Kaleidoscope: Graph Analytics on Evolving Graphs

Steffen Maass Taesoo Kim
Georgia Institute of Technology

ABSTRACT

Large-scale graphs and their analytics are common in many applications such as social networks, data mining, and machine learning. Many approaches to operate on *static* graphs of sizes in the trillions of edges have been proposed and adopted. However, real-world graphs are *evolving* in nature and are constantly changing in many application domains. We observe that current systems to process evolving graphs show three problems: they impose a *storage overhead* when adding new edges and vertices, they involve a *synchronous compaction* step to keep their internal data structures optimized and they exhibit poor *cache locality*.

We propose Kaleidoscope—a graph processing engine for evolving graphs—that aims at tackling these three problems by the use of a localized graph data structure, that is helpful in reducing both the storage overhead as well as the overhead of the synchronous compaction step. Kaleidoscope also enables the use of a locality-optimizing space-filling curve, the Hilbert order, when traversing the subgraphs built by Kaleidoscope, to allow for better cache locality.

1 MOTIVATION

Graphs are the basic building blocks to solve various problems ranging from data mining, machine learning, scientific computing to social networks and the world wide web. However, with the advent of Big Data, the sheer increase in size of the datasets [1] poses fundamental challenges to existing graph processing engines.

While the area of graph analytics on large, static graphs has been explored extensively for different setups like in-memory and on-disk computing for both a single machine and a distributed system, there exists a smaller body of work for the case of temporally evolving graphs. In these systems, we identify three problems: How to reduce the *storage overhead* associated with processing an evolving graph, how to avoid a *synchronous compaction* step, and how to optimize for *cache locality*.

Current, state-of-the-art systems handling temporally evolving graphs focus on the design of a data structure that can support updates to the underlying graph structure as well as the kind of algorithms that can be deployed using the abstractions needed to support graph updates. In particular, current systems, like LLAMA [5] or EvoGraph [8], split the storage of the graph into two portions: An inactive, compact portion, stored by using, for example, compressed sparse rows (CSR) and an active portion, stored by using a simple edge array to support inexpensive addition of edges [8]. Other design options for the active portion include delta snapshots [5] of the CSR format or an adjacency list-like design where edges are being appended to a per-vertex list as in Stinger [2].

Existing research approaches do not fulfill the important system characteristics, such as reducing the storage overhead, the compaction overhead, and improving the cache locality, needed for efficiently processing evolving graphs. For example, existing approaches to storing the additions to the graph lead to overheads in terms of storage, by as much as a factor of two when switching from the CSR format to edge lists.

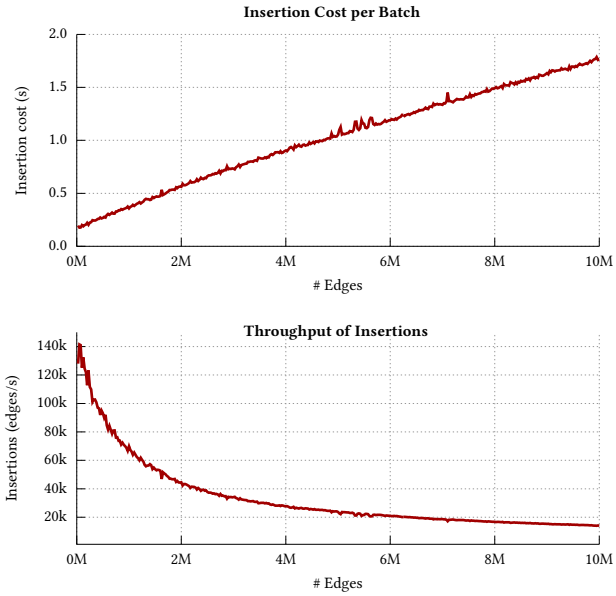

Figure 1: The cost per insertion of batch of edges and the resulting insertion throughput in edges per second supported on Stinger. Insertion is being handled at batches of 25,000 edges every two seconds and approaches a backlogged state when inserting another batch with ten million edges already being present in the graph. The throughput of insertions collapses to 14,000 insertions per second, highlighting the problem of Stinger's adjacency-list based edge storage design.

