The Many Facets of a Dynamic Graph Processing System

Juntong Luo
The University of British Columbia
Vancouver, Canada
luuo2000@ece.ubc.ca

Scott Sallinen
The University of British Columbia
Vancouver, Canada
scotts@ece.ubc.ca

Matei Ripeanu
The University of British Columbia
Vancouver, Canada
matei@ece.ubc.ca

ABSTRACT

Graphs are used to model real-world systems that often evolve over time. We have developed a streaming graph framework which, while ingesting an unbounded stream of events mirroring a graph's evolution, dynamically updates the solution to a user query, and is able to offer, on-demand and with low latency, the solution to the query. Integral to our framework is that graph topology changes and algorithmic messages are processed concurrently, asynchronously, and autonomously (i.e., without shared state). This poster uses graph colouring as a challenge problem to highlight two advantages of our framework beyond those showcased by past work (i.e., low result latency, high sustained ingestion throughput, and scalability). These additional advantages are: *(i)* the ability to efficiently leverage the "free" computational resources available when the rate of incoming topology events is below the maximum sustainable throughput, and *(ii)* the ability to produce "stable" solutions to queries as the graph evolves.

1 INTRODUCTION

Dynamic graph processing is gaining increasing attention. A recent taxonomy by Besta et al. [1] provides a summary of existing *graph streaming* frameworks. These frameworks aim to ingest the evolving graph through a stream of graph updates at high velocity, and to repeatedly provide results to a standing graph analytics query at different points in time. Prior work has shown that they offer a significant performance improvement compared to using traditional solutions originally designed for static graphs, in which the solution for each query needs to be computed from scratch [2, 4, 5].

In most existing state-of-the-art graph streaming frameworks, the system ingests *batches* of graph updates and provides the algorithm with a static snapshot of the evolving graph after each batch; the algorithm then computes the solution for each snapshot, often warm-starting with results from prior snapshots. The process of ingesting events and executing the algorithm can be either *interleaved* (event ingestion stops when the algorithm is running) [5] or *pipelined* (event ingestion continues when the algorithm is operating on a static snapshot) [3].

While offering a relatively easy-to-use algorithmic interface, this approach, however, has several drawbacks:

- High *results latency* (the delay from the moment the user asks for the result to a query – or a batch of events has fully arrived – until the result is produced), especially for frequent queries [8].
- Wasted computational resources. When the system is deployed to process real-world events in real-time, the computational resources given are usually provisioned for the upper bound of the event rate (otherwise buffered events will accumulate over time). Frameworks following the batch approach are unable to fully utilize these resources at lower event rates, as they have to wait for the entire batch to arrive before recomputing (or updating) the solution.

We proposed a more flexible system that is not subject to batch constraints. We have previously shown that the system is scalable, enables *real-time* analysis (allows the solution to be extracted on-demand, with arbitrary granularity, and low latency), and offers more than an order of magnitude performance improvement over prior work for fine granular queries [7–9].

We present two additional advantages of the system, using graph colouring as a case study. First, while prior evaluations have focused on performance bounds at (or near) the maximum sustainable incoming event rates, here we show that, unlike other designs (§3.1), our system can efficiently leverage the "free" computational resources available at lower incoming event rates to further reduce latency. Second, as our system maintains the state of the algorithm as the graph evolves, it is able to produce similar solutions when comparing a new result to a prior result. We show that, compared to computing from scratch for each query, our dynamic solution provides better *solution stability*; for example, in the case of graph colouring, solution stability implies a vertex is more likely to be re-assigned the same colour as the graph evolves and the solution is updated.

2 COMPUTATIONAL MODEL

We present an overview of the abstraction we propose in this section, and refer the reader to [8] for a more detailed description. In our model, each vertex belongs to a processing thread determined by the vertex's identifier and stores a list of outgoing edges. We view each vertex as an independent agent – there is no shared state or direct synchronization between vertices, and vertices communicate with each other only via message passing.

Each processing thread repeatedly performs two tasks: *(i)* dequeuing and applying topology updates and executing topology event handlers, and *(ii)* consuming messages from other vertices and executing message handlers. A thread only processes messages when its topology event buffer is empty (i.e. it prioritizes topology events). Note that a key difference between our model and batch-based models is that, in our model, the algorithm reacts to topology changes right after the graph store is updated, while in batch models, the algorithm reacts only after the current batch is fully ingested. (As an optimization, in our model a vertex can also make progress on consuming algorithm messages when handling topology updates.)

Result collection: Our framework provides solutions on-demand. To collect a solution, the user would insert a marker in the event stream. The system will then block topology events until the algorithm converges, and thus the solution will reflect all events before the marker and none after the marker.

