snapshot as a sequentially consistent log, but does guarantee no vertex within the snapshot will have received a message which the source has yet to send. This method can be used to produce 'retroactive snapshots' which provide a view of a previous point in the history of the graph, or 'live snapshots' which capture the graph's state with minimum staleness.

Greycat[129], the final work in this area, provides a full temporal property graph which is backed up via an underlying key-value store. Interestingly, within this model vertices are just 'conceptual identifiers' distinguished from their state which exists on a timeline of 'state chunks' representing each historic modification. A set of functions are defined to *Read* the state of a vertex at a given time, returning the closest anterior chunk on the timeline, *Insert* new chunks into the timeline, changing the state of a vertex at a chosen time, and *Remove* entities, inserting a new 'null' chunk into the timeline in order that a *Read* will see the vertex as absent from that point forwards. State chunks contain all attributes for the node and its outgoing edges and also link to other state chunks representing the neighbours of a vertex at the latest time from its perspective; similar to the flattened temporal graph model discussed in Section 2.2.2.

To analyse this graph Greycat provides an API similar to Tinkerpop, within which the user starts at a state chunk of interest (returned from a read) and may then traverse forward following the links to neighbouring state chunks. These paths are intrinsically time-aware, meaning temporal algorithms are the assumed standard. This method additionally allows Greycat to lazily load the state of the graph as new chunks are requested instead of having to read it all at the start of a query. Finally, as some nodes may have millions of updates/chunks to manage, these are indexed utilising red-black trees; noted as the most adopted structure for non-monotonic time-series. These are, however, additionally modified to allow a coarse higher level indexing, meaning only portions of the tree have to be checked when a new read takes place.

Whilst there has clearly been several successful attempts to include the evolving history of a graph as a component, prior systems do not appear to have fully realised the potential of maintaining this history in-memory. By doing so, many of the issues they faced could have perhaps been resolved. For instance, if updates were to arrive out of order, they could still be inserted correctly without expensive synchronisation steps or a centralised arbiter. This can also ensure a correct event time based ordering of updates unlike the many artificial orders seen above. Finally, whist databases are beginning to explore time-aware queries, many temporal analytics platforms are still focused on deltas between discrete points in time. Those that do explore this such as Greycat are focused more on traversals and less so on global time-aware analytics.

2.6 Summary

In summary this chapter has given a brief overview of the many areas underpinning the ultimate goal of this work, to provide scalable online analytics of temporal graphs. To this end we initially looked at what a graph consists of, the different expansions which have been proposed and how these materialise within real world implementations. This was expanded with the concept of dynamic graphs which evolve to reflect the changes seen in the underlying dataset. These concepts were then brought together in the temporal graph model which stores the full state of graph components across all points in time, disambiguating this from dynamic/time-evolving graphs which only maintain the most up-to-date version. Following the graph concepts introduction we looked at what type of graph algorithms exist, grouping these by the structural scope which they cover; from local singular vertex queries to global analytics across all entities. We then explored how algorithms across all structural scopes could be expanded to bring new insights when given access to the time dimension present in temporal graphs.

Following on we looked at the big data processing platforms which have taken over as the analytical tools of choice as datasets have increased past the capacity of a singular machine. We split these between batched processing and stream processing systems, investigating notable examples in both camps. We discussed how all systems mentioned may perform graph algorithms, highlighting that the interconnectivity of a graph does not sit well with the data structures utilised in general big data platforms and why this has lead to more bespoke solutions. Lastly we highlighted some key elements of streaming systems which would be important within our own implementations, notably windowing and watermarking.

Finally, we explored the plethora of graph-specific platforms, both commercially and within the literature, categorising these under graph databases and graph analytical systems. This split was based on the manner in which the two categories handle/store data, the sort of workloads they prioritise and the elements of the graph which are primarily involved (structure vs properties). Of these our focus was on the analytical systems, as they are the basis for the platform to be developed within this work. Within this category we found distributed systems, both batched and streaming based covering static, dynamic and temporal graph use cases. We highlighted that whilst static and dynamic graphs have found strong adaption, few systems make full use of updates being ingested and the temporal graph they may form. Those that do, have key flaws in their implementations and provide ample space to explore and improve upon.

Chapter 3

Temporal Graph Model

3.1 Introduction

The first step to providing a system which maintains an updatable, in-memory, temporal graph is to formalise the model. In this chapter such a model is presented, expanding on those seen above in Chapter 2 to provide a full history of structural and property changes within the graph. The semantics for modifying the state of this graph via a stream of updates are then established, encompassing the addition/removal of vertices and edges, as well as updating their associated properties. This includes the preconditions and effect of each update type, ensuring the consistency of the graph is never broken, such as an edge without a source vertex.

To explore how the network has evolved and to view the graph as it would have looked at any given point in its lifetime (from inception through to the most recent update) the semantics for flattening the temporal graph into a traditional graph are discussed. These flattenings may also be created with the additional aspect of a window, looking back only a set temporal depth from the decided flattening time to assist in the investigation of short and long term patterns.

Finally, as with all the above, this initial model is defined from an abstract perspective and does not consider the challenges that an instantiation of it would bring. Pertinent to this work is the challenge of distributing the model over a set of real machines. In this environment the graph must be partitioned, meaning state has to be split between machines, and this shared state must then be synchronised whenever it is modified. In this instance both the updates coming from the source and synchronisation messages between partitions may arrive out of order, breaking many of the constraints and preconditions established for this first model. To alleviate this the model and its update semantics are redefined with this distributed environment in mind, specifying what a graph partition consists of and how out of order updates may be handled.

3.1.1 Chapter Roadmap

Section 3.2: Undistributed Model The undistributed model section introduces all of the concepts of our temporal graph, defining the components it contains, their semantics and how their history is mapped on top.

Section 3.3: Undistributed Update Semantics The undistributed update semantics section then specifies how the temporal graph model may be updated, defining an update stream and semantics for additions, deletions and property updates.

Section 3.4: Flattening the Temporal Graph Here we define the semantics for 'graph flattening'. This extracts a non-temporal graph representation from the model, given some time parameters, simplifying exploration/analysis of the temporal graph.

Section 3.5: Challenges of Distribution and Implementation Drawing from the discussion in Chapter 2 we next discuss the issues faced when attempting to implement graph analysis in a scalable environment; touching topics of partitioning, distributed streams and state management.

Section 3.6: Distributed Temporal Graph Model With these challenges in mind we redefine the undistributed temporal graph model to include the concept of partitions which contain portions of the graph. These partitions may then be distributed.

Section 3.7: Distributed Update Semantics In line with this redefinition the update semantics are then also expanded to better fit the intended distributed environment; allowing partitions of the graph to communicate and providing policy for handling out-of-order updates.

3.2 Undistributed Model

A static graph G consists of a pair G=$\langle V, E\rangle$ where V is the set of all vertices V=$\{v_1, v_2, \ldots, v_n\}$ and E is the set of all edges E=$\{\langle v_i, v_j\rangle, \langle v_k, v_l\rangle, \ldots, \langle v_m, v_n\rangle\}$. 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 ($\{\langle v_j, v_k\rangle, \langle v_j, v_k\rangle\}$). Both vertices and edges are referred to as graph entities $Y = V \cup E$. To store metadata for each vertex and edge we define a set of m keys $K = \{k_1, k_2, \ldots, k_m\}$ and define properties of entities using key value pairs (where if not set, the value is null). The property for key k_i on entity y is then defined 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 lifetime of the graph; $t_0 \leq t \leq t_n$. 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 $t_1, t_2, \ldots, t_n$. 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 $t_i \leq t$; 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 $H^y = \{\langle t_i, \mathit{created}\rangle, \langle t_j, \mathit{deleted}\rangle, \ldots, \langle t_l, \mathit{created}\rangle\}$, 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 $t_0 \leq t < t' \leq t_n$. Note, when querying the state of an entity at time t, either within the full history or a subset, if no such change point exists anterior to this time (i.e. $\forall \langle t_m, \mathit{state}_m \rangle \in H^y, t < t_m$) then $H^y(t) = \mathit{removed}$ by default. This is because t is either a point in time prior to the inception of y or y has no updates in the time range of interest.

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 be combined to create the overall history of the graph.

3.2.1 Example Temporal Graph

Figure 3.1 provides a toy network with two nodes a and b which share two edges $\langle a, b \rangle$ and $\langle b, a \rangle$. These can be seen in the top left of the Figure within V_T and E_T respectively or via the graph representation on the right. To provide an application of this model we may once again refer to the social network example introduced in Section 2.2, where these two nodes can be

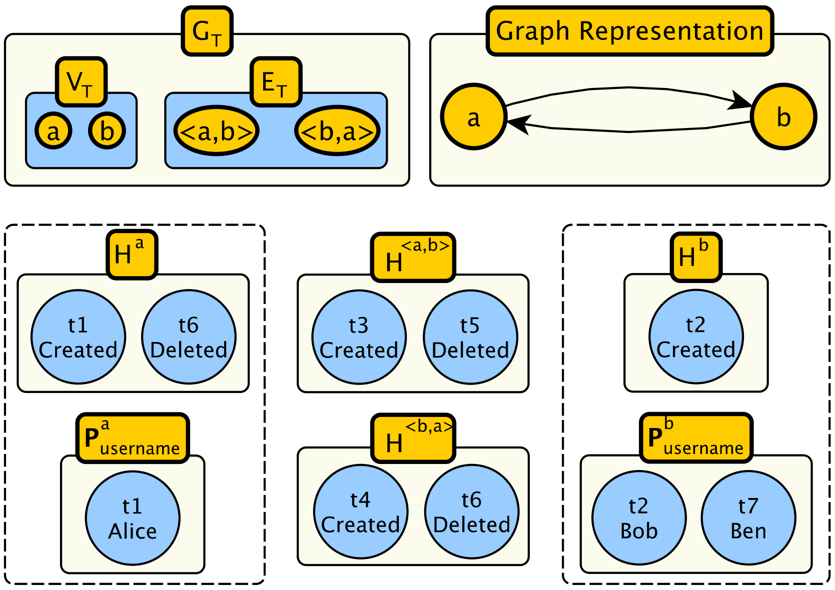


Figure 3.1: Example undistributed temporal graph based upon two users in a social network following and unfollowing each other.

imagined as two users within a twitter style social network and are both following each other. In the bottom half of this figure we can see the structural history of these entities alongside their property histories. By exploring these we can establish the full story of their interaction. At t_1 vertex a joined the network with the property 'username' set to the value 'Alice'. This was followed at time t_2 by vertex b with username 'Bob'. At time t_3 user Alice followed Bob ($\langle a, b\rangle$ *created*), prompting Bob to follow back at t_4 ($\langle b, a\rangle$ *created*). At time t_5 Alice unfollowed Bob, causing $\langle a, b\rangle$ to be removed and a *deleted* state to be inserted into its history. Alice then left the network at time t_6, appending a *deleted* state into vertex a and all remaining connections ($\langle b, a\rangle$). Finally, Bob updated his username to 'Ben' at time t_7 which was then appended into the history of this property.

3.3 Undistributed Update Semantics

Updates to this temporal graph come in the form of an unbound stream of events S=$\{\langle t_1, a_1\rangle, \langle t_2, a_2\rangle, \ldots, \langle t_n, a_n\rangle\}$, where each event depicts an action (see Table 3.1) 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 $\{\langle t_1, a_1\rangle, \langle t_2, a_2\rangle, \ldots, \langle t_i, a_i\rangle\}$.

Table 3.1: 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	$p_i^y := p_i^y \cup \{\langle t_{n+1}, value_i \rangle\}$

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 occurs 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 $\langle t_{n+1}, a_{n+1} \rangle$ 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 3.1. An example stream of updates which once ingested would create the temporal graph seen in Figure 3.1 can be seen in Figure 3.2.

3.3.1 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. 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

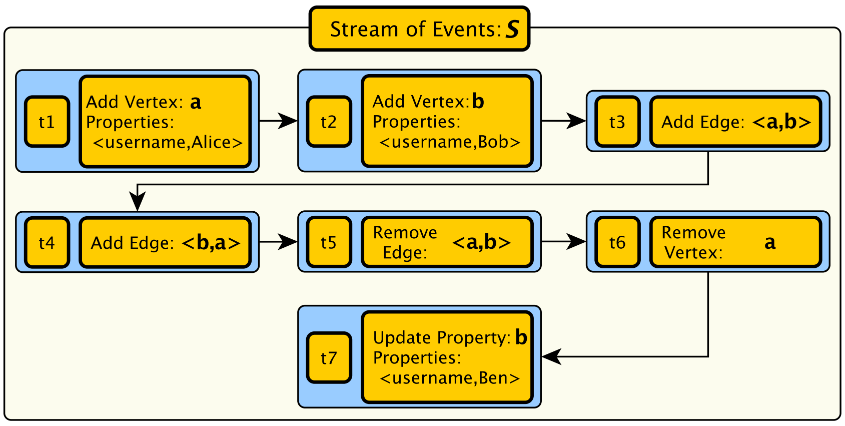


Figure 3.2: Example event stream which would create the temporal graph seen in Figure 3.1

the same two vertices (creating a multi-graph). If the latter was desired this would have to be managed via associated edge properties.

3.3.2 Entity Removal

For an edge to be eligible for removal it must first be present within the graph. If it is present and a removal performed, 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.

3.3.3 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 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.

3.4 Flattening the Temporal Graph

A system which implements the temporal graph model over a stream of updates may inspect how a static graph would have looked at any chosen point in time t_i. We define this as a *graph flattening* and t_i as the *flattening_end* or latest time point. A *graph flattening* consists of the graph $G(t_i)=\langle V(t_i), E(t_i)\rangle$, where $V(t_i)$ and $E(t_i)$ contain all vertices and edges present at time t_i. The value of any properties associated with these present entities is then considered to be the closest anterior value at t_i i.e. $p_x^y(t_i)$ for a given entity y and property x. This may be $\emptyset$ if the property was added after t_i, in which case the entity would not be attributed with this property for the flattening. We assert that this is the same graph $G(t_i)$ which may be created from the stream $S = \{\langle t_1, a_1\rangle, \langle t_2, a_2\rangle, \ldots, \langle t_i, a_i\rangle\}$ as discussed above, but is extracted from the structural and property histories within the temporal graph instead of having to reingest said stream up until t_i.

To explore the correctness of this we may consider that when ingesting a stream of updates into a non-temporal graph the final state of each entity is not the aggregation of every prior update which has affected it, but the most recent change to its state and the most recent value set for each of its properties, as all prior values are overwritten. Similarly, when each entity is viewed through the lens of a flattening within the temporal graph, only the most recent state (with respect to the chosen time) is checked, giving no heed to anything prior or after this. The culmination of doing this for all entities in the temporal graph would, therefore, return a graph with the same vertices and edges and same property values as if it had been built directly from the underlying update stream.

Next we define w as a set temporal depth to look back from the *flattening_end* t_i. Here, rather than considering all updates since the inception of the stream (t_0) to t_i, we only consider updates within the stream strictly after $t_i - w$ and up to and including t_i. We define this as a *windowed graph flattening* where w is the window size and the earliest time after the cutoff ($t_i - w$) is denoted as the *flattening_start*. This collapses all the updates within the observation period into a singular graph $G(t_i, w)=\langle V(t_i, w), E(t_i, w)\rangle$, where $V(t_i, w) \subseteq V(t_i)$ and $E(t_i, w) \subseteq E(t_i)$ such that $\forall y \in Y(t_i, w), H^y((t_i - w), t_i) = \textit{created}$. Thus, an edge is only included in a *windowed graph flattening* if its anterior update to *flattening_end* is an addition and its associated timestamp is between *flattening_start* and *flattening_end*. A vertex on the other hand will be included if itself or any associated edge meets this requirement. This expansion for vertices simply ensures that there are no hanging edges within a flattening. For example in Figure 3.3, vertex A and B are not explicitly updated within the window period which without this expansion would require them

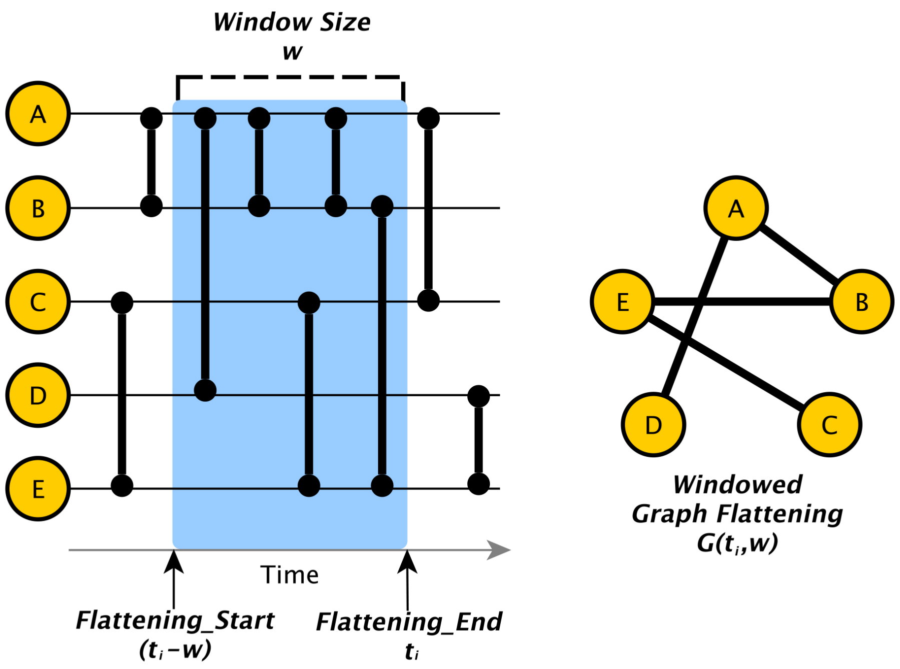


Figure 3.3: Left - Example stream of edge add updates with a window imposed over them. Right - Windowed graph flattening derived from the updates, included within this window.

to be filtered. However, the edge $\langle A, B\rangle$ is added twice and would, therefore, be included in the flattening; leaving $\langle A, B\rangle$ hanging. As such, the source and destination of an accepted edge are always included, maintaining graph integrity. Finally, in the instance of a windowed flattening, whilst the structural history is filtered, the property history is unaffected i.e. $p^y(t) = p^y(t, w)$. This is because property values do not change unless explicitly updated and would, therefore, not return to null if not set within the observed window.

The windowed flattening $G(t_i, w)$ is, therefore, not the same graph as would be generated from the Stream $S = \{\langle t_{(i-w)+1}, a_{(i-w)+1)}\rangle, \langle t_{(i-w)+2}, a_{(i-w)+2)}\rangle, \ldots, \langle t_i, a_i\rangle\}$, but is a more pragmatic interpretation of what a person querying the state of the graph may expect. For instance, returning to the stream of updates in Figure 3.2, the windowed flattening $G(t_4, 2)$ would include the two edge additions creating $\langle a, b\rangle$ and $\langle b, a\rangle$. In this instance without keeping vertex a and b in the flattening as they were not directly updated, the graph would be empty, which does not seem the desired outcome. Similarly, for the same flattening, vertex a would lose the username 'Alice', as this was set outside the window, which could be an important characteristic in locating the vertex or differentiating it from its peers.

Whilst the above semantics for a windowed graph flattening are, therefore, taken forward in this work, we acknowledge that there are many different manners in which a window over the temporal graph may be interpreted. For instance, we may hold a much stricter view of the window in which $G(t_i, w)$ does equal the graph generated from the Stream $S = \{\langle t_{(i-w)+1}, a_{(i-w)+1)}\rangle,$

$\langle t_{(i-w)+2}, a_{(i-w)+2)}\rangle, \ldots, \langle t_i, a_i\rangle\}$. Alternatively, within the defined semantics, if an entity is added and then removed within the window period it would not be included within the flattening. It could be argued that if an entity existed at a point within the observation period it should be present, however, this seems to ignore the deletion altogether. Finally, the window could be interpreted as including all entities present within the window period, even if not added within it, more akin to our standard graph flattening. A fuller exploration of the different types of windowing which could be applied over a temporal graph, and how these could be integrated into novel temporal analysis can be seen in Section 7.2.3.

3.5 Challenges of Distribution and Implementation

Whilst these semantics explore how a temporal graph may be conceptually built, updated and interacted on a single machine, to be able to expand past the limits of vertical scaling and handle the large graphs generated by modern data demands, implementation must be done in a distributed fashion. This, however, comes with several challenges which are not addressed above. 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[9]) and whether the content of the data could be taken into account to retain high locality. The temporal nature of the graph expands this question with the additional complexity of managing trade-offs between structural locality (proximity to neighbours) and temporal locality (proximity to the history of an entity) as assessed in ImmortalGraph [12], with the additional possibility of prioritising newer neighbours over older ones. Furthermore, any viable partitioning strategy would have to work for a graph built from a stream of updates, which is clearly difficult to pre-partition as there is no way of knowing what updates are coming next and, even if possible, if not actively managed, data locality will slowly degrade as more entities are added[124]. Finally, the outcome of these decisions would be heavily affected by the type of analysis users were running and what graph flattenings they were interested in. This would, therefore, require constant tweaking and data migration[124] as otherwise the established strategy could actually be detrimental to the query at hand.

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 attempted to solve this problem by blocking update insertion until they can be correctly ordered, e.g. Kineograph's[10] epoch micro-batching, or via centralised ordering, e.g. Weaver's[11] oracle. These are, however, sub-optimal. Micro-batching often ignores the true order of operations within a batch (executing all deletions first, followed by all additions) which can easily generate an incorrect graph, not truly representing the data. Additionally, as the micro-batch epochs execute on update processing time (when they arrive) instead of a set event time (when it really happened), updates may appear in different snapshots, meaning subsequent runs over the same data may differ greatly, causing issues for repeatability and testing. Centralised ordering does fair better than this, providing a consistent ground truth, but bottlenecks the system, restricting its ability to scale[1]. This should, therefore, be solved in a manner which processes updates on event time, creating the same graph for all executions, without relying on a central entity for 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 the readability of an entity's history/neighbour list during analysis[12].

3.6 Distributed Temporal Graph Model

With these challenges in mind, by incorporating the elements of distribution and synchronisation into the temporal graph model and its update semantics, we can better guide the development of real system implementations and preemptively alleviate the discussed issues. For the reasons mentioned in Section 4.4.1, we take the model to have been partitioned in an edge cut fashion, where a distributed temporal graph G_T^D consists of n partitions numbered 1 to n. As vertices are not cut, any vertex is a member of exactly one partition and the set of vertices in each partition do not intersect. Let $R(v_i)$ refer to the partition containing v_i and let V_j^D refer to the set of all vertices which have existed in partition j.

As edges are cut, they may exist in either one or two partitions depending on the location of their source and destination vertex. We define a local edge as any edge where their source v_i and destination v_j are within the same partition i.e. $R(v_i) = R(v_j)$. The set of all local edges which have existed within a partition k is defined as E_k^L. In contrast to local edges, we define split edges as any edge where its source v_i and destination v_j are stored in different partitions i.e. $R(v_i) \neq R(v_j)$. The set of all split edges which have existed within a partition k is then defined as E_k^S and the set of all edges, irrelevant of vertex location, as $E_k^D = E_k^S \cup E_k^L$. Note, for a split edge $\langle v_i, v_j \rangle$, a copy will exist on both partitions containing the source and destination i.e. $\langle v_i, v_j \rangle \in E_1^S$ and $\langle v_i, v_j \rangle \in E_2^S$ where $R(v_i) = 1$ and $R(v_j) = 2$.

3.7 Distributed Update Semantics

Whilst the distribution of the model touches upon the first challenge of partitioning, the semantics of updating are drastically affected by the possible loss of ordering and the need to synchronise edges which have been split across partitions. However, whilst many systems struggle with handling updates in this manner, because of the possible loss of updates, the temporal graph model offers a simple solution in the form of its structural and property histories. As these store all mutations in chronological order, and do not overwrite/delete prior state when updated, all changes to the graph become additive and, therefore, may be inserted in any order. This means delayed or out of order updates may be safely ingested from the stream alongside synchronisation messages between partitions. For this to be possible though, the preconditions and effect of the update types established in Table 3.1 must be modified to fit this distributed environment.

Building from the definition of an unbounded stream within Section 3.3, once distributed, the arrival of an update is no longer received by the temporal graph as a whole, but is instead received by the relevant partition. We define the original source of the updates as partition 0 and the stream of updates from the source to a given partition r as $S(0, r)$. This is a subset of the equivalent undistributed stream $S(0, r) \subseteq S$ such that for all updates in $S(0, r)$, if the action consists of a mutation to a given vertex v, $R(v) = r$. Alternatively, for actions mutating a given edge $\langle v_i, v_j \rangle$, $R(v_i) = r$, i.e. the partition storing the source vertex of an edge receives its updates. If the edge in question is split, a synchronisation update must be sent to the partition storing the second version of the edge e.g. $R(v_j)$. Therefore, we define synchronisation streams between pairs of partitions, whereby $S(1, 2)$ refers to the stream of synchronisation updates flowing from partition 1 to partition 2.

We make the formal requirement that at the update source the prerequisites of each update (as described in Table 3.1) must hold, as well as the requirement that for each new action a_{n+1} at time t_{n+1}, $t_{n+1} > t_n$. However, from the perspective of the receiving partition these are

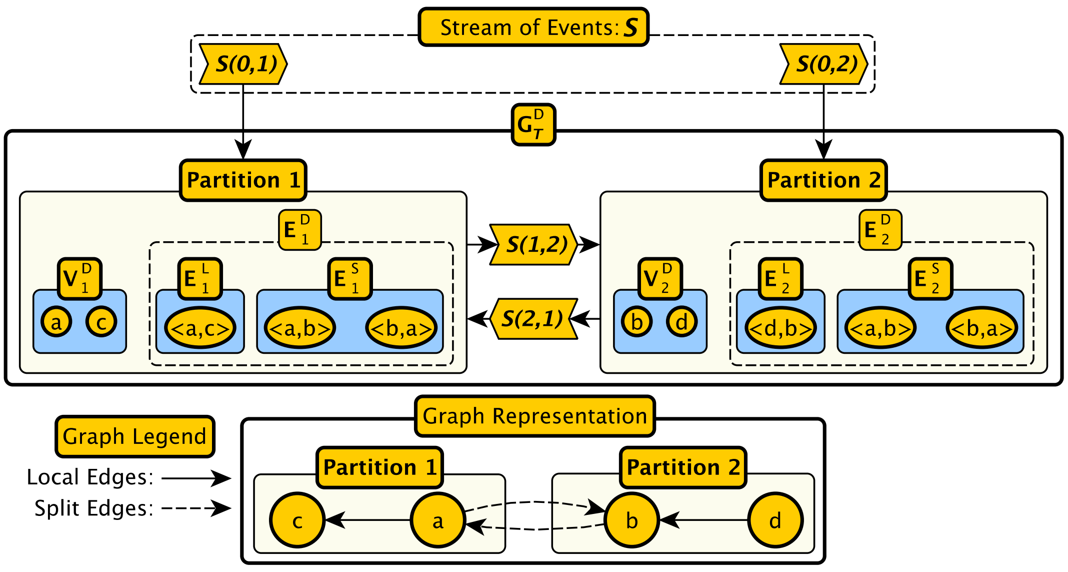


Figure 3.4: Example distributed temporal graph with two partitions, expanding on the graph seen in Figure 3.1

both relaxed. For the prerequisites, modifications are made as described below. For the action time, all timestamps must only be non-repeating discrete values, i.e. for an arriving update $\langle t_m, a_m\rangle, \forall \langle t_i, a_i\rangle \in S(0,r), t_m \neq t_i$ for $m \neq i$. This relaxation allows for the updates to arrive out of order, but still forbids the non-determinism of an insertion and deletion at the same time step. By ordering on event time which is set at the source, delayed commands may be correctly executed when received by appending the new information into the affected entity's history. This even includes the case of deletions which have arrived before the addition of the affected entity, as discussed below in 3.7.3. Finally, for synchronisation messages between partitions, the second constraint is relaxed completely as some updates (such as the edge addition below in 3.7.2) must be converted into multiple graph changes, which are considered to occur at the same time, to ensure the correct state is generated.

3.7.1 Example Distributed Temporal Graph and Update Streams

Before discussing the specifics of each update type in this distributed environment we may first view an example partitioned temporal graph and the streams across it. Figure 3.4 shows an expanded version of the example social network from Figure 3.1. In this instance the graph has been split between two partitions, there are now two new vertices (c and d) and two new edges ($\langle a, c\rangle$ and $\langle d, b\rangle$). Partition 1 contains Vertex a and c which can be seen in V_1^D and Partition 2 contains b and d which are present in V_2^D. As vertices a and b are now in different partitions the original edges ($\langle a, b\rangle$ and $\langle b, a\rangle$) are considered split and stored in both E_1^S and E_2^S. The new edges are, however, local as their source and destinations are in the same partition. These are, therefore, stored in E_1^L and E_2^L respectively. A graph representation of this can be seen on the bottom of the figure. Finally the stream of events S is now split between $S(0,1)$ and $S(0,2)$, feeding the two partitions. The synchronisation streams can then be seen in the middle of the figure ($S(1,2)$ and $S(2,1)$) providing the required bidirectional flow.

3.7.2 Entity Addition

For the addition of a new vertex v at time t_m, within a partition r such that $v \notin V_r^D$, the update is treated in exactly the same manner as in the non-distributed temporal graph. It is inserted into V_r^D and its history is established. If alternatively the vertex is already established such that $v \in V_r^D$, we remove the precondition that its prior state must be *deleted* and insert the *created* state into its history at the given update time $H^v := H^v \cup \langle t_m, \mathit{created}\rangle$. This is because a deletion of the vertex may have been delayed, but upon arrival can be inserted between two creations in the vertices history. In the intermediate view of the vertex (before the missing update arrives) we specify that, for any two consecutive change points within the history of an entity, where the associated states are equal, the most recent update may be ignored, i.e. $H^v = \big\{\langle t_1, \mathit{created}\rangle, \langle t_2, \mathit{created}\rangle\big\} = \big\{\langle t_1, \mathit{created}\rangle\big\}$.

The addition of an edge $\langle v_i, v_j\rangle$ is handled differently depending on whether it is local or split. In the instance of a local edge, rather than checking if both the source and destination vertices exist and are currently present within V_r^D, this precondition is replaced with an expansion of the edge addition to include the addition of its source and destination, i.e. $\big\langle t_m, \mathit{addE}(\langle v_i, v_j\rangle)\big\rangle \implies \big\{\big\langle t_m, \mathit{addE}(\langle v_i, v_j\rangle)\big\rangle, \big\langle t_m, \mathit{addV}(v_i)\big\rangle, \big\langle t_m, \mathit{addV}(v_j)\big\rangle\big\}$. These vertex additions are handled exactly as described above. This does not increase the number of vertices in the graph, as the source and destination vertices are required to exist for the edge to have been generated at the data source. However, the addition for either vertex may have yet to arrive at the partition, leaving the edge hanging. This expansion, therefore, assures the integrity of the graph such that no edges are hanging and once the delayed update does arrive (or if it already has) this secondary creation point within the vertex's history may just be ignored as above. The edge may then be updated in the same manner as the non-distributed temporal graph, either inserting the edge into E_r^L if it has yet to exist or updating its history with the new *created* state. As with the vertex addition, in the case of the latter, the precondition for the edge's prior state to resolve to *deleted* is removed, allowing a possibly delayed edge deletion to be placed between the two *created* change points.

If the edge is split, the initial transformation of the update still occurs, but partition r only deals with the source vertex update and the creation/update of its own edge copy (inserted into E_r^S instead of E_r^L). The destination vertex addition and the edge addition are then forwarded to the partition handling the other copy of the edge i.e $S(r, R(v_j)) := S(r, R(v_j)) \cup \big\{\big\langle t_m, \mathit{addE}(\langle v_i, v_j\rangle)\big\rangle, \big\langle t_m, \mathit{addV}(v_j)\big\rangle\big\}$. The second partition receiving these updates handles them exactly as if they had arrived from the source, with the exception that the edge add is not expanded a second time. This allows partitions to synchronise across their cut edges in parallel with updates from the source, without fear of losing updates or generating an incorrect graph state.

3.7.3 Entity Removal

In the non-distributed graph the removal of an edge has two prerequisites, for the prior state to be *created* and the edge e to be a member of E_T. Once distributed, this first prerequisite can be removed in the same manner as the addition above, such that an addition of the edge may be delayed and upon arrival, the associated *created* change point will need to be inserted between the two *deleted* states. The second prerequisite may also be removed such that if $e \notin E^D_r$ at the time of arrival of the update e is still inserted into E^D_r and its history initialised with the deletion i.e. $E^D_r := E^D_r \cup e$, $H^e{=}\{\langle t_{n+1}, deleted\rangle\}$. Edges with only deletions in their history are not considered to be a member of the graph as they are yet to exist at any point in time, would be filtered out of all flattenings and, therefore, could never be observed. However, such edges become a member as soon as a delayed addition arrives and its change point has been inserted into the history. As stated above, this addition will also cover the creation of the edge's source and destination, ensuring the edge is not hanging now that it considered a member of the graph.

Unlike the addition of an edge, the deletion does not affect its associated source and destination vertices, as edges only including deletions do not break any integrity constraints. However, if the edge is split this still requires synchronisation and, therefore, the edge removal must be sent to the partition handling the destination, i.e. $S(r, R(v_j)) := S(r, R(v_j)) \cup \big\{\big\langle t_m, rmvE(\langle v_i, v_j\rangle)\big\rangle\big\}$. The receiving partition may then handle this in exactly the same manner as the original, minus the synchronisation output.

For the removal of a given vertex v within partition r, previously this had to be a member of V_T and its most recent state had to be *created*. Both these constraints are removed in favour of initialising the vertex via the deletion, in the case of the former, or adding a secondary *deleted* change point into its history in the case of the latter. Vertices initialised as deleted are not considered part of the graph yet, but this will allow the delayed vertex addition to be slotted behind it once it arrives. As with the non-distributed vertex deletion the removal of a vertex also implies the deletion of all associated edges. Whilst for local edges this may be handled in the same way (with the exception that the removal will be placed in their history even if they are currently removed) split edges require synchronisation. This means that for all split edges where v is either the source or destination, the partition storing the second copy of the edge must be informed, i.e. $\forall\langle v_i, v_j\rangle \in E^S_r$ if $v_i = v, S(r, R(v_j)) := S(r, R(v_j)) \cup \big\{\big\langle t_m, rmvE(\langle v_i, v_j\rangle)\big\rangle\big\}$ else if $v_j = v, S(r, R(v_i)) := S(r, R(v_i)) \cup \big\{\big\langle t_m, rmvE(\langle v_i, v_j\rangle)\big\rangle\big\}$. The partition receiving this may then handle it in the same manner as a normal edge deletion synchronisation. Note, as a vertex deletion may arrive before a related edge has been inserted into E^D_r, this *deleted* change point must be inserted into the history of any new edge where v is a source or destination from this point forward, ensuring the edge does not miss this update due to incorrect update ordering. If

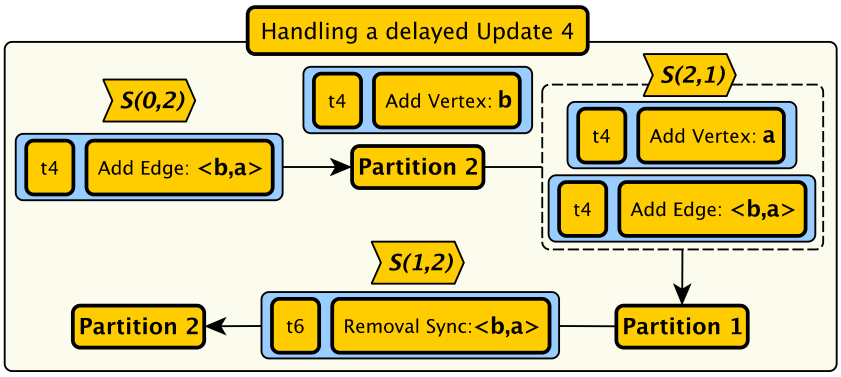


Figure 3.5: An example of how the distributed update semantics would be applied if update 4 (the creation of edge $\langle b, a\rangle$) from the stream in Figure 3.2 was delayed, arriving after the deletion of vertex b (update 6).

the edge in question is split, this will also require synchronisation.

An example of this synchronisation can be seen in Figure 3.5 which is based on the social network and its update stream from Figures 3.1 and 3.2. In this example update 4 (the addition of the edge $\langle b, a\rangle$) has arrived at the very end of the stream, after the removal of its destination vertex a at t_6. This is initially received by Partition 2 through stream $S(0, 2)$ as the source node is b and $R(b) = 2$. This edge addition is then expanded as explained above to include the addition of the source and destination vertex $\big\langle t_4, addE(\langle b, a\rangle)\big\rangle \implies \big\{\big\langle t_4, addE(\langle b, a\rangle)\big\rangle, \big\langle t_4, addV(b)\big\rangle, \big\langle t_4, addV(a)\big\rangle\big\}$. Partition 2 creates its copy of the edge and completes the vertex addition for b but, as the edge is split, forwards the remaining update requirements to Partition 1, the container of the destination node $S(2, 1) := S(2, 1) \cup \big\{\big\langle t_4, addE(\langle b, a\rangle)\big\rangle, \big\langle t_4, addV(a)\big\rangle\big\}$. Partition 1 will handle the two updates, including inserting the deletion of vertex a at t_6 into the history of the edge $H^{\langle b,a\rangle} := H^{\langle b,a\rangle} \cup \langle t_6, deleted\rangle$. Finally, as this is a split edge, Partition 1 must inform Partition 2 that its version of this edge is missing this deletion in its history, forwarding the information for it to be inserted $S(1, 2)) := S(1, 2) \cup \big\{\big\langle t_6, rmvSync(\langle b, a\rangle)\big\rangle\big\}$. Partition 2 inserts this information into the history of its edge copy, completing the update and establishing the same graph state which would have occurred if the updates have arrived in the correct order.

3.7.4 Entity Updates

Finally, updating the properties of an entity previously had the requirements that the entity must exist in Y_T and its prior state must equal *created*. To nullify these, once distributed, instead of defining property updates as a separate update type, they are converted to entity additions, i.e. $\big\langle t_m, updateE(\langle v_i, v_j\rangle)\big\rangle \implies \big\langle t_m, addE(\langle v_i, v_j\rangle)\big\rangle$, $\big\langle t_m, updateV(v)\big\rangle \implies \big\langle t_m, addV(v)\big\rangle$. By doing this, if the update arrives before the real addition it may establish/revive the entity and allow the *new* property values to be stored alongside their given key. The change point established by the real addition may then be inserted into the structural history behind this update, once it arrives. Similarly, if the property update and the delayed entity addition both change the value of the same keys, these *older* values may be inserted behind those of the update upon arrival. This conversion also means that for split edges, where a change in property values must be synchronised, this may be handled in exactly the same manner as the edge addition above.

3.8 Summary

In summary, we have taken inspiration from different graph models seen within Chapter 2 and defined a temporal property graph model which may be mutated via a stream of updates. We defined the semantics for these updates, both in the addition and deletion of entities and the updating of their properties. We explored how when inserted into a temporal graph these updates do not overwrite prior state, but instead append to it, generating both a structural and property history for all entities. We discussed how the temporal graph may be 'flattened' into a static graph, deriving how it would have looked at the flattening time from the structural and property histories instead of having to reingest the updates up until this point. These graph flattenings were then expanded to incorporate the concept of windows, only looking back a certain depth into the history of the graph.

Once the base model had been defined we explored the challenges of implementation and distribution. This required the splitting of the graph into partitions which could be placed on different machines within a deployment. These partitions would then have a shared state (either cut edges or vertices) which would have to be synchronised whenever one of the machines storing it received an update. Both the updates coming from the stream source and the synchronisation messages between partitions then have the possibility of arriving out of order, breaking the constraints established within the original update semantics. To alleviate these issues we recreated the temporal model with distribution in mind, defining what a graph partition consists of, how the now parallel streams of messages from the original source and other partitions may be interpreted and how out of order updates may be handled without losing data or breaking the consistency of the graph.

Chapter 4

Raphtory Ingestion, Modelling and Maintenance

4.1 Introduction

Chapter 3 defines the elements of a temporal graph and how it may be conceptually updated/distributed. This chapter initially introduces the Raphtory system which implements these guiding principles, but focuses mainly on the many challenges which must be solved when it comes to ingesting data into the system, modelling this as graph updates and maintaining the graph across a set of real machines. These challenges are split broadly into distributed implementation issues, as discussed in Section 3.5, and more practical challenges of how to make the system congenial from a user's perspective when ingesting their data and performing analysis.

To recap the issues of distribution, firstly it must be decided what a temporal graph partition consists of and the strategy for splitting the overall graph. Secondly, unlike single machine systems which can maintain global state, distributed systems must continuously synchronise between machines which share state to minimise inconsistencies. This shared state comes in the form of entities which have been cut by the partitioning algorithm and must be present on all machines with related entities. Thirdly, updates coming into the system may arrive out of order, which the distributed temporal graph model above makes allowances for, but there are many corner cases which must be handled when actual objects are interacted with. Finally, the model does not account for the large resource requirements of storing the graph history in-memory within real machines, which must clearly be managed as otherwise the system will crash.

With regards to simplifying the ingestion process for the user, data may be stored in a plethora of different ways, as well as being either of fixed size or continuously increasing over time (bounded or unbounded). Each combination of these may then require its own distinct retrieval method (push, pull, polling, etc.). Once the data is ingested, there must be a clear way of converting this raw information into a stream of graph updates, which must then be correctly forwarded to the right machine, built into graph entities and made available for querying. Finally, for submitted queries, analysis should only be performed on graph flattenings, as defined in Section 3.4, where all of the updates, up to and including that point of time, have fully synchronised, ensuring the correct result. Therefore, a manner of tracking the latest safe update time across the whole graph must be established.

As expanded on below, these problems have been tackled within Raphtory via three main components: the *Spout*, providing an interface for connecting Raphtory to data repositories; the *Graph Router*, which converts raw data into the stream of graph updates from which the graph is constructed; and the *Partition Managers*, which are responsible for converting the stream of graph updates into a temporal graph model, maintaining the data across a set of machines, a visualisation of which can be seen in Figure 4.1. This Chapter discusses all of these components and their interactions in depth, as well as how much involvement a user has to take at each step.

> This chapter contains deep internal details of Raphtory's implementation. For some readers this will be very insightful, for others less so. As such, each section (barring the overview) begins with one of these boxes providing the key takeaways, allowing those less concerned with the specifics to continue reading past without fear of missing something important.

4.1.1 Chapter Roadmap

Section 4.2: Raphtory Overview An initial overview of the Raphtory system, exploring how all of the elements for ingestion and analysis relate, providing the reader context for the sections following.

Section 4.3: Ingesting Data - Raphtory Spout Explanation of the Spout, Raphtory's point of connection to the outside world. Here we discuss how Raphtory ingests from different datasets and how a user may implement their own Spout.

Section 4.4: Graph Modelling and Partitioning - Graph Router Discussion of the Graph Router, the component which converts raw tuples from the user's data source into graph updates. Here we discuss how to model data as a graph, how this is partitioned in Raphtory and how the user may define their own Routers.

Section 4.5: Graph Partition Manager An introduction to the Partition Manager and its subordinates who manage the state of the in-memory graph as well as the execution of analysis.

Section 4.6: Partition Writer A deep dive into the Partition Writer which handles the ingestion of new graph updates from the Routers alongside update synchronisation with its peers across partitions.

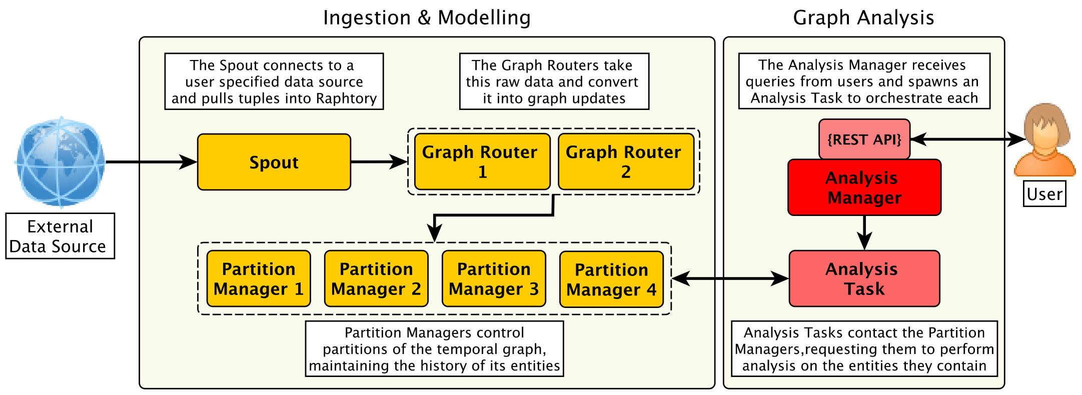

Ingestion & Modelling
Graph Analysis
The Spout connects to a user specified data source and pulls tuples into Raphtory
The Graph Routers take this raw data and convert it into graph updates
The Analysis Manager receives queries from users and spawns an Analysis Task to orchestrate each
External Data Source
Spout
Graph Router 1
Graph Router 2
{REST API}
Analysis Manager
User
Partition Manager 1
Partition Manager 2
Partition Manager 3
Partition Manager 4
Analysis Task
Partition Managers control partitions of the temporal graph, maintaining the history of its entities
Analysis Tasks contact the Partition Managers,requesting them to perform analysis on the entities they contain

Figure 4.1: Raphtory Architecture Overview.

Section 4.7: Watermarking Here we describe how Raphtory tracks the updates each Partition has ingested/synchronised; generating the live graph time and when in the history of the graph is safe to analyse.

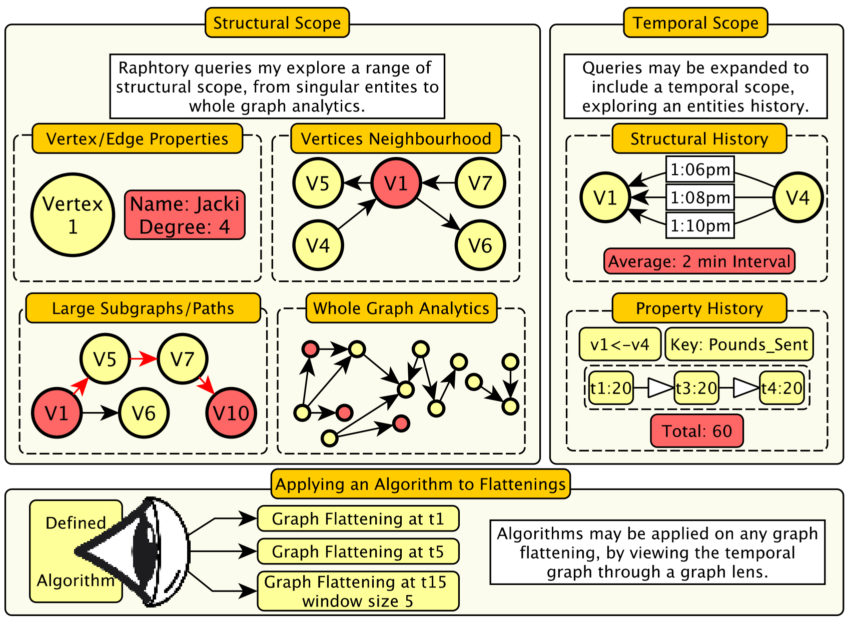
Section 4.8: Partition Archivist A look at the Partition Archivist which is in charge of ensuring each partition does not run out of memory. Here we discuss how the graph state is persisted, the oldest history 'archived' (moved out of memory), and how this is retrieved for queries if once again required.

4.2 Raphtory Overview

Given that we now understand the temporal graph model and its semantics, the next step was to use this as a blueprint to create a system which supports the model, provides online time-aware graph analytics and addresses the challenges of distribution discussed above. For this we have developed *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 fully curated, ensuring all changes are correctly ordered and allowing analysis on both the Live Graph and any point within its history.

4.2.1 Implementing the Distributed Temporal Graph Model

Raphtory's core components for modelling and ingestion consist of *Spouts*, *Graph Routers* and *Graph Partition Managers*. These can be seen on the left of Figure 4.1. *Spouts* are analogous to the stream source discussed in Section 3.7 and attach to a user specified data source external to Raphtory. Tuples are then pulled from this source and pushed into the system. These raw data tuples are received by the *Graph Routers*, which convert each into one or more of the update



Structural Scope
Raphtory queries my explore a range of structural scope, from singular entites to whole graph analytics.
Vertex/Edge Properties
Vertex 1
Name: Jacki
Degree: 4
Vertices Neighbourhood
V5
V1
V7
V4
V6
Large Subgraphs/Paths
V5
V7
V1
V6
V10
Whole Graph Analytics
Temporal Scope
Queries may be expanded to include a temporal scope, exploring an entities history.
Structural History
V1
1:06pm
1:08pm
1:10pm
V4
Average: 2 min Interval
Property History
v1<-v4
Key: Pounds_Sent
t1:20
t3:20
t4:20
Total: 60
Applying an Algorithm to Flattenings
Defined
Algorithm
Graph Flattening at t1
Graph Flattening at t5
Graph Flattening at t15 window size 5
Algorithms may be applied on any graph flattening, by viewing the temporal graph through a graph lens.