We exemplify the problem of increased memory usage and dropping insertion performance due to the design of the underlying data structure with Stinger. We insert a total of ten million edges with one million vertices into Stinger in batches of 25,000 edges, using the RMAT algorithm. The batches are inserted every two seconds, to give Stinger time to catch up with the insertion process and the execution of the attached (evolving) algorithms, in this case a simple Pagerank analysis. Figure 1 shows the cost per batched insertion and the resulting throughput, in edges per second. Stinger's throughput drops significantly and, worse, non-linearly with a larger number of edges already being present in the graph, resulting in insertion times of almost two seconds when ten million edges are already present in the graph with about 14,000 edges per second as throughput. This behavior shows a significant bottleneck of Stinger's design of using adjacency lists and limits the applicability of this technique to very small graphs: Popular graphs used

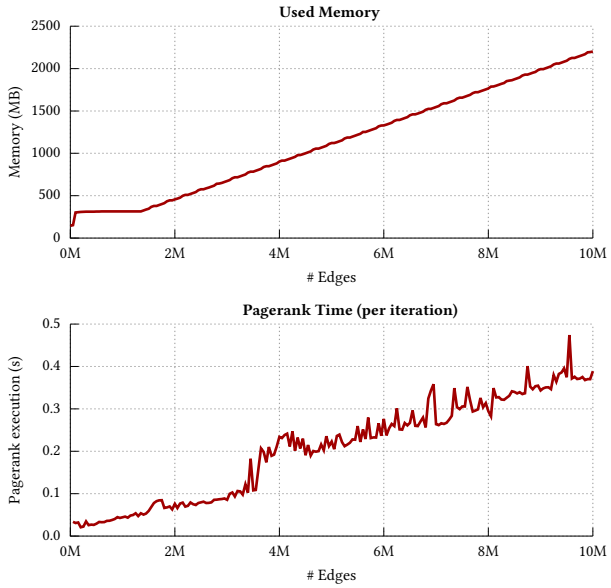

Figure 2: The amount of memory used and the time taken per iteration when executing the pagerank algorithm on Stinger. We start with an empty graph and insert ten million edges generated using the RMAT algorithm. The performance of pagerank visibly drops with an increasing number of edges while the memory overhead grows to more than 2 GB with only ten million edges while a CSR-style storage of the edges would result in less than 800 MB of storage requirements.

to evaluate graph processing systems usually have billions of edges and hundreds of million of vertices [7, 10, 11].

We further show the amount of memory used by Stinger, using the same scenario of inserting ten million edges in batches of 15,000 edges, along with the per-iteration cost of executing the Pagerank algorithm. The results are shown in Figure 2 and exemplify the problem of increased memory usage and the dropping performance of an associated algorithm when running it on an evolving graph. When inserting edges, Stinger's memory requirements grow to more than 2 GB at ten million inserted edges, while the storage of the same amount of edges in an optimized CSR-like format would only amount to about 150 MB. Even taking the overhead of Stinger into account, this amount of overhead is not sustainable for larger graphs, given the requirements of real-world applications of billions of edges. The results also show that the cost per iteration of the Pagerank algorithm, that is being run incrementally on the evolving graph, increases with a higher-than-linear rate when inserting more edges into the graph as an effect of Stinger's data structure design of using adjacency lists. This is further shown in Stinger's high last-level cache miss rates of close to 80% when executing the Pagerank algorithm with concurrent edge insertions. These two results motivate our work on Kaleidoscope, to overcome these limitations to be able to store larger graphs and execute algorithms on them efficiently.

