EdgeView: Thinking Like an Edge in a Graph

Haotian Gong
University of British Columbia
ht2012@student.ubc.ca

ABSTRACT

FlexoGraph is a graph processing system that supports both transactional operations and running graph analytics. We demonstrate that the *Edge Log* representation extends the capabilities of the system and strikes a balance between transactional performance and analytics performance.

KEYWORDS

Graph Databases, Graph Processing, Data Management, Information Systems

ACM Reference Format:
Haotian Gong. 2023. EdgeView: Thinking Like an Edge in a Graph. In *Trade-offs in Designing Computer Systems, Winter Session Term 2, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 6 pages.

1 INTRODUCTION

Modeling data as graphs allows us to capture complex relationships in a condensed form. Graph-structured data are ubiquitous across various fields, and analyzing them often yields important insights. From Google's PageRank [2] algorithm that orders results for every search, to analysis of millions lines of C code for compiler optimization [11], the ability to store and process graphs is essential to today's software systems.

However, graph datasets are large and constantly growing. Graphs with over 1 billion edges are common in practice, and organizations from academia and industry are showing the need to analyze such graphs [7]. This calls for research into scalable systems for cost-efficient graph data management and processing.

Currently, two major classes of applications for managing graphs exist: graph databases and graph processing systems. Graph databases focus on the "persistence" aspect, allowing concurrent read/write of data under the guarantee of Atomicity, Consistency, Isolation, and Durability (ACID) properties.

State-of-the-art graph processing systems are highly scalable and performant, outperforming Graph databases in the ‘"analytical" aspect. However, they typically do not provide ACID guarantees. They operate on custom-designed data layouts, while directly managing I/O and scheduling. This means we pay a pre-processing cost to bring the graph from the data source into the processing system each time we analyze the graph.

We designed and implemented *FlexoGraph* to couple graph storage with graph analytics closely. It runs graph algorithms directly on the primary persistent data source, effectively eliminating the preliminary transformation required in graph processing systems. We leverage a production-quality storage engine, WiredTiger, to support ACID-compliant transactional operations on the data source. Finally, we provide a simple, intuitive API to the user in C++.

We observed that the source data representation is critical to the performance of *FlexoGraph*. Previously, we modeled 2 representations, *Adjacency List* and *Edge Key*: *Adjacency List* condenses neighbourhood information well, making it efficient for vertex-centric iteration; *Edge Key* is modified from a compressed sparse row (CSR) representation, supporting vertex-centric iteration but produces degraded performance.

I prototyped a new *Edge Log* representation that increases the flexibility of *Edge Key*, relaxing the previous (source, destination) order to allow for any desired edge order, fully leveraging the benefits of such layout. This project makes the following contributions:

- Designed and implemented *Edge Log*.
- Created benchmarks to assess transactional insertion and deletion of data withindifferent reprentations.
- Evaluated *Edge Log* to elucidate the transactional vs. analytics performance tradeoff, comparing it against previous representations.
- Identified directions of improvement in the current system.

2 BACKGROUND

2.1 Vertex vs. Edge Centric

The concepts of vertex-centric(VC) and edge-centric(EC) iterations were first introduced by GraphChi[5] and X-stream [6]. It is important to note that these are independent of the gather-apply-scatter (GAS) model in graph processing. The iteration style is dependent only on how data is brought from secondary storage (e.g., hard disks) into primary storage (e.g., RAM).

VC iteration imposes the restriction that when we propagate values over an edge (e.g., the scatter phase in GAS), we first select the source / destination node of the propagation,

EECE 571H, Winter Session Term 2, 2023, Vancouver, BC, Canada 2023.

then bring in all edges in the in/out-neighbourhood of the selected node.

EC iteration relaxes this limitation, which means that we can extract edge data in any order. Under this intepretation, VC is a special case of EC iteration. There may be more performant EC orderings, such as the Hilbert Order, which is optimized for enhanced cache-locality. We treat EC as having "no specific ordering".