Figure 4.2: The structural and temporal scope of algorithms within Raphtory. Once an algorithm is defined it may be run on any flattening throughout the lifetime of the graph, without modification.

types established in Section 3.3 via a user defined parsing function. Updates are forwarded to the *Graph Partition Manager* handling the affected entity. By decoupling these processes the same data may be modelled as many different graphs by connecting the same *Spout* to *Routers* with unique parsing functions or, alternatively, the same *Router* may be connected to various *Spouts* pulling from independent data sources to join them into one graph.

Graph Partition Managers are Raphtory's implementation of the temporal graph partition established in Section 3.6. As their name suggests, these handle all operations of the partition, ingesting graph updates, synchronising with peers and performing analysis. As updates arrive via the pool of *Graph Routers* the manager will perform them as established in Section 3.7, creating entity objects as required and inserting updates into the histories of affected entities at the correct chronological position. Additionally, messages between *Routers* and *Partition Managers* are watermarked to track the most recent update time (the Live Graph) and to know when in the graph's history is synchronised and, therefore, safe to analyse.

4.2.2 Performing Analysis on the Temporal Graph

Once Raphtory is established and ingesting the selected input, analysis of the graph may begin. This is controlled via *Analysis Tasks* which are spawned when a user submits a query via the *Analysis Manager's* REST API; seen on the right of Figure 4.1. *Analysis Tasks* contain a user

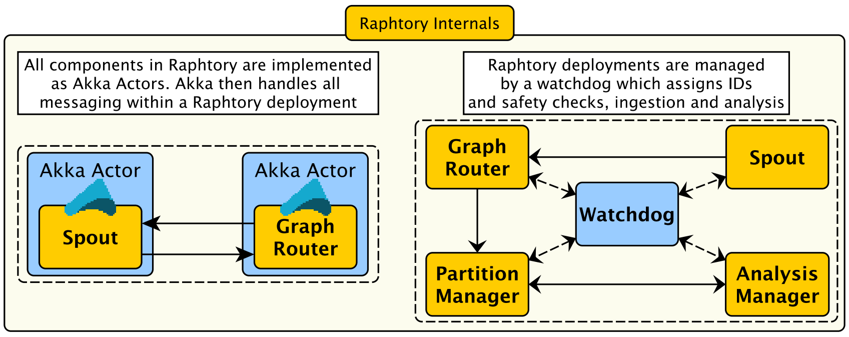

Raphtory Internals
All components in Raphtory are implemented as Akka Actors. Akka then handles all messaging within a Raphtory deployment
Raphtory deployments are managed by a watchdog which assigns IDs and safety checks, ingestion and analysis
Akka Actor
Spout
Akka Actor
Graph Router
Graph Router
Spout
Watchdog
Partition Manager
Analysis Manager

Figure 4.3: The internal management of the components which make up a Raphtory deployment.

defined vertex centric algorithm[7] and coordinate with the *Partition Managers* to execute this in BSP supersteps on the entities they control. These algorithms are implemented via Raphtory's analysis API which gives the user access to the structural and property histories of all entities. Through this they may explore the local neighbourhood of a vertex, paths and subgraphs and perform analytics across the entire graph. This can be seen at the top of Figure 4.2.

All algorithms in Raphtory are executed on graph flattenings as defined in Section 3.4. These can be created for the Live Graph, or for any point back through the graph history. Tasks may be set to run over ranges of the history, creating flattenings at set increments, or may run continuously on the Live Graph, periodically creating flattenings as it updates. In both instances the user may optionally specify a batch of windows which must all be applied at each *flattening_end*; the output from this being a set of windowed flattenings which once analysed will show the differing result of the algorithm when varying temporal depth. To simplify this process for the user, algorithms only require implementing once as they interact with the temporal graph through a *Graph Lens* which only returns the entities present once the window has been applied. This may be seen at the bottom of Figure 4.2.

4.2.3 Underlying Frameworks and Deployment

Raphtory's architecture is based on the actor model[127], a programming paradigm where 'actors' are the primitive unit of computation. Actors have no shared state and communicate via messages which, based on message type, evoke defined control flows known as behaviours. 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 deadlocks and stochastic behaviour [130]. 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 [131]. All components in Figure 4.1 are, therefore, implemented as

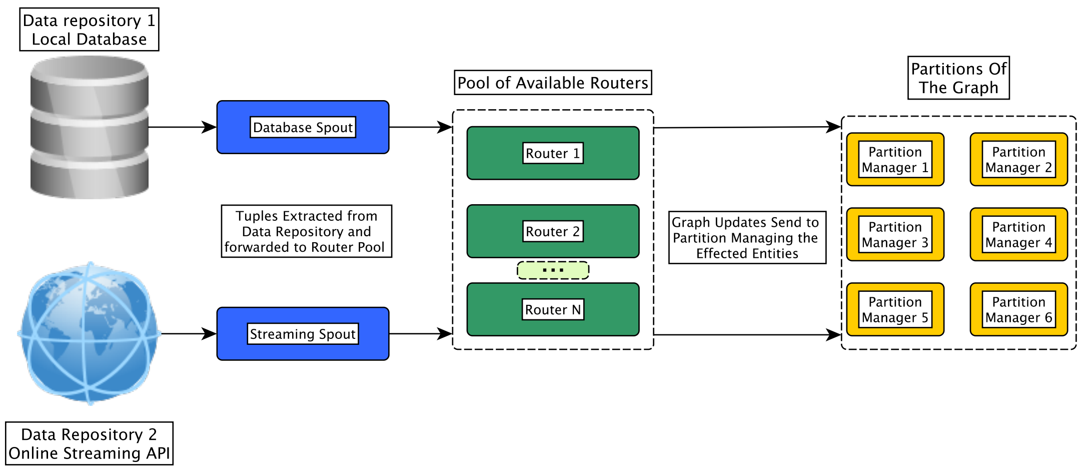


Figure 4.4: The Raphtory pipeline from data source through to Partition Manager.

actors utilising the Akka[132] Framework, as seen within Figure 4.3. Akka provides the foundation for implementing component behaviour and handles all messaging both local and remote. An additional *Watchdog* actor is also present within Raphtory which assigns UUIDs as *Graph Routers* and *Partition Managers* connect, blocking ingestion/analysis until the deployment is fully online.

4.3 Ingesting Data - Raphtory Spout

This section discusses how Raphtory ingests data from the outside world. A chosen dataset may be bounded (non-changing) or un-bounded (continuously increasing in size) and can be ingested from any source location (files, streams, databases, etc.). Ingestion is handled via a component called the Spout which provides a simple user API for connecting to the source, extracting tuples and pushing these into the next component in the Raphtory pipeline - Graph Routers. No processing of the data is done in the Spout, decoupling ingestion and parsing. This allows the same Spout to feed different Graph Routers or many different Spouts to feed into the same graph in parallel, joining their datasets.

Delving into the first component in the ingestion pipeline, the Spout provides an abstract model for all data sources, allowing Raphtory to support any source the user requires. These may range from traditional data stores, such as databases and file repositories, to streaming APIs and message queueing systems, such as blockchain nodes and social media streams. The Spout will perform the initial connection required to access data within one of these sources, consuming tuples/events and pushing these towards the modelling stage of ingestion, i.e. the Graph Routers.

Data ingestion is fully decoupled from all other processes ongoing within a Raphtory deployment, with Spouts permitted to join and leave the cluster at run-time. This is beneficial for a multitude of reasons, firstly because this enables ingestion from multiple heterogeneous sources, either consecutively or in parallel. This can be useful when, for example, historic backups of the data are stored on disk, but up-to-date changes are arriving via a message queueing system such as Kafka[79]. In this instance, one Spout can be initially set up to read and ingest data from old records on disk and, as soon as it is complete, a second Spout can then be initialised to continue ingestion from a Kafka stream, polling the latest information, until explicitly stopped. As an example of this, the Ethereum[133] use case demonstrated in Section 6.4.2 has two main Spouts, which can be seen in Appendix A.1 and A.2. The first connects to a Postgres database which contains the majority of blocks, updated every 24 hours, and the second connects to a synchronised Ethereum node which can then be polled to ingest any new blocks published to the blockchain. This can also be seen in Figure 4.4, where one Spout is connecting to a local database and the other to an online API, but both are pushing into the same pool of Routers. Alternatively, it may be that the data of interest consists of differently formatted files which are to be joined together when building the graph. As an example of this, in the LDBC Social Network Benchmark (SNB)[134] each entity and relationship type is stored as a separate file. These may, therefore, be ingested by a singular Spout aware of all the files, or in parallel by a Spout for each. Similarly the output from these may be parsed by separate Router types (one for each file), or by a singular Router with more complex logic. This second option, however, would require each Spout to tag tuples with the original source file for correct parsing. Finally, whilst these examples are all pull based in nature, where the Spout is polling some external source, the user is free to forward data in a push based manner where the spout waits to receive instead of actively requesting. This removes the need for middleware (such as Kafka above) as the source can send updates directly into Raphtory. Though, as Raphtory is not a database, in practice having a standard interface which can backup the stream and be read by multiple systems alongside Raphtory is preferable.

The second reason decoupling is beneficial, as expanded upon in Section 4.4.3, is that the same datasets may be interpreted in a variety of different graph structures, each modelled via its own Router. By separating the manner in which the data is brought into Raphtory, only one Spout is required and can feed the data into any number of varying Router deployments.

4.3.1 Spout API

As explained in Section 4.2, all components in Raphtory are Akka actors, with Spouts being no exception. However, Raphtory hides much of this away behind its Spout API, which provides a simple set of functions to get a Spout established and send tuples to the Routers. The user is free to break down the connection and ingestion of data into any number of logical sub-tasks which they may schedule via ***AllocateSpoutTask()***. ***AllocateSpoutTask()*** takes a name and a wait duration which represents the delay in seconds before the Spout is set to receive the message. Messages are then handled by the ***ProcessSpoutTasks()*** function, which acts as a switch mechanism redirecting all received messages to user-defined processing functions. This allows the user to be in complete control of both the execution flow and the handling of the data. Within the user-defined functions, users are able to further schedule other tasks and send tuples to the pool of Graph Routers via the ***SendTuple()*** function. ***SendTuple()*** takes any type of data as long as it is serialisable and will automatically load balance across the Routers via round robin.

The different ways in which the Spout API can be utilised can be seen within the Ethereum examples discussed above. Here, the Postgres Spout storing the majority of blocks has three sub-tasks which it performs. The first task conducts the initial connection to the database and then schedules the second task to run after a short pause. This second task submits an SQL query to the database pulling all transactions from the first Ethereum block and forwarding each transaction into the Router pool via ***SendTuple()***. The task then reschedules itself to run immediately, pulling the next block. This continues until the highest stored block is reached (all data in the database has been read), at which point the third task is scheduled, shutting the Spout down as it has completed its ingestion. In comparison, the Ethereum node Spout only has one task, which contacts a REST API sequentially pulling blocks starting from where the first Spout left off. If it successfully retrieves a block (set of transactions) it will send these to the Routers and immediately schedule an attempt at the next block. If the requested block is not yet available (i.e. it has fully ingested all published data) it will schedule a retry after a one second pause. This will continue until the user decides to shut it down.

4.4 Graph Modelling and Partitioning - Graph Router

This section discusses how Raphtory converts raw data into a graph. This is handled by components called Graph Routers. These receive the tuples output by Spouts, parsing each piece of information via a user defined function. Within this function the user must decide what the information means, what graph updates to extract and the entities affected by these. Choosing the appropriate graph model here is an important decision which can drastically affect the output of executed algorithms later on. Once extracted the finalised updates are then handed back to the Router which forwards them to the Partitions of the graph in control of affected entities. The graph is partitioned via a global hashing algorithm, allowing components to calculate the location of all graph entities. Raphtory recommends that datasets have a time component included, to correctly order them in the temporal model. It can, however, make do with orderable substitutes if unavailable.

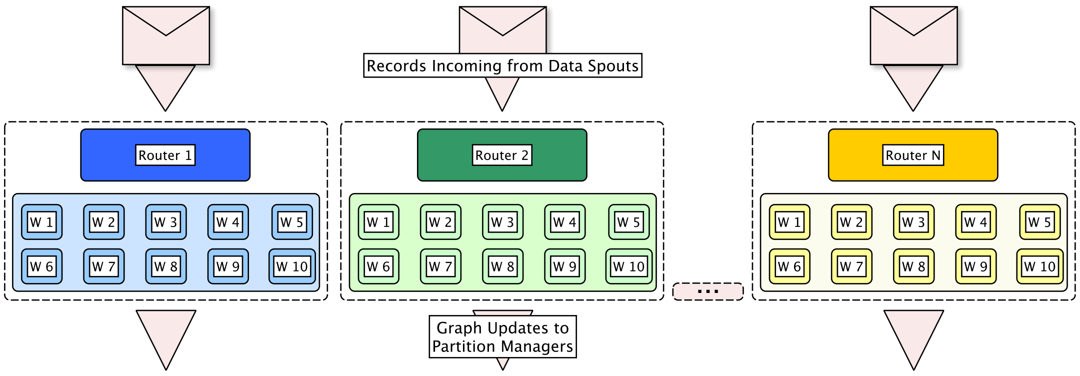

Records Incoming from Data Spouts
Router 1
Router 2
Router N
W 1
W 2
W 3
W 4
W 5
W 6
W 7
W 8
W 9
W 10
Graph Updates to Partition Managers

Figure 4.5: The internal structure of a pool of Graph Routers within an example Raphtory deployment.

Following on from the Spout, the next stage in the pipeline is Raphtory's Graph Routers. These take the raw events/tuples which have been extracted via a Spout and convert them into one or more graph update operations, as defined in Section 3.3. Graph Routers operate on a tuple by tuple basis allocating the actual data parsing to a pool of workers. These workers operate completely independently and will forward any graph updates to the Partition Manager(s) storing the affected entities as soon as they have been distilled from the data. This model allows the resources allocated for data parsing to be freely scaled to match the magnitude of data throughput by adding or removing Routers from the Raphtory cluster as required. An overview of a set of Graph Routers can be seen in Figure 4.5.

To allow for the Graph Routers to operate without synchronisation, all generated graph updates contain an assigned timestamp. This timestamp is relied upon by the Partition Managers to place the updates in the correct historic order for all entities affected by the update. 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 or, in the worst case, this at least provides an inherent and deterministic order within the data, as defined in the distributed stream semantics within Section 3.7. If such a field was not present within the data, it could be considered to utilise the internal clock of the Graph Routers, synchronised via the Precision Time Protocol (PTP) [135]. However, by allocating timestamps around processing time (similar to Kineograph [10]) rather than event time, the ingestion paradigm is shifted and all events would need to be processed in the same order to generate the same temporal graph. Alternatively, the data may be viewed as a set of updates which occur at the same time (i.e. a snapshot), as there can be no deletions or multiple property values in this instance. Whilst technically against the underlying model, the user would be free to ingest the data in such a manner and the same graph would be generated each time. It would, however, be difficult to track if all the data has been fully ingested (as discussed in Section 4.7) and drastically reduce the temporal analysis possible on the data (as discussed in Chapter 5). Timestamps within the data are, therefore, strongly encouraged as they allow events to be read in across a range of time periods in parallel, updates to be processed late/out of order or the stream completely inverted and the temporal graph can still order them correctly.

That being said, whilst it is possible to ingest data in any order, in practice it makes safe analysis difficult as it is impossible to know if all updates prior to a selected flattening have been ingested. This is because the system has no view of the outside world and can only estimate a safe time based on the timestamps it has seen. For a bounded dataset, this is not an issue as the unordered data can be bulk ingested and analysis can begin once all updates have finished and synchronised. However, for an unbounded dataset/stream this would not be the case as the data could never be considered fully ingested and updates for any point in time could still arrive. As such we make the requirement that updates coming out of each Router Worker must be chronologically ordered, i.e. the time associated with update N is equal to or less than N+1. This allows the Partition Managers to establish a global safe time for analysis via watermarking, as discussed further in Section 4.7. If this cannot be guaranteed, as the chosen stream suffers from delayed events, the times may still be relied on, but the arrival of an out of order update will raise an alert if it should have been included in a materialised flattening. The user may then decide if they wish to re-execute any affected flattenings or if this can be ignored. The process of re-executing window panes over streaming systems is common practice, as discussed in Section 2.4.2.

4.4.1 Partitioning Strategy and Routing

Graph Partitioning is an NP problem[136], even when the data is static and the temporal dimension is not included. There was, therefore, many factors to weigh up when deciding how the graph would be partitioned in Raphtory, with this additionally having a major impact on the way Routers and Partition Managers communicate. As discussed in Section 3.5, the graph may be edge-cut or vertex-cut; may involve some element of temporal locality (should vertices be located near old or new neighbours); may include domain specific knowledge to better group entities; and can be periodically adapted to maintain high data locality as new updates arrive. Unfortunately, across these options, any non-trivial partitioning algorithm will require some form of shared state (either local or global), vertex reference tables/update redirecting (such as in [124]) or a centralised decision maker (e.g. Weavers[11] oracle), which has been shown to scale poorly[1].

With this in mind, it was decided that the best initial method would be a global hashing algorithm which decides the Partition Manager responsible for each vertex. This is an edge-cut partitioning strategy where edges with the source and destination on different machines are stored in both partitions and the Partition Manager controlling the source vertex maintains the 'master' copy. Whilst this seems quite basic it actually has many benefits. The primary one being this requires no state and any Router or Partition Manager may instantly calculate the location of required vertices for initial update propagation or edge synchronisation. This allows the number of Routers and Partition Managers to scale freely, only dampened by the increased

Table 4.1: Update supported by the Graph Router, based on those defined in Table 3.1.

Update Type	**Required Parameters**
Vertex Addition	**Update Time**: The time at which the vertex add occurred, **Vertex ID**: The ID of the vertex being added
Vertex Deletion	**Update Time**: The time at which the vertex deletion occurred, **Vertex ID**: The ID of the vertex being deleted
Edge Addition	**Update Time**: The time at which the edge add occurred, **Source ID**: The ID of source vertex, **Destination ID**: The ID of destination vertex
Edge Deletion	**Update Time**: The time at which the edge deletion occurred, **Source ID**: The ID of source vertex, **Destination ID**: The ID of destination vertex

Optional Parameters for Entity Addition
Entity Type: A label denoting the type of vertex or edge being added
Property Set: One or more properties associated with the entity consisting of a property key (name) and value. This value may be a String (mutable or immutable), Long, Double or Boolean

number of split edges (which would exist no matter the algorithm). The second benefit, as with most hashing algorithms, is that vertices are evenly distributed across the partitions, meaning no machine has an inherently larger vertex centric workload. In this instance, for varied queries over differing points in the history of the graph, processing time may fare better on average in comparison to something more specialised for a set point in time.

The development of temporal aware dynamic partitioning algorithms is clearly an interesting area of research which has very little coverage in the current literature. Whilst it is not implemented here it is planned to be included in future versions of Raphtory once we have a better idea of the workloads users deploy on the platform and a thorough investigation into the topic has been completed; as discussed further in Section 7.2.

4.4.2 Graph Router API

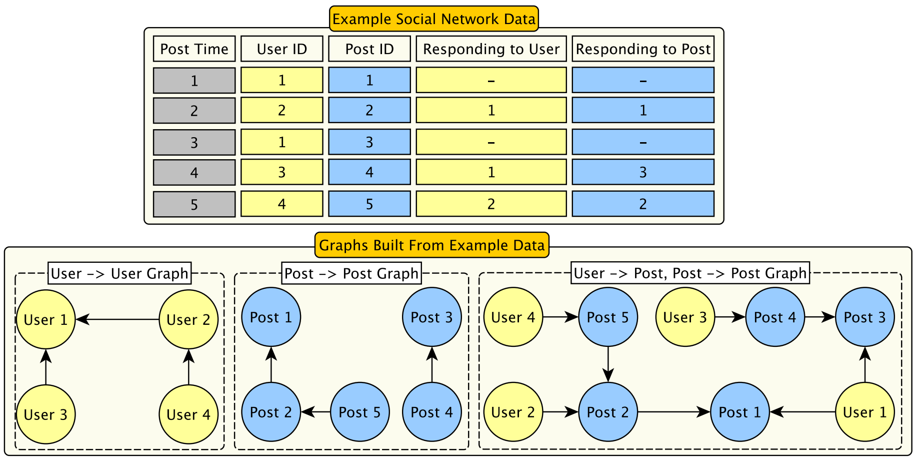
As with the Spout, Raphtory provides an API for the Graph Router which sits atop the underlying Akka actors. This API allows users to write a ***ParseTuple()*** function which is distributed to all Router Workers and is called for each individual message sent from the Spout. This method allows users to freely interpret the data in any way and output zero or more graph updates via ***SendGraphUpdate()***. The available graph updates, as seen in Table 4.1, consist of vertex addition/deletion and edge addition/deletion. Property updates do not appear in this list as they are modelled as entity additions. This allows them to execute before the original addition if required as discussed in Section 3.7 and implemented below in Section 4.6.2. For all update types, a timestamp is required specifying when the change occurred in the history of the graph. For vertex updates a unique ID (Long) must be specified to identify the entity within the graph. If such an identifier exists for the entity within the data, such as the 'person ID' within a generated LDBC SNB graph, this may be used. Alternatively, Raphtory provides the ***AssignID()*** function which takes any non-numerical field within the data (or a combination of fields) and returns a unique ID from it using a MurmurHash3[137]. MurmurHash was chosen as it is known to be very quick, whilst minimising the number of collisions, and is frequently used in large projects, notably nginx and Hadoop[138]. As an example of this, within the Ethereum Router (see Appendix B.1), the hashes representing the wallets involved in each transaction are passed to ***AssignID()*** which returns a unique long for each wallet. Edge updates similarly require the identifier for the source and destination nodes associated with the edge, which are combined to make the unique ID for the edge. These again may come directly from the data or by passing the same fields to **AssignID()**.

Following these mandatory components, there are also two optional parameters for all additions into the graph, allowing the user to specify additional information about the entity. The first of these is the entity type, which is a label associated with the vertex or edge, specifying its category/grouping. This can be used to model graphs with multiple types (as seen below in 4.4.3) which may then be utilised to enrich queries during analysis, i.e. only propagating a value along certain edge types. The second is the property set consisting of one or more property names and its associated value at the time of the addition/update. The key for a property is always a string, but the value may consist of a string, long, double or boolean depending on the type of information it stores. These may be set to mutable or immutable allowing the user to note if the property requires its own history or if a single value can be stored to save memory.

Continuing the pipeline from the Spouts established in Section 4.3.1 above, the Ethereum Router (seen in Appendix B.1) can receive tuples from either the Postgres Spout or Ethereum node Spout. Each tuple sent from these Spouts consists of an individual transaction between two wallets within the Ethereum network which is interpreted via the **ParseTuple()** function. In each instance the function extracts the time of the transaction, wallet IDs and the amount of ether transferred, converting them into three graph updates, two vertex additions, one for each wallet, and an edge addition for the transaction between them. Both vertex additions contain their wallet hash as an immutable property as this value does not change. In contrast, the transaction value is allocated on the edge as a mutable long, as many transactions may occur between these two wallets and this value may be different every time. As there is only one vertex and edge type in this network no explicit types are added.

4.4.3 Modelling Data as a Graph

When initially looking at a given dataset, there may be a clear manner in which individual tuples are converted into updates and, therefore, how the overall dataset may be modelled as a graph. However, there are often many ways to interpret data, with different vertex types which may be extracted and numerous edge types which can be drawn between them. This provides a clear motive as to why the Spout and Router are decoupled, allowing the user to view the same data in a multitude of ways. In addition to this, dependent on the graph entities and structure established by the Router, the output from a given algorithm will be completely different. This means that the process for analysis begins at the Router and often requires the user to think hard about what an algorithm is going to return, how this should be interpreted and what modifications may need to be made, depending on the chosen model of the data.



Post Time	User ID	Post ID	Responding to User	Responding to Post
1	1	1	-	-
2	2	2	1	1
3	1	3	-	-
4	3	4	1	3
5	4	5	2	2

Figure 4.6: Different graph models extracted from the Gab.ai social network dataset explored in Section 6.4.1. Columns and vertices are colour coded so they may be matched easier.

As an example of this, the data explored in Section 6.4.1, extracted from the social network Gab.ai, consists of posts and responses made by users of the network. Figure 4.6 demonstrates various ways an example set of posts and comments may be interpreted, the simplest of these being a *User*→*User* graph where edges are drawn between users when they interact with each other. The results of a connected components algorithm run on this may be interpreted as the different communities of the network and used to explore if these are isolated or largely interconnected. Alternatively, if the data was formatted as a *Post*→*Post* graph, where responses create edges between posts instead of users, the same algorithm would return the different communication threads within the network. Finally, in a more complex model, mixing both posts and users, a simple connected components algorithm may begin to make less sense and, therefore, need to be expanded to propagate based on edge/vertex type or be replaced with a more appropriate function.

4.5 Graph Partition Manager

This section gives an overview of how Raphtory manages the state of the temporal graph in a distributed environment. This is the job of the Partition Manager and its three subordinates the Writer, Reader and Archivist. Writers handle graph updates from the Routers and synchronise when needed with other partitions. Readers perform analysis on the entities within the partition when a query is submitted. Archivists work in the background to manage the in-memory history, backing up older state to be removed if the history becomes too large for memory.

Vertices and Edges are represented as objects which have their history stored in an ordered triemap for quick insertion and searching. Properties are similar, but may be defined as immutable by the user if their value will never change to save space. Vertices and Edges contain a map of their properties; vertices contain their edges in incoming and outgoing maps; vertices are tracked in an Entity Storage. A Partition Manager will control multiple Entity Storages, viewed as virtual partitions allowing controlled multi-threading inside the partition.

Graph Partition Managers are Raphtory's primary component, each storing a partition of the overall in-memory graph. These partitions are an implementation of those defined in Section 3.6 and contain a unique set of vertices and their incoming/outgoing edges. Each Partition Manager is responsible for maintaining up-to-date histories, completing analysis requests and performing incremental backups for these entities. These tasks are delegated 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 entities affected. Readers handle requests for analysis, executing the provided algorithm and returning the results. Archivists work in the background, persisting new data to permanent storage and archiving the older history to alleviate memory constraints.

4.5.1 Entity Modelling and Partition Storage

Based upon the model introduced in Section 3, the graph representation in Raphtory is split between *Entities* and *Properties*. Entities are the supertype of *Vertices* and *Edges*, containing all common attributes. The most important of these attributes is their previous state, associated properties and entity type, which together constitute the structural and property history of the entity. *Structural history* for an entity is stored in the form of a triemap where the keys are event timestamps and the values are booleans; true for a creation/update, false for a deletion/removal. This is then ordered from the newest to oldest timestamp i.e. the latest state of an entity is at the head. A triemap was chosen principally because it provides fast insertions of new updates at the head, as well as delayed or out of order updates further into the tree. However, in addition to this, as two entries within a triemap cannot have the same key (time), updates to an entity history are idempotent, i.e. if you run the same update twice it will not affect the end result. Therefore, message delivery semantics can be relaxed from exactly-once to at-least once. This fits with the update semantics defined in Section 3.7, whereby updates must occur at unique timestamps to remove the order ambiguity they would otherwise generate. Properties associated with an entity come in two varieties, mutable and immutable. Mutable properties store their history in a similar triemap structure. However, whilst the key is still a timestamp, the value is now an *'Any'* (a Scala concept denoting any object type), allowing the property value to be a string, number or boolean depending on the needs of the user. These are used for the majority of properties which change through time, such as a person's bank balance, which alters as they

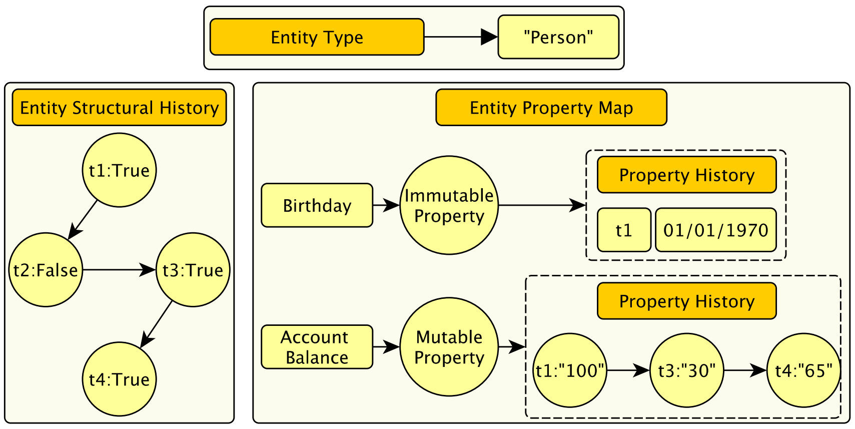


Figure 4.7: The internal structure of an entity within Raphtory.

make and receive payments; as can be seen in Figure 4.7. Immutable properties on the other-hand only store a singular value (which cannot change), together with the time this value was first seen. This reduces memory overhead as duplicate values are not required in the history for delayed updates to be slotted in-between. Immutable properties also require less attention from the Archivist and reduce lookup times during analysis. This does, however, come with the obvious caveat that it can only be chosen when the user is confident the value of the given property will be static for the lifetime of the graph. There are many instances where this makes sense, however, such as the person's birthday in Figure 4.7, as people cannot change the day they were born. Irrelevant of mutability, properties are additionally given a name allowing them to be differentiated within each entity and read/updated at a later time.

The third of these components, the entity type, is handled in a similar manner to an immutable property whereby the value cannot change once allocated. However, as these have a known name and are treated as always existing in an entity (i.e. have no associated 'first seen' time) they are simply stored as a string. A visualisation of the type and structural/property histories of an entity can be seen within Figure 4.7.

Based on the model proposed in Section 3.6, 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. Edges are a subtype of entity and, in addition to their properties and history, contain the IDs of their source and destination vertices, which in turn becomes its unique identifier. 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 copy of the edge is maintained by the Partition Manager storing the destination vertex, allowing both partitions to access its state. Split edges are represented as a further sub-class, allowing the Partition Manager to distinguish them. Split edges are stored in the same fashion as a local edge, but additionally record the location of their master/ghost

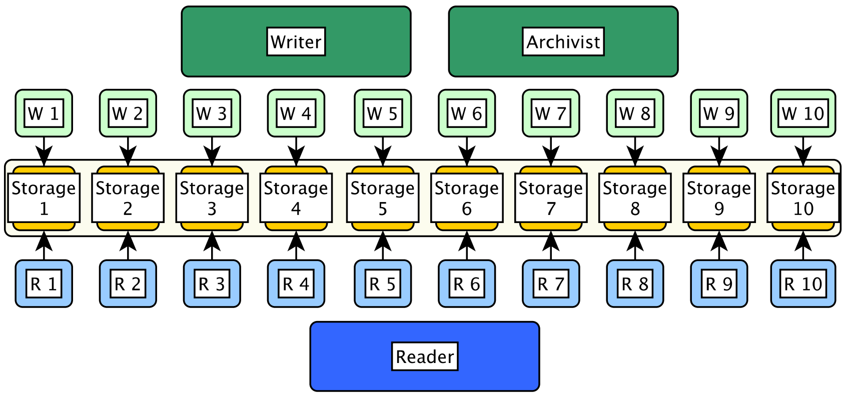


Figure 4.8: The internal storage of a Partition Manager with its data split between ten Entity Storages. Each storage is accessible by one Writer Worker (managed by the Writer and Archivist) and one Reader Worker (managed by the Reader).

copy. If a change occurs to a split edge, this information can be utilised to forward the update, synchronising the copy.

Vertices contain a unique ID (Long) which differentiates them from all other entities within the graph. In addition to information about the node itself, each vertex contains two maps storing its incoming and outgoing edges. The key for these maps is the ID of the other vertex sharing the edge, i.e. for vertex v_1, the key '2' would retrieve the edge $\langle v_1, v_2 \rangle$ from the outgoing map and $\langle v_2, v_1 \rangle$ from the incoming. Storing the edges inside of their respective vertices in this manner was chosen over one monolithic edge map to simplify the locking and maintenance of vertices, as well as reducing lookup and insertion times. This also benefits the vertex from an analysis perspective as it requires no external lookups and can easily iterate through its neighbours for messaging or aggregation.

To ensure vertices are maintained and updated correctly they are kept within an 'Entity Storage'. This class stores vertex objects and provides an API for Partition Writers to modify the graph state based on the update semantics defined in Section 3.7. It also provides an API to allow the Partition Reader to safely query the graph in parallel with updates. Partition Managers have a set of Entity Storages which each contain a portion of assigned vertices and can, therefore, be viewed as sub-partitions. Each Entity Storage has an allocated Writer Worker and Reader Worker which are the only actors allowed to read/edit its state. This allows the Partition Manager to obtain a speed-up by multi-threading, whilst avoiding interleavings and deadlocks. A visualisation of the Entity Storages within a partition and the workers which have access to each can be seen in Figure 4.8. A view of the entity hierarchy within two local storages and one remote storage can be seen in Figure 4.9.

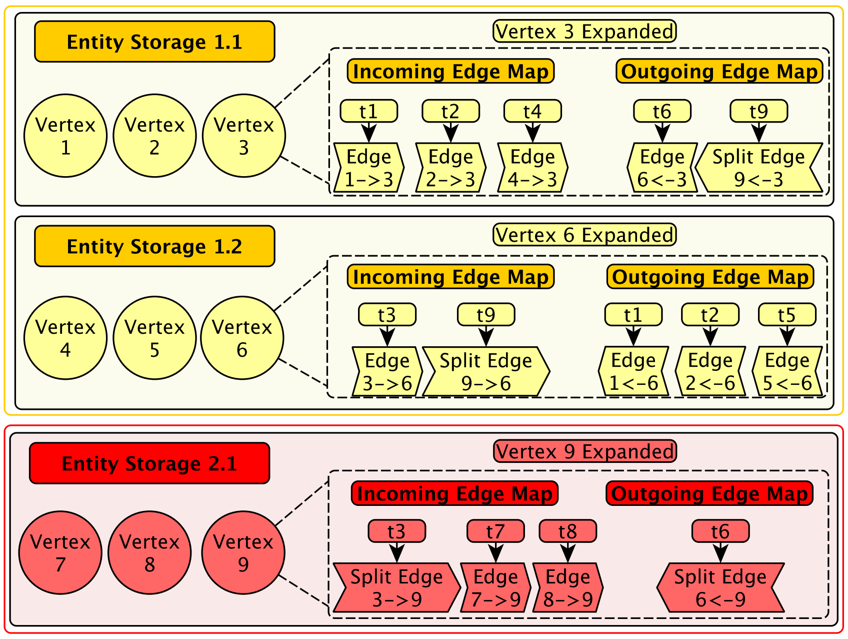


Figure 4.9: A subset of the Entity Storages present in a Raphtory cluster. Storage 1.1 and 1.2 (in yellow) are within the same Partition Manager whereas storage 2.1 (in red) is remote. Within each of these you can see the entity hierarchy, with the storage containing the set of vertex objects and each vertex containing its incoming and outgoing edges.

4.6 Partition Writer

This section provides a deep dive on how the Partition Writer implements the distributed update stream described in Section 3.7. The writer itself has its own workers, one for each Entity Storage. These writer workers receive updates directly from the Routers, which can be imagined as the stream source. Each update type must be handled in a specific manner. These all have their own subsections and summaries below.

The Partition Writer is responsible for all updates to the state of the in-memory graph. However, as touched upon in 4.5.1, the Writer does not perform the operations itself, instead delegating to a set of Writer Workers which it oversees. This permits a controlled multi-threading of updates (as Akka actors are inherently single threaded) and frees up the main actor to track the overall state of the partition and recover/restart workers if any errors occur. Writer Workers receive updates directly from the Graph Routers in parallel with synchronisation messages from their peers. The semantics for correct update management and synchronisation across a partitioned temporal graph are conceptualised and defined in Section 3.7. From the perspective of

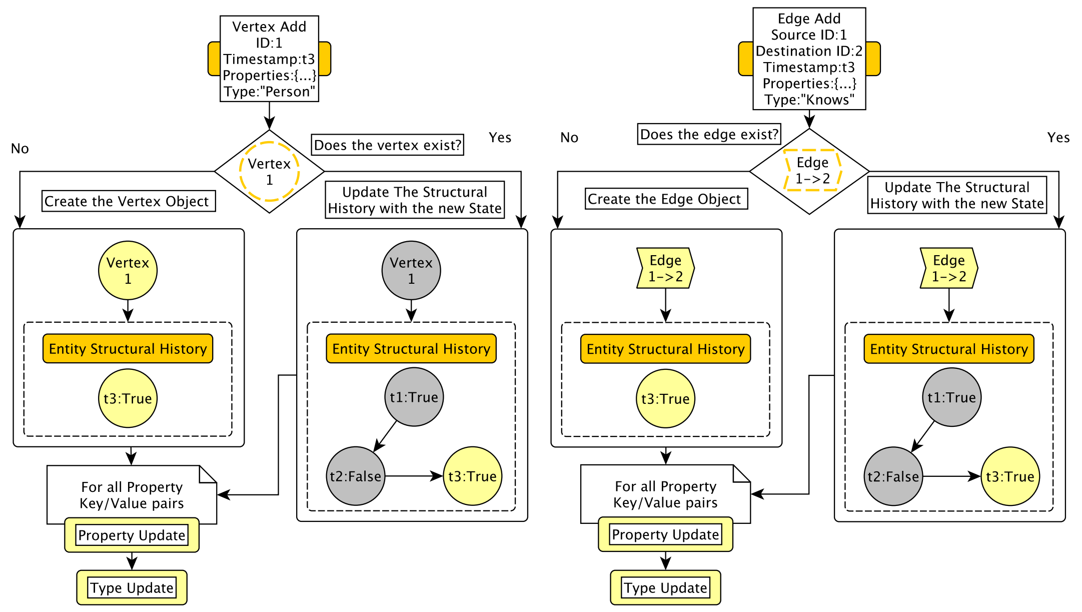


Figure 4.10: Flowcharts demonstrating the process for creating/updating a vertex or edge object within Raphtory. The sub-processes for Property/Type updates can be seen in Figure 4.11. Note: objects in grey existed prior to the update.

these semantics, the stream source is represented by the pool of Graph Routers and each Writer Worker is considered a partition in its own right. This section follows the same layout as 3.7 and explains in detail how each of the defined update types are implemented within the Writer Workers. The only exception to this is explicit property updates which are already represented as entity additions, as discussed in Section 4.4.2.

4.6.1 Adding and Updating Vertices

Vertex additions/updates are the simplest operation as they affect no other entity and involve no synchronisation. They require a vertex object to be present in the entity storage for the given ID. Once present the time of the update can be inserted into the object's history, specifying it was 'alive' at this point. Any included properties are then created/updated in a similar fashion.

This subsection is summarised within the flowchart for vertex addition on the left of Figure 4.10 and the flowcharts for property updates in Figure 4.11.

Upon receiving a vertex addition request the Writer Worker will first check within its asso-

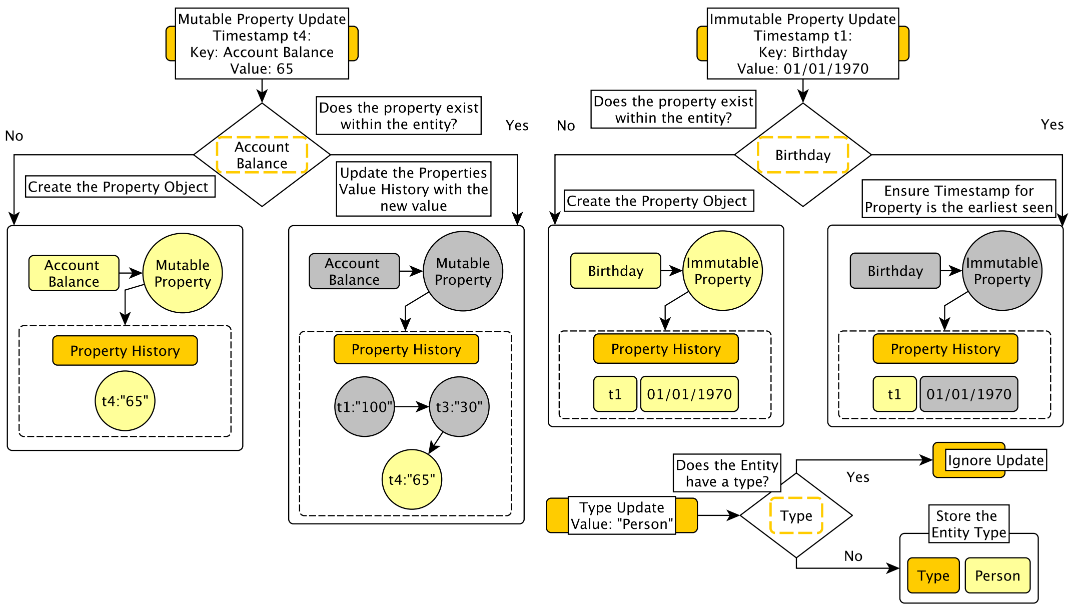


Figure 4.11: Flowcharts demonstrating the process for creating or updating the properties and type associated with an entity.

ciated Entity Storage to see if an object for the given vertex ID exists. If nothing is present one is instantiated, beginning its history at the timestamp within the update message. Property objects are then created for all given keys within the property set of an update, establishing the history of the property with the associated value and update timestamp. These are inserted into the property map of the vertex, referenced by their key, and the vertex is placed into the Entity Storage. Finally, if the update came with a vertex type, this will be stored within the object.

If a vertex object was already present, a new *True* state is inserted into its history alongside the timestamp, even if the latest state also denotes *True*. This is done for two reasons: firstly a remove command may have been delayed, which may then be slotted in-between these upon arrival, establishing the correct history; secondly, without tracking every time the vertex is touched we would not be able to accurately perform temporal windowing (as discussed in Section 3.4) as the vertex may be filtered out when the flattening is created (see Section 5.3.3). For all properties included in the update, a similar check will then be performed to see if an object already exists i.e. if the vertex already has the property. In the case of a mutable property, if an object does exist, the new value and timestamp will be inserted into its history. For immutable properties, if the new timestamp is earlier than the one already stored in the object, this timestamp will be updated (the value is assumed to be the same). This is to remove the possibility of any non-determinism where a later update arrives first and sets an incorrect earliest appearance of the property. Immutable properties will otherwise ignore all updates. If no object exists yet, one will be built as before and inserted into the vertices property map, again referenced by its key. Finally, if the update contains a vertex type and one is yet to be allocated for the entity, it will be stored; otherwise it will be ignored.

4.6.2 Adding and Updating Edges

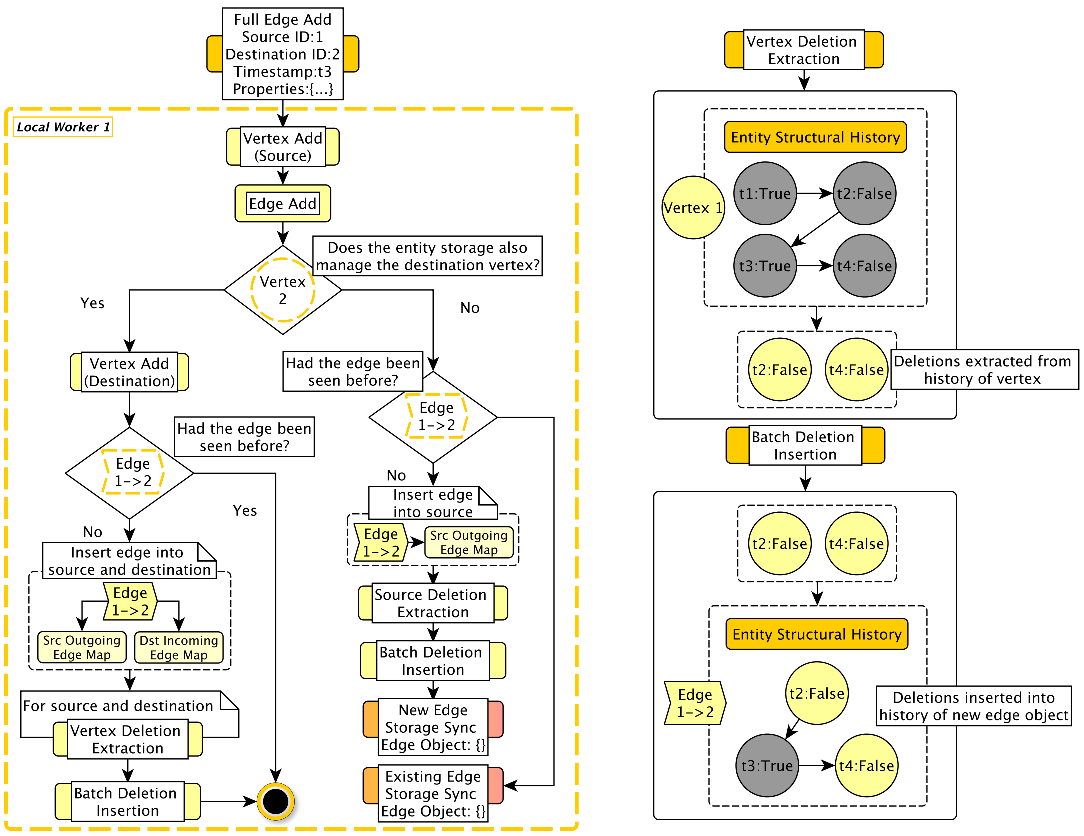
Edge additions/updates are more complex as they require the source and destination vertices to be present and must synchronise if the edge is split between workers. If the edge is 'local', requiring no synchronisation, the worker updates both the source and destination vertices as above to ensure they are present. It then creates/updates an edge object in the same fashion, ensuring that the source vertex has reference to this as an outgoing edge and the destination vertex has reference to it as an incoming edge. If the edge is 'split', requiring synchronisation, the worker only updates the source vertex and its copy of the edge. It then forwards the update to the worker in control of the destination vertex, who updates this and its own copy of the edge.

This subsection is summarised within the flowchart for full edge addition on the left of Figure 4.12. This is supported by the flowchart for edge object creation on the right of Figure 4.10 and the flowcharts for edge add synchronisation in Figure 4.13.

Case of local edge managed by one worker

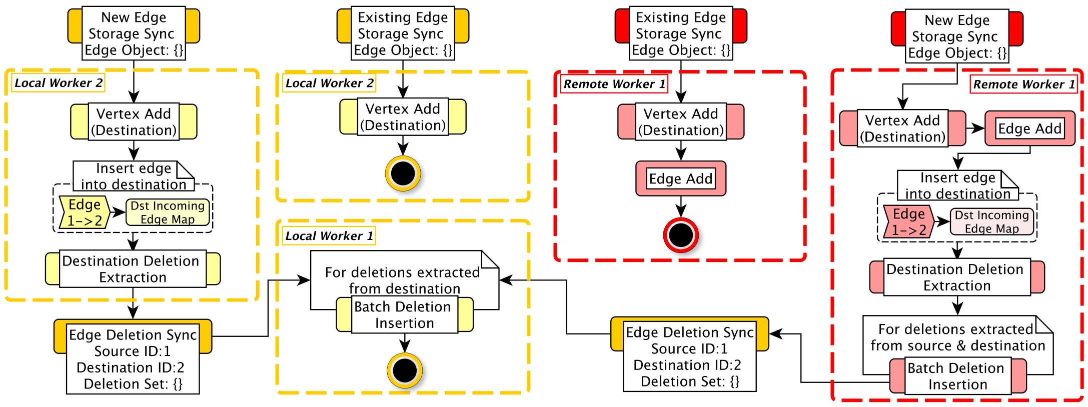
The addition of an edge is more complex than that of a single vertex due to the different levels of synchronisation required by edges, depending on the location of their destination vertex, for which there are three possibilities. The simplest of these is that the Entity Storage associated with the worker stores both the source and destination nodes, meaning the worker can fully complete the update without contacting any other actors. In this instance the worker will first check if vertex objects exist for both the source and destination, as conceptually the graph model does not allow hanging edges, but also the edge object must be stored inside of the vertex objects so they may be aware of the edge during any analysis. If either vertex does not exist they will be instantiated with *True* inserted into the history at the timestamp within the edge update. Any properties or types the vertices should have will then be created once the actual vertex addition update arrives, managed in the same manner as described above. Alternatively, if the vertices do already exist, this *True* state will still be inserted into the history to denote vertex activity at the time of update; again important for windowing based analysis.

Once the worker has checked the presence of both the source and destination, it then checks if these already contain an edge object for this ID i.e. has the edge been seen before. If it has



Full Edge Add
Source ID:1
Destination ID:2
Timestamp:t3
Properties:{...}
Local Worker 1
Vertex Add (Source)
Edge Add
Does the entity storage also manage the destination vertex?
Vertex 2
Yes
No
Vertex Add (Destination)
Had the edge been seen before?
Edge 1->2
Yes
No
Insert edge into source and destination
Src Outgoing Edge Map
Dst Incoming Edge Map
For source and destination
Vertex Deletion Extraction
Batch Deletion Insertion
Insert edge into source
Source Deletion Extraction
New Edge Storage Sync Edge Object: {}
Existing Edge Storage Sync Edge Object: {}
Vertex Deletion Extraction
Entity Structural History
Vertex 1
t1:True
t2:False
t3:True
t4:False
Deletions extracted from history of vertex
Batch Deletion Insertion
Deletions inserted into history of new edge object

Figure 4.12: Left - The full process of handling an edge add update from the perspective of the worker who first received it. The sub-processes for vertex/edge addition can be found in Figure 4.10. Right - The processes for extracting the deletions from a vertex and inserting them into a new edge.



