The Many Facets of a Dynamic Graph Processing System

Juntong Luo, Scott Sallinen, Matei Ripeanu {luuo2000, scotts, matei}@ece.ubc.ca

Networked Systems Laboratory, Department of Electrical and Computer Engineering, The University of British Columbia

This work is supported in part by the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC.

NetSysLab
netsyslab.ece.ubc.ca

Lollipop
github.com/ScottSallinen/lollipop

Introduction

Dynamic graph processing systems aim to ingest an evolving graph and repeatedly provide results to a standing graph query at different points in time.

Past work on dynamic graph processing has focused on:
- (1) Performance (e.g., sustained event ingestion throughput, result latency for standing queries)
- (2) Resource consumption (e.g., memory footprint)

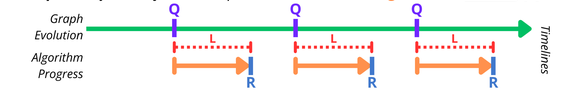
This poster highlights two *additional* important properties:
- **(3) The ability to leverage free computational resources**
  - Rationale: *A well-provisioned graph streaming system should have its maximum throughput higher than the incoming event rates (otherwise buffered events will accumulate), so its ability to utilize these "free" CPU time when event rates are low is important.*
- **(4) Stability of the solution as the graph evolves**
  - Rationale: *On dynamic graphs, it is usually more desirable to have similar results at different points in time. For example, if a clustering algorithm is used to inform resource allocation, a stable algorithm that results in less resource migration is desirable.*

Processing Models

A stream of graph updates arrives at the system, and the system is expected to provide solutions to a standing query at different points in time...

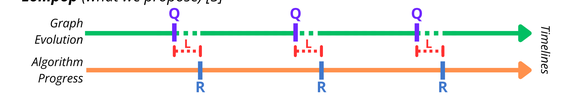
*Timeline of **real-world** **events** with **queries**; Rate below the framework's capacity*

***GraphOne** (fastest system that processes **events** and **algorithm** in batches) [4]*



- **Event** ingestion and **algorithm** execution are *pipelined*

***Lollipop** (what we propose) [5]*



- **Events** and **algorithmic messages** are processed *concurrently*, *asynchronously*, and *autonomously* (i.e., without shared state)
- For fine granular queries, provides *orders of magnitude higher throughput and lower latency* compared to *GraphOne*

Lollipop[5]

Programming Interface
- Vertex-centric, event-driven, and no shared state between vertices
- Vertices communicate via message-passing

Shared-nothing abstraction, amenable to distributed settings

Execution Flow of Each Processing Thread
- Each thread owns a set of vertices, and executes the following tasks:

```
go func() {
    while(true) {
        if(!pending_query) {
            enact_topology_changes();
            if (!topology_buffer_empty)
                continue;
        }
        // Process up to M messages
        process_algorithmic_messages(M);
        if(!pending_query && terminated)
            break;
    }
}
```

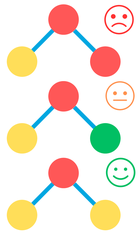
Experiment Settings

Graph [3]
- Wikipedia Hyperlinks
  - With reverse edges added
- $|V|$ = 1.9M $|E|$ = 80M
- Stored as an event log (1.09 GB)
- Timestamps from 2001-02-19 to 2007-04-05 (2,236 days)

Machine
- A commodity desktop with
  - i5-12600K (6 P-Cores & 4 E-Cores)
  - 32GB DDR4 @ 4GHz
  - *CPU + RAM costs $200 + $65 USD*
- Experiments run with 10 threads

Algorithm: Graph Colouring [7]
- *Greedily* assigns a colour to each vertex
- Ensures no two neighbours share the same colour
- Attempts to minimize the total # of colours used

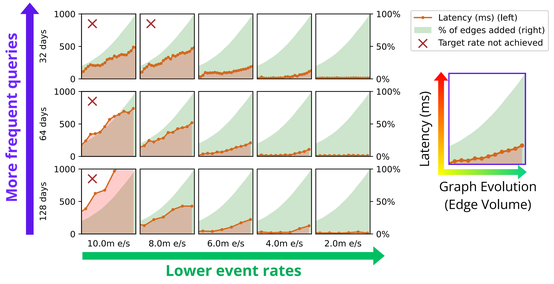
Algorithm: Graph Colouring with Wait Count [7]
- A vertex waits for all neighbours with higher IDs to colour before colouring itself