In parallel graph processing systems, VC is often more difficult to parallelize due to the phenomenon of power-law degree distributions. In network graphs, it is common to have "hub nodes" whose degrees are orders of magnitude larger than those of other nodes. Scheduling threads for these nodes and preventing stragglers dynamically is difficult. With EC, however, we can evenly partition the edges among each thread due to its non-ordering property. Recently, there are works that propose running EC iteration on GPUs.

2.2 WiredTiger

WiredTiger is a production quality, open-source data management platform. It uses B+ trees internally to store data. It uses a multi-version, optimistic concurrency control mechanism. As a result, read do not block writes, but write-write conflicts will cause all but one participating transaction to fail and roll back.

2.3 Graph Layouts

Within *FlexGraph*, we constructed the following representations for evaluation and comparison previously.

2.3.1 Adjacency List. The *Adjacency List* representation maintains 3 tables: the node table has key (`node_id`) and holds degree information for each node. The out-neighbourhood table has key (`src_id`) with its value being an array of each out-neighbour. The in-neighbourhood table has key (`dst_id`) also storing all in-neighbours as its value.

In total, *Adjacency List* stores approximately $4|V| + 2|E|$ keys and values, with $2|V|$ from the node table and $|V| + |E|$ for both neighbourhood tables.

We expect *Adjacency List* to have bad performance with transactional operations, as there will be multiple occasions for write-write conflicts. Since each key in the neighbourhood table corresponds to multiple edges, any simultaneous insertion or deletion that collides in `src` or `dst` will cause a rollback.

2.3.2 Edge Key. The *Edge Key* representation is derived from a CSR format. It combines the vertex and edge tables, using (`src`, `dst`) as a composite key. The value entry is currently unused. We denote an entry as a node by setting $dst = -1$.

Edge Key stores approximately $3|V| + 2|E|$ keys and values, as we have $2|V|$ keys for nodes and $2|E|$ for edges. We store an extra $|V|$ values for degree data corresponding to each node.

Since *Edge Key* needs to support VC iterations, we are required to build two secondary indices on the `src` and `dst` columns. Iterating over a secondary index is significantly slower than a sequential scan over the table, as we produce random accesses into the storage medium.

However, we expect a transactional performance improvement relative to *Adjacency List*, because a `src`, `dst` pair uniquely identifies an edge. We no longer run into conflicts updating the same edge, but may run into conflicts updating the node degree count in our table.

3 EDGELOG

3.1 Motivation

After identifying points in our design that could cause a slowdown of transactional writes, we devised a new representation called *Edge Log*. It takes a minimalist approach, keeping as little shared vertex information as possible to prevent write-write conflicts when edge modifications update metadata for the same node.

Furthermore, we fully leverage the fact that EC does not impose an order on how we load data into memory. Instead, we assign a new primary key `edge_id` that uniquely identifies each edge. We iterate in `edge_id` order, so by assigning `edge_ids` in a desirable manner, e.g., in Hilbert order, we provide greater flexibility in EC computation.

3.2 Layout

The *Edge Log* table stores 3 columns: (`edge_id, src, dst`). Unlike *Adjacency List* and *Edge Key*, it does not keep any degree metadata for nodes. We push the computation of such information to when analytics are run, trading off analytics performance for transactional performance.

Thus, we store a total of $3|E|$ items in our table. We also build a secondary index of (`src, dst`) on the table for deletions, as the user `edge_id` is managed internally and not exposed. The user will only need to provide source and destination of the edge to delete.

We expect excellent insertion performance, since we always write in sequential manner to the disk. Deletions will be more expensive due to the cost of traversing the secondary index; this is the cost we pay for avoiding write conflicts.

3.3 Implementation

Introducing `edge_id` requires a concurrency control mechanism that distributes the appropriate `edge_id` to each thread. A naïve implementation of a centralized counter that distributes one id at a time is not sufficient because it still forces threads to synchronize.

We choose to implement a centralized manager that distributes a range of ids whenever necessary. On every insert, a thread checks whether its local insertion range is uninitialized or depleted. If so, it requests more ids from the centralized manager. Whenever a range is assigned, the manager treats that entire range as used up and responds to subsequent requests starting at the first available id after the range returned to the thread.