New Edge Storage Sync Edge Object: {}
Local Worker 2
Vertex Add (Destination)
Insert edge into destination
Edge 1->2
Dst Incoming Edge Map
Destination Deletion Extraction
Edge Deletion Sync Source ID:1 Destination ID:2 Deletion Set: {}
Existing Edge Storage Sync Edge Object: {}
Local Worker 1
For deletions extracted from destination
Batch Deletion Insertion
Remote Worker 1
Edge Add
For deletions extracted from source & destination

Figure 4.13: Different processes of synchronising an edge add depending on if it is the first time the edge has been seen and if the second worker is local or remote.

not previously been seen, an edge object will be created in much the same way as a vertex object (as seen on the right in Figure 4.10). This edge object is then added into the outgoing edge map of the source vertex, referenced by the ID of the destination node, and the incoming edge map of the destination, referenced by the ID of the source. Finally, any *False* states (deletions) present within the history of the source and destination are inserted into the edge object. The process for deletion extraction from a vertex and insertion into an edge can be seen on the right hand side of Figure 4.12. This is done to counter the possible non-determinism caused when the edge object creation is delayed or received after a vertex removal and, therefore, will not contain this information within its history, as explained below in Section 4.6.4. This placement into the edge maps and deletion synchronisation is only required the first time the edge is seen, every time afterwards the new *True* state may simply be inserted into its history and any properties updated in the same way as a vertex object. The process for fully local edge addition can be seen by following the left hand side of the main flowchart in Figure 4.12.

Case of local edge shared between workers within one Partition Manager

The second option is that the destination node is stored in the same partition, but in another Entity Storage and, therefore, controlled by a different worker. In this instance the source's Writer Worker, who first receives the command, will check if the source vertex exists, updating it as required. It may then check if the edge has previously been seen and, given that is has not, will place the new edge object (initialised by the update) into the outgoing edge map of the source. This will then be updated with any *False* change points within the history of the source. Alternatively if the source already exists, its history will just be updated with a new *True* state at the time of update. The worker will then forward the edge object (passing by reference) along with the update information to the worker who manages the destination node to complete its half of the update. The process for this can be seen in the right hand side of the flowchart in Figure 4.12. Upon receiving this update, the second worker will see if this is a new edge and, if so, create/update the destination node as required and insert the edge object into its incoming edge map. It will then extract any deletions from the destination node and send these back to the original worker to insert into the edge object. This second synchronisation is performed as the workers operate in parallel and could end up attempting to write on the same edge object, leading to a corrupted state. To alleviate this, only the worker storing the source of a local edge is allowed to edit it, the destination worker views this as read only. Fortunately this second synchronisation has to happen only once, with any subsequent edge additions requiring the second worker to insert a new *True* state into the history of the destination node at the update time. The process for shared local edge addition can be seen by following the right hand side of the main flowchart in Figure 4.12. The synchronisation steps can then be seen on the left hand side (in yellow) of Figure 4.13.

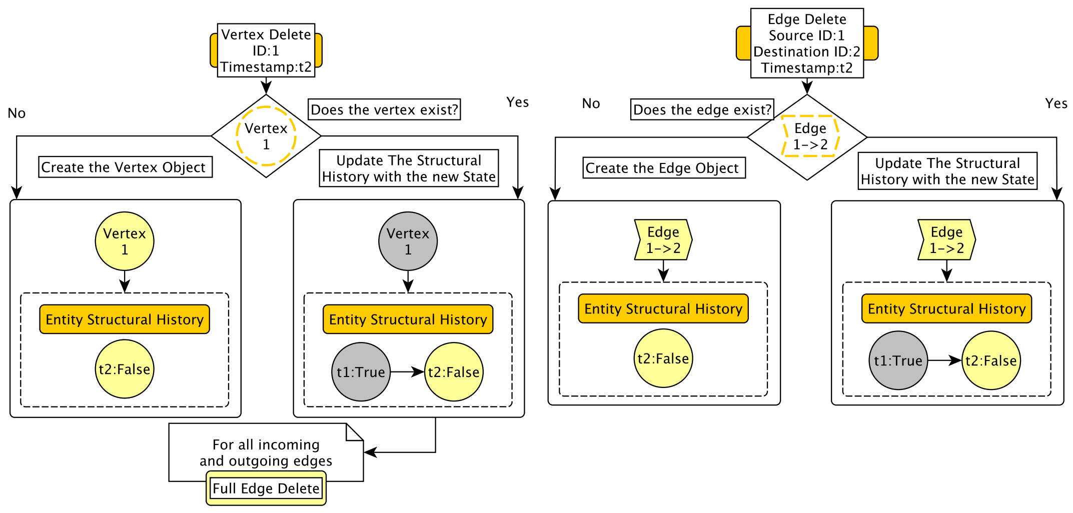


Figure 4.14: Flowcharts demonstrating the process for creating or updating entity objects with a deletion.

Case of split edge synchronised between workers across remote Partition Managers

The third option is that the destination node is stored in a separate partition i.e. a split edge. In this case, the worker storing the source vertex, and who initially receives the update, follows the same procedure as if it were a local edge shared between workers, i.e. only the synchronisation differs. For this synchronisation the worker will propagate the command details to the Partition Manager storing the destination vertex, along with any deletions present in the source vertex if its the first time the edge has been seen. The remote worker receiving this will then create/update the destination node and create its copy of the edge object, inserting this into the incoming edge map of the destination and inserting all deletions from both the source and destination into the edge. It will then take the deletions present within the destination and send them back to the first worker allowing the master copy to be synchronised with the remote copy. As before this second synchronisation only has to happen once at the first appearance of the edge. Alternatively, in the instance that the edge has already been seen, the destination will be updated along with the local split, but so will the edge as this is a completely separate object and any changes made to one version must be reflected in the other. The process for these remote synchronisations can be seen on the right hand side (in red) of Figure 4.13.

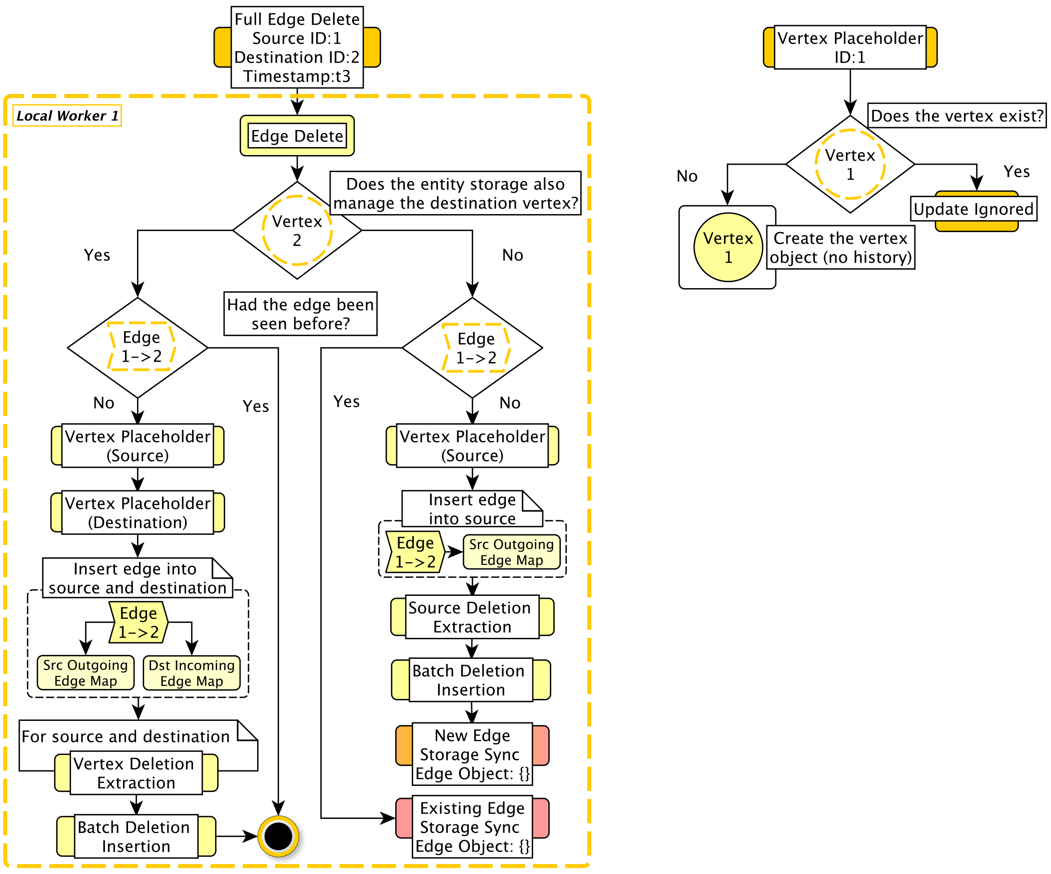

Full Edge Delete
Source ID:1
Destination ID:2
Timestamp:t3
Local Worker 1
Edge Delete
Does the entity storage also manage the destination vertex?
Vertex 2
Yes
No
Had the edge been seen before?
Edge 1->2
No
Yes
Yes
No
Vertex Placeholder (Source)
Vertex Placeholder (Destination)
Insert edge into source and destination
Src Outgoing Edge Map
Dst Incoming Edge Map
For source and destination
Vertex Deletion Extraction
Batch Deletion Insertion
Insert edge into source
Source Deletion Extraction
New Edge Storage Sync Edge Object: {}
Existing Edge Storage Sync Edge Object: {}
Vertex Placeholder ID:1
Does the vertex exist?
Vertex 1
No
Yes
Update Ignored
Create the vertex object (no history)

Figure 4.15: Full process for handling an edge delete update from the perspective of the worker who first received it.The sub-processes for vertex/edge deletion can be found in Figure 4.14.

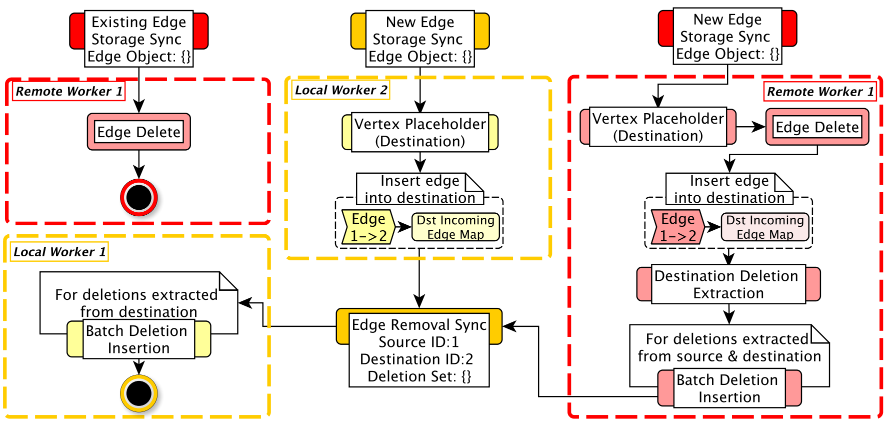

Existing Edge Storage Sync Edge Object: {}
Remote Worker 1
Edge Delete
Local Worker 1
For deletions extracted from destination
Batch Deletion Insertion
New Edge Storage Sync Edge Object: {}
Local Worker 2
Vertex Placeholder (Destination)
Insert edge into destination
Edge 1->2
Dst Incoming Edge Map
Edge Removal Sync
Source ID:1
Destination ID:2
Deletion Set: {}
New Edge Storage Sync Edge Object: {}
Remote Worker 1
Vertex Placeholder (Destination)
Edge Delete
Insert edge into destination
Edge 1->2
Dst Incoming Edge Map
Destination Deletion Extraction
For deletions extracted from source & destination
Batch Deletion Insertion

Figure 4.16: Different processes of synchronising an edge deletion depending on if it is the first time the edge has been seen and if the second worker is local or remote.

4.6.3 Removing Edges

The overall process for removing an edge is very similar to an addition/update. The key differences are that the edge's history is appended to specify it was deleted at the time of the update and that the update does not affect the history of the source and destination vertices. This second point means that if the vertices already exist nothing happens to them. However, if either vertex does not exist (i.e. the updates have arrived out of order) they will require instantiation. This creates a placeholder vertex with no history, which is not considered a member of the graph, but allows the edge object to be stored correctly and both the vertex/edge to await the arrival of their missing additions.

This subsection is summarised within the flowchart for full edge deletion on the left of Figure 4.15. This is supported by the flowchart for edge object deletion on the right of Figure 4.14 and the flowcharts for edge deletion synchronisation in Figure 4.16.

Whilst the full process for edge deletion including synchronisation differs somewhat from an addition, as all changes to the state of entities are additive/idempotent, the manner in which the edge object is handled is almost identical. As can be seen on the right of Figure 4.14, when an edge delete is processed, it is initially checked to see if the edge object exists, inserting a *False* state into its history at the time of the update if it does. This will happen even if the newest state is also *False* as an addition may have been delayed and need to be slotted in-between; analogous to the two *True* states discussed when adding an entity. If the edge has yet to be seen, the object will be created and its history will be initialised with the *False* state. Whilst this conceptually means the edge does not actually exist within the graph, it allows the delayed edge add to be placed in when it arrives and means this update is not lost.

Case of local edge managed by one worker

When synchronising the removal of an edge, the same three possibilities for synchronisation exist: local and managed by one worker; local, but shared between workers; and split between remote workers. In the first case, it is initially checked if the source and destination exist, but instead of creating full vertex objects for those not present, a placeholder object is created. This is a vertex which has no history and is, therefore, not technically part of the graph, but does provide a place to store the edge object. The process for this can be seen on the right hand side of Figure 4.15. This is done as an edge removal should have no bearing on the state of its source/destination, but is important for the corner case of an edge deletion arriving before the edge or associated vertices have even been created and ensures that this update is not lost/ignored. If this is the first time the edge has been seen, it will be placed into the incoming/outgoing edge maps of its source/destination and all deletions present in these will be inserted into the edge object. This can clearly end up with an edge with several consecutive *False* states, but does ensure no matter what the update order the same history is achieved. This process for local edge deletion can be seen on the left of the main flowchart in Figure 4.15.

Case of local edge shared between workers within one Partition Manager

As with the addition, if the edge is shared between workers, the Writer Worker storing the source vertex will perform the same process, but will omit the destination node as this must be handled by the worker that controls it. This includes creating/updating the edge object, creating a placeholder source vertex if required and inserting any deletions from the source which the edge may have missed. Following this, if the edge is new, this first worker will send the edge object (pass by reference) and update information to the worker hosting the destination node. This worker will create a destination placeholder if the vertex does not exist or gather any deletions from it if it does, sending them back to the first worker to complete the update. If the edge did previously exist, the synchronisation with the second worker is unnecessary as the destination vertex is unaffected by the deletion. This process for shared edge deletion can be seen on the right of the main flowchart in Figure 4.15. The process for local synchronisation can be seen (in yellow) within Figure 4.16.

Case of split edge synchronised between workers within remote Partition Managers

Finally, for a split edge between remote workers, the initial worker again performs the same process as if the edge was shared locally. When reaching the end of its handling of the source and edge objects, it packages up the update information and forwards the update to the remote worker dealing with the destination node. In the instance of a new edge this remote worker will check if the destination node exists, creating its own placeholder if not, then creating its own copy of the edge and inserting any deletions from both the source and destination. The deletions from the destination are then sent back to the primary worker to ensure the remote and master copies are in-sync. Unfortunately, as this worker is not on the same machine, it requires synchronisation whether the edge is new or not, as the deletion must be available within the remote copy. Therefore, in the case of an existing edge, the remote worker will insert the new *False* state into the history of the remote copy, but leave the destination vertex untouched. The process for remote deletion synchronisation can be seen (in red) within Figure 4.16.

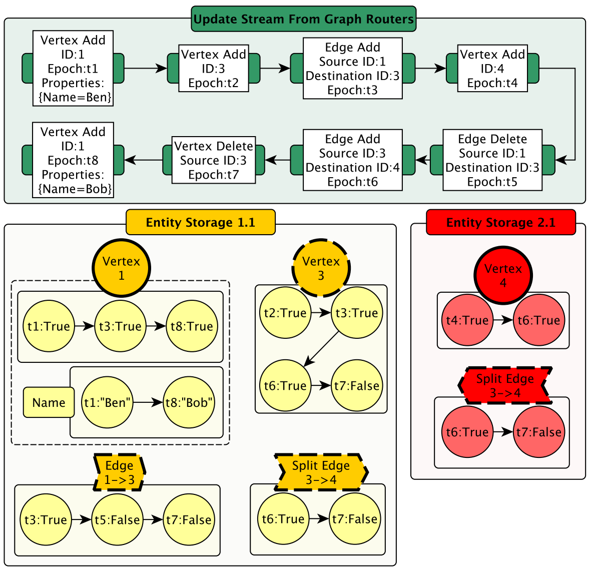

Update Stream From Graph Routers
Vertex Add ID:1 Epoch:t1 Properties: {Name=Ben}
Vertex Add ID:3 Epoch:t2
Edge Add Source ID:1 Destination ID:3 Epoch:t3
Vertex Add ID:4 Epoch:t4
Edge Delete Source ID:1 Destination ID:3 Epoch:t5
Edge Add Source ID:3 Destination ID:4 Epoch:t6
Vertex Delete Source ID:3 Epoch:t7
Vertex Add ID:1 Epoch:t8 Properties: {Name=Bob}
Entity Storage 1.1
Vertex 1
t1:True
t3:True
t8:True
Name
t1:"Ben"
t8:"Bob"
Vertex 3
t2:True
t3:True
t6:True
t7:False
Edge 1->3
t3:True
t5:False
t7:False
Split Edge 3->4
t6:True
t7:False
Entity Storage 2.1
Vertex 4
t4:True
t6:True
Split Edge 3->4
t6:True
t7:False

Figure 4.17: An example stream of graph updates & the equivalent temporal graph once distributed between two Partition Managers. Entities with dotted lines indicate their most recent state is a deletion.

4.6.4 Removing Vertices

The removal of a vertex is the most complex of the updates and affects the history of all edges with it as a source or destination. The change to the vertex itself is minimal, requiring an object to exist and its history to be appended with a deletion at the time of the update. This then generates edge deletions for all associated edges which are handled exactly as above, including synchronising where required. The real issue is that if updates are out-of-order and an edge creation arrives late, it may miss this information. To ensure this doesn't occur, whenever a new edge object is created all deletions present in the history of its source and destination must be appended into its history. This requires additional synchronisation if the edge is split between partitions, but is only required once, not every update.

This subsection is summarised within the flowchart for vertex deletion on the left of Figure 4.14.

As with the removal of an edge, the initial process for removing a vertex is also additive and handled in a similar manner to a vertex addition. The worker receiving the request will check if the vertex has previously been seen and if so add the new *False* state into its history at the timestamp provided. Alternatively, if an object is yet to exist, one will be created and the structural history initialised with this *False* state. The issue that a vertex removal causes is that all adjoining edges must also be removed from the graph so that these are not left hanging. This creates the possibility for race conditions, as 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 4.17, if the command which deleted vertex 3 arrived before the command which added edge 3→4, only edge 1→3 would exist when the deletion is executed. 1→3 would, therefore, be updated with the new *False* state, but 3→4 would miss this information. This is the reason for the extraction of removals from all source and destination nodes seen above, as it ensures all edges contain this information, not only objects which existed in memory at execution time. This overall process can be seen on the left of Figure 4.14.

To perform these edge removals the previously established method can be leveraged, only requiring the correct workers to be informed. For all outgoing edges, the worker may handle this itself as it is in control of the source node. This means it will insert a *False* state into the history of all the outgoing edges at the timestamp within the vertex deletion. Those which are split edges will then be synchronised by informing the remote worker handling the destination vertex of this new deletion. For incoming edges, if the worker also handles the source vertex it will update the edge itself as it is fully local. If the source is handled by a different worker in the same partition, the worker cannot edit the edge object so will request the other worker to complete the removal. Finally, if the source is remote, the worker will update its copy of the edge and then request the worker handling the source node to update the master copy. It should be reiterated that all of these requests are handled completely asynchronously and the worker may continue processing other updates whilst the edge synchronisation occurs.

4.7 Watermarking

This section discusses how Raphtory tracks each partition's progress in ingesting and synchronising the stream of updates and, therefore, where in the history of the graph is safe to analyse. This is managed via a watermarking system where the stream between each Router/Writer pair is viewed as a channel and all updates which flow through it are allocated a unique ID. These IDs are sequential and are added into an ordered queue for the channel once synchronised. The set of queues for all channels is viewed as a vector clock and allows the writer to establish where it believes is safe for analysis in the graph's history. The minimum value reported across all writers is then considered the global 'live graph' time. Several corner cases are additionally addressed here, notably partition starvation and the handling of bounded datasets.

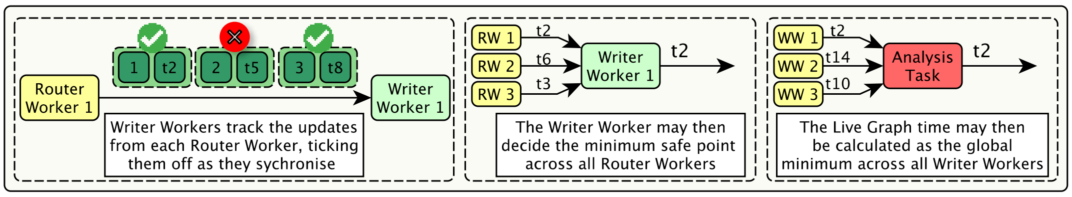

Router Worker 1
1
t2
2
t5
3
t8
Writer Worker 1
Writer Workers track the updates from each Router Worker, ticking them off as they sychronise
RW 1
RW 2
RW 3
t2
t6
t3
Writer Worker 1
t2
The Writer Worker may then decide the minimum safe point across all Router Workers
WW 1
WW 2
WW 3
t2
t14
t10
Analysis Task
t2
The Live Graph time may then be calculated as the global minimum across all Writer Workers

Figure 4.18: An overview of the watermarking process.

The update semantics defined in Section 3.7 and implemented above ensure any updates coming into the graph will be placed in the correct position within the history of affected entities and will eventually synchronise any split edges involved. Whilst this ensures updates will not be overwritten/lost, it provides no guarantees on when an update will complete and, therefore, there is no way to know if all partitions have fully synchronised up until a given point in the stream. This is made additionally harder by vertex deletions, where multiple partitions need to synchronise, and with data skew where certain partitions are receiving many more updates than others. Not knowing where in the stream is safe to execute on is an issue as, if the user wishes to materialise flattenings for analysis, it is unknown whether all updates for the given *flattening_end* have been ingested and, therefore, whether the correct result will be returned. As such, an additional procedure must be established to determine the latest point of time in the history where it is safe to analyse (i.e. the Live Graph time). This is difficult as updates arrive at each partition from many Graph Routers in parallel and, even though an update may arrive before another, it could require a larger amount of synchronisation, such as when removing a high degree vertex, and therefore not complete until much later. Additionally, each partition only sees a subset of the updates; it cannot confirm on its own what time is safe and must confer with its peers to gather the full picture.

To manage this in Raphtory, updates extracted from the raw data are watermarked from initial generation through to the final point of synchronisation. Writer Workers then track all updates received from each Router Worker, ticking these off as they synchronise. This may then be used to calculate the latest safe update time from each Router Worker based on the constraint that they are arriving in chronological order, established in Section 4.4. The time an individual Writer Worker considers safe may then be garnered by viewing the update times from all Router Workers as a vector clock[139]. This clock contains the times of the latest synchronised update from each Router Worker (where all prior updates are also synchronised) with the safe time extracted as the minimum of these. The Live Graph time is then the minimum value reported across all Writer Workers as they agree that updates prior to this time have been ingested and synchronised. An overview of this may be seen in Figure 4.18.

This section explores the implementation of the watermarking process within Raphtory, spread across three sub-processes. The first of these is the manner in which graph updates may be uniquely identified, which Router Worker they originated from and how the Writer Workers may asynchronously confirm that an update has been completed. Here it is also discussed how these acknowledgements are stored, whilst minimising memory footprint. The second sub-process is how these individual acknowledgement queues may be built into a vector clock, whereby the Writer Worker can decide the latest time it may claim is safe from the partial information it has received. This is followed by how these partial safe times are aggregated together upon an analysis request, deciding the global Live Graph time and if a given *flattening_end* can be materialised. The final sub-process deals with inequality in the amount of updates each partition receives, ensuring no starved Writer Workers are holding back the Live Graph time via overly conservative watermarking. This process also handles the manner in which a Spout ingesting a bounded dataset may propagate the final update time downstream, allowing all Writer Workers to report this value.

4.7.1 Update Tracking and Synchronisation Acknowledgement

To begin the watermarking process each Router Worker is instructed to keep a count of how many messages it has sent to each Writer Worker. As data tuples from the Spout are only processed by one Router Worker, when an update has been extracted from the data and sent to a partition it may be uniquely identified across all updates. This unique identifier (UID) is built from a combination of the ID of the Router Worker who sent it, the ID of the Writer Worker receiving it and the new count of updates sent between these, after incrementing. For instance, it can be seen on the left of Figure 4.19 that Router Worker 2 has sent Writer Worker 1 twelve updates so far. The next update sent between these will, therefore, receive the UID *[(2→1),13]* referring to the 13th update between this pair of actors i.e. *[(Router Worker ID→Writer Worker ID),Update Count]*.

To record which updates have been fully synchronised, Writer Workers maintain a queue for each Router Worker, storing the count associated with completed updates alongside its event time. When a Writer Worker receives an update it will only insert the corresponding update count into the Router Worker's queue once satisfied it has been fully synchronised. An example of these queues may be seen on the right of Figure 4.19. Within the top queue it can be seen that all twelve updates Writer Worker 1 has received from Router Worker 2 have been fully synchronised, as the update counts are all stored within the queue with no gaps in the increments. On the other hand, for Router Worker 1, whilst updates *[(1→1),2]*, *[(1→1),4]* and *[(1→1),5]* are included in the queue, signifying they are fully synchronised, update *[(1→1),3]* is not included. This means Writer Worker 1 believes the update is still synchronising, important for deciding on when in the stream it believes is safe (discussed further in Section 4.7.2).

Tracking update synchronisation builds on top of the established updating protocol discussed in Section 4.6, with the update's UID included in all synchronisation messages sent between Writer Workers. As with the methods for synchronising different updates, there are different processes for a Writer Worker to decide if a given update has been completed by all involved partitions and may, therefore, be inserted into the Router Worker queue. For vertex addition and

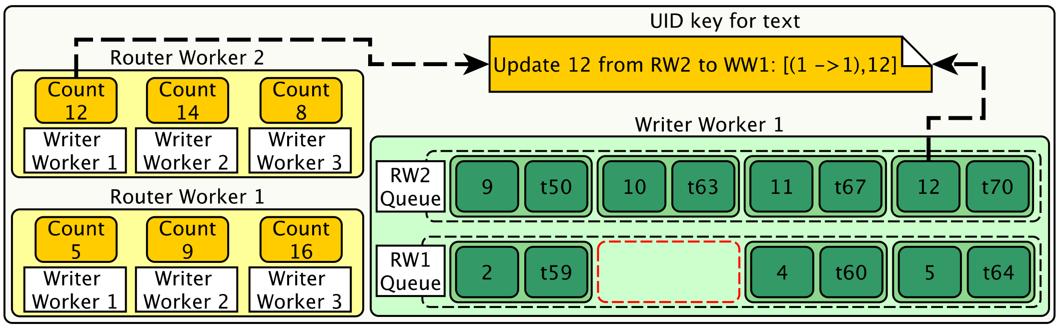


Figure 4.19: On the left - the count of updates sent from each Router Worker to each Writer Worker. On the right - the queues of completed updates for each Router Worker within Writer Worker 1. Note update *[(1→1),3]* is missing, highlighted in red. Top - The key for how update UIDs are expressed in the text.

any update to a fully local edge, as no other Writer Worker is involved, its count/timestamp is inserted into the sending Router Worker's queue once execution has completed within the Entity Storage.

For updates to split edges, once the second Writer Worker has completed its portion of the update, it will return an acknowledgement to the primary Writer Worker, including the UID and timestamp it was sent with the original request. The count may then be extracted from the UID and inserted into the corresponding Router Worker queue, signifying that the Writer Worker is now content that the update is synchronised. This approach is taken so that updates may continue to complete asynchronously without Writer Workers having to track all outstanding requests; minimising management overhead and memory footprint.

Finally, when a vertex deletion is executed, the Writer Worker will have to track the number of edge deletion synchronisation messages it sends out, which are recorded in a map referenced by the UID of the vertex deletion. The edge deletion synchronisation requests are then acknowledged in exactly the same manner as any other edge update, but when the confirmation is received back by the primary Writer Worker the count of outstanding acks is first checked and if still greater than 1 is decremented. Once this count reaches zero it means all of the edge deletions have synchronised and this update may be considered complete, allowing it to be inserted into the Router Worker queue.

4.7.2 Router Worker Vector Clocks and Live Graph Time Extraction

To enable the Writer Worker to extract the latest safe update, Router Worker queues are ordered with the lowest UID at the head. As updates are assumed to arrive in chronological order from each Router Worker (as discussed in Section 4.4), they may be iterated through until the end is reached or the UID count increments by more than 1, meaning an update from this Router

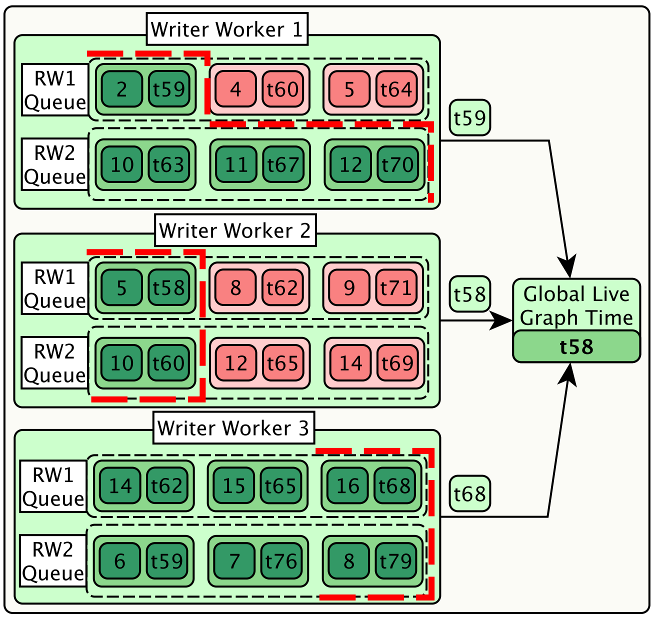


Figure 4.20: An example of three Writer Workers and their queues from two Router Workers. The safe updates within each queue which the Writer Workers may use to work out their safe time are in green. These may then be aggregated into the global Live Graph time. Those in red have prior updates yet to synchronise and are, therefore, excluded for the time being.

Worker has yet to be synchronised and is, therefore, only safe up until the prior message time. As an example of this, on the left of Figure 4.20 are the queues for three Writer Workers. Looking first at Writer Worker 1, it can be seen that the queue for Router Worker 2 is synchronised until the last update (*[(2→1),12]*) as all updates prior have counts incrementing by 1. The same cannot be said for those received from Router Worker 1, where update *[(1→1),3]* has yet to be added to the queue meaning it is currently unsynchronised. Moving across to the associated timestamps in these queues the Writer Worker may claim that $[t_{59}, t_{70}]$ are safe with respect to their source Routers. From these the lowest value (t_{59}) may be extracted to use as the overall safe timestamp for this worker and its associated Entity Storage. The lowest is used here as the Writer Worker cannot claim to know any point above this time is safe, as the Router Workers may have sent updates to other partitions within this range which are yet to complete.

The same process may be completed within Writer Workers 2 and 3, with them reporting safe times of t_{58} and t_{68} respectively. The times from all workers may then be aggregated to establish the global Live Graph time, again by selecting the minimum (t_{58} from worker 2) as this ensures that all updates prior to this point have been synchronised across all partitions and, therefore, any time prior to this may be safely analysed.

4.7.3 Router Time Sync and Partition Catch-up

As more updates arrive via the Router pool and begin to synchronise, the described watermarking method will push forward the Live Graph time, permitting the inclusion of newer updates in analysis, whilst ensuring nothing is executed on an 'incomplete' flattening of the graph. Unfortunately, graph workloads often have a data skew with a small number of entities attracting a large portion of the updates/edges[9], whereas other entities are not updated for long periods of time. Depending on how clustered these highly updated nodes are within the partitions, some workers may receive updates more frequently than their peers. This will naturally cause them to report the highest safe times as they see the newest timestamps first. The other workers, on the other hand, cannot know time has progressed until they receive new updates. This can lead to a flattening on a recently ingested time being blocked for longer than necessary, even if all the partitions have synchronised all updates, as an update starved partition is yet to see this time and, therefore, cannot report it as safe. This is especially an issue for bounded datasets as eventually the incoming updates will complete and if the safe time is still the lowest timestamp from the pool of Writer Workers, a set of the final events will never be considered safe. For example, looking back at the queues within the three workers in Figure 4.20, if no more updates were sent, t_{64} would be their final reported Live Graph time and the updates between $[t_{64} - t_{79}]$ would never be analysable.

To solve the update starvation issue, the Router Workers will periodically announce their most recently seen event time to all Writer Workers. These are considered an update from the counts perspective, arriving at each Writer Worker with their own UID, and are inserted straight into the queues. Whilst this does not specify any changes in the graph, once all prior updates have synchronised, it will allow the writer to safely report a more recent time as it knows that no updates prior to the sync time will be arriving. To solve the second issue of updates in a bounded dataset being unanalysable, once the Spout has finished ingesting all the tuples from its assigned data source, it will send a message to the last Router Worker it contacted, confirming receipt of the final tuple. The receiving Router Worker will forward the event time extracted from this tuple to all its peers who may then be used in the next round of time sync updates between the Router Workers and Writer Workers, allowing the latter to report the final tuple time and for this to be accepted as the global Live Graph time.

4.8 Partition Archivist

This last section discusses how the Archivist (introduced above) manages the state of the graph within its partition and the memory usage of its host machine. The first stage of this is graph persistence, making sure the history of the graph is saved. This is done by combining periodic snapshots of the graph alongside the recording of all updates between each Router and Writer Pair. In parallel with this, Archivists poll their host machine's available memory, ensuring it does not exceed a safe threshold. If this does occur the Archivists coordinate to iteratively remove the oldest history from memory until all hosts are back below this threshold. If a user wishes to query a time which has been removed by this process the flattening may instead be rebuilt by re-ingesting the closest snapshot and replaying the stored updates between the snapshot time and the user's chosen time. The scheduling of this process is discussed below, but it is acknowledged that much work must be done to allow the Writer, Archivist and Reader to optimally run in parallel.

The temporal graph model and its implementation within the Partition Writer resolves the challenge of update ordering and synchronisation, but this model does bring the caveat of an increased memory overhead as all previous state and property values are maintained in-memory. This means that for larger or unbounded datasets, the combined RAM of all Partition Manager host machines sets a hard limit on the overall size and temporal depth of the graph. To alleviate this, each Partition Writer in Raphtory is paired with an Archivist, who is tasked with managing the amount of memory utilised within the partition by running checks on the stored entities and offloading the oldest history/entities onto secondary storage. This allows new updates to be inserted into the graph, but also ensures history can be retrieved for analysis if the user needs to go back further than the hardware limitations will allow.

As was shown in Figure 4.8, the Archivist/Writer pairing means archiving tasks are also carried out by the pool of Writer Workers. This was done to avoid the complex locking required to allow a separate set of workers to mutate the state of the vertices in parallel with the Writer Workers, as well as the possibility of deadlocks or corrupted state. The Archivist has three main tasks which it interleaves with the ongoing ingestion of data. These are: graph persistence, snapshotting the graph at globally watermarked safe timestamps; archiving, checking which entities may be safely removed from the graph without affecting analysis; and history retrieval, bringing back history into the in-memory graph to allow the user to analyse flattenings which have been pushed to disk.

4.8.1 Graph Persistence

The first stage in the archiving process is to persist the graph entities and their history to secondary storage. This prevents data loss in the case of system errors and simplifies the actual

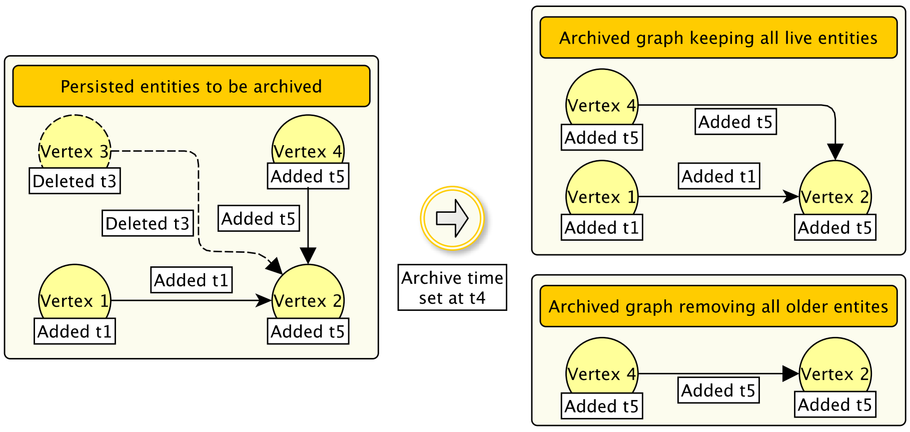


Figure 4.21: Example graph ready for archiving after it has been persisted and the two possible new graphs depending on if 'alive' entities are permitted to be removed.

archiving stage when memory needs to be freed up. As in ImmortalGraph[12], persistence in Raphtory is broken into two components, a snapshotting mechanism and an update log. For the first of these, during each run of the Archivist's persistence task a snapshot of the graph is generated at its latest safe point, as specified below in Section 4.8.4. A snapshot in this instance consists of the vertices and edges present at the snapshot time and the most recent change point of their properties. The most recent update time of each entity (prior to the snapshot time) must then be stored to allow the state of each entity to be rebuilt correctly. If this was not recorded we would only know that the entities existed at the time of the snapshot, not when they were added, losing some temporal resolution. Once re-ingested analysis performed upon this data may return different results and must, therefore, be avoided. Finally, these entities may also be stored alongside the ID of the Writer Worker in charge of them, allowing each machine to retrieve only its entities when reading back into memory.

As snapshots only handle set points in the history of the graph and are created periodically, missing out changes in-between, these are combined with an event log, which can fill in the gaps between snapshot epochs. To provide this event log, all updates sent out from Graph Router Workers must additionally be recorded in a message queueing system (such as Kafka[79]) with each worker having its own topic which can be replayed (as discussed in 4.8.3). A message queueing system is suggested over the inbuilt facilities of Akka Persist as it decouples the data from the machine hosting the Router, can be replicated across brokers and, by additionally recording the offsets associated with each snapshot, the workers can easily re-ingest only the updates between two epochs.

4.8.2 Archiving History

Once entities have been persisted they are safe to be archived, i.e. removed from the in-memory graph to free up space. For this, the Writer Worker iterates over all vertices within its Entity Storage, removing structural events outside a set temporal depth (established in 4.8.4). The worker will then do the same for the edges it controls within each vertex, i.e. outgoing edges and split incoming edges where the source node is remote. Any property values outside the archiving window will also be removed, unless its the most recent value which will always be kept. If all events in the structural history are older than this depth, the whole entity may be considered ready for archiving and, therefore, completely removed from the graph. There are two types of archiving that may be considered here. The first is that an entity is only fully removed if its latest state denotes a deletion, i.e. it would not be involved in any analysis. The second is that all entities outside of the acceptable temporal depth are removed irrelevant of last state. By only removing deleted entities we ensure all vertices and edges present at the chosen *flattening_end* are included within analysis and, therefore, results cannot be affected by the available memory. However, the machines the Partition Managers are deployed over must have at least enough memory for the most recent version of the graph. If not the dataset will have to be re-ingested on a new deployment with more partitions or a greater allocation of RAM on each machine. This second option ensures there is always enough memory for the most recent entities, but will miss very old connections similar to a windowed flattening of the data (as discussed in Section 3.4). An example of this can be seen in Figure 4.21, where vertex 1 and its edge to vertex 2 are added at the very beginning of the graph history, but will only be removed along with vertex 3 in this latter case. Whilst this second version could perhaps be considered a better solution to the issue of memory constraints, it would mean that the same query of the graph could return drastically different answers depending on how long the deployment had been running or how much data had been ingested. This would cause a host of issues for testing and repeatability, but more importantly would make Raphtory unusable in a production environment as it would not be trusted to give a correct answer. Therefore, entities are only fully removed if their latest state denotes a deletion.

4.8.3 History Retrieval

When the Partition Reader receives an analysis request (as explained in Section 5.4.1) it must first check if the timestamp given is available within the partition as it may have yet to be ingested, currently be unsafe, or have been archived due to memory limitations. In the latter case the Reader notifies the Archivist of the timestamp required and tasks it with orchestrating the re-ingestion into the in-memory temporal graph. This is completed in a two step process which can be seen in Figure 4.22. In the first step the Archivist finds the closest snapshot prior to the timestamp requested and queries the datastore to retrieve the vertices and edges within it for its partition. As the rows are returned they are converted into graph update commands

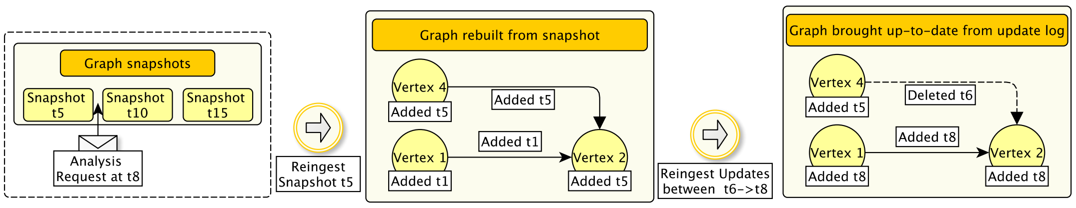


Figure 4.22: Process for retrieving a flattening of the graph at a time which has been archived due to memory constraints.

and pushed to each of the relevant Writer Workers for ingestion into the model. This can be seen in the first box of Figure 4.22 where the closest snapshot to t_8 is t_5, which has brought back the three vertices from Figure 4.21, establishing their history with the latest update prior to the snapshot time. When this is completed, the Archivist will broadcast to the Router pool requesting they re-ingest all changes between the snapshot time and the analysis *flattening_end*. Each Router Worker then connects to its message queueing system topic and pushes any updates within this period back to the Writer Worker who originally received them, to once again build the entity and propagate the update to other partitions as described in Section 4.6. This can be seen on the right hand side of Figure 4.22, where edge $2 \rightarrow 4$ was actually deleted during this period and edge $1 \rightarrow 2$ received an update to its state. Once these have all been ingested, the flattening at the requested timestamp will be ready and the Partition Reader will be allowed to proceed with analysis. The watermarking technique described in 4.7 is leveraged here to confirm this, but as a parallel track to the normal updates so as not to affect the ongoing ingestion. The Archivist will then be paused until the completion of the analysis, at which point an archiving cycle will run to once again offload this data if there is not the space for it alongside the live graph. This is, therefore, a process similar to caching and poses issues akin to the temporal partitioning within Section 3.5. Maintaining the most appropriate history within memory, based upon use workloads, is an area of ongoing research for Raphtory and is discussed further in Section 7.2.

4.8.4 Scheduling and Thresholds

When Raphtory is deployed, each Archivist will begin monitoring the Entity Storages within its partition and the available memory on its host machine. Periodically it will check if any updates have occurred and, if so, will schedule a snapshot to be built at the latest safe timestamp. To decide upon this timestamp, the Archivists will send their latest safe time (as specified in Section 4.7) to the Archivist in Partition Manager 1 (the de facto quorum leader), who upon receiving all local minimums will calculate the global minimum safe time and broadcast this back to its peers. Once this is received back, the Archivists may begin requesting the Writer Workers to send the state of each entity at this time to the chosen datastore. In order that ingestion is not fully paused, and to avoid I/O bottlenecks, this is completed one Writer Worker at a time instead of in parallel.

In conjunction with this, the Archivists continuously poll the used memory to ensure it is below a set threshold. If this threshold is broken, an archiving cycle is scheduled to run as soon as possible, i.e. straightaway, as long as a snapshot is not currently being saved. Whilst the user may change this value, the default threshold is set to 80% of the available memory. This gives the Partition Managers some buffer room such that if they are filling up fast they may still complete the archiving cycle before running out of space. As the range of times associated with data ingested by Raphtory can vary from seconds to years, the temporal cut off for which updates and entities will be compared cannot be based on a static amount. Instead it is set as a percentage of the difference between the time of the oldest in-memory update and the latest snapshot time, adjusting as more snapshots are created and as new data is ingested. This is again scheduled one Writer Worker at a time so not to fully block ingestion. Once archiving is complete across all workers, the Archivist will check to ensure used memory is back below the threshold. If this is not the case, the process is restarted with the range beginning from the new oldest update time. This continues until the used memory is back below the threshold or no more history may be removed, at which point the user will currently have to redeploy on larger machines as their graph has become too big to manage with the established resources. Assuming this is not the case, once the archiving cycles have finished, the oldest update time will be sent to the Archivist quorum leader who will broadcast this new cut off to all peers, requesting them to perform an archiving cycle to the same depth, bringing the whole graph in line.

As Raphtory may be deployed over heterogeneous machines with varying RAM availability, and because of the possibility of some partitions handling more updates than others, it was initially considered not to synchronise the temporal cut off across Archivists. This was because the most recent cutoff would have to be global, with the majority of partitions already below the threshold and, therefore, forced to remove history which they had room for. This also increased the disruption to ongoing ingestion via additional archiving cycles and meant that if a user requested an archived flattening all partitions had to read the snapshot back from the datastore, not just those running out of space. However, by allowing partitions to vary their temporal depth there will be many cases where split edges exist in one machine, but not the other, breaking the update semantics implemented in Section 4.6.

4.8.5 Future Exploration

Whilst the goal of the Archivist and its overall process structure have been established, there are still many questions to be answered before it may be fully rolled out alongside the Writer and Reader. The first of these is the default datastore to be utilised for snapshots. There are many good options for this, but a full investigation into the best option must be performed before one is chosen. The second is to investigate scheduling between the Reader/Writer/Archivist to ensure the archiving process does not cause a reduction in throughput or interfere with any objects which are currently being analysed. Thirdly, when re-ingesting history into the graph this obviously requires memory, which may not be available and, therefore, requires the temporal depth of the actual graph to be reduced. This is a trade-off which has yet to be fully explored and could cause many issues if not managed well. Finally, as archiving fully deletes entity objects from memory, there is a much higher risk of update loss due to interleavings, where one partition believes an edge to exist whilst the other does not. This is mitigated somewhat by keeping all partitions at the same temporal depth, but must be explored further to ensure there is no point or update order which could end in a corrupted state. These challenges and other trialled ways of reducing the used memory are discussed in Section 7.2.

4.9 Summary

In this section we have discussed how the distributed temporal graph model and its update semantics, defined in Chapter 3, have been implemented within the Raphtory platform. We split this between three key components, the Spout, Router and Partition Manager. Of these the role of the Spout is to act as the stream source, pulling data into Raphtory. The Router may then take these data tuples and parse them into graph updates, decoupling ingestion and graph modelling, allowing the same data to be built into different graphs. The graph updates are then handled by the Partition Managers who are each responsible for building and maintaining a partition of the temporal graph from them.

We discussed how the Partition Manager splits its responsibility over the partition between three sub-components, the Writer, Reader and Archivist. The Writer and its pool of workers are responsible for the update handling and synchronisation between partitions. These workers also track the completion of update synchronisation via watermarking to ensure no analysis executes on a point of time where updates are only partially complete. The Reader is responsible for this analysis on graph flattenings within the partition and is discussed further in the next chapter. Finally, the Archivist manages the available memory of the host machine, creating snapshots of the in-memory graph and removing older history/entities to make room for newer updates.

Chapter 5

Raphtory Analysis

5.1 Introduction

Once data ingestion has been established and the user defined graph is being maintained, the subsequent milestone is to perform analytics upon it, extracting novel metrics and insights. As with the ingestion this comes with its own set of challenges to be solved. Firstly, as the graph is built from a stream, 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 must be managed correctly though, as executing in parallel with ingestion and on the most recent graph may easily yield incorrect results, whether from race conditions, delayed updates or unsynchronised state.

Secondly, 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; comparing and contrasting. To streamline comparisons between historic points, this process should be transparent from the perspective of the user, allowing the same algorithms to run on the Live Graph and any generated flattening. Furthermore, whilst deltas between flattenings are an advantage of the temporal model, the true benefit lies in the structural and property histories of an entity being queryable within the analytical API, enabling time-aware analysis. This brings many challenges of how much information to expose to the user, how to do it in a way which can be deployed independent of the chosen time, as well as not interfering with ingestion and graph maintenance.

Finally, one of the largest overheads for the deployment of distributed analysis is the time taken to re-ingest data[140], 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 is offline for an undetermined period of time if there is a need to change some ongoing analysis. It should, therefore, be possible to submit new tasks without producing downtime for the graph.