Latency vs. Event Rate

Key Takeaways
- *Lollipop* can leverage "free" CPU resources and produce *real-time results with dramatically lower latency* for low event rates
- Note this ability is not offered by streaming systems that operate with batches like *GraphBolt* [2] or *GraphOne* [4]

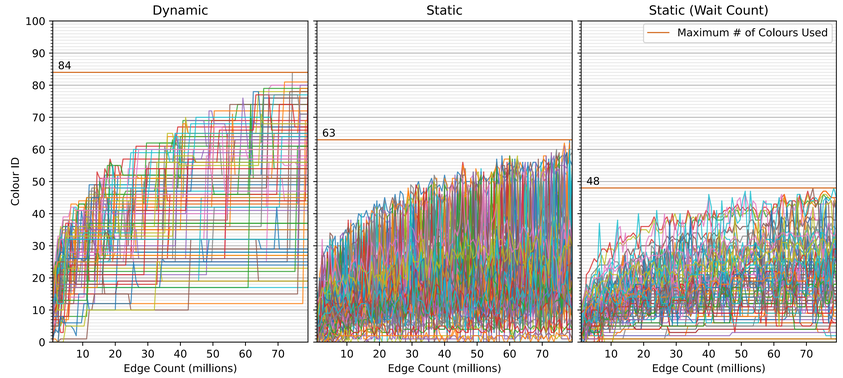
Latency of Lollipop under different event rates and query intervals



** The system is unaware of when the next query or event will be issued*

Solution Stability

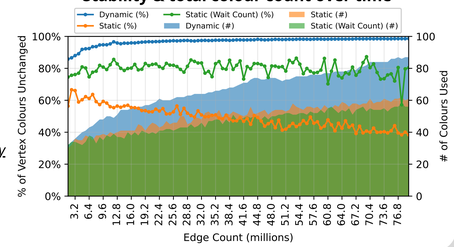
Colour IDs of the top 50 most popular (by PageRank) vertices as the graph evolves



Performance
- *Dynamic* is *orders of magnitude faster* than *Static* when used to generate many results over time
- *Static* is about 2x faster than *Static with Wait Count*

Key Takeaways
- Results provided by a *Dynamic* are *more stable* than those provided by Static ones, but have *lower quality* (uses more colours)
  - A tradeoff in addition to *performance* vs. *quality*

Stability & total colour count over time



Conclusion

We highlights two additional important properties of dynamic graph processing systems, and show Lollipop succeeds in them:
- *The ability to **leverage free computation resources** during lower event rates, to produce solutions with **extremely low latency***
- *The ability to provide **stable results** to a graph analytics problem over time*

References

1. Juntong Luo, Scott Sallinen, and Matei Ripeanu. *Going with the Flow: Real-Time Max-Flow on Asynchronous Dynamic Graphs.* (GRADES & NDA '23)
2. Mugilan Mariappan and Keval Vora. *Graphbolt: Dependency-driven synchronous processing of streaming graphs.* (EuroSys'19)
3. Alan E Mislove. *Online social networks: measurement, analysis, and applications to distributed information systems.* (Rice University 2009)
4. Pradeep Kumar and H Howie Huang. *Graphone: A data store for real-time analytics on evolving graphs.* (TOS'20)
5. Scott Sallinen, Juntong Luo, and Matei Ripeanu. *Real-Time PageRank on Dynamic Graphs* (HPDC '23)
6. Scott Sallinen, Roger Pearce, and Matei Ripeanu. *Incremental graph processing for on-line analytics.* (IPDPS'19)
7. Scott Sallinen, Keita Iwabuchi, Suraj Poudel, Maya Gokhale, Matei Ripeanu, and Roger Pearce. *Graph colouring as a challenge problem for dynamic graph processing on distributed systems.* (SC'16)