Additionally, storing new edges in the overflow part of the graph leads to difficulties when accessing all outgoing edges of a vertex: While the CSR format allows easy traversal of outgoing edges, the edge list format does not, in the worst case every edge in the list has to be checked to decide whether or not the source vertex is the vertex of interest. Worse yet, compacting the graph into the efficient CSR format is only possible in a synchronous manner, halting all insertions into the graph while temporarily suspending the execution of algorithms as the graph is being compacted and put into the CSR format. Finally, the traversal behavior of mixing adjacency lists and edge lists using an overflow area can lead to poor cache locality, as vertices are now being "randomly" traversed, in the order given by the edge list.

Taking these important system characteristics into account, we propose Kaleidoscope, a graph processing system for temporally evolving graphs. Kaleidoscope looks at addressing *three* fundamental problems in the area of temporally evolving graph analytics: How to design a system that reduces the *storage overhead* associated with current designs for evolving graphs, how to address the problem of the *synchronous*, all-or-nothing approach to compaction of the graph, and how to optimize for *cache locality*, a key goal in graph analytics (both static and evolving). We now outline our ideas for each of these problems and give background information on how these kind of problems are tackled in the case of static graph processing.

2 OVERVIEW

To tackle these problems in systems processing evolving graphs we first outline an approach of how to optimize for locality and compression in *static* graphs before applying similar, yet different techniques to the case of evolving graphs with Kaleidoscope.

2.1 Static Design for Locality: Mosaic

We build upon previous work in the domain of graph analysis on static graphs to derive a design for evolving graphs. In particular, we take a look at Mosaic [4]: Its main contribution is a design that optimizes the locality of accesses to the global vertex state while allowing for compression and independent processing of subgraphs whose union makes up the overall graph. Mosaic employs the Hilbert order, a space-filling curve, to optimize the locality of accesses to the vertex array when traversing the subgraphs built by Mosaic.

However, Mosaic's internal structure is tailored to executing on a static graph only, by tightly packing the adjacency list and using shorter, local identifiers that leave no room for further vertex insertions and is thus not suited for the task of processing on evolving graphs.

2.2 Kaleidoscope for Evolving Graphs

The main idea of Kaleidoscope is to use a similar notion of Mosaic of *independent* subgraphs/tiles coupled with a locality-maximizing traversal, like the Hilbert order, to allow for cache-locality, scalability and compression.

The local graph. Kaleidoscope builds on and extends Mosaic's concept of local subgraphs, so-called *tiles*, to enable a localized, independent processing of subgraphs. However, in contrast to Mosaic, Kaleidoscope's *tiles* are constructed in a simpler manner that allows for easier updates when future edges and vertices are being added. We propose to use simple blocks of size $2^{16} \times 2^{16}$ as the basic unit of processing in Kaleidoscope. This size ensures that vertex

IDs can be efficiently represented inside the tile, using short, 16 bit long identifiers, while being large enough to avoid many load-balancing issues an approach with fewer vertices might have. These identifiers will then get mapped into the global identifier space by the means of a prefix being stored in the metadata associated with a specific tile.

Furthermore, bucketing the graph into tiles like these allows for independent updates to be applied to many portions of the graph in parallel and opens up avenues for improving the *cache locality* when using more intelligent, space-filling traversals.

We propose Kaleidoscope to use a simple trade-off between adjacency lists, for old and compacted edges, and edge lists, for new and unorganized edges [8].

Incremental compaction. Given the partitioned approach to the graph construction, Kaleidoscope enables a trade-off between the usual all-or-nothing approach to compaction: Instead of compacting the entire graph in a single step, Kaleidoscope enables localized compaction which can finish much faster than a compaction step on the entire graph, thus interrupting the addition of new edges for a much shorter time.

Locality. As demonstrated by COST [6] and Mosaic, employing a space-filling curve, like the Hilbert order, can improve cache locality by 50% and more. We thus propose to employ a similar approach to improving cache-locality in the setting of evolving graphs, using the Hilbert order to achieve cache locality in the accesses to the global vertex array when processing neighbouring tiles. Note that Kaleidoscope executes many tiles in parallel, thus achieving cache locality across the execution of multiple, in parallel processed tiles.