This chapter discusses the different types of analysis possible within Raphtory which have emerged whilst tackling these challenges, as well as how the challenges themselves were overcome. To orchestrate analysis, Raphtory deploys an *Analysis Manager* alongside the ingestion components discussed above. This provides a REST API[141] for users submitting jobs to run on the graph, spawning *Analysis Tasks* for each one received. These oversee the workflow of graph algorithms, communicate the request to each of the *Partition Readers* and aggregate their returned results. Algorithms are then specified via the *Analyser* class which provides a time-independent API for analysis, aggregation and data publishing. These then run in conjunction with a *graph lens* which generates the user's desired graph flattening, returning *Entity Visitors* which safely expose the correct graph state. The implementation of the analysis components, as well as both APIs, are explained within this chapter and explored further via concrete examples in Chapter 6.

5.1.1 Chapter Roadmap

Section 5.2: Temporal Graph Algorithms The temporal graph algorithms section expands on the discussion in Section 2.3, exploring the full structural scope of graph algorithms. Here we additionally overview the different types of analysis Raphtory may perform, as well as the different ways time may be incorporated into a Raphtory query.

Section 5.3: The Raphtory Graph Analysis Model Following this overview we take a deeper look into the Raphtory analysis model, the API's that a user interacts with and the classes they must create to define a new algorithm.

Section 5.4: Underlying Implementation of Analysis in Raphtory This investigation continues to the core of Raphtory's analysis, looking under the hood at how these user defined classes are handled and how Raphtory builds graph flattenings for the algorithms to be run on.

Section 5.5: Submitting Queries - Analysis Manager Lastly we briefly touch on how a user interacts with a running Raphtory deployment, submitting new queries consisting of a defined algorithm and time parameters within which to run it.

5.2 Temporal Graph Algorithms

5.2.1 Structural Scope of Algorithms: From Queries to Analytics

Before discussing dynamic and temporal elements, it is important to understand the structural scope a graph algorithm may have. Figure 5.1 provides a spectrum of possible analysis which may be applied to a standard graph. This spectrum splits algorithms based upon their structural scope, i.e. the portion of the graph they interact with[27]. This can often be orthogonal to the actual difficulty of an algorithm, but is important in understanding the graph analysis ecosystem.

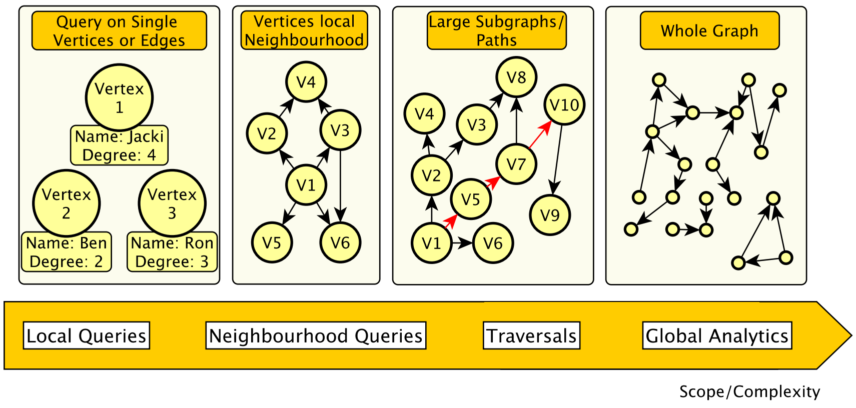


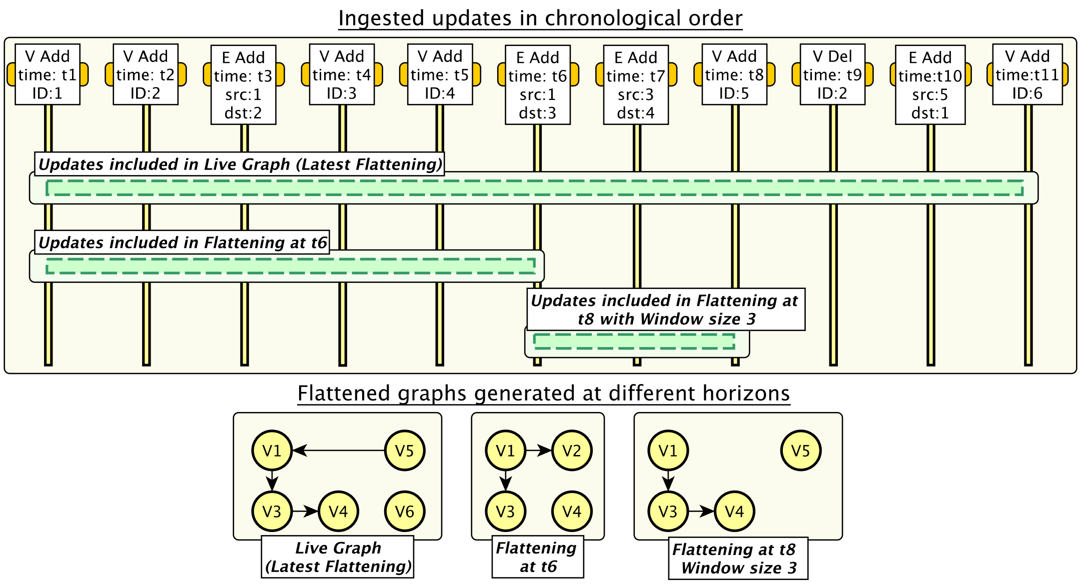
Figure 5.1: The varying complexity of analysis conducted upon graphs, ranging from local queries on singular entities to global analytics over the whole graph. Based on Figure 21 from [27].

At the left of the spectrum, or the smallest scope, are queries on singular entities. This may refer to requesting the value of a property or perhaps some structural value not involving neighbours such as the in-degree of a vertex. Moving further right are queries which explore the local neighbourhood of a vertex, for example a person's friends, or friends of friends, as specified in the LDBC interactive workload[142]. At this scope there is also graph pattern matching or network motifs[51], which can vary from very simple through to intractable[143]. The third category expanding past this is when a substantial portion of the graph begins to be interacted with, either through complex many-hop queries or graph traversals, such as random walks[144] or graph diffusion[56]. Finally the forth category, global analytics, is when the entire graph is involved in the computation. Again this may be something simple such as average in-degree for all nodes, or involve more complex and iterative algorithms such as PageRank[58], weakly connected components or graph diameter. As explored in Chapter 2, within the graph processing ecosystem there is often a divide between graph analytics and the remaining categories, with many Pregel-like[7] systems focusing exclusively on analytics[145] and graph databases such as Neo4j[34] tackling the rest.

5.2.2 Temporal Scope of Algorithms

Beyond Snapshots: Graph Flattening and Windows

The manner in which temporal scope is often explored in the context of graphs is by establishing the chosen algorithm as a time-evolving query[37]. Here the algorithm will be reapplied as the graph changes throughout time and the user may inspect the delta between each retrieved result. However, many systems only perform this on the most recent version of the data or historic snapshots taken at large intervals, often requiring re-ingestion. This limits the temporal



Ingested updates in chronological order
V Add time: t1 ID:1
V Add time: t2 ID:2
E Add time: t3 src:1 dst:2
V Add time: t4 ID:3
V Add time: t5 ID:4
E Add time: t6 src:1 dst:3
E Add time: t7 src:3 dst:4
V Add time: t8 ID:5
V Del time: t9 ID:2
E Add time:t10 src:5 dst:1
V Add time:t11 ID:6
Updates included in Live Graph (Latest Flattening)
Updates included in Flattening at t6
Updates included in Flattening at t8 with Window size 3
Flattened graphs generated at different horizons
V1
V2
V3
V4
V5
V6
Live Graph (Latest Flattening)
Flattening at t6
Flattening at t8 Window size 3

Figure 5.2: A set of graph updates between t_1 and t_{11} with a selection of graph flattenings, demonstrating the updates they would include as well as the state of the materialised graph.

scope to that of the intervals between snapshots and means the order of changes between these epochs are lost. In contrast, Raphtory's underlying temporal graph model allows the user to flatten the graph history at any chosen *flattening_end* to view exactly what it would look like at that time. The graph generated from a flattening at *flattening_end* t, will then include all updates within the stream up to and including t; as explored in Section 3.4. To illustrate this, Figure 5.2 provides a timeline of updates between t_1 and t_{11}. Based on this stream, two graphs have been materialised from flattenings at the latest time (t_{11}) and at t_6. The updates these include, as well as the graphs themselves, may be seen within the figure. Any algorithms discussed in Section 5.2 may be applied to these flattenings without modification as they are effectively seen as a static graph.

In addition to flattenings, which include all updates since the start of the stream, Raphtory may materialise graphs where a window has been applied from the chosen *flattening_end*, specifying a lower bound of time after which all vertices and edges must either have been added or updated, otherwise they are no longer considered part of the graph[146]. 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 appear high in the listing. By pruning inactive entities/connections, smaller users quickly rising in popularity, or with very active communities, will become easier to distinguish from the background noise. Furthermore, by varying the window size from minutes to months, a user will be able to experiment with both short and long term

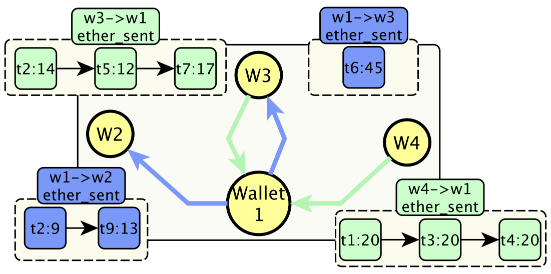


Figure 5.3: Example Ethereum transactions from the local vertex perspective of wallet 1. Each edge is labelled with the ether_sent property denoting the individual amounts of currency sent to or from the other wallets. Incoming edges are coloured green whilst outgoing edges are in blue.

patterns within their data[147]. To illustrate such a flattening, Figure 5.2 contains a third graph materialised at $t8$ with a window size of three time increments. As with the non-windowed flattenings, the algorithms discussed above may be applied without modification.

Including History: Temporal Algorithms

In contrast to the time-evolving approach, algorithms may be provided with a wider temporal scope by incorporating the graph history directly into the algorithm. The chosen algorithm can make use of this expanded set of information in a number of ways, which can relate back to the categories within Figure 5.1. Beginning again from the left, within the local scope of an entity, the user may now access all values associated with a property key. These may, therefore, be used to derive aggregates or trends which could be useful in themselves or be fed into further analysis. For instance, Figure 5.3 shows an example transaction network from the view of a singular vertex, based on the Ethereum blockchain explored in Section 6.4.2. Within this, vertices represent wallets and an edge between two wallets contains all the transactions the source has sent to the destination. With this temporal view, the user may decide to calculate the total amount sent one way between the wallets by summarising all values for the 'ether_sent' property. The values across all edges may then be combined within the vertex to generate a balance vector plotting the currency available to the wallet throughout time. Alternatively this same information could be used to generate an average and standard deviation to raise an alert on unusually high transactions. In conjunction with property queries, a vertex may compare its current structural state to how itself and its edges looked in prior windows. These may also then be aggregated to garner metrics, such as total/average consecutive windows present, allowing entities (and their relations) to be categorised as either permanent or transient members of the graph for a given window size.

Expanding the scope to the vertex neighbourhood, the time of connections between entities can now be included in pattern matching or temporal motifs[62]. Within this a user may explore the pace and duration of interactions between entities within the data, which can provide an additional dimension of insight over a snapshot or flattening. As an example of this, in Figure

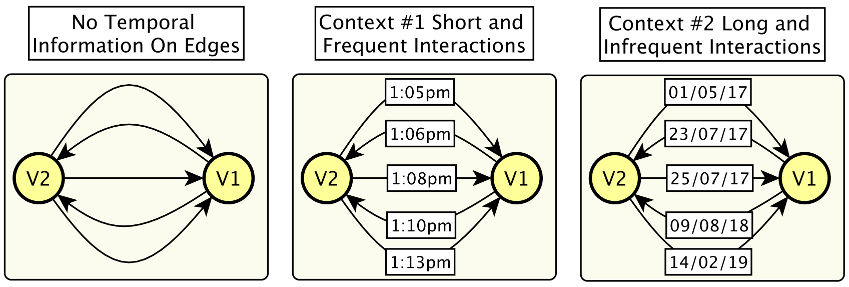

No Temporal Information On Edges
Context #1 Short and Frequent Interactions
Context #2 Long and Infrequent Interactions
V2
V1
1:05pm
1:06pm
1:08pm
1:10pm
1:13pm
01/05/17
23/07/17
25/07/17
09/08/18
14/02/19

Figure 5.4: Change in perceived context of an interaction between vertices based upon added temporal information.

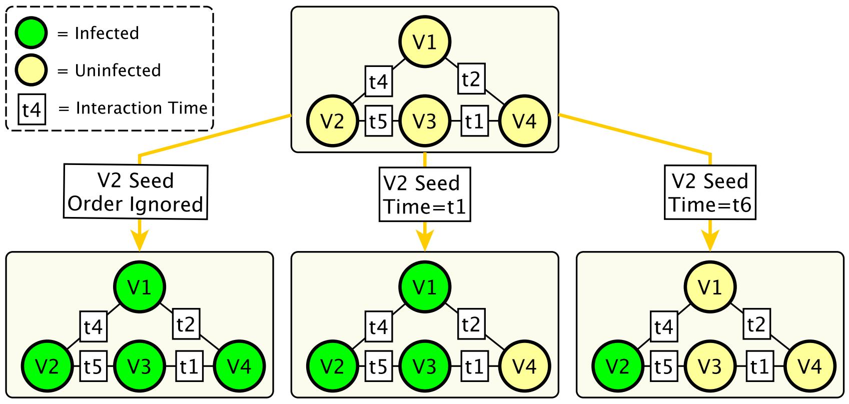


Figure 5.5: The results of an example contagion algorithm over four vertices when seeding the infection on vertex 2 at different times and when respecting the order of interactions.

5.4 a static graph can be seen with five edges between two vertices. When viewing this without temporal information, it may be difficult to draw conclusions on the type of interaction this was. However, by adding different timestamps to these the frame of reference drastically changes. In the first instance the timestamps are very close together, meaning this was a short and fast interaction, possibly a comment exchange on a post between two strangers (given a social network context). Alternatively, in the second case the times are sparse and over a long duration, which could suggest infrequent messages between old acquaintances. This context can be very important for a wide array of applications. For instance, returning to the blockchain example, 100 transactions between two wallets over a year may draw no suspicion, but the same number over the course of a minute could be someone splitting a much larger payment and trying to avoid an alert, as discussed above.

In conjunction with pace and duration, the temporal scope also provides the order in which interactions occur. This again adds to the context of a pattern/motif, but can be used to facilitate

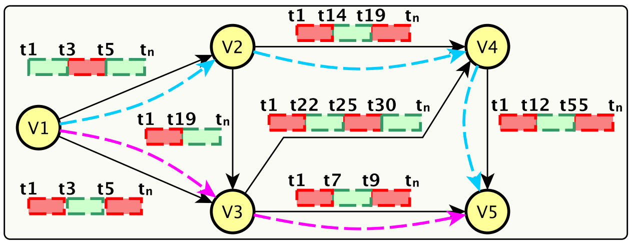


Figure 5.6: Example temporal shortest path. Edges are labelled with their history denoting periods when they existed within the graph (in green) and periods where they were absent from the graph (in red). The pink arrows denote the shortest path from vertex 1 to 5 when starting at time t_1; the blue path is shortest at time t_{12}.

more accurate traversals, especially in the context of diffusion or contagion algorithms[148]. Within this a basic diffusion will start at one (or more) 'seed' vertices and propagate through connections until the entire graph is 'infected' or it burns out (no more connections between infected and non-infected vertices). This may form the basis for modelling the spread of disease within a population, but when applied to a non-temporal graph the infection may spread in a way not possible given the actual order of interactions. For example, in Figure 5.5 we can see the times in which four vertices interact with each other. When seeding vertex 2 and ignoring this order, all vertices become infected. However, by providing a time of infection and only traversing connections newer than the one the infection previously travelled, a more realistic spread may be extracted. We can also see that it is not actually possible for the infection to reach vertex 4 from vertex 2, as it has to wait until at least t_4 to jump to a new host, at which point all interactions with vertex 4 have passed.

As such, by combining pace, duration and order we can explore and extract patterns which are inherently closer to what is happening in the data and, therefore, the *real world*. This is a powerful functionality which has numerous benefits. For instance, once again returning to the Ethereum use case, money laundering and illicit transaction patterns often contain large cycles[64] which must obviously have a start and an end, as well as follow an internal order where each transaction occurs before the next in the cycle. These also contain subtle temporal motifs such as each transaction in the cycle occurring 24 hours apart and siphoning a percentage off the value sent[149]. A temporal scope is, therefore, paramount in modelling the flow of illicit currency. However, this also has the benefit of reducing the number of edges which need to be traversed, as fewer fit the pattern, leading to a lower number of false positives and making the algorithm less resource intensive. This is explored further within the Ethereum taint propagation[150] in Section 6.4.2.

Finally, the temporal scope can also augment graph algorithms with a global scope, integrating the structural and property changes much in the same way seen above. A clear example of this is when performing a single source shortest path, respecting edge history and availability

[49][151]. Looking at Figure 5.6, all edges are labelled with the times they were added and deleted, i.e. they exist for set time ranges. When attempting to find the shortest path between vertex 1 and 5 the answer now depends on the start time of the journey. For instance, starting at time t_1 the shortest route would be to wait until time t_3 to hop to vertex 3 and then wait until t_7 to hop to vertex 5, arriving at t_7 assuming instant travel. Alternatively, if starting at t_{12}, this path would no longer be available, requiring the blue path to be taken and arriving at time t_{14}. Note at $t_{>55}$, vertex 5 would no longer be reachable as all connecting edges would have been removed; this could be explored separately as a temporal reachability problem[152]. Time-aware shortest path algorithms such as this can be used as the basis for important route planning. For example, modelling flights between airports in this manner allows the band of time a flight is available for boarding to be embedded in the edge. Holidaymakers planning multi-stop flights may, therefore, explore the best routes and departure times based on reducing the overall journey and time spent waiting in airports. Expanding this to include edge properties, if modelling a road network, a congestion weighting may be integrated for each road (edge) at different times of the day, alongside possible closures due to road maintenance etc. A navigation system may then factor all this information into its suggested route when a user requests the quickest way to a location.

5.2.3 Time Ranges, Window Batches and Live Analysis

Whilst the time-evolving model and expanded temporal scope of algorithms are initially discussed separately. These may actually be utilised in conjunction with each other within Raphtory as they exist at different levels of the analysis pipeline. Temporal algorithms require the user to incorporate the history of the entity inside their desired functions and this is, therefore, exposed inside the analysis API, alongside the building blocks for traditional graph analysis (as discussed in Section 5.3.2). Once the user has finished designing the algorithm (temporal or otherwise) this may then be set to run on any graph flattening. This will limit the temporal scope to the updates included in that graph, i.e. the algorithm will not have access to changes further back than a set window or further forward than the chosen *flattening_end*.

This is taken one step further by automating the analysis of many graph flattenings throughout a period of time. Within Raphtory this is referred to as *Time Range* analysis and allows the user to specify a period of interest, within the history of the graph, and the time increment with which they would like to move between these points. At each time step (inclusive of the start and end) a graph flattening is generated and the algorithm applied, publishing the results for each as they complete. Similarly to the stand alone graph flattenings seen above, Figure 5.7 shows the result of a time range between t_2 and t_6 of the same update stream from Figure 5.2, with an increment of two time points. This materialises graphs at time t_2, t_4 and t_6, which can be seen within the figure along with the updates these would include. Note, the flattening at t_6 from Figure 5.2 and the one inside of the range are identical.

For a non-temporal graph algorithm, this allows the user to extract some insight unavailable

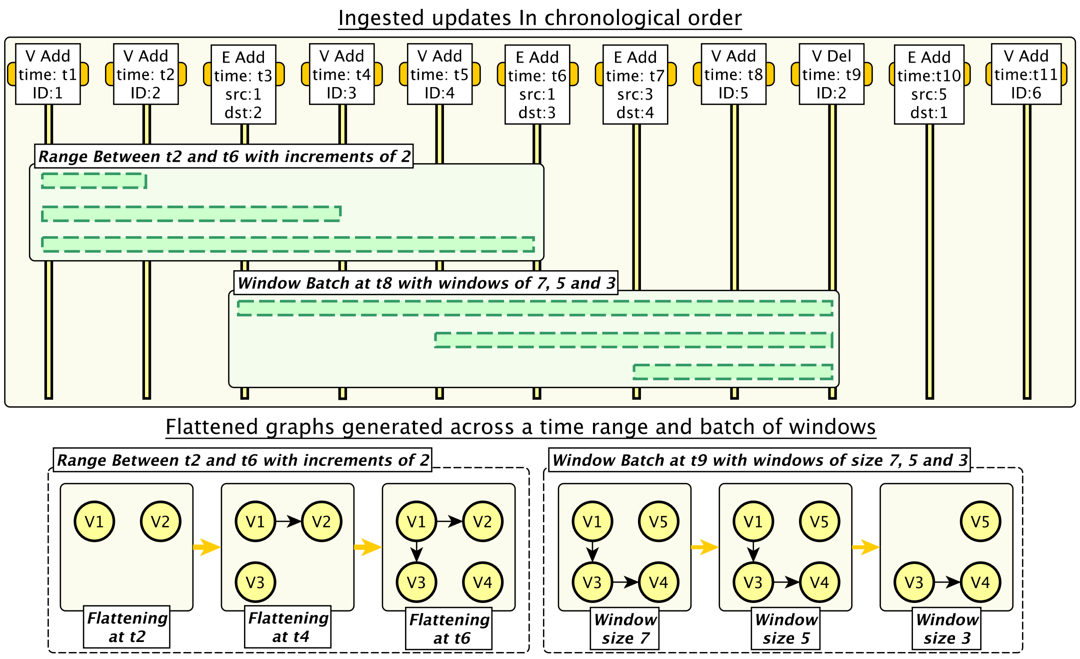

Ingested updates In chronological order
V Add time: t1 ID:1
V Add time: t2 ID:2
E Add time: t3 src:1 dst:2
V Add time: t4 ID:3
V Add time: t5 ID:4
E Add time: t6 src:1 dst:3
E Add time: t7 src:3 dst:4
V Add time: t8 ID:5
V Del time: t9 ID:2
E Add time:t10 src:5 dst:1
V Add time:t11 ID:6
Range Between t2 and t6 with increments of 2
Window Batch at t8 with windows of 7, 5 and 3
Flattened graphs generated across a time range and batch of windows
Range Between t2 and t6 with increments of 2
Window Batch at t9 with windows of size 7, 5 and 3
Flattening at t2
Flattening at t4
Flattening at t6
Window size 7
Window size 5
Window size 3

Figure 5.7: The same set of graph updates from Figure 5.2, demonstrating the updates included in materialised graph flattenings for a time range and batch of windows.

within the most recent version, such as how a set of known communities evolved to their current state within a social network[153]. However, this can also be used to explore how the results for a temporal algorithm evolve through the history of a graph in exactly the same manner. For example, in the Ethereum taint analysis explored in Section 6.4.2, each wallet may be allocated a score to show how much illicit currency has flown through them and, therefore, to what degree the wallet owner is a 'bad actor' within the network. Whilst this algorithm does utilise the history to correctly propagate the taint, scores allocated would be for the state of the graph at that point in time. It is, therefore, interesting to see how the score for wallets of interest change as new blocks are ingested or to highlight wallets which have a drastic shift in taint over a short period of time.

In a similar vain to time ranges, it is possible to materialise a set of graph flattenings at the same *flattening_end*, but with different window sizes. Within *Raphtory* this is referred to as a *Temporal Window Batch*, allowing the user to set all window sizes they are interested in. The analysis will then be performed on each windowed graph, starting with the biggest window. To illustrate this, Figure 5.7 contains a set of graph flattenings at time t_9 with a window batch of 7, 5 and 3 time increments. The figure also notes the updates these would include. An interesting point to note here is that the graphs generated for windows 7 and 5 are identical, as the only difference is the edge $1 \longrightarrow 2$ which is removed with vertex 2 upon deletion at t_9.

Both of these techniques on their own can be quite powerful in extracting new insight from the underlying data, but these may also be used in combination to truly get an understanding of how the graph has evolved. This means that the user may set a range and window batch of interest, and at every time-step within the range the batch of windowed flattenings will be created and analysed. Depending on the combination of time increment and window sizes these can be viewed as fixed, sliding[76] or disconnected windows over the stream of data. A fixed window in this instance would be when the increment and window size are the same, leading to non-overlapping, but connected views. A sliding window would be created when the window size is much larger, such as hopping forward a day at a time, but looking back a week. A disconnected window would be created if the increment was bigger, such as looking at the data ingested on the first day of every month. These clearly will have drastically different outputs and use cases even though the underlying algorithm is exactly the same. As a concrete example of this combination, in Section 6.4.1 a set of simple algorithms were applied across two and a half years of user interactions within a social network. For each algorithm the analysis moved through the history at increments of an hour, building flattenings windowed to an hour, day, week, month and year, as well as a no window aggregate graph. By comparing these windows across the lifetime of the network many interesting patterns were discovered, notably the giant connected component[154] seen in all window sizes remained present other than for the hour window, where this would break down into small communities in the early hours of the morning, rejoining together as the main userbase came online.

Live Graph Analysis

As discussed throughout this work Raphtory can continuously ingest new updates to the graph with the intention of making these available for analysis as soon as possible. To integrate this streaming/dynamic context and enable algorithm execution on the most recent state, i.e. the *Live Graph*, a special type of time range with no end is provided, denoted as *Live Graph Analysis*. There are two possible interpretations of the Live Graph which must be clarified. When an update arrives at a Partition Manager this data is inserted into the Entity Storage and is, at that point, processable. Unfortunately this update or any prior to it may require synchronisation between partitions, meaning any analysis performed on this data runs the risk of returning incorrect results. This could be characterised as some sort of approximate graph analysis[155], but there would be no way to bound the possible error from update interleavings, which could be drastic for some algorithms. For example, when calculating the shortest path between two nodes, if an edge on this path was desynchronised it could be missed during the analysis, leading to at best a longer path being returned or at worst an 'unreachable' verdict. As such, for the purposes of analysis within Raphtory the Live Graph refers to the flattening at the most recent watermarked timestamp (as described in Section 4.7). This guarantees all data is synchronised between partitions and gives a consistent state across supersteps, ensuring a correct result.

By establishing the Live Graph as a flattened graph at the latest safe point, this additionally

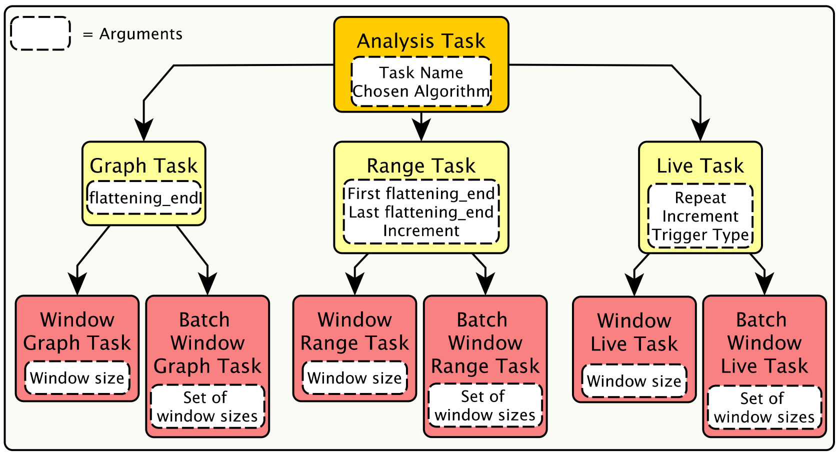


Figure 5.8: Types of Analysis Task and their subtypes for windowing. Each task has a set of arguments which the user must provide, with those in the supertype required in all subtypes.

means that all the different layers of analysis and established algorithms discussed above may be applied on each new incarnation with minimal user input. The only decisions the user must make are if the submitted analysis is to run once or repeatedly and, in the case of the latter, how often. Once decided upon the frequency of the repetition, the user may consider if they want this increment to be triggered on processing time or event time. Processing time in this sense would mean that after the chosen period had passed the next analysis cycle would fire, irrelevant of whether the latest watermark had also advanced this far. In the case of event time, the analysis would only be run once the watermark had advanced past the next designated time requested. These are similar, but can have different applications. For example, a processing time trigger may be appropriate when running Raphtory over a bursty unbounded dataset to provide an ongoing log of metrics, allowing the user to monitor the current graph state in near real-time. An event time trigger, on the other-hand, may be more appropriate when looking at more complex trends over a longer period. For instance, the user may be interested in maintaining a sliding window over the latest state[156] looking back at the last 24 hours of updates every ten minutes.

5.2.4 Wrap-Up: Available Analysis Within Raphtory

In summary, the analysis available within Raphtory explores the full structural scope of graph algorithms. Whilst analytics is the primary focus within this, as Raphtory's model is based upon rich property graphs[24], many of the traversals and queries may also be executed, albeit in a vertex-centric manner as discussed below. These standard graph algorithms are then expanded with a temporal scope by providing the user access to the full structural and property history of each vertex and its edges, with respect to the graph flattening upon which the analysis is performed.

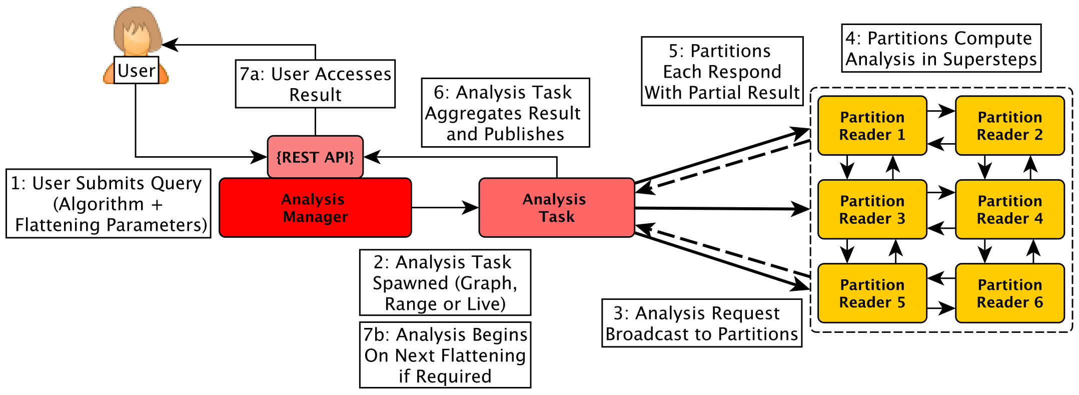

User
7a: User Accesses Result
6: Analysis Task Aggregates Result and Publishes
5: Partitions Each Respond With Partial Result
4: Partitions Compute Analysis in Supersteps
{REST API}
1: User Submits Query (Algorithm + Flattening Parameters)
Analysis Manager
Analysis Task
Partition Reader 1
Partition Reader 2
Partition Reader 3
Partition Reader 4
Partition Reader 5
Partition Reader 6
2: Analysis Task Spawned (Graph, Range or Live)
7b: Analysis Begins On Next Flattening if Required
3: Analysis Request Broadcast to Partitions

Figure 5.9: The workflow of the Raphtory Analysis Model, following a query submitted by the user and the subsequent steps carried out by the spawned Analysis Task and Partition Readers.
.

Once an algorithm has been devised (temporal or otherwise) it may be submitted to run in Raphtory at any point in the history of the graph and with any window applied, without requiring modification. The pairing of algorithm and flattening parameters is defined as an Analysis Task and, as can be seen in Figure 5.8, these fall into three major categories. For analysis at a single point of time within the history of the graph, Raphtory provides the *Graph Task*. When the user is interested in applying a time-evolving version of the algorithm they may alternatively use a *Range Task*, establishing a focal period within the graph history and how often throughout this a flattening should be materialised. Finally, the *Live Task* simplifies the inclusion of Raphtory's streaming context, executing the algorithm on the most up-to-date graph, either once or recurringly, with the recurrent trigger based on processing time or event time. For each of these task types the user may additionally apply a window, or batch of windows, and for each *flattening_end*/window pair a flattening will be materialised, analysed, and the result returned. The manner is which a Raphtory algorithm may be created, and the internal components for flattening materialisation/algorithm execution, are discussed in the remaining sections of this chapter.

5.3 The Raphtory Graph Analysis Model

5.3.1 Analysis Model Overview

At its core, Raphtory's analysis architecture is based on a Bulk Synchronous Parallel (BSP)[86] model whereby algorithms are executed in supersteps. Raphtory is vertex centric, similar to Pregel[7], meaning algorithms are designed from the perspective of the vertex which only has access to its state and that of its edges. Each vertex may then independently run a user defined function for the superstep within which it may calculate and store state about itself, send mes-

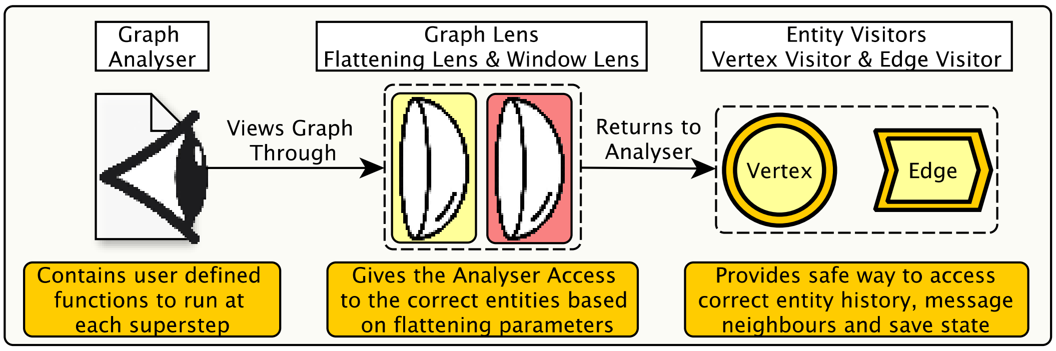

Graph Analyser
Graph Lens
Flattening Lens & Window Lens
Entity Visitors
Vertex Visitor & Edge Visitor
Views Graph Through
Returns to Analyser
Vertex
Edge
Contains user defined functions to run at each superstep
Gives the Analyser Access to the correct entities based on flattening parameters
Provides safe way to access correct entity history, message neighbours and save state

Figure 5.10: The three main components of Raphtory's analysis API: The Analyser, Graph Lens and Entity Visitor.

.

sages to its neighbours, process messages received from the prior superstep and vote to halt the algorithm if it believes it has converged. This model was chosen as Raphtory is a distributed platform and, by viewing each vertex as an independent worker, it does not matter how many partitions there are or where each vertex is stored as the algorithm may execute in the same way. This additionally simplifies many aspects of parallelism within a partition as minimal locks are required due to the lack of shared state between vertices. Finally, this is beneficial for the user as it abstracts away the complexity of implementing distributed algorithms, similar to other distributed frameworks such as MapReduce[69].

As can be seen in Figure 5.9, and discussed above in Section 5.2.4, when a user submits a query an Analysis Task is spawned. This Analysis Task controls the BSP workflow of the algorithm, i.e. deciding when supersteps should run, if the algorithm has converged across all partitions and aggregating/publishing the final results if it has. Requests to execute a superstep are sent to all Partition Readers who must then materialise their part of the graph flattening, execute the user functions on each vertex under their control and synchronise any vertex messages amongst themselves for the next superstep. Once they have finished they will report back to the Analysis Task and return to idle awaiting the next request. This continues until the algorithm has converged and the results are published for that flattening. If it was a range or live task which was spawned, with multiple *flattening_ends* to analyse, the task will begin the process again on the next flattening. Batch windows are handled in parallel within the Partition Reader, all of which is discussed in more detail in Section 5.4.

Finally, to facilitate analysis and allow the users to implement their algorithms, Raphtory provides an API to encompass all the features described in Section 5.2. For this there are three main components which the user may interact with: Analysers, Graph Lenses and Entity Visitors. The overview of these and their interactions can be seen in Figure 5.10. Beginning at the left of the figure, the functions to be executed by the partitions/vertices at each superstep are

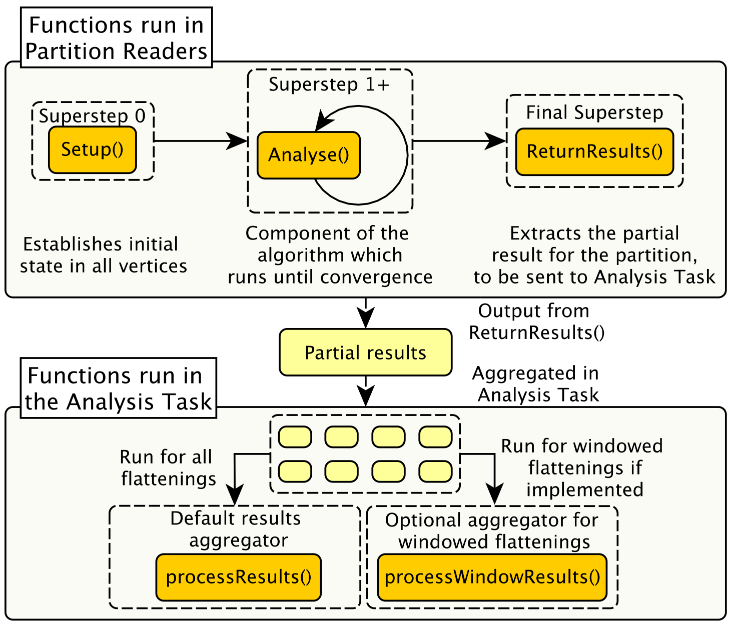

Functions run in Partition Readers
Superstep 0
Setup()
Establishes initial state in all vertices
Superstep 1+
Analyse()
Component of the algorithm which runs until convergence
Final Superstep
ReturnResults()
Extracts the partial result for the partition, to be sent to Analysis Task
Output from ReturnResults()
Partial results
Aggregated in Analysis Task
Functions run in the Analysis Task
Run for all flattenings
Run for windowed flattenings if implemented
Default results aggregator
processResults()
Optional aggregator for windowed flattenings
processWindowResults()

Figure 5.11: An overview of the functions a user must define within an analyser which constitute the general flow of an algorithm.

encapsulated within an *Analyser* class. This also specifies the partial results each partition should return and how these responses may be aggregated into the final result. Within an analyser the user will interact with the temporal graph through a graph lens. This returns the entities present within a flattening, allowing the analyser to run agnostic of partition and flattening parameters. To safeguard the graph state and ensure there are no clashes with ongoing ingestion these entities are first wrapped in a visitor[157] object, before being returned. This ensures the user can only access the data of the entity in a predefined manner, that the information returned respects the *flattening_end* and window applied and provides an interface for saving temporary analytical state/messaging neighbours.

5.3.2 Analyser

The analyser is the class which encapsulates the whole algorithmic process, what vertices should do in each superstep and how the results for each partition can be combined and returned to the user. An overview of the main functions to be implemented can be seen in Figure 5.11 and divides into two categories, those which will run within each Reader Worker and those which will run at the end of the algorithm within the *Analysis Task*. Looking first at those within the Reader Worker, there are three main functions which broadly cover the flow of an algorithm: ***Setup()***, ***Analyse()*** and ***ReturnResults()***. Within all of these the user has access to the lens

API as well as any arguments submitted alongside an analyser.

Any iterative algorithm will begin with ***Setup()***, which can be seen as superstep zero. Here the user may establish default state for all vertices and send out initial messages. For example, in the connected components algorithm from Appendix C.1, each vertex is allocated an initial component label (its own ID) which it then forwards to all neighbours for them to compare in the first superstep. This first superstep, and all those following, are then defined within ***Analyse()***. Here the user specifies what each vertex must do within an iteration of the algorithm to converge on the final answer. Continuing with the connected components example, all vertices will check through their received messages to see if these contain a component ID lower than their current ID. If one is found, they must replace their stored label with this new value and send it to all neighbours, propagating it through the graph. If no new ID is found no messages are sent, and if this happens for all vertices the algorithm is considered to have converged, i.e. all vertices are labelled correctly. Once an algorithm has converged ***ReturnResults()*** will be executed which extracts the final result from the partition and returns it to the Analysis Task. Finishing up with the connected components example, the labels for each vertex in the partition are extracted and grouped together, returning a count of vertices for each component ID seen to Analysis Task.

Algorithms may converge in two different ways. The first is that the maximum number of supersteps is reached, set by the user via ***MaxSupersteps()***. Note, if this is set to one or less (i.e. non-iterative algorithms) only ***ReturnResults()*** is executed, such as in the degree ranking analyser from Appendix C.2. The second manner for convergence is via ***voteToHalt()***, where all vertices request to stop the algorithm early as they believe they have converged on their final state. This can be seen within the connected component analyser where, if a vertex does not find a new component label within its messages it will raise a vote, which will be automatically removed if new messages reach it. If all partitions report that their vertices have voted to halt the algorithm, it may skip straight to ***ReturnResults()*** instead of completing the remaining unnecessary supersteps.

Once an algorithm has converged, and all the partial results have been returned by the Partition Readers, the Analysis Task may execute ***ProcessResults()***. Here the user is given an array of partial results from each worker and can decide how to aggregate these together. Once the aggregation is complete, the result may be made available via ***PublishResult()***, whereby it will be accessible from the REST API as discussed in Section 5.5. In the connected components example, the partial counts are combined to report on the biggest component and the total number of components, as well as several characteristics such as the total number of 'islands' (components with only one vertex). This is then published in JSON format to be plotted, as can be seen in Section 6.4.1. Whilst ***ProcessResults()*** is the default function for all flattenings, there may be instances where windowed data has to be handled differently. As such the user may optionally implement ***processWindowResults()***, which again can be seen in the connected components example reporting the window size alongside the *flattening_end* to allow different windows to be distinguished.

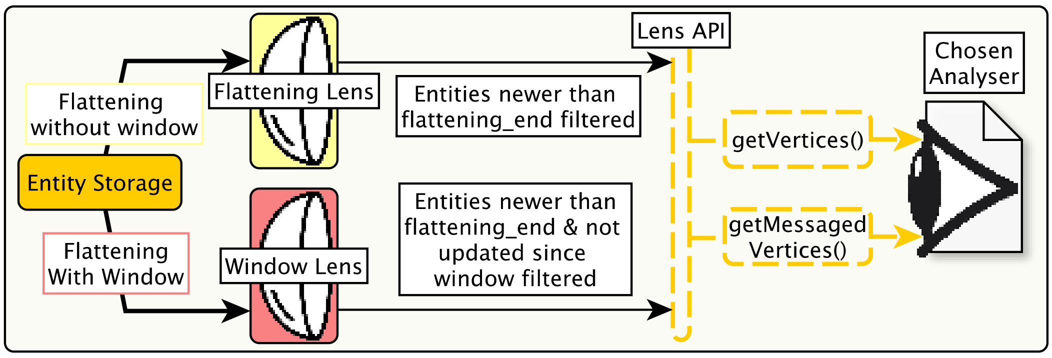

Flattening without window
Entity Storage
Flattening With Window
Flattening Lens
Window Lens
Entities newer than flattening_end filtered
Entities newer than flattening_end & not updated since window filtered
Lens API
getVertices()
getMessaged Vertices()
Chosen Analyser

Figure 5.12: An overview of the Graph Lens API demonstrating the flattening building process executed when the analyser requests the set of vertices for the partition. This abstraction allows an analyser to be applied to any flattening without modification.

5.3.3 Graph Lenses

As discussed in Section 5.2, all analysis in Raphtory is executed on a flattening of the graph. Graph lenses facilitate the ability to materialise graph flattenings at any user defined *flattening_end* and apply a given analyser to this. Figure 5.12 provides an overview of this process, whereby a user may request the set of vertices within each of the superstep functions via ***getVertices()***. Looking further into the flattening parameters, there are two possible lenses which may be applied, one for windowed flattenings and one for those with only a *flattening_end*. In either case the vertices within each Entity Storage will be run through the lens to decide if they are kept in the graph or not. Vertices which survive will be wrapped in a vertex visitor object and have their incoming and outgoing edges checked with the same lens. Those which belong in the flattening will be wrapped as edge visitors and stored within the vertex visitor, whilst those not included will be filtered out. Note, this filtration process has no bearing on the state of the actual Entity Storage/objects. Once the filtration is complete the set of vertex visitors will be returned to the user allowing them to perform the required analysis. The superstep functions may, therefore, be defined once and applied to any flattening, with the lens returning the correct entity set.

As messaging is prevalent in almost all vertex centric algorithms, in addition to this selection of all vertices within the flattening, the lens API additionally provides the function ***getMessagedVertices()*** which only returns those vertices which have received messages for the current job/superstep. This syntactic sugar means users do not have to worry about checking if the mailbox of each vertex is empty or handling the remaining unmessaged vertices, as these are considered to have implicitly voted to halt. This is used within the connected components example as only those vertices with new labels to check are required to run.

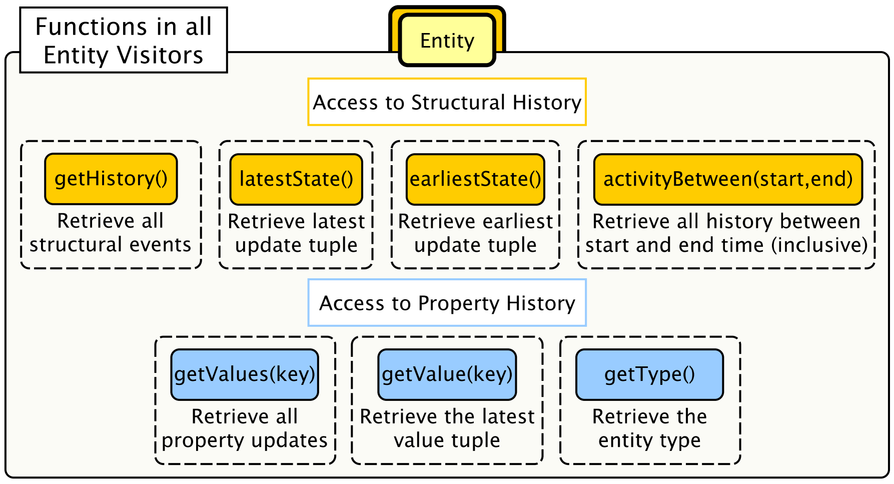

Functions in all Entity Visitors
Entity
Access to Structural History
getHistory()
Retrieve all structural events
latestState()
Retrieve latest update tuple
earliestState()
Retrieve earliest update tuple
activityBetween(start,end)
Retrieve all history between start and end time (inclusive)
Access to Property History
getValues(key)
Retrieve all property updates
getValue(key)
Retrieve the latest value tuple
getType()
Retrieve the entity type

Figure 5.13: Functions available to both vertex and edge visitors, providing access to the structural and property history of the entity with respect to the current flattening parameters.

5.3.4 Entity Visitors

As a clear goal of Raphtory is to enable parallel updates to the graph alongside analysis, there must be stringent control over the entity objects avoiding deadlocks or invalid state. To manage this, as well as provide an API for users to interact with entities in a meaningful manner, both edges and vertices are made available via a visitor[157] wrapper class. These visitors contain the original objects and are structured in a similar fashion, with the vertex visitor being the focal component of the API and containing a map of incoming and outgoing edge visitors. Both of these extend a supertype entity visitor class which provides access to the history/properties of the entity, as well as each having specific functions which can be broadly categorised into: Edge Access; Storage and Retrieval of analytical state; and Vertex Messaging.

Access to Entity State

Beginning with what is shared between both classes, the full structural and property history of an entity is made available via getter functions, as can be seen in Figure 5.13. For the structure, clearly the entity is present at the current time as it exists within the graph, but the user may additionally wish to know how many updates the entity has received within the flattening period, as well as when these have happened. For this the entity visitors contain the ***getHistory()*** function, which will return a safe copy of the structural history of the entity filtered within the bounds of the flattening, i.e. nothing past the *flattening_end* or before the window (if one has been set). This may then be utilised by the user in the exploration of temporal patterns. In addition to the basic history access, the visitors provide many helper functions such as ***latestState()***, ***earliestState()*** and ***activityBetween(****start,end****)*** to simplify history

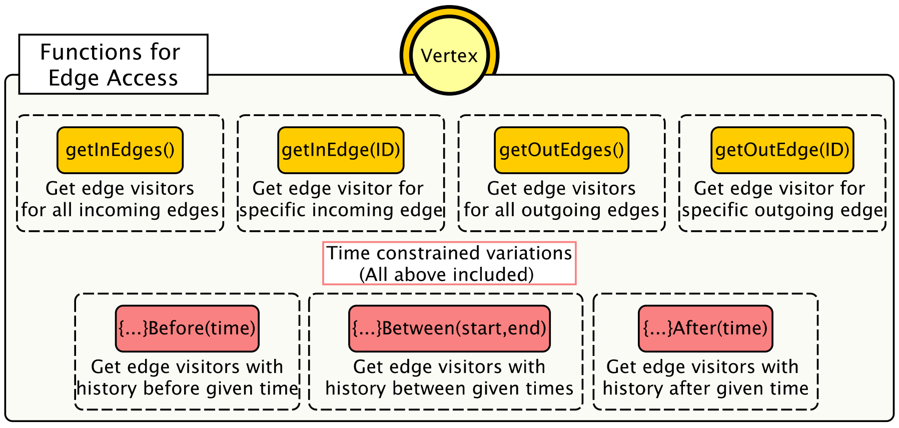

Functions for Edge Access
Vertex
getInEdges()
Get edge visitors for all incoming edges
getInEdge(ID)
Get edge visitor for specific incoming edge
getOutEdges()
Get edge visitors for all outgoing edges
getOutEdge(ID)
Get edge visitor for specific outgoing edge
Time constrained variations (All above included)
{...}Before(time)
Get edge visitors with history before given time
{...}Between(start,end)
Get edge visitors with history between given times
{...}After(time)
Get edge visitors with history after given time

Figure 5.14: The functions with which the user may access the edges associated with a given vertex. In yellow are basic functions to access all edges within the flattening. In red are time constraints which may be applied to all those in yellow, i.e. getOutEdgesAfter() to retrieve the subset of outgoing edges which have an historic event after the provided time.

exploration. As an example of this, the temporal triangle count analyser seen in Appendix C.3 looks to find triangles where the three edges ($A \longrightarrow B, B \longrightarrow C, C \longrightarrow A$) occur in chronological order. To extract these triangles correctly, the first two of these functions are used to establish a range of time within which a triangle may occur on a given vertex and the third function is used to interact only with edges which respect these times.