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

3 EVALUATION

To evaluate the proposed solution, we have implemented a prototype named Lollipop[1]. We use a dynamic algorithm for the graph colouring problem, similar to the one described in [7]. The algorithm assigns a colour to each vertex and ensures no two neighbours share the same colour, while also trying to minimize the total number of colours used in the graph (the algorithm is greedy, as optimal colouring is NP-Hard [7]).

We focus here on highlighting two advantages of our solution: *(i)* we show that our solution can efficiently harness the "free" CPU time available when the incoming event rate is below the maximum sustainable throughput; and *(ii)* we show that it produces a stable solution as the graph evolves – a property that solutions based on static algorithms do not provide.

Experiment Settings: We use the incremental temporal Wikipedia-growth graph [6] stored as an event log (1.09 GB), with reverse edges added ($|V| = 1.9M$, $|E| = 80M$), and with timestamps from 2001-02-19 to 2007-04-05 (2,236 days). We use a commodity desktop with a 10-core Intel Core i5-12600K CPU, 32GB DDR4 RAM @ 4000MHz, and NVMe SSD. We parallelize across all 10 cores.

3.1 Latency vs. Ingestion Rate

When evaluating streaming frameworks, it is common to simulate a stream of incoming events by reading from a file, and letting the framework ingest events as fast as possible. This evaluates the maximum sustainable throughput of the system. However, in practice, a system will usually be provisioned such that the average incoming event rate is (well) above its maximum sustainable throughput. Thus, we investigate such a scenario: the offered event rate is somewhere below the maximum rate the framework can handle – we evaluate whether our solution can efficiently leverage the "free" CPU time now available. To this end, we control the incoming event ingestion rates and measure the query latency. Figure 1 presents the results. For all query intervals, we observe that query latency decreases as the event rate decreases. Note that when the system cannot achieve the target ingestion rate (plots marked with a red cross), the system enters the "batch" mode, where it processes only topology updates and few algorithmic updates until a query blocks the topology stream. Remarkably, at low incoming event rates, the latency is orders of magnitude lower compared to that at high event rates, showcasing the system's ability to efficiently leverage "free" CPU resources and produce real-time solutions with extremely low latency. We note that this ability is not offered by streaming systems that operate with batches like GraphBolt [5] or GraphOne [3].

3.2 Solution Stability

In the context of graph colouring on static graphs, solutions are typically deemed of higher *quality* if they require fewer colours. However, on dynamic graphs, we argue that solution *stability* should also be considered as a secondary solution metric: that is, for a dynamic graph, a set of graph colourings is better if the difference between consecutive colourings is minimized; that is, a dynamic graph colouring solution is better if vertices preserve their colour as much as possible over time. One reason is that, clustering algorithms (such as graph colouring) on an evolving graph are used to inform decisions on resource allocation, where a more stable algorithm that results in less resource migration is preferred.

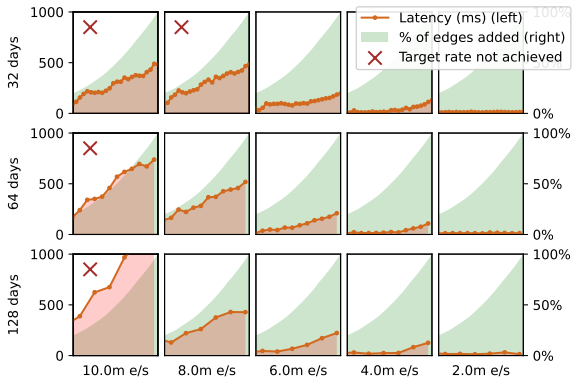

Figure 1: Query latency under different event rates and query intervals. Columns: Target event rate. Rows: Query interval. In each subplot, the x-axis is the time from 2005-03-15 to 2007-03-15, and the y-axes are query latencies in ms (red, left axis) and the percentage of edges added (green, right axis). A brown "X" indicates the target rate could not be achieved. The ingestion rate with no algorithm is 23 million events per second. Queries are generated on observing an event with a timestamp $T_{new} > T_{last} + \lambda$, where λ is the *query interval* and T_{last} is the timestamp of the last query. Note this simulates on-demand queries and the system is unaware of when the next query or event will be issued. Experiments are repeated 10 times.

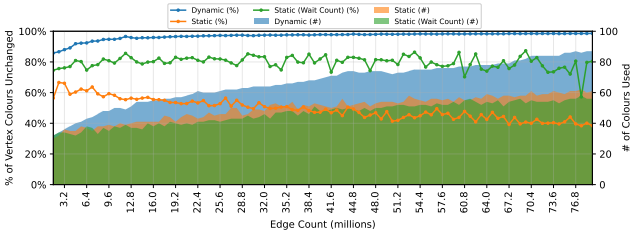

Figure 2: Stability and colour count over time. x-axis: number of edges added; y-axis (left): percentage of all vertices that have their colours unchanged compared to prior query; y-axis (right): total number of colours used. A query is generated for every 800K edges ingested.