This mechanism reduces thread blocking and enables further edge partitioning optimizations for analytics by introducing the `edge_id`. In *Edge Key*, our current partitioning method traverses a snapshot of the entire edge table in $O(|E|)$ to assign threads an evenly divided section of edges. With `edge_id`, we can determine an approximate partition in $O(1)$ by dividing the largest `edge_id`.

The partition is approximate because we do not know exactly how many of the ids in an assigned range are actually used in insertion. Due to time constraints, we will leave the evaluation and optimization of the approximation strategy for later work and use the naïve strategy in our experiments.

4 EVALUATION

We have conducted a performance benchmarking of *Edge Log*, focusing on three key aspects: storage efficiency (4.1), transactional operations performance (4.2), and graph algorithm performance (4.3).

We operate on synthetic graphs generated by the highly configurable PaRMAT generator [4]. The graphs follow a RMAT distribution, which closely models common real-world graphs [3].

We perform all experiments on a Dell Precision 5820, with 28-core Intel Xeon 3.30 GHz W-2275 CPU, 128 GB RAM, and a 1TB KIOXIA NVMe SSD.

4.1 Storage Efficiency

Figure 1 shows that the database size grows at nearly the same rate of edge (and vertex) count. For convenient evaluation, we configured PaRMAT to maintain a constant edge: vertex count of 8: 1. This shows WiredTiger does not need to store significant amounts of administrative data as the table grows.

Table 1 shows a breakdown of the storage components when the database contains a generated graph with 2^{25} edges. *Adjacency List* is 6× more efficient than *Edge Key* despite storing a larger amount of values. This is because internally WiredTiger compresses the neighbourhood array in the in / out-neighbourhood tables. *Edge Key* is significantly impacted by the secondary indices it needs to keep. The combined size of the indices exceeds half of the total storage capacity. Compared to *Edge Key*, *Edge Log* reduces disk consumption by maintaining only one secondary index.

4.2 Transactional Operations

For benchmarking transactional operations, we randomly sample 64, 000 edges for deletion, then randomly generate 64, 000 edges for insertion. Finally, we partition the insertion and deletion workloads evenly across available threads, and measure the runtime.

From Figure 2 a), we not only observe that *Edge Log* obtains a 14.5× and 6.2× speedup over the *Adjacency List* and *Edge Key* representations, but also discover that the other two representations are bottle-necked. Increasing parallelism results in zero to negative improvement in performance.

This behavior is explained by Figure 2 b), where we measure the total number of transactional rollbacks that occurred. As the number of threads increases, both *Adjacency List* and *Edge Key* experience a growing number of conflicts. Interestingly, parallelizing extensively *Edge Key* exacerbates this issue and creates even more contention than *Adjacency List*. However, *Edge Log* circumvents this issue by guaranteeing no write conflicts, and scales well to the number of threads.

4.3 Graph Algorithms

To provide representative results, we selected two widely used graph algorithms: Weakly Connected Components (WCC) and Breadth-First Search (BFS). The reference implementations of WCC and BFS available in the GAP Benchmark Suite [1] assume fast access to a node's neighborhoods. However, in *FlexoGraph*, such access can only be achieved by VC iteration with *Adjacency List* or by using the secondary indices of *Edge Key*. To ensure a fair comparison, we developed our own set of algorithms that do not rely on fast neighborhood access assumptions. We ran these algorithms on *Edge Key*