Entity properties are handled in much the same way. The user may access the 'current' value of a property from the perspective of the flattening via the ***getValue()*** function. This takes a property key and returns an 'Option'[158] object which will contain the value if the property existed at the *flattening_end*. Alternatively, if the user is interested in all associated values, they may utilise ***getValues()*** which will return a bounded copy of the property history as with the structural history above. Finally, the user may request the type of the entity via ***getType()***, the vertex ID of a vertex visitor via ***getID()*** and the source/destination ID of an edge visitor via ***getSrc()***/ ***getDst()***.

Edge Access

As the graph lens API returns a set of vertex visitors and these contain all related edges within the flattening, a set of functions are provided to extract those the user requires, as summarised in Figure 5.14. The most basic of these is ***getInEdges()*** and ***getOutEdges()*** which return the set of edge visitors within the corresponding edge map. These may be utilised to extract edge information or decide which neighbours to propagate messages to. If the user is looking for a specific edge they may use ***getInEdge()*** and ***getOutEdge()***, taking the ID of the vertex on the other end and returning an option which will contain the visitor if the edge is present.

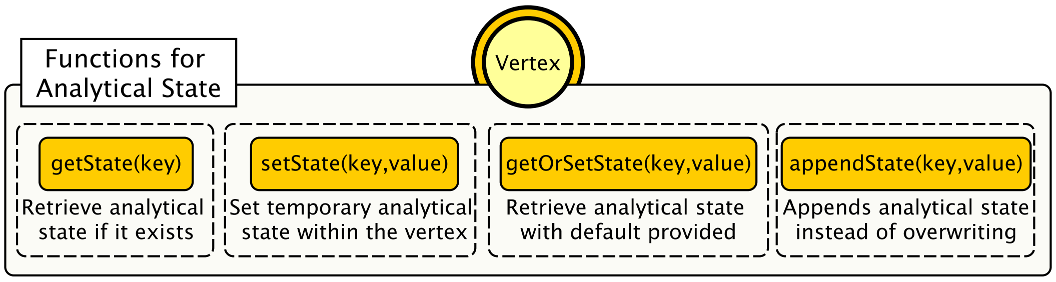

Functions for Analytical State
Vertex
getState(key)
Retrieve analytical state if it exists
setState(key,value)
Set temporary analytical state within the vertex
getOrSetState(key,value)
Retrieve analytical state with default provided
appendState(key,value)
Appends analytical state instead of overwriting

Figure 5.15: Functions for storing and retrieving temporary analytical state within each vertex for an ongoing iterative task.

As an extension to these, all edge getter functions are also available with the time constraints 'After', 'Before' and 'Between' (i.e. ***getOutEdgesAfter()***). These take time arguments and return a subset of the edges associated with the vertex if their history within the bounds of the flattening contains an update which passes this constraint. This can be utilised in temporal contagion algorithms/random walks to correctly propagate forwards or backwards through time. As an example of this, within the temporal taint algorithm in Section 6.4.2, each infected node spreading the taint to its neighbours additionally informs them of the time at which they have become infected. In the next superstep, when these newly infected nodes go to further propagate, only outgoing edges with updates after this time are selected, never tainting vertices who have not interacted with them since the infection took place.

Analytical State and Storage

Iterative vertex centric algorithms often require the storage of temporary state which can be referred to/updated within each superstep as the algorithm works towards convergence. For example, in the connected components implementation in Appendix C.1, vertices store the lowest component ID that has been messaged to them, which may then be compared to any new messages at each superstep and updated/propagated if a new minimum is found. As can be seen in Figure 5.15, the API for this is fairly intuitive with the vertex visitor offering standard getter and setter practices: ***setState()*** takes a variable name and a value (of any type) and stores this within the vertex; ***getState()*** also takes a variable name and will return an option, as with the property values above, in case the analysis variable is unset; these can then be combined via ***getOrSetState()*** where the user may additionally give a default value instead of returning an option; and finally ***appendState()*** can be used to record all partial results, such as in the case of the temporal triangle count in Appendix C.3, where all triangles discovered for the vertex must be recorded, not just the latest one.

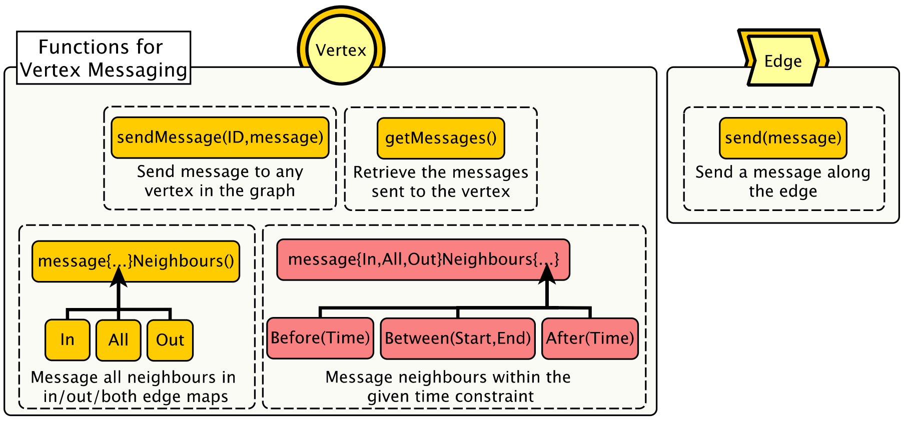

Functions for Vertex Messaging
Vertex
Edge
sendMessage(ID,message)
Send message to any vertex in the graph
getMessages()
Retrieve the messages sent to the vertex
send(message)
Send a message along the edge
message{...}Neighbours()
In
All
Out
Message all neighbours in in/out/both edge maps
message{In,All,Out}Neighbours{...}
Before(Time)
Between(Start,End)
After(Time)
Message neighbours within the given time constraint

Figure 5.16: Functions for sending and retrieving messages between vertices. Messages may be sent direct to one vertex or broadcast to all neighbours. In red are temporally constrained variations of this broadcast, similar to the edge selection in Figure 5.14.

Vertex Messaging

The final category of functions within the visitor API is how vertices communicate with each other and can be seen summarised in Figure 5.16. Of these, the default messaging function is ***sendMessage()***, which takes two arguments, the ID of the vertex to be messaged and the value to be sent (which may be any serialisable type). Note, this ID may be of any vertex in the graph, not just neighbours. There are, however, many instances where the user may want to send the same message to all neighbours of the vertex, for which they may utilise ***messageInNeighbours()***, ***messageOutNeighbours()*** and ***messageAllNeighbours()***. This is useful in global analytics such as in the PageRank implementation in Appendix C.4 where the rank of the vertex must be sent along all outgoing edges. As with the edge access functions above, the vertex visitor additionally provides 'After', 'Before' and 'Between' variations of these functions, but only to message vertices within the set constraint who have been interacted with. For instance, when propagating an infection in the temporal contagion analyser from Appendix C.5, the ***messageOutNeighboursAfter()*** function is used to propagate only the contagion to neighbours who have been interacted with after the initial time of infection for a vertex.

For a more localised message sending, the user may alternatively access a set of edge visitors, which all provide the ***send()*** function and will deliver the given message to the vertex on the other end of the edge. This is used in the temporal triangle count analyser to contact neighbours who may be part of a triangle with the sender. Finally, once all messages have been synchronised from the prior superstep, vertices may call ***getMessages()*** to access the full set of messages they have received.

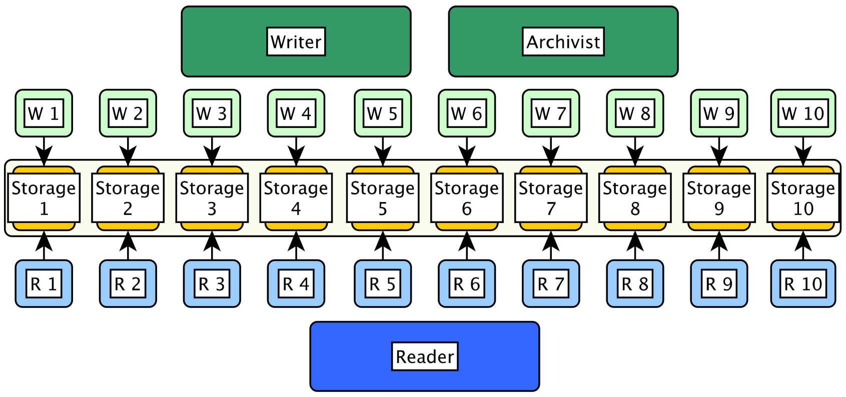


Figure 5.17: The internal storage of a Partition Manager with its data split between ten Entity Storages. Each storage is accessible by one Writer Worker (managed by the Writer and Archivist) and one Reader Worker (managed by the Reader).

5.4 Underlying Implementation of Analysis in Raphtory

Once an analyser has been created the next question is how it is executed. As discussed briefly above, the responsibility of this is split between an *Analysis Task* and the set of *Partition Readers* within each deployment. Analysis Tasks are instantiated for each submitted query and control the flow of an algorithm at each *flattening_end* of interest, requesting the execution of supersteps, confirming convergence and returning results. The Partition Readers then build the requested flattenings and orchestrate the safe execution of supersteps upon each partition.

5.4.1 Partition Reader

Partition Readers are the processing engine within Raphtory, overseeing the execution of user defined functions on the entities within their partition. Readers operate much in the same manner as the Partition Writer, managing a pool of workers who are each responsible for performing the algorithm on an Entity Storage. As can be seen in Figure 5.17, this means that each Writer Worker has a paired Reader Worker who will be analysing and reporting on the entities which it updates/maintains. Whilst the Reader is in charge of initially establishing connection with new tasks, once this is complete its managed workers will be contacted directly to check their latest watermarks and compute desired supersteps. The Reader Workers handle these requests completely independently, allowing each task to handle the full control logic of its assigned algorithm and operate irrelevant of the number of partitions. Superstep requests are additionally handled serially on the main thread of the Reader Worker, meaning no two requests will run on the same data in parallel. Furthermore, all state generated by a superstep is fully encapsulated and only accessible by other supersteps of the same task. These in combination mean the supersteps of multiple tasks may be interleaved without affecting each others output.

5.4.2 Reader Worker Superstep Execution and Flattening Generation

As discussed above, superstep requests consist of running either ***Setup()***, ***Analyse()*** or ***Return-Results()*** upon a given flattening. All requests contain the chosen analyser and job metadata required for this and are executed in the same standardised manner. The first step in this process, building the flattening, is done by creating the correct lens (according to the Analysis Task type) and injecting this, together with all arguments/job parameters into the analyser object. This is how the analyser may be flattening agnostic, but allows all internal calls to the user API to return the correct results for a given *flattening_end*/window.

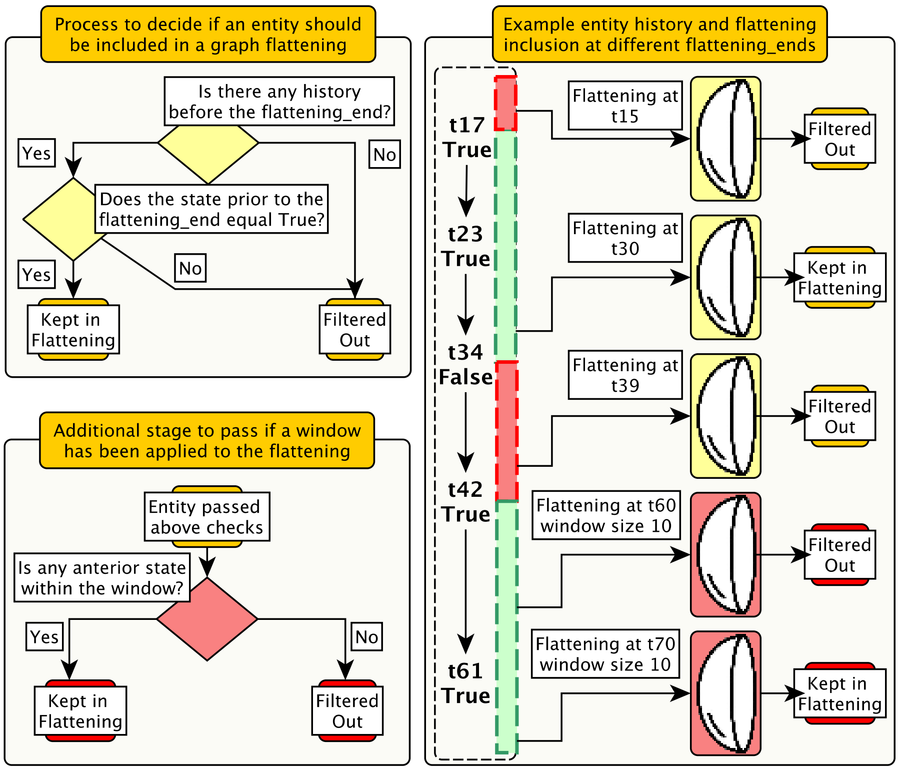

Process to decide if an entity should be included in a graph flattening
Is there any history before the flattening_end?
Yes
No
Does the state prior to the flattening_end equal True?
Yes
No
Kept in Flattening
Filtered Out
Additional stage to pass if a window has been applied to the flattening
Entity passed above checks
Is any anterior state within the window?
Yes
No
Kept in Flattening
Filtered Out
Example entity history and flattening inclusion at different flattening_ends
t17 True
t23 True
t34 False
t42 True
t61 True
Flattening at t15
Filtered Out
Flattening at t30
Kept in Flattening
Flattening at t39
Filtered Out
Flattening at t60 window size 10
Filtered Out
Flattening at t70 window size 10
Kept in Flattening

Figure 5.18: Left - flowchart for deciding if an entity should be kept in the graph for given flattening parameters. The stages carried out by both the flattening lens and window lens are denoted in yellow; only those executed by the window lens are in red. Right - the process executed on an example history of an entity, with the decision given different flattening parameters.

Digging further into the process which the lens undertakes, the left side of Figure 5.18 describes the procedure enacted on each entity, based on the graph flattening definition from Section 3.4. At each stage of the process the entity either progresses or is filtered out. This begins with a comparison between the oldest point in the history of the entity (indexed for O(1) lookup) and the given *flattening_end*, to ascertain if the entity existed within the graph prior to this time.

Entities which fail this test, as they were added to the graph after the *flattening_end*, may be removed without accessing/iterating through their full history, minimising the amount of lookups required. An example of this can be seen in the right side of Figure 5.18, which shows an entity history and how the lens decision differs given different *flattening_ends* and windows. Within this, the first lens requests the state of the entity at t_{15}, but as the first update of the entity is at t_{17} there is no need to check its history, it can simply be filtered.

Entities entering the next stage must have their histories scanned to find the update anterior to the *flattening_end* and extract its value. As the structural history is stored in chronological order, this is done by starting at the head and descending through the nodes until the first one is hit with a key (time) prior to the *flattening_end*. Once the value has been extracted, if it is *true*, the entity may progress as it would have been alive within the graph at this time; if *false* it would not have been present and is filtered. The next two lenses on the right of Figure 5.18 provide an example of this. For the flattening at t_{30}, the prior update is at t_{23} and resolves to *true*, i.e. the entity would have been present in the graph and is kept. Alternatively, for the flattening at t_{39}, the entity is filtered as its prior update is now one node deeper in the history at t_{34} which resolves to false.

Given that no window has been requested, at this point the entity would be wrapped in its equivalent visitor and made available to the analyser. If, however, there is a window, the entity may only be returned if the extracted update is within the established time range. This is confirmed by subtracting the update time from the *flattening_end*, the remainder of which must be less than the window size. The final two lenses of Figure 5.18 demonstrate how this would work on the example entity. The flattening at t_{60} would extract the value of the closest state at t_{42}, which would resolve to *true*, but as the window size is only 10 this update is too far back in time and the entity would still be filtered ($60 - 42 > 10$). On the other hand, for the flattening at t_{70}, the prior update is within the acceptable time range ($70 - 61 < 10$) and, therefore, the entity would be kept.

Handling of Batched Windows

Whilst the flattenings materialised for each *flattening_end* in a time range are done so in series, if the Analysis Task contacting the Partition Reader Worker is a batched window variant it will send all window sizes, along with the *flattening_end*, to execute these within the same superstep. In this instance, once the preprocessing of the lenses has concluded and the superstep function has been run on the first window, the worker will shrink the window within the lens to the next size and re-execute the function on the new set of vertices. This process happens for each window within the batch (starting from the largest to the smallest) until all have been computed and the worker can return to the Analysis Task. Within a superstep response will be the total messages sent for all windows and whether every vertex in all incarnations are happy to halt. For the return of partial results, this will include an array of responses (one from each window size) which the Analysis Task will group by the window size internally, passing each to the

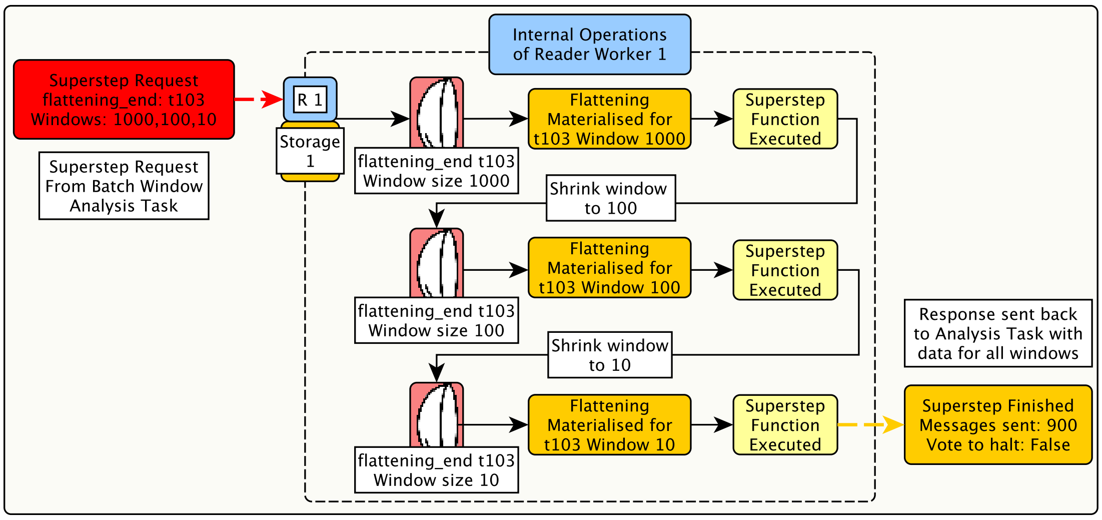

Internal Operations of Reader Worker 1
Superstep Request flattening_end: t103 Windows: 1000,100,10
Superstep Request From Batch Window Analysis Task
R 1
Storage 1
flattening_end t103 Window size 1000
Flattening Materialised for t103 Window 1000
Superstep Function Executed
Shrink window to 100
flattening_end t103 Window size 100
Flattening Materialised for t103 Window 100
Superstep Function Executed
Shrink window to 10
flattening_end t103 Window size 10
Flattening Materialised for t103 Window 10
Superstep Function Executed
Response sent back to Analysis Task with data for all windows
Superstep Finished Messages sent: 900 Vote to halt: False

Figure 5.19: The superstep cycle from the perspective of a Reader Worker. In this example the worker has received the request for a batch of windows which it executes from largest to smallest, utilising the prior set of vertices each time when generating the flattenings with a smaller window.

processWindowResults() function seen above in Section 5.3.2. This is done as it minimises the amount of times the vertices and edges have to be passed through the graph lens and removes the need to message back and forth between the Analysis Task and the Reader Workers for each window size.

An example of this batch execution can be seen in Figure 5.19. Here a Reader Worker has received the request to execute a superstep with three window sizes of 1000, 100 and 10. The first of these flattenings (t_{103} window 1000) is built from the vertices within the Reader Worker's associated Entity Storage. The analyser is then set to work with the final result. Once this has concluded, the window size is shrunk to 100 and a new flattening is established. Note here that the entity set for the larger window is used instead of the original Entity Storage as any entities the bigger window has filtered out would be removed here so there is no point in checking these again. The window is then shrunk for a final time and the output from all three is combined into the response sent back to the Analysis Task.

5.4.3 Isolation of Analytical State and Vertex Messages

Analytical State

From the independent manner in which supersteps are executed within a Partition Reader Worker, and as queries may be run in parallel with ingestion, there are several questions about the safety of writing temporary analytical state to a vertex. The first issue is that the paired Writer Worker could be updating a vertex at the same time as the analytical state is saved, leading to a deadlock. To ensure this does not occur, analysis values are stored in a separate map from any

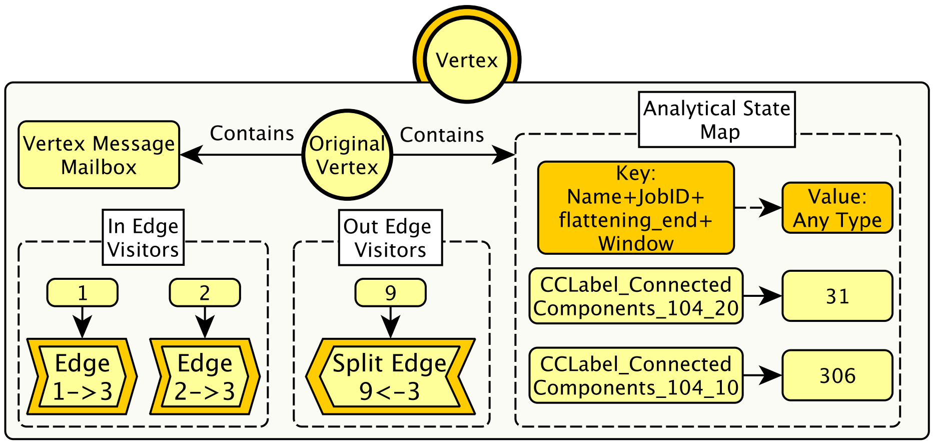
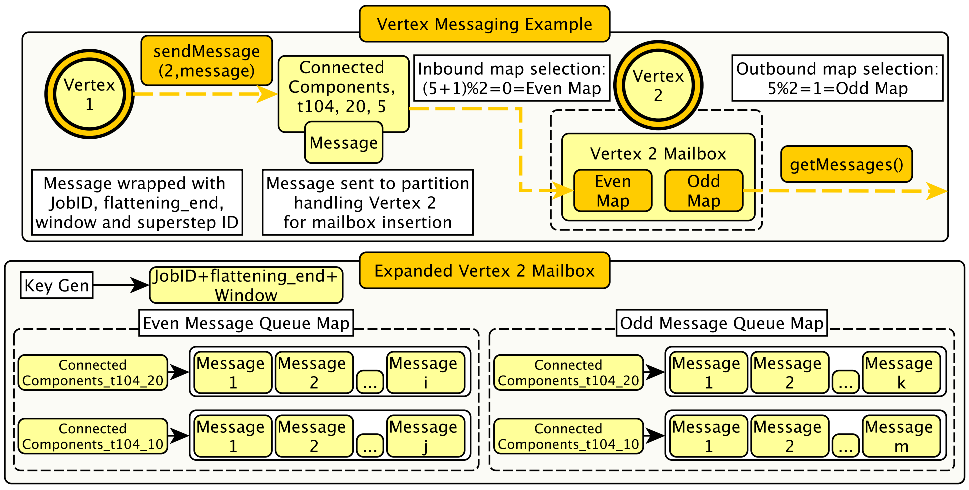


Figure 5.20: An overview of the contents of a vertex visitor. This consists of the original vertex object and its mailbox (discussed below), the in and out edge visitor maps and the analytical state map.

properties inserted via ingestion, which the Writer Worker may never access. In addition to this, there may be the concern that the variable name is non-unique, which could lead to different jobs overwriting the variables value/reading incorrect information. Variable clash may happen across analysers, but definitely occur if the same analyser is being used over many *flattening_ends* and windows. To handle this, when the variable name is given to a state function it is internally combined with the JobID, *flattening_end* and window size, making it unique across all dimensions. An example of this can be seen in Figure 5.20 where the connected components algorithm from Appendix C.1 has been run on two window sizes (20,10) at t_{104}. For the example vertex visitor, the variable 'CCLabel' has been saved internally as *CCLabel_ConnectedComponents_104_20* and *CCLabel_ConnectedComponents_104_10* respectively, allowing the superstep function for these jobs to be intertwined. As can be seen here, the associated value is different for the two windows, i.e. the vertex is in two distinct components, although this may change if the algorithm is yet to converge.

Messaging

Similarly to analytical state, messages sent between vertices must be managed to ensure they are processed in the correct superstep alongside the correct flattening. To enable this, once a message request has been initiated it is packaged up via Akka along with meta information, including the JobID, *flattening_end*, window size and current superstep. As can be seen on the top of Figure 5.21, this is then forwarded to the Reader Worker who handles the receiving vertex, much in the same way that synchronisation messages are handled between Writer Workers. Upon receiving the message the Reader Worker must store this until the next superstep, handled by the *Vertex Mailbox* which can be seen on the bottom of Figure 5.21.



Vertex Messaging Example
Vertex 1
sendMessage (2,message)
Connected Components, t104, 20, 5
Message
Inbound map selection: (5+1)%2=0=Even Map
Vertex 2
Outbound map selection: 5%2=1=Odd Map
Vertex 2 Mailbox
Even Map
Odd Map
getMessages()
Message wrapped with JobID, flattening_end, window and superstep ID
Message sent to partition handling Vertex 2 for mailbox insertion
Expanded Vertex 2 Mailbox
Key Gen
JobID+flattening_end+ Window
Even Message Queue Map
Odd Message Queue Map
Connected Components_t104_20
Connected Components_t104_10
Message 1
Message 2
...
Message i
Message j
Message k
Message m

Figure 5.21: The process a message goes through when sent from one vertex to another. In this example, vertex 1 sends a message to vertex 2 during the 5th superstep of a job with ID 'ConnectedComponents' at *flattening_end* t_{104} with window size of 20. The mailbox of vertex 2 can then be seen to have messages for two windows (20,10) of the connected components query described previously.

The mailbox is broken up into two identical maps, denoted 'odd' and 'even', as one is used for odd numbered supersteps whilst the other is used for even supersteps. This is done so that a vertex reading messages for the current superstep will not accidentally read messages for the next one, which could lead to an incorrect result in both. To select the correct map, when storing a message the accompanying superstep is incremented by 1 followed by a modulo of 2 ($superstep + 1\%2$), placing it within the map for the *next* superstep. As an example of this, in the top of Figure 5.21 a message is sent within the 5th superstep, thus the even map is selected as it should be read in the 6th. Once the map has been chosen, the JobID, *flattening_end* and window size will be combined into a key, extracting the correct queue for the message. If this is the first time the key has been seen, a new queue will be generated and placed within the map. As with the analytical state, this allows different jobs (or the same job with multiple windows) to be run at the same time, interleaving their supersteps. This can be seen in the expanded mailbox in Figure 5.21, where there are different queues for each window size of a connected components job (10,20) in both the odd and even maps.

On the other side of this process, when reading the messages via ***getMessages()***, a similar operation occurs. To select the correct map, a modulo of 2 is performed on the current superstep (without the increment) and the queue is extracted via the same key generator. This can be seen in Figure 5.21 where the analyser has accessed the mailbox of vertex 2 and is reading the messages for superstep 5 from the equivalent odd queue. Finally, there are two things to note here. Firstly, when a queue is pulled in this manner it is wiped from the map to make space for the next superstep or, in the case of the algorithm finishing, so that there is no accumulation of queues filling up memory. Secondly, the whole mailbox process is hidden from the user via the API, with the internal components passing the relevant metadata to allow it to work.

5.4.4 Analysis Task Workflow

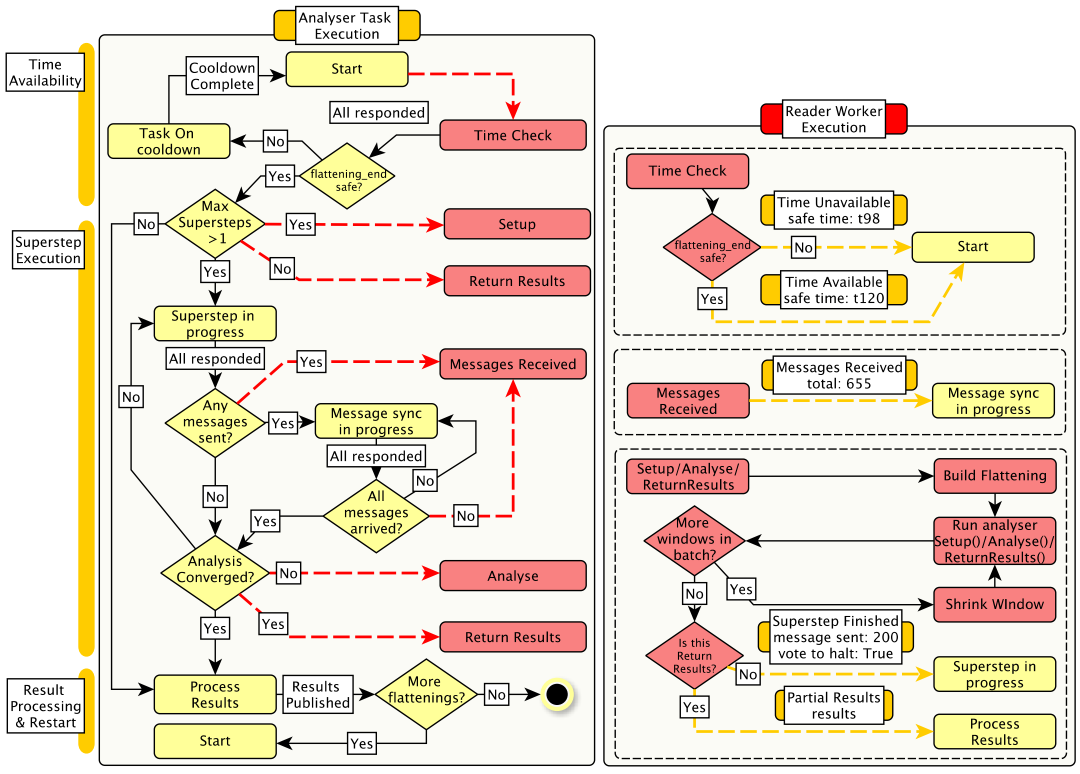


Figure 5.22: The workflow of an analyser split between the Analysis Task on the left and the Reader Worker on the right. Red dashed lines indicate the Analysis Task sending a request to all Reader Workers. Yellow Dashed lines indicate a response from the worker to the task.

Raphtory's component for managing the flow of an algorithm is the *Analysis Task*. An Analysis Task is spawned upon the submission of a user query and is responsible for the full execution of the chosen analyser, tracking which superstep the algorithm is on, which functions each Reader Worker must run and any termination conditions, such as voting to halt or the maximum supersteps reached. This execution workflow can be seen in Figure 5.22 and shows the steps taken across both components, with the chart on the left hand side representing the Analysis Task and the one on the right the Reader Workers. Here it can be seen that the process within the Analysis Task is fully connected as it is in charge of the query from inception through to finalisation, whilst the Reader Workers execute disjoint actions and return to idle once complete to await the next instruction. Across this workflow there are three main stages: Time Availability; Superstep Execution; and Result Processing/Restart.

Time Availability

Once the Analysis Task has confirmed its given analyser is available within the code base, its initial step is to establish if the first flattening may be materialised. This must be checked as the *flattening_end* may currently be beyond the watermarked history and, therefore, unsafe to analyse. For graph and range tasks this requires checking the earliest *flattening_end* submitted by the user, whilst the live tasks send a default 1 to check that ingestion has begun and to extract the latest safe time. The Analysis Task will broadcast this value to all Reader Workers, who will compare this to their latest safe timestamp (as discussed in section 4.7) and respond with a confirmation if it is safe, or a rejection if it is not.

When the Analysis Task has received the responses from all Reader Workers it will check that they all confirm timestamp availability, allowing the analysis to begin. Alternatively, if one or more workers report the time is unavailable the Analysis Task will pause and await a defined timeout before retying. This decouples the analysis from ingestion and means an Analysis Task may be left to work through a large range of *flattening_ends*, wait for a time in the future, or simply run safely on the Live Graph without concern it will get ahead of the Partition Writers and return incorrect results. Lastly, if it is a live task being executed, on confirmation that the given time is safe, this will additionally aggregate the returned times together by selecting the minimum safe point and defining this as the current Live Graph time. This time availability process can be seen in the first section of Figure 5.22.

Superstep Execution - Setup, Analyse, Message Synchronisation and Convergence

Once the *flattening_end* has been confirmed as safe, the analysis may begin. As discussed in Section 5.3.2, the submitted analyser will have a defined maximum number of supersteps. If this number is greater than 1, it is considered an iterative algorithm and the Analysis Task must first run the ***Setup()*** function in each partition. A setup request is, therefore, broadcast to all Reader Workers containing the analyser and the task's metadata consisting of the JobID, *flattening_end*/windows, type of task and current superstep. A Reader Worker receiving this will execute the ***Setup()*** function of the analyser on the chosen flattening and, upon finishing, will return an acknowledgement to the *Analysis Task*, confirming it has finished the superstep, as well as providing a count of any messages its vertices have sent during the execution. Alternatively, if the number of supersteps is one or less, this will be seen as a non-iterative algorithm and the *Analysis Task* will skip straight to ***ReturnResults()***.

Once all the *Reader Workers* have acknowledged the completion of the superstep, the total messages sent is calculated and, if this is greater than 0, the *Analysis Task* requests each *Reader Worker* to report how many messages they have received for the next superstep. This is tracked by the *Reader Workers* via a counter they increment when receiving a new vertex message for a given JobID and will be returned upon the request. Once all responses are in these will be aggregated and compared to the messages sent. If these are equal, it means all deliveries have been completed and it is safe to execute the next superstep. If not, there is a short pause and the message count is rechecked until all have arrived. This is necessary as if a superstep runs without all messages having arrived, vertices may be incorrectly filtered out, calculations may have the wrong inputs and an incorrect result is very likely.

The successful arrival of all messages signifies the end of the setup, or superstep 0, and the main analyse iterations may begin to work towards convergence. This happens much in the same way as the setup, where the task will send a broadcast to all Reader Workers with the same information, but this time requesting they execute the ***Analyse()*** function. Upon completion of the function the worker will decide if its vertices have voted to halt, by all accessed vertices raising a flag, and return this decision along with the number of messages sent.

Once the message synchronisation has completed (as with the setup) the Analysis Task will check for convergence. This will be considered the case if the maximum number of supersteps has been reached or if all Reader Workers have reported that their vertices have reached a final state. Given this is the case the Analysis Task will progress to requesting the results. If the algorithm has yet to converge, a new superstep will be undertaken via another analyse broadcast and the process will start again. This process for superstep execution can be seen in the middle block of Figure 5.22.

Result Processing and Restart

The penultimate stage on any analysis cycle is to extract the partial result from each partition, aggregating and returning these to the user. This is handled in exactly the same manner as a superstep, sending the same information via broadcast, but requesting the execution of the analyser's ***ReturnResults()*** function instead. The Reader Workers receiving this will execute the function and return the partial aggregate as specified by the user. This will be stored within a temporary array and, once all the responses are accounted for, will be passed to either ***ProcessResults()*** or ***ProcessWindowResults()*** depending on if a window was set on the flattening. If within this the user decides to utilise the ***PublishResults()*** function, the data given will be stored within the Analysis Task object, making it available to retrieve via the REST API as discussed below in Section 5.5.

Finally, the Analysis Task will check if there are more *flattening_ends* to process and, therefore, if the process should restart with different parameters. This again depends on the task type, but given there are more *flattening_ends* in the range, the *Analysis Task* will broadcast a new time check to all Reader Workers to see if this next time is available. If, however, this was the last *flattening_end* to be processed the task will become idle, awaiting further requests from the user. This may be to either retrieve the results stored inside, or to kill the task now it has concluded.

Looking at the three possible task types, a graph task will always return to idle as it only stipulates analysis on a singular point in time. A range task on the other hand will increment the current *flattening_end* it has just completed by the chosen user interval to see if this is still within the allocated range, i.e. less than the end time. If it is, the algorithm will restart on this new time via the specified Time Check broadcast. If, however, the new value is greater than the end time, this will be used instead (i.e. ranges are always inclusive of end time). This will be the last execution and the task will become idle after its completion, as with the graph task. Finally the live task will restart if the user has requested the analysis to be on repeat. In this instance if the chosen live increment is set to processing time, the task will sleep for this period, but once awoken will request the latest safe time and begin executing the algorithm on whatever has returned. If event time is selected the live task will increment its Live Graph *flattening_end* (in the same manner as a range task) and will begin polling the Reader Workers for their latest safe time. Once a safe time is available, greater than this value, the analysis will restart on this new safe point. A repeating live task will never become fully idle, with a next flattening always planned until killed by the user. The result gathering and restart process can be seen at the bottom of Figure 5.22.

5.5 Submitting Queries - Analysis Manager

The final component of the Raphtory analysis architecture, and the way in which users may interact with a running Raphtory deployment, is the *Analysis Manager*. This sits alongside the ingestion and maintenance components and provides a REST API[141] to allow the user to submit queries, check progress and extract results. The Analysis Manager itself is stateless and does not compute any results, instead spawning an *Analysis Task* to manage each submitted query. An overview of the components and scope of the Analysis Manager can be seen in Figure 5.23.

5.5.1 REST endpoints and Query API

The Analysis Manager's REST API is split between endpoints for query submission and for accessing the status of established tasks. Looking first at submission it can be seen that three end points exist, one for each type of task (graph, range and live) which may be set running. When submitting a request to any of these endpoints the user must supply a JSON object containing the required parameters for the task type (as discussed in Section 5.2.4). These may then be joined by optional arguments, such as desired windows, and an 'args' array, to pass any additional information pertinent to this run of the analyser. The Analysis Manager, upon receiving a valid request, will convert this into an Analysis Task which contains all the user parameters and may begin executing the desired analysis. List 5.1 shows an example valid query used to compute connected components on the Gab data discussed in Section 6.4.1. Here the required parameters can be seen at the top, including the JobID, chosen analyser and start/end/increment. Note these times are specified in millisecond epochs and refer to the time of the first post within the

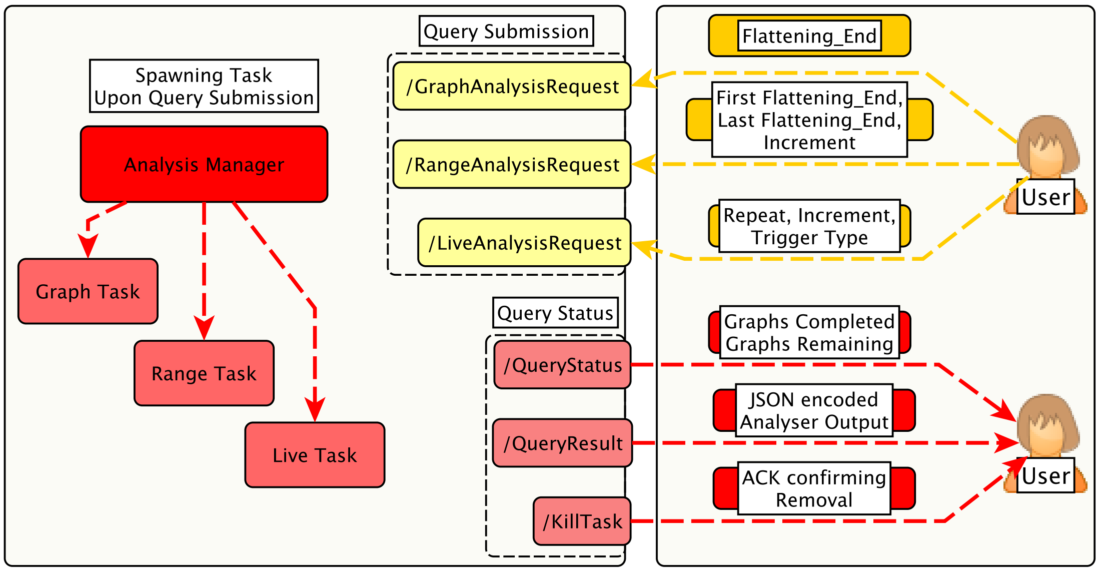


Figure 5.23: The different REST endpoints available from the Analysis Manager.

Gab network, the last post recorded during our scraping period and an increment of an hour. Following this is an optional batch of windows which, in this instance, are for 6, 4, 3 and 2 month periods.

```
{
    "jobID":"connectedComponentsGabRange",
    "analyser":"Algorithms.ConnectedComponents",
    "start":1470801517000,"end":1525372257000,"increment":3600000,
    "windowType":"batched",
    "windowSet":[
        15768000000,10512000000,7884000000,5256000000
    ]
}
```

Listing 5.1: Example range query submitted to execute connected components on the Gab dataset.

Once the query has been successfully submitted the user may leave the task to complete or begin monitoring it via the status end points. There are also three options here consisting of the query progress, extracted results and facility to kill an existing task. When querying for status the user only requires the JobID to which the Analysis Manager will respond according to task type. For a graph task this will consist of whether the analyser has converged on the flattening, for a range task how many flattenings have been executed and how many remain, and for a live task how recently the last flattening completed. When querying for results, the

Analysis Manager will return anything the user has requested to publish via ***PublishResults()***. As with the submitted query the response will be in JSON, a sample of which can be seen in Listing 5.2. This is a subset of the information returned from the connected components query from Listing 5.1 and shows the state of the network in its early stages (20 August 2016 03:58:37) with a window of 2 months applied. Finally, if the user requests a job to be killed the Analysis Manager will return an acknowledgement and begin decommissioning the task. Any ongoing analysis which it was completing will be stopped during this process.

```
{
    "time":1471665517000,
    "windowsize":5256000000,
    "biggest":152,"proportion":0.99346405,
    "total":2,"totalIslands":1
}
```

Listing 5.2: Example result returned for the query in List 5.1. The reported information consists of the *flattening_end* and applied window; the size of the biggest connected component and its proportion of the graph; and the total components/how many of these are islands (vertices without connections).

5.5.2 Handling New Analysers at Run-time

As discussed in [124] a major issue with dynamic/streaming analytical platforms is that once deployed and ingesting data it becomes quite difficult to add new forms of analysis as this often requires recompilation, redeployment or re-ingestion. This clearly can take some time, within which the platform may be offline, leading to functions being able to change only during scheduled maintenance. To remedy this within Raphtory, the user may submit new analysers via the REST API which will be compiled at run-time and utilised by both the Analysis Task and Reader Workers.

Upon instantiation of an Analysis Task it will initially check the provided analyser class path to ensure this exists within the compiled code base. If it does not, the submitted query must contain the source code which the task will then compile (via the Scala run-time toolbox[159]) into an analyser object to utilise throughout the analysis process. If the class fails to compile or the file is missing, the stacktrace will be returned to the user to help fix the issue for re-submission. Assuming the compilation was successful, the task must then propagate this throughout the cluster. This occurs following the initial connection between the Analysis Task and the Partition Readers, whereby the task will broadcast the class paths, requesting all Readers check the classes are present. Once all the responses have been received, the task will forward the provided source code for any missing classes to allow them to build the object. The new analyser object will then be stored within the worker as the 'factory master', labelled by its desired class path. For any subsequent requests for the class this master version may be cloned to utilise within a given superstep run. This means the new classes only require compilation once per partition for the lifetime of the deployment.

5.6 Summary

To summarise, in this chapter we first explored the range of graph algorithms which are available within Raphtory, covering the full structural scope and how these are expanded with the inclusion of entity history. We discussed how all analysis is run on graph flattenings, allowing the user to develop the algorithm once and deploy it at any point within the history of the graph. We introduced time ranges, which allow for flattenings to be built across periods of interest and window batches, which provide a set of temporal depths to investigate at a given point in time; further discussing how these may be combined together. Finally, within the overview, we introduced Live Graph analysis and explained how all prior techniques, including access to history and window batches, could be applied on the latest graph instance.

Following the overview we introduced the Raphtory analysis API, which is broken down into three components. The first of these is the analyser, which encompasses the full algorithm, allowing the user to implement all steps required to converge on the desired result. The second component is the entity visitor, which provides a safe way for the user to access information about the entity, save analytical state and send messages between vertices. Thirdly is the graph lens, which returns the correct entities included in a requested flattening, allowing analysers to be time independent.

Penultimately in this chapter we discussed the underlying implementation details of how the analysis works, both from the perspective of the API and the interactions between the Partition Readers/Analysis Task controlling the overall workflow of an algorithm. Lastly, the higher level REST API was introduced, showing how users may submit queries to Raphtory, check how their ongoing analysis is progressing and retrieve their results.

Chapter 6

Evaluation

6.1 Introduction

Now that we have defined the underlying graph model, discussed how Raphtory builds and maintains such a graph and how analysis may be performed alongside, this chapter provides an evaluation of the current implementation. This in itself has provided a novel challenge as, whilst there are clear methodologies for benchmarking pure streaming systems and batch oriented graph processing systems, no framework exists to combine elements of both. To this end we first define a testbed and clear methodology for the Raphtory evaluation, stressing its features in an automated and reproducible manner.

The evaluation of Raphtory is split between its two key areas, namely ingestion and analysis. Within the ingestion evaluation three major areas are investigated. Firstly, the throughput which can be achieved by the Spout, Router and Partition Manager when scaling up the computational resources allocated. Secondly, the effect of deletions on this attained throughput. Thirdly, the number of updates, and the size of the graph, which can be ingested and maintained across an increasing number of Partition Managers when scaling out Raphtory in a distributed setting. Through this it is shown that Raphtory can comfortably handle the rate of traffic output from many real world datasets, may continue to perform in situations with a high degree of node churn where entities are continuously joining and leaving the network and, finally, is able to scale the size of the stored graph in line with the amount of machines it is allocated, ingesting over 1.5 billion updates across 200 million edges within the largest deployment we have tested (16 AWS m5a.8xlarge VM's).

Following this, the evaluation of Raphtory's analytical capabilities is focused around two use cases and the insights which could be garnered via the unique model and API. The first of these is an analysis of the Gab.ai[160] social network, applying batches of windows across the full history of the network to see how it changed through time under a short and long term lens. This produced notable insights, not just about Gab, but also extracted patterns never previously seen within social networks. The second use case then focused around the Ethereum network, tracking the flow of illicit currency over a three month period following the hack of a Korean exchange[161]. This was done utilising a temporal tainting algorithm, where the history of the edges was employed to ensure the taint only travelled forwards in time. Through this Raphtory was able to track the stolen funds through almost 200,000 wallets and find a selection of exchanges where the hacker had been able to convert it into fiat currency.

Raphtory's capability to perform analytics efficiently was then tested by comparing the implementation of the above Gab use case to a replication implemented in Spark GraphX. Here Raphtory was shown to be over 300x faster in some instances with the overall job taking 110 minutes in Raphtory and over 4 days in Spark.

Finally, following the evaluation of Raphtory, it was felt that there was a need for some standard manner to compare the growing number of systems in the dynamic/temporal graph space. As such we created GraphTides[1], a framework for evaluating stream-based graph processing platforms. Within this we open the discussion of how to merge the important features of a streaming benchmark (throughput, update order, etc.) with those of a graph analytics benchmark (appropriate query workloads) in a standardised/repeatable manner.

6.1.1 Chapter Roadmap

Section 6.2: Evaluation Methodology To begin the evaluation of Raphtory a testing methodology is established to ensure the tests are repeatable. This includes defining the test environment, automating the test process and establishing which metrics are being recorded.

Section 6.3: Raphtory Ingestion Evaluation The ingestion capabilities are then tested. This includes exploration of initial issues, scale up tests for update throughput, scale out tests for maximum achievable graph size and an investigation into the effect of entity deletions on the attained throughput.

Section 6.4: Raphtory Analysis Evaluation The analysis functionality of Raphtory is then evaluated through two different use cases. This consists of a windowing focused workload over the social network Gab.ai and a temporal tainting algorithm run on a large scale cryptocurrency network.

Section 6.5: Comparison To Other Graph Analytics Platforms This analysis evaluation then continues by comparing the implementation for the Gab.ai use case to a replication in Spark GraphX where Raphtory is shown to be much more efficient.

Section 6.6: GraphTides: A Framework for Evaluating Stream-based Graph Processing Platforms Finally, drawing from what we have learnt from this evaluation, we look to open the discussion of how to standardise the evaluation of similar systems in the future, formalising these ideas in the GraphTides framework.