Vertex state update. Kaleidoscope also has to apply an incremental approach to updating the vertex state: For example, the number of outgoing and incoming edges (out- and in-degree) changes with every addition of an edge for a specific vertex. Kaleidoscope supports this by keeping delta-records of the changes for the vertex state, to allow for later processing of the historical state of the graph as well.

3 DISCUSSION

Applicable scenarios for Kaleidoscope. Kaleidoscope is targeted at a sweet spot of applications that see updates often enough that they matter, but not too often that they generate too much turnover in the resulting graph. Good scenarios for Kaleidoscope include settings like social networks or a web graph-like scenario which show a power law distribution of vertex degrees, implying a few, highly popular vertices with a high degree while many vertices have a low degree and thus a low expected turnover. This in turn allows Kaleidoscope to skip the processing (and thus overheads like TLB misses, etc.) of many of its tiles, concentrating the churn in a few highly popular tiles.

To give a consistent snapshot when executing an algorithm in Kaleidoscope under concurrent vertex insertion, we tag edges with a version identifier that allows Kaleidoscope to identify whether a given edge has to be included in the result of an algorithm execution for a specific snapshot identifier.

Applications for Kaleidoscope. Kaleidoscope is targeted to support algorithms that build upon a common gather-apply-scatter (GAS) abstraction [3], that already supports many algorithms and can be extended to an incremental variant as used in GraphIn [9]. As such, the set of potential algorithms and applications Kaleidoscope can support are traditional algorithms like a breadth-first search and a connected components analysis as well as more complicated algorithms like analyzing the clustering coefficients and counting triangles.

4 CONCLUSION

We propose Kaleidoscope, a graph processing engine for temporally evolving graphs. Kaleidoscope addresses three problems in current engines for processing evolving graphs: high memory usage when inserting edges into the graph due to untapped potential of storing the graph in a compressed form, overhead when compacting the graph into its compressed form, and poor cache locality as a result of storing the graph in an uncompressed form.

Kaleidoscope addresses these issues with a localized, tiled graph storage data structure that reduces the storage overhead as well as alleviate the problem of a synchronized graph compaction step.

5 ACKNOWLEDGMENT

We thank the anonymous reviewers for their helpful feedback. This research was supported, in part, by the NSF award CNS-1563848, CNS-1704701, CRI-1629851, CNS-1749711, ONR under grant N000141512162, DARPA TC (No. DARPA FA8650-15-C-7556), ETRI IITP/KEIT[B0101-17-0644], and gifts from Facebook, Mozilla and Intel.

REFERENCES

[1] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One Trillion Edges: Graph Processing at Facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, Aug 2015.

[2] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. STINGER: High Performance Data Structure for Streaming Graphs. In *2012 IEEE Conference on High Performance Extreme Computing*, pages 1–5, Sept. 2012.

[3] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, Hollywood, CA, Oct. 2012.

[4] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, pages 527–543, Belgrade, SR, Apr. 2017.

[5] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. In *2015 IEEE 31st International Conference on Data Engineering*, pages 363–374, April 2015.

[6] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at what COST? In *15th USENIX Workshop on Hot Topics in Operating Systems (HotOS) (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015.

[7] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[8] D. Sengupta and S. L. Song. EvoGraph: On-the-Fly Efficient Mining of Evolving Graphs on GPU. In J. M. Kunkel, R. Yokota, P. Balaji, and D. Keyes, editors, *High Performance Computing*, pages 97–119. Springer International Publishing, 2017.

[9] D. Sengupta, N. Sundaram, X. Zhu, T. L. Willke, J. Young, M. Wolf, and K. Schwan. GraphIn: An Online High Performance Incremental Graph Processing Framework. In *Proceedings of the 22nd International European Conference on Parallel and Distributed Computing (EuroPar)*, pages 319–333, Grenoble, France, Aug. 2016.

[10] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. GraM: Scaling Graph Computation to the Trillions. In *Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC)*, Kohala Coast, Hawaii, Aug. 2015.

[11] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *13th USENIX Conference on File and Storage Technologies (FAST) (FAST 15)*, pages 45–58, Santa Clara, CA, Feb. 2015.