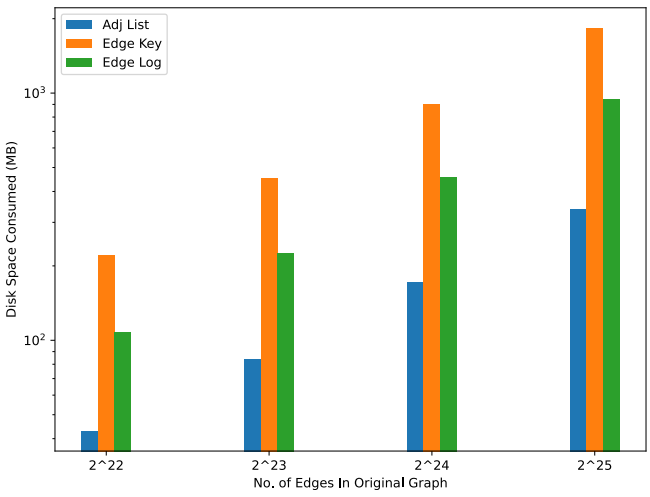
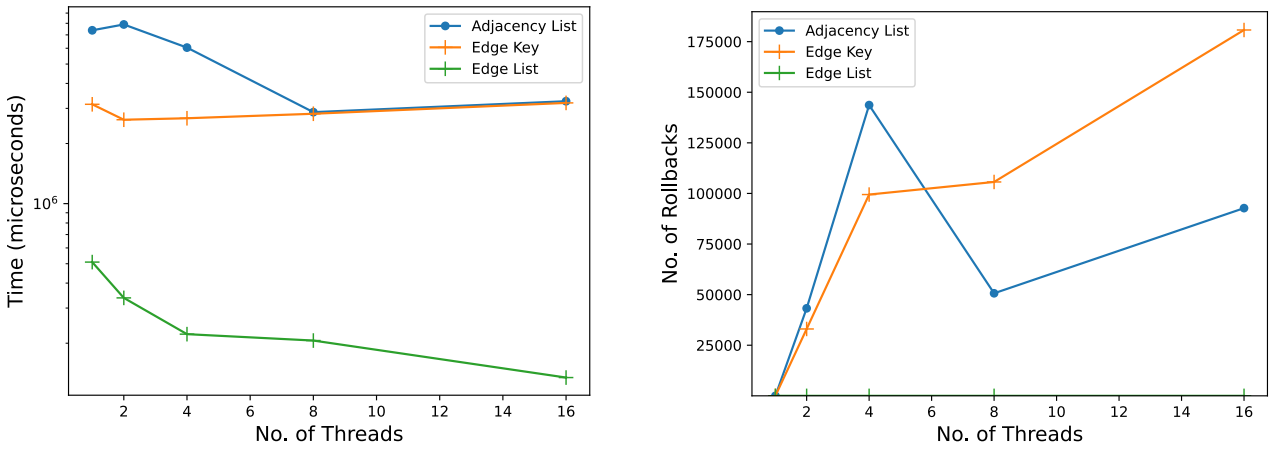


Figure 1: Comparison of disk space consumption by the 3 different representations.

Representation	No. of Values Stored	Table Size	Index Size	Total
Adjacency List	$4\|V\| + 2\|E\|$	339MB	0MB	339MB
Edge Key	$3\|V\| + 2\|E\|$	656MB	1138MB	1835MB
Edge Log	$3\|E\|$	512MB	427MB	942MB

Table 1: Breakdown of storage components in database.



(a) Runtime comparison **(b) Rollback comparison**

Figure 2: (a): Edge Key and Adjacency List fail to scale to increasing number of threads, and (b) this is caused by increasing number of rollbacks

without traversing the secondary indices and also on *Edge Log*.

WCC maintains no notion of a "node frontier". Every node can propagate values to its neighbours at any time. However, BFS requires that nodes must wait until they are on the "frontier" to propagate values, otherwise the BFS invariant will be broken. Giving up fast access to node neighbourhoods means that it requires an iteration over the entire table in $O(|E|)$ to advance the frontier by one step.

Thus, We expect WCC to perform identically across the 4 iteration scenarios, but VC-style BFS to outperform EC-style BFS. Algorithmically, VC-style BFS looks at each edge at most once and takes $O(|E|)$. EC-style BFS takes $O(d|E|)$ where d is the diameter of the graph.

We run all benchmarks on a generated RMAT graph with 2^{28} edges and 2^{25} nodes.