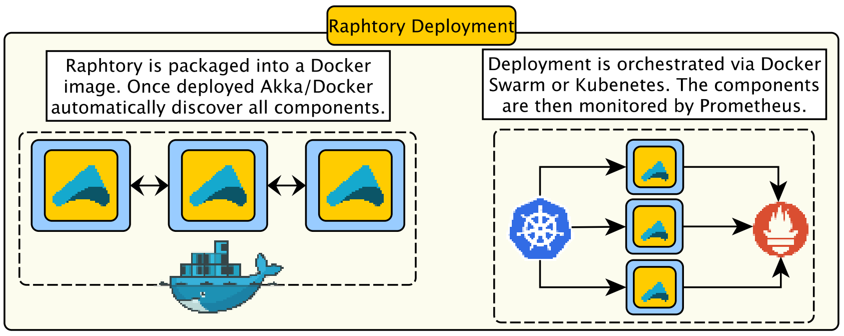

Raphtory Deployment
Raphtory is packaged into a Docker image. Once deployed Akka/Docker automatically discover all components.
Deployment is orchestrated via Docker Swarm or Kubenetes. The components are then monitored by Prometheus.

Figure 6.1: How Raphtory may be deployed onto a cluster of machines via Docker/Kubernetes and how these deployments are monitored.

6.2 Evaluation Methodology

6.2.1 Raphtory Evaluation Testbed

To begin the evaluation of Raphtory an investigation into the manner with which similar stream-based graph processing systems were benchmarked was conducted. Unfortunately no standard manner for evaluation had been established with the algorithms, datasets and test environments varying widely. Therefore, we moved to explore the wider remit of benchmarking literature to develop a methodology which borrows from well established evaluation principles.

To ensure all benchmark tests are as fair and reproducible as possible, this evaluation follows the principles laid out in Jain [162] for test definitions and the Popper[163] conventions for deployment. Jain requires the system and its constituents to be fully defined and the goals of the analysis to be laid out beforehand. The appropriate metrics relating to this goal are then established (e.g. throughput, response time) and the optimum value decided. Following this, the set of possible parameters affecting the performance of the system are constructed (both system parameters and properties of the actual workload) with those to be varied for the experiment chosen and appropriate levels defined for those remaining. The experiment may then be conducted on a variety of system setups, which may consist of both scale up and scale out. To facilitate these varying setups, Popper establishes a pipeline which automates the experiment from inception and execution through metric collection and publication of results.

To begin providing a Popper appropriate testbed Raphtory's implementation was compiled into a Docker Image[1], allowing deployment via container orchestration software (e.g. Kubernetes [164]); as can be seen within Figure 6.1. All deployments then consist of one main actor per container with Akka/Docker handling the discovery phase, connecting all actors together. This means the number of Spouts, Routers and Partition Managers can be set within a configuration file, along with the computational resources they are allocated, and instances will be spawned

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

and set about their given roles. Furthermore, the classes for these actors may also be set within configuration, minimising the need to recompile the image.

To provide metrics on each of these containers, the monitoring tool Kamon[165] is installed within Raphtory, automatically logging system metrics as well as tracking Raphtory internals. These are stored in Prometheus[166], a time series database which has also been containerised. Prometheus provides REST endpoints for all stored data allowing it to be pulled and plotted automatically. Finally, to remove the difficulty of reproducing exact hardware setups, all experiments are carried out on Amazon Web Services (AWS). Through AWS a variety of virtual machine (VM) instances were utilised across the conducted tests. A summary of their specifications can be seen within Table 6.1. The scripts for deployment and plotting of results can then be found within the Raphtory Repository[2], completing the Popper pipeline.

Table 6.1: Amazon Web Service Machines Utilised

Instance Name	**Allocated CPU**	**Available Memory**	**Network Bandwidth**
m5a.large	2 core AMD EPYC 7571 at 2.5 Ghz	8GB	Up to 10 Gigabit
m5a.xlarge	4 core AMD EPYC 7571 at 2.5 Ghz	16GB	Up to 10 Gigabit
m5a.2xlarge	8 core AMD EPYC 7571 at 2.5 Ghz	32GB	Up to 10 Gigabit
m5a.4xlarge	16 core AMD EPYC 7571 at 2.5 Ghz	64GB	Up to 10 Gigabit
m5a.8xlarge	32 core AMD EPYC 7571 at 2.5 Ghz	128GB	Up to 10 Gigabit
m5a.16xlarge	64 core AMD EPYC 7571 at 2.5 Ghz	256GB	12 Gigabit

6.2.2 Recorded Metrics

For all tests executed a selection of key metrics are extracted from those recorded by Kamon, categorised under Host System, JVM (Java Virtual Machine), Akka and Raphtory; an overview of which can be found in Figure 6.2. Starting from the lowest level, each machine hosting a Raphtory component provides metrics on the systems resource utilisation from which the average CPU usage and the amount of inbound/outbound network traffic is extracted. Moving up a layer, inside of each container the JVM is monitored to provide insight into its internal execution and memory management. For execution this consists of a breakdown of the parallelism achieved by each thread pool, of which Akka makes ample use of. On the other hand, for memory management, the allocated heap space is tracked along with the time taken/space reclaimed from garbage collection cycles. Expanding past the base JVM, Akka exposes information about the message processing of individual actors to investigate work skew or which components in the pipeline are bottlenecking those further downstream. This consists of the average processing time of a message, the average time spent in the mailbox of the actor waiting to be processed and the overall mailbox size. Alongside this it exposes errors experienced by actors split between unhandled messages (messages arriving where they shouldn't) and dead letters (messages dropped because the actor is overloaded and its mailbox is full).

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

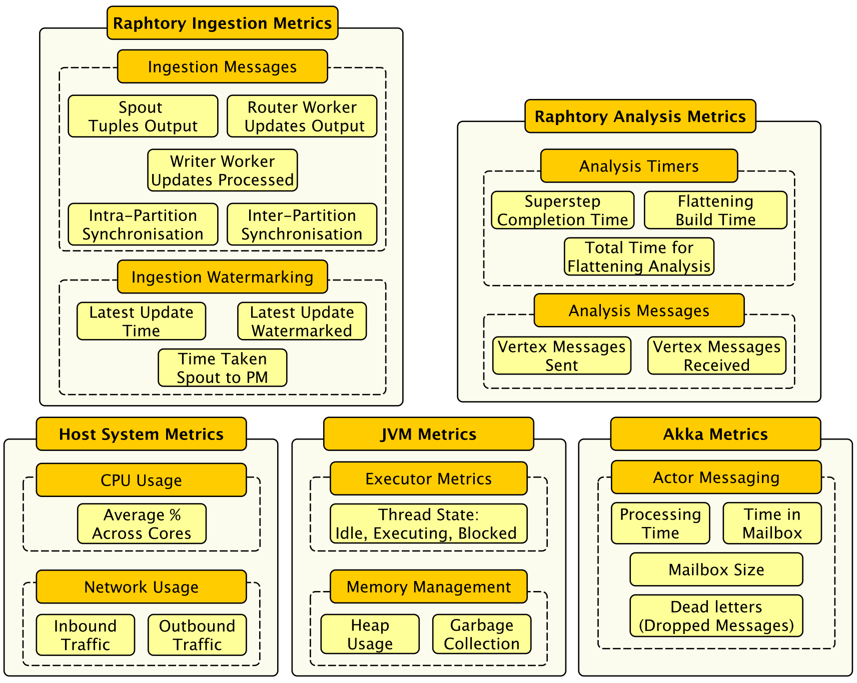


Figure 6.2: An overview of the metrics recorded during all test runs of Raphtory.

Finally, internal to Raphtory, each component tracks its own progress split across ingestion and analysis metrics. For ingestion the full journey of graph updates are tracked. This begins with the Spout recording how many tuples it is outputting, following these into the Router Workers which track how many graph updates these convert into. The Writer Workers then record how many 'primary' updates they are receiving from the Router pool, as well as how many intra-writer (workers in the same partition) and inter-writer synchronisation messages have to be processed. The writers then report the difference between the latest update time (unsynchronised) and the latest safe time which may be analysed, as well as the average time it takes for a tuple arriving at the Spout to be available for analysis. These are great indicators of the ability of the cluster to handle the current throughput.

During analysis a similar process takes place within the Reader Workers and Analysis Tasks, tracking the time taken for each flattening. For the Analysis Task this consists of the overall time for the analysis to converge and the results to return. For the Reader Workers, this consists of the time to complete the current superstep and within this the average time taken to build the flattening via the graph lens. This provides insight into bottlenecks within/between each superstep and if data skew is causing one worker to take longer than the others.

6.3 Raphtory Ingestion Evaluation

The first set of tests conducted focused on the ingestion components of Raphtory, investigating their throughput, ability to scale up and out and the current bottlenecks which will be the focus of future improvements. For this, four tests were conducted, utilising Ethereum blockchain transactions as the underlying data source. The first test looked to highlight initial bottlenecks within the components and establish a sensible testbed (machine sizes and component numbers) for the remaining experiments. Test two looked at scaling up a singular Partition Manager by deploying it within the full range of machines seen within Table 6.1. This doubled the available CPU and memory each time, ascertaining the increase in throughput as more resources were allocated. The third test compared this throughput with Ethereum transactions augmented with an increasing number of deletions (both vertex and edge) to probe the impact that these have. Finally, test four scaled out the number of Partition Managers from 1 to 16 to see how much more of the total Ethereum network may be ingested and modelled as the available machines and memory increased.

6.3.1 Test 1 - Initial Component Comparison

Test Plan

To establish a sensible standard deployment for the subsequent scale up and out stages of evaluation, initial tests were conducted to probe the stability and throughput of the Spout, Router and Partition Manager. The goal here was to discover any issues which could be resolved prior to the 'real' tests and to see which components would be a sensible focus within these. This was run on the minimum viable Raphtory cluster, i.e. one of each component (Watchdog, Spout, Router, Partition Manager and Analysis Manager), with each deployed on its own machine to minimise contention for resources and provide the fairest testbed for comparison. For these tests each component was deployed on a *m5a.16xlarge*, the biggest machine available, with the Spout set to read 300,000 blocks of Ethereum (some 33 million transactions) from disk, providing finer control over the amount of tuples output each second in comparison to pulling from a database or stream. Each transaction was converted into three graph updates, two vertex adds for the sender and receiver and an edge add specifying the transaction. The vertices were then given an immutable property of their wallet hash, with the edge containing a value property specifying the amount of currency sent. Once the cluster was established, the Spout began ingesting the transactions and outputting these at a rate of 1,000 tuples a second. This rate was increased by 1,000 every minute until either maximum Spout throughput was achieved (∼200k messages a second), a component reported a failure (denoted by one or more dead letters) or the data was fully ingested.

Preliminary Run

The preliminary runs of this test proved unsatisfactory, with both the Router and Partition Manager producing dead letters at low throughput, which was clearly a cause for concern. Upon investigation it was noted that whilst the CPU was almost idle within their respective machines, there were periods of sustained ∼100% utilisation across all cores, often followed by the dead letters. By attaching a JVM profiler to these machines it could be seen that these periods were actually garbage collection (GC) cycles. These were 'stop the world'[167] implementations, which would only run once the (large) heap was full and would take tens of seconds to complete the mark and sweep. Once the system was permitted to return to processing, its TCP connections and the Akka actor keep alive messages had all timed out, and the cluster had assumed the node had been lost, hence the dead letters.

To solve this issue several garbage collectors were trialled, with the clear winner being Shenandoah [168]. Shenandoah reduces GC pause times by performing most garbage collection concurrently, including concurrent compaction. This means pause times are no longer proportional to heap size and, therefore, collecting on the large heaps within the established testbed has similar low pauses to much smaller machines. The impact from this may be seen in Figure 6.3, the results from the second run, where in the bottom right plot both the Partition Manager and Router have large GC cycles, but no pauses until the very end. The side effect of this, however, is that Shenandoah makes ample use of the available heap space, making it quite difficult to see how much memory is actually being used within the component. This can be seen in the bottom left of Figure 6.3 where the Router and Partition Manager jump to the maximum available heap and stay there for the remainder of the execution.

There are two important aspects to note here. Firstly, to highlight the massive impact it had, Shenandoah is not plotted against other garbage collectors tested (such as G1 and CMS), not because the same tests weren't run, but because these were simply not fit for purpose and the system crashed before comparable results could be garnered. Secondly, whilst Akka has provided a fantastic base for the development of Raphtory and handles many of the frustrating networking issues that distributed systems are often plagued with, this comes at the downside of transparency. Many of Akka's classes are compiled at run-time, meaning heap dumps compose almost entirely of basic types like strings and tuples. Similarly, as it controls the internals for networking, it is very hard to draw direct correlations between in-bound/out-bound messages through Akka and the network throughput seen in the underlying system metrics (as it does a lot of batching etc. internally). These combined made it difficult to investigate many issues, notably why the garbage collector was keeping certain objects in-memory or why throughput seemed to be dropping, even when well below the available bandwidth of the host machines. This is not inherently an issue now, as can be seen in the tests below, but will need to be revisited as the development of Raphtory progresses.

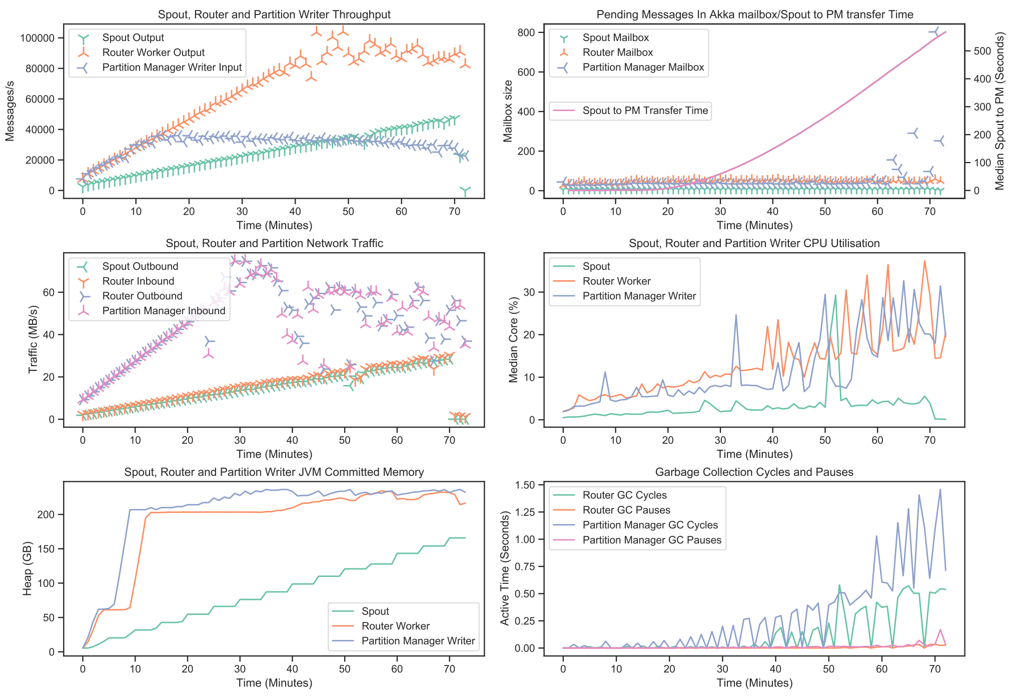


Figure 6.3: The Results and System Metrics of a minimum viable deployment with one Spout, Router and Partition Manager; slowly increasing throughput from the Spout.

Run 2 - Minimum Viable Deployment

Once the Raphtory image had been rebuilt with these GC configurations, the tests could run to completion with the Spout able to fully ingest the dataset and the full graph built within the Partition Manager. The results for the first of these runs, utilising the minimum viable Raphtory cluster, can be seen in Figure 6.3. Looking first at the throughput of each component in the top left it can be seen that the Spout achieved a maximum output of 45,000 transactions a second before running out of data. The Router seemed to be keeping up well with this until the 40 minute mark where its output became more erratic. At this point the Router had reached a maximum output of 80,000 updates a second, as each transaction translates to three graph updates (∼100million in total). The Partition Manager unfortunately lagged substantially behind both of these, keeping up with the Router until 40,000 updates a second, at which point it appeared to saturate.

In an attempt to discover the cause of this saturation it was checked where the Partition Manager or Router had been bottlenecked. Looking first at the CPU in the middle right of Figure 6.3, whilst both the Router and Partition Manager are trending upwards, overall neither were highly taxed and this was unlikely the cause of either not keeping up with the incoming data. Looking next at the network I/O in the middle left of Figure 6.3, it can be seen that whilst the Spout output and Router input are consistently in-sync, the Router output drops

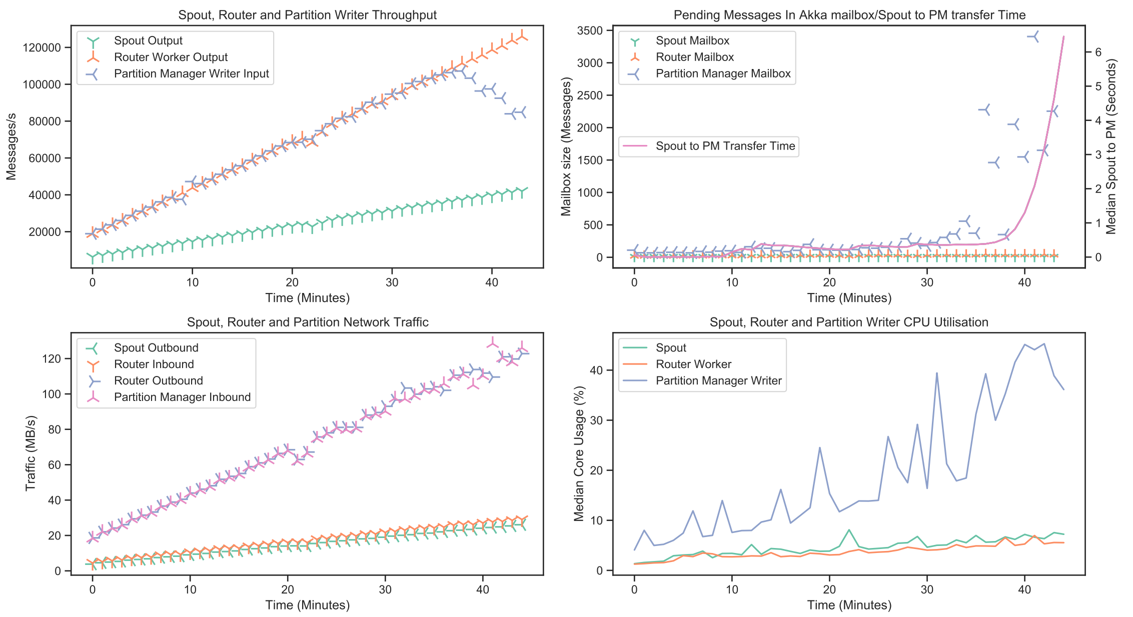


Figure 6.4: The Results and System Metrics of a deployment with one Spout, four Routers and one Partition Manager; slowly increasing the output from the Spout.

and becomes irregular shortly before its maximum achieved output. Even though this is much lower than the throughput specified for the machine by AWS this could explain the drop off of the Router, but does not explain why the Partition Manager saturates well before this. Finally, the top right of Figure 6.3 shows the size of each actors mailbox and the time taken on average for an update to reach the Partition Manager from the Spout and be processed. Interestingly here, the Partition Manager's mailbox is almost always empty, i.e. it has completed all received updates. This is even the case after the time taken for updates to arrive and be processed begins to massively increase. The point of this increase, around the 20 minute mark, also seems to coincide with the point of saturation in the updates processed by the Partition Manager. This, however, did not seem to be the fault of the Partition Manager which was completing all arriving updates as per its mailbox.

Run 3 - Four Router Deployment

From the mailboxes it appeared that the Router was the cause of the bottleneck, with the updates it was outputting clearly not arriving at the Partition Manager until well after it reported they were sent. As such, a tertiary run was executed increasing the number of Routers to four; the results of which can be seen in Figure 6.4. In this run the Spout reached a similar throughput to the prior test, but now the Routers were able to keep up fully with this, producing three times the amount of updates until the test was stopped. Interestingly, the Partition Manager was also able to keep up with this, processing over 100,000 updates a second, at which point it began to fail. The network I/O of all components was also fully in line within this run and the Routers' CPU utilisation dropped to idle whilst the Partition Manager is clearly trending up; as can be seen in the bottom two plots. Finally, looking at the mailbox size and transfer time, the drop in Partition Manager throughput now coincides with an increase in the size of its mailbox and the time taken for updates to be processed; suggesting the Partition Manager is now actually saturated. Whilst this does not provide an exact cause of why the singular Router became overwhelmed, it does establish a testbed which may be taken forward. There are clearly improvements to be made here, possibly implementing some sort of back-pressure[169] to allow components to communicate when they are receiving too much traffic. Alternatively, the pipeline could be changed to pull based where Routers and Partitions request data and would, therefore, not be overloaded by additional messages when something like a GC pause occurs. This being said, whilst not directly comparable, this is ingesting the same throughput as the results reported by Kineograph[10] with 8x fewer machines, correct event order and no batching.

6.3.2 Test 2 - Scale Up Ingestion Throughput

Test Plan

Following the establishment of a testbed where the Spouts and Routers would not interfere with the Partition Manager, the purpose of the second test was to investigate the capability of Partition Managers to make use of available resources when scaling up the number of CPU cores and allocated memory. Thus, within each run the Spout and Routers remained on m5a.16xlarge instances, whilst the Partition Manager was deployed on each of the virtual machines seen in Table 6.1, beginning with the m5a.large (2 cores, 8GB RAM). The output of the Spout was then ramped up in the same manner as the prior test until the Partition Manager failed. Again the Ethereum transactions dataset was utilised, but this now began from block 1 with all 10 million blocks available. Upon completion of a test run, the maximum achieved throughput was noted and the next run established. After five runs of the same instance type had completed and an average gathered, the Partition Manager's instance was upgraded to the next VM, allowing the next five runs to commence. Once all runs had completed, the maximum throughput could be compared across instance type to see the degree of scalability achieved.

Test Results

The results from this test can be seen in Figure 6.5, with the smallest machine achieving a throughput of 9,000 updates a second and the largest achieving over 120,000 in some runs. These results are very promising, but may be better understood given the context of real datasets/use cases. The Ethereum network itself completes ∼10 transactions a second[170]. The average number of tweets generated across Twitter is ∼6000 a second[171]. Finally, for the largest of datasets, there were 134 billion (non-cash) monetary transactions across the whole of Europe in 2017[172], averaging to a rate of ∼150,000 a second. Whilst there will be times when these use

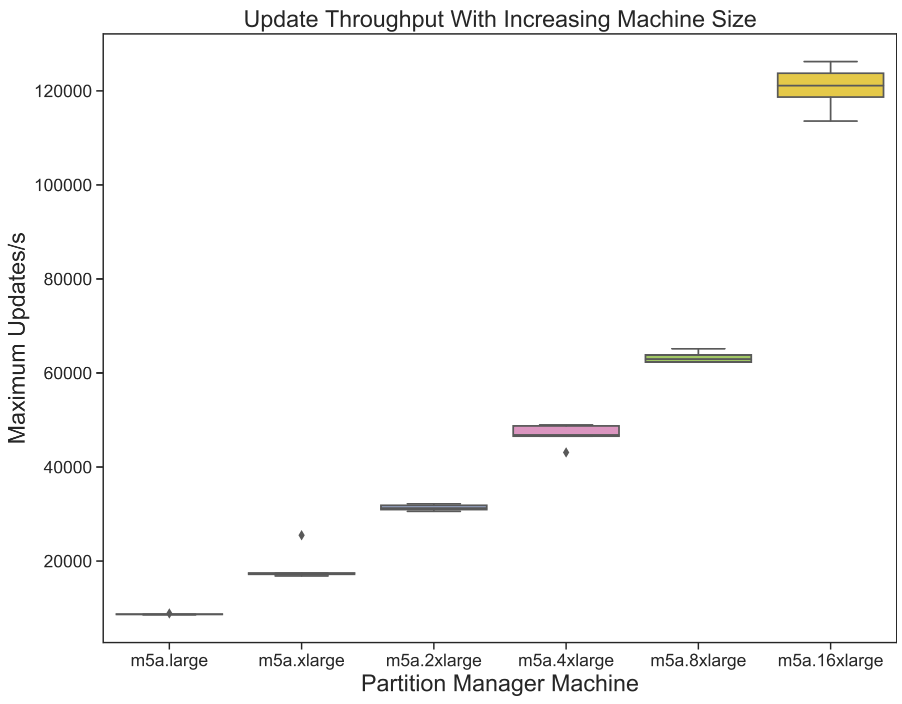


Figure 6.5: The maximum updates processed per second by Partition Managers deployed on VMs of increasing size, doubling CPU cores and allocated memory each time.

cases spike dramatically, possibly by orders of magnitude, it clearly contextualises the speed at which Raphtory may operate. Furthermore, additional throughput will be gained by scaling out, which would be required anyway to store all the data within a real world deployment(as explored in Section 6.3.4).

There are also two notable patterns within these results. Firstly, the throughput appears to be more variable as the machine size increases, but this is a by-product of recording the absolute maximum throughput. When the smaller machines become overwhelmed they quickly crash, whereas the larger machines are more stable, having ample resources to buffer updates and attempt to hold out before finally subsiding, generating some variability. Secondly, the increase in throughput between the first five machines appears linear, but then this doubles between the 8xlarge and the 16xlarge. This suggests a bottleneck is not present within the largest machine, but exists in all those prior. Unfortunately, due to the issue of Akka obscuring the finer details of thread usage, heap objects and network traffic, the investigation into this bottleneck will be future work once a better internal view can be resolved.

6.3.3 Test 3 - Deletion Workloads

Test Plan

Following the scale up evaluation, the third test was to investigate the effect of deletions on the achieved throughput. To allow these to be comparable to the figures reported in Section 6.3.2, the Ethereum transactions dataset was once again utilised, augmented with increasing percentages of deletions. To create these deletions the Routers, upon sending out a vertex or edge addition, had a configurable probability of also sending an entity removal for the same edge/vertex, set at the next time step ($current\_block + 1$). For this test the established four Router testbed was utilised, with two Partition Managers deployed to include inter-worker synchronisation within the evaluation. These Partition Managers were set to run on 8xlarge VMs, with all other components each allocated a 16xlarge.

Once deployed, the Spout slowly increased the throughput in the same manner as all previous tests until a maximum was reached. This was conducted five times per setting to gather an average, starting with no deletions to provide a baseline comparison. Next 0.1% edge deletions was run to demonstrate the difference between this, no deletions and when vertex deletions were included. Finally, both types of deletions were enabled, starting at 0.1% and increasing through to 8%. Note, during all runs the same pseudo-random seed was utilised, ensuring the generated data remained consistent across all runs.

Test Results

The results from this test can be seen in the top plot of Figure 6.6. Two Partition Managers were able to almost double the throughput of their singular counterpart in the prior test, reaching ∼115 thousand updates a second, when no deletions were included. Upon enabling edge deletions this throughput decreased by a marginal amount, as most of the generated updates would require some synchronisation between workers, but demonstrates that these can be integrated without largely affecting the system. The real notable drop occurred when vertex deletions were enabled, reducing the throughput by over half with 0.1% of vertex additions being paired with an equivalent deletion. This reduced further as the deletion percentage was increased, ending at one tenth of the original throughput when 8% of additions spawned a removal.

This is not unexpected as even 0.1% per update is larger than the rate of node churn seen within most real world networks, where vertex deletion is rare. The process for generating deletions additionally means that vertices with more transactions have a greater probability of deletion; the opposite of which is observed in e.g. social networks[173]. This is, therefore, clearly a worst case situation where the highest degree vertices are deleted, generating many more synchronisation messages to be handled. These can be seen in the bottom plot of Figure 6.6, where the number of synchronisation messages per update increases from 0.2 with no deletions (25,000 synchronisation messages for 110,000 updates) to almost 10 with 8% deletions (80,000 for 10,000 updates). Even with this being the case, Raphtory is still able to ingest above the

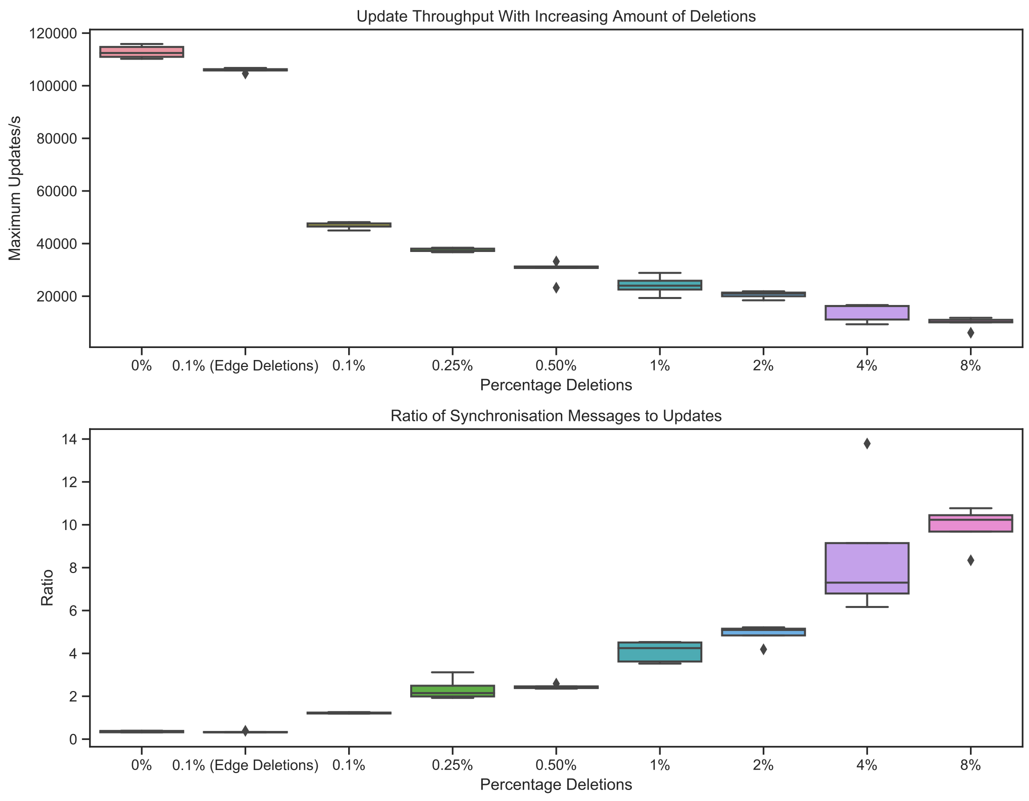


Figure 6.6: Top - The maximum throughput achieved across two Partition Managers when increasing the number of deletions within the underlying dataset (Ethereum transactions). Bottom - The corresponding ratio of synchronisation messages to graph updates for these deployments.

rate of throughput for several of the uses cases introduced in Section 6.3.2.

6.3.4 Test 4 - Scale out

Test Plan

The final test for ingestion then looked at how well Raphtory was able to make use of additional resources when scaling out the number of Partition Managers. As throughput on a singular Partition had appeared to satisfy many real world datasets, this focused on the volume of data which could be processed and the size of the graph which could be stored in memory. The Ethereum transactions dataset was once again utilised with the goal of seeing how much of the 10 million blocks and 700 million transactions could be modelled and stored in each deployment. As with the scale up tests the resources were doubled each time, beginning with 1 Partition Manager deployed on an 8xlarge VM and increasing through to 16 Partition Managers. Each deployment was run five times with a conservative throughput below the recorded maximum. This began at 40,000/s for the singular partition and was increased by 20,000 each time the partitions were doubled. Updates then continued until one of the Partition Managers reported it could no longer store new updates, at which point the size of the graph was recorded, the cluster was decommissioned and the next deployment began.

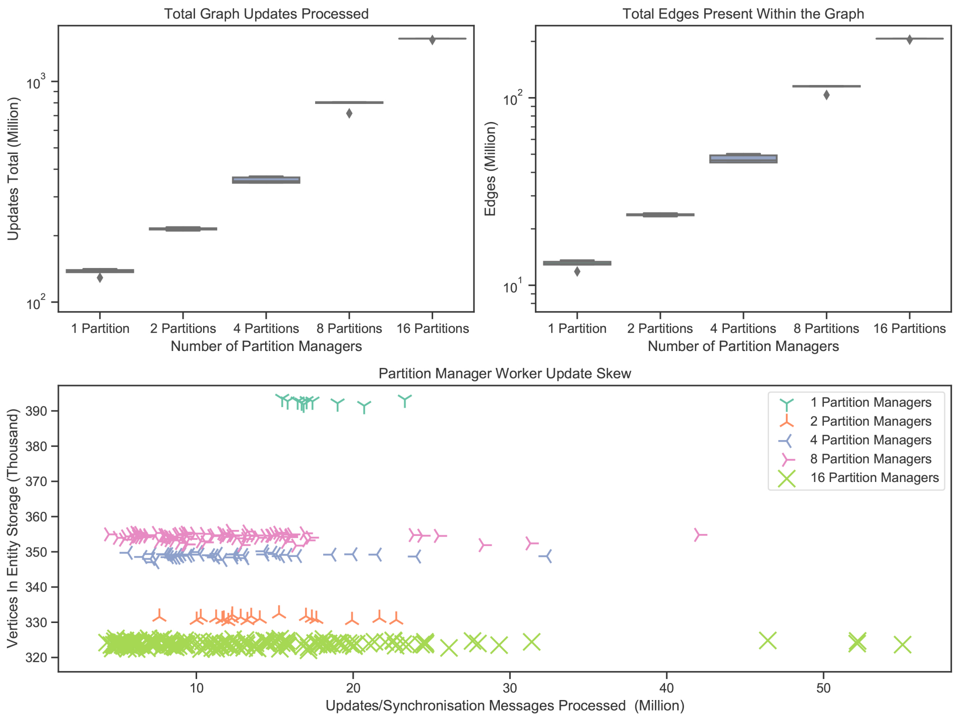


Figure 6.7: Top - The total updates processed when increasing the number of Partition Managers deployed to ingest Ethereum transactions and the count of distinct edges these created. Bottom - The skew in updates processed by each Writer Worker across Partition Manager deployments. Note: The Y Axis appears constant for each deployment due to the hashing partitioning ensuring each Writer Worker has the same number of vertices.

Test Results

The results for this test may be seen in the top of Figure 6.7. Here the singular Partition Manager was able to process ∼130million updates (43 million transactions) across 13 million edges, encompassing 6% of all Ethereum transactions. Reassuringly, as the partitions double, the number of processed updates closely follows, with the 16 partition deployment able to ingest greater than 1.5 billion updates across 200 million unique edges (74% of the total Ethereum transactions). This is especially positive as within the largest deployment almost 99% of edges would require some form of synchronisation message and ∼90% would be split and stored within two partitions.

The biggest issue faced was the underlying skew within the dataset with certain vertices, notably exchanges and Ethereum contracts, being involved in thousands of times more transactions than the average wallet. The Writer Workers, which are tasked with handling these entities must, therefore, store a disproportionate amount of edges and history and are always the first component to report an issue. This can be seen within the bottom plot of Figure 6.7 where all size deployments have clear outliers in the amount of updates processed, even though the number of vertices are evenly divided. A notable example of this is the small collection of workers within the 16 partition deployment which processed over 50 million updates; ten times that of their lightest loaded peer.

6.4 Raphtory Analysis Evaluation

Following the assessment of Raphtory's capability to ingest and maintain large datasets, the second half of the evaluation focuses on its ability to perform analytics on the generated graphs and garner new insights. For this, two use cases are presented. The first looks at applying windowed flattenings over the social network Gab.ai[160], investigating the structural characteristics of the network as it evolves through time by focusing on user interactions. The second performs taint analysis[174] on the Ethereum dataset ingested above, tracking the diffusion of currency stolen during the hack of the UpBit Exchange[161]. All ingestion and analytics was performed on the established four Router testbed with four Partition Managers deployed on 8xlarge instances to maintain the graph. The raw results from the executed analysers was then pulled back via the Analysis Managers REST API for plotting.

6.4.1 Social Network Window Analysis - Gab.ai

Gab.ai is a fairly new social media platform, in many ways similar to Twitter, where users may post character limited 'gabs' which can then be shared, liked and commented upon by their followers. Gab has received quite a lot of attention in recent years as it champions "free speech, individual liberty and the free flow of information online" and has gained quite a notorious user base[175]. However, content aside, it provides a great use case for social network analysis as, up until quite recently, full access was provided to its internal REST API. By crawling this the full Gab network could be downloaded and ingested into Raphtory. This consists of 95 GB of raw data, containing ~19 million gabs and replies posted between 10 August 2016, the start of the Gab network, and 5 May 2018, when an API update made it difficult to identify new posts.

This dataset is, therefore, perfect for investigating the efficacy of windowed flattenings at extracting short and long term structural patterns, as it does not suffer from the underlying sampling bias which plagues similar datasets, e.g [176]. For this the posts were ingested into Raphtory converting them into a user$\longrightarrow$user interaction graph via a custom Gab Router. As explored in Section 4.4.3, within this abstraction users are represented as vertices and when a user comments on a 'gab' an edge is draw between them and the original poster. The contents of posts and user profiles are stripped away during this process to look purely at the structure of user interactions. Once the data was fully ingested, batched window range analysis tasks

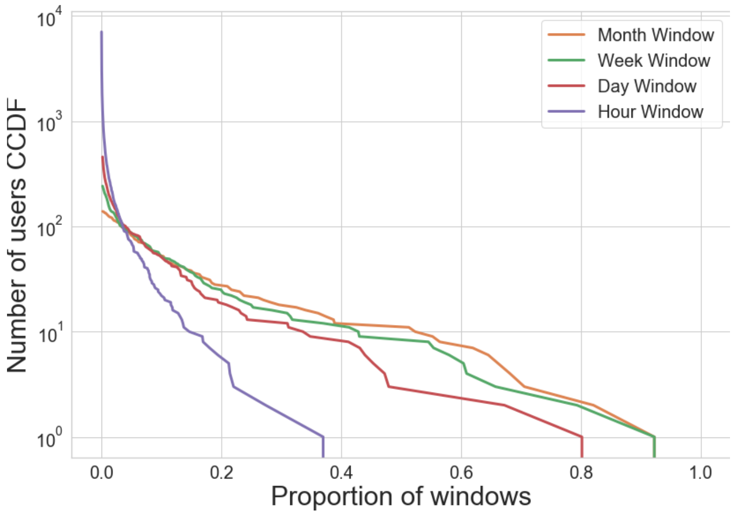


Figure 6.8: CCDF for proportion of time a user spends ranked in the top 20.

were set running across the full history of the network, moving forward an hour at a time and executing the analysers on windows ranging between an hour and a year. The results of these were then compared to see how they differ across time and across the window sizes. These results are additionally discussed further within the full publication[21].

In-degree User Ranking

The first algorithm run on the network was a singular step local scope analyser looking at ranking the users based on their in-degree. As this is an interaction network, a user with more inbound interactions is someone garnering attention/popularity. By extracting the top users at differing temporal depths we can see how many users ever become influential on the platform, and for how long this status lasts. Each Reader Worker was, therefore, tasked with ranking the vertices within their Entity Storage based on the size of their incoming edge map and returning the top 20 users for each window. These could then be aggregated within the Analysis Task to gather the global top twenty users. The code for this analyser can be seen in Appendix C.2.

Figure 6.8 shows the results of this analyser run over the network with windows of an hour, day, week, and month, plotting the proportion of windows a user spends in the top 20 at each timescale. This unearths two interesting properties of Gab. Firstly, nine users ranked within the top 20 in more than half of the month windows are also found in nearly half of the week windows, and 30% of the day windows, supporting the finding that Gab users display strong elitism [177]. Secondly, when looking through a timescale of a day, we see a large pool of more than 800 different users enter the top 20. For the most part, however, this influence is very short lived, with nearly half of these entering the ranks once and never again, demonstrating a strong ‘fifteen minutes of fame’ effect.

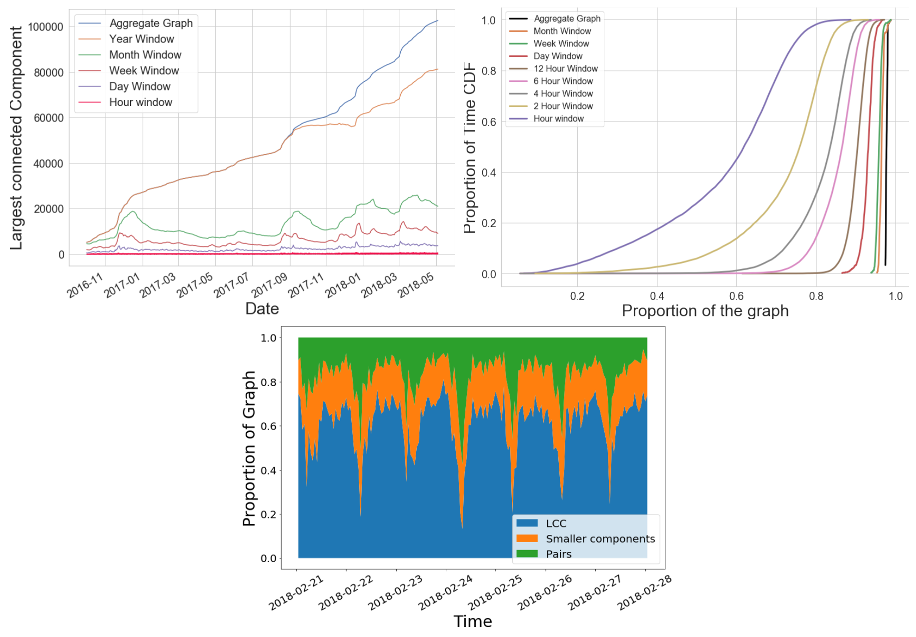


Figure 6.9: Left - The largest connected component (LCC) across the observation period. Right - CDF of the size of the LCC as a proportion of the total graph for each window size. Bottom: A zoom in on the hourly window for an average week.

Largest Connected Component

The second algorithm to run on Gab was then a multistep global scope analyser, calculating the largest connected component (LCC) within each flattening. This is implemented via label propagation where during setup all vertices label themselves with their ID and send this to all their neighbours (incoming and outgoing). In each following analyse superstep, vertices which have received messages check to see if these contain an ID with a value less than their current label. If this is the case they use this as their new label and propagate it forward to all neighbours. If not they vote to halt, keeping their current label. Once all vertices have voted to halt, the Reader Workers group their vertices per label and send the aggregates to the Analysis Task. The task summarises these values and reports the LCC in terms of total size and relative proportion of the overall graph. The code for this algorithm is available to view within Appendix C.1.

Extracting the LCC was chosen to run on Gab as its user-base is characterised by homogeneity, especially driven by political topics. It was, therefore, interesting to investigate the extent to which they form a single interacting community. Aggregate graphs of social networks almost always show the majority of users in a network are part of a ‘giant connected component’[154], but by applying a range of different window sizes to this we could dig deeper into how this grows and changes throughout the observation period.

The top left of Figure 6.9 shows the total size of the LCC across time, comparing the aggregate graph to flattenings with windows of an hour, day, week, month and year. When looking at the aggregate and yearly windows, a general upwards trend can be garnered as the network appears to grow in size and new users interact with the established community. However, viewing this on the monthly and weekly scales it can be seen that the LCC periodically swells quickly and then shrinks back down. This is caused by an influx of new users joining the network following real world events popular on Gab (notably Trumps presidential inauguration and the Unite the Right rally in Charlottesville), who mostly leave after the news cycle has moved on[21]. This can be seen clearer on the right of Figure 6.9 which shows the proportion of users connected by the LCC across an expanded set of window sizes. Here at a window size of one day (and all longer windows) the LCC connects 90% of users for the vast majority of flattenings. This means the spikes in LCC size consist of almost the whole of Gab and actually the overall size of the network is fairly consistent across the observation period, not continually growing as it appears from the aggregate. This also shows that the giant connected component still exists at this scale.

In contrast with this, however, the smaller the window size observed below a day, the higher variability found in the size of the LCC. As might be expected the number of users (and hence the absolute size of the LCC) varies considerably with time of day when studied at the hour level. In fact, at this timescale Gab usage is highly diurnal, with the proportion of users who are part of the LCC varying hugely across peak and off-peak hours; driven by a userbase that is largely US and Europe-based [177]. This can be seen in the bottom of Figure 6.9 which focuses on the proportion of users in the one hour LCC for an average week. At peak hours, 70 – 80% of active users are part of a single LCC. At off-peak hours, the LCC contains no more than 30% of users and Gab becomes several smaller completely disconnected networks that can be thought of as isolated groups talking among themselves. The daily shattering and reforming of the LCC appears to have never previously been observed in social network data. This shows the ability of Raphtory to extract new insights from datasets, even when running standard graph algorithms. Furthermore, this batched window task consisted of over 136,000 flattenings which Raphtory completed in one pass without issue.

6.4.2 Cryptocurrency Taint Tracking - Ethereum Network

Whilst the majority of cryptocurrency usage is for legitimate transactions, it has also opened a new way for criminal entities to exchange and launder money[178]. This has lead to the development of methods to track illicit currency throughout blockchain networks, warning businesses and exchanges not to accept trades from linked entities/wallets and assisting the authorities to bring those involved to justice. A popular form of this is to build the transactions into a network and 'taint' known bad actors, then allow this taint to propagate throughout the network via a contagion algorithm. The nodes which become tainted during this process may then be investigated further for their links to the original crime. In this section such an algorithm is showcased for use within Raphtory, deploying this to trace the path of stolen ether, following the hack of the UpBit exchange in late 2019[161]. During this attack 342,000 Ethereum ($50 million) was stolen and transferred to a singular wallet, after which the attacker began fanning this money into smaller and smaller amounts in an attempt to hide its origin and transfer it into real fiat.

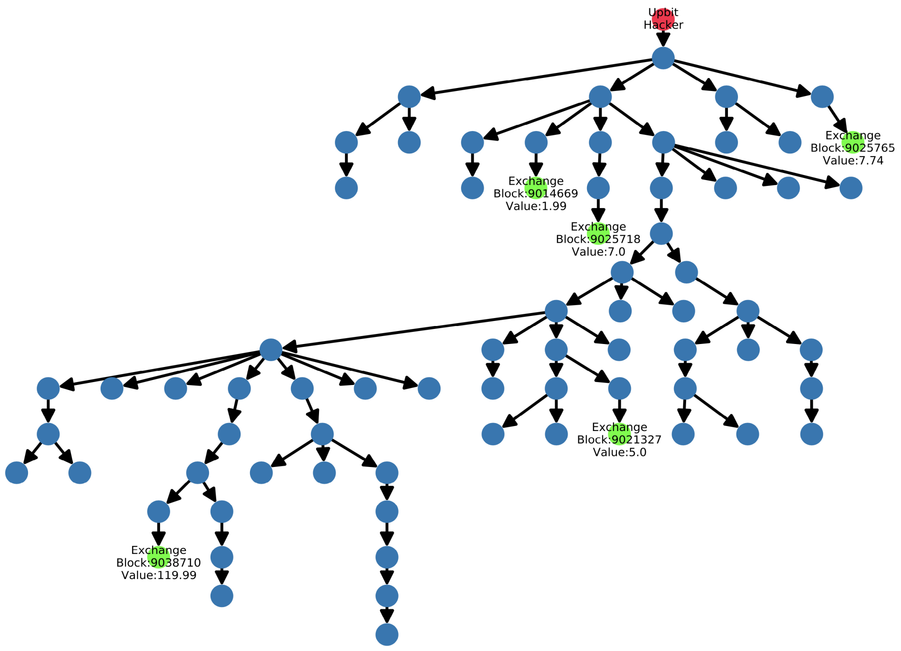
Unlike other cryptocurrencies, such as Bitcoin which have mixers[179] and multi-in/out transactions to obfuscate the source of currency, Ethereum has one input (the sending wallet) and one output (the receiving wallet). For the showcased algorithm the traditional tainting utilising FIFO and haircut[174] may, therefore, be forgone, instead demonstrating how Raphtory may calculate a path through the graph, moving forward in time, utilising the history native within the underlying model. This was run on a singular flattening across blocks 9,000,000 to 9,300,000, encapsulating the three months following the initial hack. The set objective was to trace the currency until the latest block, tagging any exchanges which had been reached. The amount sent to the exchange could then be recorded and the wallet theoretically passed onto the authorities as a possible lead. The addresses for known exchange wallets was embedded into the analyser after being scraped from the verified listings of Etherscan[180], a reputable Ethereum analysis site.

Temporal Contagion Algorithm

To begin the algorithm, in the setup, each Reader Worker checks through its vertices to see if it contains the initial source of the taint, a vertex with the ID property matching that of the UpBit hacker. If this is located within the Entity Storage, all outgoing neighbours with a transaction after the time of infection (the block including the hack) are sent a message informing them they are now tainted. This message includes a new time of infection, which will be the time associated with the first interaction of the vertex with the tainted node after the original infection time, extracted from the history of the edge. This way the infection only travels forward in time, never propagating down edges which occurred prior and would not contain any tainted currency; minimising the amount of false positives.