Unlike systems based on static algorithms which compute the result for each query from scratch, a dynamic framework can produce a significantly more stable result over time, as shown in Figure 2. The figure compares the stability of the colourings offered by a dynamic algorithm and two static strategies that generate colourings for each snapshot independently: an asynchronous approach, and a synchronized colour choosing approach. There first exists a comparison in runtime and colouring quality: the asynchronous static approach is about twice as fast as the synchronized static approach, but has lower quality. The dynamic approach – when used to generate many results over time – is orders of magnitude faster to generate all colourings, but also has lower quality. Yet, for the metric of stability as the graph evolves, we observe that the dynamic algorithm produces significantly more stable colourings as a trade-off to using more colours compared to the static algorithm approach.

[1] Lollipop is open-source: https://github.com/ScottSallinen/lollipop

REFERENCES

[1] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. 2023. Practice of Streaming Processing of Dynamic Graphs: Concepts, Models, and Systems. *IEEE Transactions on Parallel and Distributed Systems* 34, 6 (2023), 1860–1876. https://doi.org/10.1109/TPDS.2021.3131677

[2] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. 2014. Chronos: A Graph Engine for Temporal Graph Analysis. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 1, 14 pages. https://doi.org/10.1145/2592798.2592799

[3] Pradeep Kumar and H Howie Huang. 2020. Graphone: A data store for real-time analytics on evolving graphs. *ACM Transactions on Storage (TOS)* 15, 4 (2020), 1–40.

[4] Juntong Luo, Scott Sallinen, and Matei Ripeanu. 2023. Going with the Flow: Real-Time Max-Flow on Asynchronous Dynamic Graphs. In *Proceedings of the 6th Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (Seattle, WA, USA) *(GRADES & NDA '23)*. Association for Computing Machinery, New York, NY, USA, Article 5, 11 pages. https://doi.org/10.1145/3594778.3594882

[5] Mugilan Mariappan and Keval Vora. 2019. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 25, 16 pages. https://doi.org/10.1145/3302424.3303974

[6] Alan E Mislove. 2009. *Online social networks: measurement, analysis, and applications to distributed information systems*. Rice University.

[7] Scott Sallinen, Keita Iwabuchi, Suraj Poudel, Maya Gokhale, Matei Ripeanu, and Roger Pearce. 2016. Graph Colouring as a Challenge Problem for Dynamic Graph Processing on Distributed Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '16)*. IEEE Press, Article 30, 12 pages.

[8] Scott Sallinen, Juntong Luo, and Matei Ripeanu. 2023. Real-Time PageRank on Dynamic Graphs. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing* (Orlando, FL, USA) *(HPDC '23)*. Association for Computing Machinery, New York, NY, USA, 239–251. https://doi.org/10.1145/3588195.3593004

[9] Scott Sallinen, Roger Pearce, and Matei Ripeanu. 2019. Incremental Graph Processing for On-line Analytics. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1007–1018. https://doi.org/10.1109/IPDPS.2019.00108

A ARTIFACT DESCRIPTION

This appendix describes the steps to replicate the experiments presented in the poster and the extended abstract.

A.1 Experiment Setup

Source Code. Our prototype framework Lollipop is available for download at https://github.com/ScottSallinen/lollipop, and the commit hash is `e5261c6`. Lollipop is written in Go, and we use Go version 1.20.6 for the experiments.

Test Graph. The test graph, `wikipedia-growth`, has 1.9 million vertices and 40 million edges. It is stored as an event log with a size of 1.09 GB. Each edge has a capacity of 1. The timestamps range from 2001-02-19 to 2007-04-05 (2,236 days). To create an undirected graph, Lollipop add a corresponding reverse edge for every edge in the original graph. The total number of edges is 80 million. The graph is available for download at https://greymass.ca/graphs/.

Machine. Experiments were run on a commodity desktop with the following specifications:

- CPU: one 10-core Intel Core i5-12600K running at the default frequency
- RAM: 2x16GB DDR4 @ 4000MHz (F4-4000C18D-32GVK)
- Storage: 2TB NVMe SSD (WDS200T2X0E), formatted with the `btrfs` filesystem
- Operating System: Arch Linux running Linux kernel 6.4.4-arch1-1

A.2 Experiment Workflow

The Jupyter Notebook located at

`scripts/2023-sc-poster/2023-sc-poster.ipynb`

handles all tasks from running the source code to generating the figures. Please follow the instructions inside the notebook to set the number of threads, the path to the source code, and the path to the test graph.