4.3.1 Weakly Connected Components. We perform WCC analysis over the entire graph 11 times, discard the first run, and present the runtime as boxplots in Figure 3.

We note that *Edge Key* with VC iteration has a slow down of $7 - 9\times$ over other representations. *Edge Key* is required to traverse both the `src` and `dst` secondary indices. Although using the `src` index still guarantees sequential access to the table, as the table conforms to (`src`, `dst`) ordering, iterating over `dst` creates random access.

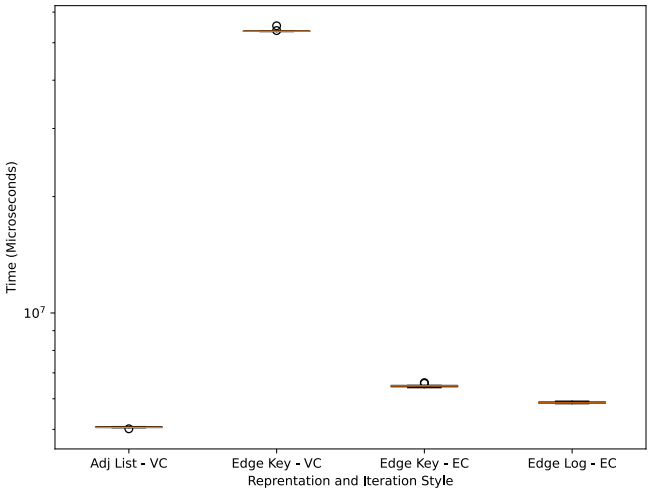


Figure 3: Weakly Connected Components

The runtime of three other scenarios are in close vicinity of each other. *Adjacency List*'s compact representation of the neighborhood array brings in less information from storage. The result indicates even with the added overhead of unpacking the array, *Adjacency List* outperforms other two representations.

4.3.2 Breadth-first Search. For BFS, we randomly sample 11 nodes from the largest connected component in the graph, and use them as starting points for our search. We discard the first run and present the runtime as boxplots in Figure 4.

Contrary to our assumptions, the median runtimes of different representations differed by at most 3×, and EC-style iteration performed closer to *Adjacency List* than expected.

One of the prominent reasons is the small diameter of the connected component we sampled from, which is only ∼ 10 despite having many nodes. This implies that EC-style BFS requires only ∼ 10 more iterations than VC-style BFS on this particular graph. However, it is noteworthy that this diameter may vary significantly in other types of graphs such as road networks.

Moreover, EC enables us to distribute the workload more evenly during parallelization, mitigating the effect of stragglers. Unlike GapBS, our EC-oriented algorithms require fewer parallel data structures, leading to less synchronization overhead.

The outliers in EC-style BFS are possibly attributed to hub nodes in the graph being the starting point. This causes the frontier to expand quickly, resulting in BFS terminating in only a few iterations.

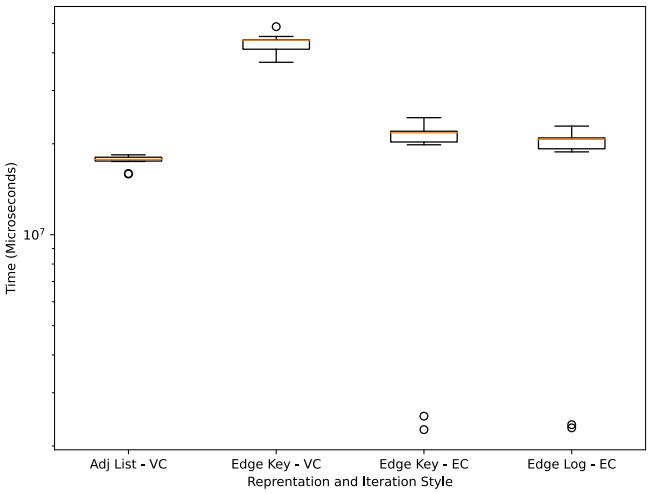


Figure 4: Breadth-first Search

5 DISCUSSION

The evaluation shows *Edge Log* to be a balanced representation compared to the other ones. In terms of storage, *Edge Log* sits between the 2 other representations. Additionally, we have not traded away a lot of performance on analytical workloads for efficient and scalable transactional operations.