In all subsequent analyse supersteps, nodes which have received messages informing them of their tainted state will find the message with the earliest infection time and label themselves with this. These nodes will then once again propagate the taint forward on all edges which have an interaction after the given time of infection. During this process if a vertex which is already infected receives new messages, it will check if the earliest infection time within these is prior to its record time and if so will repropagate. This is because illicit edges may have been missed with the prior time filtering them out. If, however, the times are later, it is simply a cycle and is ignored. Finally, the algorithm burns out once it reaches an infection time which has no edges occurring after it, or an exchange is hit which will not propagate forward as this would infect innocent nodes. An infected exchange will record the total amount of ether sent from this tainted wallet to itself after the given time of infection, extracted from the history of the 'value' property.

Once the algorithm has concluded, the tainted vertices and connecting edges are extracted as a subgraph and sent to the Analysis Task. Here the partial subgraphs from each worker are aggregated into the final result and published for the user to access. This algorithm can be seen in Appendix C.5.



Upbit Hacker
Exchange Block:9025765 Value:7.74
Exchange Block:9014669 Value:1.99
Exchange Block:9025718 Value:7.0
Exchange Block:9021327 Value:5.0
Exchange Block:9038710 Value:119.99

Figure 6.10: Subsection of the tracing of the UpBit hacker through the Ethereum Network. Looking at the first 40,000 blocks, prior to the ramping up of transactions. The hacker is coloured in red, exchanges are coloured green and tainted wallets in-between are coloured blue.

Results

Upon completion of the flattening, the subgraph extracted was much larger than expected, with almost 200,000 vertices tainted by the algorithm. It appears that whilst the hacker initially made very few movements, after the first month this began to massively ramp up, breaking their spoils into thousands of tiny amounts and fanning out across a large array of wallets. Of all the nodes touched, only 78 were known exchanges with ∼12,000 Ethereum extracted in total, roughly 4%. This suggests that the hacker is understandably very cautious with this process and it may be years before the full amount is siphoned out. However, as ether has been extracted these transactions could prove an initial lead to trace back to the original actor/group responsible.

To provide a visual insight of how this looks, Figure 6.10 shows the tainting algorithm running on the first 40,000 blocks following the attack, prior to the ramping up of transactions. The initial fanning out of transactions can be seen here, alongside paths of arbitrary depth beginning to form. This is probably done as most algorithms set a maximum depth to check and by quickly exceeding this the currency will be safer to exchange. Interestingly, however, exchanges appear at all depths within the tree, with notable temporal anomalies. For example, the exchange in the bottom right was reached 3,000 blocks (roughly 12 hours) prior to the exchange at the top, two hops away from the original source. These structural/temporal patterns prove an interesting line of inquiry for which Raphtory is clearly suited and is discussed further in Chapter 7.

6.5 Comparison To Other Graph Analytics Platforms

The final tests carried out were to compare the capabilities of Raphtory to those of the graph processing systems introduced in Section 2.5. It was initially planned to perform ingestion comparisons against graph streaming systems (i.e. Kineograph[10] and Weaver[11]) and analysis comparisons against those with temporal capabilities (i.e. Chronograph[15] and Greycat[129]). Unfortunately, the vast majority of platforms discussed in this section are closed source and, therefore, not possible to compare in the same environments or with the same use cases. Those which were accessible, notably only Weaver and Greycat, were then plagued with issues when attempting to install and run even their basic examples. It was, therefore, decided to compare Raphtory to Spark GraphX as it would be the option of choice for the majority of people attempting to perform similar analysis in the real world, given Spark's prevalence and the lack of other appropriate platforms.

6.5.1 Test Plan

As Raphtory was going to be compared to a 'batched' processing system, it did not make sense to compare the speed of ingestion. The devised test, therefore, had to focus on the analytical capabilities of both systems. Spark does not natively support temporal analytics, but can filter datasets on timestamps, allowing it to effectively extract the same graph flattenings as Raphtory. It was, therefore, decided to repeat the windowed analysis of Gab as this could be implemented on both platforms. This would focus on the connected components algorithm as this has a global scope, is iterative in nature and requires a final aggregation of all allocated values; stressing the system's analytical, messaging and querying capabilities. This would run over the full range of the dataset, moving forward a day at a time, applying five different windows (an hour, day, week, month and year). Finally, for budgetary reasons the deployment had to be swapped from AWS to Microsoft's Azure cloud platform. Within this Raphtory was deployed over 6 'E32s v3' virtual machines (32 vcpus, 256 GiB memory) with 4 allocated to Partition Managers. These machines were the closest available to the m5a.8xlarge used above, with similar CPU and network specifications, but double the available memory. Spark was then deployed via the Azure Databricks environment, with an underlying cluster of the same 6 machines. This was Databricks version 7.5 which includes Apache Spark 3.0.1 and Scala 2.12.

6.5.2 Spark Code

The implementation of this test within Spark can be found in Appendix D.1, however, it can be briefly summarised as follows: We first define a class for the user interactions in the Gab data including the timestamp, source and destination IDs. The raw data is then loaded from the Azure distributed file storage and parsed into this format. We then define the start time and end time of interest (Aug 10 2016 to May 03 2018 – 1470797917000 to 1525368897000) and the window sizes as stated above in milliseconds (3600000, 86400000, 604800000, 2592000000, 31536000000).

With these a 2 tier for-loop is established, performing the desired operation on all windows for every day within the period. This operation consists of filtering the raw interactions to within the timestamp and window (building the flattening), extracting the unique vertices, converting the remaining interactions to GraphX edge objects and building the final graph. We then call the *connectedComponents*() function on the graph (predefined in GraphX) and then extract the largest component via a *groupBy*(). Finally, when all windows for the day are complete the time taken is recorded.

6.5.3 Raphtory Implementation

For Raphtory's implementation the data was stored in a file on the machine hosting the Spout and read directly by Raphtory's default *FileSpout*[3]. This was then parsed by the *GabUserRouter* which can be seen in Appendix D.2. Here the IDs of the two interacting users are extracted from each line along with the time of interaction. This is then used to generate two Vertex Adds for the users and an Edge Add representing the interaction between them. Once the data is fully ingested a connected components analyser request is submitted to the Range Analysis API end-point, specifying the same start and end times and window sizes as the Spark code above. The implementation of this algorithm was discussed in Section 5.3.2 and can be found in Appendix C.1. The submitted query can be seen in Listing 6.1. Raphtory by default reports the time taken for each view, this was therefore used to compare to the output from the Spark code.

```
{"jobID":"connectedComponentsGabRange",
 "analyser":"Algorithms.ConnectedComponents",
 "start":1470797917000,"end":1525368897000,"increment":86400000,
 "windowType":"batched",
 "windowSet":[3600000,86400000,604800000,2592000000,31536000000]}
```

Listing 6.1: Query submitted to execute connected components on the Gab dataset.

6.5.4 Results

To initially make sure that the Spark algorithm was correct and producing the same output as Raphtory, both were tested locally on the first three months of the data. The execution times for this can be found on the left of Figure 6.11, with the individual times at the top and the cumulative times at the bottom. Note that the time taken by Raphtory to ingest the data is included in the cumulative plot to represent the full time taken to execute the job. Here we can see that Raphtory is across the board faster than Spark, ranging from 300x for the earlier points in time and 10x for the later points in time, leading to Raphtory completing the job in 85 seconds (15 for ingestion) whilst Spark took 32 minutes. This is because Raphtory can quickly

[3]https://github.com/Raphtory/Raphtory/blob/master/mainproject/src/main/scala/com/raphtory/spouts/FileSpout.scala

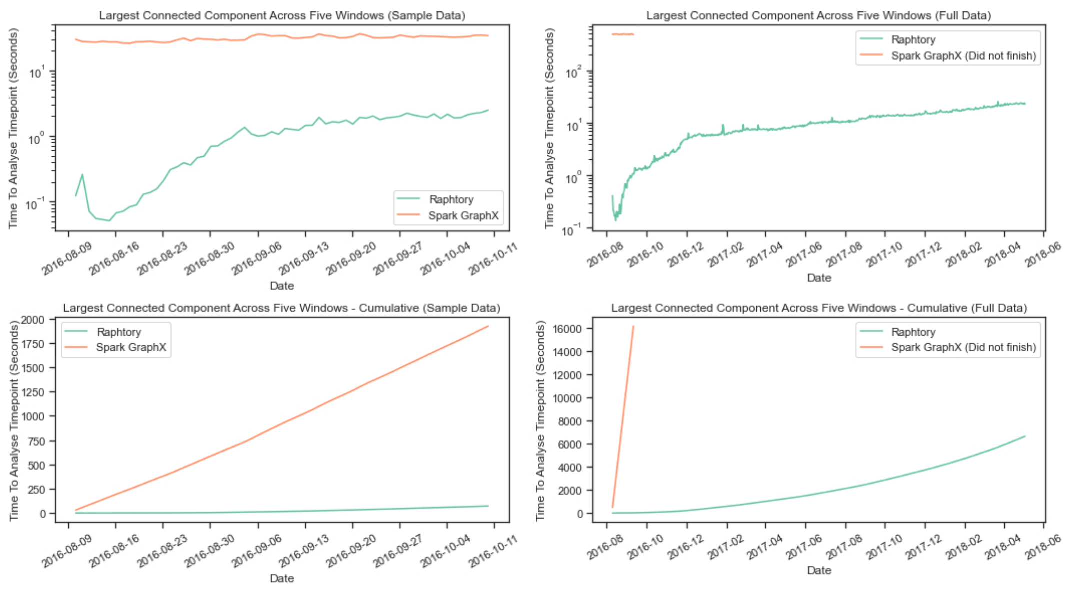


Figure 6.11: Comparison of time taken between Raphtory and Spark GraphX, performing the windowed connected components analysis of the Gab network discussed in Section 6.4.1. The test was conducted both locally on a sample dataset (left of figure) and then in a distributed setting on the full dataset (right of figure). Note the cumulative time for Raphtory includes the time for ingestion.

generate a flattening from the temporal model, with the processing time then proportional to the number of vertices included and messages propagated. Spark is somewhat similar in that the time to complete the connected components algorithm once the graph is built increases from the start of the sample period through to the end. However, this is dwarfed by the amount of time it takes to rebuild the graph from scratch for each window and timestamp, which averaged 90% of the total execution time for each day. This seemed quite extreme and warranted different Spark models to be trialled to see if an improvement could be garnered. Several changes were attempted, notably building a graph with all interactions and labeling them with the time of occurrence. This was intended to be then filtered down via the *subgraph*() function, but did not generate a noticeable improvement.

Once the systems were swapped into the defined Azure cluster and run on the full dataset this gap was only widened. The results for this can be found on the right of Figure 6.11 with the individual times at the top and the cumulative times at the bottom, again including the ingestion. Here we can see that Raphtory was able to complete the job in 110 minutes (3 for ingestion) with the time taken for each day increasing from 500ms at the beginning through to 25 seconds for the last flattenings. Spark on the other hand was taking ∼500 seconds for each day from the outset and was only increasing as it progressed through the workload. This was halted after four hours as at that pace it would take almost four and a half days to finish. The cause of this was exactly the same as the local deployment, whereby the time taken to extract

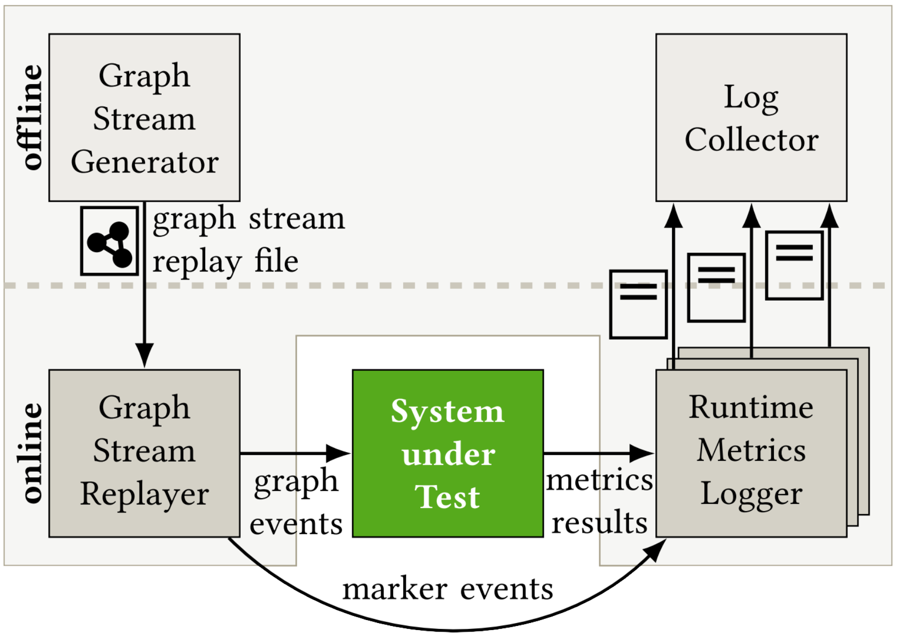


Figure 6.12: Conceptual overview of the GraphTides framework and test harness.

the graph from the millions of interactions was a huge proportion of the processing time for each day. This demonstrates the benefit garnered from Raphtory's temporal model and why in instances where we wish to drop the increments even lower (such as the hourly increments used in the real analysis of Gab), no other available tool would be appropriate.

6.6 GraphTides: A Framework for Evaluating Stream-based Graph Processing Platforms

As discussed in Section 6.2, when initially investigating the best manner in which Raphtory could be evaluated, similar stream-based graph processing systems were checked to see if a standard manner for evaluation had been established. Unfortunately this was not the case, with the algorithms, datasets and test environments varying widely. This was interesting as, whilst there are various benchmarking approaches for traditional batch-oriented graph processing systems, as well as pure streaming systems, there were no common procedures for evaluating stream-based graph systems. To remedy this, we developed GraphTides[1], a generic test harness and methodology to support system development and comparisons of relevant performance measurements; an overview of which can be seen in Figure 6.12.

GraphTides explores the intersecting parameter space between graph and stream benchmarks. It extracts standard computations from graph benchmarks such as LDBC Graphalytics[181] and integrates concepts of varying load/throughput from stream benchmarks such as StreamBench

[182]. Through these we defined the concept of a workload, which consists of one or more streams of data and computations to be run on the resulting graph. A stream in this context is defined by multiple compositional dimensions, notably: Topology changes, growth and/or decay of the graph over time; State changes, updating the properties of vertices and edges; and stream rate variability, consistent throughput vs inconsistent traffic spikes. As shown in the top of the figure, these streams are generated offline prior to testing, creating a replayable file. Once a system is under test this file may be output in a repeatable manner alongside marker events (watermarks) to track the true throughput of the system i.e. the time between the source outputting a tuple and this tuple being seen within the results. This setup also allows for the controlled manipulation of the stream, such as deterministic delay/dropping of updates, simulating real world environments to see how systems would react when deployed in 'production'; similar to the principles of chaos engineering[183]. By understanding what a stream consists of, and by providing replayability via a test harness, systems can be fairly compared across a variety of workloads.

To enable this comparison 'Runtime metric loggers' record output from the stream alongside metrics reported by all components of the system under test. These are then coalesced by a log collector to allow for automated plotting and comparison. To maximise the fairness when comparing these metrics, alongside a Popper[163] pipeline and Jain[162] compliant methodology, we also wanted to address the varying levels of access a evaluator may have of the systems under test. For this we define three evaluation levels and the comparisons which would be appropriate when taking these into consideration. This begins at level 0 where the system is considered a black box, it only provides an interface for ingesting the graph stream and for accessing computation results. In this instance only host system measurements may be made. Level 1 encompasses level 0 by exposing a native metrics interface. This additional source of information can be used to collect platform-internal data at runtime (e.g., current throughput, platform load). Finally at Level 2 a system will provide complete access to its internals and the evaluator is able to modify the source code to inject specific measurement logic. The evaluation goal and its execution, therefore, depend on the maximum level supported by the systems. For instance, the comparison of two systems in regard to their average CPU usage for a given workload is possible on level 0. In-depth performance comparison of internal scheduling components would require level 2.

Whilst it is not expected that GraphTides itself will be picked up as the explicit framework that all graph systems are compared with, it was important to bring these issues to light within the community in the hopes of being the catalyst for change in more established benchmarks. To this end we have had success in many of these ideas being adopted into the LDBC benchmarks[20], with the intention of also integrating temporal queries in the near future.

6.7 Summary

In summary, an initial methodology was taken forward to investigate the current ingestion and analysis capabilities of Raphtory. For the ingestion this included a scale-up throughput test, demonstrating that Partition Managers make good use of the resources allocated to them and may ingest over 100,000 updates a second, comfortably encapsulating many real world use cases. These results were then compared to the same dataset augmented with large proportions of worst case vertex deletions, demonstrating how detrimental these are, but that Raphtory can handle high network churn. Finally, a scale out test was performed to see how much of the Ethereum network could be stored across increasing number of graph partitions. Here Raphtory was able to almost double the size of the stored graph inline with the additional allocated resources, only hampered by the underlying data skew and the edge distribution this generated.

For analysis two use cases were explored. The first looked at applying batched windowed flattenings over the social network Gab.ai, extracting the difference between the same algorithms run with varying temporal depth. Within this Raphtory was able to extract interesting new insights, not only about Gab, but patterns not previously seen within online social networks. This was namely the continuous formation and collapse of a giant connected component following the diurnal cycle of user activity at the hour window scale. The second use case then performed taint analysis across the Ethereum network ingested in the prior tests, tracking the currency stolen during a high profile hack of a cryptocurrency exchange. Within this algorithm the history of edges was exploited to propagate the taint forward in time, looking for transactions with exchanges where the hacker had successfully converted the stolen currency into fiat. Within the 3 months following the hack, Raphtory was able to discover 78 exchanges which had accepted the illicit currency after it had been transferred through the hands of almost 200,000 wallets. During this process many interesting structural and temporal patterns were noted, as well as methodologies for extracting new insights from historic data sources. Alongside the continuation of the work above, these are being taken in a variety of exciting directions on a diverse array of datasets. This includes legal, insurance, Covid-19 and Urban Analytics to name a few. These are discussed in detail in Section 7.2.4.

Following this, when comparing how a user would implement the Gab analysis within more generic big data tooling such as Spark GraphX, Raphtory was shown to be over 300x more efficient at generating and analysing flattenings for the graph; taking 2 hours to complete a job that would take Spark 4 days on the same hardware. This was due to the large amount of preprocessing Spark had to do to build each flattening of the data, where as Raphtory could simply extract it from the temporal model.

Finally, based on what we have learnt from the evaluation of Raphtory, we created GraphTides, a framework/methodology to evaluate and compare stream-based graph processing systems. GraphTides establishes the concept of a graph stream workload, which can be replayed through its test harness in a repeatable manner to fairly compare systems under different conditions. It additionally introduces a three tier hierarchy to describe the level of internal access an evaluator has to the systems under test, discussing appropriate comparisons for each. This work has then fed into more established graph benchmarks as they have begun to provide their own stream based ecosystems.

Chapter 7

Conclusions and Future Work

Returning briefly to the initial motivation behind this work, it was discussed how the application of graph modelling finds an array of application across business sectors. Unfortunately, many of these applications are still being run periodically as offline batched analysis on stale data repositories, whilst the original source of the data continues to grow and evolve. To solve this a variety of dynamic graph processing solutions have been proposed, but only consider the most up-to-date version of the graph, failing to realise that by overwriting older property values and not maintaining the order of graph evolution many potential insights are lost. The temporal graph processing systems which attempt to tackle this are often offline, work on coarse snapshots (which lose temporal resolution), inject artificial event ordering and often do not natively support 'time-aware' graph algorithms where the history/update order is included.

In conjunction with this, graph processing faces a number of challenges which must be overcome[5] as they mature. Pertinent within this are issues of scalability, difficult ETL (extract/transform/load) pipelines to ingest the raw data into a graph, complex deployment (especially in distributed environments) and convoluted user APIs. To this end the goal of this work was to build a distributed dynamic temporal graph processing engine which could scale along with the ever increasing demands of modern datasets, providing intuitive ingestion/analysis APIs and innovative analytical functionality.

7.1 Summary of Contributions

Whilst there is still a way to go in many respects, the contributions within this work, culminating in the production of Raphtory, has seen this goal come to fruition. These contributions may be categorised under the four chapters which house them, namely: The distributed temporal graph model and stream semantics in Chapter 3; Raphtory's novel approach to ingestion and maintenance of the in-memory temporal graph in Chapter 4; The Raphtory analytical model in Chapter 5; and finally the GraphTides evaluation methodology, Raphtory's deployment infrastructure, together with the novel insights the platform produced in Chapter 6.

Beginning with the first of these, it was shown in Chapter 2 that a wide variety of graph models have been proposed and defined for both standard and temporal graphs. Within the latter of these the evolution towards temporal property graphs was explored, with each model possessing useful characteristics able to be expanded upon. However, none of these works defined the semantics for updating their temporal graph or reasoned about the model in a distributed context, off-setting this as purely an implementation detail. A dynamic temporal property graph was, therefore, defined within Chapter 3, additionally conceptualising a stream of events at discrete times, from which an equivalent graph could be garnered by ingesting all updates from it. The semantics for entity addition, deletion and property updates were then defined from the perspective of a temporal graph, where all updates provide additive information which become part of the graph history. Thirdly, the concept of a 'graph flattening' was introduced, discussing how the temporal graph could be viewed at any point within its history, returning a standard graph equivalent to one built from the updates ingested until the chosen point. The concept of stream windows was then additionally applied, defining a 'windowed flattening' where the resolved graph only contained entities updated within the set window period.

After contemplating the issues of graph distribution, focusing on partitioning, synchronisation and out of order messages, the model was redefined in a distributed context. This included a set of partitions overseeing the structural and property history for a portion of the graph entities, split in an edge-cut fashion. Similarly the stream semantics were defined in this new context, specifying streams between all partitions to facilitate synchronisation messaging. The constraints for updates were then relaxed and the manner in which updates could occur was brought in line with the desired distributed setting. These semantics allowed the graph to ingest updates in any order whilst still producing the same temporal graph, mitigating many of the issues faced in prior distributed system implementations.

Following on from the definition of the model, Chapter 4 discussed how the distributed temporal graph and the stream which feeds it could be materialised. Taking architectural direction from streaming systems such as Storm and Kineograph, the Spout was introduced as the representation of the stream source and the user's interface to ingest data. This was then paired with the Graph Routers which could scale along with the throughput of tuples ingested by the Spout, parsing these into graph updates via user defined functions. This decoupling of ingestion and modelling was important as it meant that the same data parsing could easily be executed across multiple data sources. This also allowed the same data source to be transformed into a number of different graph models by extracting different entities and relationships from the data. These models could return very different results for the same algorithms adding a new dimension to the analysis; one of the differentiators of graph analytics.

Once the desired extraction had been established, the updates were automatically routed to the Partition Manager responsible for the entities involved; handled via vertex hashing, which required no centralised organisation. These were the representation of the partition introduced in Chapter 3, splitting their responsibility between the Partition Writer for ingestion/synchronisation of updates and Partition Archivist for memory management. Similar to Routers, the Partition Managers could be scaled along with the size of the data, distributing the graph across any number of machines. Updates arriving at a Partition Writer were handled by a full implementation of the distributed stream semantics defined in Section 3.7. This meant that updates may arrive out of order and by following the the Writer would still be able to create the correct objects, update the relevant entity histories and synchronise with its peers to generate the same end state, utilising the event time of the update to establish the correct order. This was shown in Chapter 6 to scale well, both in terms of update throughput and graph capacity, encompassing many real world use cases.

Alongside ingestion a novel watermarking approach was implemented between Routers and Writers, tagging each message with a unique identifier. The UIDs for fully synchronised updates were then stored in a queue for their sending Router, generating a vector clock with which the Writer could calculate what time it considered safe for execution. By aggregating this time across all the partitions, Raphtory was able to establish a global 'live' time for the graph before which it was safe to analyse. Finally, the Partition Archivists worked in the background to snapshot the temporal graph, backing it up onto secondary storage, alongside monitoring the amount of memory used by each partition. If this memory exceeded a set threshold, Archivists across partitions coordinated to offload older history which occurred before an agreed point. This could be loaded back from storage if a user wished to perform analysis at a point prior to the cutoff.

In parallel with ingestion and maintenance, Raphtory may also run analytics on the graph at any point within its safe history. This is executed on graph flattenings, as defined in Chapter 3 and materialised via the graph lens. This returns a set of entity visitors to the user, which grant access to the structural and property history within the bounds of the flattening, whilst protecting the underlying data structures which may still be changing as new updates arrive. Algorithms executed on this set of entities are completed in a vertex centric manner and may span the range of structural scopes from local queries to global analytics. Across this range the algorithms may also be time-aware, making use of order, pace and duration within singular entities or whole graph patterns. These are then executed by the Partition Readers, which are in charge of analysis within each partition, orchestrated by the Analysis Manager whom the user submits queries to.

Algorithms are packaged as an analyser which may be run on any graph flattening. This means the algorithm only needs to be written once and can then be executed throughout the lifetime of the graph and with any window size. This includes ranges of time, generating flattenings at set increments, and with batches of windows, defining a variety of temporal depths; managed by the Raphtory REST API. This allows the user to see how a metric garnered from the algorithm differs across time by calculating deltas between results (as with snapshot based systems), allows the exploration of different temporal depths alongside, executed in window batches, and allows time-aware algorithms to be executed in the same manner, meaning the user may see how these also evolve and differ with depth, a combination not previously explored. With even the most basic of algorithms this approach was able to extract new insights from well analysed datasets, as demonstrated with the Gab social network and Ethereum blockchain in Chapter 6. As this is an online streaming system, all of these different analytical techniques may be used in combination on the Live Graph, returning ongoing results for the most recent state of the network. Comparing this implementation to a replication within Spark GraphX yielded a 300x speedup for Raphtory over the more generic big data framework, reducing multi day jobs to sub hour times and demonstrating the power of the temporal graph model.

Finally, in the evaluation of Raphtory it was discovered that whilst streaming systems and batched graph systems had clearly defined benchmarks, those which intersect the two (i.e. Raphtory) do not. As such the GraphTides evaluation framework was devised, consisting of a test harness and methodology for providing repeatable and comparable results. A notable element of this was the definition of a workload, which combined a stream context (speed, update consistency, spikes, delays, etc.) with the algorithm executing on top, as this obviously had a large impact on the resulting evaluation metrics. This methodology was developed alongside the evaluation of Raphtory, resulting in a fully containerised and monitored implementation which allowed repeatable tests and improved the usability of Raphtory when deployed in a distributed environment.

7.2 Future Work

Looking forward from this point, there are many expansions and improvements which may be made to Raphtory as it develops. Within this there are two main perspectives, one viewing the project from a software engineering standpoint, productising Raphtory and improving usability, and the other a wider research context, looking at the interesting directions this project has unearthed. Whilst the first of these is not covered in detail here, it is worth noting some examples which have made appearances throughout this work to get a flavour of what this will entail. For instance: adding an alerting system for delayed updates and the re-execution of flattenings which should have included them; adding backpressure between the Spouts, Routers and Partition Managers to stop components becoming overwhelmed.

Looking into the wider areas of research, there are four major directions which will be pursued next within the Raphtory project. The first of these involves revisiting the partitioning utilised within Raphtory to improve locality in a dynamic manner, whilst additionally leveraging the history available within the graph. The second will look at providing a concrete archiving strategy and answer questions on the scheduling of concurrent queries executing at times within and beyond the archiving cut off. The third will aim to expand the analytical API, adding interesting new functionality in both the generation of flattenings and state access. Finally, the fourth direction will be the exploration of new use cases and datasets, as well as digging deeper into those seen in Chapter 6.

7.2.1 Partitioning

As discussed in Section 4.4.1, Raphtory currently utilises a hash partitioning algorithm over an edge cut graph. There are, however, many interesting avenues to explore in improving this. An initial aspect would be to provide some sort of stream partitioning algorithm, as discussed in Vaquero et al[124], where new vertices are initially sent to a partition based upon a hash, but may then elect to move around partitions as they gain/lose edges with the growth of the graph. This provides a difficult engineering challenge as all Routers and Partition Managers need to know the location of vertices for updates, synchronisation and vertex messaging. Managing this in a centralised manner should clearly be avoided because of the issues it causes in systems such as Weaver[11]. One possible option here could be to have a local cache where the hash is initially relied upon, but if an actor receives a message for a vertex it no longer controls it may forward it on and inform the sender of its new location. This may, however, generate more traffic overall, so would need to be closely monitored, feeding back into the vertex movement heuristic.

Once the manner in which vertices can be safely re-partitioned has been established, the temporal element of the graph brings an additional layer of complexity. It is unclear if the history of a vertex, or frequency in which it interacts with certain neighbours, may be a help or hindrance in creating higher data locality. For example, should a vertex be placed near its most recent neighbours, those it has the most interactions with, or its oldest neighbours? The answer to this may depend drastically on the workload the user intends to run over the graph. For instance if there is only interest in running live analysis on the most recent version of the graph, this may have a very different partitioning strategy compared to an aim to build flattenings at small increments throughout the lifetime of the graph. The structural scope of user queries would also have a large effect in this regard. A sensible approach to this may be to allow the user to provide some level of information about their desired outcomes which could be fed into the re-partitioning process. In a similar vein, information about the underlying dataset could be included here which can have a large impact. The final expansion to this, of course, is that Raphtory could learn and adapt as more queries are submitted, finding the best balance between the number of updates coming in, which entities are predominantly affected and what analysis is run in parallel.

An interesting solution in this regard could come in the parallel deployment of multiple sets of Partition Managers, each ingesting the same data, but storing the graph with different partitioning strategies. If there are many parallel queries being submitted across a range of times and structural scopes, this could garner a speed up as they are no longer contending for resources and could be directed to the partition set with the most appropriate partitioning strategy. This would, however, have the obvious caveat of a lot more compute power being utilised, so would need to be justified on a case by case basis.

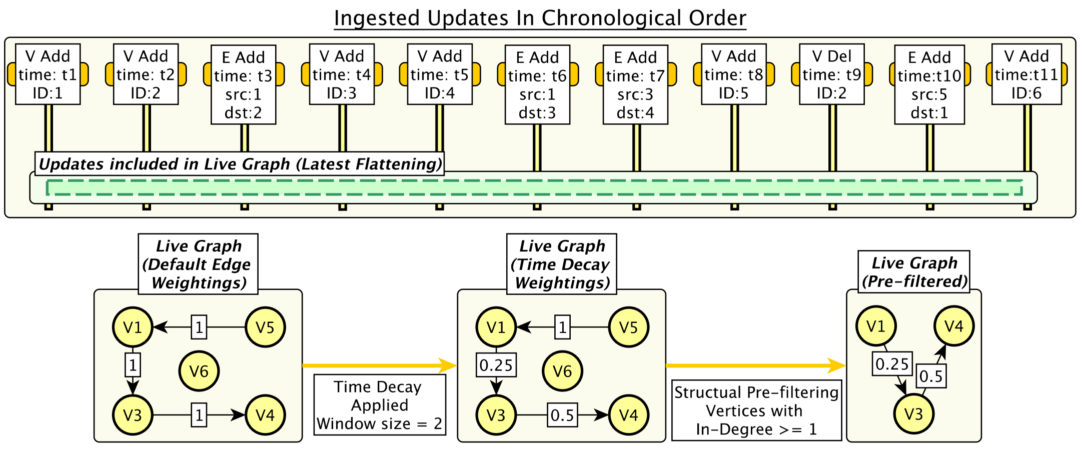
7.2.2 Archiving

As initially covered in Section 4.8, memory management within Raphtory is controlled by the Archivist which tracks the used memory on a partition's host machine, removing older history if this surpasses a set threshold. There are still, however, many open questions surrounding this process. Firstly, for a concrete implementation, it must be decided which back-end datastore to use. Several have previously been trialled, notably MongoDB[184] and Cassandra [185], with the former unable to handle the throughput and the latter not quite fitting as snapshot storage. The next direction with this will be to try Apache Hive [186], a database wrapper for the HDFS which converts SQL like queries into MapReduce jobs. This should be able to scale along with the number of partitions and provide high availability and fault tolerance as standard. This could also reduce the complexity of establishing Raphtory in a new cluster, as most have HDFS, as well as exposing the snapshots to more standard big data platforms, such as Hadoop and Spark, for external offline analysis. Such a setup could also facilitate the batch ingestion of the latest state of the graph into a new larger set of partitions if the current deployment has run out of memory (even with archiving).

Secondly, the thresholds for when older history is removed, and the portions which are removed in each archiving cycle, must be established. Alongside this there must be a full investigation into the possible interleavings which may occur when deleting history/objects from memory, i.e. one partition believes an edge to exist, but the other no longer does so. This could break the current semantics for updating and would, therefore, require an additional set of post archiving update steps or strict global synchronisation on what history is considered available. These both have possible downsides, as the first may require considerably more synchronisation messages between workers and the latter queries on the history of an entity before deciding how to proceed with an update.

A third direction, not mentioned above, but explored in the original Raphtory paper[19], is looking for a manner to compress the older history/entities which still leaves them in a readable state, but reduces the memory footprint. In prior attempts, the structural/property history of each entity was reduced by pruning all consecutive updates of the same type (past a cutoff established by the watermarking) keeping only the oldest instance. This meant entities had a 'writable' portion of history, where new updates could still be inserted, and an older 'read-only' compressed history, which would take less space, but could not be changed. However, this brought numerous issues. For instance, breaking the manner in which windows are generated, losing information for certain properties where duplicates mattered and opening the ambiguity of what to do if a delayed update arrived past the compression cutoff. As such it was disabled, but remains a clear line of enquiry to alleviate the issues of archiving and allow Raphtory to ingest larger graphs. This is somewhat orthogonal to the engineering issue of reducing the memory footprint of all entities and updates, but obviously shares the same goal.

The final area of query for archiving revolves around scheduling and priority. There are several layers to this. Firstly on how the resources of a Partition Manager should be shared between the



Ingested Updates In Chronological Order
V Add time: t1 ID:1
V Add time: t2 ID:2
E Add time: t3 src:1 dst:2
V Add time: t4 ID:3
V Add time: t5 ID:4
E Add time: t6 src:1 dst:3
E Add time: t7 src:3 dst:4
V Add time: t8 ID:5
V Del time: t9 ID:2
E Add time:t10 src:5 dst:1
V Add time:t11 ID:6
Updates included in Live Graph (Latest Flattening)
Live Graph (Default Edge Weightings)
Time Decay Applied Window size = 2
Live Graph (Time Decay Weightings)
Structual Pre-filtering Vertices with In-Degree >= 1
Live Graph (Pre-filtered)
V1
V3
V4
V5
V6
1
0.25
0.5

Figure 7.1: Top - Example update stream and Live Graph flattening, originally seen within Figure 5.2. Bottom - Time decay and prefiltering of the Live Graph flattening.

Writer, Reader and Archivist when all three are active. In this instance, it is unclear if it is more important to let the Archivist offload older history, the Writer to maintain throughput or the current analysis to finish running. This may depend on a number of factors, some automated and some user defined. Secondly, in a similarly vein, if there are two or more queries submitted in parallel looking at opposite extremes of the history within the graph, which is given priority? In the worst case the Archivist may be required to load back a large amount of history for one query, which could even push out the required range of the other. There is currently no clear answer to these questions and the development of some heuristics may be required to attempt to balance CPU allocation and intelligently order flattening supersteps.

7.2.3 Analytics Improvements

We are only just scratching the surface with the types of analysis which may be performed over the temporal graph model established in Raphtory and, as such, there are many directions the API may be taken in. Three interesting lines of enquiry which do stand out, however, are: the materialisation of 'complex flattenings'; providing entities with a safe way to access information from prior flattenings; and the creation and distribution of a global graph state.

Complex Flattenings

Whilst the flattenings of the graph discussed in Section 3.4 focus purely on the time of updates, there is also space to explore more complex preprocessing of the entities prior to the application of the chosen algorithm. One option here is a softer form of windowing known as time decay[187]. In this instance, rather than pushing old entities out of the graph, they are allocated an 'importance' score within the flattening based upon their last update time and the chosen window size. A variety of decay algorithms exist for this, a popular choice being exponential, where the importance is halved for each window of time back required to reach the latest update. The importance of an entity may then be exposed to the user, allowing them to integrate this into their algorithm in whichever way they see fit. This has many applications. For example, within a social network this could allow newer connections to contribute to the majority of a user's ranking (by multiplying each connections vote by its importance), but still permit older connections to have some effect.

As an example, taking the stream of updates at the top of Figure 7.1 and a simple node ranking algorithm where one incoming edge equals one vote, the bottom of Figure 7.1 explores the affect of time decay. Looking at the first graph with default edge rankings, a three way tie can be seen between vertex 1, 3 and 4, but by applying a time decay with a window size of 2, prioritising very recent edges, a clear ranking can be established. Within this vertex 1 leads as its edge $5 \longrightarrow 1$ exists in the first window and, therefore, its score remains untouched. Vertex 4 is ranked second as its edge $3 \longrightarrow 4$ is one window back and, therefore, has had its value halved. Finally Vertex 3 is in third as its edge $1 \longrightarrow 3$ is two windows back and has had its value halved twice.

Following this, given the expressiveness of the property graph model and the size of the graphs being operated on, there may be instances where the user is not interested in the whole graph at a given point in time, but a subgraph containing vertices and edges with certain characteristics. A second more broader type of flattening preprocessing may, therefore, be envisaged, selectively filtering entities based on their structural and property values. This could cover everything within the scope of a local query, meaning the user may choose vertices and edges of a given type (or set of types), select those with properties valued at, above or below thresholds, or base selection on local structural values such as in/out degree. The results of prior algorithms could also be utilised here. This obviously would require an additional input from the user, but once a filter is established it could be applied to flattenings at any point in time and with any window size. As an example of this process, the third flattening in the bottom of Figure 7.1 shows the Live Graph after it has been filtered to remove entities which have no in-coming edges, i.e. nodes which have no rank according to the algorithm. This drops vertex 5 and 6 (as well as their edges) and means that vertex 1 now falls to the bottom of the rankings as its votes were only from nodes without any capital (i.e. bots in a social network).

Aggregate Windows and Prior Flattening Values

Currently an entity may look at its history (within the bounds of the flattening) and calculate its presence within smaller windows. For example, for a flattening with a window size of a year, edges may calculate how many of the days within that year they were active, possibly feeding into an investigation of the permanence of relationships in a local neighbourhood or across the network as a whole. However, there is currently no way to feed this information into future analysis or allow it to be done across the window flattenings submitted by the user. As such the analyser API could be expanded to allow entities access to information about multiple window flattenings or prior flattening state. In this instance entities would be able to ask questions such as 'did I exist in the prior flattening?', 'how many of the total flattenings have I been involved in?' as well as for vertices, ascertaining their final state from the execution of prior algorithms.

This opens many possible opportunities for new forms of composite analysis, speeding up convergence of subsequent flattenings as well as basing entity actions on the deltas between current and prior states (such as raising alerts for an anomaly). There are, however, many practical and semantic challenges here, predominantly surrounding the breaking of the current isolation that flattenings have. This means the results of algorithms using this part of the API require some number of prior flattenings to have run and, therefore, the end output may depend on the start and end times of a range, along with a host of other factors. This would obviously be worse if the API allowed access to the results of other analysis tasks/running algorithms, which may have differing start/end times, window size and increment. A manner in which this could be mediated is by formalising a set of queries into a DAG (similar to Flink) specifying run order. There may, however, be large webs of dependencies here, confusing to the user to establish and difficult to schedule for the Partition Managers. As such this is an open problem.

Global State

A final area where the API could be improved is to provide access to a global state/aggregate. This is important in the application of many algorithms, for example in the normalisation of PageRank where each score is divided by the total number of vertices. This could, however, be additionally used, in combination with the ideas of flattening aggregates above, to expand the analysis possible within Raphtory. An example of this would be requesting the average number of vertices/edges over n prior flattenings to see how the current flattening compares. Alternatively, compare the global median consecutive number of windows which edges appear in to the same metric for an individual edge, to gather a better idea of its relative permanence in the network.

Within the current implementation this would require the aggregation to take place in the Analysis Task, followed by a broadcast back to the Reader Workers. However, each superstep would require a reduction at the end of it, which means a larger amount of networking and a clear possible bottleneck. This must be managed in a way which mitigates this, possibly by a partial aggregation on the partition side, all of which would require defining by the user. This would additionally require a reworking of the current analyser API, where the user would now have to specify if the global value only needed to be calculated in the setup or, if not, the value to be returned from each execution of the analyse function.

7.2.4 Future Use Cases

The final area of future research currently being looked at within the Raphtory project is the expansion of use cases for which the tool may be used, together with digging further into those started in Chapter 6. Beginning with those previously seen, investigations may be carried out in other social networks similar to Gab, utilising the full range of analytical capabilities within Raphtory, not just windowing. The result of these may then be compared to extract which temporal patterns are generalisable across social networks and which are unique to each ecosystem. Secondly, the initial investigation into Ethereum was just the beginning of our delve into blockchain/cryptocurrency datasets, where temporal graph analysis is gaining traction[188]. Within this there is a large push to improve the algorithms for identifying bad actors to assist in the fight to legitimise the sector. Our next direction in this space will be developing algorithms to extract semantically significant temporal motifs and cluster wallets based on how often these patterns appear in their history. This will be done with the intention of improving the automated detection of wallets associated with markets, both dark and legitimate, and 'mixers', services which try to obscure the activities of their customers.

Raphtory is also being applied in a multitude of new use cases, both by the core team as well as researchers which have found the tool a perfect addition to their repertoire. For instance, the techniques used to analyse Gab are being applied in a legal context, looking at the patterns within 400 years of citations between court judgements. Within many legal systems, but especially English common law, court cases will cite prior decisions to provide evidence for why a given verdict has been made. This creates a long running temporal network that could provide insight into how the impact of legal cases grow and shrink through time, as well as predicting the importance of recent judgements and their possible effects on the future.

In a totally different direction, there is ongoing work applying Raphtory in the urban analytics space where the movement of people within a city, across time and on different forms of transport, may be modelled as a typed geospatial temporal network. Through probing this we may be able to better understand issues within the city, denote clear areas of congestion and see the positive/negative effects of government projects (such as the pedestrianisation of a street) by viewing the footfall before and after the change point. In a similar nature, there is clear scope to expand this work in the realm of contact tracing for Covid-19, which naturally fits a graph model and where the time, order and duration of interactions are paramount for accurate predictions of spreading. Finally, these ideas are in parallel being taken forward in the insurance sector for supply-chains and tracked goods. Within this domain we are looking at extracting patterns leading up to fraudulent claims with the aim of predicting if/when an insured entity is going to attempt to defraud the insurer.

Appendices

Appendix A

Example Spouts

A.1 Postgres Spout

```
class EthereumPostgresSpout extends SpoutTrait {
  var startBlock = System.getenv().getOrDefault("STARTING_BLOCK", "46147").trim.toInt
      //first block to have a transaction by default
  val batchSize = System.getenv().getOrDefault("BLOCK_BATCH_SIZE", "100").trim.toInt
      //number of blocks to pull each query
  val maxblock  = System.getenv().getOrDefault("MAX_BLOCK", "8828337").trim.toInt
      //Maximum block in database to stop querying once this is reached

  val dbURL      = System.getenv().getOrDefault("DB_URL", "jdbc:postgresql:ether").trim
      //db connection string, default is for local with db called ether
  val dbUSER     = System.getenv().getOrDefault("DB_USER", "postgres").trim      //db
      user defaults to postgres
  val dbPASSWORD = System.getenv().getOrDefault("DB_PASSWORD", "").trim
      //default no password

  // querying done with doobie wrapper for JDBC (https://tpolecat.github.io/doobie/)
  implicit val cs = IO.contextShift(ExecutionContexts.synchronous)
  val dbconnector = Transactor.fromDriverManager[IO](
         "org.postgresql.Driver",
         dbURL,
         dbUSER,
         dbPASSWORD,
         Blocker.liftExecutionContext(ExecutionContexts.synchronous)
  )

  override def ProcessSpoutTask(message: Any): Unit = message match {
    case StartSpout => AllocateSpoutTask(Duration(1, MILLISECONDS), "nextBatch")
    case "nextBatch" => running()
    case _          => println("message not recognized!")
  }

  protected def running(): Unit = {
    sql"select from_address, to_address, value,block_timestamp from transactions where
        block_number >= $startBlock AND block_number < ${startBlock + batchSize} "
      .query[
            (String, String, String, String)
      ]                                  //get the to,from,value and time for
        transactions within the set block batch
      .to[List]                          // ConnectionIO[List[String]]
      .transact(dbconnector)             // IO[List[String]]
      .unsafeRunSync                     // List[String]
      .foreach(x => sendTuple(x.toString())) //send each transaction to the routers

    startBlock += batchSize                              //increment batch for the next
        query
    if (startBlock > maxblock) stop()                    //if we have reached the max
        block we stop querying the database
    AllocateSpoutTask(Duration(1, MILLISECONDS), "nextBatch") // line up the next batch
  }
}
```

A.2 Ethereum Node Spout

```
class EthereumNodeSpout extends SpoutTrait {

  var currentBlock = System.getenv().getOrDefault("SPOUT_ETHEREUM_START_BLOCK_INDEX",
      "9014194").trim.toInt
  var highestBlock = System.getenv().getOrDefault("SPOUT_ETHEREUM_MAXIMUM_BLOCK_INDEX",
      "10026447").trim.toInt
  val nodeIP      = System.getenv().getOrDefault("SPOUT_ETHEREUM_IP_ADDRESS",
      "127.0.0.1").trim
  val nodePort    = System.getenv().getOrDefault("SPOUT_ETHEREUM_PORT", "8545").trim
  val baseRequest = requestBuilder()

  implicit val materializer = ActorMaterializer()
  implicit val EthFormat = jsonFormat14(EthResult)
  implicit val EthTransactionFormat = jsonFormat3(EthTransaction)

  override protected def ProcessSpoutTask(message: Any): Unit = message match {
    case StartSpout => pullNextBlock()
    case "nextBlock" => pullNextBlock()
  }

  def pullNextBlock(): Unit = {
    if (currentBlock > highestBlock)
      return
    try {
      log.debug(s"Trying block $currentBlock")
      val transactionCountHex = executeRequest("eth_getBlockTransactionCountByNumber",
          "\"0x" + currentBlock.toHexString + "\"");
      val transactionCount = Integer.parseInt(transactionCountHex.fields("result")
          .toString().drop(3).dropRight(1), 16)
      if(transactionCount>0){
        var transactions = "["
        for (i <- 0 until transactionCount)
          transactions = transactions +
              batchRequestBuilder("eth_getTransactionByBlockNumberAndIndex",
                        s"0x${currentBlock.toHexString}","0x${i.toHexString}")+","
        val trasnactionBlock = executeBatchRequest(transactions.dropRight(1)+"]")
        val transList = trasnactionBlock.parseJson.convertTo[List[EthTransaction]]
        transList.foreach(t => {
          try{sendTuple(s"${t.result.blockNumber.get},
              ${t.result.from.get},${t.result.to.get},${t.result.value.get}")}
          catch {case e:NoSuchElementException =>}
        })
      }
      currentBlock += 1
      AllocateSpoutTask(Duration(1, NANOSECONDS), "nextBlock")
    } catch {
      case e: NumberFormatException => AllocateSpoutTask(Duration(1, SECONDS),
          "nextBlock")
      case e: Exception           => e.printStackTrace(); AllocateSpoutTask(Duration(1,
          SECONDS), "nextBlock")
    }
  }

  def batchRequestBuilder(command:String,params:String):String = s"""{"jsonrpc": "2.0",
      "id":"100", "method": "$command", "params": [$params]}"""
  def executeBatchRequest(data: String) = requestBatch(data).execute().body.toString
  def requestBatch(data: String): HttpRequest = baseRequest.postData(data)
  def requestBuilder() =
    if (nodeIP.matches(Utils.IPRegex))
      Http("http://" + nodeIP + ":" + nodePort).header("content-type",
          "application/json")
    else
      Http("http://" + hostname2Ip(nodeIP) + ":" + nodePort).header("content-type",
          "application/json")

  def request(command: String, params: String = ""): HttpRequest =
    baseRequest.postData(s"""{"jsonrpc": "2.0", "id":"100", "method": "$command",
        "params": [$params]}""")

  def executeRequest(command: String, params: String = "") = request(command,
      params).execute().body.toString.parseJson.asJsObject

  def hostname2Ip(hostname: String): String =
      InetAddress.getByName(hostname).getHostAddress()

}
```

Appendix B

Example Routers

B.1 Ethereum Router

```
class EthereumRouter(override val routerId: Int,override val workerID:Int, val
    initialManagerCount: Int) extends RouterWorker {

  def hexToInt(hex: String) = Integer.parseInt(hex.drop(2), 16)

  override protected def parseTuple(value: Any): Unit = {
    print(value)
    val transaction = value.toString.split(",")
    val blockNumber = hexToInt(transaction(0))

    val from = transaction(1).replaceAll("\"", "").toLowerCase
    val to  = transaction(2).replaceAll("\"", "").toLowerCase
    val sent = transaction(3).replaceAll("\"", "")
    val sourceNode    = assignID(from) //hash the id to get a vertex ID
    val destinationNode = assignID(to) //hash the id to get a vertex ID

    sendGraphUpdate(
           VertexAddWithProperties(blockNumber, sourceNode, properties =
               Properties(ImmutableProperty("id", from)))
    )
    sendGraphUpdate(
           VertexAddWithProperties(blockNumber, destinationNode, properties =
               Properties(ImmutableProperty("id", to)))
    )
    sendGraphUpdate(
           EdgeAddWithProperties(
                    blockNumber,
                    sourceNode,
                    destinationNode,
                    properties = Properties(StringProperty("value", sent))
            )
    )
  }
}
```

Appendix C

Example Analysers

C.1 Connected Components

```
class ConnectedComponents(args:Array[String]) extends Analyser(args){

  override def setup(): Unit =
    view.getVertices().foreach { vertex =>
      vertex.setState("cclabel", vertex.ID)
      vertex.messageAllNeighbours(vertex.ID)
    }

  override def analyse(): Unit =
    view.getMessagedVertices().foreach { vertex =>
      val label = vertex.messageQueue[Long].min
      if (label < vertex.getState[Long]("cclabel")) {
        vertex.setState("cclabel", label)
        vertex messageAllNeighbours label
      }
      else
        vertex.voteToHalt()
    }

  override def returnResults(): Any =
    view.getVertices()
      .map(vertex => vertex.getState[Long]("cclabel"))
      .groupBy(f => f)
      .map(f => (f._1, f._2.size))
  override def processResults(results: ArrayBuffer[Any], timestamp: Long,
      viewCompleteTime: Long): Unit = {
    val er = extractData(results)
    val text = s"{"time":$timestamp,"top5":[${er.top5.mkString(",")}]",...
    publishData(text)
  }

  override def processWindowResults(results: ArrayBuffer[Any], timestamp: Long,
      windowSize: Long, viewCompleteTime: Long): Unit = {
      val er = extractData(results)
      var output_folder = System.getenv().getOrDefault("OUTPUT_FOLDER", "/app").trim
      var output_file = output_folder + "/" +
          System.getenv().getOrDefault("OUTPUT_FILE","ConnectedComponents.json").trim
      val text = s"{"time":$timestamp,"windowsize":$windowSize,"...
      publishData(text)
  }

  def extractData(results:ArrayBuffer[Any]):extractedData ={
    val endResults = results.asInstanceOf[ArrayBuffer[immutable.ParHashMap[Long, Int]]]
    try {
      val grouped = endResults.flatten.groupBy(f => f._1).mapValues(x => x.map(_._2).sum)
      val groupedNonIslands = grouped.filter(x => x._2 > 1)
      val biggest = grouped.maxBy(_._2)._2
      val sorted = groupedNonIslands.toArray.sortBy(_._2)(sortOrdering).map(x=>x._2)
      val top5 = if(sorted.length<=5) sorted else sorted.take(5)
      val total = grouped.size
      val totalWithoutIslands = groupedNonIslands.size
      val totalIslands = total - totalWithoutIslands
      val proportion = biggest.toFloat / grouped.map(x => x._2).sum
      val totalGT2 = grouped.filter(x => x._2 > 2).size
      extractedData(top5,total,totalIslands,proportion,totalGT2)
    } catch {
      case e: UnsupportedOperationException => extractedData(Array(0),0,0,0,0)
    }
  }
    override def defineMaxSteps(): Int = 100
}
```

C.2 Degree Ranking

```
class DegreeRanking(args:Array[String]) extends Analyser(args){

  val weighted = false
  override def analyse(): Unit = {}
  override def setup(): Unit = {}
  override def returnResults(): Any = {
    val degree =
      if (weighted) {
      view.getVertices().map { vertex =>
      val outDegree = vertex.getOutEdges.map{e => e.getHistory().size}.sum
      val inDegree = vertex.getIncEdges.map{e => e.getHistory().size}.sum
      (vertex.ID, outDegree, inDegree)}}
    else {
      view.getVertices().map { vertex =>
        val outDegree = vertex.getOutEdges.size
        val inDegree = vertex.getIncEdges.size
        (vertex.ID, outDegree, inDegree)}
    }

    val totalV  = degree.size
    val totalOut = degree.map(x => x._2).sum
    val totalIn = degree.map(x => x._3).sum
    val topUsers = degree.toArray.sortBy(x => x._3)(sortOrdering).take(20)
    (totalV, totalOut, totalIn, topUsers)
  }

  override def defineMaxSteps(): Int = 1

  override def processResults(results: ArrayBuffer[Any], timeStamp: Long,
      viewCompleteTime: Long): Unit = {
    val endResults = results.asInstanceOf[ArrayBuffer[(Int, Int, Int, Array[(Int, Int,
        Int)])]]
    val totalVert  = endResults.map(x => x._1).sum
    val totalEdge  = endResults.map(x => x._3).sum

    val degree =
      try totalEdge.toDouble / totalVert.toDouble
      catch { case e: ArithmeticException => 0 }
    var bestUserArray = "["
    val bestUsers = endResults
      .map(x => x._4)
      .flatten
      .sortBy(x => x._3)(sortOrdering)
      .take(20)
      .map(x => s"""{"id":${x._1},"indegree":${x._3},"outdegree":${x._2}}""")
      .foreach(x => bestUserArray += x + ",")
    bestUserArray = if (bestUserArray.length > 1) bestUserArray.dropRight(1) + "]" else
        bestUserArray + "]"
    val text =
      s"{"time":$timeStamp,"vertices":$totalVert,"edges":$totalEdge,
      "degree":$degree,"bestusers":$bestUserArray,"...
    publishData(text)
  }

  override def processWindowResults(results: ArrayBuffer[Any],timestamp:
       Long,windowSize: Long,viewCompleteTime: Long): Unit = {
    val endResults = results.asInstanceOf[ArrayBuffer[(Int, Int, Int, Array[(Int, Int,
        Int)])]]
    val startTime  = System.currentTimeMillis()
    val totalVert  = endResults.map(x => x._1).sum
    val totalEdge  = endResults.map(x => x._3).sum

    val degree =
      try totalEdge.toDouble / totalVert.toDouble
      catch { case e: ArithmeticException => 0 }
    var bestUserArray = "["
    val bestUsers = endResults
      .map(x => x._4)
      .flatten
      .sortBy(x => x._3)(sortOrdering)
      .take(20)
      .map(x => s"""{"id":${x._1},"indegree":${x._3},"outdegree":${x._2}}""")
      .foreach(x => bestUserArray += x + ",")
    bestUserArray = if (bestUserArray.length > 1) bestUserArray.dropRight(1) + "]" else
        bestUserArray + "]"
    val text =
      s"{"time":$timestamp,"windowsize":$windowSize,"vertices":$totalVert,
      "edges":$totalEdge,"degree":$degree,"bestusers":$bestUserArray,"...
    publishData(text)
  }
}
```

C.3 Temporal Triangle Count

```
class TemporalTriangleCount(args:Array[String]) extends Analyser(args) {

  override def setup(): Unit =
    view.getVertices().foreach { vertex =>
      val edgeTimes = vertex.getIncEdges.map(edge => edge.latestActivity()._1)
      val t_max = if(edgeTimes.nonEmpty) edgeTimes.max else -1 //get incoming edgeTimes
          and then find the most recent edge with respect to timestamp and window
      vertex.getOutEdgesBefore(t_max).foreach(neighbour => {
        neighbour.send((Array(vertex.ID()),t_max))
      })
    }

  override def analyse(): Unit =
    view.getMessagedVertices().foreach { vertex =>
      val queue = vertex.messageQueue[(Array[Long], Long)]
      queue.foreach(message=> {
        val path = message._1
        val sender = path(path.length-1)
        val t_max = message._2
        val t_min = vertex.getInEdge(sender).get.earliestActivity()._1 //to include
            deletions check
        if(path.length<2) { //for step two of the algorithm i.e. the second node in the
            triangle
          vertex.getOutEdgesBetween(t_min, t_max).foreach(neighbour => {
            neighbour.send((message._1 ++ Array(vertex.ID()), t_max))
          })
        }
        else{ //for the 3rd node in the triangle to see if the final edge exists
          val source = path(0)
          vertex.getOutEdgeBetween(source,t_min,t_max) match {
            case Some(edge) =>
              vertex.appendToState("TrianglePath",(path ++
                  Array(vertex.ID())).mkString("[",", ","]"))
            case None =>
          }
        }
      })
    }

  override def returnResults(): Any =
    view.getVertices()
      .filter(vertex=>
        vertex.containsState("TrianglePath"))
      .flatMap(vertex =>
        vertex.getState[Array[String]]("TrianglePath")).to



  override def processResults(results: ArrayBuffer[Any], timeStamp: Long,
      viewCompleteTime: Long): Unit = {
    val endResults = results.asInstanceOf[ArrayBuffer[ParArray[String]]].flatten.toArray
    publishData(s"""{"timestamp":$timeStamp,triangles:[""" +endResults.map(triangle =>
        triangle+",").fold("")(_+_).dropRight(1)+"]}")
  }

  override def defineMaxSteps(): Int = 2
}
```

C.4 PageRank

```
class PageRank(args:Array[String]) extends Analyser(args) {
  val d = 0.85 // damping factor
  override def setup(): Unit =
    view.getVertices().foreach { vertex =>
      val outEdges = vertex.getOutEdges
      val outDegree = outEdges.size
      if (outDegree > 0) {
        val toSend = 1.0/outDegree
        vertex.setState("prlabel",toSend)
        outEdges.foreach(edge => {
          edge.send(toSend)
        })
      } else {
        vertex.setState("prlabel",0.0)
      }
    }

  override def analyse(): Unit =
    view.getMessagedVertices().foreach {vertex =>
      val currentLabel = vertex.getState[Double]("prlabel")
      val newLabel = 1 - d + d * vertex.messageQueue[Double].sum
      vertex.setState("prlabel",newLabel)
      if (Math.abs(newLabel-currentLabel)/currentLabel > 0.01) {
        val outEdges = vertex.getOutEdges
        val outDegree = outEdges.size
        if (outDegree > 0) {
          val toSend = newLabel/outDegree
          outEdges.foreach(edge => {
            edge.send(toSend)
          })
        }
      }
      else {
        vertex.voteToHalt()
      }
    }

  override def returnResults(): Any = {
    val pageRankings = view.getVertices().map { vertex =>
      val pr = vertex.getState[Double]("prlabel")
      (vertex.ID, pr)
    }
    val totalV = pageRankings.size
    val topUsers = pageRankings.toArray.sortBy(x => x._2)(sortOrdering).take(10)
    (totalV, topUsers)
  }

  override def defineMaxSteps(): Int = 20

  override def processResults(results: ArrayBuffer[Any], timeStamp: Long,
      viewCompleteTime: Long): Unit = {
    val endResults = results.asInstanceOf[ArrayBuffer[(Int, Array[(Long,Double)])]]
    val totalVert = endResults.map(x => x._1).sum
    val bestUsers = endResults
      .map(x => x._2)
      .flatten
      .sortBy(x => x._2)(sortOrdering)
      .take(10)
      .map(x => s"""{"id":${x._1},"pagerank":${x._2}}""").mkString("[",",","]")
    val text = s"""{"time":$timeStamp,"vertices":$totalVert,
    "bestusers":$bestUsers,"viewTime":$viewCompleteTime}"""
    publishData(text)
  }

  override def processWindowResults(results: ArrayBuffer[Any], timestamp: Long,
      windowSize: Long, viewCompleteTime: Long ):
  Unit = {
        val endResults = results.asInstanceOf[ArrayBuffer[(Int, Array[(Long,Double)])]]
    val totalVert = endResults.map(x => x._1).sum
    val bestUsers = endResults
      .map(x => x._2)
      .flatten
      .sortBy(x => x._2)(sortOrdering)
      .take(10)
      .map(x => s"""{"id":${x._1},"pagerank":${x._2}}""").mkString("[",",","]")
    val text =
      s"""{"time":$timestamp,"windowsize":$windowSize,"vertices":$totalVert,
      "bestusers":$bestUsers,"viewTime":$viewCompleteTime}"""
   publishData(text)
  }
}
```

C.5 Temporal Contagion (Ethereum)

```
class TaintTrackExchangeStop(args:Array[String]) extends Analyser(args) {

   val infectedNode = args(0).trim.toLowerCase
  val infectionStartingBlock = args(1).trim.toLong
  val listOfExchanges = Array("0x83053C32b7819F420dcFed2D218335fe430Fe3b5",
  "0x05f51aab068caa6ab7eeb672f88c180f67f17ec7",...)
    .map(x=>x.toLowerCase())

  override def setup(): Unit = view.getVertices().foreach { vertex =>
      val walletID = vertex.getPropertyValue("id").get.asInstanceOf[String]
      if(walletID contains infectedNode) {
        vertex.getOrSetState("infected", infectionStartingBlock)
        vertex.getOrSetState("infectedBy", "Start")
        vertex.getOutEdgesAfter(infectionStartingBlock).foreach { neighbour =>
          neighbour.send((walletID,neighbour.firstActivityAfter
               (infectionStartingBlock),vertex.ID()))
        }
      }
    }

  override def analyse(): Unit =
    view.getMessagedVertices().foreach { vertex =>
      try{
      var infectionBlock = infectionStartingBlock
      var infector = infectedNode
      val queue = vertex.messageQueue[(String,Long,Long)]
      infectionBlock = queue.map(x=>x._2).min
      val tuple = queue.filter(x=>x._2==infectionBlock).head
      infector = tuple._1
      val walletID = vertex.getPropertyValue("id").get.asInstanceOf[String]
      if(vertex.containsState("infected")){
        if(infectionBlock<vertex.getState[Long]("infected")){
          if(listOfExchanges contains walletID ){
            vertex.setState("exchangeHit", true)
            vertex.setState("sent",vertex.getInEdge(tuple._3).get
            .getPropertyValuesAfter("value",infectionBlock).get
            .map(x=>x.asInstanceOf[Double]).sum)
            vertex.setState("infected", infectionBlock)
            vertex.setState("infectedBy",infector)
        }
        else{
          vertex.setState("infected", infectionBlock)
          vertex.setState("infectedBy",infector)
          vertex.getOutEdgesAfter(infectionBlock).take(100).foreach { neighbour =>
            neighbour.send((walletID,neighbour
            .firstActivityAfter(infectionBlock),vertex.ID()))
          }
        }
      }
    }
    else{
      if(listOfExchanges contains walletID){
        vertex.setState("exchangeHit", true)
        vertex.setState("sent",vertex.getInEdge(tuple._3).get
        .getPropertyValuesAfter("value",infectionBlock).get
        .map(x=>x.asInstanceOf[Double]).sum)
        vertex.setState("infected", infectionBlock)
        vertex.setState("infectedBy",infector)
      }
      else{
        vertex.setState("infected", infectionBlock)
        vertex.setState("infectedBy",infector)
        vertex.getOutEdgesAfter(infectionBlock).take(100).foreach { neighbour =>
          neighbour.send((walletID,neighbour
          .firstActivityAfter(infectionBlock),vertex.ID()))
        }
      }
    }
    }catch {case e:Exception => }
  }
override def returnResults(): Any = {
  val tosend = view
    .getVertices()
    .map { vertex =>
      if (vertex.containsState("exchangeHit"))
        (vertex.getPropertyValue("id").get.asInstanceOf[String],
            vertex.getState[Long]("infected"),vertex.getState[String]("infectedBy")
        ,true,vertex.getState[Double]("sent"))
      else if (vertex.containsState("infected"))
        (vertex.getPropertyValue("id").get.asInstanceOf[String],
            vertex.getState[Long]("infected"),vertex.getState[String]("infectedBy"),
        false,0)
        else
          ("", -1L,"",false,-1)

      }
      .filter(f => f._2 >= 0)
  }

  override def defineMaxSteps(): Int = 100

  override def processResults(results: ArrayBuffer[Any], timeStamp: Long,
      viewCompleteTime: Long): Unit = {
    val endResults = results.asInstanceOf[ArrayBuffer[immutable.ParIterable[(String,
        Long,String,Boolean,Double)]]].flatten
    var data = s"""{"block":$timeStamp,"edges":["""
    for (elem <- endResults)
      data+=s"""{"infected":"${elem._1}","block":${elem._2},
      "infector":"${elem._3}","exchange":"${elem._4}","value":${elem._5}},"""
    data+="]}"
    publishData(data)
  }
}
```

Appendix D

Code For Spark Comparison

D.1 Windowed Connected Components Across Time in Spark

```
import java.text.SimpleDateFormat

import org.apache.spark.{SparkConf, SparkContext, graphx}
import org.apache.spark.rdd.RDD
import org.apache.spark.graphx.{VertexId, _}

case class Interaction(time:Long,source:Int,destination:Int) //class to represent each
    user interation

def dateToUnixTime(timestamp: => String): Long = { //convert timestamps to epochs
   val sdf = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss")
   val dt = sdf.parse(timestamp)
   val epoch = dt.getTime
   epoch
}

def parseGabData(raw: RDD[String]) = { //turn lines into interactions
   raw.mapPartitionsWithIndex {
      (idx, iter) => if (idx == 0) iter.drop(1) else iter
   }.map(line=> {
      val fileLine = line.split(";")
      Interaction(dateToUnixTime(fileLine(0)),fileLine(2).toInt,fileLine(5).toInt)
   })
val gabdata = parseGabData(sc.textFile("/FileStore/tables/gab.csv"))

val startTimestamp = 1470797917000L //set the time parameters for the analysis
val endTimestamp = 1525368897000L //whole range of gab data
val windows = Array(3600000L,86400000L,604800000L,2592000000L,31536000000L) // windows
    of an hour to year
val range = startTimestamp.to(endTimestamp,86400000L) //hopping forward a day at a time

for(timestamp <- range){
   val startingTime = System.currentTimeMillis()
   for(window <- windows) {
      val currentDataset = gabdata.filter(interaction => //get data between timestamp
          and window
         interaction.time >= timestamp - window && interaction.time <= timestamp)

      val vertices: RDD[(VertexId, (String, String))] = currentDataset
      .map(interaction => (interaction.source.toLong, ("", "")))
      .union(currentDataset.map(interaction =>
         (interaction.destination.toLong, ("", "")))) //extract unique users

      val edges = currentDataset.filter(interaction => interaction.destination >= 0)
         .map(interaction => Edge(interaction.source.toLong,
             interaction.destination.toLong, "x")) //convert interactions to GraphX Edges

      val graph = Graph(vertices, edges, ("", "")) //build the graph
      graph.connectedComponents().vertices
         .groupBy(x => x._2).map(x => (x._1, x._2.size))
         .collect().mkString(",")) //get the biggest components
}
reflect.io.File("/FileStore/tables/sparktest.csv")
   .appendAll(s"$timestamp,${System.currentTimeMillis()-startingTime} \n")
   //output the time taken
}
```

D.2 Router For Generating Gab User Graph

```
class GabUserRouter(override val routerId: Int,override val workerID:Int, val
    initialManagerCount: Int) extends RouterWorker {

  override def parseTuple(tuple: String) = {
    val fileLine = tuple.split(";").map(_.trim)

    val sourceNode = fileLine(2).toInt // extract the source and destination nodes
    val targetNode = fileLine(5).toInt

    val creationDate = dateToUnixTime(timestamp = fileLine(0).slice(0, 19)) //turn the
        date into a unix time
    sendUpdate(VertexAdd(creationDate, sourceNode, Type("User")))
    sendUpdate(VertexAdd(creationDate, targetNode, Type("User")))
    sendUpdate(EdgeAdd(creationDate, sourceNode, targetNode, Type("User to User")))
    //send update for the source, destnation and interaction between
  }

  def dateToUnixTime(timestamp: => String): Long = {
    val sdf = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss")
    val dt = sdf.parse(timestamp)
    val epoch = dt.getTime
    epoch
  }
}
```

Bibliography

[1] B. Erb, D. Meissner, F. Kargl, B. Steer, F. Cuadrado, D. Margan, and P. Pietzuch, "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)*, pp. 1–10, 2018.

[2] P. Holme and J. Saramäki, "Temporal networks," *Physics Reports*, vol. 519, no. 3, pp. 97 – 125, 2012.

[3] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu, "Path problems in temporal graphs," *Proceedings of the VLDB Endowment*, vol. 7, no. 9, pp. 721–732, 2014.

[4] X. Qiu, W. Cen, Z. Qian, Y. Peng, Y. Zhang, X. Lin, and J. Zhou, "Real-time constrained cycle detection in large dynamic graphs," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 1876–1888, 2018.

[5] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, "The ubiquity of large graphs and surprising challenges of graph processing: extended survey," *The VLDB Journal*, pp. 1–24, 2019.

[6] K. Ammar and T. Ozsu, "Experimental analysis of distributed graph systems," *arXiv preprint arXiv:1806.08082*, 2018.

[7] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135–146, 2010.

[8] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2014.

[9] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pp. 17–30, 2012.

[10] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, “Kineograph: taking the pulse of a fast-changing and connected world,” in *Proceedings of the 7th ACM European conference on Computer Systems*, pp. 85–98, 2012.

[11] A. Dubey, G. Hill, R. Escriva, and E. G. Sirer, “Weaver: A high-performance, transactional graph database based on refinable timestamps,” *Proceedings of the VLDB Endowment*, vol. 9, no. 11, 2016.

[12] Y. Miao, W. Han, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, E. Chen, and W. Chen, “Immortalgraph: A system for storage and analysis of temporal graphs,” *ACM Transactions on Storage (TOS)*, vol. 11, no. 3, pp. 1–34, 2015.

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

[14] P. Macko, V. Marathe, D. Margo, and M. Seltzer, “Llama: Efficient graph analytics using large multiversioned arrays,” in *2015 IEEE 31st International Conference on Data Engineering*, pp. 363–374, IEEE, 2015.

[15] B. Erb, D. Meissner, J. Pietron, and F. Kargl, “Chronograph: A distributed processing platform for online and batch computations on event-sourced graphs,” in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, pp. 78–87, 2017.

[16] B. Steer, F. Cuadrado, and R. Clegg, “Raphtory: Decentralised streaming for temporal graphs: Doctoral symposium,” in *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*, DEBS ’17, (New York, NY, USA), p. 363–365, Association for Computing Machinery, 2017.

[17] “Debs 2017 award winners.” Available at: `http://www.debs2017.org/awards/`. Accessed: 28/08/2017.

[18] B. Steer, A. Di Stefano, R. Clegg, and F. Cuadrado, “Building distributed temporal graphs from event streams,” *Proceedings of the VLDB Endowment*, vol. 11, no. 8, 2018.

[19] B. Steer, F. Cuadrado, and R. Clegg, “Raphtory: Streaming analysis of distributed temporal graphs,” *Future Generation Computer Systems*, vol. 102.

[20] J. Waudby, B. Steer, A. Prat-Pérez, and G. Szárnyas, “Supporting dynamic graphs and temporal entity deletions in the ldbc social network benchmark’s data generator.,” in *GRADES*, pp. 8–1, 2020.

[21] N. Arnold, B. Steer, I. Hafnaoui, H. Parada, R. Mondragon, F. Cuadrado, R. G. Clegg, *et al.*, “Moving with the times: Investigating the alt-right network gab with temporal interaction graphs,” *arXiv preprint arXiv:2009.08322*, 2020.

[22] S. A. Myers, A. Sharma, P. Gupta, and J. Lin, “Information network or social network? the structure of the twitter follow graph,” in *Proceedings of the 23rd International Conference on World Wide Web*, pp. 493–498, 2014.

[23] A. Bretto, *Hypergraphs: Basic Concepts*, pp. 1–21. Heidelberg: Springer International Publishing, 2013.

[24] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, “Cypher: An evolving query language for property graphs,” in *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD ’18, (New York, NY, USA), p. 1433–1445, Association for Computing Machinery, 2018.

[25] J. Hayes, “A graph model for rdf,” *Darmstadt University of Technology/University of Chile*, 2004.

[26] O. Hartig and J. Hidders, “Defining schemas for property graphs by using the graphql schema definition language,” GRADES-NDA’19, (New York, NY, USA), ACM, 2019.

[27] M. Besta, E. Peter, R. Gerstenberger, M. Fischer, M. Podstawski, C. Barthels, G. Alonso, and T. Hoefler, “Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries,” *arXiv preprint arXiv:1910.09017*, 2019.

[28] L. Ehrlinger and W. Wöß, “Towards a definition of knowledge graphs.,” *SEMANTICS (Posters and Demos Track)*, vol. 48, pp. 1–4, 2016.

[29] T. Berners-Lee, J. Hendler, and O. Lassila, “The semantic web,” *Scientific American*, vol. 284, no. 5, pp. 34–43, 2001.

[30] J. Spinrad, *Efficient graph representations*. American Mathematical Society, 2003.

[31] M. Besta, M. Fischer, V. Kalavri, M. Kapralov, and T. Hoefler, “Practice of streaming and dynamic graphs: Concepts, models, systems, and parallelism,” *arXiv preprint arXiv:1912.12740*, 2019.

[32] A. Davoudian, L. Chen, and M. Liu, “A survey on nosql stores,” *ACM Computing Surveys*, vol. 51, pp. 1–43, 04 2018.

[33] C. Rickett, U. Haus, J. Maltby, and K. Maschhoff, “Loading and querying a trillion rdf triples with cray graph engine on the cray xc,” *Cray Users Group*, 2018.

[34] J. Miller, “Graph database applications and concepts with neo4j,” in *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, vol. 2324, 2013.

[35] A. Zaki, M. Attia, D. Hegazy, and S. Amin, "Comprehensive survey on dynamic graph models," *International Journal of Advanced Computer Science and Applications*, vol. 7, no. 2, pp. 573–582, 2016.

[36] F. Harary and G. Gupta, "Dynamic graph models," *Mathematical and Computer Modelling*, vol. 25, no. 7, pp. 79–87, 1997.

[37] A. Iyer, E. Li, T. Das, and I. Stoica, "Time-evolving graph processing at scale," in *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, (New York, NY, USA), ACM, 2016.

[38] B. Wheatman and H. Xu, "Packed compressed sparse row: A dynamic graph representation," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pp. 1–7, 2018.

[39] S. Firmli and C. Dalila, *A Review of Engines for Graph Storage and Mutations*, pp. 214–223. 01 2020.

[40] Y. Yang, J. Yu, H. Gao, J. Pei, and J. Li, "Mining most frequently changing component in evolving graphs," *World Wide Web*, vol. 17, no. 3, pp. 351–376, 2014.

[41] F. Kuhn and R. Oshman, "Dynamic networks: Models and algorithms," *SIGACT News*, vol. 42, pp. 82–96, Mar. 2011.

[42] D. Kempe, J. Kleinberg, and A. Kumar, "Connectivity and inference problems for temporal networks," *Journal of Computer and System Sciences*, vol. 64, no. 4, pp. 820 – 842, 2002.

[43] J. Moody, "The importance of relationship timing for diffusion," *Social Forces - SOC FORCES*, vol. 81, pp. 25–56, 09 2002.

[44] N. Santoro, W. Quattrociocchi, P. Flocchini, A. Casteigts, and F. Amblard, "Time-varying graphs and social network analysis: Temporal indicators and metrics," *AISB 2011: Social Networks and Multiagent Systems*, 02 2011.

[45] P. Holme, "Network dynamics of ongoing social relationships," *EPL (Europhysics Letters)*, vol. 64, 08 2003.

[46] O. Michail, "An introduction to temporal graphs: An algorithmic perspective," *Internet Mathematics*, vol. 12, no. 4, pp. 239–280, 2016.

[47] S. Ramesh, A. Baranawal, and Y. Simmhan, "A distributed path query engine for temporal property graphs," *arXiv preprint arXiv:2002.03274*, 2020.

[48] J. Byun, "Enabling time-centric computation for efficient temporal graph traversals from multiple sources," *IEEE Transactions on Knowledge and Data Engineering*, 06 2020.

[49] J. Byun, S. Woo, and D. Kim, "Chronograph: Enabling temporal graph traversals for efficient information diffusion analysis over time," *IEEE Transactions on Knowledge and Data Engineering*, vol. 32, no. 3, pp. 424–437, 2019.

[50] Y. Wang, Y. Yuan, Y. Ma, and G. Wang, "Time-dependent graphs: Definitions, applications, and algorithms," *Data Science and Engineering*, vol. 4, no. 4, pp. 352–366, 2019.

[51] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network motifs: simple building blocks of complex networks," *Science*, vol. 298, no. 5594, pp. 824–827, 2002.

[52] Z. Shen and N. Sundaresan, "Ebay: An e-commerce marketplace as a complex network," WSDM '11, (New York, NY, USA), p. 655–664, ACM, 2011.

[53] A. Masoudi-Nejad, F. Schreiber, and Z. Kashani, "Building blocks of biological networks: a review on major network motif discovery algorithms," *IET Systems Biology*, vol. 6, pp. 164–174(10), October 2012.

[54] U. Meyer and P. Sanders, "δ-stepping: A parallel single source shortest path algorithm," in *European symposium on algorithms*, pp. 393–404, Springer, 1998.

[55] L. Lovász, "Random walks on graphs: A survey," *Combinatorics, Paul erdos is eighty*, vol. 2, no. 1, pp. 1–46, 1993.

[56] T. Valente and G. Vega Yon, "Diffusion/contagion processes on social networks," *Health Education & Behavior*, 2020.

[57] S. Chu and J. Cheng, "Triangle listing in massive networks and its applications," in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, (New York, NY, USA), p. 672–680, ACM, 2011.

[58] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web.," tech. rep., Stanford InfoLab, 1999.

[59] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of mathematical sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[60] U. Khurana and A. Deshpande, "Efficient snapshot retrieval over historical graph data," *Proceedings - International Conference on Data Engineering*, 07 2012.

[61] M. He, S. Pathak, U. Muaz, J. Zhou, S. Saini, S. Malinchik, and S. Sobolevsky, "Pattern and anomaly detection in urban temporal networks," *arXiv preprint arXiv:1912.01960*, 2019.

[62] A. Paranjape, A. Benson, and J. Leskovec, "Motifs in temporal networks," in *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pp. 601–610, 2017.

[63] J. F. Allen, “Maintaining knowledge about temporal intervals,” *Communications of the ACM*, vol. 26, no. 11, pp. 832–843, 1983.

[64] C. Jedrzejek, J. Bak, and M. Falkowski, “Graph mining for detection of a large class of financial crimes,” in *17th International Conference on Conceptual Structures, Moscow, Russia*, vol. 46, 2009.

[65] K. Cooke and E. Halsey, “The shortest route through a network with time-dependent internodal transit times,” *Journal of mathematical analysis and applications*, vol. 14, no. 3, pp. 493–498, 1966.

[66] V. Nicosia, J. Tang, C. Mascolo, M. Musolesi, G. Russo, and V. Latora, “Graph metrics for temporal networks,” in *Temporal networks*, pp. 15–40, Springer, 2013.

[67] T. Peixoto and M. Rosvall, “Modelling sequences and temporal networks with dynamic community structures,” *Nature communications*, vol. 8, no. 1, pp. 1–12, 2017.

[68] R. Rao, P. Mitra, R. Bhatt, and A. Goswami, “The big data system, components, tools, and technologies: a survey,” *Knowledge and Information Systems*, pp. 1–81, 2019.

[69] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[70] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, I. Stoica, *et al.*, “Spark: Cluster computing with working sets,” *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[71] D. Borthakur, “Hdfs architecture guide,” *Hadoop Apache Project*, vol. 53, no. 1-13, p. 2, 2008.

[72] V. Vavilapalli, A. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, and S. Seth, “Apache hadoop yarn: Yet another resource negotiator,” in *Proceedings of the 4th annual Symposium on Cloud Computing*, pp. 1–16, 2013.

[73] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pp. 15–28, 2012.

[74] R. Xin, J. Gonzalez, M. Franklin, and I. Stoica, “Graphx: A resilient distributed graph system on spark,” in *First international workshop on graph data management experiences and systems*, pp. 1–6, 2013.

[75] A. Dave, A. Jindal, E. Li, R. Xin, J. Gonzalez, and M. Zaharia, “Graphframes: an integrated api for mixing graph and relational queries,” in *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, pp. 1–8, 2016.

[76] T. Akidau, S. Chernyak, and R. Lax, *Streaming systems: the what, where, when, and how of large-scale data processing.* " O'Reilly Media, Inc.", 2018.

[77] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. Peng, and P. Poulosky, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1789–1792, 2016.

[78] Twitter, Inc, "Filter realtime tweets." Available at `https://developer.twitter.com/en/docs/tweets/filter-realtime/overview`, 2018. Accessed: 03-05-2018.

[79] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, vol. 11, pp. 1–7, 2011.

[80] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, and E. Schmidt, "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," 2015.

[81] Z. Nabi, *Pro Spark Streaming: The Zen of Real-Time Analytics Using Apache Spark.* Apress, 2016.

[82] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, and S. Owen, "Mllib: Machine learning in apache spark," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[83] A. Jain and A. Nalya, *Learning storm.* Packt Publishing, 2014.

[84] A. Jain, *Mastering apache storm: Real-time big data streaming using kafka, hbase and redis.* Packt Publishing, 2017.

[85] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[86] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[87] W. Fan, C. Hu, and C. Tian, "Incremental graph computations: Doable and undoable," in *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, (New York, NY, USA), p. 155–169, ACM, 2017.

[88] A. McGregor, "Graph stream algorithms: a survey," *ACM SIGMOD Record*, vol. 43, no. 1, pp. 9–20, 2014.

[89] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang, "On graph problems in a semi-streaming model," *Departmental Papers (CIS)*, 2005.

[90] A. Iyer, Z. Liu, X. Jin, S. Venkataraman, V. Braverman, and I. Stoica, "Asap: Fast, approximate graph pattern mining at scale," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 745–761, 2018.

[91] V. Sekara, A. Stopczynski, and S. Lehmann, "Fundamental structures of dynamic social networks," *Proceedings of the National Academy of Sciences*, vol. 113, p. 9977–9982, Aug 2016.

[92] J. Saramäki and E. Moro, "From seconds to months: an overview of multi-scale dynamics of mobile telephone calls," *The European Physical Journal B*, vol. 88, Jun 2015.

[93] W. Bejeck, N. Narkhede, *et al.*, *Kafka streams in action.* Manning Publications Co.,, 2018.

[94] R. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Making state explicit for imperative big data processing," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, (USA), p. 49–60, USENIX Association, 2014.

[95] S. Batra and C. Tyagi, "Comparative analysis of relational and graph databases," *International Journal of Soft Computing and Engineering (IJSCE)*, vol. 2, no. 2, pp. 509–512, 2012.

[96] B. Steer, A. Alnaimi, M. Lotz, F. Cuadrado, L. Vaquero, and J. Varvenne, "Cytosm: Declarative property graph queries without data migration," in *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, pp. 1–6, 2017.

[97] I. Robinson, J. Webber, and E. Eifrem, *Graph databases: new opportunities for connected data.* " O'Reilly Media, Inc.", 2015.

[98] "Stardog: Turn your data into knowledge… fast." https://www.stardog.com/. Accessed: 02/09/2020.

[99] A. Deutsch, Y. Xu, M. Wu, and V. Lee, "Tigergraph: A native mpp graph database," *arXiv preprint arXiv:1901.08248*, 2019.

[100] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and complexity of sparql," *ACM Transactions on Database Systems (TODS)*, vol. 34, no. 3, pp. 1–45, 2009.

[101] R. Angles, M. Arenas, P. Barceló, P. Boncz, G. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, and J. Sequeda, "G-core: A core for future graph query languages," in *Proceedings of the 2018 International Conference on Management of Data*, pp. 1421–1432, 2018.

[102] "Gql standard." https://www.gqlstandards.org/. Accessed: 02/10/2020.

[103] "Apache tinkerpop." https://tinkerpop.apache.org/. Accessed: 02/10/2020.

[104] "Janusgraph: Distributed, open source, massively scalable graph database." https://janusgraph.org/. Accessed: 02/10/2020.

[105] M. Haeusler, T. Trojer, J. Kessler, M. Farwick, E. Nowakowski, and R. Breu, "Chronograph: A versioned tinkerpop graph database," in *International Conference on Data Management Technologies and Applications*, pp. 237–260, Springer, 2017.

[106] "Programmable unified memory architecture." https://archive.fosdem.org/. Accessed: 02/10/2020.

[107] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, *et al.*, "Mathematical foundations of the graphblas," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–9, IEEE, 2016.

[108] M. Kumar, J. Moreira, and P. Pattnaik, "Graphblas: handling performance concerns in large graph analytics," in *Proceedings of the 15th ACM International Conference on Computing Frontiers, CF 2018, Ischia, Italy, May 08-10, 2018* (D. Kaeli and M. Pericàs, eds.), pp. 260–267, ACM, 2018.

[109] T. A. Davis, "Algorithm 1000: Suitesparse:graphblas: Graph algorithms in the language of sparse linear algebra," *ACM Trans. Math. Softw.*, vol. 45, Dec. 2019.

[110] M. Aznaveh, J. Chen, T. Davis, B. Hegyi, S. Kolodziej, T. Mattson, and G. Szárnyas, "Parallel graphblas with openmp," in *2020 Proceedings of the SIAM Workshop on Combinatorial Scientific Computing*, pp. 138–148, SIAM, 2020.

[111] A. Buluç and J. Gilbert, "The combinatorial blas: Design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011.

[112] C. Yang, A. Buluc, and J. Owens, "Graphblast: A high-performance linear algebra-based graph framework on the gpu," *arXiv preprint arXiv:1908.01407*, 2019.

[113] R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Comput. Surv.*, vol. 48, Oct. 2015.

[114] C. Avery, "Giraph: Large-scale graph processing infrastructure on hadoop," *Hadoop Summit*, vol. 11, no. 3, pp. 5–9, 2011.

[115] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.

[116] H. Kung and C. Leiserson, "Algorithms for VLSI processor arrays," *Introduction to VLSI systems*, pp. 271–292, 1980.

[117] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pp. 31–46, 2012.

[118] M. Armbrust, R. Xin, C. Lian, Y. Huai, D. Liu, J. Bradley, X. Meng, T. Kaftan, M. Franklin, A. Ghodsi, *et al.*, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pp. 1383–1394, 2015.

[119] W. Lightenberg, Y. Pei, G. Fletcher, and M. Pechenizkiy, "Tink: A temporal graph analytics library for apache flink," in *Companion Proceedings of the The Web Conference 2018*, pp. 71–72, 2018.

[120] D. Ediger, R. McColl, J. Riedy, and D. Bader, "Stinger: High performance data structure for streaming graphs," in *2012 IEEE Conference on High Performance Extreme Computing*, pp. 1–5, IEEE, 2012.

[121] G. Feng, X. Meng, and K. Ammar, "Distinger: A distributed graph data structure for massive dynamic graph processing," in *2015 IEEE International Conference on Big Data (Big Data)*, pp. 1814–1822, IEEE, 2015.

[122] J. Mondal and A. Deshpande, "Managing large dynamic graphs efficiently," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 145–156, 2012.

[123] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *ACM SIGKDD*, pp. 1222–1230, ACM, 2012.

[124] L. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella, "Adaptive partitioning for large-scale dynamic graphs," in *2014 IEEE 34th International Conference on Distributed Computing Systems*, pp. 144–153, IEEE, 2014.

[125] D. Dossot, *RabbitMQ essentials*. Packt Publishing Ltd, 2014.

[126] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen, "Chronos: a graph engine for temporal graph analysis," in *Proceedings of the Ninth European Conference on Computer Systems*, pp. 1–14, 2014.

[127] G. Agha, "Actors: A model of concurrent computation in distributed systems.," tech. rep., MIT Cambridge Artificial Intelligence Lab, 1985.

[128] B. Erb, G. Habiger, and F. Hauck, "On the potential of event sourcing for retroactive actor-based programming," PMLDC '16, (New York, NY, USA), ACM, 2016.

[129] T. Hartmann, F. Fouquet, M. Jimenez, R. Rouvoy, and Y. Le Traon, "Analyzing complex data in motion at scale with temporal graphs," 2020.

[130] S. Rehfeld, H. Tramberend, and M. Latoschik, "An actor-based distribution model for realtime interactive systems," in *2013 6th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pp. 9–16, IEEE, 2013.

[131] E. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[132] "The Akka Actor Framework." Available at: `http://akka.io/`. Accessed: 28/08/2017.

[133] V. Buterin *et al.*, "Ethereum white paper," *GitHub repository*, vol. 1, pp. 22–23, 2013.

[134] R. Angles, J. Antal, A. Averbuch, P. Boncz, O. Erling, A. Gubichev, V. Haprian, M. Kaufmann, J. L. Pey, N. Martínez, *et al.*, "The ldbc social network benchmark," *arXiv preprint arXiv:2001.02299*, 2020.

[135] 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.

[136] K. Andreev and H. Racke, "Balanced graph partitioning," *Theory of Computing Systems*, vol. 39, no. 6, pp. 929–939, 2006.

[137] "Murmurhash3 information and brief performance results." Available at: `https://github.com/aappleby/smhasher/wiki/MurmurHash3`. Accessed: 13/08/2020.

[138] M. Kidoň and R. Dobai, "Evolutionary design of hash functions for ip address hashing using genetic programming," in *2017 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1720–1727, 2017.

[139] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," in *Proc. 11th Australian Comput. Science Conf.*, pp. 56–66, 1988.

[140] L. Vaquero, A. Celorio, F. Cuadrado, and R. Cuevas, "Deploying large-scale datasets on-demand in the cloud: treats and tricks on data distribution," *IEEE Transactions on Cloud Computing*, vol. 3, no. 2, pp. 132–144, 2014.

[141] M. Masse, *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces.* " O'Reilly Media, Inc.", 2011.

[142] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M. Pham, and P. Boncz, "The ldbc social network benchmark: Interactive workload," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 619–630, 2015.

[143] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu, "Graph pattern matching: From intractable to polynomial time," *Proc. VLDB Endow.*, vol. 3, p. 264–275, Sept. 2010.

[144] N. Masuda, M. Porter, and R. Lambiotte, "Random walks and diffusion on networks," *Physics reports*, vol. 716, pp. 1–58, 2017.

[145] M. Han, K. Daudjee, K. Ammar, T. Özsu, X. Wang, and T. Jin, "An experimental comparison of pregel-like graph processing systems," *Proc. VLDB Endow.*, vol. 7, p. 1047–1058, Aug. 2014.

[146] A. Aghasadeghi, S. Schelter, and J. Stoyanovich, "Zooming out on an evolving graph," in *Proceedings of the 23rd International Conference on Extending Database Technology (EDBT)*, 2020.

[147] J. Saramäki and E. Moro, "From seconds to months: an overview of multi-scale dynamics of mobile telephone calls," *The European Physical Journal B*, vol. 88, no. 6, p. 164, 2015.

[148] K. Lerman, R. Ghosh, and T. Surachawala, "Social contagion: An empirical study of information spread on digg and twitter follower graphs," 2012.

[149] S. Jordan, D. McCoy, and G. Savage, "A fistful of bitcoins: Characterizing payments among men with no names," 2013.

[150] R. Anderson, I. Shumailov, and M. Ahmed, "Making bitcoin legal," in *Cambridge International Workshop on Security Protocols*, pp. 243–253, Springer, 2018.

[151] S. Huang, J. Cheng, and H. Wu, "Temporal graph traversals: Definitions, algorithms, and applications," *arXiv preprint arXiv:1401.1919*, 2014.

[152] J. Whitbeck, M. Dias de Amorim, V. Conan, and J. Guillaume, "Temporal reachability graphs," in *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12, (New York, NY, USA), p. 377–388, ACM, 2012.

[153] D. Greene, D. Doyle, and P. Cunningham, "Tracking the evolution of communities in dynamic social networks," in *2010 international conference on advances in social networks analysis and mining*, pp. 176–183, IEEE, 2010.

[154] R. Kumar, J. Novak, and A. Tomkins, "Structure and evolution of online social networks," in *Link mining: models, algorithms, and applications*, pp. 337–357, Springer, 2010.

[155] M. Elkin, "Distributed approximation: A survey," *SIGACT News*, vol. 35, p. 40–57, Dec. 2004.

[156] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows," *SIAM journal on computing*, vol. 31, no. 6, pp. 1794–1813, 2002.

[157] J. Palsberg and C. Jay, "The essence of the visitor pattern," in *Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac '98) (Cat. No.98CB 36241)*, pp. 9–15, 1998.

[158] T. Morris, "Asymmetric lenses in scala," 2012.

[159] "Scala: Code interpretation at runtime." Available at: `https://scalerablog.wordpress.com/2016/06/20/scala-code-interpretation-at-runtime/`. Accessed: 13/05/2020.

[160] S. Zannettou, B. Bradlyn, E. De Cristofaro, H. Kwak, M. Sirivianos, G. Stringini, and J. Blackburn, "What is gab: A bastion of free speech or an alt-right echo chamber," in *Companion Proceedings of the The Web Conference 2018*, pp. 1007–1014, 2018.

[161] "Hackers steal $48.7m in ethereum from south korean cryptocurrency exchange upbit." https://thenextweb.com/hardfork/2019/11/27/ethereum-upbit-cryptocurrency-exchange-hackers-stolen-million-hot-wallet/.

[162] R. Jain, *The Art of Computer Systems Performance Analysis - Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley Professional Computing, Wiley, 1991.

[163] I. Jimenez, M. Sevilla, N. Watkins, C. Maltzahn, J. Lofstead, K. Mohror, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "The popper convention: Making reproducible systems evaluation practical," in *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, pp. 1561–1570, IEEE, 2017.

[164] G. Sayfan, *Mastering kubernetes.* Packt Publishing, 2017.

[165] "Automatically instrument, monitor and debug distributed systems." Available at: `https://kamon.io/`. Accessed: 26/05/2020.

[166] Prometheus, "Prometheus homepage." Available at `https://prometheus.io/`, 2017. Accessed: 22-11-2018.

[167] "Stop-the-world vs. incremental vs. concurrent." Available at: `https://en.wikipedia.org/wiki/Tracing_garbage_collection#Stop-the-world_vs._incremental_vs._concurrent`. Accessed: 13/05/2020.

[168] C. Flood, R. Kennke, A. Dinn, A. Haley, and R. Westrelin, “Shenandoah: An open-source concurrent compacting garbage collector for openjdk,” in *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pp. 1–9, 2016.

[169] “Backpressure in akka streams.” Available at: `https://dzone.com/articles/backpressure-in-akka-streams`. Accessed: 02/09/2020.

[170] “Ethereum transactions per second.” https://blockchair.com/ethereum/charts/transactions-per-second. Accessed: 02/09/2020.

[171] “Twitter usage statistics.” https://www.internetlivestats.com/twitter-statistics/. Accessed: 02/09/2020.

[172] “Non-cash transactions globally and regionally – forecast growth rates, 2017-2022f.” https://worldpaymentsreport.com/non-cash-payments-volume-2/non-cash-transactions-2017-2022f. Accessed: 02/09/2020.

[173] L. Lőrincz, J. Koltai, A. Győr, and K. Takács, “Collapse of an online social network: Burning social capital to create it?,” *Social Networks*, vol. 57, pp. 43–53, 2019.

[174] R. Anderson, I. Shumailov, M. Ahmed, and A. Rietmann, “Bitcoin redux,” *Workshop on the Economics of Information Security*, 2019.

[175] L. Lima, J. Reis, P. Melo, F. Murai, L. Araujo, P. Vikatos, and F. Benevenuto, “Inside the right-leaning echo chambers: Characterizing Gab, an unmoderated social system,” in *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pp. 515–522, IEEE, 2018.

[176] Pfeffer, Jürgen, Mayer, Katja, and Morstatter, Fred, “Tampering with twitter´s sample api,” *EPJ Data Sci.*, vol. 7, no. 1, p. 50, 2018.

[177] Y. Zhou, M. Dredze, D. Broniatowski, and W. Adler, “Elites and foreign actors among the alt-right: The gab social media platform,” *First Monday*, vol. 24, no. 9, 2019.

[178] S. Foley, J. Karlsen, and T. Putniņš, “Sex, drugs, and bitcoin: How much illegal activity is financed through cryptocurrencies?,” *The Review of Financial Studies*, vol. 32, no. 5, pp. 1798–1853, 2019.

[179] R. Van Wegberg, J. Oerlemans, and O. van Deventer, “Bitcoin money laundering: mixed results?,” *Journal of Financial Crime*, 2018.

[180] “A list of centralized cryptocurrency exchanges which are online platforms that allow customers to buy and sell cryptocurrencies for other assets..” https://etherscan.io/accounts/label/exchange.

[181] M. Capotă, T. Hegeman, A. Iosup, A. Prat-Pérez, O. Erling, and P. Boncz, "Graphalytics: A big data benchmark for graph-processing platforms," in *Proceedings of the GRADES'15*, pp. 1–6, 2015.

[182] R. Lu, G. Wu, B. Xie, and J. Hu, "Stream bench: Towards benchmarking modern distributed stream computing frameworks," in *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pp. 69–78, IEEE, 2014.

[183] A. Basiri, N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.

[184] K. Chodorow, *MongoDB: the definitive guide: powerful and scalable data storage.* " O'Reilly Media, Inc.", 2013.

[185] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[186] E. Capriolo, D. Wampler, and J. Rutherglen, *Programming Hive: Data warehouse and query language for Hadoop.* " O'Reilly Media, Inc.", 2012.

[187] G. Cormode, V. Shkapenyuk, D. Srivastava, and B. Xu, "Forward decay: A practical time decay model for streaming systems," in *2009 IEEE 25th international conference on data engineering*, pp. 138–149, IEEE, 2009.

[188] Q. Bai, C. Zhang, Y. Xu, X. Chen, and X. Wang, "Evolution of ethereum: A temporal graph perspective," *arXiv preprint arXiv:2001.05251*, 2020.