However, the evaluation is far from comprehensive, and we hypothesize that only looking at BFS, *Edge Log* would perform orders of magnitudes worse than *Adjacency List* on graphs such as road networks.

In future work, our aim is to evaluate all graph algorithms available in GapBS on *FlexoGraph*. Our evaluation will include graphs with diverse characteristics, and we plan to design more equitable metrics that minimize the impact of outliers.

6 RELATED WORK

Commercial graph databases have evolved from the family of NoSQL databases, and provide require robust ACID guarantees in production. Most graph databases are not benchmarked and evaluated for their analytics performance [8].

Single-machine, multi-core graph processing systems, and their many different computation paradigms are driven by designing new graph data structures and layouts that maximize locality and IO Bandwidth utilization.

Wolfgraph[10] leverages the fact that edge-centric iteration is highly parallelizable and amenable to GPU usage. Mermaid [9] suggests degree-based heuristics to switch between edge and vertex-centric computation styles.

7 CONCLUSION

This projects analyzes in-database graph layouts from the perspectives of both transactional operations and analytical queries. Our benchmarks find that maintaining vertex ordering and storing vertex-level information in the layout impedes scalability and write throughput. To optimize write performance, we made a trade-off by sacrificing efficient access to neighborhood information in *Edge Log*.

We demonstrate that *Edge Log* performs comparably to previous graph representations when running algorithms, indicating that we can further explore this trade-off. In future work, we plan to implement optimizations on top of *Edge Log* and leverage the flexibility of edge-centric iteration by experimenting with specific edge orderings such as the Hilbert Ordering.

ACKNOWLEDGMENTS

Huge thanks to Puneet for discussing project ideas with me and pointing me in the correct direction for deeper analysis. Puneet was the main maintainer of the *FlexoGraph* codebase and the other representations. He kindly helped in debug

errors in my code. Thanks to Professor Margo Seltzer for helping review the initial drafts, and giving me the idea of the project.

REFERENCES

[1] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[2] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1):107–117, 1998. Proceedings of the Seventh International World Wide Web Conference.

[3] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. volume 6, 04 2004.

[4] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. Scalable simd-efficient graph processing on gpus. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, PACT '15, pages 39–50, 2015.

[5] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 31–46, USA, 2012. USENIX Association.

[6] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 472–488, New York, NY, USA, 2013. Association for Computing Machinery.

[7] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *The VLDB Journal*, 29(2-3):595–618, jun 2019.

[8] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei R. Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. The future is big graphs: A community view on graph processing systems. *Commun. ACM*, 64(9):62–71, aug 2021.

[9] Jinhong Zhou, Chongchong Xu, Xianglan Chen, Chao Wang, and Xuehai Zhou. Mermaid: Integrating vertex-centric with edge-centric for real-world graph processing. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 780–783, 2017.

[10] Huanzhou Zhu, Ligang He, Matthew Leeke, and Rui Mao. Wolfgraph: The edge-centric graph processing on gpu. *Future Generation Computer Systems*, 111:552–569, 2020.

[11] Zhiqiang Zuo, Yiyu Zhang, Qiuhong Pan, Shenming Lu, Yue Li, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. Chianina: An evolving graph system for flow- and context-sensitive analyses of million lines of c code. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 914–929, New York, NY, USA, 2021. Association for Computing Machinery.

8 APPENDIX

A GRAPH LAYOUTS

Nodes

Key	Value
node_id	attr1, attr2, …

Edges

Key	Value
src_id,dst_id	edge_weight, attrs...

Adjacency List

Key	Value
node_id	dst_id1, dst_id2, ...

(a) Adjacency List Representation

Edges

Key	Value
src_id, dst_id	edge_weight, attrs
src_id, -1	node_attrs

(b) Edge Key Representation

Edges

Key	Value
edge_id,src_id, dst_id	0

(c) Edge Log Representation

Figure 5: The different graph representations